UNIVERSITÀ DEGLI STUDI DI MODENA
E REGGIO EMILIA
Facoltà di Ingegneria – Sede di Modena

Corso di Laurea in Ingegneria Informatica

Interoperabilità di componenti software fra sistemi operativi eterogenei attraverso il protocollo SOAP. Confronto e sperimentazione.

Relatore

Chiar.mo Prof.

Sonia Bergamaschi

Correlatore

Ing. Maurizio Vincini

Tesi di Laurea di

Bernardini Sandro

Anno Accademico 2001-2002

Parole chiave:

- Interoperabilità software
- Protocollo SOAP
- Servizi Web e linguaggio WSDL
- Apache SOAP 2.2/ Axis, Microsoft SOAP Toolkit
- Momis

Ringraziamenti:

Desidero ringraziare la mia ragazza per il sostegno dato e l'enorme pazienza dimostrata in questi quattro mesi.

Inoltre desidero ringraziare per l'aiuto fornito durante lo svolgimento della tesi e per la disponibilità dimostrata l'Ing. Maurizio Vincini, l'Ing. Francesco Guerra, l'Ing. Ilario Benetti, l'Ing. Gionata Gelati e la Professoressa Sonia Bergamaschi.

Indice

Introduzione ed obiettivi della tesi *9*

Introduzione ed obiettivi tesi **9**

Capitolo 1- Introduzione generale *13*

Introduzione generale **13**

Capitolo 2- Introduzione a SOAP *15*

Introduzione a SOAP **15**
Descrizione 16
Obiettivi 16

Valutazioni su SOAP **18**
Scalabilità 18
Prestazioni 18
Attivazione remota 18
Controllo dello stato 18
Garbage Collection (raccolta cestino) 18
Sicurezza 19
Servizi Web 19

Capitolo 3- XML e SOAP *21*

Introduzione **21**

XML e HTML **21**

Struttura dei documenti XML **22**
I documenti XML devono essere ben formati 23
I documenti XML devono essere validi 23
XML Schema 24
XML Namespaces 27

Parser XML- DOM e SAX **29**
DOM 30
SAX 30

Capitolo 4- Specifica SOAP 1.2 ..... *32*

Struttura di un messaggio SOAP (il payload XML) **32**
Envelope 34
L'attributo encodingStyle 34
Attributi aggiuntivi 35
Elementi addizionali 35
Header 36
Azioni di un nodo SOAP e l'attributo actor 37
L'attributo mustUnderstand 37
L'attributo Root 37
Body 38

Situazioni d'errore **39**

RPC (Remote Procedure Call) **41**
RPC e SOAP Body 41
Call 41
Response 42
RPC e SOAP Header 43
RPC Faults 43

Serializzazione dei dati **44**
Tipologie di passaggio dei parametri 44
Serializzazione del passaggio per valore 44
Serializzazione del passaggio per riferimento singolo 45
Serializzazione del passaggio per riferimento multiplo 46
Serializzazione dei dati in XML 47
Tipi di dato semplici 48
Tipi di dato composti 49

Principali cambiamenti fra SOAP 1.1 e SOAP 1.2 ..... **52**
Comunicazione fra nodi SOAP 1.1 e nodi SOAP 1.2 53

Capitolo 5- HTTP e SOAP ***55***

HTTP **55**
La sicurezza 55
La comunicazione 55
Il metodo GET 56
Il metodo POST 56

HTTP e SOAP **58**

Capitolo 6 - Specifica WSDL 1.1 ..... ***60***

Introduzione **60**

Definizione di un servizio di rete **61**
Struttura di un documento WSDL 61
Types 62
Messages 63
Port Types 65
Operazioni One-way 66
Operazioni Request-response 66
Operazioni Solicit-response 66
Operazioni Notification 67
Nome degli elementi all'interno di Operation 67
Ordine dei parametri all'interno di Operation 67
Bindings 68
Ports 69
Services 69

Binding per SOAP **70**
Esempi SOAP 70
Grammatica del Binding per SOAP 73
soap:binding 75
soap:operation 75
soap:body 76
soap:fault 77
soap:header e soap:headerfault 77
soap:address 78

Capitolo 7- L'ambiente Apache SOAP 2.2 ..... *79*

L'ambiente Apache SOAP 2.2 ..... **79**
Introduzione 79
Cliente RPC 79
Server RPC 80

Esempio HelloWorld2 ..... **81**
Il server HelloWorld2 81
Il Client HelloWorld2 82
Messaggi SOAP di HelloWorld2 84
Fault di HelloWorld2 86

Esempio Adder-COM (web server Tomcat e client Apache SOAP) **87**
Server Adder-COM 87
Client Adder-COM 88
Messaggi SOAP di Adder-COM 91

Esempio Exchange **93**
Server Exchange 93
Client Exchange 94
Messaggi SOAP 95

Capitolo 8- L'ambiente Microsoft SOAP Toolkit 2.0 ..... *97*

Caratteristiche generali **97**

Lato Client **98**
Flusso dati sul lato Client 98
Settaggio dell'oggetto SoapClient 98
Elaborazione interna all'oggetto SoapClient 99

Lato Server **100**
Flusso dati sul lato server 100
Elaborazione interna all'oggetto SoapServer 100

Esempio Adder-COM (web server ISS e client Apache SOAP) **101**
Server Adder-COM 102
Cliet Adder-COM 107
Messaggi SOAP di Adder-COM 109

Esempio Add (web server Tomcat e client VB) **111**
Server Add 111
Client Add 111
Messaggi SOAP di Add 115

Esempio Indirizzo (Web server ISS e client Visual Basic) **116**
Server Indirizzo 121
Client Indirizzo 124
Messaggi SOAP di Indirizzo 125

Capitolo 9- L'ambiente Apache Axis *127*

Caratteristiche generali **127**

Esempio Add (web server Tomcat e client Apache Axis) **128**
Server Add 128
Client Add 130
Messaggi SOAP di Add 133

Supporto WSDL con Axis **134**

Tool Java2WSDL **134**

Tool WSDL2Java **139**
Lato client 142
Types 142
Holders 146
PortTypes 146
Bindings 147
Services 153
Lato server 157
Binding 158
Services 161

Esempio Indirizzo (web server Tomcat e client Apache Axis) **163**
Messaggi SOAP di Indirizzo 164

Esempio Indirizzo (web server Tomcat e client Visual Basic) **165**
Server Indirizzo 166
Client Indirizzo 166
Messaggi SOAP di Indirizzo 172

Esempio Rubrica (web server Tomcat e client Apache Axis) **174**
Server Rubrica 175
Serializzatore per Indirizzo 176
Serializzatore per Rubrica 179
Deserializzatore per Indirizzo 182
Deserializzatore per Rubrica 186
Pubblicazione del servizio 188
Client Rubrica 190
Messaggi SOAP di Rubrica 193

Capitolo 10-Integrazione del sistema MOMIS con il protocollo SOAP ***197***

Il sistema Momis **197**
L'architettura di MOMIS 197
Il processo d'integrazione delle informazioni 198
Il Global Schema Builder 200

Il sistema MOMIS ed il protocollo SOAP **200**

Il servizio ClassiLocali (client Apache Axis) **201**
Server ClassiLocali 202
Client ClassiLocali 208
Messaggi SOAP di ClassiLocali 210

Il servizio ClassiLocali (client Visual Basic/ASP) **212**
Il supporto WSDL di Apachi Axis per il servizio ClassiLocali 212
Client Visual Basic/ASP per ClassiLocali 214
Client Visual Basic di ClassiLocali 217
Client ASP di ClassiLocali 218

Conclusioni **219**

Conclusioni generali ***220***

Bibliografia ***222***

Indice delle figure

Figura 1:Schema di comunicazione SOAP (utilizzando HTTP) 16

Figura 2:Struttura di un messaggio SOAP 32

Figura 3: Esempio di messaggio SOAP 33

Figura 4:Messaggio SOAP con l'attributo Root attivo 38

Figura 5:Messaggio SOAP d'errore 40

Figura 6: Messaggio SOAP d'errore per header block obbligatorio non capito 40

Figura 7: Messaggio SOAP per richiesta RPC 42

Figura 8: Messaggio SOAP di risposta alla chiamata RPC precedente 43

Figura 9: Passaggio di parametri per valore. 45

Figura 10:Passaggio di parametri per riferimento singolo. . 46

Figura 11:Passaggio di parametri per riferimento multiplo. 47

Figura 12:Messaggio d'errore VersionMismatch con estensione Upgrade multipla 54

Figura 13:Richiesta RPC attraverso un messaggio HTTP.. 58

Figura 14:Risposta RPC corretta attraverso un messaggio HTTP 59

Figura 15:Messaggio HTTP contenente un errore..... 59

Figura 16:Funzionamento dell'utility tcTrace..... 84

Figura 17:Flusso dati sul lato client..... 98

Figura 18: Elaborazione interna dell'oggetto SoapClient.... 99

Figura 19:Flusso dei dati sul lato server 100

Figura 20:Elaborazione interna dell'oggetto SoapServer . 101

Figura 21:L'architettura MOMIS 198

Figura 22:Le Fasi dell'integrazione delle informazioni in MOMIS 199

Figura 23: Architettura del Global Schema Builder..... 200

Introduzione ed obiettivi della tesi

Introduzione ed obiettivi tesi

L'obiettivo degli studi eseguiti nell'ambito di questa tesi è triplice:

- Studio del protocollo SOAP e del linguaggio WSDL secondo le specifiche del W3C (Word Wide Web Consortium), ossia del consorzio che si occupa dello sviluppo e della standarizzazione di tecnologie nell'ambito del Web che assicurino l'interoperabilità fra sistemi eterogenei.

  Il protocollo SOAP è basato su XML e nasce con lo scopo di migliorare il trasporto dei dati fra sistemi distribuiti eterogenei e decentralizzati. Attraverso il protocollo SOAP (Simple Object Access Protocol) è possibile far interagire/comunicare in modo trasparente applicazioni scritte in linguaggi di programmazioni diversi o addirittura controllati da sistemi operativi diversi. È possibile aspettarsi, ad esempio, che un'applicazione client scritta in Visual Basic (in azione su un'architettura Windows 2000), utilizzi un servizio offerto da un'applicazione scritta in Java (ed in azione su un'architettura Unix).
  Attualmente per il protocollo SOAP è disponibile la specifica 1.2 del W3C, rilasciata nel Dicembre 2001.
  Il linguaggio WSDL (Web Services Description Language) definisce, in modo strutturato, le specifiche d'interfaccia di un singolo Web Service (localizzazione del servizio, metodi remoti disponibili, formato/tipologia dei parametri di richiesta, formato/tipologia dei parametri/risultati di risposta, ecc.). Il linguaggio WSDL è anch'egli basato sulla grammatica XML e si inserisce nel contesto d'interoperabilità fornendo un ulteriore supporto allo sviluppatore di applicazioni SOAP.
  Ad esempio, dopo aver sviluppato un servizio remoto basato su SOAP, è possibile fornire il corrispondente documento WSDL che ne descrive le caratteristiche essenziali. In questo modo, un applicazione client, può reperire le informazioni sul servizio automaticamente dal documento WSDL. Così facendo si facilita il compito dello sviluppatore (sul lato client), il quale non deve più settare manualmente i parametri di servizio, ed inoltre si rende standard la rappresentazione del servizio web. Attualmente per il linguaggio WSDL è disponibile la specifica 1.1 del W3C, rilasciata nel Marzo 2001.

  I primi capitoli della tesi hanno la seguente struttura:
  - Nel Capitolo 2 sono state prese in esame le caratteristiche principali del protocollo SOAP, valutandone vantaggi e svantaggi.

  - Nel Capitolo 3 sono stati richiamati i concetti principali del protocollo XML tenendo in considerazione l'utilizzo di questo all'interno di SOAP.

  - Nel Capitolo 4 è stata introdotta la specifica SOAP 1.2 (facendo riferimento ai working draft W3C rilasciati il 17 Dicembre 2001 e riportati nella bibliografia) ed i principali cambiamenti con la precedente specifica SOAP 1.1.

- Nel Capitolo 5 è stato analizzato l'utilizzo del protocollo HTTP come mezzo di trasporto per le comunicazioni SOAP.

- Nel Capitolo 6 è stata introdotta la specifica WSDL 1.1 (facendo riferimento al relativo documento W3C rilasciato il 15 Marzo 2001 e riportato nella bibliografia).

- Studio ed utilizzo dei principali toolkit che implementano il protocollo SOAP (ed eventualmente il linguaggio WSDL):
  - Apache SOAP 2.2
  - Microsoft SOAP Toolkit
  - Apache Axis

  Nel Capitolo 7 di questa tesi è stato analizzato il toolkit Apache SOAP 2.2. Questo toolkit è composto essenzialmente da un collezione di package Java che forniscono il supporto, se pur con qualche limitazione, alla specifica SOAP 1.1 (rilasciata dal W3C nell'anno 2000). Questo toolkit non prevede il supporto per il linguaggio WSDL.
  Apache SOAP 2.2 può essere installato sia come client-side, per invocare servizi SOAP che si trovino altrove, sia come server-side per pubblicare servizi SOAP utilizzabili in remoto. Sul lato server, per la pubblicazione dei servizi, il toolkit richiede la presenza di un web application server (ad esempio ApacheTomcat).
  Questo capitolo, attraverso lo sviluppo di alcuni esempi, ha permesso di realizzare i primi servizi SOAP e di sperimentare alcuni aspetti dell'interoperabilità.
  Per quanto riguarda l'interoperabilità sono stati eseguiti i seguenti esperimenti:
  - Accedere, con un client Apache SOAP 2.2, ad un servizio sviluppato attraverso Visual Basic ma pubblicato attraverso Apache SOAP 2.2 (il quale è in grado di pubblicare anche oggetti COM oltre che Java).

  - Accedere, con un client Apache SOAP 2.2, ad un servizio già presente sul web (riguardante l'andamento, in tempo reale, dei tassi di cambio fra monete) e sviluppato/pubblicato attraverso GLUE Electric (un altro toolkit SOAP).

  Nel Capitolo 8 è stato analizzato Microsoft SOAP Toolkit, il quale permette di invocare servizi o di pubblicare servizi attraverso due livelli di API (Application Program Interface): "high" (che include il supporto per il linguaggio WSDL) oppure "low". La scelta dipenderà dalle caratteristiche dei messaggi che si desidera inviare o dal livello di monitoraggio desiderato. Questo toolkit contiene la libreria Microsoft Type Library che fornisce il supporto SOAP ai principali linguaggi di programmazione di casa Microsoft (come Visual Basic e Visual C++). Inoltre fornisce un tool che aiuta l'utente a generare il file WSDL relativo ad un servizio.
  In questo capitolo sono stati eseguiti i seguenti esperimenti:
  - Ampliamento del discorso sull'interoperabilità pubblicando un servizio COM (sviluppato con Visual Basic e Microsoft SOAP Toolkit) attraverso il web application server ISS (Internet Information Server) di casa Microsoft; il rispettivo client è stato creato con Apache SOAP 2.2.
    In realtà non è stato possibile far interagire direttamente il server ed il client, ma è stato necessario definire un nuovo deserializzatore sul lato client. Questo perché il client Apache SOAP 2.2 richiede che per tutti i dati ricevuti come risposta dal server sia specificato il loro tipo (intero, stringa, ecc.) mentre il toolkit della Microsoft non invia queste informazioni.
    Il deserializzatore che creato per Apache SOAP 2.2, permette di superare questo problema.

- È stato ripreso il servizio descritto al punto precedente ma invertendo i ruoli: server (sviluppato con Apache SOAP 2.2) pubblicato su web application server Tomcat e client Visual Basic (con supporto delle librerie di Microsoft SOAP Toolkit). La difficoltà incontrata è stata analoga al caso precedente: il server Apache accetta solo richieste dove per tutti i parametri è specificato il loro tipo, mentre il client Microsoft non invia queste informazioni. In questo caso è stato creato un serializzatore sul lato client in grado di superare questo problema.

- È stato sviluppato un servizio in cui il dato di risposta non è più di tipo semplice (come interi, reali, stringhe, ecc.), ma di tipo complesso (creato dallo sviluppatore). Mentre per il tipo semplice il toolkit fornisce i serializzatori/deserializzatori standard, per il tipo complesso è stato necessario creare dei serializzatori/deserializzatori ad hoc. Per fare ciò, stato necessario prendere confidenza con il parser DOM ed approfondire la conoscenza dello standard WSML (utilizzato internamente dal toolkit della Microsoft). In questo caso sia il server che il client sono stati sviluppati con Visual Basic (con supporto di Microsoft SOAP Toolkit), ed il server è stato pubblicato su ISS.

Nel Capitolo 9 è stato preso in considerazione il toolkit Apache Axis, il quale rappresenta l'evoluzione di Apache SOAP 2.2. Da Maggio 2002 è distribuita la versione beta2 in attesa di quella definitiva. Apache Axis, oltre ad includere le caratteristiche del predecessore, ne aggiunge altre: prima fra tutte un ottimo supporto per il linguaggio WSDL.
In questo capitolo sono stati eseguiti i seguenti esperimenti:

- È stato ripreso un servizio, precedentemente pubblicato, che prevedeva un server sviluppato con Apache SOAP 2.2 e un client sviluppato con Microsoft SOAP Toolkit. In questo caso, il server è stato sviluppato con Apache Axis. In questo modo si è potuto apprezzare che il client ed server, di questi due toolkit, possono interagire direttamente senza difficoltà.

- È stato analizzato in dettaglio il supporto WSDL fornito da Apache Axis. In particolare il tool Java2WSDL che permette di ottenere automaticamente il documento WSDL che descrive il servizio, ed il tool WSDL2Java che permette di generare tutte le classi/interfacce Java necessarie per implementare/accedere al servizio descritto dal documento WSDL.

- È stato sviluppato un servizio in cui il dato di risposta è di tipo complesso (creato dallo sviluppatore). Anche in questo caso è stato necessario sviluppare, utilizzando Apache Axis, i serializzatori/deserializzatori opportuni. Per questo servizio sia il server che il client sono stati sviluppati con Apache Axis, ma è stato sviluppato anche un client Microsoft SOAP Toolkit. In questo modo è stato testato il concetto d'interoperabilità sui dati di tipo complesso.

- È stato sviluppato un servizio il cui oggetto della comunicazione è un dato di tipo complesso che a sua volta contiene un array di dati di tipo complesso. Questo ha permesso di capire come avviene la gestione di più livelli di serializzatori/deserializzatori.
  Infatti il serializzatore/deserializzatore del dato complesso a sua volta deve richiamare il serializzatore/deserializzatore dell'array complesso contenuto.

- Integrazione, di base, del sistema MOMIS con il protocollo SOAP.
  Il sistema MOMIS (Mediator EnvirOnment for Multiple Information Source) è frutto della collaborazione fra l' Università di Modena e Reggio Emilia e l'Università di Milano e Brescia nell'ambito di un progetto nazionale di ricerca. MOMIS è stato progettato per fornire un accesso integrato ad informazioni eterogenee memorizzate sia in database di tipo tradizionale (ad esempio relazionali e object oriented) o file system, sia in sorgenti di tipo semistrutturato.
  Attualmente, per la gestione dei dati, MOMIS utilizza l'architettura ad oggetti distribuiti CORBA. All'interno di CORBA è possibile sfruttare il protocollo IIOP (basato su TCP/IP) per la trasmissione dei dati, il quale presenta tre notevoli svantaggi:
  1. È un protocollo nativo per l'architettura CORBA, quindi può essere utilizzato solo all'interno di un sistema in cui tutti i nodi oggetto della comunicazione sono basati sulla stessa architettura CORBA. Questo è un grosso limite al concetto d'interoperabilità.
  2. È un protocollo binario e non testuale come SOAP (che infatti è composto da un payload XML, quindi semplice testo). Questo comporta dei problemi nell'attraversamento dei firewall, i quali trovano difficile analizzare gli scopi/le richieste del messaggio e quindi sono propensi a rifiutarlo. Mentre adottando SOAP, in congiunta con HTTP, il firewall si trova ad analizzare un messaggio direttamente leggibile (composto da header HTTP e dal payload informativo XML del messaggio SOAP); questo gli consente di raccogliere rapidamente i contenuti rilevanti delle richieste in entrata e di occuparsi del monitoraggio del traffico in modo semplice.
  3. Essendo IIOP un protocollo nativo per l'architettura CORBA richiede l'utilizzo di porte TCP/IP non standard. Anche questa caratteristica comporta delle difficoltà nell'attraversare i firewall che, tendenzialmente, sarà configurato per non permettere comunicazioni su porte TCP/IP non standard. Mentre adottando SOAP, in congiunta con HTTP, è possibile comunicare su porte TCP/IP standard (tipicamente la porta 80), quindi non si ha la necessità di cambiare la configurazione dei firewall con la conseguente diminuzione del livello di sicurezza.

  Nel Capitolo 10 è stato sviluppato un servizio SOAP che permette l'accesso ad alcune informazioni di uno schema integrato MOMIS, utilizzando HTTP come protocollo di trasporto sulla porta 80.
  In particolare il client fornisce il nome di un file, dove si trova memorizzato un oggetto CORBA Global Schema, e il nome di una classe globale contenuta al suo interno. Il server recupera questo oggetto, ricava il nome di tutte le classi locali appartenenti alla classe globale indicata ed invia queste informazioni al client. Il client è stato sviluppato in due linguaggi di programmazione: Apache Axis e Visual Basic (estendibile anche ad ASP, per accedere direttamente al servizio da un browser Internet).
  Il server è stato sviluppato con Apache Axis, ma al suo interno utilizza le stesse componenti CORBA utilizzate normalmente dal sistema MOMIS. In altre parole, per accedere alle informazioni vere e proprie non sono state necessarie delle modifiche al sistema MOMIS, cioè il sistema MOMIS è stato integrato con SOAP per permettere l'acceso remoto alle informazioni/ai servizi con questo protocollo.
  Quindi senza la necessità di rivedere un sistema già sviluppato, è stato possibile estendere le sue potenzialità verso una maggiore interoperabilità fra linguaggi e sistemi oprativi differenti.

Capitolo 1- Introduzione generale

Introduzione generale

Nel giro di pochi anni il mondo dell'Information Tecnology è profondamente cambiato e maturato.
Internet in generale, ed il Web in particolare, hanno rivoluzionato il modo di utilizzare il personal computer, e di conseguenza anche il modo di sviluppare soluzioni architetturali rivolte ai consumatori di servizi informativi.
L'apice tecnologico di questa evoluzione è stata, in tutto il mondo, l'esplosione di soluzioni rivolte alla New Economy, grazie alle quali è stata mostrata in tutto il suo splendore la potenza economica, oltre che mediatica, del fenomeno.
Le grosse sfide che , nel presente e nel prossimo futuro, i produttori di soluzioni Internet dovranno affrontare sono fondamentalmente due: soddisfare i bisogni sempre più impegnativi degli utenti ed integrare sistemi informativi tra loro eterogenei e sempre più complessi.
In questo quadro appare chiaro che soltanto chi riuscirà a far comunicare tra loro sistemi eterogenei in modalità rapida, controllata e sicura, sarà in grado di fornire soluzioni architetturali che soddisfino i bisogni degli utilizzatori. [V]
La parola d'ordine è quindi ***integrazione***.
L'integrazione è una problematica che ha al suo interno molte insidie, spesso i componenti da integrare sono di tipo diverso, con caratteristiche morfologiche differenti, con gestione delle problematiche analizzate in modo ortogonale, quando non addirittura incompatibili.
Riuscendo a trovare un formalismo d comunicazione che renda chiare le regole per scambiare pacchetti di dati o applicativi da una qualsiasi piattaforma elaborativa ad un'altra, si è in grado di far comunicare fra loro sistemi diversi, a condizione che tutti gli attori della comunicazione rispettino le regole imposte dal formalismo stesso.
La soluzione può essere ottenuto tramite un semplice e leggero protocollo applicativo: SOAP.
SOAP nasce nel 1999 dalla collaborazione di importanti aziende come Microsoft DevelopMentor ed IBM ed è oggetto di standardizzazione ad opera del W3C.
L'essere protocollo standard permette a SOAP di non rimanere vittima di conflitti aziendali o di essere controllato da un azienda monopolistica. Infatti attualmente, pur esistendo diverse aziende che propongono differenti implementazioni SOAP, queste devono necessariamente sottostare allo standard del W3C così da poter essere il più possibile interoperative.
SOAP (Simple Object Access Protocol) rappresenta il mezzo per mettere in contatto i fornitori di servizi distribuiti con i rispettivi utilizzatori, tralasciando problematiche legate alle comunicazioni fisiche, ai sistemi operativi, ai protocolli di comunicazione.
Allo stesso modo SOAP, si combina molto bene alle continue evoluzioni strategiche che frequentemente riguardano le aziende: alleanze strategiche, integrazioni con altre aziende, assorbimenti di altre aziende, cambiamenti di proprietà, ecc.
In queste situazioni non è possibile (in primo luogo per motivi economici, tecnici e formativi) abbandonare i diversi sistemi informativi, i diversi applicativi presenti nelle

azienda per adottarne altri di uso comune. La strada da percorrere è sicuramente la più ampia e trasparente integrazione possibile.
SOAP fornisce una "lingua" comune che tutti gli attori della comunicazione devono saper scrivere ed interpretare.
Il payload del messaggio (cioè l'informazione vera e propria) viene spedita utilizzando XML, quindi solo testo, interpretabile da qualsiasi sistema informativo e tra gli standard più riconosciuti ed avanzati; inoltre tra i protocolli di comunicazione si tende a privilegiare HTTP, cioè lo standard della comunicazione su Internet, un'altra garanzia.
Il client dovrà formalizzare la richiesta di un servizio al server utilizzando le regole dettate dal protocollo e si attenderà una risposta che segua le medesime regole, in modo da poterle gestire.
Il server dovrà interpretare la richiesta di servizio spedita dal client con i relativi parametri correlati, dovrà effettuare l'elaborazione necessaria ed infine spedire i risultati al client utilizzando nuovamente lo stesso formalismo.
Per fare interagire sistemi eterogenei, quindi è necessario basare la comunicazione su linguaggi e protocolli di comunicazione evoluti, aperti e supportati, come XML e HTTP.
Per quanto detto il concetto di base è piuttosto semplice, la sua potenza tuttavia è data dal fatto che dietro la stesura del protocollo SOAP non ci sono interessi aziendali particolari: si tratta infatti di una specifica trasversale al mondo economico dell'Information Tecnology.
Con queste premesse, tecniche prima ancora che di business, SOAP è destinato a diventare un vero e proprio standard per le comunicazioni tra componenti applicativi diversi, la linfa vitale, quindi di applicazioni B2B e B2C che della comunicazione non possono certo fare a meno.

Capitolo 2- Introduzione a SOAP

Introduzione a SOAP

Una delle principali intuizioni di SOAP è quella di fornire un protocollo applicativo completo evitando di inventare nuove tecnologie.
SOAP infatti, utilizza due architetture molto robuste e ben collaudate, si tratta in particolare di HTTP per il trasporto dati e di XML per la loro serializzazione.
Sono sufficienti poche righe di codice, scritto utilizzando un qualsiasi linguaggio di programmazione, che utilizzino HTTP per trasportare i dati ed XML per immagazzinarli, per costruire un'architettura RPC (Remote Procedure Call) funzionante.
Quello che manca, in questo contesto, è solo la standardizzazione, una regola precisa per la trasformazione delle informazioni in XML (processo di serializzazione) e la regola inversa (processo di deserializzazione) per la trasformazione dell'informazione XML serializzata in qualcosa che l'applicazione sia in grado di utilizzare in maniera nativa (come un dato, una struttura, un oggetto, ecc.).
Questa standardizzazione è fornita da SOAP.
Un metodo SOAP non è altro che l'unione di una request ed una response HTTP contenenti il payload informativo codificato in XML.
Un endpoint SOAP è un URL che identifica l'oggetto remoto da chiamare.
Una richiesta SOAP è una request HTTP arricchita dal payload XML e da eventuali dichiarazioni supplementari di sicurezza.
Una risposta SOAP è una response HTTP contenete il payload XML di ritorno.
Il payload SOAP non è altro che un documento XML presente nella richiesta SOAP e nella relativa risposta. Il payload contiene, oltre all'endpoint, anche tutti i parametri da passare all'oggetto remoto, codificati utilizzando le regole di serializzazione.
Il formato della risposta SOAP deve essere analogo a quello della richiesta, le differenze sostanziali sono le seguenti:

- La richiesta trasporta parametri di tipo [in] e [in,out].
- La risposta trasporta parametri di tipo[in,out] e [out], ed un eventuale risultato di ritorno del metodo invocato.
- Il payload della risposta deve contenere un elemento con lo stesso nome del metodo invocato, ma concatenato alla stringa "Response".

Come si può vedere SOAP rappresenta una regolamentazione delle possibili modalità di serializzazione e del rispettivo trasporto dei dati.
Va sottolineato che lo standard SOAP non è vincolato ad un particolare protocollo di trasporto dati.
La Figura 1 illustra la filosofia di base: un applicazione client che necessita di utilizzare servizi remoti, non deve far altro che utilizzare una strato applicativo intermedio che si occupi di serializzare le informazioni utilizzando XML ed instradarle attraverso un protocollo di trasporto (ad esempio HTTP) verso l'applicazione server.
Questa provvederà a spacchettare l'informazione racchiusa nel payload XML, elaborarla e restituire il risultato dell'elaborazione all'applicazione chiamante, sempre utilizzando lo standard SOAP. Infine, anche il client, provvederà a spacchettare l'informazione racchiusa nel payload XML inviatogli dal server, allo scopo di elaborare i risultati prodotti dai servizi remoti.

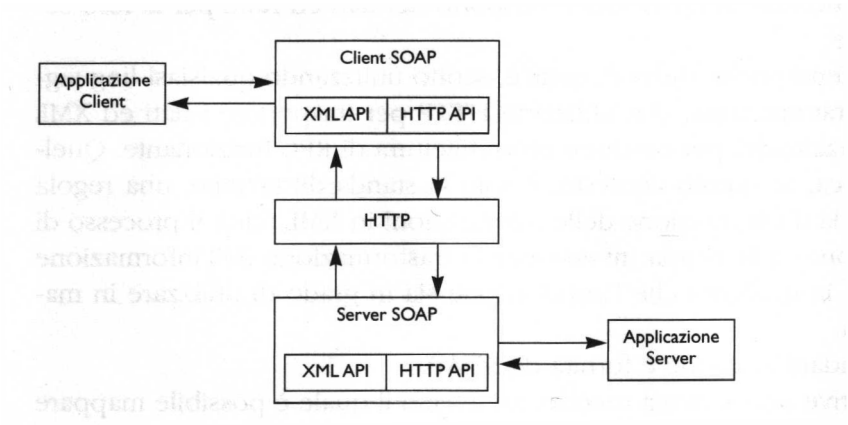


Figura 1:Schema di comunicazione SOAP (utilizzando HTTP)

Descrizione

SOAP è un protocollo applicativo, consiste cioè in una serie di regole che permettono ad applicazioni distribuite di scambiarsi informazioni.
Il pacchetto di informazioni scambiate prende il nome di payload, ed è una struttura dati, anche complessa, descritta utilizzando il linguaggio di marcatura XML.
Il protocollo di trasporto non è predefinito, si può adottare un qualunque protocollo in grado di trasportare informazioni testuali (infatti XML non è altro che testo).
In realtà si tende a privilegiare HTTP per la sua ampia distribuzione e perché attraverso HTTP è possibile utilizzare facilmente i meccanismi di richiesta/risposta tipici di SOAP.

Obiettivi

SOAP (Simple Object Access Protocol) nasce nel dicembre 1999, da un gruppo di sviluppatori appartenenti a diversi consorzi (fra cui Microsoft, DevelopMentor ed IBM), come protocollo (basato su XML) con lo scopo di migliorare il trasporto dei dati fra sistemi distribuiti eterogenei decentralizzati.
Il protocollo SOAP è stato oggetto di standardizzazione dal consorzio W3C (Word Wide Web Consortium), il quale ha di recente rilasciato la specifica 1.2.

In passato la maggior parte delle soluzioni individuate tendevano ad essere specifiche della piattaforma usata, non scalavano molto bene (ostili ad evoluzioni future), spesso richiedendo molti round-trip (viaggi andata e ritorno) fra client e server. In parte questi problemi sono stati risolti dai protocolli CORBA, RMI e DCOM, i quali però incontravano difficoltà nell'attraversare i Firewall.

SOAP nasce con il preciso obbiettivo di creare un meccanismo di RPC in ambiente distribuito, senza però soffrire delle limitazioni tipiche di altre soluzioni, più complesse e per questo più difficili da generalizzare, come appunto CORBA, RMI e DCOM.
Tale meccanismo deve essere utilizzabile attraverso qualunque sistema operativo o linguaggio di programmazione in maniera trasparente.
È possibile aspettarsi, ad esempio, che un'applicazione client scritta in Visual Basic in azione su un'architettura Windows 2000, utilizzi un servizio offerto da un'applicazione server scritta in Java ed in azione su un'architettura Unix.
Questo è possibile grazie all'utilizzo di XML e HTTP.

> ***Perché XML ?***
> - *Text-based (human authorable and readable)*
> - *Semplice da codificare (poche regole)*

- *Adottato nel Word Wide Web*
- *Estendibile con i namespace*

L'avere un payload scritto in XML, inoltre, garantisce che eventuali evoluzioni del protocollo possano essere assorbite senza grandi effetti collaterali sulle applicazioni preesistenti e questa è un'ottima garanzia di qualità dal punto di vista della manutenzione di tutto il codice già scritto.

I messaggi SOAP devono poter viaggiare sulle architetture distribuite senza limitazioni, e questo è possibile grazie al protocollo HTTP, che per sua natura permetta la distribuzione di messaggi testuali.

Attualmente SOAP utilizza HTTP come protocollo di trasporto dei messaggi diventando indipendente dalla piattaforma e ponendosi ad un livello indipendente dal sottostante protocollo di trasporto. Teoricamente è possibile effettuare le richieste utilizzando un qualsiasi protocollo di trasporto ed avere le risposte con qualsiasi protocollo di trasporto.

Perché HTTP ?

- *Protocollo Web più usato*
- *Disponibile su tutte le piattaforme*
- *Non necessita di runtime particolari*
- *E' connection-less (nessun pacchetto per iniziare/mantenere la sessione)*
- *La sicurezza è semplice da implementare e molto efficiente*
- *L'unico vero protocollo firewall-friendly*
- *L'header HTTP è estendibile (può contenere qualunque informazione)*
- *Adatto al modello request/response tipico di RPC*

L'utilizzo degli header HTTP, inoltre, facilita la gestione della sicurezza e permette un controllo accurato di ciò che viaggia sulla rete. I firewall, infatti, non hanno alcuna difficoltà nel verificare la provenienza e i servizi richiesti dai messaggi SOAP, proprio per la presenza di header HTTP personalizzati.

SOAP non dice nulla sulla semantica delle specifiche applicazioni che si scambiano i messaggi, così come sul routing dei messaggi , l'affidabilità dei dati trasferiti, l'attraversamento dei firewall, etc. Questo permette a SOAP di essere estremamente semplice, mantenendo la possibilità di interfacciarsi con diverse piattaforme.

Comunque, SOAP fornisce una struttura con la quale le specifiche informazioni dell'applicazione possono essere trasportate in maniera estendibile. Inoltre, SOAP fornisce una completa descrizione delle azioni che deve intraprendere un processore SOAP al momento della ricezione del messaggio.

L'essere protocollo applicativo vuol dire anche astrarsi dall'architettura di distribuzione degli oggetti, e di conseguenza SOAP non tratta al suo interno problematiche tipiche dei sistemi distribuiti, quali ad esempio:

- Distribuited Garbage Collection
- Boxcarring
- Batching of messages
- Object by references
- Object activation

I messaggi SOAP sono fondamentalmente delle trasmissioni a senso unico fra un nodo SOAP mittente e un nodo SOAP ricevente, ma i messaggi SOAP sono pronti a combinarsi con le applicazioni che implementano modelli di comunicazione più complessi (ad esempio richiesta/risposta multipla).

Valutazioni su SOAP

Al contrario di altre architettura di oggetti distribuiti, come DCOM o CORBA, SOAP è soltanto un semplice protocollo di distribuzione delle informazioni.
Volendo fare un confronto di massima possiamo dire che SOAP ha alcuni vantaggi, ed altrettanti svantaggi, che lo rendono in ogni caso interessante e su cui è opportuno effettuare alcune valutazioni di carattere tecnico nel momento in cui sia necessario scegliere le architetture per la distribuzione degli oggetti in ambienti professionali.

Scalabilità

Certamente SOAP è il protocollo più facilmente scalabile, infatti utilizzando HTTP come protocollo di trasporto, che ha come caratteristiche principali la distribuzione, il modello request/response e l'assenza di gestione dello stato, si ottiene un protocollo applicativo che può crescere in funzione delle nuove opportunità offerte dal protocollo di trasporto stesso e dal modello di serializzazione per il payload.

Prestazioni

SOAP non è un protocollo ottimizzato a livello prestazionale.
Il modello di SOAP, infatti, richiede che i dati da trasferire siano serializzati in XML utilizzando opportuni tools; questo ne rallenta le prestazioni durante l'invio delle informazioni verso il server.
Inoltre il fornitore di servizi sarà tenuto a deserializzare il payload per poter utilizzare le informazioni di richiesta, rallentando il processo di esecuzione del servizio remoto, in quanto sarà necessario utilizzare un parser XML per estrarre i dati dal payload.
Al momento dell'invio della risposta del server verso il client abbiamo lo stesso rallentamento delle prestazioni. Infatti il server avrà la necessità di serializzare le informazioni in XML, mentre il client avrà la necessità di deserializzare il payload XML per accedere alle stesse informazioni.

Attivazione remota

SOAP non gestisce in maniera nativa l'attivazione di metodi remoti.
Questa è stata una scelta precisa, non una mancanza o un'omissione nella stesura della specifica del protocollo. SOAP infatti, essendo un protocollo applicativo, delega completamente la gestione dell'attivazione remota all'architettura di distribuzione che lo implementa.

Controllo dello stato

Quando SOAP si appoggia ad HTTP per il trasporto di messaggi tra client e server, è chiaramente un protocollo applicativo "stateless".
Se, infatti, il protocollo di trasporto non è in grado di mantenere informazioni sullo stato tra una connessione e l'altra, non è tecnicamente possibile che ci riesca SOAP, trovandosi ad un livello comunicativo superiore.
La gestione dello stato si può comunque emulare utilizzando metodi simili a quelli che si usano nello sviluppo di soluzioni Internet complesse: i cookies, il trasporto delle sessioni, il trasporto di parametri nascosti, ecc.

Garbage Collection (raccolta cestino)

SOAP non indirizza direttamente gli oggetti applicativi, ma si limita a fornire una specifica per il loro trasporto e la loro distribuzione.

Il tempo di vita di un oggetto trasportato da SOAP, pertanto, è limitato dalla request e dalla response.
La garbage collection è un'altra caratteristica che è totalmente a carico dell'architettura di distribuzione e non può essere gestita da SOAP.

Sicurezza

Anche il meccanismo di sicurezza non è gestito direttamente da SOAP.
Quando SOAP utilizza un qualsiasi protocollo di trasporto, ne usa anche le caratteristiche di sicurezza delegandone a questo la gestione.
Nel caso si utilizzi, come protocollo di trasporto, HTTP la sicurezza che si ottiene non è di alto livello.
Se si ha la necessità di utilizzare una sicurezza più elevata è sufficiente utilizzare SOAP insieme ad HTTPS, in questo modo è possibile avere un protocollo di trasporto sicuro che utilizza SSL3 e di conseguenza la chiamata SOAP acquisisce le caratteristiche di sicurezza proprie di HTTPS.
È necessario considerare che le chiamate di metodi remoti, a volte, hanno la necessità di transitare attraverso firewall posti a protezione di reti intranet.
La penetrazione di un messaggio SOAP all'interno di una rete protetta da un firewall dipende, ovviamente, dal protocollo di trasporto utilizzato.
In generale si può pensare che se il protocollo di trasporto è HTTP, la porta di riferimento, la 80, spesso si trova aperta per permettere il traffico web.
Un amministratore di rete però, se vuole limitare il traffico SOAP che transita sulla propria rete, ha la possibilità di intervenire sul singolo messaggio testando gli header HTTP per capire dove il messaggio SOAP sta cercando di andare e cosa ha intenzione di fare.
In questo modo è possibile permettere o meno la comunicazione SOAP, semplicemente configurando il firewall nella maniera opportuna.

Servizi Web

Un servizio web è, sostanzialmente, un'applicazione le cui elaborazioni sono rese disponibili su una rete HTTP.
Tale applicazione, in ascolto su una certa porta (tipicamente la 80), può essere interrogata attraverso l'invio di messaggi la cui struttura sia ben definita e, di conseguenza, sia ad essa comprensibile.
A questo punto se il servizio web riceve il messaggio ed è in grado di decodificarlo, può provare ad elaborare la richiesta ed a fornire il risultato, che dovrà anch'esso spedito verso il client in un formato che sia comprensibile ed accettato.
Viceversa, in caso d'errore, il client dovrà essere in grado di ricevere un messaggio con lo stesso tipo di formato, contenente le più ampie informazioni disponibili sull'errore.
Come si può vedere le analogie con CORBA, Javabeans e DCOM sono moltissime, ma la differenza sta nel fatto che, nel caso di SOAP, la piattaforma non è proprietaria, non si parla più di Java (CORBA) o Windows (DCOM).
La piattaforma diventa lo stesso web, cioè la rete che condivide HTTP, il vero collante per il trasporto dei messaggi.
SOAP fornisce una specifica a livello applicativo sul formato dei messaggi scambiati tra il client ed il server (in questo caso i Web Services).
Esistono alcune specifiche che tentano di chiudere il cerchio, ovvero di fornire indicazioni sulla gestione completa dei servizi web di cui SOAP tenta di fornire un formato per lo scambio delle informazioni.
Si tratta di:

- WSDL (Web Services Description Language): definisce, in modo strutturato, le specifiche d'interfaccia di un singolo Web Service (localizzazione del servizio, metodi remoti disponibili, formato/tipologia dei parametri di richiesta, formato/tipologia dei parametri/risultati di risposta, ecc.).

- DISCO (Discovery of Web Services): che fornisce un meccanismo per interrogare un server e per sapere quali particolari Web Services sono disponibili sul server stesso.
- UDDI (Universal Description Discovery and Integration): una tipologia di elenco all'interno del quale un fornitore qualsiasi può descrivere, e quindi pubblicizzare, i Web Services che rende disponibili.

Capitolo 3- XML e SOAP

Introduzione

XML è un meta linguaggio che permette di creare autonomamente linguaggi personalizzati di markup.
Le specifiche del linguaggio XML sono s tate definite dal W3C (World Wide Web Consortium), esattamente come per HTML, questo ne fa uno standard aperto a cui tutti i produttori di software devono fare riferimento. Al pari di SOAP non ci sono elementi che possano far inglobare la tecnologia all'interno di soluzioni proprietarie di qualche azienda e questa è una garanzia per l'evoluzione la standardizzazione futura del linguaggio.

XML e HTML

XML è aperto, basato su testo
La nascita di HTML, come linguaggio prodotto utilizzando SGML (Standard Generalized Markup Language), aveva come obiettivo la diffusione di informazioni ipertestuali (ed al più immagini per completare le informazioni) su sistemi differenti attraverso il protocollo di trasporto HTTP.
La diffusione di nuovi browser in concorrenza fra loro, ha fatto sì che l'informazione ipertestuale dovesse portare con se, oltre all'informazione stessa, anche alcune specifiche legate alla sua rappresentazione, con l'obiettivo di rendere la fruizione delle pagine HTML sempre più multimediale ed attraente per l'utente finale.
Questa situazione ha costretto HTML a fornire, sempre più di frequente, risposte a problemi per i quali non era stato progettato. Spesso queste risposte sono arrivate attraverso soluzioni proprietarie, che ovviamente hanno creato moltissimi problemi agli sviluppatori di soluzioni ed applicazioni che dovevano essere distribuite attraverso la rete.
A volte, purtroppo, si rendeva necessario sviluppare versioni differenti delle stesse pagine HTML per fare in modo che ogni browser riuscisse ad interpretarle correttamente.
Nello specifico HTML non è adatto allo scambio di informazioni sul web per i seguenti motivi:

- Non è nato per rappresentare dati, ma per formattarli.
- Non richiedendo nessuna validazione, non è richiesta una struttura rigorosa delle informazioni.
- Non è estendibile, e pertanto non può essere adattato a seconda delle esigenze e delle evoluzioni.

L'utilizzo di XML permette di superare queste limitazioni, infatti XML nasce dall'esigenza di definire un formato comune, indipendente dalla piattaforma, che possa garantire l'interscambio di informazioni tra sistemi eterogenei senza alcun bisogno di conversioni.
Più precisamente i motivi che hanno decretato la superiorità di XML come standard per l'interscambio di informazioni sono i seguenti:

- XML è utilizzabile in modo semplice ed efficiente su Internet.
- XML è utilizzabile da qualsiasi categoria di applicazione (ad es. gestionale, database, grafica, ecc.).

- XML è estendibile e può essere adattato alle proprie esigenze, infatti rispetto ad HTML:
    - In XML è possibile definire nuovi tag ed attributi.
    - La struttura di un documento XML può essere vista in modo gerarchico utilizzando la nidificazione dei tag per descrivere la complessità dell'informazione.
    - Un documento XML può contenere una descrizione esterna della sua grammatica, in modo da essere valicato.
- Lo sviluppo di applicativi che elaborino documenti XML è relativamente semplice.
- I documenti sono leggibili ed interpretabili anche dagli utenti oltre che dagli elaboratori.
- La realizzazione di documenti XML è formale e concisa.

Per tutte queste caratteristiche è facile comprendere perché il payload di SOAP è rappresentato in XML.

Struttura dei documenti XML

Come HTML,XML è un linguaggio di markup, pertanto la sua sintassi è formata essenzialmente da tag che possono avere attributi e/o contenere ricorsivamente altri tag.
I tag devono essere a coppie, ovvero deve esserci la presenza contemporanea di tag di apertura e chiusura, all'interno dei quali è presente il corpo ossia il contenuto informativo.
La struttura tipica di un documento XML è gerarchica e ad albero.
Di seguito viene presentato un semplice documento XML che rappresenta l'ordine di arrivo di un gran premio di Formula 1.

```
<?xml version="1.0"?>
<Gran_Premio tipo_GP="Formula 1">
  <Nazione>Italia</Nazione>
  <Circuito>Imola</Circuito>
  <Anno>2002</Anno>
  <Ordine_di_arrivo>
    <Pilota>
      <Posizione>1</Posizione>
      <Nome>M.Schumacher</Nome>
      <Vettura>Ferrari</Vettura>
      <Note></Note>
    </Pilota>
    <Pilota>
      <Posizione>2</Posizione>
      <Nome>R.Barichello</Nome>
      <Vettura>Ferrari</Vettura>
      <Note>Primi_punti</Note>
    </Pilota>
    <Pilota>
        …
    </Pilota>
  </Ordine_di_arrivo>
</Gran_Premio>
```

La semplicità di XML rende immediatamente chiaro cosa vuole rappresentare l'esempio. Infatti tutta l'informazione si autodescrive attraverso i tag, che grazie ai loro nomi, spiegano il significato dell'informazione contenuta nel documento stesso..

Quello che bisogna mantenere è la sintassi per la costruzione dei tag, ma non esistono parole chiave che sia obbligatorie per mantenere la struttura delle informazioni

La prima riga `<?xml version="1.0"?>` è un'istruzione speciale (facoltativa) che identifica la tipologia del documento e la versione di XML utilizzata all'interno del documento.

Tutto il documento è contenuto all'interno dei tag `<Gran_Premio>` e `</Gran_Premio>` che rappresentano l'elemento principale detto Root Element; mentre `tipo_GP` è l'attributo di questo elemento.
Gli attributi forniscono un informazione più specifica circa un determinato elemento.
Non è sempre semplice determinare se utilizzare un sottoelemento o un attributo, in molti casi sono possibili entrambi le soluzioni. Una possibilità consiste nell'utilizzo di attributi per indicare le classificazioni degli elementi in base al tipo di problema.

L'elemento `<Ordine_di_arrivo>` è uno dei figli dell'elemento principale, e a sua volta ha altri figli come `<Pilota>`. In questo modo si costruisce una gerarchia che aiuta sia l'elaborazione che l'interpretazione del documento.

Gli elementi che non hanno corpo non necessitano obbligatoriamente del tag di chiusura infatti la dichiarazione `<Note></Note>` è del tutto equivalente alla dichiarazione `<Note/>`.

Inoltre è bene notare che XML è case sensitive (`<Pilota>` è diverso da `<pilota>`) e che, a differenza di HTML, gli spazi bianchi vengono mantenuti e non raggruppati in un unico carattere.

I documenti XML devono essere ben formati

Un documento XML per essere ben formato deve rispettare le regole sintattiche stabilite dal W3C in sede di stesura della specifica.
Le regole sintattiche sono solo tre:

1. Deve essere presente uno ed un solo elemento principale (Root Element) e tutti gli altri devono essere subordinati ad esso.
   Ad esempio <a>… <a>… viola questa regola, dato che il documento è composto da due elementi entrambi principali.
2. Tutti i tag devono essere annidati in maniera corretta.
   Ad esempio <a> viola questa regola, dato che l'elemento a termina all'interno dell'elemento b.
3. Tutti i tag devono essere correttamente chiusi.
   Ad esempio <a> viola questa regola, mancando il relativo tag di chiusura.

I documenti XML devono essere validi

Mentre la "forma" di un documento XML rappresenta la correttezza delle sue singole componenti in funzione delle regole sintattiche, la sua validità rappresenta la correttezza del documento in funzione di regole generali cui il documento stesso, nella sua interezza, deve essere conforme.
Infatti, affinché i dati scambiati in formato XML possano essere interpretati allo stesso modo da tutti i partecipanti di una comunicazione, è necessario che venga codificato un unico documento che funga da prototipo. Tale prototipo ha lo scopo di definire una volta per tutte, la struttura e la grammatica degli elementi che compongono i documenti scambiati.

Inizialmente, allo scopo di definire la validità di un documento XML, si ricorreva all'uso dei DTD (Document Type Definition): una serie di regole che permette di definire con precisione tutti gli aspetti architettali e semantici da verificare nel documento XML.
Tuttavia, avendo limitate potenzialità descrittive, la proposta DTD è stata recentemente sostituita da XML Schema (rilasciata dal W3C).
Per tornare al caso SOAP, il payload XML deve essere composto da una struttura di questo tipo:

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
 SOAP-ENV:encodingStyle
 ="http://schemas.xmlsoap.org/soap/encoding/>
<SOAP-ENV:Header>
...
</SOAP-ENV:Header>
<SOAP-ENV:Body>
...
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In questo caso, se l'elemento `<SOAP-ENV:Body>` manca per qualche motivo, oppure se viene scritto prima l'elemento `<SOAP-ENV:Header>`, allora il documento XML sarà "non valido", anche se magari sarà ben "formato".
Avremo cioè una situazione in cui il payload SOAP sarà formato da un frammento di codice XML sintatticamente corretto, ma semanticamente sbagliato.

XML Schema

Nel caso specifico di SOAP, è importante verificare che il payload XML sia formato esattamente dalle giuste componenti, nel giusto ordine e facendo attenzione che le componenti obbligatorie siano in effetti presenti.
La struttura di un payload XML relativo ad un messaggio SOAP, è la seguente:

```
<Envelope>
  <Header>  (facoltativa)
  </Header>
  <Body>
  </Body>
</Envelope>
```

Il seguente schema è deputato a verificare la struttura appena vista:

```
<xs:schema
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:tns="http://schemas.xmlsoap.org/soap/envelope/"
   targetNamespace="http://schemas.xmlsoap.org/soap/envelope
"/>

<!—Envelope, header and body -->
<xs:element name="Envelope" type="tns:Envelope" />
<xs:complexType name="Envelope">
   <xs:sequence>
     <xs:element ref="tns:Header" minOccurs="0" />
     <xs:element ref="tns:Body" minOccurs="1" />
     <xs:any namespace="##other" minOccurs="0"
       maxOccurs="unbounded" processContents="lax" />
```

```
   </xs:sequence>
   <xs:anyAttribute namespace="##other"
processContents="lax" />
</xs:complexType>

<xs:element name="Header" type="tns:Header" />
<xs:complexType name="Header">
   <xs:sequence>
     <xs:any namespace="##other" minOccurs="0"
       maxOccurs="unbounded" processContents="lax" />
   </xs:sequence>
   <xs:anyAttribute namespace="##other"
processContents="lax" />
</xs:complexType>

<xs:element name="Body" type="tns: Body" />
<xs:complexType name="Body">
   <xs:sequence>
     <xs:any namespace="##other" minOccurs="0"
       maxOccurs="unbounded" processContents="lax" />
   </xs:sequence>
   <xs:anyAttribute namespace="##other"
processContents="lax" >
     <xs:annotation>
       <xs:documentation>
       </xs:documentation>
     </xs:annotation>
   </xs:anyAttribute>
</xs:complexType>

<xs:attribute name="mustUnderstand" default="0" >
   <xs:simpleType>
   <xs:restriction base='xs:boolean' >
     <xs:pattern value='0|1' />
   </xs:restriction>
   </xs:simpleType>
</xs:attribute>

<xs:attribute name="actor" type="xs:anyURI" />

<xs: simpleType name="encodingStyle" />
   <xs:annotation>
     <xs:documentation>
     </xs:documentation>
   </xs:annotation>
<xs:list itemType="xs:anyURI" />
</xs:simpleType>

<xs:attributeGroup name="encodingStyle" >
   <xs:attribute name="encodingStyle"
type="tns:encodingStyle" />
</xs:attributeGroup>

<xs:complexType name="Fault" final="extension" >
   <xs:annotation>
     <xs:documentation>
```

```
        Fault reporting structure
      </xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="faultcode" type="xs:QName" />
      <xs:element name="faultstring" type="xs:string" />
      <xs:element name="faultactor" type="xs:anyURI"
minOccurs="0" />
      <xs:element name="detail" type="tns:detail"
minOccurs="0" />
    </xs:sequence>
</xs:complexType>

<xs:complexType name="detail">
    <xs:sequence>
      <xs:any namespace="##any" minOccurs="0"
maxOccurs="unbounded" processContents="lax" />
    </xs:sequence>
    <xs:anyAttribute namespace="##any"
processorContents="lax" />
</xs:complexType>

</xs:schema>
```

Senza entrare troppo nel dettaglio di questo documento XML, che rappresenta lo schema di riferimento per i payload SOAP, grazie al seguente frammento di codice:

```
<xs:element ref="tns:Header" minOccurs="0" />
<xs:element ref="tns:Body" minOccurs="1" />
```

appare subito chiaro come l'elemento Envelope debba contenere almeno un elemento figlio Body, mentre l'elemento figlio Header è opzionale avendo un numero minimo di occorrenze valorizzato a 0.
Da quanto detto si capisce che un documento XML contenente la sezione Envelope ma senza la sezione Body non potrà essere utilizzato come payload XML all'interno di un messaggio SOAP perché, anche se formalmente corretto, non è semanticamente valido.
La sintassi per ottenere un legame tra il documento XML ed una validazione effettuata attraverso uno schema è la seguente:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"
/>
```

L'istruzione precedente `xmlns:` indica che il file specificato (`"http://schemas.xmlsoap.org/soap/envelope"`) farà da schema per il documento XML in questione.
Dal punto di vista di SOAP appare chiaro che XML Schema viene utilizzato per verificare in ogni istante che il payload SOAP sia conforme alle specifiche che sono contenute proprio nel file di schema.
Il file di schema, come visto, viene legato al documento XML attraverso la stringa `xmlns:` che fornisce un collegamento tra un nome mnemonico, nel nostro caso `SOAP-ENV` ed il file di schema corrispondente.

I vantaggi dell'utilizzo del modello XML Schema, come strumento di validazione dei documenti XML, rispetto al modello DTD sono i seguenti:

- Il documento da convalidare e il relativo documento schema utilizzano entrambi la sintassi XML, e per così dire parlano la stessa "lingua".
- La possibilità di supportare ed includere diversi namespace.
- La presenza di un numero elevato di tipi di dato standard (es. string, integer, ecc.).
- La possibilità di definire propri tipi di dati.
- La gestione della modularità attraverso cui è possibile riutilizzare schemi o parti di essi all'interno di altri schemi.

XML Namespaces

L'istruzione appena analizzata, `xmlns:`, è un acronimo per XML Namespaces.
Il concetto di namespace è molto simile a quello utilizzato con i linguaggi di programmazione come C++ e Java ed il suo obiettivo è di evitare collisioni tra i nomi degli elementi. Per risolvere eventuali conflitti vengono utilizzati dei nomi qualificati, cioè preceduti da un prefisso che indica lo spazio dei nomi di riferimento per l'elemento in questione.
Un namespace XML, quindi, è un nome fittizio, un alias, che si è soliti assegnare al percorso remoto di un file, tipicamente un file di schema che viene legato al documento XML in questione.

La sintassi per definire un namespace XML è la seguente: `xmlns:Nome_Namespace=URI`.
L'utilizzo del prefisso permette di distinguere quali elementi appartengono ad un namespace piuttosto che ad un altro.
L'attributo `xmlns` serve come introduzione al prefisso utilizzato dal namespace, l'URI invece viene indicato come discriminante del namespace ed ha un valore informativo e non dichiarativo. Si utilizza un URI, piuttosto che un nome fittizio, perché per definizione un URI è unico, quindi si evitano ulteriori problematiche di collisione dei nomi.

L'istruzione seguente:
`xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"`
ha il duplice scopo di legare un certo XML Schema al documento che la contiene e di assegnare all'alias `SOAP-ENV` proprio il valore del percorso del file di schema.
L'obiettivo è quello di poter utilizzare, nel resto del documento XML, l' alias `SOAP-ENV` al posto del più prolisso URL completo.
Gli XML Namespaces hanno il compito di definire degli spazi dei nomi differenti per quei tag che per qualsiasi motivo, siano stati definiti con lo stesso nome.
L'estrema facilità, infatti, attraverso la quale si definiscono nuovi tag, è arrivata al punto di far sorgere il problema della possibile esistenza, all'interno dello stesso documento XML, di tag dallo stesso nome ma dal significato semantico totalmente differente.
L'esempio seguente illustra proprio questo principio, consideriamo de documenti XML:

```
<?xml version="1.0" ?>
  <Indirizzo>
    <Via>Giardini 15</Via>
    <Città>Pavullo n/F</Città>
    <CAP>41026</CAP>
    <Provincia>Modena</Provincia>
    <Paese>Italia</Paese>
  </Indirizzo>
```

e

```
<?xml version="1.0" ?>
  <Server>
    <Nome>WebServer</Nome>
    <Indirizzo>123.156.123.8</Indirizzo>
  </Server>
```

In questo caso ogni singolo documento utilizza un proprio XML, con elementi differenti, quindi l'applicazione che si occuperà di parsificare i documenti non avrà problemi nell'interpretare i documenti.
Il problema sopraggiunge nel momento in cui si vogliano integrare le informazioni dei due documenti precedenti all'interno di un singolo documento XML.
In questo caso avremmo, infatti, il seguente documento XML:

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

<Azienda>

  <Indirizzo>
    <Via>Giardini 15</Via>
    <Città>Pavullo n/F</Città>
    <CAP>41026</CAP>
    <Provincia>Modena</Provincia>
    <Paese>Italia</Paese>
  </Indirizzo>

  <Server>
    <Nome>WebServer</Nome>
    <Indirizzo>123.156.123.8</Indirizzo>
  </Server>

</Azienda>
```

Come si può vedere il documento integrato contiene due elementi `<Indirizzo>` che sono posizionati diversamente nella struttura.
Questo è certamente un problema per le applicazioni che tentino d'accedere ai dati contenuti nel documento XML, infatti, queste non potrebbero distinguere i due elementi con lo stesso nome.
Per risolvere il problema si utilizzano gi XML Namespaces, utilizzando tale tecnologia è possibile scrivere il documento XML nel modo seguente:

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

<Azienda>

  <ind:Indirizzo xmlns:ind="http://www.mysite.com/ind">
    <ind:Via>Giardini 15</Via>
    <ind:Città>Pavullo n/F</Città>
    <ind:CAP>41026</CAP>
    <ind:Provincia>Modena</Provincia>
    <ind:Paese>Italia</Paese>
  </ind:Indirizzo>

  <srv:Server xmlns:srv="http://www.mysite.com/srv">
    <srv:Nome>WebServer</Nome>
    <srv:Indirizzo>123.156.123.8</Indirizzo>
```

```
  </srv:Server>

</Azienda>
```

In questo modo l'applicazione che parsifica il documento XML è in grado, senza incertezze, di distinguere i vari elementi che formano la struttura del documento stesso, e di estrarre l'elemento giusto, indipendentemente dal nome che possiede. La discriminazione infatti è data dal namespace corrispondente.

Il concetto di XML Namespace può essere applicato anche agli attributi presenti nel documento XML.
Consideriamo il seguente payload XML di un messaggio SOAP (che sarà spiegato nel dettaglio nei prossimi capitoli):

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:sayHello xmlns:ns1="urn:HelloWorldServer2" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
">
<st xsi:type="xsd:string">Sandro</st>
</ns1:sayHello>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In caso vengono utilizzati alcuni namespace che sono standard per i payload XML trasmessi attraverso messaggi SOAP. Si tratta in particolare dei seguenti:

```
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"
```

Questi namespace permettono l'utilizzo di attributi speciali per trasmettere informazioni standard recepibili dagli oggetti in grado di comprendere un messaggio SOAP.
Ad esempio un elemento contenuto all'interno del documento XML precedente è il seguente:

```
<st xsi:type="xsd:string">Sandro</st>
```

In questo caso si è in grado di passare nel messaggio SOAP, oltre al valore del parametro `st`, anche il suo tipo.
In particolare l'attributo `type` (identificato univocamente dal namespace `xsi`) associato all'elemento `st`, specifica che si tratta di un elemento di tipo `string` (`string`, a sua volta, è identificato univocamente dal namespace `xsd`).

Parser XML- DOM e SAX

Una volta capita l'importanza del linguaggio XML e delle sue applicazioni, c'è da chiedersi in che modo sia possibile gestirne i contenuti dal punto di vista applicativo; in altre parole quali sono le interfacce di sviluppo in possesso di chi ha il compito d'elaborare le informazioni contenute nei documenti XML?

La risposta sta, principalmente, in due API per l'analisi di documenti XML, si tratta di DOM e SAX.

DOM

DOM è un'interfaccia proposta dal W3C ed è basata su un Object Model simile a quello utilizzato per descrivere, al suo interno, un documento HTML.
Utilizzando DOM un documento XML viene visto a livello applicativo come un normale modello ad oggetti, dotato di tutte le caratteristiche tipiche del paradigma Object Oriented.
Ogni oggetto possiede, quindi, attributi e metodi attraverso i quali è possibile manipolarne le caratteristiche.
Considerando la struttura gerarchica ad albero di cui è dotato un documento XML, questo tipo di rappresentazione appare la più indicata.
La struttura interna della gerarchia di oggetti, e la sua implementazione, dipendono ovviamente dalla piattaforma e dall'ambiente di sviluppo utilizzato, ma per supportare le API del DOM proposto dal W3C è necessario che qualsiasi implementazione esporti determinate interfacce, ovvero proprietà e metodi pubblici standard da utilizzare per accedere alla gerarchia interna degli oggetti costituenti il documento XML.
Utilizzando una qualsiasi implementazione DOM, il documento XML viene caricato integralmente in memoria e viene mappato sulla gerarchia d'oggetti di cui è composta la specifica, in questo modo, utilizzando le interfacce pubbliche esportate è possibile accedere in maniera trasparente a tutti gli oggetti interni del documento XML.
È previsto l'accesso alla gerarchia d'oggetti del documento XML per effettuare inserimenti, modifiche, ricerche e cancellazioni. In alcune implementazioni di DOM è possibile anche l'applicazione di uno Schema al documento in fase di caricamento oppure l'applicazione di trasformazioni attraverso l'utilizzo di XSL[1].
Uno dei grandi limiti delle attuali implementazioni della specifica DOM riguarda la necessità di dover caricare in memoria tutto il documento XML prima di poterlo processare.
Questo comportamento, in molti casi, risulta molto limitativo, in quanto non permette di elaborare documenti le cui dimensioni siano più grandi delle risorse computazionali disponibili. È necessario quindi trovare altre strade quando si desideri elaborare documenti molto grandi.

SAX

Una soluzione che non soffre dei limiti appena descritti è quella offerta da SAX 2.0 (Simple API for XML).
Mentre DOM si basa sul caricamento preventivo del documento, in modo tale da creare in memoria una struttura ad oggetti all'interno della quale l'applicazione possa navigare, la filosofia SAX si basa su un modello ad eventi.
L'applicazione potrà richiamare delle funzioni apposite quando, durante la lettura del documento, si verifichino degli eventi predeterminati.
L'esempio tipico d'utilizzo di un modello ad eventi di questo tipo è la necessità di effettuare operazioni specifiche sul documento XML nel momento in cui si incontra un determinato tag.
Durante il parsing, quindi, l'applicazione è in grado di sapere quando viene trovato un particolare tag, oppure quando si verifica una certa condizione, e può quindi effettuare delle operazioni sulla parte del documento interessata, ad esempio la sostituzione di un tag.

[1] XSL è formato da un vocabolario per la formattazione e da un linguaggio di trasformazione XSLT. XSL permette di trasformare la struttura di un documento XML in un'altra struttura, ad esempio HTML oppure WML (legato alla navigazione attraverso telefono cellulare WAP).

Questo approccio, come si può capire, non necessita del caricamento completo del documento in memoria per ottenerne una sua manipolazione.
Questo tipo di soluzione, però, soffre di un'altra tipologia di problemi, legata strettamente al mantenimento dello stato applicativo.
Durante il parsing del documento, infatti, si scatenano in maniera asincrona una serie di processi che hanno l'obiettivo di effettuare modifiche al documento XM, di conseguenza ogni processo non è in grado di avere informazioni legate al comportamento degli altri e questo può essere un limite strutturale.
Esistono altre proposte, successive a SAX e DOM, che pare abbiano aver superato queste limitazioni, ma al momento non esistono standard che ne uniformino il comportamento.

Capitolo 4- Specifica SOAP 1.2

Struttura di un messaggio SOAP (il payload XML)

La struttura di un messaggio SOAP prevede tre elementi:

- **Envelope** = costrutto sintattico per esprimere la struttura completa di un messaggio SOAP, al suo interno trovano spazio tutti gli altri elementi sintattici del messaggio. Si tratta di un messaggio protetto SOAP, tutto quello che è racchiuso nell'Envelope fa parte del messaggio SOAP, tutto quello che è all'esterno non ne fa parte.

- **Header** = elemento opzionale, progettato per facilitare ed individuare le elaborazioni richieste ai processori SOAP intermedi, incontrati nel cammino del messaggio dal mittente al destinatario finale.

- **Body** = elemento obbligatorio, indirizzato al destinatario finale, che contiene le informazioni vere e proprie del messaggio.

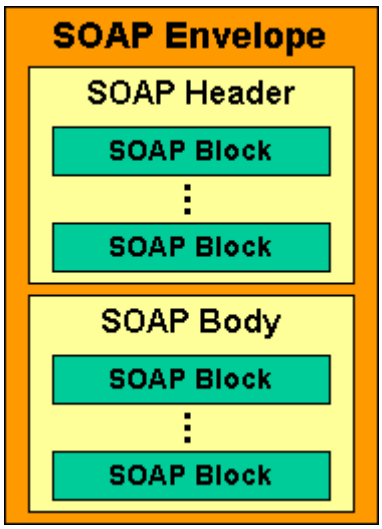


Figura 2:Struttura di un messaggio SOAP

Quindi un messaggio SOAP ha il seguente scheletro XML:

```
<env:Envelope>
   <env:Header>
   …
   </env:Header>
   <env:Body>
   …
```

```
   </env:Body>
</env:Envelope>
```

Nell'esempio che segue, l'impiegato di una compagnia di viaggi compila una prenotazione che verrà trasferita, attraverso un messaggio SOAP, al servizio di prenotazione per la sua registrazione.


```
<?xml version='1.0' ?>
<env:Envelope xmlns:env=http://www.w3.org/2001/12/soap-
envelope env:encodingStyle="http://example.com/encoding">
 <env:Header>
  <m:reservation
xmlns:m="http://travelcompany.org/reservation"
          env:actor="http://www.w3.org/2001/12/soap-
envelope/actor/next"
           env:mustUnderstand="true">
    <m:reference>uuid:093a2da1-q345-739r-ba5d-
pqff98fe8j7d</reference>
    <m:dateAndTime>2001-11-29T13:20:00.000-
05:00</m:dateAndTime>
  </m:reservation>
  <n:passenger xmlns:n="http://mycompany.com/employees"
          env:actor="http://www.w3.org/2001/12/soap-
envelope/actor/next"
           env:mustUnderstand="true">
   <n:name>John Q. Public</n:name>
  </n:passenger>
 </env:Header>
 <env:Body>
  <p:itinerary
xmlns:p="http://travelcompany.org/reservation/travel">
   <p:departure>
     <p:departing>New York</p:departing>
     <p:arriving>Los Angeles</p:arriving>
     <p:departureDate>2001-12-14</p:departureDate>
     <p:departureTime>late afternoon</p:departureTime>
     <p:seatPreference>aisle</p:seatPreference>
   </p:departure>
   <p:return>
     <p:departing>Los Angeles</p:departing>
     <p:arriving>New York</p:arriving>
     <p:departureDate>2001-12-20</p:departureDate>
     <p:departureTime>mid morning</p:departureTime>
     <p:seatPreference/>
   </p:return>
  </p:itinerary>
  <q:lodging
xmlns:q="http://travelcompany.org/reservation/hotel">
   <q:preference>none</q:preference>
  </q:lodging>
 </env:Body>
</env:Envelope>
```

Figura 3: Esempio di messaggio SOAP

Nell'esempio, la sezione Header contiene due header block, ognuno dei quali definisce il proprio namespace, i propri attributi e rappresenta un aspetto (informazioni aggiuntive)

nel processo d'elaborazione del Body. L'header block `reservation` contiene il riferimento e l'istante di prenotazione; l'header block `passenger` contiene il nominativo del passeggero.
La sezione Body contiene due body block: `itinerary` e `lodging`.
A sua volta, `itinerary`, contiene due elementi figli `departure` e `return` che specificano il luogo di partenza, il luogo d'arrivo, la data di partenza, l'ora di partenza e il posizionamento in aereo per il viaggio d'andata e di ritorno rispettivamente.
Occorre sottolineare che il contenuto delle sezioni Header e Body è definito dall'applicazione in uso e non fa parte delle specifiche SOAP, sebbene le specifiche indichino come tali sezioni devono essere maneggiate.

Envelope

L'elemento `<Envelope>`, come si può intuire, rappresenta la "busta" che viene utilizzata come contenitore per tutto il payload XML.
Tutte le informazioni che viaggiano dal mittente al destinatario devono essere contenute all'interno di questo elemento che è la radice del documento, come da specifica XML.
La specifica SOAP 1.2 a proposito della sezione Envelope, prescrive le seguenti regole:

- Può contenere dichiarazioni di namespace.
- Può contenere l'attributo speciale `env:encodingStyle` che, salvo indicazioni diverse, si riferisce a tutta la sezione Envelope.
- Può contenere attributi aggiuntivi, qualificati attraverso l'utilizzo di namespace.
- Può contenere la sezione Header.
- Deve contenere la sezione Body.
- Può contenere qualsiasi numero di sottoelementi, a condizione che siano qualificati attraverso l'utilizzo di namespace e seguano la sezione Body.

L'attributo encodingStyle

L'attributo `env:encodingStyle` viene utilizzato per indicare l'indirizzo di uno schema XML contenente le definizioni di regole standard che sono utilizzate per la serializzazione delle informazioni.
Nel caso in cui questo attributo fosse assente, viene preso come default lo schema standard `http://schemas.xmlsoap.org/soap/encoding`.
L'attributo `env:encodingStyle`, oltre che nell'Envelope, può essere adottato anche a livello di Header e di Body; resta inteso, che tale valore, rimane valido per tutto il documento a seguire se non viene specificato un diverso valore dell'attributo `env:encodingStyle`.
Questo è un grande punto di forza della specifica SOAP. Essa prescrive infatti l'utilizzo di determinate regole di encoding, ma non esclude la possibilità di adoperare regole di encoding alternative così da essere flessibile in funzione delle esigenze di sviluppo delle applicazioni.
Un client SOAP, durante la preparazione del payload XML, può utilizzare regole di encoding alternative. Per permettere al server SOAP, che riceverà il messaggio contenente tale payload, di comprendere la codifica del documento XML contenuto nel payload stesso, è necessario che utilizzi i namespace in modo da indicare la posizione del file XML che descrive le regole di encoding utilizzate.

Attributi aggiuntivi

La sezione SOAP Envelope può contenere un numero qualsiasi di attributi aggiuntivi, anche se l'utilizzo di questa possibilità viene sconsigliata.
Scrivendo documenti XML, ci si rende conto che vi sono due tipologie, diametralmente opposte, per effettuare la codifica dell'informazione. La prima comporta l'utilizzo di pochi elementi, ma dotati di molti attributi; la seconda, invece, prevede l'utilizzo di molti elementi, ciascuno dotato di pochi attributi.

Vediamo un esempio riguardo alla prima tipologia di codifica:

```
<element1 attr1="value1" attr2="value2" attr3="value3">
   value
</element1>
```

Mentre per la seconda tipologia di codifica:

```
<element1 attr1="value1">
  <element2 attr2="value2">
    <element3 attr3="value3">
         value
    </element3>
  </element2>
</element1>
```

Nonostante la prima soluzione sia migliore sotto molti punti di vista, SOAP privilegia la seconda, cioè l'utilizzo di molti elementi dotati di pochi attributi. La motivazione, fondamentalmente, risiede nel fatto che questa è preferita dal punto di vista computazionale dai parser XML. Per il parser, infatti, il comportamento naturale è la ricerca e l'analisi di un particolare elemento. All'interno di questo, successivamente, sarà possibile ricercare il singolo attributo che interessa.
Nel caso in cui, però, si debba ricercare, all'interno del documento XML un particolare attributo, sarà necessario effettuare la parificazione dell'intero documento.

Elementi addizionali

La specifica SOAP prevede, all'interno dell'elemento Envelope, la presenza di due sottoelementi fondamentali, la sezione Header (facoltativa) e la sezione Body (obbligatoria).
In ogni caso non vi sono limitazioni sull'utilizzo di elementi addizionali che possono essere inseriti dopo la sezione Body.
Se presenti, questi saranno trattati come elementi di controllo di situazioni anomale, come il supporto a piattaforme particolari, l'attivazione remota di procedure, l'attivazione di servizi, ecc. Tutte informazioni, in ogni caso, che per qualche motivo non è possibile collocare all'interno della sezione Header.
L'unico vincolo richiesto per l'inserimento di elementi aggiuntivi all'interno della sezione Envelope è il fatto che questi, oltre ad essere successivi alla sezione Body, debbano essere qualificati attraverso l'utilizzo di namespace, questo permette di avere uno schema di riferimento per interpretare le informazioni contenute negli elementi aggiuntivi. In caso contrario lo schema tradizionale non potrà validare il contenuto degli elementi aggiuntivi e ritornerà una condizione d'errore.
Lo schema di una situazione di questo tipo è il seguente:

```
<env:Envelope xmlns:env ="http://www.w3.org/2001/12/soap-
envelope/"

xmlns :OTHER_ENC_RULES="http://www.mysite.com/soap/myenc_rul
es/"
```

```
xmlns :encodingStyle= ”http://schemas.xmlsoap.org/soap/encod
ing/”>

  <env :Header>
  ...
  </env :Header>

  <env :Body>
  ...
  </env :Body>

  <OTHER_ENC_RULES :OtherElement>
  ...
  </OTHER_ENC_RULES :OtherElement>

</env:Envelope>
```

Header

La sezione Header è la porzione del payload all'interno della quale è possibile specificare dei parametri di comunicazione ed informazioni aggiuntive sul messaggio trasmesso.
Questa sezione deve rispettare le seguenti regole:

- La sezione Header è facoltativa, ma se presente deve sempre essere la prima all'interno dell'elemento Envelope.

- La sezione Header deve rispettare le regole di encoding standard specificate in `http://schemas.xmlsoap.org/soap/encoding`.
  Regole di codifica differenti devono essere dichiarate preventivamente utilizzando il valore dell'attributo `env:encodingStyle`.

- La sezione Header può contenere degli attributi aggiuntivi, qualificati attraverso l'utilizzo di namespace.

- La sezione Header può contenere l'attributo `env:mustUnderstand` valorizzato a “0” (“false”) oppure ad ”1” (“true”).

- La sezione Header può contenere l'attributo `env:root` valorizzato a “0” (“false”) oppure ad ”1” (“true”).

- La sezione Header può contenere dei sottoelementi, qualificati attraverso l'utilizzo di namespace.

- La sezione Header deve essere autoreferenziale, questo significa che un elemento esterno non può essere utilizzato all'interno.

- Il parser XML che interpreterà il Body deve preventivamente aver interpretato la sezione Header, in modo da poterne utilizzare i riferimenti.

- Può contenere informazioni sullo stato del sistema, sullo stato del client, sulle impostazioni relative alla sessione corrente dell'oggetto e tutto ciò che può essere utile alla personalizzazione della comunicazione.

Ogni sottoelemento della sezione Header prende il nome di header block, il quale deve rispettare le seguenti regole:

- Essere qualificato attraverso l'utilizzo di namespace, come già anticipato.

- Può contenere l'attributo `env:mustUnderstand` valorizzato a "0" ("false") oppure ad "1" ("true").

- Può contenere l'attributo `env:encodingStyle`.

- Può contenere l'attributo `env:actor`.

Azioni di un nodo SOAP e l'attributo actor

Le seguenti azioni sono intraprese da un nodo Soap al momento della ricezione di un messaggio:

1. Analisi sintattica del messaggio per verificare che l'Header e il Body contengano elementi XML corretti.

2. Elaborazione del messaggio solo sugli header block che nell'attributo `env:actor` contengono l'URI del nodo stesso.

L'attributo `env:actor` può essere un qualunque URI oppure assumere uno dei valori standard:

- **none** = tale header block non deve essere elaborato da nessun nodo;

- **next** = tale header block può essere elaborato da tutti i nodi incontrati nel cammino;

- **anonymous** = tale header block può essere elaborato solo dal nodo finale; il valore anonymous non si può esprimere esplicitamente, ma viene assunto quando nell'header non compare l'attributo actor. Infatti per la sezione Body non esiste l'attributo actor, perché questo può assumere solo il valore anonymous, essendo destinato solo al nodo finale.

L'attributo mustUnderstand

Se nell'header block compare l'attributo `env:mustUnderstand` con il valore true allora il nodo indicato nell'attributo actor deve obbligatoriamente elaborare l'header block stesso.
Nel caso in cui il nodo non sia in grado di elaborare tale header block tutte le elaborazioni a seguire terminano e si genera un errore.
Da quanto detto si capisce che se un elemento Header non contiene l'attributo `env:mustUnderstand` con valore true, non c'è garanzia che il nodo destinatario lo interpreti, può quindi essere ignorato.

L'attributo Root

Un altro attributo facoltativo che può essere introdotto all'interno di un sottoelemento della sezione Header (o della sezione Body, come vedremo) è `env:Root`.
L'obiettivo di questo attributo è fornire all'elemento in questione un carattere di predominanza rispetto ad altri all'interno della stessa struttura, in modo che il parser di destinazione possa discriminare sull'importanza da assegnare agli elementi e possa trattarli nell'ordine corretto.

Facciamo un esempio:

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

```
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-
envelope" >
 <env:Header>
       <t:transaction1

xmlns:t="http://thirdparty.example.org/transaction"
           env:encodingStyle="http://example.com/encoding"
           env:mustUnderstand="true"
           env:Root="1" >
               5
       </t:transaction1>

       <t:transaction2

xmlns:t="http://thirdparty.example.org/transaction"
           env:encodingStyle="http://example.com/encoding"
           env:mustUnderstand="true" >
               15
       </t:transaction2>

 </env:Header>
…
…
```

Figura 4:Messaggio SOAP con l'attributo Root attivo

Come si vede l'attributo env:Root ha l'obiettivo di indicare al server quale elemento della sezione Header, fra quelli inviati, ha la priorità su tutti gli altri.
Il valore dell'attributo può essere "0" ("false") oppure "1" ("true"), il primo è di default.
Il valore "1" può essere indicato per un solo elemento della sequenza di elementi. Se il server riceve più di un elemento con il parametro `env:Root="1"` deve restituire errore.

Body

Questa è la parte del messaggio SOAP obbligatoria che descrive le informazioni, il comportamento richiesto al nodo finale del percorso di comunicazione.
Il Body deve rispettare le seguenti regole:

- La sezione Body è obbligatoria.
- La sezione Body deve rispettare le regole di encoding standard specificate in `http://schemas.xmlsoap.org/soap/encoding`.
  Regole di codifica differenti devono essere dichiarate preventivamente utilizzando il valore dell'attributo `env:encodingStyle`.
- La sezione Body non può contenere degli attributi aggiuntivi.
- La sezione Body può contenere dei sottoelementi, qualificati attraverso l'utilizzo di namespace.

Ogni sottoelemento della sezione Body deve rispettare le seguenti regole:

- Essere qualificato attraverso l'utilizzo di namespace, come già anticipato.
- Può contenere l'attributo `env:encodingStyle`.

Il Body può essere utilizzato semplicemente per trasportare informazioni (semplici messaggi) oppure, come vedremo fra breve, per richiedere l'invocazione di metodi remoti (RPC).

Situazioni d'errore

Durante la comunicazione o l'elaborazione di messaggi SOAP esiste la possibilità che ci siano dei problemi, con la conseguente presenza di errori.
Gli errori possono essere fondamentalmente di due tipi: errori di trasporto ed errori SOAP.
Nel primo caso la gestione degli errori è a carico del protocollo di trasporto (ad esempio un messaggio HTTP con codice d'errore 404 per un file non trovato).
Nel secondo caso, la gestione degli errori è a carico dei componenti SOAP, i quali segnalano le eccezioni attraverso l'utilizzo dell'elemento speciale `Fault`.

Le regole che devono essere seguite dall'elemento `Fault` sono le seguenti:

- Se presente, l'elemento `Fault` deve comparire all'interno dell'elemento Body.
- L'elemento `Fault` può comparire al massimo una volta all'interno dell'elemento Body.
- L'elemento `Fault` può contenere i seguenti sottoelementi:
  - `faultcode` = specifica il codice (standard) d'errore, è obbligatorio e deve essere qualificato;
  - `faultstring` = fornisce una descrizione testuale (non standard, dipendente dall'implementazione SOAP in uso) dell'errore, è obbligatorio e deve essere qualificato;
  - `faultactor` = facoltativo, specifica l'URI del nodo che ha generato l'errore, se non specificato si sottintende il mittente del messaggio d'errore;
  - `details` = facoltativo, lasciato alla specifica applicazione per ulteriori dettagli. Può contenere attributi ed elementi (che devono essere qualificati e possono avere l'attributo `env:encodingStyle`). Si noti che il sottoelemento `details` non può descrivere dettagli su errori relativi ad header block; questi dettagli devono essere descritti utilizzando, a sua volta, degli header block.

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-
envelope"
            xmlns:f='http://www.w3.org/2001/12/soap-faults'>
  <env:Body>
    <env:Fault>
      <faultcode>env:Receiver</faultcode>
      <faultstring>Processing Error</faultstring>
      <detail>
        <e:myfaultdetails
xmlns:e="http://travelcompany.example.org/faults" >
          <message>Name does not match card number</message>
          <errorcode>999</errorcode>
        </e:myfaultdetails>
      </detail>
```

```
    </env:Fault>
 </env:Body>
</env:Envelope>
```

Figura 5:Messaggio SOAP d'errore

Un particolare errore avviene nel caso in cui un nodo riceve un header block da elaborare obbligatoriamente (attributo `env:musUnderstand="true"`), ma non riesce a capirlo/elaborarlo. In questo caso il messaggio d'errore generato, oltre al relativo elemento `Fault` (il sottoelemento `faultcode` ha valore `MustUnderstand`), contiene un particolare header: `Misunderstood`. All'interno di quest'ultimo, nell'attributo `qname`, viene specificato l'header block che non è stato capito/elaborato.

Questo risulta fondamentale nel caso vi siano più header block obbligatori per lo stesso nodo, solo in questo modo è possibile sapere quale header block ha generato l'errore.

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-
envelope"
            xmlns:f='http://www.w3.org/2001/12/soap-faults'>
 <env:Header>
    <f:Misunderstood qname='m:reservation'

xmlns:m="http://travelcompany.example.org/reservation"  />
 </env:Header>
 <env:Body>
    <env:Fault>
      <faultcode>env:MustUnderstand</faultcode>
      <faultstring>Header not understood</faultstring>
    </env:Fault>
 </env:Body>
</env:Envelope>
```

Figura 6: Messaggio SOAP d'errore per header block obbligatorio non capito

La modalità di propagazione del messaggio d'errore dipende dal protocollo di trasporto scelto.

In riferimento al namespace http://www.w3.org/2001/12/soap-envelope, possiamo trovare i seguenti codici d'errore, che l'applicazione può utilizzare come valore del sottoelemento `faultcode`:

- `VersionMismatch` = è stato trovato un namespace non valido nell'elemento Envelope.
- `MustUnderstand` = un elemento, della sezione Header contenete l'attributo `env:MustUnderstand` attivo, non è stato capito/elaborato dal nodo destinatario.
- `Sender` = il messaggio non è scritto in modo conforme alla specifica, oppure non contiene le informazioni attese nell'ordine stabilito.
- `Receiver` = il messaggio non può essere correttamente processato, ma le ragioni non sono da ricercare all'interno del messaggio stesso.
- `DTDNotSupported` = Il messaggio SOAP contiene un DTD (Document Type Definition).

RPC (Remote Procedure Call)

Uno degli obiettivi principali di SOAP è l'incapsulamento di chiamate remote a metodi/procedure sfruttando l'estensibilità e la flessibilità di XML.

In generale per invocare una RPC sono necessarie le seguenti informazioni:

- l'URI del nodo SOAP obiettivo[2];
- il nome del metodo o della procedura;
- una firma opzionale del metodo o della procedura;
- i parametri del metodo o della procedura;
- uno o più header opzionali per informazioni supplementari.

In linea di principio utilizzando SOAP per le RPC si è indipendenti dal protocollo di trasporto utilizzato. In realtà, ovviamente, vi sono protocolli che meglio si adattano alle RCP. E' bene che nella progettazione delle applicazioni si tenga presente il protocollo che meglio si adatta agli scopi. Ad esempio, il protocollo HTTP, mappa naturalmente le invocazioni RPC con le rispettive risposte RPC.

RPC e SOAP Body

La gestione RPC attraverso messaggi SOAP prevede i seguenti componenti nella sezione Body:

- **Call**: la chiamata al metodo remoto ed i suoi parametri.
- **Response**: il risultato d'elaborazione del metodo remoto (con eventuali parametri di ritorno).

Call

L'invocazione RPC è modellata come una struct nel Body SOAP:

- L'invocazione è vista come una singola struct contenente un elemento figlio per ogni parametro di [in] o di [in/out]. La struct ha lo stesso nome e lo stesso tipo del metodo invocato.
- Ogni parametro di [in] o di [in/out] è visto come un elemento figlio della struct, dove il nome rappresenta il nome del parametro e il tipo rappresenta il tipo di parametro. Questi devono comparire nello stesso ordine in cui sono definiti nel metodo.

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-
envelope" >
 <env:Header>
   <m:reservation
xmlns:m="http://travelcompany.example.org/reservation"
```

[2] SOAP affida al protocollo di trasporto la gestione dell'URI; ad esempio, in HTTP, l'URI è presente nell'invocazione POST.

```
          env:actor="http://www.w3.org/2001/12/soap-
envelope/actor/next"
           env:mustUnderstand="true">
    <m:reference>uuid:093a2da1-q345-739r-ba5d-
pqff98fe8j7d</reference>
    <m:dateAndTime>2001-11-29T13:36:50.000-
05:00</m:dateAndTime>
   </m:reservation>
   <t:transaction

xmlns:t="http://thirdparty.example.org/transaction"
           env:encodingStyle="http://example.com/encoding"
           env:mustUnderstand="true" >
               5
   </t:transaction>
 </env:Header>
 <env:Body>
    <m:reserveAndCharge
        env:encodingStyle="http://www.w3.org/2001/12/soap-
encoding"
             xmlns:m="http://travelcompany.example.org/" >
      <n:name
xmlns:n="http://mycompany.example.com/employees">
           John Q. Public
      </n:name>
        <o:creditCard
xmlns:o="http://mycompany.example.com/financial">
           <o:number>123456789099999</o:number>
           <o:expiration>2005-02</o:expiration>
         </o:creditCard>
       </m:reserveAndCharge>
  </env:Body>
</env:Envelope>
```

Figura 7: Messaggio SOAP per richiesta RPC

Nell'esempio sopra, l'elemento `reserveAndCharge` indica il metodo da invocare, dove gli elementi figli `name` e `creditCard` (a sua volta composto da due sottoelementi `number` ed `expiration`) sono i due parametri d'ingresso del metodo.

Response

La risposta RPC è modellata come una struct nel Body SOAP:

- La risposta è vista come una singola struct contenente un elemento figlio per ogni parametro di [out] o di [in/out]. L'ordine di ritorno deve rispettare quello specificato nel metodo, preceduto dall'eventuale valore di ritorno. La struct di risposta ha lo stesso nome del metodo invocato nella request con l'aggiunta del suffisso *Response*.

- Gli elementi figli della struct hanno un nome, che rappresenta il nome del parametro, e un tipo, che rappresenta il tipo di parametro. L'eventuale valore di ritorno del metodo, in genere, ha nome "result" (anche se in alcune implementazioni troviamo il nome "return"). Non si hanno parametri di ritorno se il metodo invocato è void (procedura).

Errori nelle invocazioni sono gestiti secondo le regole RPC Fault. Se il protocollo di trasporto richiede delle regole aggiuntive, queste devono essere seguite.

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-
envelope" >
 <env:Header>
       <t:transaction

xmlns:t="http://thirdparty.example.org/transaction"
           env:encodingStyle="http://example.com/encoding"
           env:mustUnderstand="true" >
               5
       </t:transaction>
 </env:Header>
 <env:Body>
       <m:reserveAndChargeResponse

env:encodingStyle="http://www.w3.org/2001/12/soap-encoding"
             xmlns:m="http://travelcompany.example.org/" >
           <m:confirmation>
               <reference>FT35ZBQ</reference>
               <viewAt>

http://travelcompany.example.org/reservations?code=FT35ZBQ
               </viewAt>
           </m:confirmation>
       </m:reserveAndChargeResponse>
   </env:Body>
</env:Envelope>
```

Figura 8: Messaggio SOAP di risposta alla chiamata RPC precedente

Nell'esempio sopra, l'elemento `reserveAndChargeResponse` contiene la risposta del metodo invocato in precedenza. L'elemento figlio confirmation (a sua volta composto da due sottoelementi `reference` e `viewAt`) è il parametro d'uscita del metodo.

RPC e SOAP Header

Altre informazioni utili per l'invocazione, l'elaborazione e la risposta del metodo, ma che non sono formalmente parte del metodo, possono essere trasportate attraverso gli header block.
Nell'esempio di Figura 7, attraverso l'header block `transaction`, si specifica un'informazione aggiuntiva evitando di intaccare direttamente la chiamata al metodo remoto.

RPC Faults

La gestione degli errori per le RPC avviene, ancora una volta, utilizzando l'elemento `Fault` nel Body del messaggio SOAP di risposta.
I principali codici d'errore che il sottoelemento `faultcode` può assumere sono i seguenti (in ordine decrescente di precedenza):

- `env:Server` = il server non può gestire il messaggio (ad esempio non ha sufficiente memoria).

- `env:DataEncodingUnknown` = il server non comprende come sono stati codificati i dati all'interno del Body e/o dell'Header di richiesta (il valore dell'attributo `env:encodingStyle`).

- `rpc:ProcedureNotPresent`[3] = il server non riesce a trovare la procedura specificata.

- `rpc:BadArguments`[4] = il server non riesce ad analizzare i parametri oppure non c'è corrispondenza fra ciò che il server si aspetta e ciò che gli ha inviato il client.

Per quanto riguarda i sottoelementi faultstring e detail il loro valore dipende dall'implementazione SOAP in uso.

Serializzazione dei dati

In questa sezione si analizzano le metodologie di passaggio dei parametri (d'ingresso e/o d'uscita) fra client e server SOAP. Inoltre si prende in esame i diversi tipi di payload ottenuti serializzando diversi tipi di dato (semplici, strutturati ed array).

Tipologie di passaggio dei parametri

Esistono due tipiche tipologie di passaggio dei parametri:

- Passaggio per valore.
- Passaggio per riferimento.

Nel primo caso viene passato a server remoto un parametro che è di fatto una copia della variabile di partenza. Qualunque modifica venga fatta a questo parametro dalla funzione di destinazione, questa non si ripercuoterà in alcun modo sul valore della variabile di partenza. In questo caso si parla di parametri di tipo in.
Nel secondo caso, invece, si richiede di trasferire alla funzione di destinazione , e quindi al server che la ospita, tutto lo stack di chiamata. Infatti, in questo caso, è necessario che la funzione di destinazione conosca non soltanto il valore dei parametri , ma abbia anche una rappresentazione della loro collocazione fisica nella memoria dell'applicazione chiamante.
In questo modo il server sarà in grado di trasformare tale rappresentazione in uno schema di memoria reale creandosi un vero e proprio stack di chiamata. Soltanto a questo punto il server sarà in grado di effettuare modifiche ai valori dei parametri.
Al termine delle operazioni d'elaborazione il server dovrà ritrasmettere al chiamante la rappresentazione del suo stack di chiamata in modo che il client possa ricostruirsela in locale simulando il comportamento di una chiamata locale (e non remota).
In questo caso si parla di parametri di tipo in/out.

Serializzazione del passaggio per valore

Il passaggio per valore significa che l'oggetto passato come parametro non ha referenze all'interno dell'applicazione.
Il caso tipico è il passaggio di un numero ad una funzione di elaborazione che deve, ad esempio, incrementarlo di una unità.

[3] *rcp* sottintende il nome del namespace associato
[4] *rcp* sottintende il nome del namespace associato

In questo caso, qualunque sia il linguaggio di programmazione utilizzato, quello che stiamo passando alla funzione remota è una copia del valore contenuto nella variabile.

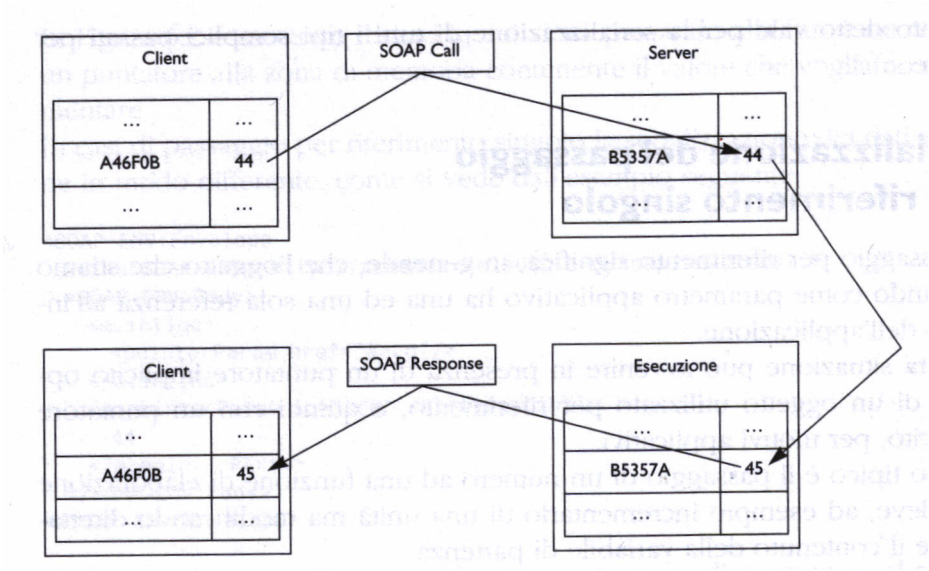


Figura 9: Passaggio di parametri per valore.

In questi casi la serializzazione dei dati può avvenire in maniera piuttosto semplice, come si vede nell'esempio seguente:

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope" >
  <SOAP-ENV:Body>
    <m:intInc xmlns:m="http://mysite.com/intFunctions">
      <IntParam>44</IntParam>
    </m:intInc>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Quanto detto vale per tutti i tipi semplici passati per valore.

Serializzazione del passaggio per riferimento singolo

Il passaggio per riferimento significa, in generale, che l'oggetto che stiamo passando come parametro applicativo ha una ed una sola referenza all'interno dell'applicazione.
Il caso tipico è il passaggio di un numero ad una funzione di elaborazione che deve, ad esempio, incrementarlo di una unità ma modificando direttamente il contenuto della variabile di partenza.

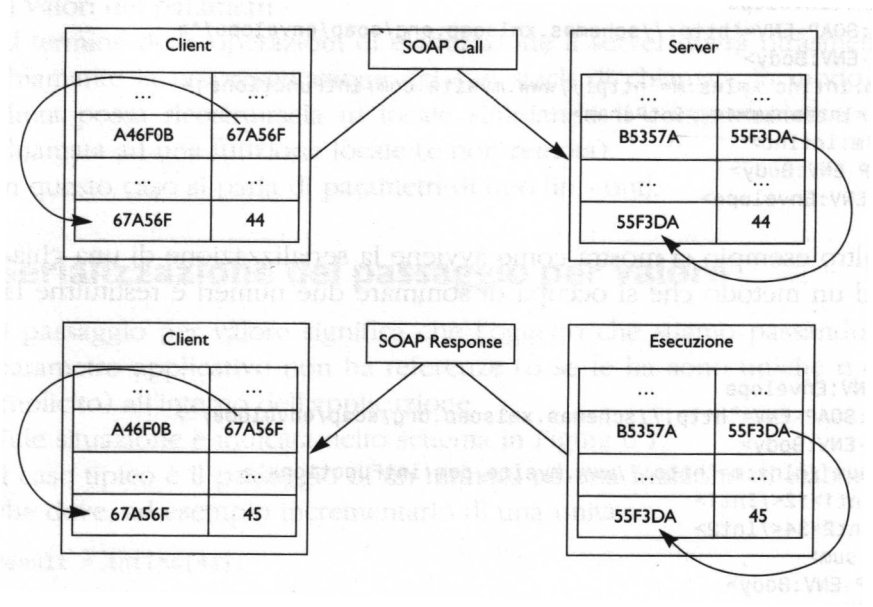


Figura 10:Passaggio di parametri per riferimento singolo.

In caso di passaggio per riferimento singolo la serializzazione dei dati avviene in modo differente, come si vede nell'esempio seguente:

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope" >
  <SOAP-ENV:Body>
    <m:intInc xmlns:m="http://mysite.com/intFunctions">
      <pointerParam href="#arg"/>
    </m:intInc>
    <m:pointerParam id="arg"
       xmlns:m="http://mysite.com/intFunctions">
      44
    </m:pointerParam>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

L'obiettivo di questa serializzazione è quello di trasmettere al server, oltre al valore del parametro d'elaborazione, anche la struttura della memoria dell'applicazione chiamante, in modo che questa essere replicata sull'applicazione server. Il comportamento della funzione chiamata, pur essendo remota, deve essere analogo a quello che avrebbe una funzione locale.

Serializzazione del passaggio per riferimento multiplo

Il passaggio per riferimento multiplo consiste nell'avere un oggetto referenziato contemporaneamente da più parti all'interno dello stack applicativo.

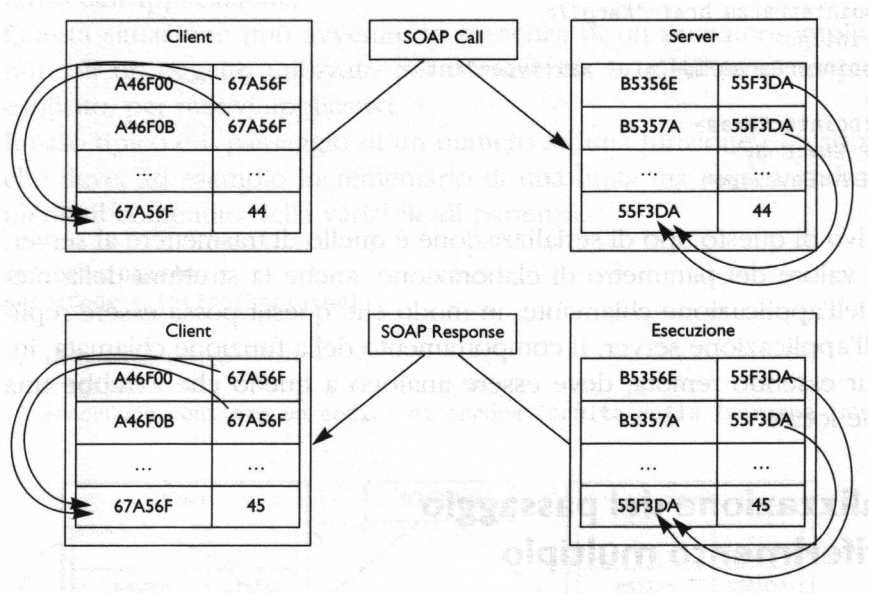


Figura 11:Passaggio di parametri per riferimento multiplo.

In caso di passaggio per riferimento multiplo la serializzazione dei dati avviene in modo leggermente più complesso, come si vede nell'esempio seguente:

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope" >
  <SOAP-ENV:Body>
    <m:intInc xmlns:m="http://mysite.com/intFunctions">
      <pointerParam1 href="#arg"/>
      <pointerParam2 href="#arg"/>
    </m:intInc>
    <m:pointerParam id="arg"
       xmlns:m="http://mysite.com/intFunctions">
      44
    </m:pointerParam>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Come si vede dagli esempi la strategia per serializzare oggetti che devono essere passati per riferimento consiste nell'utilizzare l'attributo `href` di XML (multi-referencing), il quale permette di collegare un puntatore virtuale (cioè il cui scopo è limitato al solo codice XML del payload) al valore reale dell'oggetto (anch'esso contenuto nel codice XML serializzato).

Serializzazione dei dati in XML

Il linguaggio XML si appoggia, per la descrizione dei tipi di dati, al linguaggio XML-Schema, n particolare alla sua seconda parte: "Datatypes".
Vi sono due namespace che vengono normalmente utilizzati all'interno dei payload XML trasportati da SOAP, si tratta di `xsd` ed `xsi`.
Il seguente esempio mette in risalto l'utilizzo di questi namespace:

```
<SOAP-ENV:Envelope
```

```
 xmlns:SOAP-ENV=”http://schemas.xmlsoap.org/soap/envelope”
 xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
 xmlns:xsd=”http://www.w3.org/2001/XMLSchema”>
  <SOAP-ENV:Body>
    <m:methodName xmlns:m=”some-URI”
      <Param1 xsi:type=”xsd:integer”>5</Param1>
      <Param2 xsi:type=”xsd:string”>una stringa</Param2>
    </m:methodName>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Come si può vedere si tratta di due namespace utilizzati all’interno di un payload di una chiamata SOAP, ed ognuno fa riferimento ad un certo URI contenente le specifiche di definizione.
In particolare ci troviamo nella seguente situazione:

- `xsd=”http://www.w3.org/2001/XMLSchema”`, è un namespace che identifica i possibili tipi di dato utilizzabili all’interno del documento XML. Il match viene eseguito con il documento specifico di XML-Schema.

- `xsi=”http://www.w3.org/2001/XMLSchema-instance”`, è un namespace che identifica le possibili istanze di dati utilizzabili all’interno del documento XML. Il match viene eseguito con il documento specifico di XML-Schema-Instance.

Da quanto detto si può capire il significato del codice `xsi:type=”xsd:integer”` che serve a descrivere, per il parametro a cui si riferisce, il tipo di dato di riferimento. Quello che ci serve infatti è descrivere il tipo di parametro, ed a questo pensa il codice `xsi:type`, poi vogliamo dare un valore all’istanza descritta da `xsi`, ed a questo provvede il codice `xsd:integer`.

Tipi di dato semplici

Utilizzando la specifica XML-Schema è possibile utilizzare i seguenti tipi di base:

string	Stringa di caratteri
Boolean	Valore booleano
int	Valore numerico intero
float	Valore numerico in virgola mobile
timeInstant	Data ed ora codificate in stringa
timeDuration	Data estesa ed ora codificate in stringa
Binary	Elemento binario
Uri	Elemento di localizzazione URI
Language	Elemento di riconoscimento della lingua

In associazione a questi tipi di base esistono anche dei tipi derivati direttamente da essi. Seguono alcuni tipi di dato derivati direttamente:

Name	Attributo Name di XML
ID	Attributo ID di XML
non-negative integer	Valore numerico intero positivo, compreso lo zero
positiveInteger	Valore numerico intero positivo
Date	Valore che rappresenta una data
Time	Valore che rappresenta un istante di tempo

In genere i tipi di dato semplice vengono serializzati utilizzando semplici stringhe di caratteri, l'unica condizione che deve essere rispettata è che ogni tipo semplice deve contenere un dato conforme al proprio tipo.

Tipi di dato composti

Esempi di tipi di dato composti sono le strutture (che permettono l'associazione in classi di oggetti eterogenei) e gli array (che permettono l'unione di oggetti omogenei).
Ovviamente unendo i concetti di strutture ed array è possibile avere array di strutture oppure array di array (cioè array multidimensionali) in modo da coprire in massima parte le più comuni esigenze implementative.
Naturalmente le specifiche di serializzazione sono diverse a seconda della tipologia di tipo composto che si sta trattando. Vediamo qualche esempio.

Strutture

In una sintassi simile al linguaggio C si può avere una struttura del tipo:

```
struct Libro {
              int id;
              string titolo;
              string autore;
              string ISBN;
            } Libro;
```

Questa struttura può essere serializzata nel seguente modo:

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <m:stampaLibro xmlns:m="http://mysite.com/Libro">
      <Libro href="#arg1"/>
    </m:stampaLibro>
    <m:Libro id="arg1"
       xmlns:m="http://mysite.com/Libro">
      <id xsi:type="xsd:integer">184</id>
      <titolo xsi:type="xsd:string">My book</titolo>
      <autore xsi:type="xsd:string">Name</autore>
      <ISBN xsi:type="xsd:string">1234-345664-2311</ISBN>
    </m:Libro>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

L'oggetto struttura, nel nostro caso Libro, viene serializzato come un puntatore `href="#arg1"` che rimanda, nel payload, alla zona nella quale l'oggetto viene trattato e dove viene scomposto in tutte le sue parti.
All'interno di quest'ultima tutti i tipi di dato semplice vengono serializzati immediatamente (come nel nostro esempio), mentre i tipi di dato composto vengono a loro volta gestiti come valori referenziati da puntatori.

Array

Supponiamo di avere il seguente array di stringhe:

```
string Mesi[12]= {"Gennaio", "Febbraio", "Marzo",
```

```
                    "Aprile", "Maggio", "Giugno",
                    "Luglio", "Agosto", Settembre",
                    "Ottobre", "Novembre" , "Dicembre" }
```

In SOAP avremo la corrispondente serializzazione:

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"
 xmlns:SOAP-ENC="http://www.w3.org/2001/12/soap-encoding"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <m:StampaArray xmlns:m="http://mysite.com/Array">
      <arrayMesi href="#array"/>
    </m:StampaArray>
    <SOAP-ENC:Array id="array"
       SOAP-ENC:arrayType="xsd:string[12]">
      <SOAP-ENC:string>Gennaio</string>
      <SOAP-ENC:string>Febbraio</string>
      <SOAP-ENC:string>Marzo</SOAP-ENC:string>
      <SOAP-ENC:string>Aprile</SOAP-ENC:string>
      <SOAP-ENC:string>Maggio</SOAP-ENC:string>
      <SOAP-ENC:string>Giugno</SOAP-ENC:string>
      <SOAP-ENC:string>Luglio</SOAP-ENC:string>
      <SOAP-ENC:string>Agosto</SOAP-ENC:string>
      <SOAP-ENC:string>Settembre</SOAP-ENC:string>
      <SOAP-ENC:string>Ottobre</SOAP-ENC:string>
      <SOAP-ENC:string>Novembre</SOAP-ENC:string>
      <SOAP-ENC:string>Dicembre</SOAP-ENC:string>
    </SOAP-ENC:Array>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Come si può vedere la trasmissione di array implica la spedizione al server di un payload contenente tutte le informazioni relative all'array stesso, sia dal punto di vista dimensionale (`"xsd:string[12]"`) che dal punto di vista del tipo di dato contenuto (`SOAP-ENC:arrayType`).

Esistono casi, però, nei quali questo tipo di serializzazione non è conveniente. Si pensi, ad esempio ad un array di dimensioni molto elevate, ma con pochi elementi valorizzati.
Si consideri il seguente array:

```
int postiDisponibili[10000];
```

In questo modo abbiamo creato un array vuoto pronto a contenere 10000 interi. Supponiamo che solo una piccola parte dell'array contenga informazioni significative, dopo la seguente inizializzazione:

```
postiDisponibili[245]=1;
postiDisponibili[345]=1;
postiDisponibili[1365]=1;
postiDisponibili[4566]=1;
postiDisponibili[8988]=1;
```

Se ora vogliamo richiamare un metodo remoto, inviandogli solo la parte d'array significativa, è necessario procedere in modo diverso dal caso precedente.

```
<SOAP-ENV:Envelope
```

```
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"
 xmlns:SOAP-ENC="http://www.w3.org/2001/12/soap-encoding"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <m:StampaPostiDisp xmlns:m="http://mysite.com/Array">
      <arrayInt href="#array"/>
    </m:StampaPostiDisp>
    <SOAP-ENC:Array id="array"
       SOAP-ENC:arrayType="xsd:int[10000]">
      <SOAP-ENC:int SOAP-ENC:position="245">1</SOAP-ENC:int>
      <SOAP-ENC:int SOAP-ENC:position="345">1</SOAP-ENC:int>
      <SOAP-ENC:int SOAP-ENC:position="1365">1</SOAP-
ENC:int>
      <SOAP-ENC:int SOAP-ENC:position="4566">1</int>
      <SOAP-ENC:int SOAP-ENC:position="8988">1</SOAP-
ENC:int>
    </SOAP-ENC:Array>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Come si può vedere, in questo caso, oltre alla trasmissione delle informazioni contenute all'interno dell'array, ogni singolo elemento trasmissibile, cioè dotato di un valore significativo, viene identificato dalla sua posizione all'interno dell'array stesso (`SOAP-ENC:position="???"`).
Il codice che si occuperà di deserializzare questo payload creerà, nello stack di chiamate del server, un array completo vuoto ed inserirà nelle posizioni dell'array indicate i valori trasmessi.

Il seguente esempio di serializzazione, definisce un array contenente tipi di dato diversi come interi, reali, stringhe, ecc. (`SOAP-ENC:arrayType="xsd:anyType[4]"`). Per ciascun elemento dell'array viene data, attraverso l'istruzione `xsi:type="xsd:???"`, la definizione del tipo di dato contenuto.

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"
 xmlns:SOAP-ENC="http://www.w3.org/2001/12/soap-encoding"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <SOAP-ENC:Array
       SOAP-ENC:arrayType="xsd:anyType[4]">
      <thing xsi:type="xsd:int">1</thing>
      <thing xsi:type="xsd:float">1.45</thing>
      <thing xsi:type="xsd:string">una stringa</thing>
      <thing xsi:type="xsd:URI">http://site.com</thing>
    </SOAP-ENC:Array>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Il seguente esempio di serializzazione, definisce un array di strutture (di tipo `m:Order`).

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"
 xmlns:SOAP-ENC="http://www.w3.org/2001/12/soap-encoding">
  <SOAP-ENV:Body>
```

```
    <SOAP-ENC:Array
       SOAP-ENC:arrayType="m:Order[2]">
      <Order>
         <Product>Apple</Product>
         <Price>1.56</Price>
      </Order>
      <Order>
         <Product>Peach</Product>
         <Price>1.48</Price>
      </Order>
    </SOAP-ENC:Array>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Il seguente esempio di serializzazione, definisce un array multidimensionale di stringhe (`SOAP-ENC:arrayType="xsd:string[2,3]"`).

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"
 xmlns:SOAP-ENC="http://www.w3.org/2001/12/soap-encoding"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <SOAP-ENC:Array
       SOAP-ENC:arrayType="xsd:string[2,3]">
      <item>stringa_r1c1</item>
      <item>stringa_r1c2</item>
      <item>stringa_r1c3</item>
      <item>stringa_r2c1</item>
      <item>stringa_r2c2</item>
      <item>stringa_r2c3</item>
    </SOAP-ENC:Array>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Ovviamente combinando le situazioni precedenti è possibile ottenere esempi sempre più complessi.

Principali cambiamenti fra SOAP 1.1 e SOAP 1.2

È importante sottolineare alcune differenze presenti fra la versione 1.1 e la versione 1.2 dello standard SOAP rilasciate dal W3C; infatti la versione 1.2 è stata pubblicata solo recentemente (dicembre 2001), mentre finora le applicazioni reali implementano lo standard 1.1.

Segue un elenco dei cambiamenti principali:

- In SOAP 1.2 l'header HTTP Content-type deve contenere "application/soap", mentre in SOAP 1.1 contiene "text/xml".

- I namespaces delle due versioni sono ovviamente differenti.

- SOAP 1.2 definisce due nuovi valori per l'attributo actor: `none` ed `anonymous`.
- In SOAP 1.2 l'header HTTP SOAPAction non è più richiesto.
- SOAP 1.2 introduce l'header `Misunderstood` nel quale si specifica l'header block obbligatorio che non è stato capito/elaborato dal nodo destinatario.
- SOAP 1.2 introduce nuovi codici d'errore fra cui: `MustUnderstand`, `DTDNotSupported` e `DataEncodingUnknown`.
- SOAP 1.2 introduce nuovi codici d'errori riguardo ai namespace nelle chiamate RPC.
- SOAP 1.2 rinomina i `faultcode client` e `server` rispettivamente in `Seder` e `Reciever`.

Comunicazione fra nodi SOAP 1.1 e nodi SOAP 1.2

A questo punto analizziamo quali regole definisce SOAP per riconoscere a quale specifica SOAP appartiene un certo messaggio. Questo risulta molto importante quando un nodo SOAP faccia riferimento, ad esempio, alla specifica 1.1 ed un altro nodo alla specifica 1.2.
Nei messaggi SOAP non compare esplicitamente un numero che indichi una più o una meno recente versione della specifica SOAP. Per riconoscere la specifica a cui si riferisce un messaggio SOAP, si fa riferimento al namespace associato all'elemento Envelope.
La regola generale (per la specifica SOAP 1.1) dice che un nodo ricevente un messaggio SOAP, il cui namespace associato all'elemento Envelope sia diverso da quello atteso (`http://schemas.xmlsoap.org/soap/envelope/`), deve rifiutare tale messaggio e generare un errore di tipo `VersionMismatch`.
Le regole per la comunicazione fra implementazioni di SOAP 1.1 e SOAP 1.2 sono le seguenti:

- A causa delle regole di SOAP 1.1, un nodo SOAP 1.1 che riceve un messaggio SOAP versione 1.2, genera un errore di tipo `VersionMismatch`.
- Un nodo SOAP 1.2 che riceve un messaggio SOAP versione 1.1 può gestire il messaggio oppure generare un errore di tipo `VersionMismatch`.
  Come parte del messaggio d'errore di tipo `VersionMismatch`, il nodo SOAP 1.2 può includere una lista di versioni Envelope che supporta. Per fare s
  Questo si utilizza un particolare header block: `Upgrade` (che fa riferimento al namespace `http://www.w3.org/2001/12/soap-upgrade`).
  A sua volta l'header block `Upgrade` può contenere uno o più sottoelementi qualificati che, nel proprio attributo `qname`, specificano una versione Envelope supportata dal nodo.

```
<?xml version="1.0" ?>
<env:Envelope
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
 <env:Header>
  <V:Upgrade xmlns:V="http://www.w3.org/2001/12/soap-
upgrade">
   <envelope qname="ns1:Envelope"
             xmlns:ns1="http://www.example.org/2002/10/soap-
envelope"/>
```

```
   <envelope qname="ns2:Envelope"
             xmlns:ns2="http://www.w3.org/2001/12/soap-
envelope"/>
  </V:Upgrade>
 </env:Header>
 <env:Body>
   <env:Fault>
    <faultcode>env:VersionMismatch</faultcode>
    <faultstring>Version Mismatch</faultstring>
   </env:Fault>
 </env:Body>
</env:Envelope>
```

Figura 12:Messaggio d'errore VersionMismatch con estensione Upgrade multipla

Capitolo 5- HTTP e SOAP

HTTP

La sicurezza

Uno dei punti di forza di SOAP è la possibilità di utilizzare HTTP anche attravers la rete Internet, cosa che, invece, crea notevoli problemi se si utilizzano protocolli più complessi e delicati, come IIOP (di CORBA) oppure ORPC (di DCOM).

I motivi principali sono tendenzialmente due:

- IIOP e ORCP trasmettono payload informato binario
- Tentano di accedere ai server utilizzando porte TCP/IP non standard

La gestione dei dati in formato binario, benché sia a volte più efficiente dal punto di vista prestazionale in quanto il protocollo applicativo gestisce al suo interno una rappresentazione dei dati ottimale per lo scopo specifico dell'applicazione, in realtà ha l'effetto collaterale di essere spesso rifiutata dai firewall in quanto potenzialmente dannosa.
Se si pretende di utilizzare una chiamata IIOP o ORPC dall'interno di una rete chiusa, attraverso la rete Internet, per giungere ad un server che i trova all'interno di un'altra rete chiusa, i firewall che si frappongono tra le Web-Farm e Internet con buona probabilità rifiuteranno la richiesta di passaggio dei dati per motivi di sicurezza, in primo luogo perché si tratta di informazioni binarie, e poi perché le porte TCP/IP attraverso le quali si tenta d'accedere al server di solito non sono aperte.
Dal punto di vista della sicurezza, infatti, aprire una porta al passaggio di informazioni binarie, può essere molto pericoloso, in ambito aziendale è difficile trovare amministratori di rete che permettano tale tipo di comunicazione.
L'utilizzo di porte diverse da quelle standard, inoltre, è soggetto allo stesso tipo di ragionamento. Un firewall sarà, tendenzialmente, configurato per non permettere comunicazioni su porte TCP/IP non standard.
L'utilizzo di HTTP ha, in questo caso, innumerevoli vantaggi, primo fra tutti il fatto che l'enorme diffusione gli garantisce la pressoché totale distribuzione su qualsiasi rete basata su TCP/IP. Lo strato HTTP, infatti, comunica direttamente con quelli TCP/IP utilizzando una delle porte standard, tipicamente la 80.
Inoltre la comunicazione HTTP è formata totalmente da un payload XML, cioè da semplice testo. In quanto protocollo leggibile, HTTP, semplifica l'analisi visiva del traffico per gli utenti e consente ai firewall di raccogliere rapidamente i contenuti rilevanti delle richieste in entrata e di occuparsi del monitoraggio del traffico.
Quindi messaggi HTTP possono passare attraverso i firewall con molti meno problemi.
Altro vantaggio dell'uso di HTTP è che la sua estensione HTTPS, che permette i margini di sicurezza dati dal protocollo SSL3, è supportata in maniera nativa.

La comunicazione

La comunicazione HTTP si articola in una serie di passi predeterminati:

- Il client stabilisce una connessione TCP

- Il client spedisce un messaggio HTTP request e rimane in attesa
- Il server risponde con un messaggio HTTP response
- La connessione TCP viene chiusa

Per quanto detto si vede chiaramente lo schema request/response del protocollo di trasporto.

Il metodo GET

Il metodo GET è di gran lunga il metodo HTTP più utilizzato in quanto è lo standard di comunicazione in ambito web.
Ogni volta che un utente effettua un click su un link di una pagina web, il browser nella maggior parte dei casi, esegue una richiesta HTTP con il metodo GET.
Il metodo GET ha la caratteristica di poter effettuare richieste testuali particolari passando eventuali parametri sulla QueryString.
Ad esempio una richiesta che utilizzi il metodo GET verso una pagina web può essere la seguente:

http://www.hello.com/mypage.html?par1=val&par2=val2

Questa richiesta GET recupera dal server www.hello.com la pagina mypage.html e le passa come parametri le coppie par1=val1 e par2=val2.
Il metodo GET ha alcuni limiti:

- La QueryString non può superare i 1024 caratteri di lunghezza.
- Tutti i parametri vengono passati in chiaro, cioè sono visibili in quanto fanno parte della URL e vengono quindi tracciati e memorizzati dai proxy e dai firewall.

Come si può capire questi sono limiti che impediscono l'utilizzo in SOAP del metodo GET, infatti, dal punto di vista applicativo non è infrequente avere payload che superanoi 1024 byte ed inoltre non è accettabile che tutti i parametri vengano passati in chiaro per ovvi motivi di sicurezza e privacy. Si pensi, ad esempio, al passaggio di una password per l'autenticazione si un servizio. È necessario pertanto utilizzare metodi differenti.

Il metodo POST

Il metodo POST,m al contrario del metodo GET, trasmette il payload all'interno del corpo del messaggio HTTP e non sulla QueryString.
Questo, per quanto detto, si traduce in notevoli vantaggi in termini di flessibilità e sicurezza.
Altro vantaggio del metodo POST è che non è necessario convertire i caratteri speciali (come lo spazio) in codifiche esadecimali particolari, ma i dati vengono inseriti nel payload così come sono, in particolare per SOAP vengono serializzati in XML. Questo aumenta notevolmente la leggibilità del testo del messaggio e di conseguenza la sua deserializzazione da parte del servizio remoto che dovrà utilizzare le informazioni memorizzate all'interno del payload.
Vediamo un esempio di richiesta attraverso il metodo POST:

```
POST /helloServer HTTP/1.1
Host: www.hello.com:8080
Content-type: text/*; charset="utf-8"
Content-Length: 34
SOAPAction: "http://www.hello.com/helloServer#sayHello"
……
```

- La prima riga, chiamata *campo del metodo*, è costituita da tre campi di testo delimitati da uno spazio. Il primo campo fornisce il metodo di richiesta, il secondo identifica la risorsa alla quale il client è interessato e il terzo specifica la versione HTTP supportata dal client.

- Il campo Content-typet fornisce un tipo MIME che specifica quali dati un client può accettare, nell'esempio accetta qualunque contenuto testuale.
- Il campo Host contiene il nome (URL) del server richiesto seguito da un numero di porta opzionale (se non specificato si sottintende la porta 80).
- Il campo Content-Length indica in byte la lunghezza del pacchetto inviato.
- A questo punto seguono eventuali *header personalizzati* come SOAPAction.
  Ciascun header personalizzato viene trasmesso in coda agli header propri del protocollo di trasporto e di conseguenza viene ricevuto ed elaborato dal server.
  LA SOAPAction (obbligatoria per SOAP 1.1 e facoltativa per SOAP 1.2) viene spesso utilizzata come chiave d'accesso al firewall. All'interno del sistema di sicurezza, infatti, è possibile autorizzare o inibire l'accesso al traffico TCP/IP in funzione degli headers che la request porta con sé.
  Il contenuto dell'header SOAPAction nell'esempio sopra, ci dice che l'obiettivo è raggiungere il metodo sayHello dell'oggetto helloServer che si trova sul server www.hello.com. Il valore di tale header, dal punto di vista del formato, deve essere una URI; ma non è necessario che tali URI sia raggiungibile o risolvibile con un indirizzo reale, il suo scopo è solo quello di identificare un'azione che rappresenta l'obiettivo della request.
  Ritornando al discorso sui firewall, se si vuole ottenere l'autorizzazione per transitare al proprio interno, è necessario conoscere i target che sono permessi.
  In altre parole solo le richieste HTTP contenenti una SOAPAction tra quelle predeterminate saranno effettivamente eseguite sul server, le altre saranno rifiutate.
  Questo è molto importante in quanto l'amministratore del firewall sarà in grado di discriminare il traffico in entrata nella sua rete semplicemente analizzando gli header della richiesta HTTP, senza la necessità di deserializzare il payload XML per verificarne il contenuto, cosa molto più costosa in termini prestazionale.
- Segue il corpo del messaggio che nel nostro caso sarà un messaggio SOAP.

Una volta ricevuta la richiesta HTTP, il server tenterà di elaborarla e di rispondere in maniera appropriata.
Nel caso non vi siano errori la risposta HTTP sarà un messaggio di questo tipo:

```
HTTP/1.1 200 OK
Content-type: text/*; charset="utf-8"
Content-Length: 154
……
```

Dove la prima riga (il codice 200) indica che non ci sono stati errori, quindi la response (dopo gli header Content-type e Content-Lengt) conterrà esattamente quello che il client sia spetta di ricevere.
Nel caso di errori nella richiesta o nell'elaborazione, il server HTTP ritornerà un codice d'errore specifico.
Seguono alcuni esempi di codici d'errore:

- 400 Bad request Content-Length: 0
  che identifica una richiesta che il server non è in grado di soddisfare;
- 401.2 Unauthorized Content-Length: 0
  che identifica la mancanza di autorizzazioni per la richiesta specifica;
- 307 Temporaly Moved Content-Length: 0
  che identifica lo spostamento temporaneo della risorsa richiesta.

HTTP e SOAP

Come già affermato in precedenza, i messaggi SOAP possono essere scambiati usando una varietà di protocolli di trasporto. SOAP non fornisce nessun meccanismo per compensare le diverse caratteristiche di differenti protocolli. Qualunque altra caratteristica richiesta dall'applicazione, e non presente nel protocollo scelto, può essere fornita con un estensione di SOAP (ad esempio con un header block).

Allo stato attuale, solo l'uso di HTTP come protocollo di trasporto è standardizzato. Tuttavia sono possibili anche altre scelte, ad esempio il trasporto via e-mail.

HTTP è un ben noto modello di connessione e scambio messaggi. Il cliente identifica il server attraverso un URI, si connette ad esso attraverso la sottostante rete TCP/IP, rilascia un messaggio HTTP di richiesta e riceve un messaggio HTTP di risposta sulla stessa connessione. Il protocollo HTTP implicitamente correla messaggi di richiesta e risposta, adattandosi molto bene alle RPC.

```
POST[5] /Charging[6] HTTP/1.1[7]
Host: travelcompany.example.org
Content-Type: application/soap;[8] charset="utf-8"
Content-Length: nnnn

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-
envelope" >
 <env:Header>
   ::::::
 </env:Header>
 <env:Body>
    <m:reserveAndCharge>
        ::::::
    </m:reserveAndCharge>
  </env:Body>
</env:Envelope>
```

Figura 13:Richiesta RPC attraverso un messaggio HTTP

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

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

[5] Metodo HTTP
[6] richiesta URI, identifica la risorsa dell'host
[7] versione protocollo
[8] settaggio obbligatorio (per la versione SOAP 1.2)

```
</env:Envelope>
```

Figura 14:Risposta RPC corretta attraverso un messaggio HTTP

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-
envelope">
  <env:Body>
    <env:Fault>
      <faultcode>env:Receiver</faultcode>
      <faultstring>Processing Error</faultstring>
      <detail>
        <e:myfaultdetails
xmlns:e="http://travelcompany.example.org/faults" >
          <message>Name does not match card number</message>
          <errorcode>999</errorcode>
        </e:myfaultdetails>
      </detail>
    </env:Fault>
 </env:Body>
</env:Envelope>
```

Figura 15:Messaggio HTTP contenente un errore

Capitolo 6 - Specifica WSDL 1.1

Introduzione

Dopo la standardizzazione dei protocolli di comunicazione e del formato dei messaggi nella web community, è diventato man mano importante e possibile poter descrivere i servizi di rete in un modo strutturato. WSLD (Web Services Description Language) ha raccolto questa necessità definendo una grammatica, basata su XML, per descrivere tali servizi di rete come una collezione di endpoints in grado di scambiarsi messaggi.
Le operazioni ed i messaggi coinvolti sono definiti in modo astratto, il loro legame con il reale protocollo di comunicazione ed il formato dei dati definisce un endpoint.
La descrizione di servizi di rete attraverso WSDL fornisce una documentazione per sistemi distribuiti che permette, alle applicazioni coinvolte nella comunicazione, di ricavare automaticamente i dettagli relativi ai servizi scambiati.

Un documento WSDL definisce i servizi (`service`) come una collezione di network endpoints (`port`). In WSDL, la definizione astratta delle operazioni e dei messaggi è separata dall'effettivo protocollo di comunicazione e dal formato dei dati utilizzati. Questo permette il riutilizzo delle definizioni astratte: `messages` (che sono la descrizione astratta dei dati scambiati) e `port type` (che sono una collezione astratta di operazioni [`operation`]).
L'effettivo protocollo ed il formato dei dati scambiati, per un certo `port type`, costituiscono un `binding` riutilizzabile.
Un `port` è definito associando un indirizzo di rete ad un `binding`, ed una collezione di `port` definisce un `service`.
Quindi, un documento WSDL utilizza i seguenti elementi per descrivere i servizi di rete:

- `types`
  Contiene, sfruttando una qualche grammatica (ad esempio XSD), le definizioni dei tipi di dato non predefiniti usati dai servizi (come array e strutture).

- `message`
  Contiene la descrizione astratta dei dati scambiati nei messaggi di richiesta e risposta ai servizi.

- `portType`
  Contiene una collezione astratta di operazioni supportate da uno o più endpoints. Nella maggior parte di casi, ogni operazione consiste in una coppia di messaggi di richiesta/risposta, corrispondenti al nome di uno dei messaggi (`message`) definiti in precedenza.

- `binding`
  Contiene la descrizione sull'effettivo protocollo e formato dei dati utilizzato da un `portType` definito in precedenza. Per gli scopi di questa tesi il protocollo usato sarà SOAP. Nel caso in cui un'operazione fosse implementata in più protocolli, dovrà essere definito un `binding` per ciascuno di essi.

- `port`
  Contiene la descrizione di un singolo endpoint come combinazione di un `binding` e di un indirizzo di rete.

- `service`
  Contiene una collezione di endpoints (`port`).

È importante osservare che WSDL non introduce un nuovo linguaggio di definizione, infatti supporta ed adotta normalmente le specifiche di XML Schema (XSD).
Comunque essendo improbabile che un singolo tipo di grammatica sia in grado di descrivere tutti i possibili formati dei messaggi presenti e futuri, WSDL permette, attraverso la sua estensibilità, di usare altri linguaggi di definizione.

Inoltre WSDL definisce un comune meccanismo di binding. Questo meccanismo è utilizzato per collegare un specifico protocollo o un formato per i dati ad una definizione astratta (come message, operation, oppure un endpoint). Questo, come già detto, permette il riutilizzo delle definizioni astratte.
Le grammatiche più comuni,per il meccanismo di binding, riguardano: SOAP 1.1, MIME ed HTTP.

Definizione di un servizio di rete

Struttura di un documento WSDL

Un documento WSDL è una semplice collezione di definizioni, che rispettano la seguente grammatica:

```
<wsdl:definitions name="nmtoken"? targetNamespace="uri"?>

    <import namespace="uri" location="uri"/>*

    <wsdl:documentation .... /> ?

    <wsdl:types> ?
        <wsdl:documentation .... />?
        <xsd:schema .... />*
        <-- extensibility element --> *
    </wsdl:types>

    <wsdl:message name="nmtoken"> *
        <wsdl:documentation .... />?
        <part name="nmtoken" element="qname"?
type="qname"?/> *
    </wsdl:message>

    <wsdl:portType name="nmtoken">*
        <wsdl:documentation .... />?
        <wsdl:operation name="nmtoken">*
           <wsdl:documentation .... /> ?
           <wsdl:input name="nmtoken"? message="qname">?
               <wsdl:documentation .... /> ?
           </wsdl:input>
           <wsdl:output name="nmtoken"? message="qname">?
               <wsdl:documentation .... /> ?
           </wsdl:output>
```

```
            <wsdl:fault name="nmtoken" message="qname"> *
                <wsdl:documentation .... /> ?
            </wsdl:fault>
        </wsdl:operation>
    </wsdl:portType>

    <wsdl:binding name="nmtoken" type="qname">*
        <wsdl:documentation .... />?
        <-- extensibility element --> *
        <wsdl:operation name="nmtoken">*
            <wsdl:documentation .... /> ?
            <-- extensibility element --> *
            <wsdl:input> ?
                <wsdl:documentation .... /> ?
                <-- extensibility element -->
            </wsdl:input>
            <wsdl:output> ?
                <wsdl:documentation .... /> ?
                <-- extensibility element --> *
            </wsdl:output>
            <wsdl:fault name="nmtoken"> *
                <wsdl:documentation .... /> ?
                <-- extensibility element --> *
            </wsdl:fault>
        </wsdl:operation>
    </wsdl:binding>

    <wsdl:service name="nmtoken"> *
        <wsdl:documentation .... />?
        <wsdl:port name="nmtoken" binding="qname"> *
            <wsdl:documentation .... /> ?
            <-- extensibility element -->
        </wsdl:port>
        <-- extensibility element -->
    </wsdl:service>

    <-- extensibility element --> *

</wsdl:definitions>
```

Come già anticipato, i servizi sono descritto utilizzando sei elementi principali (`types`, `message`, `portType`, `binding`, `port`, `service`) dettagliati qui di seguito.

Types

Gli elementi `types` racchiudono le definizioni di tipi di dati non predefiniti che sono pertinenti ai messaggi scambiati. Per la massima interoperabilità e neutralità della piattaforma, WSDL preferisce l'utilizzo di XSD come tipo di sistema canonico.

```
    <definitions .... >
        <types>
            <xsd:schema .... />*
        </types>
    </definitions>
```

Un esempio per l'elemento di tipo `types` è riportata nel seguente paragrafo messages.

Messages

Gli elementi di tipo `message` consistono in una o più parti logiche (`part`). Ogni parte è associata ad un tipo, a partire da un sistema di base, utilizzando attributi message-typing. Il set degli attributi message-typing è estendibile.

WSDL definisce due attributi message-typing per l'utilizzo con il sistema XSD:

- **`element`**, riferisce ad un elemento XSD utilizzando un QName.

- **`type`**, riferisce ad un tipo di dato XSD semplice oppure ad un tipo di dato complesso utilizzando un QName.

La sintassi per definire un elemento di tipo message è la seguente (in neretto sono indicati gli attributi message-typing):

```
<definitions .... >
    <message name="nmtoken"> *
        <part name="nmtoken" element="qname"?
type="qname"?/> *
    </message>
</definitions>
```

L'attributo `name` , del `message`, definisce un nome univoco valido per tutti gli elementi di tipo `message` del documento WSDL.
L'attributo `name` , del `part` nel `message`, definisce un nome univoco valido per tutte le `part` all'interno del `message` in esame.

L'elemento `part` è un meccanismo flessibile per descrivere il contenuto logico di un messaggio astratto. Per esempio, se definiamo un `message` per utilizzarlo con RPC, un elemento `part` può rappresentare un parametro nel messaggio.

Nel caso di messaggi con più unità logiche si utilizzano più elementi `part`. Per esempio consideriamo il seguente messaggio che trasporta i dati di tipo Purchase Order e Invoice:

```
<definitions .... >
    <types>
        <schema .... >
           <element name="PO" type="tns:POType"/>
           <complexType name="POType">
               <all>
                   <element name="id" type="string/>
                   <element name="name" type="string"/>
                   <element name="items">
                       <complexType>
                           <all>
                               <element name="item"
type="tns:Item" minOccurs="0" maxOccurs="unbounded"/>
                           </all>
                       </complexType>
                   </element>
              </all>
           </complexType>

           <complexType name="Item">
```

```
                <all>
                    <element name="quantity" type="int"/>
                    <element name="product"
type="string"/>
                </all>
            </complexType>
            <element name="Invoice"
type="tns:InvoiceType"/>
            <complexType name="InvoiceType">
                <all>
                    <element name="id" type="string"/>
                </all>
            </complexType>
        </schema>
    </types>

    <message name="PO">
        <part name="po" element="tns:PO"/>
        <part name="invoice" element="tns:Invoice"/>
    </message>
</definitions>
```

Esiste una sintassi alternativa per esprimere gli stessi concetti, utilizzando un solo elemento `part`:

```
<definitions .... >
    <types>
        <schema .... >
            <complexType name="POType">
                <all>
                    <element name="id" type="string/>
                    <element name="name" type="string"/>
                    <element name="items">
                        <complexType>
                            <all>
                                <element name="item"
type="tns:Item" minOccurs="0" maxOccurs="unbounded"/>
                            </all>
                        </complexType>
                    </element>
                </all>
            </complexType>

            <complexType name="Item">
                <all>
                    <element name="quantity" type="int"/>
                    <element name="product"
type="string"/>
                </all>
            </complexType>
            <complexType name="InvoiceType">
                <all>
                    <element name="id" type="string"/>
                </all>
            </complexType>
```

```
            <complexType name="Composite">
                <choice>
                    <element name="PO" minOccurs="1"
maxOccurs="1" type="tns:POType"/>
                    <element name="Invoice" minOccurs="0"
maxOccurs="unbounded" type="tns:InvoiceType"/>
                </choice>
            </complexType>
        </schema>
    </types>

    <message name="PO">
        <part name="composite" type="tns:Composite"/>
    </message>
</definitions>
```

Port Types

Un elemento di tipo `portType` corrisponde ad un insieme di operazioni collegate ai rispettivi `message` definiti in precedenza.

```
<wsdl:definitions .... >
    <wsdl:portType name="nmtoken">
        <wsdl:operation name="nmtoken" .... /> *
    </wsdl:portType>
</wsdl:definitions>
```

L'attributo `name`, del `portType`, definisce un nome univoco valido per tutti gli elementi di tipo `portType` del documento.
Una operazione è identificata attraverso l'attributo `name` associato.

WSDL definisce quattro tipi di trasmissioni primitive che un endpoint può supportare:

- **One-way**. L'endpoint riceve un messaggio.
- **Request-response**. L'endpoint riceve un messaggio, ed invia un messaggio di risposta correlato.
- **Solicit-response**. L'endpoint invia un messaggio, e riceve un messaggio di risposta correlato.
- **Notification**. L'endpoint invia un messaggio.

Le operazioni in WSDL possono riferire a questi tipi primitivi per indicare il tipo di trasmissione che necessitano.
Sebbene le trasmissioni di tipo request-response e solicit-response possono essere modellate in astratto utilizzando due messaggi di tipo one-way, è conveniente modellare queste come operazioni primitive per i seguenti motivi:

- Sono tipi di trasmissioni molto comuni.
- Una sequenza può essere correlata senza aver introdotto flussi complessi d'informazioni.
- Qualche endpoints può solo ricevere messaggi se questi sono il risultato di una request-response sincrona.
- Un semplice flusso può essere, a livello di algoritmi, derivato da queste primitive nel momento in cui la definizione del flusso lo richieda.

Il documento WSDL definisce logicamente il tipo di trasmissione, è compito del meccanismo di binding correlarle al protocollo di trasporto. Sebbene la struttura WSDL di base supporta logicamente questi quattro tipi di trasmissione, il meccanismo di binding, attualmente, è definito solo per i modelli one-way e request-response.

Operazioni One-way

La grammatica per un'operazione di tipo one-way è la seguente:

```
<wsdl:definitions .... > <wsdl:portType .... > *
        <wsdl:operation name="nmtoken">
           <wsdl:input name="nmtoken"? message="qname"/>
        </wsdl:operation>
    </wsdl:portType >
</wsdl:definitions>
```

L'elemento `input` specifica quale dei messaggi astratti (`message`) definiti in precedenza è utilizzato come formato per l'operazione di tipo one-way.

Operazioni Request-response

La grammatica per un'operazione di tipo request-response è la seguente:

```
<wsdl:definitions .... >
    <wsdl:portType .... > *
        <wsdl:operation name="nmtoken"
parameterOrder="nmtokens">
            <wsdl:input name="nmtoken"? message="qname"/>
            <wsdl:output name="nmtoken"?
message="qname"/>
            <wsdl:fault name="nmtoken" message="qname"/>*
        </wsdl:operation>
    </wsdl:portType >
</wsdl:definitions>
```

Gli elementi `input` ed `output` specificano il formato rispettivamente del messaggio di richiesta e del messaggio di risposta fra quelli (`message`) definiti in precedenza.
L'elemento `fault` è opzionale, e specifica quale dei messaggi astratti (`message`) definiti in precedenza è utilizzato nel caso in cui l'output dell'operazione generi un errore.

Si noti che l'operazione definisce in modo astratto (logico) il modello di trasmissione adoperato; sarà compito del particolare binding determinare come il messaggio viene effettivamente inviato: all'interno di una singola comunicazione (come una trasmissione HTTP request/response), oppure come due indipendenti comunicazioni (come due trasmissioni HTTP di request).

Operazioni Solicit-response

La grammatica per un'operazione di tipo solicit-response è la seguente:

```
<wsdl:definitions .... >
    <wsdl:portType .... > *
        <wsdl:operation name="nmtoken"
parameterOrder="nmtokens">
            <wsdl:output name="nmtoken"?
message="qname"/>
```

```
            <wsdl:input name="nmtoken"? message="qname"/>
            <wsdl:fault name="nmtoken" message="qname"/>*
        </wsdl:operation>
    </wsdl:portType >
</wsdl:definitions>
```

Gli elementi `output` ed `input` specificano il formato rispettivamente del messaggio di richiesta e del messaggio di risposta fra quelli (`message`) definiti in precedenza.
L'elemento `fault` è opzionale, e specifica quale dei messaggi astratti (`message`) definiti in precedenza è utilizzato nel caso in cui l'output dell'operazione generi un errore.

Si noti che l'operazione definisce in modo astratto (logico) il modello di trasmissione adoperato; sarà compito del particolare binding determinare come il messaggio viene effettivamente inviato: all'interno di una singola comunicazione (come una trasmissione HTTP request/response), oppure come due indipendenti comunicazioni (come due trasmissioni HTTP di request).

Operazioni Notification

La grammatica per un'operazione di tipo notification è la seguente:

```
<wsdl:definitions .... >
    <wsdl:portType .... > *
        <wsdl:operation name="nmtoken">
            <wsdl:output name="nmtoken"?
message="qname"/>
        </wsdl:operation>
    </wsdl:portType >
</wsdl:definitions>
```

L'elemento `output` specifica quale dei messaggi astratti (`message`) definiti in precedenza è utilizzato come formato per l'operazione di tipo notification.

Nome degli elementi all'interno di Operation

L'attributo `name` degli elementi `input` ed `output` definisce un nome univoco per tutti gli elementi di tipo `input` ed `output` presenti all'interno dell'elemento `portType` in esame.

Per evitare di definire il `name` di ogni elemento di tipo `input` ed `output`, WSDL fornisce dei valori di default basati sull'attributo `name` di `operation`.
Se non viene specificato il nome in una operazione di tipo one-way o notification, di default assume il nome dell'operazione.
Se non viene specificato il nome in una operazione di tipo request-response o solicit-respons, di default il nome dell'elemento `input` ed `output` diventa il nome dell'operazione con rispettivamente i suffissi "Request"/"Solicit" e "Response".

Ordine dei parametri all'interno di Operation

L'elemento operation non specifica se l'operazione verrà o non utilizzata in un binding di tipo RPC. Comunque, quando utilizziamo un operazione con binding RPC, è necessario rispettare la struttura originale della procedura chiamata.
Per questa ragione, nelle operazioni di tipo request-response o solicit-response è necessario specificare la lista dei parametri all'interno dell'attributo `parameterOrder`.

Il valore di questo attributo è una lista di parti di messaggi astratti, fra quelli definiti in precedenza (`part` di `message`), separati fra loro da uno spazio.
Il valore dell'attributo `parameterOrder` deve rispettare le seguenti regole:

- L'ordine dei `part` deve riflettere l'ordine dei parametri della procedura chiamata.
- Il `part` che rappresenta il valore di ritorno non è presente nella lista.
- Se un `part`, attraverso il corrispondente `message`, appare sia in un elemento di tipo `input` che di tipo `output`, allora è un parametro di in/out.
- Se un `part`, attraverso il corrispondente `message`, appare un elemento di tipo `input`, allora è un parametro di in.
- Se un `part`, attraverso il corrispondente `message`, appare un elemento di tipo `output`, allora è un parametro di out.

Si noti che questa informazione serve come riferimento e può essere tranquillamente ignorata quando non riguarda un binding RPC.

Bindings

Un elemento di tipo `binding` definisce il formato del messaggio e dettagli del protocollo per le operazioni ed i messaggi astratti definiti in un certo `portType`.
È possibile avere più `binding` per un certo `portType`.
La grammatica per un elemento `binding` è la seguente:

```
<wsdl:definitions .... >
    <wsdl:binding name="nmtoken" type="qname"> *
        <-- extensibility element (1) --> *
        <wsdl:operation name="nmtoken"> *
           <-- extensibility element (2) --> *
           <wsdl:input name="nmtoken"? > ?
               <-- extensibility element (3) -->
           </wsdl:input>
           <wsdl:output name="nmtoken"? > ?
               <-- extensibility element (4) --> *
           </wsdl:output>
           <wsdl:fault name="nmtoken"> *
               <-- extensibility element (5) --> *
           </wsdl:fault>
        </wsdl:operation>
    </wsdl:binding>
</wsdl:definitions>
```

L'attributo `name`, del `binding`, definisce un nome univoco per tutti gli elementi di tipo `binding` presenti nel documento WSDL.

Il `binding` riferisce il `portType` in esame attraverso l'attributo `type`.

L'estensibilità del `binding` è utilizzata per specificare l'effettiva grammatica per i messaggi di `input` (3), d'`output` (4) e di `fault` (5). È prevista l'estensibilità anche a livello di `operation` del `binding` (2) e per il `binding` stesso (1). In pratica la grammatica degli elementi estensibili utilizzati dipendano protocollo scelto (es. SOAP 1.1, oppure MIME, oppure HTTP, ecc.).

Un elemento `operation` del `binding` specifica le informazioni, necessarie al meccanismo di binding, per l'`operation` che ha lo stesso `name` all'interno del

portType in esame. Siccome il name di tale operation non è necessariamente univoco nel portType (per esempio, in caso di overloading del nome di un metodo), l'attributo name nell'elemento operation del binding potrebbe essere insufficiente per localizzare univocamente l'operazione stessa. In tal caso, l'operazione può essere identificata correttamente fornendo l'attributo name in corrispondenza degli elementi wsdl:input e wsdl:output del binding.

Un binding deve specificare esattamente un solo protocollo.

Un binding non deve specificare informazioni sull'indirizzo.

Ports

Un elemento di tipo port definisce un singolo endpoint specificando un singolo indirizzo per un certo binding.

```
<wsdl:definitions .... >
    <wsdl:service .... > *
        <wsdl:port name="nmtoken" binding="qname"> *
           <-- extensibility element (1) -->
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

L'attributo name specifica un nome univoco per tutti gli elementi di tipo port presenti nel documento WSDL.

L'attributo binding (di tipo QName) riferisce all'elemento di tipo binding a cui associare l'indirizzo.

La specifica sulle informazioni dell'indirizzo avviene tramite l'elemento estendibile (1), la sua grammatica dipende del protocollo scelto (es. SOAP 1.1, oppure MIME, oppure HTTP, ecc.).

Un port non deve specificare più di un indirizzo.

Un port non deve specificare informazioni sul meccanismo di binding al di fuori dell'indirizzo.

Services

Un elemento di tipo service raggruppa fra loro alcuni port:

```
<wsdl:definitions .... >
    <wsdl:service name="nmtoken"> *
        <wsdl:port .... />*
    </wsdl:service>
</wsdl:definitions>
```

L'attributo name definisce un nome univoco per tutti gli elementi d tipo service all'interno del documento WSDL.

Gli elementi port riferiti all'interno di un elemento service hanno la seguente relazione:

- Nessun `port` comunica con un altro `port` (ad esempio l'output di un `port` non può essere l'input di un altro `port`).

- Se un `service` riferisce più `port` che condividono lo stesso `portType`, ma con un differente indirizzo o `binding`, allora i `port` sono alternativi.
  Ogni `port` segue lo stesso comportamento semantico (con le limitazioni imposta dal proprio `binding` sul trasporto e sul formato del messaggio). Questo permette, all'utilizzatore del documento WSDL, di scegliere un particolare `port` per comunicare in base ad un qualche criterio (protocollo, distanza, ecc.).

Binding per SOAP

WSDL supporta il (meccanismo di) binding con endpoint di tipo SOAP 1.1, attraverso le seguenti informazioni:

- Un'indicazione specifica che il binding si riferisce al protocollo SOAP 1.1.

- Un modo per specificare l'indirizzo di un endpoint SOAP.

- L'URI per l'header SOAPAction, quando si utilizza HTTP come protocollo di trasporto per SOAP.

- Una lista di definizioni per gli Header che sono trasmessi come parte dell'Envelope SOAP.

Esempi SOAP

Negli esempi a seguire si fa riferimento al binding SOAP relativo ad operazioni RPC request-response utilizzando il protocollo di trasporto HTTP.
Vediamo un primo esempio di documento WSDL contenete il binding per SOAP:

```
<?xml version="1.0"?>
<definitions name="StockQuote"

targetNamespace="http://example.com/stockquote.wsdl"
          xmlns:tns="http://example.com/stockquote.wsdl"

xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
          xmlns:xsd1="http://example.com/stockquote.xsd"

xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/">

    <message name="GetTradePriceInput">
        <part name="tickerSymbol" element="xsd:string"/>
        <part name="time" element="xsd:timeInstant"/>
    </message>

    <message name="GetTradePriceOutput">
        <part name="result" type="xsd:float"/>
    </message>

    <portType name="StockQuotePortType">
        <operation name="GetTradePrice">
```

```
            <input message="tns:GetTradePriceInput"/>
            <output message="tns:GetTradePriceOutput"/>
        </operation>
    </portType>

    <binding name="StockQuoteSoapBinding"
type="tns:StockQuotePortType">
        <soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="GetTradePrice">
            <soap:operation
soapAction="http://example.com/GetTradePrice"/>
            <input>
                <soap:body use="encoded"
namespace="http://example.com/stockquote"

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
"/>
            </input>
            <output>
                <soap:body use="encoded"
namespace="http://example.com/stockquote"

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
"/>
            </output>
        </operation>>
    </binding>

    <service name="StockQuoteService">
        <documentation>My first service</documentation>
        <port name="StockQuotePort"
binding="tns:StockQuoteBinding">
            <soap:address
location="http://example.com/stockquote"/>
        </port>
    </service>
</definitions>
```

Questo esempio descrive una richiesta SOAP 1.1 (richiesta RPC al metodo GetTradePrices) inviata al servizio StockQuote utilizzando HTTP come protocollo di trasporto. Il servizio fornisce la quotazione, per un istante di tempo specificato, del titolo azionario specificato. Per fare ciò il messaggio di richiesta invia una stringa `tickerSymbol` contente il nome del titolo ed un istante di tempo `time` (di tipo timeIstant). Il messaggio di risposta contiene un valore di ritorno `result` di tipo float che rappresenta la quotazione del titolo.

Per evidenziare alcuni concetti, già spiegati, sulla grammatica WSDL possiamo complicare l'esempio precedente in questo modo:

```
<?xml version="1.0"?>
<definitions name="StockQuote"

targetNamespace="http://example.com/stockquote.wsdl"
          xmlns:tns="http://example.com/stockquote.wsdl"
```

```
xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"

xmlns:xsd1="http://example.com/stockquote/schema"

xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"

xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/
"
          xmlns="http://schemas.xmlsoap.org/wsdl/">

    <types>
       <schema
targetNamespace="http://example.com/stockquote/schema"

xmlns="http://www.w3.org/2000/10/XMLSchema">
           <complexType name="TimePeriod">
              <all>
                  <element name="startTime"
type="xsd:timeInstant"/>
                  <element name="endTime"
type="xsd:timeInstant"/>
              </all>
           </complexType>
           <complexType name="ArrayOfFloat">
              <complexContent>
                  <restriction base="soapenc:Array">
                      <attribute ref="soapenc:arrayType"
wsdl:arrayType="xsd:float[]"/>
                  </restriction>
              </complexContent>
           </complexType>
       </schema>
    </types>

    <message name="GetTradePricesInput">
        <part name="tickerSymbol" element="xsd:string"/>
        <part name="timePeriod"
element="xsd1:TimePeriod"/>
    </message>

    <message name="GetTradePricesOutput">
        <part name="result" type="xsd1:ArrayOfFloat"/>
        <part name="frequency" type="xsd:float"/>
    </message>

    <portType name="StockQuotePortType">
        <operation name="GetLastTradePrice"
parameterOrder="tickerSymbol timePeriod frequency">
           <input message="tns:GetTradePricesInput"/>
           <output message="tns:GetTradePricesOutput"/>
        </operation>
    </portType>

    <binding name="StockQuoteSoapBinding"
type="tns:StockQuotePortType">
```

```
        <soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="GetTradePrices">
           <soap:operation
soapAction="http://example.com/GetTradePrices"/>
           <input>
               <soap:body use="encoded"
namespace="http://example.com/stockquote"

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
"/>
           </input>
           <output>
               <soap:body use="encoded"
namespace="http://example.com/stockquote"

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
"/>
           </output>
        </operation>>
    </binding>

    <service name="StockQuoteService">
        <documentation>My first service</documentation>
        <port name="StockQuotePort"
binding="tns:StockQuoteBinding">
           <soap:address
location="http://example.com/stockquote"/>
        </port>
    </service>
</definitions>
```

Questo esempio descrive una richiesta SOAP 1.1 (richiesta RPC al metodo GetLastTradePrices) inviata al servizio StockQuote utilizzando HTTP come protocollo di trasporto. Il servizio fornisce una serie di quotazione, comprese fra un istante di tempo iniziale ed un istante di tempo finale specificati, del titolo azionario specificato. Per fare ciò il messaggio di richiesta invia una stringa contente il nome del titolo ed una struttura di tipo `Timeperiod` contenente i due istanti di tempo. Il messaggio di risposta contiene un valore di ritorno di tipo `ArrayOfFloat` che rappresenta la lista delle quotazione per il titolo, ed un valore di tipo float che rappresenta il numero di quotazioni presenti nella lista.
Si noti che, attraverso l'attributo `parameterOrder`, si comprende che:

- `tickerSymbol` e `timePeriod` sono parametri d'ingresso;
- `frequency` è un parametro d'uscita;
- `result` è il valore di ritorno del metodo (infatti non si trova nella lista dell'attributo `parameterOrder`).

Inoltre le strutture `Timeperiod` ed `ArrayOfFloat`, non essendo di tipo predefinito, necessitano della loro dichiarazione all'interno dell'elemento di tipo `types`.

Grammatica del Binding per SOAP

Il (meccanismo di) binding per SOAP estende WSDL con i seguenti elementi estensibili:

```
<definitions .... >
    <binding .... >
        <soap:binding style="rpc|document"
transport="uri">
        <operation .... >
           <soap:operation soapAction="uri"?
style="rpc|document"?>?
           <input>
               <soap:body parts="nmtokens"?
use="literal|encoded"
                          encodingStyle="uri-list"?
namespace="uri"?>
               <soap:header message="qname"
part="nmtoken" use="literal|encoded"
                           encodingStyle="uri-list"?
namespace="uri"?>*
                 <soap:headerfault message="qname"
part="nmtoken" use="literal|encoded"
                                    encodingStyle="uri-
list"? namespace="uri"?/>*
               <soap:header>
           </input>
           <output>
               <soap:body parts="nmtokens"?
use="literal|encoded"
                          encodingStyle="uri-list"?
namespace="uri"?>
               <soap:header message="qname"
part="nmtoken" use="literal|encoded"
                           encodingStyle="uri-list"?
namespace="uri"?>*
                 <soap:headerfault message="qname"
part="nmtoken" use="literal|encoded"
                                    encodingStyle="uri-
list"? namespace="uri"?/>*
               <soap:header>
           </output>
           <fault>*
               <soap:fault name="nmtoken"
use="literal|encoded"
                           encodingStyle="uri-list"?
namespace="uri"?>
            </fault>
        </operation>
    </binding>

    <port .... >
        <soap:address location="uri"/>
    </port>
</definitions>
```

Ogni elemento estensibile è analizzato nei seguenti paragrafi.

soap:binding

Lo scopo del elemento `soap:binding` è quello di segnalare che il binding si riferisce al protocollo SOAP. Quest'elemento non ha nessuna pretesa di definire la codifica o il formato del messaggio, questo compito è rimandato alla specifica SOAP vera e propria.

L'elemento `soap:binding` deve essere presente quando eseguiamo il binding per SOAP.

```
<definitions .... >
    <binding .... >
        <soap:binding transport="uri"?
style="rpc|document"?>
    </binding>
</definitions>
```

Il valore dell'attributo `style` viene adoperato quando l'attributo `style` dell'elemento `soap:operation` (si veda il paragrafo successivo) è omesso. Se l'attributo `style` dell'elemento `soap:binding` non è presente, di default si assume abbia valore `document`.

Il valore dell'attributo `transport` indica quale protocollo di trasporto dati viene associato a SOAP. Ad esempio se il valore di `uri` è http://schemas.xmlsoap.org/soap/http allora si adotta HTPP come protocollo di trasporto. Altri valori di `uri` possono essere presenti per utilizzate SMTP, FTP, ed altri al posto di HTTP.

soap:operation

L'elemento soap:operation fornisce informazioni sulle intere informazioni.

```
<definitions .... >
    <binding .... >
        <operation .... >
           <soap:operation soapAction="uri"?
style="rpc|document"?>?
        </operation>
    </binding>
</definitions>
```

L'attributo `style` indica se l'operazione è RPC-oriented (messaggi che contengono parametri e valori di ritorno) oppure document-oriented (messaggi contenenti documenti). Quest'informazione può essere utilizzata per selezionare un appropriato modello di programmazione. Inoltre, tale informazione, influisce anche sul modo in cui il Body del messaggio SOAP è costruito (si veda il paragrafo soap:body).
Se l'attributo non è specificato, di default il suo valore è quello presente nell'attributo `style` dell'elemento `soap:binding`.

L'attributo `soapAction` specifica il valore dell'header HTTP SOAPAction per l'operazione in esame. Il valore di `uri` deve essere un indirizzo URI, il quale comparirà direttamente nell'header SOAPAction.
Utilizzando HTTP con SOAP il valore di tale attributo deve essere obbligatoriamente specificato. Utilizzando protocolli diversi da HTTP, questo attributo non è obbligatori,ed anche l'elemento `soap:operation` può essere omesso.

soap:body

L'elemento `soap:body` specifica come le parti (`part`) del messaggio (`message`) appaiono all'interno del Body SOAP.

Le parti del messaggio possono essere definizioni di tipo astratto, oppure schemi di definizione concreti.

L'elemento `soap:body` fornisce informazioni su come assemblare le differenti parti del messaggio nel Body SOAP. L'elemento `soap:body` è utilizzato sia in messaggi RPC-oriented che nei messaggi document-oriented, ma lo stile (`style`) delle operazioni contenute ha importanti conseguenze su come la sezione Body è strutturata:

- Se l'operazione ha `style rpc`, ogni parte è un parametro oppure un valore di ritorno ed appare come elemento wrapper all'interno del Body. Il nome del wrapper è identico al nome dell'operazione, ed il namespace coincide con il valore del attributo `namespace`. Ogni parte del messaggio (parametro) compare sotto il wrapper, rappresentato da una entry con lo stesso nome del corrispondente parametro di chiamata. Le parti compaiono nello stesso ordine dei parametri di chiamata.

- Se l'operazione ha `style document` non ci sono wrapper addizionali, e le parti del messaggio compaiono direttamente sotto il Body SOAP.

```
<definitions .... >
    <binding .... >
        <operation .... >
           <input>
               <soap:body parts="nmtokens"?
use="literal|encoded"?
                          encodingStyle="uri-list"?
namespace="uri"?>
           </input>
           <output>
               <soap:body parts="nmtokens"?
use="literal|encoded"?
                          encodingStyle="uri-list"?
namespace="uri"?>
           </output>
        </operation>
    </binding>
</definitions>
```

L'attributo opzionale `parts` indica quale parte compare all'interno di una qualche porzione del Body SOAP. Se questo attributo è omesso, allora tutte le parti (`part`) definite dal messaggio (`message`) compaiono all'interno della porzione Body SOAP.

L'attributo `use` è richiesto se le parti del messaggio sono codificate utilizzando una qualche regola di codifica, oppure se le parti definiscono un concreto schema del messaggio.

Se il valore di `use` è `encoded`, allora ogni parte del messaggio riferisce ad un tipo astratto utilizzando l'attributo `type`. Questi tipi astratti sono utilizzati per produrre un concreto messaggio applicando una codifica specificata dall'attributo `encodingStyle`.

Il name, il type delle parti ed il valore dell'attributo namespace sono tutte informazioni in ingresso alla codifica. Se lo stile di codifica referenziato ammette delle variazioni, allora tutte queste variazioni sono supportate.

Se il valore di use è literal, allora ogni parte del messaggio riferisce ad uno schema concreto utilizzando l'attributo element o type. Nel primo caso, l'elemento riferito dalle parti comparirà direttamente nel Body (per un messaggio document-orienter) oppure in un sottoelemento (per un messaggio RPC-orienter).
Nel secondo caso, il tipo riferito dalle parti diventa lo schema tipo per gli elementi racchiusi (nel Body per i messaggi document-orienter oppure nei sottoelementi per i messaggi RPC-orienter). Il valore dell'attributo encodingStyle, in questo caso, indica che il formato concreto deriva da questa codifica; ma solo le variazioni specificate sono supportate.

Il valore dell'attributo encodingStyle è una lista di indirizzi URI, separati fra loro da singoli spazi. Tali URI specificano le codifiche utilizzate all'interno del messaggio, ordinate dalle più restrittive alle meno restrittive.

soap:fault

L'elemento soap:fault specifica il contenuto di un sottoelemento details dell'elemento Fault in SOAP.

```
<definitions .... >
    <binding .... >
        <operation .... >
           <fault>*
               <soap:fault name="nmtoken"
use="literal|encoded"
                                        encodingStyle="uri-
list"? namespace="uri"?>
           </fault>
        </operation>
    </binding>
</definitions>
```

L'attributo name riferisce l'elemento soap:fault all'elemento wsdl:fault dell'operazione.

Il messaggio d'errore deve avere una singola parte.
Gli attributi use, encodingStyle e namespace sono utilizzati nello stesso modo visto per l'elemento soap:body, e style='document' dichiara che il messaggio d'errore non contiene parametri.

soap:header e soap:headerfault

Gli elementi soap:header e soap:headerfault permettono di definire ciò che è trasmesso all'interno degli Header SOAP.

```
<definitions .... >
    <binding .... >
        <operation .... >
           <input>
             <soap:header message="qname" part="nmtoken"
use="literal|encoded"
```

```
                                encodingStyle="uri-list"?
namespace="uri"?>*
                <soap:headerfault message="qname"
part="nmtoken" use="literal|encoded"
                                        encodingStyle="uri-
list"? namespace="uri"?/>*
              <soap:header>
            </input>
            <output>
                <soap:header message="qname"
part="nmtoken" use="literal|encoded"
                                encodingStyle="uri-list"?
namespace="uri"?>*
                  <soap:headerfault message="qname"
part="nmtoken" use="literal|encoded"
                                          encodingStyle="uri-
list"? namespace="uri"?/>*
                <soap:header>
            </output>
        </operation>
    </binding>
</definitions>
```

Gli attributi `use`, `encodingStyle` e `namespace` sono utilizzati nello stesso modo visto per l'elemento `soap:body`, e `style='document'` dichiara che l'header non contiene parametri.

L'attributo `message` e `part` riferiscono alla parte del messaggio che definisce il tipo di header. Lo schema di riferimento può includere definizioni per gli attributi soap:actor e soap:MustUnderstand se `use='literal'`, mentre non può se `use='encoded'`.

L'elemento opzionale `headerfault` (che appare all'interno di `soap:header`) ha la stessa sintassi del `soap:header`, e permette di specificare il tipo di header utilizzato nel caso si verifichino degli errori per il `soap:header` in esame. Infatti la specifica SOAP dice che un errore pertinente ad un Header deve essere specificato in un Header a sua volta.

soap:address

L'elemento `soap:address` è utilizzato per dare ad un `port` un indirizzo (un URI). Nell'eseguire il binding con SOAP un `port` deve riferire esattamente ad un indirizzo.
La tipologia dell'URI deve rispettare lo schema definito nell'attributo `transport` dell'elemento `soap:binding`.

```
<definitions .... >
    <port .... >
        <binding .... >
           <soap:address location="uri"/>
        </binding>
    </port>
</definitions>
```

Capitolo 7- L'ambiente Apache SOAP 2.2

L'ambiente Apache SOAP 2.2

Introduzione

La Apache Software Foundation sta sviluppando da tempo un progetto ambizioso, si tratta di xml.apache.org, con l'obiettivo di costruire un supporto XML che sia robusto, completo, qualitativamente evoluto e gratuito, alla più ampia quantità possibile di esigenze implementative.
In questo contesto si inserisce Apache SOAP.
La versione 2.2 , basata su Java, è conforme alla versione 1.1 della specifica SOAP, anche se con alcune limitazioni ed è open source.
Apache SOAP 2.2 è un package Java che può essere installato sia come client side, per invocare servizi SOAP che si trovino altrove, sia come server side per pubblicare servizi SOAP.
Un'installazione client consiste nell'arricchire un ambiente di sviluppo Java con il package soap.jar, il cuore dell'implementazione.
Un'installazione server side, invece, consiste nel rendere disponibile ad un application server (ad esempio Apache Tomcat), la componentistica giusta per permettergli di rispondere a chiamate SOAP, si tratta ancora del package soap.jar.
Attuali limitazioni:

- l'attributo encodingStyle può avere solo l'encoding style predefinito;
- non supporta l'attributo mustUnderstand;
- non supporta l'attributo root di XML;
- non supporta l'attributo actor (quindi non supporta nodi SOAP intermediari);
- non utilizza attributi multi-ref durante la serializzazione (quindi non utilizza l'attributo XML href, che contiene il riferimento ad un altro elemento XML successivo nel messaggio).

Cliente RPC

I passi basilari per creare un cliente, il quale interagisca con un servizio SOAP RPC-based, sono i seguenti:

- Ottenere la descrizione dell'interfaccia del servizio SOAP, così da conoscere la struttura del metodo che si desidera invocare. In genere tale descrizione è fornita dal file WSDL associato al servizio.
- Accertarsi che la serializzazione in XML dei parametri inviati e la deserializzazione da XML dei parametri ricevuti in risposta sia registrata/supportata. Apache SOAP fornisce un numero predefinito di serializzatori/deserializzatori
  Per trasmettere o ricevere un tipo di dato non registrato è necessario registrarsi un propio serializzatore/deserializzatore.
- Creare un nuovo oggetto `Call` (org.apache.soap,rpc.RPCMessage.Call object). Questo è l'oggetto chiave per una chiamata SOAP, infatti i suoi metodi vengono utilizzati per settare le caratteristiche della comunicazione.

- Settare, all'interno dell'oggetto `Call`, l'URI[9] del servizio web da utilizzare attraverso il metodo `setTargetObjectURI`.
- Settare il nome del metodo che si desidera invocare all'interno dell'oggetto `Call` attraverso il metodo `setMethodName`.
- Creare gli oggetti `Parameter` necessari per la chiamata RPC e settarli all'interno dell'oggetto `Call` attraverso il metodo `setParams`. Ovviamente i parametri devono essere in numero, ordine e tipo coincidenti con quelli richiesti dal servizio. Inoltre il serializzatore/deserializzatore per gli oggetti da trasmettere/ricevere deve essere registrato.
- Eseguire sull'oggetto `Call` il metodo `invoke` e catturare, con un oggetto `Response`, l'oggetto ritornato dal metodo `invoke`. Il metodo `invoke` prevede due parametri: il primo è l'URL che identifica l'endpoint sul quale il servizio risiede, il secondo è il valore che sarà posizionato nell'header SOAPAction (presente in un messaggio HTTP).
- Verificare se l'oggetto `Response` contiene un errore attraverso il metodo `generatedFault`. Tale metodo ritorna un valore `false` se non vi sono stati errori; `true` altrimenti. In caso d'errore attraverso il metodo `getFault`, che ritorna un oggetto `Fault`, è possibile indagare sul tipo d'errore.
- Nel caso non vi siano errori, è possibile estrarre il risultato (i parametri) di ritorno dall'oggetto `Response` attraverso il metodo `getReturnValue` (`getParams`).

Server RPC

La pubblicazione di un servizio RPC si articola in due passi:

- Creare il codice che descrive il servizio attraverso un linguaggio supportato da Apache SOAP, ad esempio utilizzando Java. Tale sorgente Java non ha nulla di speciale che lo renda in qualche modo assimilabile ad un servizio web utilizzabile in remoto. La potenza di questa implementazione è anche questa, non dover utilizzare codice particolare per definire i servizi web, sarà il client a richiamare il sevizio utilizzando SOAP.
- La pubblicazione vera e propria richiede di creare un Apache SOAP deployment descriptor del servizio. Tale deployment descriptor fornisce tutte le informazioni necessarie per maneggiare una richiesta ed offrire il servizio; solitamente si crea un documento XML contenente le specifiche della pubblicazione . Ad esempio, per un implementazione Java, il file contiene principalmente il nome del servizio, il nome della classe che fornisce il servizio e il nome del metodo che fornisce il servizio.

  Per effettuare fisicamente il deploy del servizio si utilizza la seguente istruzione:

  ```
  java  org.apache.server.ServiceManagerClient
  http://localhost:8080/soap/servlet/rpcrouter
  deploy file_deployment_descriptor.xml.
  ```

  Si utilizza la classe Java org.apache.server.ServiceManagerClient che si occupa di registrare i servizi, ad essa si passa come parametri il nome della servlet da utilizzare per effettuare il deploy (cioè la rpcrouter), il comando deploy ed il documento XML contenente il deployment descriptor.

[9] URI = Uniform Resource Identifier, si tratta di una stringa, adeguatamente formattata, che identifica in maniera univoca una risorsa all'interno dello spazio informativo. L'insieme degli URI si divide in due parti distinte, gli URL (es. http://untimo.it) e gli URN (es. urn:nomeMetodo).

In questo modo si rimanda alle funzionalità specifiche dell'ambiente Apache SOAP il compito di accettare le richieste SOAP provenienti dai clienti, passare i parametri al metodo destinatario ed, infine, rispondere al cliente con un messaggio SOAP contenente il risultato dell'invocazione del metodo.
Quindi la gestione dei messaggi SOAP, con relative specifiche, è del tutto trasparente all'utente che ha scritto il codice del server.

Esempio HelloWorld2

L'obiettivo dell'applicazione HelloWorld2 è molto semplice: si tratta di un servizio web che esporta un metodo `sayHello` il quale prende in input un parametro di tipo stringa e restituisce la stringa "Hello xxx Welcome to SOAP", dove xxx rappresenta il valore del parametro che il metodo ha ricevuto.

Il server HelloWorld2

Il codice sorgente Java del server è il seguente:

```
package esempi.helloworld2;

import java.io.*;
import java.util.*;

public class HelloWorldServer2
{

  public String sayHello(String st)
   {
      return ("Hello  "+st+" ! Welcome to SOAP");
   }
}
```

Guardando questo file sorgente Java appare chiaro che non ha nulla di speciale, nulla che lo renda in qualche modo assimilabile ad un servizio web utilizzabile da remoto.

Una volta che il codice è stato scritto e compilato è necessario pubblicarlo.
Per fare ciò è necessario creare un file XML contenente le specifiche per la pubblicazione: il deployment descriptor
Nel caso in esame, il documento XML ha il seguente contenuto:

```
<isd:service xmlns:isd="http://xml.apache.org/xml-
soap/deployment"
             id="urn:HelloWorldServer2">
  <isd:provider type="java"
                scope="Request"
                methods="sayHello">
    <isd:java class="esempi.helloworld2.HelloWorldServer2"
static="false"/>
  </isd:provider>
</isd:service>
```

Il file in questione contiene tutte le informazioni che sono necessarie per la pubblicazione, in particolare si tratta delle seguenti:

- `id="urn:HelloWorldServer2"` → nome del servizio remoto.

- `scope="Request"` → tipo d'operazione eseguita dal server, attende una richiesta.
- `provider type="java"` → linguaggio di sviluppo del server.
- `methods="sayHello">` → nome del metodo pubblicato.
- `class="esempi.helloworld2.HelloWorldServer2"` → classe che contiene il metodo.
- `static="false"` → tipologia della classe.

Come si può vedere viene indicato con precisione che si tratta di un servizio remoto di nome `HelloWorldServer2` sviluppato in Java ed implementato nella classe `esempi.helloworld2.HelloWorldServer2` ed in particolare utilizza il metodo `sayHello`.

Per effettuare fisicamente il deploy del servizio dobbiamo eseguire, dal prompt comandi, la seguente istruzione:

```
java org.apache.soap.server.ServiceManagerClient
http://localhost:8080/soap/servlet/rpcrouter deploy
helloworld2_deploy.xml
```

Il Client HelloWorld2

Il codice sorgente Java del client è il seguente:

```
package esempi.helloworld2;

import java.net.*;
import java.util.*;

import org.apache.soap.util.xml.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;

public class HelloWorld2
{
   public final static void main(String args[])
      throws MalformedURLException, SOAPException
   {
      Call call = new Call();
      call.setTargetObjectURI("urn:HelloWorldServer2");
      call.setMethodName("sayHello");
      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

      Vector params = new Vector();
      params.addElement(new
Parameter("st",String.class,args[0],null));
      call.setParams(params);
      URL url =  new
URL("http://localhost:8081/soap/servlet/rpcrouter");

      Response resp = call.invoke(url,"");
```

```
      if(!resp.generatedFault())
      {
         Parameter ret = resp.getReturnValue();
         System.out.println(ret.getValue());
      }
      else
      {
         Fault fault = resp.getFault();
         System.err.println("------------------------------
-");
         System.err.println("Attenzione: Condizione di
Fault");
         System.err.println("Codice: "+
fault.getFaultCode());
         System.err.println("Descrizione: "+
fault.getFaultString());
         System.err.println("------------------------------
-");
       }
   }
}
```

Analizzando il codice del client SOAP, appare subito evidente l'utilizzo di due oggetti particolari: Call e Response.

- `Call call = new Call();`
  `Call` è l'oggetto chiave della chiamata e viene creato attraverso questo costruttore; seguono alcuni suoi metodi utilizzati per settare le caratteristiche della comunicazione.

- `call.setTargetObjectURI("urn:HelloWorldServer2");`
  Imposta il servizio web da utilizzare (deve coincidere con il valore assunto dall'attributo `id` nel deployment descriptor `helloworld2_deploy.xml`).

- `call.setMethodName("sayHello");`
  Imposta il metodo da utilizzare tra quelli messi a disposizione del server.

- `call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);`
  Imposta l'encodingStyle del payload XML, in realtà avrà sempre un valore costante predefinito.

- `Vector params = new Vector();`
  Attraverso questo costruttore si crea un oggetto di tipo `Vector` da utilizzare come veicolo per trasportare i parametri all'interno della request SOAP.

- `params.addElement(new Parameter("st",String.class,args[0],null));`
  I parametri necessari sono aggiunti attraverso il metodo `addElement`, al quale occorre dare in input un oggetto istanza della classe `Parameter`. In questo caso, l'oggetto stesso, è creato direttamente all'interno del metodo `addElement` utilizzando il costruttore di `Parameter`; il nome dell'oggetto è `st`, è di tipo stringa, assume il valore di `args[0]` ed il valore relativo all'ecodingStyleURI viene dichiarato `null`.

- `call.setParams(params);`
  Al termine della procedura di aggiunta dei parametri è necessario passare l'oggetto istanza della classe `Vector` al metodo `setParams` dell'oggetto `call`.

Fin ora abbiamo realizzato tutto il necessario per la chiamata al metodo *sayHello()* passandogli come parametro una stringa di testo.
Facendo un parallelo con una chiamata ad un metodo locale, siamo nella seguente situazione: *stringa = sayHello('parametro stringa')*.
A questo punto si ci deve occupare della chiamata vera e propria al metodo e della sua risposta.

- `Response resp = call.invoke(url,"");`
  All'oggetto di tipo `Response` viene restituito il valore della chiamata al metodo che è pubblicato sull'url indicato come primo parametro della `invoke`; il secondo parametro della `invoke` è il valore assunto dall'header SOAPAction della request SOAP, ed assume importanza nel caso si lavori attraverso firewall.
  L'oggetto `Response` contiene tutte le informazioni della response SOAP.
  Quello che ci si aspetta , in condizioni di normalità, è la presenza al suo interno del risultato dell'elaborazione. In tal caso il valore ritornato da `resp.generatedFault()` è false, altrimenti si può indagare sul codice ed il tipo d'errore.

- `Parameter ret = resp.getReturnValue();`
  In questo modo si ottengono i parametri di ritorno.

- `System.out.println(ret.getValue());`
  In questo modo si stampa la stringa di ritorno; infatti il metodo `getValue` ritorna un oggetto di tipo Object, in questo caso una stringa.

Messaggi SOAP di HelloWorld2

A questo punto valutiamo cosa accade a livello di call e response SOAP, ovvero cosa veramente viene passato come payload all'interno della chiamata HTTP.
Per fare questo si utilizza l'utility freeware tcpTrace che permette di analizzare il traffico tcp/ip tra un client ed un server e, volendo, anche di effettuarne un logging su file.
Per eseguire correttamente questa analisi il client comunica sulla porta 8081 mentre il server ascolta sulla porta 8080, come si è potuto notare dal codice riportato in precedenza.
In questo modo, l'utility di monitoraggio può effettuare il tunneling fra client e server in maniera trasparente alle relative implementazioni e l'analisi del traffico non subisce interferenze dalla stessa utility.

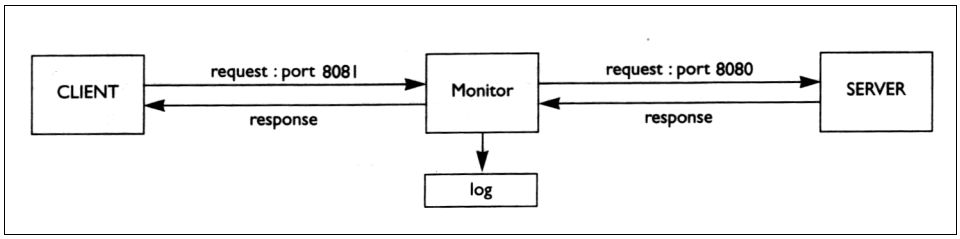


Figura 16:Funzionamento dell'utility tcTrace

Il messaggio SOAP di richiesta inviato dal cliente è il seguente:

```
POST /soap/servlet/rpcrouter HTTP/1.0
Host: localhost
```

```
Content-Type: text/xml; charset=utf-8
Content-Length: 449
SOAPAction: ""

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:sayHello xmlns:ns1="urn:HelloWorldServer2" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
">
<st xsi:type="xsd:string">Sandro</st>
</ns1:sayHello>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Come si può vedere si tratta di una chiamata HTTP che utilizza il metodo POST.
All'header è stato aggiunto la riga relativa a SOAPAction che però non contiene valori in quanto non è stata riempita dal codice dell'applicazione client.
L'host di destinazione è `localhost`, come indicato nel codice dell'applicazione client ed il contenuto della chiamata HTTP è un payload XML che segue le specifiche SOAP.
Si tratta cioè esattamente di quello che ci si aspettava.
Analizzando il payload XML possiamo notare che il servizio remoto di destinazione è la servlet `/soap/servlet/rpcrouter` ed in particolare il metodo `sayHello` che era stato mappato, a livello di deploy, sul servizio `urn:HelloWorldServer2` ed implementato dalla classe `esempi.helloworld2.HelloWorldServer2`.
Nel Body SOAP come elemento figlio di `<ns1:sayHello>`, è presente `<st xsi:type="xsd:string">Sandro</st>`.
Come è facile intuire, è proprio la presenza di questo elemento che serve per trasportare il parametro st="Sandro" di tipo string, poi elaborato dal metodo invocato.

Il messaggio SOAP di risposta inviato dal server è il seguente:

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 500
Set-Cookie2:
JSESSIONID=2fz4fr57k1;Version=1;Discard;Path="/soap"
Set-Cookie: JSESSIONID=2fz4fr57k1;Path=/soap
Set-Cookie2:
JSESSIONID=2fz4fr57k1;Version=1;Discard;Path="/soap"
Set-Cookie: JSESSIONID=2fz4fr57k1;Path=/soap
Servlet-Engine: Tomcat Web Server/3.2.3 (JSP 1.1; Servlet
2.2; Java 1.3.1; Windows 2000 5.1 x86; java.vendor=Sun
Microsystems Inc.)

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
```

```
<ns1:sayHelloResponse xmlns:ns1="urn:HelloWorldServer2"
SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
">
<return xsi:type="xsd:string">Hello  Sandro ! Welcome to
SOAP</return>
</ns1:sayHelloResponse>

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

Il server HTTP risponde con il codice 200 OK per informare che la comunicazione ha avuto successo e quindi restituisce alcune righe di header.
All'interno del payload XML, più precisamente nella sezione Body, è presente l'elemento `<ns1:sayHelloResponse>` che rappresenta il contenitore delle informazioni di risposta all'elemento `<ns1:sayHello>` presente nella richiesta.
Il valore di ritorno vero e proprio è costituito dall'elemento XML:

```
<return xsi:type="xsd:string">
Hello  Sandro ! Welcome to SOAP
</return>
```

questa response contiene esattamente quello che ci aspettiamo, cioè la presenza della stringa di ritorno e il valore del parametro che era stato passato in input al metodo remoto.

Fault di HelloWorld2

Per provare come Apache SOAP gestisce le condizioni di Fault, immaginiamo che lato server sia in grado di accetta le richieste dai client, ma il servizio HelloWorld2 non sia pubblicato e che il client tenti ugualmente di inviargli una richiesta.
Per la richiesta del client possiamo far riferimento esattamente a quanto visto nel precedente paragrafo nella request SOAP.
Invece la risposta del server in ascolto sarà la seguente:

```
HTTP/1.0 500 Internal Server Error
Content-Type: text/xml; charset=utf-8
Content-Length: 473
Set-Cookie2:
JSESSIONID=s4db2qgdn1;Version=1;Discard;Path="/soap"
Set-Cookie: JSESSIONID=s4db2qgdn1;Path=/soap
Set-Cookie2:
JSESSIONID=s4db2qgdn1;Version=1;Discard;Path="/soap"
Set-Cookie: JSESSIONID=s4db2qgdn1;Path=/soap
Servlet-Engine: Tomcat Web Server/3.2.3 (JSP 1.1; Servlet
2.2; Java 1.3.1; Windows 2000 5.1 x86; java.vendor=Sun
Microsystems Inc.)

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<SOAP-ENV:Fault>
<faultcode>SOAP-ENV:Server</faultcode>
<faultstring>service 'urn:HelloWorldServer2'
unknown</faultstring>
<faultactor>/soap/servlet/rpcrouter</faultactor>
```

```
</SOAP-ENV:Fault>

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

Il server HTTP risponde con il codice 500 Internal Sever Error, ed in accordo con le specifiche SOAP, nel payload XML è presente la sezione `<SOAP-ENV:Fault>` con tutti i dettagli sulla condizione d'errore.
In particolare:

- L'elemento `faultcode` con valore `SOAP-ENV:Server` ci informa che il server non può gestire il messaggio.
- L'elemento `faultstring` con valore `service 'urn:HelloWorldServer2' unknown` ci conferma che il server non conosce il sevizio richiesto.
- L'elemento `faultactor` con valore `/soap/servlet/rpcrouter` indica il processore che ha generato l'errore.

Osservando il codice sorgente del cliente ci si accorge che le informazioni sul faultcode e sul faultstring si possono ottenere, anche a livello di codice, applicando le funzioni `fault.getFaultCode()`e `fault.getFaultString()`all'oggetto `fault`. di tipo `Fault`. Infatti, attraverso la funzione `resp.getFault ()`, il valore dell'elemento `SOAP-ENV:Fault` contenuto nel body del messaggio di risposta è memorizzato nell'oggetto `fault`.

Esempio Adder-COM (web server Tomcat e client Apache SOAP)

Questo esempio rientra nello studio pratico sull'interoperabilità del protocollo SOAP.
L'esempio Adder-COM prevede un servizio dal lato server scritto in Visual Basic, ed un client scritto in Java.
Il client, attraverso una request SOAP, invia al servizio in ascolto due numeri interi; il server, invocando il metodo `add`, ne calcola la somma e spedisce il risultato al client attraverso una response SOAP.

Server Adder-COM

Il codice del server è il seguente:

```
Public Function add(ByVal n1 As Integer, ByVal n2 As
Integer) As Integer
add = n1 + n2
End Function
```

Ancora una volta il file sorgente Visual Basic del server non ha nulla di speciale, nulla che lo renda in qualche modo assimilabile ad un servizio web utilizzabile da remoto.
Si tratta a tutti gli effetti di un normalissimo componente COM.
Dopo aver scritto il codice è necessario generare il componente COM, quindi creare la libreria APACHEADDER.dll (infatti il progetto Visual Basic si chiama appunto APACHEADDER).
A questo punto occorre registrare il componente COM in modo che possa essere utilizzato da altre applicazioni, per fare ciò si digita `regsvr32 APACHEADDER` dal prompt.
Apache SOAP fornisce il supporto per pubblicare servizi riguardanti componenti COM.
Il deploy del servizio, al prompt comandi, si ottiene digitando:

```
java org.apache.soap.server.ServiceManagerClient
http://localhost:8080/soap/servlet/rpcrouter deploy
adder.xml
```

Rispetto all'esempio HelloWorld2, pur lavorando su un componente COM, la procedura per eseguire il deploy non cambia. Ciò che cambia, in parte, è il contenuto del file XML (`adder.xml`) che definisce il deployment descriptor:

```
<!--Apache SOAP specific deployment descriptor.-->
<isd:service xmlns:isd="http://xml.apache.org/xml-
soap/deployment"
             id="urn:adder-COM">
  <isd:provider
type="org.apache.soap.providers.com.RPCProvider"
                scope="Application"
                methods="add">
    <isd:java class="required not needed for COMProvider"/>
    <isd:option key="progid" value="Apacheadder.adder" />
  </isd:provider>

<isd:faultListener>org.apache.soap.server.DOMFaultListener</
isd:faultListener>
</isd:service>
```

Il file in questione contiene tutte le informazioni che sono necessarie per la pubblicazione, in particolare si tratta delle seguenti:

- `id="urn:adder-COM"` → nome del servizio remoto.
- `provider type="org.apache.soap.providers.com.RPCProvider"` → è il nome della classe Java che si occupa di fornire il supporto alla pubblicazione di componenti COM.
- `scope="Application"` → tipo d'operazione eseguita dal server.
- `methods="add">` → nome del metodo pubblicato.
- `class="required not needed for COMProvider"` → essendo un applicazione COM non ha senso parlare di classe Java.
- `option key="progid" value="Apacheadder.adder"` → il valore dell'attributo rappresenta il ProgId dell'oggetto COM pubblicato.

Client Adder-COM

Il codice del client è il seguente:

```
package samples.com.client;

import java.io.*;
import java.net.*;
import java.util.*;
```

```
import org.apache.soap.*;
import org.apache.soap.rpc.*;

public class Addit
{
 public static void main( String[] args) throws Exception
 {

   Integer n1=null;
   Integer n2=null;
   final String urn= "urn:adder-COM";

   if( args.length != 2 ) explain();

   try{
   n1= new Integer( args[0]);
   n2= new Integer( args[1]);
   }catch ( NumberFormatException e)
   {
     explain();
   }


   Vector params = new Vector ();
   params.addElement (new Parameter("n1" , Integer.class,
n1, null));
   params.addElement (new Parameter("n2" , Integer.class,
n2, null));

   URL url = new URL ("http://" + serverhost + ":" +
serverport+ soapservlet);

   // Build the call.
   Call call = new Call ();
   call.setTargetObjectURI (urn);
   call.setMethodName ("add");
   call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

   call.setParams (params);
   Response resp = call.invoke (/* router URL */ url, /*
actionURI */ "" );

   if (resp.generatedFault ()) {
     Fault fault = resp.getFault ();
     System.out.println ("Ouch, the call failed: ");
     System.out.println ("  Fault Code   = " +
fault.getFaultCode ());
     System.out.println ("  Fault String = " +
fault.getFaultString ());
   } else {
     Parameter result = resp.getReturnValue ();
     System.out.println("The sum of " + args[0] + " and " +
args[1] + " is " + result.getValue());
   }
 }
 private static void explain()
```

```
 {
   System.err.println("Please provide two integers as inputs
to add!");
   System.exit(8);
 }
 static String serverhost= "localhost";
 static String serverport= "8081";
 static String soapservlet= "/soap/servlet/rpcrouter";

} // end  addit
```

Le funzioni e gli oggetti utilizzati dal client di Add-COM sono già stati analizzati per il client HelloWorld2. Rispetto a quest'ultimo cambiano il numero ed il tipo di parametri, lavoriamo con due numeri interi.
Per quanto riguarda l'oggetto Call:

- `Call call = new Call();`
  `Call` è l'oggetto chiave della chiamata e viene creato attraverso questo costruttore; seguono alcuni suoi metodi utilizzati per settare le caratteristiche della comunicazione.

- `call.setTargetObjectURI(urn);`
  Imposta il nome del servizio web da contattare, in particolare `urn` contiene il valore `urn:adder-COM` (deve coincidere con il valore assunto dall'attributo `id` nel deployment descriptor `adder.xml`).

- `call.setMethodName("add");`
  Imposta il metodo da utilizzare tra quelli messi a disposizione del server.

- `call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);`
  Imposta l'EncodingStyle del payload XML, in realtà avrà sempre un valore costante predefinito.

- `Vector params = new Vector();`
  Attraverso questo costruttore si crea un oggetto `Vector` da utilizzare come veicolo per trasportare i parametri all'interno della request SOAP.

- `params.addElement (new Parameter("n1" , Integer.class, n1, null));`
  Il primo parametro, di nome `n1`, è un intero che rappresenta il primo addendo ed assume il valore di `args[0]`.

- `params.addElement (new Parameter("n2" , Integer.class, n2, null));`
  Il secondo parametro, di nome `n2`, è un intero che rappresenta il secondo addendo ed assume il valore di `args[1]`.

- `call.setParams(params);`
  Al termine della procedura di aggiunta dei parametri è necessario passare l'oggetto istanza della classe `Vector` al metodo `setParams` dell'oggetto `call`.

Fin ora abbiamo realizzato tutto il necessario per la chiamata al metodo *add()* passandogli come parametri i due addendi.

Facendo un parallelo con una chiamata ad un metodo locale, siamo nella seguente situazione: *somma = add(addendo1,addendo2)*.
A questo punto si ci deve occupare della chiamata vera e propria al metodo e della sua risposta.

- `Response resp = call.invoke (/* router URL */ url, /* actionURI */ "" );`
  In questo modo, si invoca il sevizio precedentemente pubblicato all'indirizzo `http://localhost:8081/soap/servlet/rpcrouter`, specificando `null` il valore assunto dall'header SOAPAction nella request SOAP (non abbiamo particolari esigenze con i firewall).
  L'oggetto `Response` contiene tutte le informazioni della response SOAP.
  Quello che ci si aspetta , in condizioni di normalità, è la presenza al suo interno del risultato dell'elaborazione. In tal caso il valore ritornato da `resp.generatedFault()` è false, altrimenti si può indagare sul codice ed il tipo d'errore.

- `Parameter ret = resp.getReturnValue();`
  In questo modo si ottengono i parametri di ritorno.

- `System.out.println(ret.getValue());`
  In questo modo si stampa il risultato della somma.

- `Fault fault = resp.getFault ();`
  Come detto, se `resp.generatedFault()` ritorna un valore true, allora nella comunicazione SOAP si è verificato un errore. In tal caso è possibile memorizzare in un oggetto di tipo `Fault`, attraverso la funzione `resp.getFault()`, il valore dell'elemento `SOAP-ENV:Fault` contenuto nel body del messaggio di risposta.

- `System.out.println (" Fault Code = " + fault.getFaultCode ());`
  Si visualizza il valore del `faultcode` (elemento figlio di `SOAP-ENV:Fault`), che specifica in modo univoco quale errore si è verificato.

- `System.out.println (" Fault String = " + fault.getFaultString ());`
  Si visualizza il valore della stringa `faultstring` (elemento figlio di `SOAP-ENV:Fault`), che esprime in parole sintetiche quale errore si è verificato.

Messaggi SOAP di Adder-COM

Per valutare cosa accade a livello di call e response si utilizza ancora l'utility tpcTrace, con l'accortezza di inviare la call sulla porta 8081 e mantenere il server in ascolto sulla porta 8080 così da permettere il tunneling.

Il messaggio di richiesta inviato dal client Java è il seguente:

```
POST /soap/servlet/rpcrouter HTTP/1.0
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: 454
SOAPAction: ""
```

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:add xmlns:ns1="urn:adder-COM" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
">
<n1 xsi:type="xsd:int">2</n1>
<n2 xsi:type="xsd:int">4</n2>
</ns1:add>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

La richiesta HTTP contiene un payload XML che rispetta le specifiche SOAP.
Infatti il body SOAP contiene una singola struct con il nome del metodo invocato e il relativo namespace che identifica il servizio contenente tale metodo.
Gli elementi figli della struct `add` (`<n1 xsi:type="xsd:int">2</n1>` e `<n2 xsi:type="xsd:int">4</n2>`) rappresentano i due parametri del metodo remoto `add` e contengono i due addendi interi da sommare.

Il messaggio di risposta inviato dal server COM è il seguente:

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 451
Set-Cookie2:
JSESSIONID=tosifo6821;Version=1;Discard;Path="/soap"
Set-Cookie: JSESSIONID=tosifo6821;Path=/soap
Set-Cookie2:
JSESSIONID=tosifo6821;Version=1;Discard;Path="/soap"
Set-Cookie: JSESSIONID=tosifo6821;Path=/soap
Servlet-Engine: Tomcat Web Server/3.2.3 (JSP 1.1; Servlet
2.2; Java 1.3.1; Windows 2000 5.0 x86; java.vendor=Sun
Microsystems Inc.)

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:addResponse xmlns:ns1="urn:adder-COM" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
">
<return xsi:type="xsd:short">6</return>
</ns1:addResponse>

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

Il server HTTP risponde con il codice 200 OK per informare che la comunicazione ha avuto successo e quindi restituisce alcune righe di header.
La risposta HTTP contiene un payload XML che rispetta le specifiche SOAP.

Infatti il body SOAP contiene una singola struct con il nome del metodo invocato concatenato al suffisso `Response`.
L'elemento figlio della struct (`<return xsi:type="xsd:short">6</return>`) rappresenta il valore di ritorno del metodo, quindi contiene la somma dei due numeri interi inviati con la richiesta SOAP.

Esempio Exchange

Anche questo esempio rientra nello studio pratico sull'interoperabilità del protocollo SOAP.
Questa volta il servizio web è realizzato e pubblicato su piattaforma GLUE Electric, mentre il client è sviluppato su piattaforma Apache SOAP.

Server Exchange

Il servizio accetta in input due parametri (di tipo String), che rappresentano i nomi di due paesi, e fornisce come output un valore (di tipo float) che rappresenta l'attuale tasso di cambio fra le valute dei due paesi.
Il sevizio è pubblicato all'indirizzo http://services.xmethods.com:80/soap dove ogni venti minuti vengono aggiornati i tassi di cambio in base all'andamento dei mercati finanziari.
La descrizione sulla piattaforma di sviluppo, sul profilo di chiamata/risposta RPC e il relativo file WSDL si possono trovare all'indirizzo http://services.xmethods.net. Inoltre, a questo indirizzo, è possibile visionare la descrizione di altri servizi pubblicati utilizzando differenti piattaforme.

Il contenuto del documento WSDL è il seguente:

```
<?xml version="1.0" ?>
<definitions name="CurrencyExchangeService"
targetNamespace="http://www.xmethods.net/sd/CurrencyExchange
Service.wsdl"
xmlns:tns="http://www.xmethods.net/sd/CurrencyExchangeServic
e.wsdl" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
<message name="getRateRequest">
  <part name="country1" type="xsd:string" />
  <part name="country2" type="xsd:string" />
</message>
<message name="getRateResponse">
  <part name="Result" type="xsd:float" />
  </message>
<portType name="CurrencyExchangePortType">
<operation name="getRate">
  <input message="tns:getRateRequest" />
  <output message="tns:getRateResponse" />
</operation>
</portType>
<binding name="CurrencyExchangeBinding"
type="tns:CurrencyExchangePortType">
  <soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http" />
 <operation name="getRate">
  <soap:operation soapAction="" />
```

```
  <input>
   <soap:body use="encoded" namespace="urn:xmethods-
CurrencyExchange"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
  </input>
  <output>
   <soap:body use="encoded" namespace="urn:xmethods-
CurrencyExchange"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
  </output>
 </operation>
</binding>
<service name="CurrencyExchangeService">
  <documentation>Returns the exchange rate between the two
currencies</documentation>
  <port name="CurrencyExchangePort"
binding="tns:CurrencyExchangeBinding">
   <soap:address
location="http://services.xmethods.net:80/soap" />
  </port>
</service>
</definitions>
```

Client Exchange

Come già detto, il client è stato sviluppato con Apache SOAP, quindi le operazioni e le funzioni eseguite sono già state visionate negli esempi precedenti.

Il codice del client è il seguente :

```
import java.io.*;
import java.net.*;
import java.util.*;
import org.apache.soap.util.xml.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;

public class CurrencyClient{

  public static float getRate (URL url, String country1,
String country2) throws Exception {

    Call call = new Call ();

    // Set encoding style. Use the standard SOAP encoding
    String encodingStyleURI = Constants.NS_URI_SOAP_ENC;
    call.setEncodingStyleURI(encodingStyleURI);

    // Set service locator parameters
    call.setTargetObjectURI ("urn:xmethods-
CurrencyExchange");
    call.setMethodName ("getRate");

    // Create the input parameter vector
    Vector params = new Vector ();
```

```
    params.addElement (new Parameter("country1",
String.class, country1, null));
    params.addElement (new Parameter("country2",
String.class, country2, null));
    call.setParams (params);

    // Invoke the service ...
    Response resp = call.invoke (url,"");

    // ... and evaluate the result
    if (resp.generatedFault ()) {
   throw new Exception();
    } else {

      // Call was succesfull. Extract response parameter and
return
      Parameter result = resp.getReturnValue ();
      Float rate=(Float) result.getValue();
      return rate.floatValue();
    }
  }

// Driver to illustrate invocation of service
public static void main(String[] args)
  {
    try {
          URL url=new
URL("http://services.xmethods.com:80/soap");
          String country1= "us";
          String country2= "taiwan";
          float rate = getRate(url,country1,country2);
          System.out.println(rate);
    }
    catch (Exception e) {e.printStackTrace();}
  }
}
```

Messaggi SOAP

Il messaggio di richiesta inviato dal client Apache SOAP è il seguente:

```
POST /soap HTTP/1.0
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: 514
SOAPAction: ""

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getRate xmlns:ns1="urn:xmethods-CurrencyExchange" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
">
```

```
<country1 xsi:type="xsd:string">us</country1>
<country2 xsi:type="xsd:string">taiwan</country2>
</ns1:getRate>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Con questa call si richiede il tasso di cambio attuale fra la valuta USA e la valuta di Taiwan. Il metodo remoto a cui ci si riferisce è `getRate`, presente nel servizio di nome `urn:xmethods-CurrencyExchange`.
A tale metodo si inviano due parametri stringa rappresentati dagli elementi (figli della struct `getRate`) `<country1 xsi:type="xsd:string">us</country1>` e `<country2 xsi:type="xsd:string">taiwan</country2>`.
L'indirizzo di destinazione è effettivamente http://services.xmethods.com:80/soap, anche se nella richiesta HTTP compare `localhost`. Ancora una volta, per visualizzare i messaggi di richiesta e risposta SOAP, si è utilizzato l'utility tcpTrace, quindi si è eseguito il tunneling fra il client locale e il server remoto.

Il messaggio di risposta inviato dal server GLUE Electric è il seguente:

```
HTTP/1.0 200 OK
Date: Sun, 14 Apr 2002 15:38:30 GMT
Content-Type: text/xml
Server: Electric/1.0
Content-Length: 490

<?xml version='1.0' encoding='UTF-8'?>
<soap:Envelope
xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
xmlns:xsi='http://www.w3.org/1999/XMLSchema-instance'
xmlns:xsd='http://www.w3.org/1999/XMLSchema'
xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding
/'><soap:Body><n:getRateResponse xmlns:n='urn:xmethods-
CurrencyExchange'><Result
xsi:type='xsd:float'>34.9</Result></n:getRateResponse></soap
:Body></soap:Envelope>
```

Il valore di ritorno è contenuto nella struct `getRateResponse` dall'elemento figlio `<Result xsi:type='xsd:float'>34.9</Result>`.

Capitolo 8- L'ambiente Microsoft SOAP Toolkit 2.0

Caratteristiche generali

Microsoft SOAP Toolkit 2.0 consiste in:

- Un componente client-side che permette ad un'applicazione di invocare un servizio web descritto da un documento WSDL (Web Services Description Language).
- Un componente server-side che utilizza i documenti WSDL e WSML (Web Service Meta Language) residenti sul server per permettere il mapping fra il servizio web pubblicato e gli oggetti COM corrispondenti.
- Componenti per deserializzare, trasmettere e riserializzare i messaggi SOAP.

In aggiunta, SOAP Toolkit 2.0 fornisce un tool che semplifica lo sviluppo delle applicazioni:

- Il tool WSDL/WSML Generetor aiuta l'utente, attraverso una wizard, a generare i file WSDL e WSML necessari.

Le caratteristiche supportate da SOAP Toolkit 2.0 sono le seguenti:

- Specifiche del consorzio W3C relative a WSDL 1.1 (Marzo 2001).
- Specifiche del consorzio W3C relative a SOAP 1.1 (Maggio 2000).
- Specifiche del consorzio W3C relative a XML Schema Part 0 (Primer) ,Part 1 (Structure) e Part (Datatypes).
- Array di tipo semplice e complesso.
- Array multidimensionali.
- Type Complex.
- Supporto per operazioni WSDL RPC-encoded e document-literal.
- SOAP Headers.

Il toolkit permette di eseguire chiamate a servizi e pubblicare questi attraverso due livelli di API (Application Program Interface): "high" oppure "low". La scelta dipenderà dalle caratteristiche dei messaggi SOAP che si desidera inviare o dal livello di monitoraggio desiderato. Ovviamente le "high-level" API facilitano il lavoro dello sviluppatore il quale può ignorare diversi meccanismi interni come: connessioni, serializzazioni, deserializzazioni, ecc.

Fra le "high-level" API troviamo gli oggetti:

- `SoapClient`.
- `SoapServer`.

Fra le "low-level" API troviamo gli oggetti:

- `SoapConnector`.

- `SoapSerializer`.
- `SoapReader`.

WSDL è un formato XML per descrivere i servizi remoti offerti dal server. Attraverso il file WSDL sii identifica i servizi forniti dal server e le relative operazioni che costituiscono tali servizi. Per ogni operazione, il file WSDL inoltre descrive il formato che il cliente deve seguire nel richiedere le operazioni.

In aggiunta al file WSDL, è necessario creare sul server un opportuno file WSML.
Il file WSML fornisce le informazioni necessarie per legare le operazioni del servizio (come descritto nel file WSDL) al metodo specifico nell'oggetto COM. Il file WSML determina quale oggetto COM si prende carico d'eseguire il servizio per ognuna delle operazioni possibili.

Lato Client

Flusso dati sul lato Client

Il seguente diagramma illustra come l'implementazione Microsoft SOAP Toolkit gestisce il flusso di dati sul lato client. L'applicazione utilizzatrice invia un messaggio all'oggetto `SoapClient` richiedendo un'operazione (in questo caso, la somma di due numeri). L'oggetto `SoapClient` processa la richiesta e formula una richiesta SOAP al server. Il server riceve la richiesta, esegue l'operazione richiesta, ed invia il risultato dell'elaborazione al client, sottoforma di risposta SOAP. L'oggetto `SoapClient` processa questa risposta SOAP ed invia un messaggio, contenente il risultato dell'elaborazione, all'applicazione utilizzatrice.

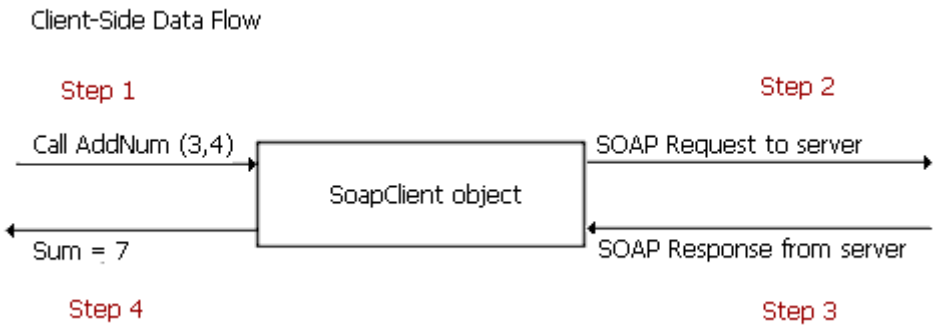


Figura 17:Flusso dati sul lato client

Settaggio dell'oggetto SoapClient

Prima di inviare la richiesta all'oggetto `SoapClient` (implementato nella libreria MSSOAP1.dll), è necessario che l'applicazione utilizzatrice istanzi l'oggetto.
Per fare ciò, l'applicazione deve chiamare il metodo `SoapClient.mssoapinit` passandogli i seguenti parametri:

- Il nome del file WSDL.
- Il nome del servizio.
- La porta su cui è attivo il servizio.

Internamente, tutti i metodi descritti nel file WSDL (un servizio può disporre di più metodi) sono legati dinamicamente all'oggetto `SoapClient` durante l'inizializzazione. Questo legame dinamico permette di chiamare un metodo qualunque metodo descritto nel file WSDL.

Nell'esempio seguente (VBScript), si accede al servizio `TestService` e si chiede l'utilizzo del metodo `AddNumbers(2, 3)`; il tutto avviene come se questo metodo appartenesse all'applicazione utilizzatrice anziché ad un servizio remoto.

```
SET soapclient = CreateObject("MSSOAP.SoapClient")
Call
soapclient.mssoapinit("http://IISServer/VirtualRoot/MyWSDL.w
sdl",
                              "TestService",
                              "TestPort")
'a questo punto è possibile accedere a tutti i metodi
'specificati nel file MyWSDL.wsdl
wscript.echo soapclient.AddNumbers(2, 3)
```

Elaborazione interna all'oggetto SoapClient

L'oggetto `SoapClient` è considerato parte delle "high-level" API. Infatti, come visto nell'esempio precedente, l'applicazione utilizzatrice non deve preoccuparsi delle molteplici operazioni che internamente l'oggetto `SoapClient` esegue per inviare la richiesta al server.
Il seguente diagramma illustra come l'oggetto `SoapClient` processa la richiesta dell'applicazione utilizzatrice, formula un messaggio SOAP di richiesta, e processa il messaggio SOAP di risposta.

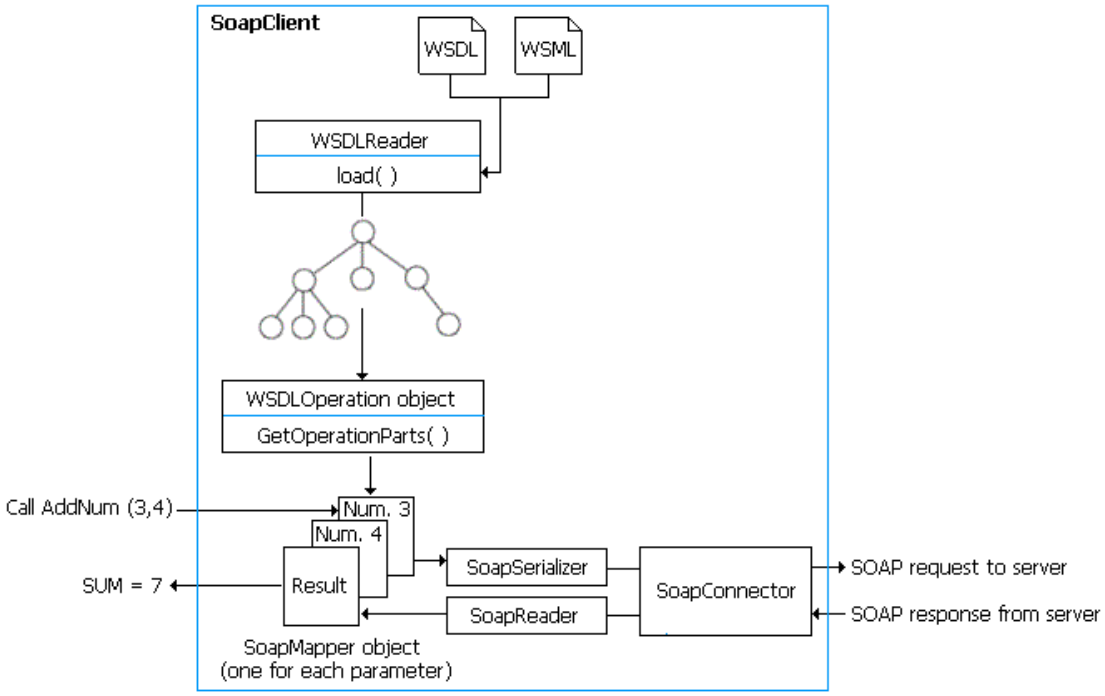


Figura 18: Elaborazione interna dell'oggetto SoapClient

L'oggetto `WSDLReader` legge i file WSDL e WSML attraverso il parser DOM e gli analizza. In base a questa analisi, l'oggetto `WSDLReader` crea un oggetto `WSDLOperation` per richiedere l'operazione (nell'esempio `AddNum(3,4)`). L'oggetto `WSDLOperation` chiama il metodo `GetOperationParts`, il quale ritorna dall'albero DOM una collezione d'elementi definiti dall'input e dall'output

dell'operazione richiesta. L'oggetto `SoapClient` crea un oggetto `SoapMapper` per ognuno di questi elementi e carica il valore specificato nella richiesta all'interno di questi oggetti. L'oggetto `SoapSerializer`, a questo punto, costruisce un messaggio SOAP di richiesta con gli appropriati oggetti `SoapMapper` e lo invia al server.

Il server processa la richiesta SOAP e ritorna una risposta SOAP al cliente. Da questa risposta, l'oggetto `SoapClient` legge il risultato dell'operazione dagli appropriati oggetti `SoapMapper` e ritorna il risultato all'applicazione utilizzatrice.

`WSDLReader`, `WSDLOperation` e `SoapReader` sono considerati parte delle "low-level" API.

Lato Server

Flusso dati sul lato server

Sul lato server, si ha la possibilità di scegliere fra due listen SOAP (si occupano di gestire le richieste SOAP in arrivo al server):

- Un Internet Server API (ISAPI) server.
- Un Internet Server Pages (ASP) server.

Nel file WSDL, l'URL identifica come il lato server gestisce una richiesta SOAP.
Se nel file WSDL troviamo un URL del tipo 'http://localhost/LatoServer/DocServer.wsdl' allora si è in presenza di un listen ISAPI, altrimenti se troviamo un URL del tipo 'http://localhost/LatoServer/DocServer.asp' allora si è in presenza di un listen ASP.

Il modo in cui la richiesta SOAP invoca un listen ISAPI o ASP, come il server gestisce l'arrivo e la spedizione dati non cambia. L'oggetto `SoapServer` riceve la richiesta SOAP dal cliente, elabora la richiesta, ed avanza una chiamata al metodo COM che contiene l'operazione richiesta (nel seguente diagramma, si esamina il metodo `AddNum`).
Il metodo che è stato chiamato invia il risultato dell'operazione richiesta all'oggetto `SoapServer`, il quale include quest'informazione all'interno della risposta SOAP che l'oggetto spedisce al cliente.

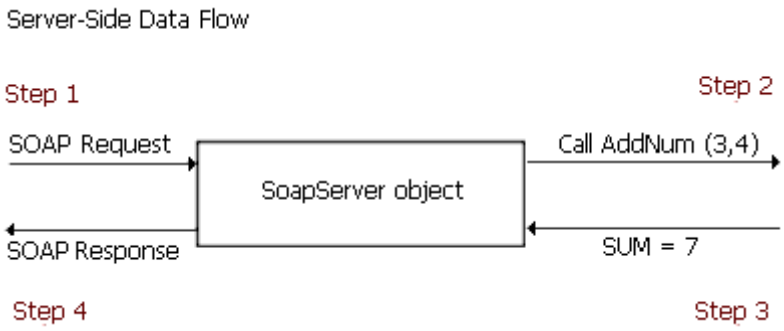


Figura 19:Flusso dei dati sul lato server

Elaborazione interna all'oggetto SoapServer

L'oggetto `SoapServer` è considerato parte delle "high-level" API, quindi l'applicazione che l'utilizza non deve preoccuparsi delle molteplici operazioni che internamente l'oggetto `SoapServer` esegue per elaborare le richieste del cliente.
Il seguente diagramma illustra come l'oggetto `SoapServer` elabora il messaggio SOAP di richiesta e formula l'appropriato messaggio SOAP di risposta.

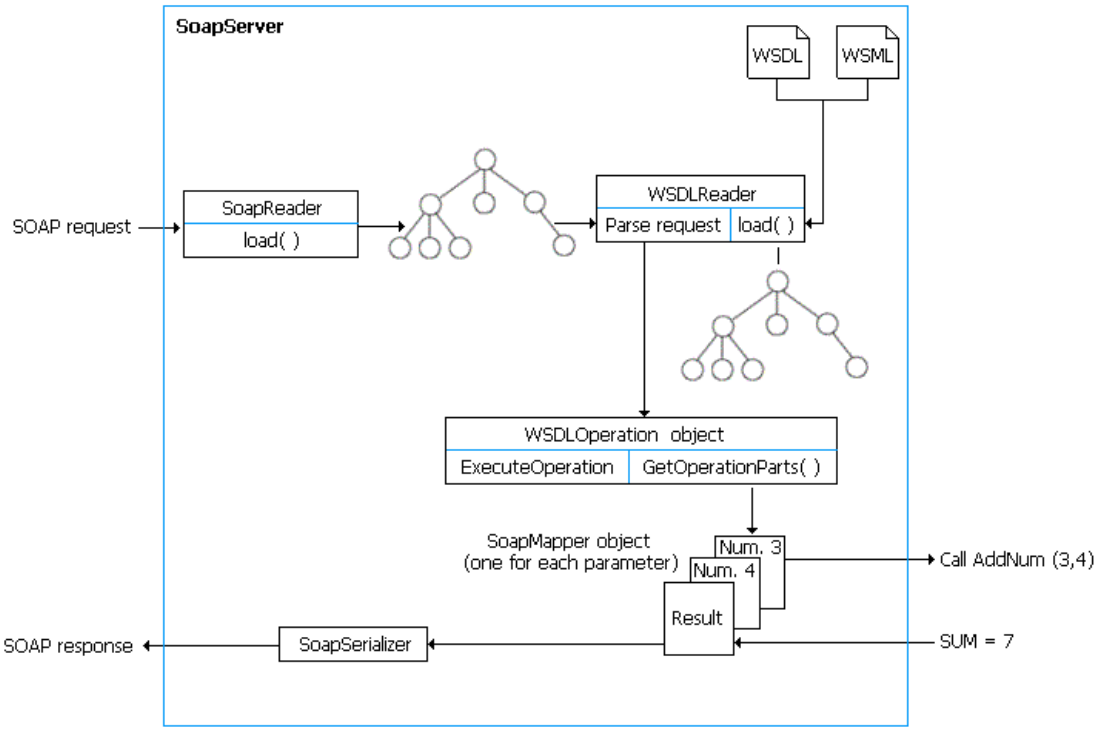


Figura 20:Elaborazione interna dell'oggetto SoapServer

Alla ricezione della richiesta SOAP proveniente dal cliente, l'oggetto `SoapReader` carica questo messaggio di richiesta su una struttura DOM, mentre l'oggetto `WSDLReader` carica i file WSDL e WSML all'interno di un'altra struttura DOM.
L'oggetto `WSDLReader` elabora la richiesta e crea un'oggetto `WSDLOperation` per l'operazione richiesta. L'oggetto `WSDLOperation` chiama il metodo `GetOperationParts`, il quale ritorna dall'albero WSDL/WSML DOM una collezione d'elementi definiti dall'input e dall'output dell'operazione richiesta. L'oggetto `SoapServer` crea un oggetto `SoapMapper` per ognuno di questi elementi e carica il corrispondente valore della richiesta in questi oggetti. L'oggetto `SoapServer`, a questo punto, chiama il metodo COM corrispondente all'operazione richiesta.

Il metodo COM elabora i parametri inclusi nella chiamata e ritorna il risultato all'oggetto `SoapServer`, il quale trasferisce questo su un opportuno oggetto `SoapMapper`. A questo punto, l'oggetto `SoapServer` utilizza un oggetto `SoapSerializer` per costruire il messaggio SOAP di risposta con il risultato. Infine, l'oggetto `SoapServer` spedisce il messaggio SOAP di risposta al cliente.

Come già detto, `WSDLReader`, `WSDLOperation` e `SoapReader` sono considerati parte delle "low-level" API.

Esempio Adder-COM (web server ISS e client Apache SOAP)

L'esempio Adder-COM è già stato studiato in precedenza e prevedeva un client Java e un server Visual Basic. La limitazione di questo studio riguardava la modalità di pubblicazione del sevizio che avveniva, pur essendo un componente COM, attraverso il

web server Tomcat che risulta completamente compatibile con Apache SOAP (quindi Java).
In generale la pubblicazione di un servizio COM avviene sul web server Internet Information Server (IIS), infatti, questi due tipi di prodotti sono fortemente legati fra loro facendo capo al mondo Microsoft.
Inoltre, nello studio dell'interoperabilità di SOAP, occorre prevedere che un client (Apache SOAP/Tomcat) si trovi su una piattaforma Unix, mentre il server (Visual Basic/IIS) si trovi su una piattaforma Windows.
In particolare immaginiamo di avere un client Apache SOAP e un server Visual Basic pubblicato su Internet Information Server utilizzando l'utility Microsoft SOAP Toolkit 2.0.
Purtroppo, in questo contesto, non è possibile far interagire direttamente il client ed il server così come sono stati concepiti nello studio precedente.
Questo deriva dal fatto che fra le implementazioni SOAP Apache e di Microsoft vi sono delle piccole, ma importanti, differenze.
Infatti, Apache SOAP per tutti i parametri di richiesta e di risposta, specifica sempre la loro tipologia.
Ad esempio:

- `<n1 xsi:type="xsd:int">2</n1>` è il primo parametro della somma e viene specificato esplicitamente che si tratta di un intero.
- `<return xsi:type="xsd:short">6</return>` è il valore di risposta e viene specificato esplicitamente che si tratta di uno short.

Al contrario per l'implementazione Microsoft ciò non accade, infatti, il valore di risposta generato dal web server è del tipo `<Result>6</Result>`; quindi non è riportato il tipo di dato ed inoltre cambia anche il nome da `return` diventa `Result`.
Con lo studio precedente, queste differenze non si erano evidenziato perché, sia dal lato client che dal lato server, la gestione dei messaggi SOAP era affidata ad Apache SOAP.
In questa nuova situazione, occorre gestire direttamente questa differenza, altrimenti l'implementazione Apache SOAP rigetta la risposta generata da Microsoft SOAP Toolkit.

Server Adder-COM

Il codice del server è il seguente:

```
Public Function add(ByVal n1 As Integer, ByVal n2 As
Integer) As Integer
add = n1 + n2
End Function
```

Come si può notare, si tratta dello stesso codice server già utilizzato in precedenza.
Il cambiamento riguardo la modalità con cui avviene la pubblicazione del servizio.
Utilizzando lo strumento WSDL Generator di Microsoft SOAP Toolkit, è possibile generare (con l'aiuto di una wizard) i file WSDL e WSML relativi al servizio da pubblicare.
All'interno di questa wizard non è necessario conoscere nessuna specifica per la creazione dei file WSDL e WSML, ma solamente le seguenti informazioni:

- il nome del servizio da pubblicare (Adder-COM);
- il percorso del componente COM da pubblicare (D:\...\Adder-COM.dll);
- i metodi da esportare per servizio che stiamo creando (add);
- la posizione di listen del servizio (http://localhost:80/Adder-COM/);
- la tipologia di listen (ASP);
- la versione del XSD Schema Namespace da utilizzare (1999);
- il charset da utilizzare (UTF-8);
- il percorso dove memorizzare i file WSDL e WSML (D:\...).

La servelt ASP che si occupa di ascoltare richieste provenienti dal client è realizzata automaticamente dal WSDL Generetor in VBScript, ed ha il seguente codice:

```
<%@ LANGUAGE=VBScript %>
<%
Option Explicit
On Error Resume Next
Response.ContentType = "text/xml"
If IsEmpty(Application("SoapServer")) Then
  Application.Lock
  If IsEmpty(Application("SoapServer")) Then
    Dim SoapServer
    Dim WSDLFilePath
    Dim WSMLFilePath
    WSDLFilePath = Server.MapPath("Adder-COM.wsdl")
    WSMLFilePath = Server.MapPath("Adder-COM.wsml")
    Set SoapServer =
Server.CreateObject("MSSOAP.SoapServer")
    If Err Then SendFault "Cannot create SoapServer object.
" & Err.Description
    SoapServer.Init WSDLFilePath, WSMLFilePath
    If Err Then SendFault "SoapServer.Init failed. " &
Err.Description
    Set Application("SoapServer") = SoapServer
  End If
  Application.UnLock
End If
Set SoapServer = Application("SoapServer")
SoapServer.SoapInvoke Request, Response, ""
If Err Then SendFault "SoapServer.SoapInvoke failed. " &
Err.Description
Sub SendFault(ByVal LogMessage)
  Dim Serializer
  On Error Resume Next
  ' "URI Query" logging must be enabled for AppendToLog to
work
  Response.AppendToLog " SOAP ERROR: " & LogMessage
  Set Serializer =
Server.CreateObject("MSSOAP.SoapSerializer")
  If Err Then
    Response.AppendToLog "Could not create SoapSerializer
object. " & Err.Description
    Response.Status = "500 Internal Server Error"
  Else
    Serializer.Init Response
    If Err Then
      Response.AppendToLog "SoapSerializer.Init failed. " &
Err.Description
      Response.Status = "500 Internal Server Error"
    Else
      Serializer.startEnvelope

      Serializer.startBody
      Serializer.startFault "Server", "The request could not
be processed due to a problem in the server. Please contact
the system admistrator. " & LogMessage
```

```
      Serializer.endFault
      Serializer.endBody
      Serializer.endEnvelope
      If Err Then
        Response.AppendToLog "SoapSerializer failed. " &
Err.Description
        Response.Status = "500 Internal Server Error"
      End If
    End If
  End If
  Response.End
End Sub
%>
```

In questo modo si cerca di creare un Server SOAP con le specifiche definite dai file Adder-COM.wsdl e Adder-COM.wsml definiti con Microsoft SOAP Toolkit.
All'interno del codice appena visto si utilizzano le high API del toolkit.

Il file Adder-Com.wsdl ha il seguente contenuto:

```
<?xml version='1.0' encoding='UTF-8' ?>
 <!-- Generated 04/24/02 by Microsoft SOAP Toolkit WSDL File
Generator, Version 1.02.813.0 -->
<definitions  name ='Adder-COM'   targetNamespace =
'http://tempuri.org/wsdl/'
    xmlns:wsdlns='http://tempuri.org/wsdl/'
    xmlns:typens='http://tempuri.org/type'
    xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
    xmlns:xsd='http://www.w3.org/1999/XMLSchema'
    xmlns:stk='http://schemas.microsoft.com/soap-
toolkit/wsdl-extension'
    xmlns='http://schemas.xmlsoap.org/wsdl/'>
  <types>
    <schema targetNamespace='http://tempuri.org/type'
      xmlns='http://www.w3.org/1999/XMLSchema'
      xmlns:SOAP-
ENC='http://schemas.xmlsoap.org/soap/encoding/'
      xmlns:wsdl='http://schemas.xmlsoap.org/wsdl/'
      elementFormDefault='qualified'>
    </schema>
  </types>
  <message name='adder.add'>
    <part name='n1' type='xsd:short'/>
    <part name='n2' type='xsd:short'/>
  </message>
  <message name='adder.addResponse'>
    <part name='Result' type='xsd:short'/>
  </message>
  <portType name='adderSoapPort'>
    <operation name='add' parameterOrder='n1 n2'>
      <input message='wsdlns:adder.add' />
      <output message='wsdlns:adder.addResponse' />
    </operation>
  </portType>
  <binding name='adderSoapBinding'
type='wsdlns:adderSoapPort' >
```

```
    <stk:binding preferredEncoding='UTF-8'/>
    <soap:binding style='rpc'
transport='http://schemas.xmlsoap.org/soap/http' />
    <operation name='add' >
      <soap:operation
soapAction='http://tempuri.org/action/adder.add' />
      <input>
        <soap:body use='encoded'
namespace='http://tempuri.org/message/'

encodingStyle='http://schemas.xmlsoap.org/soap/encoding/' />
      </input>
      <output>
        <soap:body use='encoded'
namespace='http://tempuri.org/message/'

encodingStyle='http://schemas.xmlsoap.org/soap/encoding/' />
      </output>
    </operation>
  </binding>
  <service name='Adder-COM' >
    <port name='adderSoapPort'
binding='wsdlns:adderSoapBinding' >
      <soap:address location='http://localhost:80/Adder-
COM/Adder-COM.ASP' />
    </port>
  </service>
</definitions>
```

Il file Adder-Com.wsml ha il seguente contenuto:

```
<?xml version='1.0' encoding='UTF-8' ?>
 <!-- Generated 04/24/02 by Microsoft SOAP Toolkit WSDL File
Generator, Version 1.02.813.0 -->
<servicemapping name='Adder-COM'>
  <service name='Adder-COM'>
    <using PROGID='APACHEADDER.adder' cachable='0'
ID='adderObject' />
    <port name='adderSoapPort'>
      <operation name='add'>
        <execute uses='adderObject' method='add'
dispID='1610809344'>
          <parameter callIndex='1' name='n1'
elementName='n1' />
          <parameter callIndex='2' name='n2'
elementName='n2' />
          <parameter callIndex='-1' name='retval'
elementName='Result' />
        </execute>
      </operation>
    </port>
  </service>
</servicemapping>
```

Analizziamo meglio la struttura del documento WSML:

- `<servicemapping name='Adder-COM'>`

L'elemento `servicemapping` è la radice del documento WSML; quest'elemento contiene tre elementi figli: `service`, `port` ed `operation`.

- `<service name='Adder-COM'>`
  `<using PROGID='APACHEADDER.adder' cachable='0' ID='adderObject' />`

  Il `name` del `service` identifica a quale servizio (elemento `service`), del corrispondente file WSDL, si fa riferimento. A sua volta l'elemento `service` contiene, fra gli altri, l'elemento figlio: `using`, il quale identifica l'oggetto COM necessario al servizio.
  - `PROGID='APACHEADDER.adder'`
    L'attributo `PROGID` identifica la classe COM che implementa tutti i metodi del servizio in esame.
  - `cachable='0'`
    L'attributo booleano `cachable` specifica se l'istanza della classe (l'oggetto COM) deve rimanenre in memoria per tutto il tempo in cui vi rimane l'oggetto `soapServer` (valore `'1'`) oppure no (valore `'0'`).
  - `ID='adderObject'`
    L'attributo `ID` specifica il nome con cui l'oggetto COM sarà riferito nel seguito del documento WSML.

- `<port name='adderSoapPort'>`
  Il `name` del `port` identifica a quale elemento `portType`, del corrispondente file WSDL, si fa riferimento.

- `<operation name='add'>`
  Per ogni operazione definita all'interno del `portType` (nel documento WSDL), c'è un elemento `operation` nel file WSML corrispondente.

- `<execute uses='adderObject' method='add' dispID='1610809344'>`
  L'elemento `execute` (figlio dell'elemento `operation`) specifica l'oggetto che ha il compito di eseguire l'operazione in esame.
  - L'attributo `uses` identifica il nome dell'oggetto.
  - L'attributo `method` identifica il nome del metodo.
  - L'attributo opzionale `dispID` fornisce il "dispatch ID" del metodo; la presenza di quest'attributo migliora le performance, ma non è strettamente necessaria.

- `<parameter callIndex='1' name='n1' elementName='n1' />`
  L'elemento `parameter` (figlio dell'elemento `execute`) descrive un parametro per il metodo in esame.
  - L'attributo `callIndex` fornisce il numero del parametro (il primo parametro ha valore `'1'`, il secodo ha valore `'2'`, e così via). Il valore `'-1'` identifica il parametro di ritorno.
  - L'attributo `name` fornisce un nome univoco per il parametro (utilizzato principalmente per scopi interni al documento WSML).
  - L'attributo `elementName` fornisce il nome dell'elemento, nella sezione `message` del documento WSDL, che contiene il valore del parametro.

Cliet Adder-COM

Il codice del client è il seguente:

```
package samples.com.client;

import java.io.*;
import java.util.*;
import java.net.*;
import org.w3c.dom.*;
import org.apache.soap.util.xml.*;
import org.apache.soap.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;

import org.apache.soap.rpc.*;
import org.apache.soap.transport.http.SOAPHTTPConnection;

public class Addit_ms
{
 public static void main( String[] args) throws Exception
 {

   Integer n1=null;
   Integer n2=null;
   final String urn="http://tempuri.org/message/"; //e non
più "urn:adder-COM";
   if( args.length != 2 ) explain();

   try{
   n1= new Integer( args[0]);
   n2= new Integer( args[1]);
   }catch ( NumberFormatException e)
   {
     explain();
   }


    //novità
    SOAPMappingRegistry smr = new SOAPMappingRegistry ();
    StringDeserializer sd = new StringDeserializer ();
    smr.mapTypes (Constants.NS_URI_SOAP_ENC,
                  new QName ("", "Result"), null, null, sd);

    // settaggio procollo trasporto e parametri
    SOAPHTTPConnection st = new SOAPHTTPConnection();


   Vector params = new Vector ();
   params.addElement (new Parameter("n1" , Integer.class,
n1, null));
   params.addElement (new Parameter("n2" , Integer.class,
n2, null));

   URL url = new URL ("http://" + serverhost + ":" +
serverport+ soapservlet);
```

```
   // Settaggio call.
   Call call = new Call ();

   //novità
   call.setSOAPTransport(st);
   call.setSOAPMappingRegistry (smr);

   call.setTargetObjectURI (urn);
   call.setMethodName ("add");
   call.setEncodingStyleURI
("http://schemas.xmlsoap.org/soap/encoding/");
   //equivalente a
call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

   call.setParams (params);
   Response resp = call.invoke (/* router URL */ url, /*
actionURI */ "http://tempuri.org/action/adder.add" );

   if (resp.generatedFault ()) {
     Fault fault = resp.getFault ();
     System.out.println ("Ouch, the call failed: ");
     System.out.println ("  Fault Code   = " +
fault.getFaultCode ());
     System.out.println ("  Fault String = " +
fault.getFaultString ());
   } else {
     Parameter result = resp.getReturnValue ();
     System.out.println("The sum of " + args[0] + " and " +
args[1] + " is " + result.getValue());
   }

 }
 private static void explain()
 {
   System.err.println("Please provide two integers as inputs
to add!");
   System.exit(8);
 }
 static String serverhost= "localhost";
 static String serverport= "8081";
 static String soapservlet= "/Adder-COM/Adder-COM.asp";

} // end  addit_ms
```

Il client contiene le novità di quest'esempio, in particolare le seguenti istruzioni:

- `SOAPMappingRegistry smr = new SOAPMappingRegistry ();`
  É il costrutto relativo alla classe org.apache.soap.encoding , l'oggetto SOAPMappingRegistry è un XMLJavaMappingRegistry con i serializzatori e deserializzatori preregistrati per il supporto di SOAP.

- `StringDeserializer sd = new StringDeserializer ();`
  É un deserializzatore, utilizzato per deserializzare un qualcosa in una stringa.

- `smr.mapTypes (Constants.NS_URI_SOAP_ENC, new QName ("", "Result"), null, null, sd);`
  É un metodo della classe org.apache.soap.util.xml.XMLJavaMappingRegistry ed ha la seguente struttura:
  public void mapTypes(java.lang.String encodingStyleURI,
  QName elementType,
  java.lang.Class javaType,
  Serializer s,
  Deserializer ds)
  In questo modo si è creato un nuovo deserializzatore che informa il client Apache SOAP di non aspettarsi l'attributo `xsi:type` nella risposta ricevuta, ma solo l'elemento `Result`.

- `SOAPHTTPConnection st = new SOAPHTTPConnection();`
  Costrutto della classe org.apache.soap.transport.http.SOAPHTTPConnection, per utilizzare HTTP come protocollo di trasporto (non è una novità).

- `call.setSOAPTransport(st);`
  Imposta l'utilizzo di HTTP come protocollo di trasporto (non è una novità).

- `call.setSOAPMappingRegistry (smr);`
  Imposta l'utilizzo dell'oggetto `smr` di (tipo SOAPMappingRegistry) per la serializzazione e deserializzazione dei dati scambiati.

- `call.setTargetObjectURI (urn);`
  Imposta il nome del servizio web da contattare, in particolare `urn` contiene il valore `"http://tempuri.org/message/"` (diversamente da `"urn:adder-COM"` visto per il server Tomcat), questo cambiamento è legato all'utilizzo di Microsoft SOAP Toolkit e al relativo file WSDL generato.

- `Response resp = call.invoke (/* router URL */ url, /* actionURI */ "http://tempuri.org/action/adder.add" );`
  In questo caso è necessario impostare l'header SOAPAction nella request SOAP, questo cambiamento è legato all'utilizzo di Microsoft SOAP Toolkit ed il valore `"http://tempuri.org/action/adder.add"` dipende dal relativo file WSDL generato.

Si deve notare che il codice client appena visto funziona correttamente (cambiando ovviamente il nome della risorsa e del servizio opportunamente) anche se il server che esegue la somma è scritto in Apache SOAP (Java) e pubblicato con Tomcat. Infatti, le istruzioni appena viste aggiungono un nuovo deserializzatore, ma lasciano immutato quello predefinito che può continuare ad operare.

Messaggi SOAP di Adder-COM

Per valutare cosa accade a livello di call e response si utilizza ancora l'utility tpcTrace, con l'accortezza di inviare la call sulla porta 8081 e mantenere il server in ascolto sulla porta 80 così da permettere il tunneling.

Il messaggio di richiesta inviato dal client Apache SOAP è il seguente:

```
POST /Adder-COM/Adder-COM.asp HTTP/1.0
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: 468
SOAPAction: "http://tempuri.org/action/adder.add"

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:add xmlns:ns1="http://tempuri.org/message/" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
">
<n1 xsi:type="xsd:int">2</n1>
<n2 xsi:type="xsd:int">4</n2>
</ns1:add>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Come già detto, utilizzando un web server di tipo ISS, i cambiamenti riguardano il nome della risorsa richiesta (`/Adder-COM/Adder-COM.asp`), l'header SOAPAction (`"http://tempuri.org/action/adder.add"`) e il namespace del metodo invocato (`"http://tempuri.org/message/"`).
Per il resto non si hanno cambiamenti rispetto a quanto studiato con il web server Tomcat.

Il messaggio di risposta inviato dal server Visual Basic è il seguente:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Wed, 24 Apr 2002 14:28:19 GMT
Connection: Keep-Alive
Content-Length: 358
Content-Type: text/xml; charset="UTF-8"
Set-Cookie: ASPSESSIONIDGGQGQTOK=KBIBIAGDHGEJEHPCIECLBGEG;
path=/
Cache-control: private

<?xml version="1.0" encoding="UTF-8" standalone="no"?><SOAP-
ENV:Envelope SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"><SOAP-
ENV:Body><SOAPSDK1:addResponse
xmlns:SOAPSDK1="http://tempuri.org/message/"><Result>6</Resu
lt></SOAPSDK1:addResponse></SOAP-ENV:Body></SOAP-
ENV:Envelope>
```

Come già detto, diversamente da Apache SOAP, l'implementazione di Microsoft SOAP Toolkit genera un valore di ritorno `Result` senza specificare l'attributo sul tipo di dato.
Per il resto non si hanno cambiamenti rispetto a quanto studiato con il web server Tomcat.

Esempio Add (web server Tomcat e client VB)

Questo esempio, dal punto di vista del servizio offerto, coincide esattamente con l'esempio Adder-COM: entrambi eseguono la somma di due addendi.
Ciò che cambia è l'implementazione, in questo caso, il server è scritto in Java (Apache SOAP) e pubblicato su web server Tomcat, mentre il client è scritto in Visual Basic.
Nell'implementare questo servizio occorre tener presente che Apache SOAP accetta richieste solo se per tutti i parametri è indicato il loro tipo.
Ad esempio: `<Num1 xsi:type="xsd:int">2</n1>` è il primo parametro della somma e si deve specificare esplicitamente che si tratta di un intero.
Infatti, inviando al server Apache SOAP una richiesta contenente `<Num1>2</n1>`, questa viene rigettata.
Al contrario le implementazioni SOAP di Microsoft non fanno questa discriminazione.

Server Add

Il codice del server è il seguente:

```
import java.io.*;
import java.util.*;

public class add_Server
{

  public int add(int a1,int a2)
   {
      return (a1+a2);
   }
}
```

Come già sottolineato in precedenza, questo sorgente Java non ha nulla di particolare che lo renda assimilabile ad un servizio web.
Per la sua pubblicazione è necessario creare il deployment descriptor (`add_deploy.xml`), che ha il seguente contenuto:

```
<isd:service xmlns:isd="http://xml.apache.org/xml-
soap/deployment"
             id="urn:add_Server">
  <isd:provider type="java"
                scope="Request"
                methods="add">
    <isd:java class="add_Server" static="false"/>
  </isd:provider>
</isd:service>
```

Infine per effettuare fisicamente il deploy del servizio dobbiamo eseguire, dal prompt comandi, la seguente istruzione:

```
java org.apache.soap.server.ServiceManagerClient
http://localhost:8080/soap/servlet/rpcrouter deploy
add_deploy.xml
```

Client Add

Siccome Apache SOAP non fornisce il supporto per la creazione del file WSDL relativo al servizio pubblicato, non è possibile accedere al sevizio con un client Microsoft SOAP

Toolkit che sfrutti le “high-level” API. Quindi si è creato un client Microsoft SOAP Toolkit che utilizza le “low-level” API per accedere al servizio.
Il codice sorgente Visual Basic del client è il seguente:

```
'Project Add-client
Option Explicit

Private Sub cmdadd_Click()

    On Error GoTo ErrorHandler

    Dim Serializer As SoapSerializer
    Dim Reader As SoapReader
    Dim Connector As SoapConnector

    Set Connector = New HttpConnector
    Connector.Property("EndPointURL") = cbURL.Text
    If chkProxyUse.Value = vbChecked Then
        Connector.Property("UseProxy") = True
    End If

    Connector.BeginMessage

    Set Serializer = New SoapSerializer
    Serializer.Init Connector.InputStream

    'Envelope
    Serializer.startEnvelope
    'attributi relativi a namespace necessari per comunicare
con Apache SOAP
    Serializer.SoapAttribute "xsi", "",
"http://www.w3.org/1999/XMLSchema-instance", "xmlns"
    Serializer.SoapAttribute "xsd", "",
"http://www.w3.org/1999/XMLSchema", "xmlns"

    'Body
    Serializer.startBody

    'nome del metodo, relativo namespace, encodingStyle,
prefisso del namespace dell'elemento
    Serializer.startElement "add", "urn:add_Server",
"http://schemas.xmlsoap.org/soap/encoding/", "ns"

    'primo addendo
    Serializer.startElement "Num1"
    'attributo relativo al tipo di dato del primo addendo
    'necessario per comunicare con Apache SOAP
    Serializer.SoapAttribute "type", "", "xsd:int", "xsi"
    'valore del primo addendo
    Serializer.writeString txtNum1.Text
    Serializer.endElement

    'secondo addendo
    Serializer.startElement "Num2"
    'attributo relativo al tipo di dato del secondo addendo
    'necessario per comunicare con Apache SOAP
```

```
    Serializer.SoapAttribute "type", "", "xsd:int", "xsi"
    'valore del secondo addendo
    Serializer.writeString txtNum2.Text
    Serializer.endElement

    Serializer.endElement

    Serializer.endBody
    Serializer.endEnvelope

    Connector.EndMessage

    'lettura valore di risposta del servizio
    Set Reader = New SoapReader
    Reader.Load Connector.OutputStream

    If Not Reader.Fault Is Nothing Then
        MsgBox Reader.faultstring.Text, vbExclamation
'indicazione sulla situazione d'errore
    Else
        MsgBox "La somma dei due addendi vale " &
Reader.RPCResult.Text
    End If

    Exit Sub

ErrorHandler:

    MsgBox "ERROR: " & Err.Description, vbExclamation
    Err.Clear
    Exit Sub

End Sub

Private Sub Form_Load()
    cbURL.AddItem
"http://localhost:8081/soap/servlet/rpcrouter"
    cbURL.ListIndex = 0
End Sub
```

Il client richiede all'utente, tramite una dialog box, di inserire i due addendi da sommare, dopodiché invia la richiesta al server, attende la risposta ed infine visualizza il risultato di tale somma.
È importante sottolineare, che gli elementi utilizzati appartengono alla libreria Microsoft Soap Type Library che deve, quindi, essere inclusa nel progetto.

Gli elementi più interessanti sono i seguenti ("low-level" API):

- `Dim Connector As SoapConnector`
  Ha il compito di inviare (ricevere) messaggi SOAP attraverso un protocollo di trasporto come HTTP, SMTP o FTP.
  - `Set Connector = New HttpConnector`
    Rappresenta l'implementazione di `SoapConnector` attraverso il protocollo HTTP.
  - `Connector.Property("EndPointURL") = cbURL.Text`

In questo modo si imposta l'endpoint del servizio (nell'esempio `http://localhost:8081/soap/servlet/rpcrouter`).

- `Connector.BeginMessage`
  Segnala l'inizio di un messaggio SOAP.
- `Connector.EndMessage`
  Segnala la fine di un messaggio SOAP.

- `Dim Serializer As SoapSerializer`
  Ha il compito di costruire un messaggio SOAP.
  - `Serializer.Init Connector.InputStream`
    Il metodo `Init` ha il compito d'inizializzare l'oggetto `SoapSerializer`, in particolare viene indicata la destinazione a cui inviare il testo scritto dall'oggetto `SoapSerializer`.
    In questo caso la destinazione è l'oggetto di tipo `SoapConnector`, infatti, si deve inviare il messaggio SOAP attraverso la connessione HTTP impostata.
  - `Serializer.startEnvelope(Serializer.endEnvelope)`
    Segnala l'inizio (la fine) dell'elemento Envelope nel messaggio SOAP.
  - `Serializer.SoapAttribute "xsi", "", "http://www.w3.org/1999/XMLSchema-instance", "xmlns"`
    Il metodo `SoapAttribute` aggiunge un attributo ad un elemento del messaggio SOAP. Il metodo prevede quattro parametri: il nome dell'attributo, l'URI del namespace relativo all'attributo (opzionale), il valore dell'attributo (opzionale) ed il prefisso del namespace usato per l'elemento (opzionale).
    In questo caso definisco il namespace `xsi`.
  - `Serializer.startBody (Serializer.endBody)`
    Segnala l'inizio (la fine) dell'elemento Body nel messaggio SOAP.
  - `Serializer.startElement "add", "urn:add_Server", "http://schemas.xmlsoap.org/soap/encoding/", "ns"`
    Il metodo `startElement` indica l'inizio di una entry (elemento figlio) nel Body di un messaggio SOAP. Il metodo prevede quattro parametri: il nome dell'elemento, il namespace URI relativo all'elemento (opzionale), l'URI dell'attributo encodingStyle dell'elemento (opzionale) ed il prefisso per il namespace usato dall'elemento (opzionale).
    In questo caso si crea una entry che indica il nome del metodo da invocare, il nome del servizio che lo contiene (ossia il namespace del metodo), con l'encodingStyle standard ed il prefisso `ns` per il namespace del metodo.
  - `Serializer.startElement "Num1"`
    Si crea la entry per il primo parametro da inviare al metodo.
  - `Serializer.SoapAttribute "type", "", "xsd:int", "xsi"`
    Si definisce l'attributo relativo al tipo di parametro (assolutamente necessario per comunicare con Apache SOAP).
  - `Serializer.writeString txtNum1.Text`
    Scrive il valore del primo parametro.
  - `Serializer.endElement`
    Chiude la entry in uso del Body.

- `Dim Reader As SoapReader`

Ha il compito di leggere un messaggio SOAP.

- `Reader.Load Connector.OutputStream`
  Il metodo `Load` legge un documento XML da file, stream o da URL. In questo caso legge dallo stream di output dell'oggetto Connector, cioè dal messaggio HTTP di risposta in arrivo.
- `Reader.Fault`
  Ritorna l'eventuale elemento Fault presente nel messaggio SOAP.
- `Reader.faultstring.Text`
  Ritorna il valore dell'elemento faultstring dell'elemento Fault presente nel messaggio SOAP.
- `Reader.RPCResult.Text`
  Ritorna il valore del primo elemento figlio della prima entry del Body presente nel messaggio SOAP di risposta.

Messaggi SOAP di Add

Il messaggio di richiesta inviato dal client Visual Basic è il seguente:

```
POST /soap/servlet/rpcrouter HTTP/1.1
Content-Type: text/xml
Host: localhost
SOAPAction: ""
Content-Length: 459

<?xml version="1.0" encoding="UTF-8" standalone="no"?><SOAP-
ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"><SOAP-
ENV:Body><ns:add xmlns:ns="urn:add_Server" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
"><Num1 xsi:type="xsd:int">6</Num1><Num2
xsi:type="xsd:int">-2</Num2></ns:add></SOAP-ENV:Body></SOAP-
ENV:Envelope>
```

Come si può vedere, in quest'esempio, per comunicare correttamente con il server Apache SOAP oltre ad indicare la tipologia dei parametri occorre definire i namespace `xsi`, `xsd` e l'`encodingStyle` opportuno per il metodo invocato.

Il messaggio di risposta inviato dal server Apache SOAP è il seguente:

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 450
Set-Cookie2:
JSESSIONID=6g1n83a281;Version=1;Discard;Path="/soap"
Set-Cookie: JSESSIONID=6g1n83a281;Path=/soap
Set-Cookie2:
JSESSIONID=6g1n83a281;Version=1;Discard;Path="/soap"
Set-Cookie: JSESSIONID=6g1n83a281;Path=/soap
Servlet-Engine: Tomcat Web Server/3.2.3 (JSP 1.1; Servlet
2.2; Java 1.3.1; Windows 2000 5.0 x86; java.vendor=Sun
Microsystems Inc.)

<?xml version='1.0' encoding='UTF-8'?>
```

```
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:addResponse xmlns:ns1="urn:add_Server" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
">
<return xsi:type="xsd:int">4</return>
</ns1:addResponse>

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

Esempio Indirizzo (Web server ISS e client Visual Basic)

La novità di questo servizio, rispetto agli esempi visti finora, riguardo lo scambio di un dato di tipo complesso (Indirizzo) e non più un dato di tipo semplice (come interi, reali, stringhe, ecc.). Per quanto detto, sarà necessario definire ad hoc un serializzatore ed un deserializzatore che facciano riferimento al dato di tipo complesso.
In quest'esempio sia il server che il client hanno codice Visual Basic, il server è pubblicato su ISS ed i file WSDL e WSML del servizio sono ottenuti a partire dal WSDL Generator di Microsoft SOAP Toolkit.

Il documento di progetto `Indirizzo_Mapper` contiene al suo interno due classi:

- `Indirizzo`, che definisce la struttura del dato complesso di tipo Indirizzo, ed alcuni metodi associati.
- `IndirizzoMapper`, che definisce il mapper (serializzatore e deserializzatore) per il dato complesso di tipo Indirizzo.

La classe `Indirizzo` è la seguente:

```
Option Explicit

Private nomeVia As String
Private numCivico As Integer
Private CAP As Integer
Private citta As String
Private provincia As String

Public Property Get get_nomeVia() As String
    get_nomeVia = nomeVia
End Property

Public Property Let let_nomeVia(ByVal Value As String)
    nomeVia = Value
End Property

Public Property Get get_numCivico() As Integer
    get_numCivico = numCivico
End Property

Public Property Let let_numCivico(ByVal Value As Integer)
```

```
    numCivico = Value
End Property

Public Property Get get_CAP() As Integer
    get_CAP = CAP
End Property

Public Property Let let_CAP(ByVal Value As Integer)
    CAP = Value
End Property

Public Property Get get_citta() As String
    get_citta = citta
End Property

Public Property Let let_citta(ByVal Value As String)
    citta = Value
End Property
Public Property Get get_provincia() As String
    get_provincia = provincia
End Property

Public Property Let let_provincia(ByVal Value As String)
    provincia = Value
End Property
```

Come si può notare, il dato complesso Indirizzo è composto dai seguenti elementi:

- `nomeVia`, stringa contenente il nome della via;
- `numCivico`, intero contenente il numero civico;
- `CAP`, intero contenente il codice d'avviamento postale;
- `citta`, stringa contenente il nome della città;
- `provincia`, stringa contenente il nome della provincia.

I metodi associati ad Indirizzo servono per settare (`let_`) oppure per leggere (`get_`) il valore dei suoi attributi.

La classe `IndirizzoMapper` è la seguente:

```
Option Explicit

Implements ISoapTypeMapper

Private myMapper As ISoapTypeMapper

'inizializzazione
Private Sub ISoapTypeMapper_Init( _
    ByVal pFactory As MSSOAPLib.ISoapTypeMapperFactory, _
    ByVal pSchema As MSXML2.IXMLDOMNode, _
    ByVal xsdType As MSSOAPLib.enXSDType)

    Set myMapper = pFactory.getMapper(enXSDanyType, Nothing)
End Sub

'deserializzazione
```

```
Private Function ISoapTypeMapper_read(ByVal pNode As
MSXML2.IXMLDOMNode, _
ByVal bstrEncoding As String, ByVal encodingMode As _
MSSOAPLib.enEncodingStyle, ByVal lFlags As Long) As Variant

    Dim Indirizzo As New Indirizzo
    Dim Node As IXMLDOMNode

    Indirizzo.let_nomeVia =
myMapper.read(pNode.selectSingleNode("nomeVia"), _
bstrEncoding, encodingMode, lFlags)

    Indirizzo.let_numCivico =
myMapper.read(pNode.selectSingleNode("numCivico"), _
 bstrEncoding, encodingMode, lFlags)

    Indirizzo.let_CAP =
myMapper.read(pNode.selectSingleNode("CAP"), _
 bstrEncoding, encodingMode, lFlags)

    Indirizzo.let_citta =
myMapper.read(pNode.selectSingleNode("citta"), _
 bstrEncoding, encodingMode, lFlags)

    Indirizzo.let_provincia =
myMapper.read(pNode.selectSingleNode _
("provincia"), bstrEncoding, encodingMode, lFlags)

    Set ISoapTypeMapper_read = Indirizzo
End Function

Private Function ISoapTypeMapper_varType() As Long
    ISoapTypeMapper_varType = vbObject
End Function

'serializzazione
Private Sub ISoapTypeMapper_write(ByVal pSoapSerializer As _
MSSOAPLib.ISoapSerializer, ByVal bstrEncoding As String,
ByVal _
encodingMode As MSSOAPLib.enEncodingStyle, ByVal lFlags As
Long, _
pvar As Variant)
    Dim Indirizzo As Indirizzo
    Set Indirizzo = pvar

    pSoapSerializer.startElement "nomeVia"
    myMapper.write pSoapSerializer, bstrEncoding,
encodingMode, _
lFlags, Indirizzo.get_nomeVia
    pSoapSerializer.endElement

    pSoapSerializer.startElement "numCivico"
    myMapper.write pSoapSerializer, bstrEncoding,
encodingMode, _
lFlags, Indirizzo.get_numCivico
    pSoapSerializer.endElement
```

```
    pSoapSerializer.startElement "CAP"
    myMapper.write pSoapSerializer, bstrEncoding,
encodingMode, _
lFlags, Indirizzo.get_CAP
    pSoapSerializer.endElement

    pSoapSerializer.startElement "citta"
    myMapper.write pSoapSerializer, bstrEncoding,
encodingMode, _
lFlags, Indirizzo.get_citta
    pSoapSerializer.endElement

    pSoapSerializer.startElement "provincia"
    myMapper.write pSoapSerializer, bstrEncoding,
encodingMode, _
lFlags, Indirizzo.get_provincia
    pSoapSerializer.endElement

End Sub
```

Analizziamo la classe `IndirizzoMapper`:

- `Implements ISoapTypeMapper`
  Per poter inviare e ricevere un dato complesso, attraverso messaggi SOAP, è necessario implementare l'interfaccia `ISoapTypeMapper`.

- `Private myMapper As ISoapTypeMapper`
  Definisco un oggetto mapper di tipo `ISoapTypeMapper`.

- ```
  Private Sub ISoapTypeMapper_Init( _
   ByVal pFactory As MSSOAPLib.ISoapTypeMapperFactory, _
   ByVal pSchema As MSXML2.IXMLDOMNode, _
   ByVal xsdType As MSSOAPLib.enXSDType)
  ```
  Questo metodo, dell'interfaccia `ISoapTypeMapper`, permette di inizializzare l'oggetto mapper. In particolare, vi sono tre argomenti d'ingresso:
  - `pFactory` è un oggetto (di tipo `ISoapTypeMapperFactory`) utilizzato nel creare un nuovo mapper, il quale ci permette di serializzare/deserializzare qualunque tipo di dato standard per XML Schema Definition (XSD).
  - `pSchema` è un riferimento (di tipo `IXMLDOMNode`) al nodo complexType nello schema che definisce il tipo di dato complesso. `IXMLDOMNode` è l'oggetto principale del parser DOM, da cui ricavare informazioni sugli elementi, sugli attributi, ecc. del documento XML in esame al marser.
  - Per un nuovo mapper creato il suo valore è enXSDUndefined (-1), cioè non si fa riferimento a nessun mapper XSD di base.

  Come valore d'uscita, questo metodo, ritorna un oggetto mapper di tipo `ISoapTypeMapper`.

- `Set myMapper = pFactory.getMapper(enXSDanyType, Nothing)`
  Questo metodo crea e ritorna un mapper per il tipo di dato specificato, che nel nostro caso è `enXSDanyType`. Con `enXSDanyType` si specifica che dovremo serializzare/deserializzare più tipi di dato XSD di base, nel nostro caso si tratta di
  ```

stringhe ed interi. Il secondo argomento (del tipo `IXMLDOMNode`) è un riferimento al nodo complexType nello schema se abbiamo a che fare con degli array. Quest'ultimo argomento può avere valore nullo, come nel nostro esempio.

- `Private Function ISoapTypeMapper_read(ByVal pNode As MSXML2.IXMLDOMNode, _`
  `ByVal bstrEncoding As String, ByVal encodingMode As _`
  `MSSOAPLib.enEncodingStyle, ByVal lFlags As Long) As Variant`
  Questo metodo, dell'interfaccia `ISoapTypeMapper`, permette di convertire (deserializzare) un nodo XML, presente all'interno di un messaggio SOAP, in un dato di tipo complesso. Il metodo prevede quattro argomenti in ingresso:
  - `pNode`, è il riferimento al nodo che contiene i dati da leggere.
  - `bstrEncoding`, è il valore dell'attributo `encodingStyle` specificato nel file WSDL.
  - `encodingMode`, è il valore degli attribute `style` ed `use` specificati nel file WSDL.
  - `lFlags`, argomento riservato.

  Come valore d'uscita, questo metodo, ritorna il valore deserializzato.

- `Dim Indirizzo As New Indirizzo`
  Istanzio un oggetto di tipo Indirizzo.

- `Dim Node As IXMLDOMNode`
  Istanzio un oggetto di tipo `IXMLDOMNode`.

- `Indirizzo.let_nomeVia = myMapper.read(pNode.selectSingleNode("nomeVia"), _`
  `bstrEncoding, encodingMode, lFlags)`
  Con il metoto `read` applicato all'oggetto mapper, si converte (deserializza) il sottonodo XML `nomeVia` presente nel messaggio SOAP. Questo valore, tramite il metodo `let_nomeVia`, viene assegnato al campo `nomeVia` di Indirizzo.

- `Set ISoapTypeMapper_read = Indirizzo`
  Assegno il contenuto di `Indirizzo` come valore di ritorno per il metodo `ISoapTypeMapper_read`, ossia i dati ottenuti dalla deserializzazione.

- `Private Function ISoapTypeMapper_varType() As Long`
  Questo metodo, che contiene l'istruzione `ISoapTypeMapper_varType = vbObject`, specifica il tipo di dato atteso dal metodo `write` (che vedremo fra breve) e ritornato dal metodo `read` (già visto).

- `Private Sub ISoapTypeMapper_write(ByVal pSoapSerializer As _`
  `MSSOAPLib.ISoapSerializer, ByVal bstrEncoding As String, ByVal _`
  `encodingMode As MSSOAPLib.enEncodingStyle, ByVal lFlags As Long, _`
  `pvar As Variant)`
  Questo metodo, dell'interfaccia `ISoapTypeMapper`, permette di convertire (serializzare) un dato di tipo complesso in un nodo XML, all'interno di un messaggio SOAP in uscita. Il metodo prevede cinque argomenti in ingresso:

- `pSoapSerializer`, è il riferimento serializzatore utilizzato per scrivere il nodo XML sul messaggio SOAP.
- `bstrEncoding`, è il valore dell'attributo `encodingStyle` specificato nel file WSDL.
- `encodingMode`, è il valore degli attribute `style` ed `use` specificati nel file WSDL.
- `lFlags`, argomento riservato.
- `pvar`, contiene il dato da serializzare.

- `Dim Indirizzo As Indirizzo`
  Istanzio un oggetto di tipo Indirizzo.

- `Set Indirizzo = pvar`
  Alla variabile `Indirizzo` viene assegnato il dato da serializzare.

- `pSoapSerializer.startElement "nomeVia"`

  ```
  myMapper.write pSoapSerializer, bstrEncoding,
  encodingMode,lFlags, Indirizzo.get_nomeVia

  pSoapSerializer.endElement
  ```

  Con questa serie di istruzioni serializzo il campo `nomeVia` di `Indirizzo`.

Server Indirizzo

Il server di quest'esempio riceve, dal client, il nome di una persone (una stringa) e ne restituisce il suo indirizzo (metodo `getAddressfromName`). Per semplicità, l'indirizzo restituito ha un valore standard che non è influenzato dal nome della persona.
Per poter serializzare il dato di tipo Indirizzo ed utilizzare alcuni dei suoi metodi, il server necessita di essere legato alle due classi precedentemente analizzate (ciò lo si può fare creando una libreria DLL opportuna).
Vediamo il codice del server:

```
Option Explicit

Public Function getAddressfromName(ByVal inputName As
String) As Indirizzo
    'ritorno sempre un indirizzo costante indipendentemente
dal inputName
    Dim IndirizzoObj1 As New Indirizzo
    IndirizzoObj1.let_nomeVia = "boccaccio"
    IndirizzoObj1.let_numCivico = 15
    IndirizzoObj1.let_CAP = 426
    IndirizzoObj1.let_citta = "Pavullo n/f"
    IndirizzoObj1.let_provincia = "Modena"

    Set getAddressfromName = IndirizzoObj1
End Function
```

A questo punto, attraverso lo strumento WSDL Generator di Microsoft SOAP Toolkit, si crea i documenti WSDL e WSML per il servizio implementato dal server appena visto.
In realtà i documenti generati automaticamente non contengono tutte le informazioni necessarie per gestire, via SOAP, un tipo di dato complesso com'è Indirizzo. Quindi, è

stato necessario aggiungere manualmente delle informazioni (segnalate in grassetto) ai due documenti.
Nel documento WSDL è stato necessario riportare la struttura del dato complesso Indirizzo, mentre nel documento WSML è stato necessario specificare la classe che definisce il mapper (serializzatore/deserializzatore) per lo stesso dato Indirizzo.

Documento WSDL:

```
<?xml version='1.0' encoding='UTF-8' ?>
 <!-- Generated 06/02/02 by Microsoft SOAP Toolkit WSDL File
Generator, Version 1.02.813.0 -->
<definitions  name ='Indirizzo'   targetNamespace =
'http://tempuri.org/wsdl/'
    xmlns:wsdlns='http://tempuri.org/wsdl/'
    xmlns:typens='http://tempuri.org/type'
    xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
    xmlns:xsd='http://www.w3.org/2001/XMLSchema'
    xmlns:stk='http://schemas.microsoft.com/soap-
toolkit/wsdl-extension'
    xmlns='http://schemas.xmlsoap.org/wsdl/'>
  <types>
    <schema targetNamespace='http://tempuri.org/type'
      xmlns='http://www.w3.org/2001/XMLSchema'
      xmlns:SOAP-
ENC='http://schemas.xmlsoap.org/soap/encoding/'
      xmlns:wsdl='http://schemas.xmlsoap.org/wsdl/'
      elementFormDefault='qualified'>
   <complexType name='Indirizzo'>
    <sequence>
     <element name='nomeVia' type='xsd:string'/>
     <element name='numCivico' type='xsd:int'/>
     <element name='CAP' type='xsd:int'/>
     <element name='citta' type='xsd:string'/>
     <element name='provincia' type='xsd:string'/>
    </sequence>
   </complexType>
    </schema>
  </types>
  <message name='server.getAddressfromName'>
    <part name='inputName' type='xsd:string'/>
  </message>
  <message name='server.getAddressfromNameResponse'>
    <part name='Result' type='typens:Indirizzo'/>
  </message>
  <portType name='serverSoapPort'>
    <operation name='getAddressfromName'
parameterOrder='inputName'>
      <input message='wsdlns:server.getAddressfromName' />
      <output
message='wsdlns:server.getAddressfromNameResponse' />
    </operation>
  </portType>
  <binding name='serverSoapBinding'
type='wsdlns:serverSoapPort' >
    <stk:binding preferredEncoding='UTF-8'/>
```

```
    <soap:binding style='rpc'
transport='http://schemas.xmlsoap.org/soap/http' />
    <operation name='getAddressfromName' >
      <soap:operation
soapAction='http://tempuri.org/action/server.getAddressfromN
ame' />
      <input>
        <soap:body use='encoded'
namespace='http://tempuri.org/message/'

encodingStyle='http://schemas.xmlsoap.org/soap/encoding/' />
      </input>
      <output>
        <soap:body use='encoded'
namespace='http://tempuri.org/message/'

encodingStyle='http://schemas.xmlsoap.org/soap/encoding/' />
      </output>
    </operation>
  </binding>
  <service name='Indirizzo' >
    <port name='serverSoapPort'
binding='wsdlns:serverSoapBinding' >
      <soap:address
location='http://localhost:8081/Indirizzo/Indirizzo.WSDL' />
    </port>
  </service>
</definitions>
```

Documento WSML :

```
<?xml version='1.0' encoding='UTF-8' ?>
 <!-- Generated 06/02/02 by Microsoft SOAP Toolkit WSDL File
Generator, Version 1.02.813.0 -->
<servicemapping name='Indirizzo'>
  <service name='Indirizzo'>
    <using PROGID='Indirizzo_server.server' cachable='0'
ID='serverObject' />
    <using PROGID='Indirizzo_Mapper.IndirizzoMapper'
cachable='0' ID='IndirizzoMap' />
    <types>
     <type name='Indirizzo'
           targetNamespace='http://tempuri.org/type'
           uses='IndirizzoMap' />
    </types>
    <port name='serverSoapPort'>
      <operation name='getAddressfromName'>
        <execute uses='serverObject'
method='getAddressfromName' dispID='1610809344'>
          <parameter callIndex='1' name='inputName'
elementName='inputName' />
          <parameter callIndex='-1' name='retval'
elementName='Result' />
        </execute>
      </operation>
    </port>
```

```
  </service>
</servicemapping>
```

Analizziamo meglio le modifiche sul documento WSML:

- **`<using PROGID='Indirizzo_Mapper.IndirizzoMapper' cachable='0' ID='IndirizzoMap' />`**
  In questo modo ci riferiamo alla classe COM che implementa il mapper (serializzatore/deserializzatore) per il tipo di dato complesso.

- **`<types>`**
  `<type name='Indirizzo'`
  `targetNamespace='http://tempuri.org/type'`
  `uses='IndirizzoMap' />`
  `</types>`
  In questo modo colleghiamo il mapper al suo tipo di dato complesso definito nel documento WSDL (sezione `types`).

Client Indirizzo

Il codice del client è il seguente:

```
Option Explicit
Private soapclient As soapclient
Private Sub Form_Load()
On Error GoTo fail
Set soapclient = New soapclient
soapclient.mssoapinit
"http://localhost:8081/Indirizzo/Indirizzo.wsdl", "", "",
"http://localhost/Indirizzo/Indirizzo.wsml"

Dim inputName As String
inputName = "pippo"
MsgBox "Richiesta al server dell'indirizzo di: " & inputName

Dim IndirizzoObjReturned As Indirizzo
Set IndirizzoObjReturned =
soapclient.getAddressfromName(inputName)
Dim strIndirizzo As String

strIndirizzo = "nomeVia: " &
IndirizzoObjReturned.get_nomeVia & vbCrLf & _
            "numCivico: " &
IndirizzoObjReturned.get_numCivico & vbCrLf & _
            "CAP: " & IndirizzoObjReturned.get_CAP & vbCrLf
& _
            "città: " & IndirizzoObjReturned.get_citta &
vbCrLf & _
            "provincia: " &
IndirizzoObjReturned.get_provincia & vbCrLf
MsgBox strIndirizzo
Exit Sub
fail:
MsgBox soapclient.detail

End Sub
```

Analizziamo le caratteristiche principali del client:

- `Private soapclient As soapclient`
  Fornisce sul lato client un'interfaccia "high-level" per inviare (attraverso metodi e proprietà opportune) una richiesta SOAP al server ed elaborare la risposta di ritorno.

- `Set soapclient = New soapclient`
  Creazione di un nuovo oggetto di tipo `soapclient`.

- `Client.mssoapinit WSDL`
  Tramite il metodo `mssoapinit` viene specificato a quale documento WSDL deve far riferimento l'oggetto di tipo `soapclient`.
  Il metodo in generale prevede quattro parametri:
  - L'URL del documento WSDL che descrive il servizio.
  - Il nome del tag di tipo `service` che contiene l'operazione specificata nella richiesta SOAP. Questo parametro è opzionale, infatti è necessario solo nel caso siano descritti più servizi nel documento WSDL. Se non viene specificato, di default, viene considerato il primo tag `service` del documento WSDL.
  - Il nome del tag di tipo `port` (figlio del tag indicato nel parametro precedente) che contiene l'operazione specificata nella richiesta SOAP. Questo parametro è opzionale, infatti è necessario solo nel caso siano descritti più operazioni nel documento WSDL. Se non viene specificato, di default, viene considerato il primo tag `service` del documento WSDL.
  - L'URL del documento WSML (Web Services Meta Language). Questo parametro è opzionale, infatti è necessario solo nel caso si utilizzano tipi di dato complessi definiti dallo sviluppatore in questo documento.

- `Dim IndirizzoObjReturned As Indirizzo`
  Istanza di un oggetto di tipo Indirizzo.

- `Set IndirizzoObjReturned = soapclient.getAddressfromName(inputName)`
  Qui si nota la potenza delle "high-level" API, infatti l'operazione remota (relativa ad un servizio SOAP) viene gestita come una semplice operazione locale. In particolare si invia all'operazione il nome di una persona (contenuto nella stringa `inputName`) la quale restituisce il suo indirizzo.

- `MsgBox soapclient.detail`
  In caso d'errore SOAP, una dialog box ci informa sui dettagli d'errore.

Messaggi SOAP di Indirizzo

Messaggio di richiesta inviato dal client:

```
POST /Indirizzo/Indirizzo.WSDL HTTP/1.1
Content-Type: text/xml; charset="UTF-8"
Host: localhost
SOAPAction:
"http://tempuri.org/action/server.getAddressfromName"
Content-Length: 382
```

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?><SOAP-
ENV:Envelope SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"><SOAP-
ENV:Body><SOAPSDK1:getAddressfromName
xmlns:SOAPSDK1="http://tempuri.org/message/"><inputName>pipp
o</inputName></SOAPSDK1:getAddressfromName></SOAP-
ENV:Body></SOAP-ENV:Envelope>
```

Messaggio di risposta inviato dal server:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Sun, 02 Jun 2002 12:35:50 GMT
Content-Type: text/xml; charset="UTF-8"
Content-Length: 1216
Expires: -1;

<?xml version="1.0" encoding="UTF-8" standalone="no"?><SOAP-
ENV:Envelope SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"><SOAP-
ENV:Body><SOAPSDK1:getAddressfromNameResponse
xmlns:SOAPSDK1="http://tempuri.org/message/"><Result><nomeVi
a xmlns:SOAPSDK2="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SOAPSDK3="http://www.w3.org/2001/XMLSchema"
SOAPSDK2:type="SOAPSDK3:string">boccaccio</nomeVia><numCivic
o xmlns:SOAPSDK4="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SOAPSDK5="http://www.w3.org/2001/XMLSchema"
SOAPSDK4:type="SOAPSDK5:short">15</numCivico><CAP
xmlns:SOAPSDK6="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SOAPSDK7="http://www.w3.org/2001/XMLSchema"
SOAPSDK6:type="SOAPSDK7:short">426</CAP><citta
xmlns:SOAPSDK8="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SOAPSDK9="http://www.w3.org/2001/XMLSchema"
SOAPSDK8:type="SOAPSDK9:string">Pavullo
n/f</citta><provincia
xmlns:SOAPSDK10="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SOAPSDK11="http://www.w3.org/2001/XMLSchema"
SOAPSDK10:type="SOAPSDK11:string">Modena</provincia></Result
></SOAPSDK1:getAddressfromNameResponse></SOAP-
ENV:Body></SOAP-ENV:Envelope>
```

Capitolo 9- L'ambiente Apache Axis

Caratteristiche generali

Apache Axis rappresenta l'evoluzione di Apache SOAP 2.2, intatti viene indicato come la terza generazione di Apache SOAP. Quindi nell'analizzare questa implementazione SOAP ci troveremo di fronte a molte caratteristiche, eventualmente arricchite, già studiate per Apache SOAP 2.2.
Attualmente Apache Axis è in fase di test ed è disponibile la versione beta 2.
Axis è essenzialmente un "motore" SOAP, una struttura per costruire processori SOAP come clients, servers, gateways, ecc. La versione attuale di Axis è scritta in Java, ma è allo studio un'implementazione C++ sul lato client.

Nella stesura (e realizzazione) del progetto Apache Axis, i vari committenti, hanno individuato alcuni concetti chiave:

- **Velocità.** Axis utilizza il parser SAX (come già faceva Apache SOAP 2.2) per elaborare con grande velocità i payload XML del messaggio SOAP.
- **Flessibilità.** L'architettura Axis fornisce allo sviluppatore completa libertà nell'inserire estensioni al "motore" di base così da personalizzare l'ambiente (es.: l'elaborazione degli header, la gestione del sistema, ecc.).
- **Stabilità.** Le classi e le interfacce presenti in Axis sono progettate come "published" (pubblicate), ciò significa che sono relativamente stabili. In altre parole utilizzando le interfacce pubblicate (a differenza di quelle "unpublished" definite dallo sviluppatore) si rendono minimi i problemi di migrazione verso nuove versioni di AXIS.
- **Pubblicazione orientata ai componenti.** Axis permette di definire facilmente reti riusabili di Handlers (classi richiamate da Axis all'arrivo di una richiesta o all'invio di una risposta) che implementino modelli comuni d'elaborazione per le proprie applicazioni, o per patners distribuiti.
- **Modello di trasporto libero.** Axis permette una semplice e trasparente astrazione sul modello di trasporto dei dati utilizzato, infatti, il cuore del "motore" è completamente indipendente dal protocollo di trasporto (sia esso HTTP, SMPT, FTP ed altri).
- **Supporto WSDL.** Axis supporta la specifica WSDL 1.1, la quale permette di costruire facilmente interfacce per l'accesso a servizi remoti pubblicati, e di esportare automaticamente una descrizione (machine-readable) per i servizi pubblicati con AXIS (assoluta novità rispetto ad Apache SOAP 2.2).

L'attuale versione di Apache Axis include caratteristiche già presenti Apache SOAP 2.2, come:

- Supporto completo delle specifiche SOAP 1.1 del W3C.
- Supporto per la serializzazione/deserializzazione per i tipi di dato basilari.
- Supporto per la serializzazione/deserializzazione per i tipi di dato definiti dallo sviluppatore.

- Serializzazione/deserializzazione automatica per Java Beans, ossia classi Java che forniscono i metodi `get/set` per accedere/definire ogni elemento/sottoelemento della classe stessa.
  Ciò avviene rispettivamente attraverso il serializzatore `org.apache.axis.encoding.ser.BeanSerializerFactory` ed il deserializzatore `org.apache.axis.encoding.ser.BeanDeserializerFactory`.
- Supporto per del modello RPC e di messaggistica SOAP.
- Supporto per sessioni orientate ai servizi, attraverso cookies HTTP oppure attraverso header SOAP indipendenti.
- Supporto per le specifiche "SOAP with Attachements", per l'invio di un documento MIME in allegato al messaggio SOAP.

A queste caratteristiche se ne aggiungono altre, che rappresentano le vere novità rispetto ad Apache SOAP 2.2:

- Supporto parziale delle specifiche SOAP 1.2 del W3C.
- Supporto per la pubblicazione automatica (Java Web Service) di servizi SOAP.
- Generazione automatica del file WSDL per i servizi pubblicati.
- WSDL2Java tool per costruire classi (interface, stub, skeleton, ecc.) Java a partire da documenti WSDL.
- Java2WSDL tool per costruire i corrispondenti documenti WSDL a partire da classi Java.

Esempio Add (web server Tomcat e client Apache Axis)

Nell'esempio `Add` il client invia al server due numeri interi, il compito del server è quello di restituire la somma dei due interi ricevuti.
Il server è stato sviluppato con Apache Axis e pubblicato sul web server Tomcat, anche il client è stato sviluppato con Apache Axis.
Questo semplice esempio ha lo scopo di introdurre lo studio dell'ambiente Apache Axis.

Server Add

Il codice del server è il seguente:

```
public class add_server{
  public int add(int i1, int i2)
  {
    return i1 + i2;
  }
}
```

Così come accadeva per Apache SOAP 2.2, il file sorgente Java del server non ha nulla di speciale che lo renda assimilabile ad un servizio web utilizzabile in remoto.
La prima novità di Apache Axis rispetto ad Apache SOAP 2.2 riguarda la pubblicazione di questo servizio.
Apache Axis consente di pubblicare automaticamente (Java Web Service) un servizio SOAP.
Per fare ciò sono sufficienti due semplici operazioni:

- copiare il file sorgente Java del server in una specifica sottodirectory del web server Tomcat (in particolare webapps\axis\);
- cambiare l'estensione del file sorgente da java a jws.

Con queste elementari operazioni il servizio è già pronto per accogliere le richieste dei client e fornirgli una risposta.

In Apache Axis esiste anche la possibilità di pubblicare un servizio attraverso un deployment descriptor utilizzando il formato Web Service Deployment Descriptor (WSDD). Questo tipo di pubblicazione è più flessibile rispetto a quella automatica, permettendo allo sviluppatore di specificare caratteristiche aggiuntive ed evolute per il servizio.
Il contenuto di un documento WSDD è paragonabile al contenuto di un documento XML utilizzato per il deploy di un servizio attraverso Apache SOAP 2.2.
In particolare, per l'esempio in questione, il deployment descriptor ha il seguente contenuto:

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

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

 <service name="add_service" provider="java:RPC">
  <parameter name="className" value="add_server"/>
  <parameter name="allowedMethods" value="add"/>
 </service>

</deployment>
```

La comprensione di questo documento è immediata:

- `deployment`
  L'elemento radice del deployment descriptor che informa il "motore" che si tratta del deploy di un servizio.
- `service name="add_service" provider="java:RPC"`
  Nome del servizio pubblicato e sua tipologia.
- `parameter name="className" value="add_server"`
  Parametro dell'elemento `service` che indica il nome della classe che implementa il servizio pubblicato.
- `parameter name="allowedMethods" value="add"`
  Parametro dell'elemento `service` che indica il nome dei metodi disponibili per il servizio pubblicato.

A questo punto si può effettuare fisicamente il deploy del servizio eseguendo, dal prompt comandi, la seguente istruzione:

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

In questo modo il servizio è utilizzabile attraverso SOAP all'indirizzo `http://localhost:8080/axis/services/add_service` (più in generale `http://<host>:<num_port>/axis/services/<nome_servizio>`), ovviamente con le opzioni del comando java è possibili indicare un indirizzo e un numero di porta differenti.

Nel caso sia necessario cancellare la pubblicazione del servizio, si deve creare un file WSDD opportuno:

```
<undeployment
    xmlns="http://xml.apache.org/axis/wsdd/">
  <service name="add_service"/>
</undeployment>
```

La comprensione di questo documento è immediata:

- `undeployment`
  L'elemento radice del deployment descriptor che informa il "motore" che si tratta del undeploy di un servizio, cioè della sua cancellazione .
- `service name="add_service"`
  Nome del servizio pubblicato in precedenza che si desidera cancellare.

A questo punto si può effettuare fisicamente la cancellazione del servizio eseguendo, dal prompt comandi, la seguente istruzione:

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

Client Add

Il codice sorgente Java del client è il seguente:

```
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;
import org.apache.axis.utils.Options;
import javax.xml.rpc.namespace.QName;
import javax.xml.rpc.ParameterMode;

public class add_client_jws
{
   public static void main(String [] args) throws Exception
{
       Options options = new Options(args);

       String endpoint = "http://localhost:" +
options.getPort() +
                         "/axis/add_server.jws";

       args = options.getRemainingArgs();

       if (args == null || args.length != 2) {
           System.err.println("Usage: add_client
arg1(integer) arg2(integer)");
           return;
       }


       Integer i1 = new Integer(args[0]);
       Integer i2 = new Integer(args[1]);

       Service  service = new Service();
       Call     call    = (Call) service.createCall();

       call.setTargetEndpointAddress( new
java.net.URL(endpoint) );
       call.setOperationName(new
QName("urn:add_service","add"));
       call.addParameter( "op1", XMLType.XSD_INT,
ParameterMode.PARAM_MODE_IN );
```

```
        call.addParameter( "op2", XMLType.XSD_INT,
ParameterMode.PARAM_MODE_IN );
        call.setReturnType( XMLType.XSD_INT );

        Integer ret = (Integer) call.invoke( new Object [] {
i1, i2 });

        System.out.println("Risultato somma: " + ret);
    }
}
```

Analizziamo il codice:

- `Options options = new Options(args);`
  `Options` è un oggetto della classe `org.apache.axis.utils.Option` utilizzato per ricavare/gestire i parametri e le opzioni indicate dall'utente (da linea di comando) al momento dell'invocazione del client.

- `String endpoint = "http://localhost:" + options.getPort() + "/axis/add_server.jws";`
  Il comando `options.getPort()` permette di ottenere l'eventuale numero di porta (su cui inviare la richiesta SOAP) specificato come opzione dal cliente.
  In questo caso la stringa `endpoint` fa riferimento all'URL del servizio pubblicato automaticamente attraverso Java Web Service.
  Nel caso si voglia richiamare il servizio pubblicato attraverso il deployment descriptor , la stringa `endpoint` assume il valore `"http://localhost:" + options.getPort() + "/axis/services/add_service"`.
  Per tutto il resto del codice, la chiamata ad un servizio pubblicato automaticamente o attraverso il deployment descriptor non influisce in nessun modo.

- `args = options.getRemainingArgs();`
  In questo modo si assegna all'array `args` gli args inutilizzati, ossia tutti gli args passati da linea di comando che non costituicono delle opzioni.

- `Service service = new Service();`
  L'oggetto `Service` appartiene alla classe `org.apache.axis.client.Service` e viene utilizzato come punto di partenza per l'accesso ad un servizio SOAP. Attraverso opportuni costruttori e metodi è possibile ricavare informazioni sul servizio direttamente dal corrispondente documento WSDL eventualmente fornito. In questo caso particolare si utilizza un costruttore senza parametri, questo indica che si è scelto di specificare manualmente tutti i campi opportuni, anziché acquisirli dal documento WSDL. Invece, per acquisire le informazioni sul servizio direttamente dal corrispondente documento WSDL, occorre utilizzare un altro costruttore che ha come parametro il percorso di memorizzazione del documento WSDL.

- `Call call = (Call) service.createCall();`
  L'oggetto `Call` appartiene alla classe `org.apache.axis.client.Call` e viene utilizzato per invocare un servizio; può essere riempito riferendosi al contenuto di un documento WSDL (se è stato precedentemente utilizzato l'opportuno costruttore sull'oggetto `Service`) oppure essere riempito manualmente. Nel caso in esame, attraverso il metodo `createCall()`

applicato all'oggetto di tipo `Service`, si crea un nuovo oggetto di tipo `Call` vuoto.

- `call.setTargetEndpointAddress( new java.net.URL(endpoint) );`
  In questo modo si imposta l'URL del servizio.

- `call.setOperationName(new QName("urn:add_service","add"));`
  In questo modo si imposta il metodo da richiedere al servizio; con questa istruzione non viene valutato se nel documento WSDL (eventualmente presente) relativo al servizio, esiste l'operazione richiesta.
  Il metodo `setOperationName` prevede come parametro un oggetto di tipo `QName`, oppure una stringa che viene automaticamente convertita in un oggetto di tipo `QName`.
  Un oggetto di tipo `QName` (rappresentabile come QName::=(Prefix':')?LocalPart) appartiene alla classe `javax.xml.rpc.namespace.QName`, e rappresenta un nome qualificato basato sulle specifiche di XML Namespace.
  In genere si utilizza un costruttore che prevede due parametri: il primo è l'URI del namespace, il secondo una stringa che indica la LocalPart (nel nostro caso il nome del metodo da invocare).

- `call.addParameter( "op1", XMLType.XSD_INT, ParameterMode.PARAM_MODE_IN );`
  Aggiunge il parametro specificato alla lista dei parametri da inviare al metodo associato all'oggetto di tipo `Call` in questione.
  Il metodo `call.addParameter` prevede tre argomenti:
  - una stringa (o più in generale un oggetto `QName`) che indica il nome dato al parametro nel payload XML del messaggio SOAP;
  - un oggetto di tipo `QName` che indica il tipo di parametro, in questo caso si tratta di un intero (`XSD_INT` è un campo di tipo `QName` dell'oggetto `XMLType` appartenente alla classe `org.apache.axis.encoding.XMLType`). Successivamente (per effetto delle assegnazioni appena fatte) analizzando il payload XML del messaggio SOAP di richiesta, sarà presente l'elemento `<op1 xsi:type="xsd:int">`.
  - un oggetto di tipo `ParameterMode` appartenente alla classe `javax.xml.rpc.ParameterMode`, che specifica il tipo di parametro inviato. Tale oggetto può assumere tre valori: `PARAM_MODE_IN`, `PARAM_MODE_OUT` oppure `PARAM_MODE_INOUT`.

  Occorre notare che senza l'istruzione appena vista, Axis automaticamente serializza i parametri della chiamata SOAP (con il nome `arg0`, `arg1`, ecc.), ed indica il loro tipo. Per fare ciò si basa sui valore dei parametri che, come vedremo fra breve, sono passati al metodo `call.invoke`.

- `call.setReturnType( XMLType.XSD_INT );`
  Con questo metodo (che ha come parametro un oggetto di tipo `QName`) si definisce il tipo del risultato di ritorno presente nella risposta SOAP. Normalmente Axis recupera questo tipo, quando tale indicazione è presente, direttamente dal messaggio di risposta SOAP. Nel caso non sia presente tale

informazione, con l'istruzione appena vista, Axis considera il risultato di ritorno del tipo indicato manualmente.

- `Integer ret = (Integer) call.invoke( new Object [] { i1, i2 });`
  Con il metodo `invoke` si invoca il metodo precedentemente specificato per oggetto di tipo `Call`, passandogli come parametri gli argomenti del metodo stesso. Il valore di ritorno di tale metodo è un oggetto di tipo `Object` che rappresenta il risultato del operazione richiesta al servizio SOAP. Avere un oggetto di tipo `Object`, sia come argomento sia come valore di ritorno del metodo `invoke`, rende la programmazione molto flessibile.

Messaggi SOAP di Add

A questo punto valutiamo cosa succede a livello di richiesta e risposta SOAP. In realtà questi messaggi sono paragonabili a quelli incontrati nell'implementazione Apache SOAP 2.2.
Il messaggio di richiesta inviato dal client è il seguente:

```
POST /axis/add_server.jws HTTP/1.0
Content-Length: 521
Host: localhost
Content-Type: text/xml; charset=utf-8
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
 <SOAP-ENV:Body>
  <ns1:add xmlns:ns1="urn:add_service">
   <op1 xsi:type="xsd:int">2</op1>
   <op2 xsi:type="xsd:int">3</op2>
  </ns1:add>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Il messaggio di risposta inviato dal server è il seguente:

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 455
Servlet-Engine: Tomcat Web Server/3.2.3 (JSP 1.1; Servlet
2.2; Java 1.4.0; Windows XP 5.1 x86; java.vendor=Sun
Microsystems Inc.)

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

```
  <ns1:addResponse SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
" xmlns:ns1="urn:add_service">
   <addResult xsi:type="xsd:int">5</addResult>
  </ns1:addResponse>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Come si può notare il risultato di ritorno si trova nell'elemento `addResult` (nome del metodo invocato concatenato alla stringa Result).

Supporto WSDL con Axis

La specifica WSDL (Web Service Description Language) nasce dalla collaborazione di Microsoft ed IBM, ma è adottata da molte altre organizzazioni.
WSDL permette di descrivere un servizio web in un modo strutturato, indicando l'interfaccia del servizio, i tipi di dato utilizzati, e dove sono localizzati.

Axis supporta il formato WSDL in tre modi:

- Una volta eseguito in Axis il deploy del servizio, richiamando da un browser web il servizio con il proprio URL concatenato al suffisso "?WSDL", è possibile ottenere la generazione automatica (e la visualizzazione) del documento WSDL che descrive il sevizio.
- Utilizzando il tool WSDL2Java vengono costruite automaticamente tutte le classi Java per il servizio descritto dal documento WSDL indicato.
  Il tool prevede due modalità di funzionamento:
  - Lato client, costruzione di tutti gli oggetti necessari per accedere al servizio.
  - Lato server, costruzione di tutti gli oggetti necessari per implementare il servizio.
- Utilizzando il tool Java2WSDL viene costruito automaticamente il documento WSDL relativo alla classe/interfaccia Java indicata.

Tool Java2WSDL

Come già detto, con il tool Java2WSDL siamo in grado di ottenere automaticamente il documento WSDL che descrive un servizio.
L'idea è di costruire il documento WSDL associato al servizio Add (pubblicato attraverso Axis), e di avere un client Visual Basic che acceda al servizio utilizzando le "high-level" API (supportate da Microsoft SOAP Toolkit). Quindi, si vuole, verificare le potenzialità dlla specifica WSDL.

Digitando dal prompt comandi l'istruzione

```
java org.apache.axis.wsdl.Java2WSDL –o add.wsdl –l
"http://localhost:8080/axis/services/add_service" -n
"urn:add_service" add_server
```

si ottiene il documento WSDL relativo alla classe `add_server`.

Le opzioni principali per il tool Java2WSDL sono le seguenti:

- `-o` specifica dove memorizzare il documento WSDL;
- `-l` specifica, all'interno del documento WSDL, dove è pubblicato il servizio;
- `-n` specifica, all'interno del documento WSDL, il namespace utilizzato.

Il contenuto del documento WSDL è il seguente:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="urn:add_service"
xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:impl="urn:add_service-impl"
xmlns:intf="urn:add_service"
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="addRequest">

      <wsdl:part name="in0" type="xsd:int"/>

      <wsdl:part name="in1" type="xsd:int"/>

   </wsdl:message>

   <wsdl:message name="addResponse">

      <wsdl:part name="return" type="xsd:int"/>

   </wsdl:message>

   <wsdl:portType name="add_server">

      <wsdl:operation name="add" parameterOrder="in0 in1">

         <wsdl:input message="intf:addRequest"/>

         <wsdl:output message="intf:addResponse"/>

      </wsdl:operation>

   </wsdl:portType>

   <wsdl:binding name="add_serverSoapBinding"
type="intf:add_server">

      <wsdlsoap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>

      <wsdl:operation name="add">

         <wsdlsoap:operation soapAction=""/>

         <wsdl:input>

            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:add_service" use="encoded"/>

         </wsdl:input>
```

```
         <wsdl:output>

            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:add_service" use="encoded"/>

         </wsdl:output>

      </wsdl:operation>

   </wsdl:binding>

   <wsdl:service name="add_serverService">

      <wsdl:port binding="intf:add_serverSoapBinding"
name="add_server">

         <wsdlsoap:address
location="http://localhost:8080/axis/services/add_server"/>

      </wsdl:port>

   </wsdl:service>

</wsdl:definitions>
```

Il documento WSDL risulta facilmente comprensibile, in particolare:

- Il tag `message` di nome `addRequest` contiene due tag figli di nome `in0` ed `in1` entrambi di tipo `int`. Si specifica che il messaggio di richiesta è formato da due parametri di tipo intero.
- il tag `message` di nome `addResponse` contiene un tag figlio di nome `return` di tipo `int`. Si specifica che il messaggio di risposta contiene un risultato di tipo intero.
- Il tag `portType` di nome `add_server` specifica che l'operazione `add` è oggetto dei messaggi di richiesta e risposta sopra descritti.
- Il tag `binding` di nome `add_serverSoapBinding` definisce il protocollo utilizzato per il `portType add_server`: si tratta di una `rpc SOAP` attraverso `HTTP`. Inoltre si specifica il namespace e l'encodingStyle utilizzati nel Body dei messaggi di richiesta e risposta.
- Il tag `service` di nome `add_serverService` specifica l'indirizzo a cui trovare il `binding add_serverSoapBinding`.

Il codice Visual Basic del client è il seguente:

```
Option Explicit

Private Client As SoapClient
Private sConnectedWSDL As String


Private Sub cmd_add_Click()

    On Error GoTo ErrorHandler
    Me.MousePointer = vbHourglass
```

```
    Connect
    risultato.Text = Client.Add(addendo_1.Text,
addendo_2.Text)
    Me.MousePointer = vbDefault
    Exit Sub

ErrorHandler:

    Me.MousePointer = vbDefault
    MsgBox Client.faultstring, vbExclamation

End Sub


Private Sub Connect()

    Dim WSDL
    WSDL = "http:\\localhost\my_axis\Add\add.wsdl"

    If sConnectedWSDL <> WSDL Then
        Set Client = New SoapClient
        Client.mssoapinit WSDL
        sConnectedWSDL = WSDL
    End If

End Sub
```

Appare subito chiaro, che utilizzando le specifiche WSDL, la costruzione di un client è estremamente semplice. Per rendercene conto meglio possiamo far riferimento all'esempio `Add` incontrato nel capitolo precedente. Siccome Apache SOAP 2.2 non forniva il supporto WSDL sui servizi pubblicati, siamo stati costretti a costruirci un client utilizzando le "low-level" API; quindi a conosce a fondo le specifiche sul trasporto, la serializzazione e la deserializzazione dei messaggi scambiati.
Grazie al supporto WSDL fornito da Axis, tutto quello che dobbiamo fare è indicare dove si trova memorizzato il documento WSDL. Tutto il resto viene gestito automaticamente, con tutti i benefici del caso.
Il client in esame richiede all'utente, tramite una dialog box, di inserire i due addendi da sommare, dopodiché invia la richiesta al server, attende la risposta e visualizza il risultato di tale somma.
Analizziamo le caratteristiche principali del client:

- `Private Client As SoapClient`
  Fornisce sul lato client un'interfaccia "high-level" per inviare (attraverso metodi e proprietà opportune) una richiesta SOAP al server ed elaborare la risposta di ritorno.

- `Connect`
  Compito di questa procedura è di accedere al documento WSDL che descrive il servizio. Di seguito sono riportati alcuni dei suoi elementi:
  - `Dim WSDL`
    `WSDL = "http:\\localhost\my_axis\Add\add.wsdl"`
    Percorso (URL) del documento WSDL.
  - `Set Client = New SoapClient`
    Creazione di un nuovo oggetto di tipo `SoapClient`.
  - `Client.mssoapinit WSDL`

Tramite il metodo `mssoapinit` viene specificato a quale documento WSDL deve far riferimento l'oggetto di tipo `SoapClient`.
Il metodo in generale prevede quattro parametri:

- L'URL del documento WSDL che descrive il servizio.
- Il nome del tag di tipo `service` che contiene l'operazione specificata nella richiesta SOAP. Questo parametro è opzionale, infatti è necessario solo nel caso siano descritti più servizi nel documento WSDL. Se non viene specificato, di default, viene considerato il primo tag `service` del documento WSDL.
- Il nome del tag di tipo `port` (figlio del tag indicato nel parametro precedente) che contiene l'operazione specificata nella richiesta SOAP. Questo parametro è opzionale, infatti è necessario solo nel caso siano descritti più operazioni nel documento WSDL. Se non viene specificato, di default, viene considerato il primo tag `service` del documento WSDL.
- L'URL del documento WSML (Web Services Meta Language). Questo parametro è opzionale, infatti è necessario solo nel caso si utilizzano tipi di dato complessi definiti dallo sviluppatore in questo documento.

- `risultato.Text = Client.Add(addendo_1.Text, addendo_2.Text)`
  Qui si nota la potenza delle "high-level" API, infatti l'operazione remota (relativa ad un servizio SOAP) viene gestita come una semplice operazione locale. In particolare l'operazione acquisisce come parametri i due addendi raccolti dalla dialog box e restituisce la somma sulla stessa dialog box.

- `ErrorHandler`
  Questa procedura ha il compito di visualizzare, tramite una dialog box, eventuali errori verificatesi nell'accesso al servizio. In particolare stampa la `faultstring` (`Client.faultstring`) relativa all'errore.

Il messaggio di richiesta inviato dal clientVisual Basic è il seguente:

```
POST /axis/add_server.jws HTTP/1.1
Content-Type: text/xml; charset="UTF-8"
Host: localhost
SOAPAction: ""
Content-Length: 336

<?xml version="1.0" encoding="UTF-8" standalone="no"?><SOAP-
ENV:Envelope SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"><SOAP-
ENV:Body><SOAPSDK1:add
xmlns:SOAPSDK1="urn:add_service"><in0>1</in0><in1>2</in1></S
OAPSDK1:add></SOAP-ENV:Body></SOAP-ENV:Envelope>
```

Il messaggio di risposta inviato dal server Apache Axis è il seguente:

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

```
Content-Length: 455
Servlet-Engine: Tomcat Web Server/3.2.3 (JSP 1.1; Servlet
2.2; Java 1.4.0; Windows XP 5.1 x86; java.vendor=Sun
Microsystems Inc.)

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <SOAP-ENV:Body>
  <ns1:addResponse SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
" xmlns:ns1="urn:add_service">
   <addResult xsi:type="xsd:int">3</addResult>
  </ns1:addResponse>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Entrambi i messaggi, per quanto visto finora, sono d'immediata comprensione.

Tool WSDL2Java

Nel seguito si farà riferimento, per illustrare il funzionamento del tool WSDL2Java, al seguente file `Indirizzo.wsdl`:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="urn:Indirizzo_servizio"
xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:impl="urn:Indirizzo_servizio-impl"
xmlns:intf="urn:Indirizzo_servizio"
xmlns:tns1="http://Indirizzo.my_axis.samples"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <types>
  <schema targetNamespace="http://Indirizzo.my_axis.samples"
xmlns="http://www.w3.org/2001/XMLSchema">
   <complexType name="Indirizzo">
    <sequence>
     <element name="nome_via" nillable="true"
type="xsd:string"/>
     <element name="num_civico" type="xsd:int"/>
     <element name="CAP" type="xsd:int"/>
     <element name="citta" nillable="true"
type="xsd:string"/>
     <element name="provincia" nillable="true"
type="xsd:string"/>
    </sequence>
   </complexType>
   <element name="Indirizzo" nillable="true"
type="tns1:Indirizzo"/>
  </schema>
 </types>
```

```
   <wsdl:message name="getAddressfromNameResponse">

      <wsdl:part name="return" type="tns1:Indirizzo"/>

   </wsdl:message>

   <wsdl:message name="getAddressfromNameRequest">

      <wsdl:part name="in0" type="SOAP-ENC:string"/>

   </wsdl:message>

   <wsdl:portType name="Indirizzo_server">

      <wsdl:operation name="getAddressfromName"
parameterOrder="in0">

         <wsdl:input
message="intf:getAddressfromNameRequest"/>

         <wsdl:output
message="intf:getAddressfromNameResponse"/>

      </wsdl:operation>

   </wsdl:portType>

   <wsdl:binding name="Indirizzo_servizioSoapBinding"
type="intf:Indirizzo_server">

      <wsdlsoap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>

      <wsdl:operation name="getAddressfromName">

         <wsdlsoap:operation soapAction=""/>

         <wsdl:input>

            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:Indirizzo_servizio" use="encoded"/>

         </wsdl:input>

         <wsdl:output>

            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:Indirizzo_servizio" use="encoded"/>

         </wsdl:output>

      </wsdl:operation>
```

```
   </wsdl:binding>

   <wsdl:service name="Indirizzo_serverService">

      <wsdl:port
binding="intf:Indirizzo_servizioSoapBinding"
name="Indirizzo_servizio">

         <wsdlsoap:address
location="http://localhost:8080/axis/services/Indirizzo_serv
izio"/>

      </wsdl:port>

   </wsdl:service>

</wsdl:definitions>
```

In realtà, questo file WSDL, è stato ottenuto utilizzando il tool Java2WSDL sul seguente documento Java:

```
package samples.my_axis.Indirizzo;

import java.io.*;
import java.util.*;

public class Indirizzo_server
{

  public Indirizzo getAddressfromName(String st)
   {  Indirizzo ind= new Indirizzo("via
giardini",15,41028,"Serramazzoni","Modena");
            return (Indirizzo) ind;

   }
}
```

Si tratta del codice che descrive un server, il quale riceve il nome di una persona e restituisce il suo indirizzo (o meglio, per comodità, un indirizzo prefissatto).
Questo servizio prevede lo scambio di un dato di tipo complesso (`Indirizzo`) e non più un dato di tipo semplice (interi, reali, stringhe, ecc.). Per quanto detto, sarà necessario definire ad hoc un serializzatore ed un deserializzatore che facciano riferimento al dato di tipo complesso.

Il dato di tipo complesso `Indirizzo` è definito nella classe `samples.my_axis.Indirizzo.Indirizzo`, che ha il seguente contenuto:

```
package samples.my_axis.Indirizzo;

public class Indirizzo{

   public String    nome_via;
   public int num_civico;
   public int CAP;
   public String    citta;
   public String    provincia;
```

```
   //costruttori
   public Indirizzo() { }

   public Indirizzo(String nome_via,int num_civico,int
CAP,String citta,String provincia) {
         this.nome_via=nome_via;
         this.num_civico=num_civico;
         this.CAP=CAP;
         this.citta=citta;
         this.provincia=provincia;

   }

}
```

A questo punto andiamo ad analizzare il tool WSDL2Java.

Lato client

L'esecuzione di base avviene digitando, dal prompt comandi, la seguente istruzione:
`java org.apache.axis.wsdl.WSDL2Java URL-documento-WSDL`
in questo modo vengono generati, seguendo le specifiche JAX-RPC, tutti gli elementi (le classi) necessari al client per accedere al servizio descritto dal documento WSDL.

In particolare la seguente tabella specifica le classi generate per ogni sezione presente nel documento WSDL:

Sezione WSDL	**Classe/i Java generate**
Per ogni entry nella sezione types	Una classe Java
	Una classe Holder (solo se questo elemento della entry types è usato come parametro d'ingresso/uscita)
Per ogni entry nella sezione portType	Un'interfaccia Java (SDI)
Per ogni entry nella sezione binding	Una classe stub (che implementa l'interfaccia SDI)
Per ogni entry nella sezione service	Un'interfaccia service
	Una classe service locator (che implementa l'interfaccia service)

Types

Nella sezione types di un documento WSDL trovano posto le definizioni di tutti i dati complessi (quindi non di base) creati dallo sviluppatore.
Le classi generate per ogni entry di tipo types prendono il nome della stessa entry.

Per esempio, dato il seguente spezzone WSDL:

```
<types>
```

```
  <schema targetNamespace="http://Indirizzo.my_axis.samples"
xmlns="http://www.w3.org/2001/XMLSchema">
   <complexType name="Indirizzo">
    <sequence>
     <element name="nome_via" nillable="true"
type="xsd:string"/>
     <element name="num_civico" type="xsd:int"/>
     <element name="CAP" type="xsd:int"/>
     <element name="citta" nillable="true"
type="xsd:string"/>
     <element name="provincia" nillable="true"
type="xsd:string"/>
    </sequence>
   </complexType>
   <element name="Indirizzo" nillable="true"
type="tns1:Indirizzo"/>
  </schema>
 </types>
```

Il toolWSDL2Java genererà il seguente documento Java:

```
/**
 * Indirizzo.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis Wsdl2java emitter.
 */

package samples.my_axis.Indirizzo;

public class Indirizzo implements java.io.Serializable {
    private java.lang.String nomeVia;
    private int numCivico;
    private int CAP;
    private java.lang.String citta;  // attribute
    private java.lang.String provincia;  // attribute

    public Indirizzo() {
    }

    public java.lang.String getNomeVia() {
        return nomeVia;
    }

    public void setNomeVia(java.lang.String nomeVia) {
        this.nomeVia = nomeVia;
    }

    public int getNumCivico() {
        return numCivico;
    }

    public void setNumCivico(int numCivico) {
        this.numCivico = numCivico;
    }
```

```
    public int getCAP() {
        return CAP;
    }

    public void setCAP(int CAP) {
        this.CAP = CAP;
    }

    public java.lang.String getCitta() {
        return citta;
    }

    public void setCitta(java.lang.String citta) {
        this.citta = citta;
    }

    public java.lang.String getProvincia() {
        return provincia;
    }

    public void setProvincia(java.lang.String provincia) {
        this.provincia = provincia;
    }

    // Type metadata
    private static org.apache.axis.description.TypeDesc
typeDesc =
        new org.apache.axis.description.TypeDesc();

    static {
        org.apache.axis.description.FieldDesc field = new
org.apache.axis.description.ElementDesc();
        field.setFieldName("numCivico");
        field.setXmlName(new
javax.xml.rpc.namespace.QName("http://Indirizzo.my_axis.samp
les", "num_civico"));
        typeDesc.addFieldDesc(field);
        field = new
org.apache.axis.description.ElementDesc();
        field.setFieldName("nomeVia");
        field.setXmlName(new
javax.xml.rpc.namespace.QName("http://Indirizzo.my_axis.samp
les", "nome_via"));
        typeDesc.addFieldDesc(field);
    };

    /**
     * Return type metadata object
     */
    public static org.apache.axis.description.TypeDesc
getTypeDesc() {
        return typeDesc;
    }

    public boolean equals(Object obj) {
        // compare elements
```

```
        Indirizzo other = (Indirizzo) obj;
        if (obj == null) return false;
        if (this == obj) return true;
        if (! (obj instanceof Indirizzo)) return false;
        return
            ((nomeVia==null && other.getNomeVia()==null) ||
             (nomeVia!=null &&
              nomeVia.equals(other.getNomeVia()))) &&
            numCivico == other.getNumCivico() &&
            CAP == other.getCAP() &&
            ((citta==null && other.getCitta()==null) ||
             (citta!=null &&
              citta.equals(other.getCitta()))) &&
            ((provincia==null && other.getProvincia()==null)
||
             (provincia!=null &&
              provincia.equals(other.getProvincia())));
    }
}
```

In pratica vengono generate i seguenti elementi:

- La classe `Indirizzo` (o meglio `samples.my_axis.Indirizzo.Indirizzo`) con il rispettivo costruttore.
- Tanti metodi `set?` per settare singolarmente il valore di ogni attributo presente nella classe.
- Tanti metodi `get?` per ottenere singolarmente il valore di ogni attributo presente nella classe.
- Una sezione `Type metadata` per supportare il mapping fra XML e Java, ed abilitare la serializzazione/deserializzazione degli attributi XML.
    - `private static org.apache.axis.description.TypeDesc typeDesc= new org.apache.axis.description.TypeDesc();`
      Un oggetto `TypeDesc` rappresenta un legame fra dati XML e dati Java. È essenzialmente una collezione di `FieldDesc` che descrivono come mappare ogni campo XML in un campo Java (e viceversa).
    - `org.apache.axis.description.FieldDesc field = new org.apache.axis.description.ElementDesc();`
      Come anticipato, `FieldDesc` rappresenta un oggetto metadata con il quale controllare il mapping da campi XML a campi Java (e viceversa).
    - `field.setFieldName("numCivico");`
      Il metodo `setFieldName` applicato ad un oggetto di tipo `FieldDesc`, imposta il nome del campo.
    - `field.setXmlName(new javax.xml.rpc.namespace.QName("http://Indirizzo.my_axis.samples", "num_civico"));`
      Il metodo `setXMLName` applicato ad un oggetto di tipo `FieldDesc`, imposta il `QName` (passato come parametro) per il campo. Come già detto in precedenza, un oggetto di tipo `QName` rappresenta un nome qualificato basato sulla specifica XML Namespace.
    - `typeDesc.addFieldDesc(field);`
      Il metodo `addFieldDesc` applicato ad un oggetto di tipo `TypeDesc`, aggiunge un nuovo campo di tipo `FieldDesc`.

- Una sezione `Return type metadata object` contiene un metodo che ritorna l'oggetto di tipo `TypeDesc` e un metodo che confronta gli elementi di un oggetto.

Nel caso in cui l'oggetto della entry `types` si fosse chiamato `indirizzo` (anziché `Indirizzo`), per convenzione, la classe creata si sarebbe comunque chiamata `Indirizzo` (con la prima lettera maiuscola).
Inoltre, si deve tener presente che XML è meno restrittivo rispetto a Java riguardo l'assegnazione dei nomi ai propri componenti. Per questo motivo, se `Indirizzo` avesse contenuto un sottoelemento di nome `new`, essendo questa una parola Java riservata, la compilazione del codice generato sarebbe fallita.

Holders

Una classe holder viene creata solo quando l'elemento della entry `types` è contemporaneamente un parametro d'ingresso e d'uscita. Questo è necessario, nel rispetto delle specifiche JAX-RPC, perché Java non contempla il concetto di parametri d'ingresso ed uscita.
Una classe holder è semplicemente una classe che contiene un'istanza dell'elemento definito nella entry `types`.
Ad esempio, se la classe `Indirizzo` fosse un parametro d'ingresso/uscita, verrebbe generata la seguente classe holder:

```
public final class IndirizzoHolder implements
javax.xml.rpc.holders.Holder
{
 public Indirizzo value;

 public IndirizzoHolder(){}

 public IndirizzoHolder (Indirizzo value)
  { this.value=value }
}
```

Le classi holder per dati di tipo primitivo si trovano nel package `javax.xml.rpc.holders`.

PortTypes

La Service Definition Interface (SDI) è un'interfaccia che deriva da un elemento della entry `portType` del documento WSDL. Quest'interfaccia è utilizzata per accedere alle operazioni del servizio.
Ad esempio, dato il seguente spezzone WSDL:

```
<wsdl:message name="getAddressfromNameResponse">

   <wsdl:part name="return" type="tns1:Indirizzo"/>

</wsdl:message>

<wsdl:message name="getAddressfromNameRequest">

   <wsdl:part name="in0" type="SOAP-ENC:string"/>

</wsdl:message>
```

```
   <wsdl:portType name="Indirizzo_server">

      <wsdl:operation name="getAddressfromName"
parameterOrder="in0">

         <wsdl:input
message="intf:getAddressfromNameRequest"/>

         <wsdl:output
message="intf:getAddressfromNameResponse"/>

      </wsdl:operation>

   </wsdl:portType>
```

WSDL2Java genererà l'interfaccia SDI:

```
/**
 * IndirizzoServer.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis Wsdl2java emitter.
 */

package Indirizzo_servizio;

public interface IndirizzoServer extends java.rmi.Remote {
    public samples.my_axis.Indirizzo.Indirizzo
getAddressfromName(java.lang.String in0) throws
java.rmi.RemoteException;
}
```

Il nome dell'interfaccia SDI è tipicamente il nome dell'elemento `portType`.
Comunque, la costruzione dell'interfaccia SDI, necessita di informazioni contenute sia nel `portType` che nel `binding` del documento WSDL. Infatti se ad un elemento `portType` sono associati due diversi elementi `binding`, non possiamo utilizzare una sola interfaccia per entrambi i `binding`. È necessario avere due interfacce e due stub (vedremo fra breve), in questo caso anche i nomi faranno riferimento al nome del `binding`.

Bindings

Una classe stub implementa un'interfaccia SDI. Il suo nome è il nome del corrispondente `binding` seguito dal suffisso `Stub`. Questa classe contiene il codice per effettuare l'invocazione di un metodo nella chiamata SOAP, utilizzando gli oggetti `Service` e `Call` di Axis. La classe stub, visto il suo funzionamento, può prendere il nome di classe proxy.
Ad esempio, dato il seguente spezzone WSDL:

```
<wsdl:binding name="Indirizzo_servizioSoapBinding"
type="intf:Indirizzo_server">

      <wsdlsoap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
```

```
      <wsdl:operation name="getAddressfromName">

         <wsdlsoap:operation soapAction=""/>

         <wsdl:input>

            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:Indirizzo_servizio" use="encoded"/>

         </wsdl:input>

         <wsdl:output>

            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:Indirizzo_servizio" use="encoded"/>

         </wsdl:output>

      </wsdl:operation>

   </wsdl:binding>
```

WSDL2Java genererà la seguente classe stub:

```
/**
 * IndirizzoServizioSoapBindingStub.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis Wsdl2java emitter.
 */

package Indirizzo_servizio;

public class IndirizzoServizioSoapBindingStub extends
org.apache.axis.client.Stub implements
Indirizzo_servizio.IndirizzoServer {
    private java.util.Vector cachedSerClasses = new
java.util.Vector();
    private java.util.Vector cachedSerQNames = new
java.util.Vector();
    private java.util.Vector cachedSerFactories = new
java.util.Vector();
    private java.util.Vector cachedDeserFactories = new
java.util.Vector();

    public IndirizzoServizioSoapBindingStub() throws
org.apache.axis.AxisFault {
         this(null);
    }

    public IndirizzoServizioSoapBindingStub(java.net.URL
endpointURL, javax.xml.rpc.Service service) throws
org.apache.axis.AxisFault {
         this(service);
```

```
         super.cachedEndpoint = endpointURL;
    }

    public
IndirizzoServizioSoapBindingStub(javax.xml.rpc.Service
service) throws org.apache.axis.AxisFault {
        try {
            if (service == null) {
                super.service = new
org.apache.axis.client.Service();
            } else {
                super.service = service;
            }
            Class cls;
            javax.xml.rpc.namespace.QName qName;
            Class beansf =
org.apache.axis.encoding.ser.BeanSerializerFactory.class;
            Class beandf =
org.apache.axis.encoding.ser.BeanDeserializerFactory.class;
            Class enumsf =
org.apache.axis.encoding.ser.EnumSerializerFactory.class;
            Class enumdf =
org.apache.axis.encoding.ser.EnumDeserializerFactory.class;
            Class arraysf =
org.apache.axis.encoding.ser.ArraySerializerFactory.class;
            Class arraydf =
org.apache.axis.encoding.ser.ArrayDeserializerFactory.class;
            Class simplesf =
org.apache.axis.encoding.ser.SimpleNonPrimitiveSerializerFac
tory.class;
            Class simpledf =
org.apache.axis.encoding.ser.SimpleDeserializerFactory.class
;
            qName = new
javax.xml.rpc.namespace.QName("http://Indirizzo.my_axis.samp
les", "Indirizzo");
            cachedSerQNames.add(qName);
            cls = samples.my_axis.Indirizzo.Indirizzo.class;
            cachedSerClasses.add(cls);
            cachedSerFactories.add(beansf);
            cachedDeserFactories.add(beandf);

        }
        catch(Exception t) {
            throw org.apache.axis.AxisFault.makeFault(t);
        }
    }

    private org.apache.axis.client.Call getCall() throws
java.rmi.RemoteException {
        try {
            org.apache.axis.client.Call call =
                    (org.apache.axis.client.Call)
super.service.createCall();
            if (super.maintainSessionSet) {
```

```
call.setMaintainSession(super.maintainSession);
            }
            if (super.cachedUsername != null) {
                call.setUsername(super.cachedUsername);
            }
            if (super.cachedPassword != null) {
                call.setPassword(super.cachedPassword);
            }
            if (super.cachedEndpoint != null) {

call.setTargetEndpointAddress(super.cachedEndpoint);
            }
            if (super.cachedTimeout != null) {
                call.setTimeout(super.cachedTimeout);
            }
            java.util.Enumeration keys =
super.cachedProperties.keys();
            while (keys.hasMoreElements()) {
                String key = (String) keys.nextElement();
                call.setProperty(key,
super.cachedProperties.get(key));
            }
            // All the type mapping information is
registered
            // when the first call is made.
            // The type mapping information is actually
registered in
            // the TypeMappingRegistry of the service, which
            // is the reason why registration is only needed
for the first call.
            if (firstCall()) {
                // must set encoding style before
registering serializers

call.setEncodingStyle(org.apache.axis.Constants.URI_SOAP_ENC
);
                for (int i = 0; i <
cachedSerFactories.size(); ++i) {
                    Class cls = (Class)
cachedSerClasses.get(i);
                    javax.xml.rpc.namespace.QName qName =
                            (javax.xml.rpc.namespace.QName)
cachedSerQNames.get(i);
                    Class sf = (Class)
                             cachedSerFactories.get(i);
                    Class df = (Class)
                             cachedDeserFactories.get(i);
                    call.registerTypeMapping(cls, qName, sf,
df, false);
                }
            }
            return call;
        }
        catch (Throwable t) {
```

```
            throw new org.apache.axis.AxisFault("Failure
trying to get the Call object", t);
        }
    }

    public samples.my_axis.Indirizzo.Indirizzo
getAddressfromName(java.lang.String in0) throws
java.rmi.RemoteException{
        if (super.cachedEndpoint == null) {
            throw new org.apache.axis.NoEndPointException();
        }
        org.apache.axis.client.Call call = getCall();
        javax.xml.rpc.namespace.QName p0QName = new
javax.xml.rpc.namespace.QName("", "in0");
        call.addParameter(p0QName, new
javax.xml.rpc.namespace.QName("http://schemas.xmlsoap.org/so
ap/encoding/", "string"),
javax.xml.rpc.ParameterMode.PARAM_MODE_IN);
        call.setReturnType(new
javax.xml.rpc.namespace.QName("http://Indirizzo.my_axis.samp
les", "Indirizzo"));
        call.setUseSOAPAction(true);
        call.setSOAPActionURI("");
        call.setOperationStyle("rpc");
        call.setOperationName(new
javax.xml.rpc.namespace.QName("urn:Indirizzo_servizio",
"getAddressfromName"));

        Object resp = call.invoke(new Object[] {in0});

        if (resp instanceof java.rmi.RemoteException) {
            throw (java.rmi.RemoteException)resp;
        }
        else {
             return (samples.my_axis.Indirizzo.Indirizzo)
resp;
        }
    }

}
```

Senza entrare nel dettaglio del codice stub , nelle righe finali (metodo `getAddressfromName`), si può notare come per l'accesso al servizio remoto utilizzi gli oggetti (e i relativi metodi) di tipo `Service` e `Call` esattamente come nel client dell'esempio `Add`.
Quindi, a parte i legami opportuni (con i package, le interfacce SDI, ecc.), è del tutto simile ad un client creato manualmente dallo sviluppatore.
Concludendo, lo stub, può essere considerato il cuore nell'implementare automaticamente un client a partire da un documento WSDL.

È bene analizzare meglio il funzionamento del metodo `IndirizzoServizioSoapBindingStub`:

- `public IndirizzoServizioSoapBindingStub(javax.xml.rpc.Service service) throws org.apache.axis.AxisFault`

Questo il metodo riceve in ingresso un oggetto dello stesso tipo del corrispondente service locator, infatti sarà il service locator a chiamare direttamente o indirettamente (a seconda del numero di parametri inviati) questo metodo.
Il compito del metodo è quello di creare un serializzatore ed un deserializzatore per il tipo di dato complesso scambiato nei messaggi SOAP (nel nostro caso `Indirizzo`).

- `Class cls;`
  Definizione di un oggetto di tipo `Class`.

- `javax.xml.rpc.namespace.QName qName;`
  Definizione di un oggetto di tipo `QName`.

- `Class beansf = org.apache.axis.encoding.ser.BeanSerializerFactory.class;`
  La classe `org.apache.axis.encoding.ser.BeanSerializerFactory` rappresenta un generico serializzatore per un arbitrario Java Bean.

- `Class beandf = org.apache.axis.encoding.ser.BeanDeserializerFactory.class;`
  La classe `org.apache.axis.encoding.ser.BeanDeserializerFactory` rappresenta un generico deserializzatore per un arbitrario Java Bean.

- `Class enumsf = org.apache.axis.encoding.ser.EnumSerializerFactory.class;`
  La classe `org.apache.axis.encoding.ser.EnumSerializerFactory` rappresenta un generico serializzatore per dati di tipo Enum (enumerativi).

- `Class enumdf = org.apache.axis.encoding.ser.EnumDeserializerFactory.class;`
  La classe `org.apache.axis.encoding.ser.EnumDeserializerFactory` rappresenta un generico deserializzatore per dati di tipo Enum (enumerativi).

- `Class arraysf = org.apache.axis.encoding.ser.ArraySerializerFactory.class;`
  La classe `org.apache.axis.encoding.ser.ArraySerializerFactory` rappresenta un generico serializzatore per dati di tipo Array.

- `Class arraydf = org.apache.axis.encoding.ser.ArrayDeserializerFactory.class;`

La classe
`org.apache.axis.encoding.ser.ArrayDeserializerFactory`
rappresenta un generico deserializzatore per dati di tipo Array.

- `Class simplesf = org.apache.axis.encoding.ser.SimpleNonPrimitiveSerializerFactory.class;`
  La classe
  `org.apache.axis.encoding.ser.SimpleNonPrimitiveSerializerFactory` rappresenta un generico serializzatore per attributi semplici che potrebbero essere multi-refed (ad esempio java.lang.Integer).

- `Class simpledf = org.apache.axis.encoding.ser.SimpleDeserializerFactory.class;`
  La classe
  `org.apache.axis.encoding.ser.SimpleNonPrimitiveDeserializerFactory` rappresenta un generico deserializzatore per attributi semplici che potrebbero essere multi-refed (ad esempio java.lang.Integer).

- `qName = new javax.xml.rpc.namespace.QName("http://Indirizzo.my_axis.samples", "Indirizzo");`
  Definisce un nome qualificato.

- `cachedSerQNames.add(qName);`
  Definisce un nuovo serializzatore/deserializzatore con il nome qualificato definito in precedenza.

- `cls = samples.my_axis.Indirizzo.Indirizzo.class;`
  Istanzia un oggetto di tipo `Class`, si tratta della classe che specifica il dato di tipo complesso `Indirizzo`.

- `cachedSerClasses.add(cls);`
  Definisce la classe oggetto della serializzazione/deserializzazione.

- `cachedSerFactories.add(beansf);`
  Definisce la classe che si occuperà della serializzazione.

- `cachedDeserFactories.add(beandf);`
  Definisce la classe che si occuperà della deserializzazione.

Invece, il metodo `Call` è chiamato dal metodo `getAddressfromName`, ed ha il compito di verificare il tipo di comunicazione (mantenimento sessione, utilizzo di password, timeout impostati, ecc.) e di recuperare le informazioni sulla serializzazione/deserializzazione (che sono state memorizzate dal metodo `IndirizzoServizioSoapBinding`) necessarie per accedere al servizio.
Infatti, si deve notare, che il metodo `IndirizzoServizioSoapBinding` è invocato dal corrispondente service locator, mentre il metodo `getAddressfromName` è invocato dal codice dell'utente finale; quindi ci troviamo in due istanti temporali diversi.

Services

WSDL2Java per ogni elemento `service` del documento WSDL genera due oggetti:

- un'interfaccia service;

- un corrispondente service locator che implementa quest'interfaccia.

Ad esempio, dato il seguente spezzone WSDL:

```
   <wsdl:service name="Indirizzo_serverService">

      <wsdl:port
binding="intf:Indirizzo_servizioSoapBinding"
name="Indirizzo_servizio">

         <wsdlsoap:address
location="http://localhost:8080/axis/services/Indirizzo_serv
izio"/>

      </wsdl:port>

   </wsdl:service>
```

WSDL2Java genererà l'interfaccia:

```
/**
 * IndirizzoServerService.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis Wsdl2java emitter.
 */

package Indirizzo_servizio;

public interface IndirizzoServerService extends
javax.xml.rpc.Service {
    public String getIndirizzoServizioAddress();

    public Indirizzo_servizio.IndirizzoServer
getIndirizzoServizio() throws
javax.xml.rpc.ServiceException;

    public Indirizzo_servizio.IndirizzoServer
getIndirizzoServizio(java.net.URL portAddress) throws
javax.xml.rpc.ServiceException;
}
```

Ed il relativo service locator che implementa l'interfaccia service precedente:

```
/**
 * IndirizzoServerServiceLocator.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis Wsdl2java emitter.
 */

package Indirizzo_servizio;

public class IndirizzoServerServiceLocator extends
org.apache.axis.client.Service implements
Indirizzo_servizio.IndirizzoServerService {
```

```
    // Use to get a proxy class for IndirizzoServizio
    private final java.lang.String IndirizzoServizio_address
= "http://localhost:8080/axis/services/Indirizzo_servizio";

    public String getIndirizzoServizioAddress() {
        return IndirizzoServizio_address;
    }

    public Indirizzo_servizio.IndirizzoServer
getIndirizzoServizio() throws javax.xml.rpc.ServiceException
{
       java.net.URL endpoint;
        try {
            endpoint = new
java.net.URL(IndirizzoServizio_address);
        }
        catch (java.net.MalformedURLException e) {
            return null; // unlikely as URL was validated in
WSDL2Java
        }
        return getIndirizzoServizio(endpoint);
    }

    public Indirizzo_servizio.IndirizzoServer
getIndirizzoServizio(java.net.URL portAddress) throws
javax.xml.rpc.ServiceException {
        try {
            return new
Indirizzo_servizio.IndirizzoServizioSoapBindingStub(portAddr
ess, this);
        }
        catch (org.apache.axis.AxisFault e) {
            return null; // ???
        }
    }
}
```

L'interfaccia service definisce una coppia di metodi `get` per ogni sottoelemento `port` di `service` presente nel documento WSDL. Uno per accedere al sevizio con l'URL indicato nel documento WSDL originale, l'altro per accedere al servizio tramite un URL specificato al momento della chiamata.
Il service locator implementa questi metodi `get`, che a sua volta fanno riferimento ad un metodo definito nello stub.

Un tipico utilizzo delle classi generate automaticamente potrebbe essere il seguente:

```
public class Indirizzo_client_wsdl
{
   public static void main(String [] args) throws Exception
   {//costruiamo un servizio
    Indirizzo_servizio.IndirizzoServerService service = new
Indirizzo_servizio.IndirizzoServerServiceLocator();

    //Adesso utilizzo il servizio per ottenere uno stub che
implementi l'interfaccia SDI.
```

```
            Indirizzo_servizio.IndirizzoServer port =
        service.getIndirizzoServizio();
           //dichiaro un oggetto di tipo Indirizzo
           samples.my_axis.Indirizzo.Indirizzo indirizzo = new
samples.my_axis.Indirizzo.Indirizzo();

            //Faccio una chiamata
            indirizzo= (samples.my_axis.Indirizzo.Indirizzo)
        port.getAddressfromName("pippo");

            //visualizzo il risultato della chiamata
            System.out.println("OK");
            System.out.println("Nome via:"+indirizzo.nomeVia);
            System.out.println("num civico:"+indirizzo.numCivico);
            System.out.println("CAP:"+indirizzo.CAP);
            System.out.println("Città:"+indirizzo.citta);
            System.out.println("Provincia:"+indirizzo.provincia);
           }
        }
```

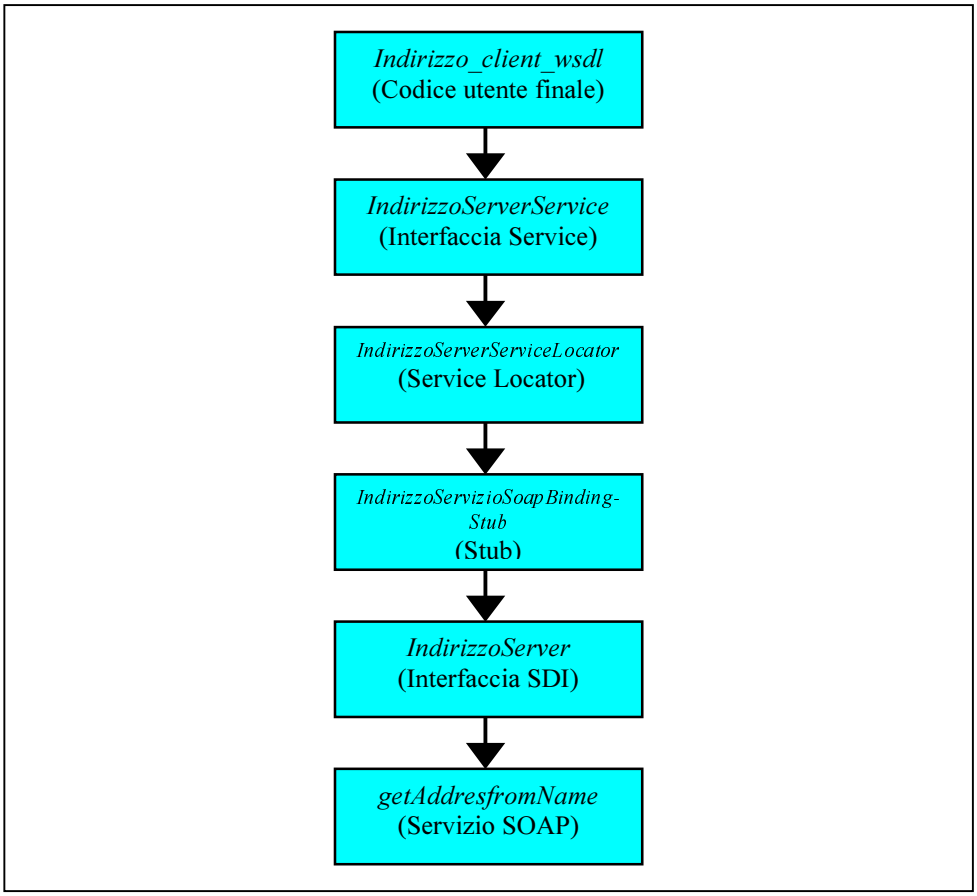
Dove:

- `Indirizzo_servizio.IndirizzoServerService service = new Indirizzo_servizio.IndirizzoServerServiceLocator();`
  Inizializziamo l'oggetto `service` di tipo `Indirizzo_servizio.IndirizzoServerService` (descritto nell'interfaccia service) con la corrispondente implementazione descritta nel service locator `Indirizzo_servizio.IndirizzoServerServiceLocator`.

- `Indirizzo_servizio.IndirizzoServer port = service.getIndirizzoServizio();`
  All'oggetto `service` applico il metodo `getIndirizzoServizio()`, il quale invoca (con l'URL predefinito del servizio) lo stub `Indirizzo_servizio.IndirizzoServizioSoapBindingStub`.
  All'interno di questo stub si accede al servizio utilizzando gli oggetti di tipo `Service` e `Call` che già abbiamo visto in precedenza. Il valore restituito dal metodo è di tipo `Indirizzo_servizio.IndirizzoServer` che è descritto nell'interfaccia SDI.

- `indirizzo= (samples.my_axis.Indirizzo.Indirizzo) port.getAddressfromName("pippo");`
  Chiamo il metodo `getAddressfromName` del servizio pubblicato, che dato il nome `pippo` mi restituisce il suo indirizzo nella variabile `indirizzo` (che ovviamente è di tipo `samples.my_axis.Indirizzo.Indirizzo`).

Per capire meglio il legame (ed in parte l'ordine di chiamata) degli oggetti creati per l'esempio visto fin ora, possiamo far riferimento al seguente diagramma:



Lato server

Così come lo stub rappresenta il lato client di un Web Service, lo skeleton è l'equivalente sul lato server. Per costruire la classe skeleton, occorre specificare le opzioni `--server-side` e `--skeletonDeploy true` nella chiamata al tool WSDL2Java. Per esempio, continuando ad utilizzare il file `Indirizzo.wsdl`, dal prompt comandi occorre digitare la seguente istruzione:

```
java org.apache.axis.wsdl.WSDL2Java --server-side --
skeletonDeploy true Indirizzo.wsdl
```

A questo punto saranno generate tutte le classi che sono state generate in precedenza per il lato client, ed i seguenti documenti:

Sezione WSDL	**Classe/i Java generate**
Per ogni entry nella sezione binding	Una classe skeleton Un'implementazione template class
Per ogni entry nella sezione service	Un file deploy.wsdd (con eventuali operazioni metadata(
	Un file undeploy.wsdd

Se viene specificata l'opzione `--skeletonDeploy false`, la classe skeleton non viene generata; siamo nella seguente situazione:

Sezione WSDL	Classe/i Java generate
Per ogni entry nella sezione binding	Un'implementazione template class
Per ogni entry nella sezione service	Un file deploy.wsdd (con eventuali operazioni metadata)
	Un file undeploy.wsdd

In questo caso il file `deploy.wsdd` fa riferimento alla classe template, mentre con l'opzione `--skeletonDeploy true` fa riferimento alla classe skeleton.

Binding

Descrizione Skeleton

La classe skeleton è una classe intermedia fra il "motore" Axis e l'implementazione effettiva del servizio. Può essere utilizzata per preparare il servizio settando tutti gli opportuni parametri della comunicazione, mentre l'implementazione del servizio viene lasciata alla classe template.
Il suo nome è dato dal nome della entry `binding` con il suffisso `Skeleton`.
Ad esempio, per il `binding Indirizzo_servizioSoapBinding`, verrà generato:

```
/**
 * IndirizzoServizioSoapBindingSkeleton.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis Wsdl2java emitter.
 */

package Indirizzo_servizio;

public class IndirizzoServizioSoapBindingSkeleton implements
Indirizzo_servizio.IndirizzoServer,
    org.apache.axis.wsdl.Skeleton {
    private Indirizzo_servizio.IndirizzoServer impl;
    private static org.apache.axis.wsdl.SkeletonImpl skel =
null;

    public IndirizzoServizioSoapBindingSkeleton() {
        this.impl = new
Indirizzo_servizio.IndirizzoServizioSoapBindingImpl();
        init();
    }

    public
IndirizzoServizioSoapBindingSkeleton(Indirizzo_servizio.Indi
rizzoServer impl) {
        this.impl = impl;
        init();
    }
```

```
    public javax.xml.rpc.namespace.QName
getParameterName(String opName, int i) {
        return skel.getParameterName(opName, i);
    }

    public static javax.xml.rpc.namespace.QName
getParameterNameStatic(String opName, int i) {
        init();
        return skel.getParameterName(opName, i);
    }

    public javax.xml.rpc.ParameterMode
getParameterMode(String opName, int i) {
        return skel.getParameterMode(opName, i);
    }

    public static javax.xml.rpc.ParameterMode
getParameterModeStatic(String opName, int i) {
        init();
        return skel.getParameterMode(opName, i);
    }

    public static String getInputNamespaceStatic(String
opName) {
        init();
        return skel.getInputNamespace(opName);
    }

    public static String getOutputNamespaceStatic(String
opName) {
        init();
        return skel.getOutputNamespace(opName);
    }

    public static String getSOAPAction(String opName) {
        init();
        return skel.getSOAPAction(opName);
    }

    protected static void init() {
        if (skel != null)
            return;
        skel = new org.apache.axis.wsdl.SkeletonImpl();
        skel.add("getAddressfromName",
                 new javax.xml.rpc.namespace.QName[] {
                   new javax.xml.rpc.namespace.QName("",
"return"),
                   new javax.xml.rpc.namespace.QName("",
"in0"),
                 },
                 new javax.xml.rpc.ParameterMode[] {

javax.xml.rpc.ParameterMode.PARAM_MODE_OUT,

javax.xml.rpc.ParameterMode.PARAM_MODE_IN,
                 },
```

```
                "urn:Indirizzo_servizio",
                "urn:Indirizzo_servizio",
                "");
    }

    public samples.my_axis.Indirizzo.Indirizzo
getAddressfromName(java.lang.String in0) throws
java.rmi.RemoteException
    {
        samples.my_axis.Indirizzo.Indirizzo ret =
impl.getAddressfromName(in0);
        return ret;
    }

}
```

Lo skeleton contiene un implementazione dell'interfaccia SDI `IndirizzoServer`, già descritta per il lato client.
Quando il "motore" Axis chiama il metodo `getAddressfromName` della skeleton, questo semplicemente delega l'invocazione alla reale implementazione del metodo (la classe template).
All'interno del codice skeleton troviamo diverse funzioni `get?`, queste possono essere utilizzate per ottenere informazioni sul nome dei parametri, tipo (ingresso, uscita, ingresso/uscita) dei parametri, il namespace del servizio e l'header SOAPAction del servizio.
Sempre nel codice skeleton troviamo il metodo `init`, il quale ha il compito di settare il nome dell'operazione fornita dal servizio, i relativi parametri con la relativa tipologia (ingresso, uscita, ingresso/uscita) ed i namespace associati.

Descrizione implementazione Template

WSDL2Java genera un'implementazione template dal `binding` (nome del `binding` con il suffisso `Impl`):

```
/**
 * IndirizzoServizioSoapBindingImpl.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis Wsdl2java emitter.
 */

package Indirizzo_servizio;

public class IndirizzoServizioSoapBindingImpl implements
Indirizzo_servizio.IndirizzoServer {
    public samples.my_axis.Indirizzo.Indirizzo
getAddressfromName(java.lang.String in0) throws
java.rmi.RemoteException {
        return null;
    }

}
```

Questa implementazione, attualmente, potrebbe essere utilizzata come test. Infatti, da quanto si può vedere, non fa nulla (ritorna un valore nullo per l'oggetto di tipo `samples.my_axis.Indirizzo.Indirizzo`).
Resta inteso che il servizio dovrà essere esteso manualmente a partire da questa template.
Quando WSDL2Java è chiamato a generare un'implementazione template (opzione `--server-side`), questa verrà generata solamente se non è già presente. Se l'implementazione template esiste già, non sarà soprascritta.

Services

Il tool WSDL2Java costruisce anche un file `deploy.wsdl` e un file `undeploy.wsdl` per ogni entry di tipo `service`.
Entrambi i file sono utilizzati nel comando, già spiegato in precedenza, `java org.apache.axis.client.AdminClient` per eseguire fisicamente il deploy/undeploy di un servizio.
Il primo file, una volta completata l'implementazione template e compilate tutte le classi generate, contiene tutte le informazioni per pubblicare il servizio.
Il secondo file, invece, rimuove il servizio pubblicato.

Contenuto del file `deploy.wsdd` (opzione `--SkeletonDeploy true`) :

```
<!-- Use this file to deploy some handlers/chains and
services   -->
<!-- Two ways to do this:                                -->
<!--   java org.apache.axis.utils.Admin deploy.wsdd      -->
<!-- from the same directory that the Axis engine runs   -->
<!-- or                                                  -->
<!--java org.apache.axis.client.AdminClient deploy.wsdd  -->
<!--      after the axis server is running               -->

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

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

 <!-- Services from Indirizzo_serverService WSDL service -->

  <service name="Indirizzo_servizio" provider="java:RPC">
      <parameter name="className"
value="Indirizzo_servizio.IndirizzoServizioSoapBindingSkelet
on"/>
      <parameter name="allowedMethods"
value="getAddressfromName"/>

      <typeMapping
        xmlns:ns="http://Indirizzo.my_axis.samples"
        qname="ns:Indirizzo"
        type="java:samples.my_axis.Indirizzo.Indirizzo"

serializer="org.apache.axis.encoding.ser.BeanSerializerFacto
ry"

deserializer="org.apache.axis.encoding.ser.BeanDeserializerF
actory"

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
```

```
      />
  </service>
</deployment>
```

Con l'opzione `--SkeletonDeploy false` cambia solo la seguente parte:

```
  <service name="Indirizzo_servizio" provider="java:RPC">
      <parameter name="className"
value="Indirizzo_servizio.IndirizzoServizioSoapBindingImpl"/
>
      <parameter name="allowedMethods"
value="getAddressfromName"/>
```

Quindi si fa riferimento alla classe template invece che alla classe skeleton.

Guardando il file `deploy.wsdd` è possibile notare alcune novità che riguardano la serializzazione/deserializzazione del dato di tipo complesso `Indirizzo`:

- `typeMapping`
  Specifica il mapping fra un nome XML qualificato e una classe Java (e viceversa).

- `xmlns:ns=http://Indirizzo.my_axis.samples`
  Specifica un namespace.

- `qname="ns:Indirizzo"`
  Specifica il nome XML qualificato.

- `type="java:samples.my_axis.Indirizzo.Indirizzo"`
  Specifica la classe Java.

- `serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"`
  Specifica il serializzatore utilizzato.
  Si tratta del nome della classe Java che ha il compito di serializzare i dati della classe Java (specificata in `type`) in XML.
  La classe `org.apache.axis.encoding.ser.BeanSerializerFactory` rappresenta un generico serializzatore per un arbitrario Java Bean (cioè per una classe Java che sia dotata di tutti i metodi `get/set` necessari per accedere/modificare ogni elemento della classe stessa).

- `deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"`
  Specifica il deserializzatore utilizzato.
  Si tratta del nome della classe Java che ha il compito di deserializzare i dati da XML alla classe Java (specificata in `type`). La classe

`org.apache.axis.encoding.ser.BeanDeserializerFactory` rappresenta un generico deserializzatore per un arbitrario Java Bean(cioè per una classe Java che sia dotata di tutti i metodi `get/set` necessari per accedere/modificare ogni elemento della classe stessa).

- `encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"`
  Specifica l'attributo encodingStyle di SOAP.

Si noti che nello stub (`IndirizzoServizioSoapBinding`) basandosi sulle classi specificate in `serializer` e `deserializer`, è stato specificato il serializzatore/deserializzatore opportuno, per la classe `samples.my_axis.Indirizzo.Indirizzo`, con il nome `Indirizzo` e namespace `http://Indirizzo.my_axis.samples`.
In altre parole, questo frammento del documento WSDD, dice al "motore" Axis dove trovare il serializzatore/deserializzatore per il dato di tipo complesso specificato.

Contenuto del file `undeploy.wsdd`:

```
<!-- Use this file to undeploy some handlers/chains and
services -->
<!-- Two ways to do this:                                  -->
<!--   java org.apache.axis.utils.Admin undeploy.wsdd    -->
<!-- from the same directory that the Axis engine runs   -->
<!-- or                                                   -->
<!--java org.apache.axis.client.AdminClient undeploy.wsdd-->
<!--      after the axis server is running               -->

<undeployment
    xmlns="http://xml.apache.org/axis/wsdd/">

  <!--Services from Indirizzo_serverService WSDL service -->

  <service name="Indirizzo_servizio"/>
</undeployment>
```

Esempio Indirizzo (web server Tomcat e client Apache Axis)

In realtà l'esempio Indirizzo è stato già stato sviluppato nell'analizzare il tool WSDL2Java. Come visto dai documenti generati il client è sviluppato in Apache Axis, così come il server che sarà pubblicato attraverso Tomcat.
A questo punto, occorre completare la classe template perché si realizzi effettivamente il servizio desiderato.
Quindi, il file `IndirizzoServizioSoapBindingImpl` , avrà il seguente contenuto:

```
/**
 * IndirizzoServizioSoapBindingImpl.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis Wsdl2java emitter.
 */
```

```
package Indirizzo_servizio;

public class IndirizzoServizioSoapBindingImpl implements
Indirizzo_servizio.IndirizzoServer {
    public samples.my_axis.Indirizzo.Indirizzo
getAddressfromName(java.lang.String in0) throws
java.rmi.RemoteException {
   samples.my_axis.Indirizzo.Indirizzo ind=new
samples.my_axis.Indirizzo.Indirizzo("giardini",15,41028,"Ser
ramazzoni","Mo");
        System.out.println("Servizio Indirizzo_servizio
OK");
        return (samples.my_axis.Indirizzo.Indirizzo)ind;
    }

}
```

Come si può notare, il servizio restituisce un indirizzo costante qualunque sia il nome della persona passato come parametro dal client. Ovviamente, questa è una semplificazione che non ha nulla a che vedere con le potenzialità di Axis e SOAP.

Una volta compilato il file modificato, possiamo pubblicare fisicamente il servizio con il solito comando:
`java org.apache.axis.client.AdminClient deploy.wsdd`.

Per effettuare una richiesta SOAP al servizio, facciamo eseguire il client con il comando:
`java Indirizzo_client_wsdl`.

A video comparirà, a conferma della corretta esecuzione del servizio, il seguente risultato:
```
OK
Nome via:giardini
num civico:15
CAP:41028
Città:Serramazzoni
Provincia:Mo
```

Messaggi SOAP di Indirizzo

Il messaggio SOAP di richiesta da parte del client è il seguente:

```
POST /axis/services/Indirizzo_servizio HTTP/1.0
Content-Length: 530
Host: localhost
Content-Type: text/xml; charset=utf-8
SOAPAction: ""

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

```
  <ns1:getAddressfromName
xmlns:ns1="urn:Indirizzo_servizio">
   <in0 xsi:type="xsd:string">pippo</in0>
  </ns1:getAddressfromName>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Il messaggio SOAP di risposta da parte del server è il seguente:

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 935
Servlet-Engine: Tomcat Web Server/3.2.3 (JSP 1.1; Servlet
2.2; Java 1.4.0; Windows XP 5.1 x86; java.vendor=Sun
Microsystems Inc.)

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <SOAP-ENV:Body>
  <ns1:getAddressfromNameResponse SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
" xmlns:ns1="urn:Indirizzo_servizio">
   <getAddressfromNameResult href="#id0"/>
  </ns1:getAddressfromNameResponse>
  <multiRef id="id0" SOAP-ENC:root="0"
xsi:type="ns2:Indirizzo" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns2="http://Indirizzo.my_axis.samples">
   <ns2:nome_via
xsi:type="xsd:string">giardini</ns2:nome_via>
   <ns2:num_civico xsi:type="xsd:int">15</ns2:num_civico>
   <CAP xsi:type="xsd:int">41028</CAP>
   <citta xsi:type="xsd:string">Serramazzoni</citta>
   <provincia xsi:type="xsd:string">Mo</provincia>
  </multiRef>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Esempio Indirizzo (web server Tomcat e client Visual Basic)

Riprendiamo ancora una volta l'analisi del tipo di dato complesso `Indirizzo`. Nel precedente paragrafo abbiamo studiato un implementazione con web server Tomcat (codice Java/Apache Axis per il server) e client Apache Axis. Mentre, nel precedente capitolo, abbiamo studiato un implementazione con web server ISS (codice Visual Basic per il server) e client Visual Basic.
A questo punto, per mettere in atto il concetto d'interoperabilità sui tipi di dato complessi, studiamo un'implementazione con web server Tomcat (codice Java/Apache Axis per il server) e client Visual Basic.

Server Indirizzo

Per quanto riguarda il server, si tratta esattamente dello stesso codice e della stessa pubblicazione vista per il servizio del precedente paragrafo. Quindi il servizio riceve il nome di una persona dal client e gli restituisce l'indirizzo della persona.

Client Indirizzo

Anche il codice Visual Basic del client resta pressoché invariato:

```
Option Explicit
Private soapclient As soapclient
Private Sub Form_Load()
On Error GoTo fail
Set soapclient = New soapclient

soapclient.mssoapinit "D:\Indirizzo_axis\Indirizzo_ms.wsdl",
"", "", "D:\Indirizzo_axis\Indirizzo_ms.wsml"

Dim inputName As String
inputName = "pippo"
MsgBox "Richiesta al server dell'indirizzo di: " & inputName

Dim IndirizzoObjReturned As Indirizzo
Set IndirizzoObjReturned =
soapclient.getAddressfromName(inputName)
Dim strIndirizzo As String

strIndirizzo = "nomeVia: " &
IndirizzoObjReturned.get_nomeVia & vbCrLf & _
            "numCivico: " &
IndirizzoObjReturned.get_numCivico & vbCrLf & _
            "CAP: " & IndirizzoObjReturned.get_CAP & vbCrLf
& _
            "città: " & IndirizzoObjReturned.get_citta &
vbCrLf & _
            "provincia: " &
IndirizzoObjReturned.get_provincia & vbCrLf
MsgBox strIndirizzo
Exit Sub
fail:
MsgBox soapclient.detail

End Sub
```

Ovviamente cambiano i documenti WSDL e WSML di riferimento (si veda il metodo `mssoapinit` d'inizializzazione per l'oggetto `soapclient`).

Per quanto riguarda il documento WSDL si riprende, come punto di partenza, il file `Indirizzo.wsdl` visto nel precedente paragrafo `Tool WSDL2Java`. In altre parole il documento WSDL, di riferimento per il client, è quello messo a disposizione sul lato server del servizio; infatti una volta definito il servizio sul lato server, attraverso il tool WSDL2Java di Apache Axis, è possibile ottenere il corrispondente documento WSDL.
Il documento `Indirizzo_ms.wsdl` si differenzia dal documento `Indirizzo.wsdl` per il namespace **`xsd`** anzichè `SOAP-ENC` associato al parametro di richiesta `in0` del `message getAddressfromNameRequest`. Questo cambiamento è stato necessario

perché il client Visual Basic potesse trovare lo schema di definizione per il tipo `string` del parametro.

Il contenuto del documento `Indirizzo_ms.wsdl` è il seguente:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="urn:Indirizzo_servizio"
xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:impl="urn:Indirizzo_servizio-impl"
xmlns:intf="urn:Indirizzo_servizio"
xmlns:tns1="http://Indirizzo.my_axis.samples"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <types>
  <schema targetNamespace="http://Indirizzo.my_axis.samples"
xmlns="http://www.w3.org/2001/XMLSchema">
   <complexType name="Indirizzo">
    <sequence>
     <element name="nome_via" nillable="true"
type="xsd:string"/>
     <element name="num_civico" type="xsd:int"/>
     <element name="CAP" type="xsd:int"/>
     <element name="citta" nillable="true"
type="xsd:string"/>
     <element name="provincia" nillable="true"
type="xsd:string"/>
    </sequence>
   </complexType>
   <element name="Indirizzo" nillable="true"
type="tns1:Indirizzo"/>
  </schema>
 </types>

   <wsdl:message name="getAddressfromNameResponse">

      <wsdl:part name="return" type="tns1:Indirizzo"/>

   </wsdl:message>

   <wsdl:message name="getAddressfromNameRequest">

      <wsdl:part name="in0" type="xsd:string"/>

   </wsdl:message>

   <wsdl:portType name="Indirizzo_server">

      <wsdl:operation name="getAddressfromName"
parameterOrder="in0">

         <wsdl:input
message="intf:getAddressfromNameRequest"/>
```

```
         <wsdl:output
message="intf:getAddressfromNameResponse"/>

      </wsdl:operation>

   </wsdl:portType>

   <wsdl:binding name="Indirizzo_servizioSoapBinding"
type="intf:Indirizzo_server">

      <wsdlsoap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>

      <wsdl:operation name="getAddressfromName">

         <wsdlsoap:operation soapAction=""/>

         <wsdl:input>

            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:Indirizzo_servizio" use="encoded"/>

         </wsdl:input>

         <wsdl:output>

            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:Indirizzo_servizio" use="encoded"/>

         </wsdl:output>

      </wsdl:operation>

   </wsdl:binding>

   <wsdl:service name="Indirizzo_serverService">

      <wsdl:port
binding="intf:Indirizzo_servizioSoapBinding"
name="Indirizzo_servizio">

         <wsdlsoap:address
location="http://localhost:8081/axis/services/Indirizzo_serv
izio"/>

      </wsdl:port>

   </wsdl:service>

</wsdl:definitions>
```

Il documento `Indirizzo_ms.wsml` ha il seguente contenuto:

```
<?xml version='1.0' encoding='UTF-8' ?>
<servicemapping name='Indirizzo_axis'>
  <service name='Indirizzo_serverService'>
    <using PROGID='Indirizzo_Mapper_axis.IndirizzoMapper'
cachable='0' ID='IndirizzoMap' />
    <types>
     <type name='Indirizzo'

targetNamespace='http://Indirizzo.my_axis.samples'
           uses='IndirizzoMap' />
    </types>
  </service>
</servicemapping>
```

Come si può notare, il documento WSML, non contiene più nessun riferimento a classi COM che implementano il servizio. Questo perché il server non è più scritto in Visual Basic ma in Java, ed è pubblicato su Tomcat e non più su ISS.
Al contrario, la parte riguardante la classe mapper ed il legame con il tipo di dato complesso, è rimasta. In neretto sono riportate le modifiche necessarie per avere coerenza con il documento WSDL fornito al client.

Perché il lato client possa gestire correttamente la risposta (la deserializzazione) contenente il tipo di dato complesso `Indirizzo`, è necessario che definisca la classe `Indirizzo` ed un opportuno mapper (deserializzatore).
Per fare ciò, si utilizzano rispettivamente le classi `Indirizzo` ed `Indirizzo_mapper_axis` (che sono molto simili alle classi `Indirizzo` ed `Indirizzo_mapper` viste nel capitolo precedente).

Il contenuto della classe `Indirizzo` è il seguente:

```
Option Explicit

Private nomeVia As String
Private numCivico As Integer
Private CAP As Integer
Private citta As String
Private provincia As String

Public Property Get get_nomeVia() As String
    get_nomeVia = nomeVia
End Property

Public Property Let let_nomeVia(ByVal Value As String)
    nomeVia = Value
End Property

Public Property Get get_numCivico() As Integer
    get_numCivico = numCivico
End Property

Public Property Let let_numCivico(ByVal Value As Integer)
    numCivico = Value
End Property
```

```
Public Property Get get_CAP() As Integer
    get_CAP = CAP
End Property

Public Property Let let_CAP(ByVal Value As Integer)
    CAP = Value
End Property

Public Property Get get_citta() As String
    get_citta = citta
End Property

Public Property Let let_citta(ByVal Value As String)
    citta = Value
End Property
Public Property Get get_provincia() As String
    get_provincia = provincia
End Property

Public Property Let let_provincia(ByVal Value As String)
    provincia = Value
End Property
```

Come si può notare, ovviamente, la classe `Indirizzo` non ha subito cambiamenti.

Il contenuto della classe `Indirizzo_mapper` è il seguente:

```
Option Explicit

Implements ISoapTypeMapper

Private myMapper As ISoapTypeMapper

'inizializzazione
Private Sub ISoapTypeMapper_Init( _
    ByVal pFactory As MSSOAPLib.ISoapTypeMapperFactory, _
    ByVal pSchema As MSXML2.IXMLDOMNode, _
    ByVal xsdType As MSSOAPLib.enXSDType)

    Set myMapper = pFactory.getMapper(enXSDanyType, Nothing)
End Sub

'ricerca, a partire dal nodo radice, un nodo dato il suo
base_name ed il suo namespace
Public Function FindElement(ByVal objRootNode As
MSXML2.IXMLDOMNode, ByVal strName As String, ByVal
strNamespaceURI As String) As MSXML2.IXMLDOMNode

  Dim objChildNode As MSXML2.IXMLDOMNode

  For Each objChildNode In objRootNode.childNodes
    If objChildNode.nodeType = NODE_ELEMENT Then
      If objChildNode.baseName = strName And
objChildNode.namespaceURI = strNamespaceURI Then
        Set FindElement = objChildNode
        Exit Function
```

```
      End If
    End If
  Next

  Set FindElement = Nothing

End Function

'deserializzazione
Private Function ISoapTypeMapper_read(ByVal pNode As
MSXML2.IXMLDOMNode, _
ByVal bstrEncoding As String, ByVal encodingMode As _
MSSOAPLib.enEncodingStyle, ByVal lFlags As Long) As Variant
  Dim Indirizzo As New Indirizzo
    Dim node As IXMLDOMNode

    MsgBox "alcune informazioni sui nodi"
    Dim objChildNode As MSXML2.IXMLDOMNode
    For Each objChildNode In pNode.childNodes
         MsgBox "BASE NAME: " + objChildNode.baseName _
         + " NODE NAME: " + objChildNode.nodeName _
         + " NAMESPACE: " + objChildNode.namespaceURI _
         + " TEXT: " + objChildNode.Text _
         + " PREFIX: " + objChildNode.prefix
    Next


    Indirizzo.let_nomeVia = FindElement(pNode, "nome_via",
"http://Indirizzo.my_axis.samples").Text

    Indirizzo.let_numCivico = FindElement(pNode,
"num_civico", "http://Indirizzo.my_axis.samples").Text

    Indirizzo.let_CAP = FindElement(pNode, "CAP", "").Text

    Indirizzo.let_citta = FindElement(pNode, "citta",
"").Text

    Indirizzo.let_provincia = FindElement(pNode,
"provincia", "").Text

    Set ISoapTypeMapper_read = Indirizzo
End Function

Private Function ISoapTypeMapper_varType() As Long
    ISoapTypeMapper_varType = vbObject
End Function

'serializzazione
Private Sub ISoapTypeMapper_write(ByVal pSoapSerializer As _
MSSOAPLib.ISoapSerializer, ByVal bstrEncoding As String,
ByVal _
encodingMode As MSSOAPLib.enEncodingStyle, ByVal lFlags As
Long, _
pvar As Variant)
    Dim Indirizzo As Indirizzo
```

```
    Set Indirizzo = pvar

    pSoapSerializer.startElement "nome_via"
    myMapper.write pSoapSerializer, bstrEncoding,
encodingMode, _
lFlags, Indirizzo.get_nomeVia
    pSoapSerializer.endElement

    pSoapSerializer.startElement "num_civico"
    myMapper.write pSoapSerializer, bstrEncoding,
encodingMode, _
lFlags, Indirizzo.get_numCivico
    pSoapSerializer.endElement

    pSoapSerializer.startElement "CAP"
    myMapper.write pSoapSerializer, bstrEncoding,
encodingMode, _
lFlags, Indirizzo.get_CAP
    pSoapSerializer.endElement

    pSoapSerializer.startElement "citta"
    myMapper.write pSoapSerializer, bstrEncoding,
encodingMode, _
lFlags, Indirizzo.get_citta
    pSoapSerializer.endElement

    pSoapSerializer.startElement "provincia"
    myMapper.write pSoapSerializer, bstrEncoding,
encodingMode, _
lFlags, Indirizzo.get_provincia
    pSoapSerializer.endElement

End Sub
```

Come si può notare la classe `Indirizzo_mapper` non subisce particolari modifiche. L'unico cambiamento riguarda la ricerca dei sottonodi (del parser DOM) contenuti nel messaggio di risposta, che avviene attraverso il metodo `FindElement` (che oltre al nome di base del sottonodo tiene in considerazone anche il suo namespace, così da rendere più generale la ricerca).

Messaggi SOAP di Indirizzo

Il messaggio di richiesta inviato dal client Visual Basic è il seguente:

```
POST /axis/services/Indirizzo_servizio HTTP/1.1
Content-Type: text/xml; charset="UTF-8"
Host: localhost
SOAPAction: ""
Content-Length: 365

<?xml version="1.0" encoding="UTF-8" standalone="no"?><SOAP-
ENV:Envelope SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"><SOAP-
ENV:Body><SOAPSDK1:getAddressfromName
```

```
xmlns:SOAPSDK1="urn:Indirizzo_servizio"><in0>pippo</in0></SO
APSDK1:getAddressfromName></SOAP-ENV:Body></SOAP-
ENV:Envelope>
```

Il messaggio di risposta inviato dal server Apache Axis è il seguente:

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 933
Servlet-Engine: Tomcat Web Server/3.2.3 (JSP 1.1; Servlet
2.2; Java 1.4.0; Windows XP 5.1 x86; java.vendor=Sun
Microsystems Inc.)

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <SOAP-ENV:Body>
  <ns1:getAddressfromNameResponse SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
" xmlns:ns1="urn:Indirizzo_servizio">
   <getAddressfromNameResult href="#id0"/>
  </ns1:getAddressfromNameResponse>
  <multiRef id="id0" SOAP-ENC:root="0"
xsi:type="ns2:Indirizzo" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns2="http://Indirizzo.my_axis.samples">
   <ns2:nome_via
xsi:type="xsd:string">giardini</ns2:nome_via>
   <ns2:num_civico xsi:type="xsd:int">15</ns2:num_civico>
   <CAP xsi:type="xsd:int">426</CAP>
   <citta xsi:type="xsd:string">Serramazzoni</citta>
   <provincia xsi:type="xsd:string">Mo</provincia>
  </multiRef>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Alla richiesta dell'indirizzo di `pippo`, il server risponde con la seguente struttura (di tipo `Indirizzo`):

```
nome_via: giardini
num_civico: 15
CAP: 426
Citta: Serramazzoni
Provincia: Mo.
```

Esempio Rubrica (web server Tomcat e client Apache Axis)

Questo esempio prevede un server scritto in Java e pubblicato, attraverso Apache Axis, su web server Tomcat ed un client scritto in Apache Axis
Nell'esempio si prende in considerazione la seguente classe Java `Rubrica`:

```
package samples.my_axis.Rubrica;

import samples.my_axis.Indirizzo_ser.Indirizzo;

public class Rubrica{

   public String nome;
   public String cognome;
   public String[] num_telefono;
   public Indirizzo[] indirizzi;

   //costruttori
   public Rubrica() { }

   public Rubrica(String nome,String cognome,String[]
num_telefono,Indirizzo[] indirizzi) {
         this.nome=nome;
         this.cognome=cognome;
         this.num_telefono=num_telefono;
         this.indirizzi=indirizzi;
   }

}
```

In pratica la classe `Rubrica` descrive per una certa persona: il suo nome, il suo cognome, una lista di numeri telefonici (ad esempio il numero di casa, del cellulare, dell'ufficio, ecc.) ed una lista di indirizzi (ad esempio l'indirizzo di casa e dell'ufficio).

L'esempio `Rubrica` ci permette di analizzare le seguenti novità:

- La serializzazione/deserializzazione di un array di stringhe, quindi di tipo semplice (`num_telefono`).

- La serializzazione/deserializzazione di un array di `Indirizzo`, quindi di tipo complesso (`indirizzi`).

- Definizione ed utilizzo di un serializzatore e di un deserializzatore, in Apache Axis, per la classe `Indirizzo`. Infatti, finora nell'ambiente Apache Axis, per la classe `Indirizzo` abbiamo utilizzato un serializzatore ed un deserializzatore predefinito (di tipo Java Beans), rispettivamente `org.apache.axis.encoding.ser.BeanSerializerFactory` e `org.apache.axis.encoding.ser.BeanDeserializerFactory`.
  Se immaginiamo che per la classe `Indirizzo` non siano più disponibili i metodi `get/set` per accedere/definire ogni singolo elemento della classe, non sarà più possibile utilizzare questo serializzatore/deserializzatore di tipo Java Beans.

- Definizione ed utilizzo di un serializzatore e di un deserializzatore, in Apache Axis, per la classe `Rubrica`, per gli stessi motivi indicati al punto precedente.

- Indicazione dei serializzatori/deserializzatori, definiti per le classi `Indirizzo` e `Rubrica`, all'interno del documento `deploy.wsdd` (per la pubblicazione del servizio) e del codice client. In questo modo, il server ed il client rispettivamente, sono in grado di recuperare le classi Java che si occupano di serializzare/deserializzare correttamente tutti gli elementi contenuti nella classe `Rubrica`.

Server Rubrica

Compito del server è di restituire un oggetto di tipo `Rubrica` per una certa persona specificata dai parametri (nome e cognome) inviati nella richiesta del client. In realtà, per semplicità, l'oggetto `Rubrica` restituito è costante, cioè indipendente dai parametri inviati dal client.
Il codice del server è il seguente:

```
package samples.my_axis.Rubrica;

import java.io.*;
import java.util.*;
import samples.my_axis.Rubrica.Rubrica;
import samples.my_axis.Indirizzo_ser.Indirizzo;

public class Rubrica_server
{

  public Rubrica getRubrica(String Nome,String Cognome)
   {
      String nome=new String("Paolo");
      String cognome=new String("Rossi");
      String[] num_telefono;
      num_telefono=new String[2];
      num_telefono[0]=new String("casa:02345678");
      num_telefono[1]=new String("ufficio:02123456");
      Indirizzo[] indirizzi;
      indirizzi=new Indirizzo[2];
      indirizzi[0]=new Indirizzo("via Emilia",new
Integer("1"),new Integer("100"),"Modena","Mo");
      indirizzi[1]=new Indirizzo("via Italia",new
Integer("6"),new Integer("101"),"Bologna","Bo");


      Rubrica rubrica= new
Rubrica(nome,cognome,num_telefono,indirizzi);

      System.out.println("Rubrica_server in azione");

      return (Rubrica) rubrica;
   }
}
```

Per il momento non ci sono novità rispetto agli esempi precedenti, e come sempre il file sorgente del server non ha nulla di speciale che lo renda assimilabile ad un servizio web utilizzabile, attraverso SOAP in remoto.

Come già ricordato in precedenza, il contenuto della classe `Indirizzo` è il seguente:

```
package samples.my_axis.Indirizzo_ser;

public class Indirizzo{

   public String nome_via;
   public Integer num_civico;
   public Integer CAP;
   public String citta;
   public String provincia;

   //costruttori
   public Indirizzo() { }

   public Indirizzo(String nome_via,Integer
num_civico,Integer CAP,String citta,String provincia) {
         this.nome_via=nome_via;
         this.num_civico=num_civico;
         this.CAP=CAP;
         this.citta=citta;
         this.provincia=provincia;


   }

}
```

Serializzatore per Indirizzo

La seguente classe, `IndirizzoSerFactory`, ha il compito principale di istanziare il serializzatore per un oggetto di tipo `Indirizzo`:

```
package samples.my_axis.Indirizzo_ser;

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;

import javax.xml.rpc.namespace.QName;
import java.io.IOException;

import org.apache.axis.encoding.Serializer;
import org.apache.axis.encoding.SerializerFactory;
import org.apache.axis.encoding.SerializationContext;
import org.apache.axis.encoding.Deserializer;
import org.apache.axis.encoding.DeserializerFactory;
import org.apache.axis.encoding.DeserializationContext;
import org.apache.axis.encoding.DeserializerImpl;
import org.apache.axis.Constants;

import java.util.Iterator;
import java.util.Vector;
```

```
/**
 * SerializerFactory per Indirizzo *
 */

public class IndirizzoSerFactory implements
SerializerFactory {
    private Vector mechanisms;

    public IndirizzoSerFactory() {   }


    public javax.xml.rpc.encoding.Serializer
getSerializerAs(String mechanismType) {
        return new IndirizzoSer();
    }

    public Iterator getSupportedMechanismTypes() {
        if (mechanisms == null) {
            mechanisms = new Vector();
            mechanisms.add(Constants.AXIS_SAX);
        }
        return mechanisms.iterator();
    }
}
```

I metodi presenti in questa classe sono utilizzati internamente dal "motore" Axis per gestire (registrare/richiamare) il serializzatore per il tipo di dato in esame (`Indirizzo`). Analizziamo il codice:

- `public class IndirizzoSerFactory implements SerializerFactory`
  L'interfaccia `SerializerFactory` (appartenente al package `org.apache.axis.encoding.SerializerFactory`) descrive la `SerializerFactory` (la "fabbrica" dei serializzatori) di Axis. Ogni sua implementazione deve contenere uno o più dei seguenti metodi:
  - `public static create(Class javaType, QName xmlType)`
  - `public (Class javaType, QName xmlType)`
  - `public ()`

  Il meccanismo di deployment (pubblicazione) tenterà di invocare questi metodi nell'ordine visto sopra. Gli argomenti `xmlType` e `javaType` sono riempiti con i valori forniti durante la registrazione del `SerializerFactory` dal codice di deployment. Nell'esempio in esame è presente solo l'ultimo metodo (`public IndirizzoSerFactory()`, che è un semplice costruttore della classe in esame).

- Inoltre l'interfaccia `SerializerFactory` (appartenente al package `org.apache.axis.encoding.SerializerFactory`) estende l'interfaccia `SerializerFactory` (appartenente al package `org.xml.rpc.encoding.SerializerFactory`). Ogni implementazione, di quest'ultima interfaccia deve contenere i seguenti metodi:
  - `getSerializerAs(java.lang.String mechanismType)`
    Ritorna il serializzatore per il tipo/meccanismo (d'elaborazione XML) specificato (`return new IndirizzoSer()` nel nostro esempio).

- `getSupportedMechanismTypes()`
  Ritorna una lista di tipi/meccanismi supportati da questo `SerializerFactory` (nell'esempio `Constants.AXIS_SAX` ,ossia meccanismi d'elaborazione XML del parser SAX).

La seguente classe, `IndirizzoSer`, rappresenta il serializzatore per un oggetto di tipo `Indirizzo`:

```
package samples.my_axis.Indirizzo_ser;

import samples.my_axis.Indirizzo_ser.Indirizzo;

import org.apache.axis.encoding.SerializationContext;
import org.apache.axis.encoding.Serializer;
import org.apache.axis.message.SOAPHandler;
import org.apache.axis.Constants;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.apache.axis.Constants;
import org.apache.axis.wsdl.fromJava.Types;

import javax.xml.rpc.namespace.QName;
import java.io.IOException;
import java.util.Hashtable;

public class IndirizzoSer implements Serializer
{
    public void serialize(QName name, Attributes attributes,
                          Object value, SerializationContext
context)
        throws IOException
    {
        if (!(value instanceof Indirizzo))
            throw new IOException("Can't serialize a " +
value.getClass().getName() + " with a
IndirizzoSerializer.");
        Indirizzo indirizzo = (Indirizzo)value;

        context.startElement(name, attributes);
        context.serialize(new QName("nome_via"), null,
indirizzo.nome_via, String.class);
        context.serialize(new QName("num_civico"), null,
indirizzo.num_civico, Integer.class);
        context.serialize(new QName("CAP"), null,
indirizzo.CAP, Integer.class);
        context.serialize(new QName("citta"), null,
indirizzo.citta, String.class);
        context.serialize(new QName("provincia"), null,
indirizzo.provincia, String.class);
        context.endElement();
    }
    public String getMechanismType() { return
Constants.AXIS_SAX; }
```

```
    public boolean writeSchema(Types types) throws Exception
{
        return false;
    }
}
```

I metodi presenti in questa classe sono utilizzati internamente dal "motore" Axis per serializzare il tipo di dato in esame (`Indirizzo`).
Analizziamo il codice:

- `public class IndirizzoSer implements Serializer`
  Quest'interfaccia descrive un serializzatore Axis, e prevede due metodi:
  - `public void serialize(QName name, Attributes attributes,Object value, SerializationContext context)`
    Indica l'elemento (`name`) da serializzare con gli attributi (`attributes`) ed il valore (`value`) indicati, per il contesto di serializzazione specificato (`context`). Sarà compito del "motore" Axis fornire questi valori nel momento in cui sia necessario serializzare un elemento del tipo in esame. Il contesto di serializzazione, fra gli altri, prevede i seguenti metodi:
    - `context.startElement(QName qName, org.xml.sax.Attributes attributes);`
      Scrive, nel messaggio SOAP in preparazione, il tag di apertura per l'elemento `qName` aggiungendo eventuali attributi (`attributes`).
    - `context.serialize(QName qName, org.xml.sax.Attributes attributes, java.lang.Object value, java.lang.Class javaType);`
      Serializza il valore `value` (di tipo `javaType`), nel messaggio SOAP in preparazione, come elemento chiamato `qName`; vengono tenuti in considerazione eventuali attributi (`attributes`).
    - `context.endElement();`
      Scrive, nel messaggio SOAP in preparazione, il tag di chiusura per l'elemento aperto dal corrispondente tag di apertura.

  - `public boolean writeSchema(Types types)`
    Questo metodo è utilizzabile per scrivere lo schema XML, per il tipo `Types` specificato, all'interno di un elemento `types` in un documento WSDL.
    Deve ritornare `true` se inseriamo lo schema XML nel documento WSDL, `false` altrimenti.
    L'oggetto di tipo `Types` fa parte del package `org.apache.axis.wsdl.fromJava.Types`. Questa classe è utilizzata per serializzare una classe Java in uno schema XML. Al suo interno troviamo metodi per creare/gestire nodi dello schema con relativi valori, attributi, namespace, ecc.

Serializzatore per Rubrica

In modo analogo a quanto visto per gli oggetti di tipo `Indirizzo`, anche per gli oggetti di tipo `Rubrica` occorre definire il `SerializerFactory` ed il `Serializer`.

La seguente classe, `RubricaSerFactory`, ha il compito principale di istanziare il serializzatore per un oggetto di tipo `Rubrica`:

```
package samples.my_axis.Rubrica;

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;

import javax.xml.rpc.namespace.QName;
import java.io.IOException;

import org.apache.axis.encoding.Serializer;
import org.apache.axis.encoding.SerializerFactory;
import org.apache.axis.encoding.SerializationContext;
import org.apache.axis.encoding.Deserializer;
import org.apache.axis.encoding.DeserializerFactory;
import org.apache.axis.encoding.DeserializationContext;
import org.apache.axis.encoding.DeserializerImpl;
import org.apache.axis.Constants;

import java.util.Iterator;
import java.util.Vector;

/**
 * SerializerFactory per Rubrica
 *
 */

public class RubricaSerFactory implements SerializerFactory
{
    private Vector mechanisms;

    public RubricaSerFactory() {
    }
    public javax.xml.rpc.encoding.Serializer
getSerializerAs(String mechanismType) {
        return new RubricaSer();
    }
    public Iterator getSupportedMechanismTypes() {
        if (mechanisms == null) {
            mechanisms = new Vector();
            mechanisms.add(Constants.AXIS_SAX);
        }
        return mechanisms.iterator();
    }
}
```

Come si vede il codice ed il funzionamento è del tutto analogo a quanto già visto con `IndirizzoSerFactory`, ma ovviamente, in questo caso viene istanziato il serializzatore per gli oggetti di tipo `Rubrica` (`return new RubricaSer()`).

La seguente classe, `RubricaSer`, rappresenta il serializzatore per un oggetto di tipo `Rubrica`:

```
package samples.my_axis.Rubrica;
```

```
import samples.my_axis.Rubrica.Rubrica;

import org.apache.axis.encoding.SerializationContext;
import org.apache.axis.encoding.Serializer;
import org.apache.axis.message.SOAPHandler;
import org.apache.axis.Constants;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.apache.axis.Constants;
import org.apache.axis.wsdl.fromJava.Types;

import javax.xml.rpc.namespace.QName;
import java.io.IOException;
import java.util.Hashtable;
import java.util.Vector;
import java.util.TreeMap;
import java.util.*;

import samples.my_axis.Indirizzo_ser.Indirizzo;

public class RubricaSer implements Serializer
{

    public void serialize(QName name, Attributes attributes,
                          Object value, SerializationContext
context)
        throws IOException
    {
        if (!(value instanceof Rubrica))
            throw new IOException("Can't serialize a " +
value.getClass().getName() + " with a RubricaSerializer.");
        Rubrica rubrica = (Rubrica)value;

        context.startElement(name, attributes);
        context.serialize(new QName("nome"), null,
rubrica.nome, String.class);
        context.serialize(new QName("cognome"), null,
rubrica.cognome, String.class);
        context.serialize(new QName("num_telefono"), null,
rubrica.num_telefono, Object.class);
        context.serialize(new QName("indirizzi"), null,
rubrica.indirizzi, Indirizzo.class);
        context.endElement();
    }
    public String getMechanismType() { return
Constants.AXIS_SAX; }

    public boolean writeSchema(Types types) throws Exception
{
        return false;
    }
}
```

Come si vede il codice ed il funzionamento è del tutto analogo a quanto già visto con `IndirizzoSer`. In questo caso, oltre a dati di tipo semplice, viene serializzato anche

un dato di tipo complesso: `indirizzi`. Si tratta di un array di tipo `Indirizzo`. Apache Axis, come vedremo nel successivo paragrafo messaggi, serializza questo dato rispettando le specifiche W3C sugli array. Ciascun elemento di questo array, viene serializzato utilizzando il serializzatore `IndirizzoSer`.

Deserializzatore per Indirizzo

Per l'esempio in esame, così come sul lato server si ha la necessità di serializzare un dato di tipo `Indirizzo` (risposta alla richiesta del client), sul lato client si ha la necessità di deserializzare un dato di tipo `Indirizzo` (per ricavare la risposta inviata dal server).
La seguente classe, `IndirizzoDeserFactory`, ha il compito principale di istanziare il deserializzatore per un oggetto di tipo `Indirizzo`:

```
package samples.my_axis.Indirizzo_ser;

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;

import javax.xml.rpc.namespace.QName;
import java.io.IOException;

import org.apache.axis.encoding.Serializer;
import org.apache.axis.encoding.SerializerFactory;
import org.apache.axis.encoding.SerializationContext;
import org.apache.axis.encoding.Deserializer;
import org.apache.axis.encoding.DeserializerFactory;
import org.apache.axis.encoding.DeserializationContext;
import org.apache.axis.encoding.DeserializerImpl;
import org.apache.axis.Constants;

import java.util.Iterator;
import java.util.Vector;

/**
 * DeserializerFactory per IndirizzoDeser
 *
 */

public class IndirizzoDeserFactory implements
DeserializerFactory {
    private Vector mechanisms;

    public IndirizzoDeserFactory() {
    }
    public javax.xml.rpc.encoding.Deserializer
getDeserializerAs(String mechanismType) {
        return new IndirizzoDeser();
    }
    public Iterator getSupportedMechanismTypes() {
        if (mechanisms == null) {
            mechanisms = new Vector();
            mechanisms.add(Constants.AXIS_SAX);
        }
        return mechanisms.iterator();
    }
}
```

Le definizioni, i concetti ed i principi di funzionamento ricalcano quelli già visti in precedenza con la classe `IndirizzoSerFactory`, ma ovviamente si fa riferimento all'interfaccia `DeserializerFactory` e viene istanziato il deserializzatore `IndirizzoDeser`.

La seguente classe, `IndirizzoDeser`, rappresenta il deserializzatore per un oggetto di tipo `Indirizzo`:

```
package samples.my_axis.Indirizzo_ser;

import samples.my_axis.Indirizzo_ser.Indirizzo;

import org.apache.axis.encoding.DeserializationContext;
import org.apache.axis.encoding.Deserializer;
import org.apache.axis.encoding.DeserializerImpl;
import org.apache.axis.encoding.FieldTarget;
import org.apache.axis.Constants;
import org.apache.axis.message.SOAPHandler;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;

import javax.xml.rpc.namespace.QName;
import java.io.IOException;
import java.util.Hashtable;

public class IndirizzoDeser extends DeserializerImpl
{

    private Hashtable typesByMemberName = new Hashtable();

    public IndirizzoDeser()
    {
        typesByMemberName.put("nome_via",
Constants.XSD_STRING);
        typesByMemberName.put("num_civico",
Constants.XSD_INT);
        typesByMemberName.put("CAP", Constants.XSD_INT);
        typesByMemberName.put("citta",
Constants.XSD_STRING);
        typesByMemberName.put("provincia",
Constants.XSD_STRING);
        value = new Indirizzo();
    }

    public SOAPHandler onStartChild(String namespace,
                                    String localName,
                                    String prefix,
                                    Attributes attributes,
                                    DeserializationContext
context)
        throws SAXException
    {
        QName typeQName =
(QName)typesByMemberName.get(localName);
        if (typeQName == null)
```

```
            throw new SAXException("Invalid element in Data
struct - " + localName);

        // These can come in either order.
        Deserializer dSer =
context.getDeserializerForType(typeQName);
        try {
            dSer.registerValueTarget(new FieldTarget(value,
localName));
        } catch (NoSuchFieldException e) {
            throw new SAXException(e);
        }

        if (dSer == null)
            throw new SAXException("No deserializer for a "
+ typeQName + "???");

        return (SOAPHandler) dSer;
    }
}
```

Analizziamo meglio il codice:

- `public class IndirizzoDeser extends DeserializerImpl`
  La classe `DeserializerImpl` (appartenente al package `org.apache.axis.encoding.DeserializerImpl`) contiene tutti i deserializzatori per i dati base di Java (come interi, float, stringhe, array, ecc.).

- `typesByMemberName.put("nome_via", Constants.XSD_STRING);`
  Con questa istruzione, all'interno del costruttore `IndirizzoDeser`, aggiungiamo alla nostra `Hashtable` il nome di un elemento (in questo caso `nome_via`) con il corrispondente tipo (in questo caso `Constants.XSD_STRING`, ossia `xsi:type="xsd:string"`) che troveremo nella parte di messaggio SOAP contenente un dato di tipo `Indirizzo`.

- `public SOAPHandler onStartChild(String namespace, String localName, String prefix, Attributes attributes, DeserializationContext context)`
  Questo metodo viene richiamato, dal "motore" Axis, ogni qual volta si incontra un tag d'apertura nella parte di messaggio SOAP che contiene un dato di tipo `Indirizzo`.
  Questo metodo prevede i seguenti parametri:
  - `String namespace`
    Il namespace dell'elemento incontrato.
  - `String localName`
    Il nome dell'elemento incontrato.
  - `String prefix`
    Il prefisso (legato alla definizione del namespace) dell'elemento incontrato.
  - `Attributes attributes`

Eventuali attributi dell'elemento incontrato.

- `DeserializationContext context`
  Il contesto di deserializzazione.

Questo metodo controlla che l'elemento incontrato (attraverso il suo nome) sia stato catalogato all'interno della `Hashtable`. In caso negativo ritorna un errore, mentre in caso affermativo recupera il suo nome completo (un `QName`). Questo controllo avviene attraverso le seguenti istruzioni:

- `QName typeQName = (QName)typesByMemberName.get(localName);`
- `if (typeQName == null) throw new SAXException("Invalid element in Data struct - " + localName);`

Se il controllo ha dato esito positivo, si procede a recuperare il valore dell'elemento incontrato attraverso le seguenti istruzioni:

- `Deserializer dSer = context.getDeserializerForType(typeQName);`
  Ricerca il deserializzatore per il tipo di dato dell'elemento incontrato.

- ```
  try
  {
   dSer.registerValueTarget(new FieldTarget(value,
                                            localName));
  } catch (NoSuchFieldException e) {
  throw new SAXException(e);}
  ```
  Con il metodo `registerValueTarget` è possibile registrare il nome di un elemento incontrato, ma il cui valore non sia ancora disponibile. Questo avviene per i dati di tipo complesso che sono serializzati, rispettando le specifiche W3C, utilizzando il multi-referecing (si veda l'attributo `href` nei messaggi SOAP). Quando i valori degli elementi registrati sono noti, allora automaticamente per ognuno di questi, viene eseguito il metodo `setValue` che recupera il valore dell'elemento. In questo esempio il metodo `registerValueTarget` prevede come parametro un nuovo oggetto di tipo `FieldTarget` attraverso cui si specifica l'oggetto obiettivo e l'elemento obiettivo della registrazione.
  Il costruttore `FieldTarget` (appartenente al package `org.apache.axis.encoding.FieldTarget`) ha la seguente struttura:
  ```
  public
  FieldTarget(java.lang.Object targetObject,
  java.lang.reflect.Field targetField)
  ```

- `if (dSer == null) throw new SAXException("No deserializer for a " + typeQName + "???");`
  Se non è stato possibile trovare il deserializzatore cercato ritorna un errore.

- `return (SOAPHandler) dSer;`
  Ritorna, al "motore" Axis, il deserializzatore con gli opportuni valori settati.
  ```

Deserializzatore per Rubrica

Anche per `Rubrica` si procede a definire il suo deserializzatore in modo analogo a quanto fatto per `Indirizzo`.
La seguente classe, `Rubrica DeserFactory`, ha il compito principale di istanziare il deserializzatore per un oggetto di tipo `Rubrica`:

```
package samples.my_axis.Rubrica;

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;

import javax.xml.rpc.namespace.QName;
import java.io.IOException;

import org.apache.axis.encoding.Serializer;
import org.apache.axis.encoding.SerializerFactory;
import org.apache.axis.encoding.SerializationContext;
import org.apache.axis.encoding.Deserializer;
import org.apache.axis.encoding.DeserializerFactory;
import org.apache.axis.encoding.DeserializationContext;
import org.apache.axis.encoding.DeserializerImpl;
import org.apache.axis.Constants;

import java.util.Iterator;
import java.util.Vector;

/**
 * DeserializerFactory per Rubrica
 *
 */

public class RubricaDeserFactory implements
DeserializerFactory {
    private Vector mechanisms;

    public RubricaDeserFactory() {
    }
    public javax.xml.rpc.encoding.Deserializer
getDeserializerAs(String mechanismType) {
        return new RubricaDeser();
    }
    public Iterator getSupportedMechanismTypes() {
        if (mechanisms == null) {
            mechanisms = new Vector();
            mechanisms.add(Constants.AXIS_SAX);
        }
        return mechanisms.iterator();
    }
}
```

La seguente classe, `RubricaDeser`, rappresenta il deserializzatore per un oggetto di tipo `Rubrica`:

```
package samples.my_axis.Rubrica;
```

```
import samples.my_axis.Rubrica.Rubrica;

import org.apache.axis.encoding.DeserializationContext;
import org.apache.axis.encoding.Deserializer;
import org.apache.axis.encoding.DeserializerImpl;
import org.apache.axis.encoding.FieldTarget;
import org.apache.axis.Constants;
import org.apache.axis.message.SOAPHandler;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;

import javax.xml.rpc.namespace.QName;
import java.io.IOException;
import java.util.Hashtable;

public class RubricaDeser extends DeserializerImpl
{

    private Hashtable typesByMemberName = new Hashtable();

    public RubricaDeser()
    {
        typesByMemberName.put("nome", Constants.XSD_STRING);
        typesByMemberName.put("cognome",
Constants.XSD_STRING);
        typesByMemberName.put("num_telefono",
Constants.SOAP_ARRAY);
        typesByMemberName.put("indirizzi",
Constants.SOAP_ARRAY);
        value = new Rubrica();
    }

    public SOAPHandler onStartChild(String namespace,
                                    String localName,
                                    String prefix,
                                    Attributes attributes,
                                    DeserializationContext
context)
        throws SAXException
    {
        QName typeQName =
(QName)typesByMemberName.get(localName);
        if (typeQName == null)
            throw new SAXException("Invalid element in Data
struct - " + localName);

        // These can come in either order.
        Deserializer dSer =
context.getDeserializerForType(typeQName);
        try {
            dSer.registerValueTarget(new FieldTarget(value,
localName));
        } catch (NoSuchFieldException e) {
            throw new SAXException(e);
        }
```

```
        if (dSer == null)
            throw new SAXException("No deserializer for a "
+ typeQName + "???");

        return (SOAPHandler) dSer;
    }
}
```

Tutto è analogo al deserializzatore visto per `Indirizzo`.

Pubblicazione del servizio

A questo punto è necessario comunicare al "motore" Axis l'associazione fra i tipi di dato complesso che sono stati definiti (`Indirizzo` e `Rubrica`) ed i relativi serializzatori/deserializzatori creati ad hoc. In questo modo, ogni qual volta sia necessario serializzare/deserializzare uno di questi dati, si possono utilizzare le classi opportune (`IndirizzoSerFactory`, `RubricaSerFactory/IndirizzoDeserFactory`, `RubricaDeserFactory`).
Questa associazione viene comunicata dal programmatore all'interno del documento WSDD che si utilizza normalmente per pubblicare il servizio.
Si noti che queste informazioni sono utili solamente per il lato server, e per gli scopi di questo servizio, non sarebbe strettamente necessario definire il legame con i deserializzatori creati ad hoc. Infatti il server ha solamente la necessità di inviare, quindi serializzare (e non deserializzare), i dati di tipo complesso definiti. Nel codice client, come vedremo, avremo la necessità di definire i legami fra tipi di dati complessi definiti ed i deserializzatori creati ad hoc.
Il documento WSDD, per l'esempio in esame, è il seguente:

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

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


  <service name="Rubrica_servizio" provider="java:RPC">
      <parameter name="className"
value="samples.my_axis.Rubrica.Rubrica_server"/>
      <parameter name="allowedMethods" value="getRubrica"/>

      <typeMapping
        xmlns:ns="http://Rubrica.my_axis.samples"
        qname="ns:Rubrica"
        type="java:samples.my_axis.Rubrica.Rubrica"

serializer="samples.my_axis.Rubrica.RubricaSerFactory"

deserializer="samples.my_axis.Rubrica.RubricaDeserFactory"

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      />

      <typeMapping
        xmlns:ns="http://Indirizzo_ser.my_axis.samples"
        qname="ns:Indirizzo"
        type="java:samples.my_axis.Indirizzo_ser.Indirizzo"
```

```
serializer="samples.my_axis.Indirizzo_ser.IndirizzoSerFactor
y"

deserializer="samples.my_axis.Indirizzo_ser.IndirizzoDeserFa
ctory"

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      />

  </service>
</deployment>
```

Nell'elemento `service`, come già descritto in precedenza, viene indicato il nome del servizio (`Rubrica_servizio`), il metodo pubblicato (`getRubrica`) e la classe Java (`samples.my_axis.Rubrica.Rubrica_server`) che contiene questo metodo.
La novità di questo documento WSDD riguarda gli elementi `typeMapping`.
All'interno di un elemento `typeMapping` si definisce, per un certo tipo di dato, il serializzatore ed il deserializzatore opportuno. Vediamo un esempio:

- `xmlns:ns="http://Rubrica.my_axis.samples"`
  `qname="ns:Rubrica"`
  Specifica il valore assunto dall'attributo `xsi:type` nei messaggi SOAP per gli oggetti della classe in esame (specificata nell'elemento `type` che segue). Inoltre questo valore (che è un nome completo, defininendo anche il namespace corrispondente) è utilizzato internamente dal "motore" Axis per registrare l'associazione fra la classe in esame ed il corrispondenti serializzatore/deserializzatore (specificato nell'elemento `serializer/deserializer` che segue).

- `type="java:samples.my_axis.Rubrica.Rubrica"`
  Classe che definisce il tipo di dato da serializzare/deserializzare.

- `serializer="samples.my_axis.Rubrica.RubricaSerFactory"`
  Specifica la classe da richiamare ogni qual volta sia necessario serializzare il tipo di dato in esame (si noti che a sua volta `RubricaSerFactory` richiama `RubricaSer`, il serializzatore vero e proprio).

- `deserializer="samples.my_axis.Rubrica.RubricaDeserFact`
  `ory"`
  Specifica la classe da richiamare ogni qual volta sia necessario deserializzare il tipo di dato in esame (si noti che a sua volta `RubricaDeserFactory` richiama `RubricaDeser`, il deserializzatore vero e proprio).

- `encodingStyle="http://schemas.xmlsoap.org/soap/encodin`
  `g/"`
  Specifica il valore dell'attributo di `encodingStyle` per il tipo di dato in esame.

L'altro elemento `typeMapping` del documento WSDD definisce, in modo analogo, le associazioni per il dato complesso `Indirizzo`.

Client Rubrica

Il codice del client è il seguente:

```
package samples.my_axis.Rubrica;

import org.apache.axis.AxisFault;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.utils.Options;

import javax.xml.rpc.ParameterMode;
import javax.xml.rpc.namespace.QName;


public class Rubrica_client
{
    public static void main(String [] args) throws Exception
    {
        Options options = new Options(args);

        String endpointURL =
"http://localhost:"+options.getPort()+"/axis/services/Rubric
a_servizio";
        args=options.getRemainingArgs();

        Service  service = new Service();
        Call     call    = (Call) service.createCall();

        QName    qn       = new
QName("http://Rubrica.my_axis.samples","Rubrica");
        QName    qn1      = new
QName("http://Indirizzo_ser.my_axis.samples","Indirizzo_ser"
);


call.registerTypeMapping(samples.my_axis.Complex.Complex.cla
ss, qn,
                         new
samples.my_axis.Rubrica.RubricaSerFactory(),
                         new
samples.my_axis.Rubrica.RubricaDeserFactory());


call.registerTypeMapping(samples.my_axis.Indirizzo_ser.Indir
izzo.class, qn1,
                         new
samples.my_axis.Indirizzo_ser.IndirizzoSerFactory(),
                         new
samples.my_axis.Indirizzo_ser.IndirizzoDeserFactory());

        String result;
        try {
           call.setTargetEndpointAddress( new
java.net.URL(endpointURL) );
```

```
            call.setOperationName(new
QName("Complex_servizio","getRubrica"));
            call.addParameter(
"arg1",org.apache.axis.encoding.XMLType.XSD_STRING,Parameter
Mode.PARAM_MODE_IN );
            call.addParameter(
"arg2",org.apache.axis.encoding.XMLType.XSD_STRING,Parameter
Mode.PARAM_MODE_IN );
            String Nome=new String("Paolo");
            String Cognome=new String("Rossi");
            call.setReturnType(qn);
            samples.my_axis.Rubrica.Rubrica rubrica=new
samples.my_axis.Rubrica.Rubrica();
            rubrica=(samples.my_axis.Rubrica.Rubrica)
call.invoke( new Object[] { Nome,Cognome } );
            System.out.println("Rubrica_client in azione,
risposta ricevuta:");
            System.out.println("Nome: "+rubrica.nome);
            System.out.println("Cognome: "+rubrica.cognome);
            System.out.println("Numeri telefonici...");
      for(int i=1; i<= rubrica.num_telefono.length; i++)
System.out.println(rubrica.num_telefono[i-1]);
            System.out.println("Indirizzi...");
      for(int i=1; i<= rubrica.indirizzi.length; i++)
         {samples.my_axis.Indirizzo_ser.Indirizzo
ind=(samples.my_axis.Indirizzo_ser.Indirizzo)
rubrica.indirizzi[i-1];
                  System.out.println("Indirizzo "+i);
           System.out.println("Nome via: "+ind.nome_via);
         System.out.println("Numero civico:
"+ind.num_civico);
               System.out.println("CAP :"+ind.CAP);
               System.out.println("Città: "+ind.citta);
               System.out.println("PRovincia:
"+ind.provincia);
         }
            System.out.println("OK");

        } catch (AxisFault fault) {
            result = "Error : " + fault.toString();
        }

            }
}
```

Le novità introdotte nel client sono le seguenti:

- `QName qn = new QName("http://Rubrica.my_axis.samples","Rubrica");`
  Definisco il nome completo assunto da un elemento di tipo `Rubrica` all'interno di messaggio SOAP.

- `QName qn1 = new QName("http://Indirizzo_ser.my_axis.samples","Indirizzo_ser");`

Definisco il nome completo assunto da un elemento di tipo `Indirizzo` all'interno di messaggio SOAP.

- ```
  call.registerTypeMapping(samples.my_axis.Rubrica.Rubri
  ca.class, qn,
  new samples.my_axis.Rubrica.RubricaSerFactory(),
  new samples.my_axis.Rubrica.RubricaDeserFactory());
  ```
  Con il metodo `registerTypeMapping` si procede, sul lato client, a registrare l'associazione fra il tipo di dato `Rubrica` ed il serializzatore/deserializzatore creato ad hoc. Il metodo prevede quattro parametri:
  - `samples.my_axis.Rubrica.Rubrica.class`
    La classe che definisce gli oggetti di tipo `Rubrica`.
  - `qn`
    Il nome completo per identificare gli elementi di tipo `Rubrica` all'interno di un messaggio SOAP.
  - `new samples.my_axis.Rubrica.RubricaSerFactory()`
    La classe che gestisce il serializzatore per gli elementi di tipo `Rubrica`.
  - `new samples.my_axis.Rubrica.RubricaDeserFactory()`
    La classe che gestisce il deserializzatore per gli elementi di tipo `Rubrica`.

- ```
  call.registerTypeMapping(samples.my_axis.Indirizzo_ser
  .Indirizzo.class, qn1,
  new
  samples.my_axis.Indirizzo_ser.IndirizzoSerFactory(),
  new
  samples.my_axis.Indirizzo_ser.IndirizzoDeserFactory())
  ;
  ```
  Con il metodo `registerTypeMapping` si procede, sul lato client, a registrare l'associazione fra il tipo di dato `Indirizzo` ed il serializzatore/deserializzatore creato ad hoc. Il metodo prevede quattro parametri:
  - `samples.my_axis.Indirizzo_ser.Indirizzo.class`
    La classe che definisce gli oggetti di tipo `Indirizzo`.
  - `qn1`
    Il nome completo per identificare gli elementi di tipo `Indirizzo` all'interno di un messaggio SOAP.
  - `new samples.my_axis.Indirizzo_ser.IndirizzoSerFactory()`
    La classe che gestisce il serializzatore per gli elementi di tipo `Indirizzo`.

    - `new samples.my_axis.Indirizzo_ser.IndirizzoDeserFactory()`
      La classe che gestisce il deserializzatore per gli elementi di tipo `Indirizzo`.

- `call.setReturnType(qn);`
  Informa il client che la risposta SOAP del server è del tipo `Rubrica` (individuato dal proprio nome completo).

Si noti, che per gli scopi di questo servizio, non sarebbe strettamente necessario definire il legame con i serializzatori creati ad hoc. Infatti il client ha solamente la necessità di recuperare, quindi deserializzare (e non serializzare), i dati di tipo complesso definiti.

Messaggi SOAP di Rubrica

Il messaggio SOAP di richiesta inviato dal client è il seguente:

```
POST /axis/services/Rubrica_servizio HTTP/1.0
Content-Length: 554
Host: localhost
Content-Type: text/xml; charset=utf-8
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
 SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
"
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
 <SOAP-ENV:Body>
  <ns1:getRubrica xmlns:ns1="Complex_servizio">
   <arg1 xsi:type="xsd:string">Paolo</arg1>
   <arg2 xsi:type="xsd:string">Rossi</arg2>
  </ns1:getRubrica>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In pratica si richiede al server di fornire la rubrica di `Paolo Rossi` (o meglio l'oggetto di tipo `Rubrica` corrispondente a questo nome e cognome).

Il messaggio SOAP di risposta inviato dal server è il seguente:

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 2212
Servlet-Engine: Tomcat Web Server/3.2.3 (JSP 1.1; Servlet
2.2; Java 1.4.0; Windows XP 5.1 x86; java.vendor=Sun
Microsystems Inc.)
```

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <SOAP-ENV:Body>
  <ns1:getRubricaResponse
   SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
"
   xmlns:ns1="Rubrica_servizio">
   <getRubricaResult href="#id0"/>
  </ns1:getRubricaResponse>
  <multiRef id="id0" SOAP-ENC:root="0"
   xsi:type="ns2:Rubrica"
   xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:ns2="http://Rubrica.my_axis.samples">
   <nome xsi:type="xsd:string">Paolo</nome>
   <cognome xsi:type="xsd:string">Rossi</cognome>
   <num_telefono href="#id1"/>
   <indirizzi href="#id2"/>
  </multiRef>
  <multiRef id="id1" SOAP-ENC:root="0"
   xsi:type="SOAP-ENC:Array"
   SOAP-ENC:arrayType="xsd:string[2]"
   xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/">
   <item xsi:type="xsd:string">casa:02345678</item>
   <item xsi:type="xsd:string">ufficio:02123456</item>
  </multiRef>
  <multiRef id="id2" SOAP-ENC:root="0"
   xsi:type="SOAP-ENC:Array"
   SOAP-ENC:arrayType="ns3:Indirizzo[2]"
   xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:ns3="http://Indirizzo_ser.my_axis.samples">
   <item href="#id3"/>
   <item href="#id4"/>
  </multiRef>
  <multiRef id="id3" SOAP-ENC:root="0"
   xsi:type="ns4:Indirizzo"
   xmlns:ns4="http://Indirizzo_ser.my_axis.samples"
   xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/">
   <nome_via xsi:type="xsd:string">via Emilia</nome_via>
   <num_civico xsi:type="xsd:int">1</num_civico>
   <CAP xsi:type="xsd:int">100</CAP>
   <citta xsi:type="xsd:string">Modena</citta>
   <provincia xsi:type="xsd:string">Mo</provincia>
  </multiRef>
  <multiRef id="id4" SOAP-ENC:root="0"
   xsi:type="ns5:Indirizzo"
   xmlns:ns5="http://Indirizzo_ser.my_axis.samples"
```

```
   xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/">
   <nome_via xsi:type="xsd:string">via Italia</nome_via>
   <num_civico xsi:type="xsd:int">6</num_civico>
   <CAP xsi:type="xsd:int">101</CAP>
   <citta xsi:type="xsd:string">Bologna</citta>
   <provincia xsi:type="xsd:string">Bo</provincia>
  </multiRef>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Come si può notare essendo la risposta di tipo complesso (`Rubrica`), nel rispetto delle specifiche W3C, questo viene serializzato sfruttando il multi-referecing (`href="#id0"`). Quando viene esplicitato il riferimento `id0`, tutti i dati di tipo semplice sono riportati direttamente (`nome` e `cognome`), mentre i dati di tipo complesso (`num_telefono` ed `indirizzi`) a loro volta utilizzano il multi-referecing:

- ```
  <multiRef id="id0" SOAP-ENC:root="0"
   xsi:type="ns2:Rubrica"
   xmlns:SOAP-
   ENC="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:ns2="http://Rubrica.my_axis.samples">
   <nome xsi:type="xsd:string">Paolo</nome>
   <cognome xsi:type="xsd:string">Rossi</cognome>
   <num_telefono href="#id1"/>
   <indirizzi href="#id2"/>
  </multiRef>
  ```
  Si noti che l'elemento di tipo `Rubrica` è identificato attraverso il nome completo indicato nel documento WSDD durante la pubblicazione del servizio.

E così via.

Anche gli array sono serializzati nel rispetto delle specifiche W3C:

- ```
  <multiRef id="id1" SOAP-ENC:root="0"
   xsi:type="SOAP-ENC:Array"
   SOAP-ENC:arrayType="xsd:string[2]"
   xmlns:SOAP-
   ENC="http://schemas.xmlsoap.org/soap/encoding/">
   <item xsi:type="xsd:string">casa:02345678</item>
   <item xsi:type="xsd:string">ufficio:02123456</item>
  </multiRef>
  ```
  Questa è la serializzazione per un array contenente due stringhe.

- ```
  <multiRef id="id2" SOAP-ENC:root="0"
   xsi:type="SOAP-ENC:Array"
   SOAP-ENC:arrayType="ns3:Indirizzo[2]"
   xmlns:SOAP-
   ENC="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:ns3="http://Indirizzo_ser.my_axis.samples">
   <item href="#id3"/>
   <item href="#id4"/>
  </multiRef>
  ```
  Questa è la serializzazione per un array contenente due elementi di tipo `Indirizzo`.
  ```

Ad esempio, il primo elemento di questo array viene serializzato nel modo seguente:

```
o  <multiRef id="id3" SOAP-ENC:root="0"
   xsi:type="ns4:Indirizzo"
   xmlns:ns4="http://Indirizzo_ser.my_axis.samples"
   xmlns:SOAP-
   ENC="http://schemas.xmlsoap.org/soap/encoding/">
   <nome_via xsi:type="xsd:string">via
                                  Emilia</nome_via>
   <num_civico xsi:type="xsd:int">1</num_civico>
   <CAP xsi:type="xsd:int">100</CAP>
   <citta xsi:type="xsd:string">Modena</citta>
   <provincia xsi:type="xsd:string">Mo</provincia>
   </multiRef>
```

Si noti che tutti gli elementi di tipo `Indirizzo` sono identificati attraverso il nome completo indicato nel documento WSDD durante la pubblicazione del servizio.

Capitolo 10-Integrazione del sistema MOMIS con il protocollo SOAP

Il sistema Momis

Il sistema MOMIS (Mediator EnvirOnment for Multiple Information Source) è stato progettato per fornire un accesso integrato ad informazioni eterogenee memorizzate sia in database di tipo tradizionale (ad esempio relazionali e object oriented) o file system, sia in sorgenti di tipo semistrutturato.
MOMIS nasce all'interno del progetto MURST 40% INTERDATA, come collaborazione fra l'Università di Modena e Reggio Emilia e l'Università di Milano e Brescia.
Attualmente, per la gestione dei dati, MOMIS utilizza l'architettura ad oggetti distribuiti CORBA.

L'architettura di MOMIS

Seguendo l'architettura di riferimento I^3, in MOMIS si possono distinguere cinque componenti principali (come si può vedere nella figura 21):

1. Wrapper: posti al di sopra di ciascuna sorgente, sono i moduli che rappresentano l'interfaccia tra il mediatore vero e proprio e le sorgenti locali di dati.
La loro funzione è duplice:

- in fase di integrazione, forniscono la descrizione delle informazioni in essa contenute. Questa descrizione viene fornita attraverso il linguaggio ODL_{I3};
- in fase di query processing, traducono la query ricevuta dal mediatore (espressa quindi nel linguaggio comune di interrogazione OQL_{I3}, definito in analogia al linguaggio OQL) in una interrogazione comprensibile (e realizzabile) dalla sorgente stessa. Devono inoltre esportare i dati ricevuti in risposta all'interrogazione, presentandoli al mediatore attraverso il modello comune di dati utilizzato dal sistema;

2. mediatore: è il cuore del sistema, ed è composto da due moduli distinti:

- Global Schema Builder (GSB): è il modulo di integrazione degli schemi locali che, partendo dalle descrizioni delle sorgenti espresse attraverso il linguaggio ODL_{I3}, genera un unico schema globale da presentare all'utente;
- Query Manager (QM): è il modulo di gestione delle interrogazioni. In particolare, genera le query in linguaggio OQL_{I3} da inviare ai wrapper partendo dalla singola query formulata dall'utente sullo schema globale. Servendosi delle tecniche di Logica Descrittiva, il QM genera automaticamente la traduzione della query sottomessa nelle corrispondenti sub-query delle singole sorgenti.

3. ODB-Tools Engine, un tool basato sulla OLCD Description Logics che compie la validazione di schemi e l'ottimizzazione di query.

4. ARTEMIS-Tool Environment, un tool che compie analisi e clustering di schemi.

5. WordNet, un database lessicale della lingua inglese, capace di individuare relazioni lessicali e semantiche fra termini.

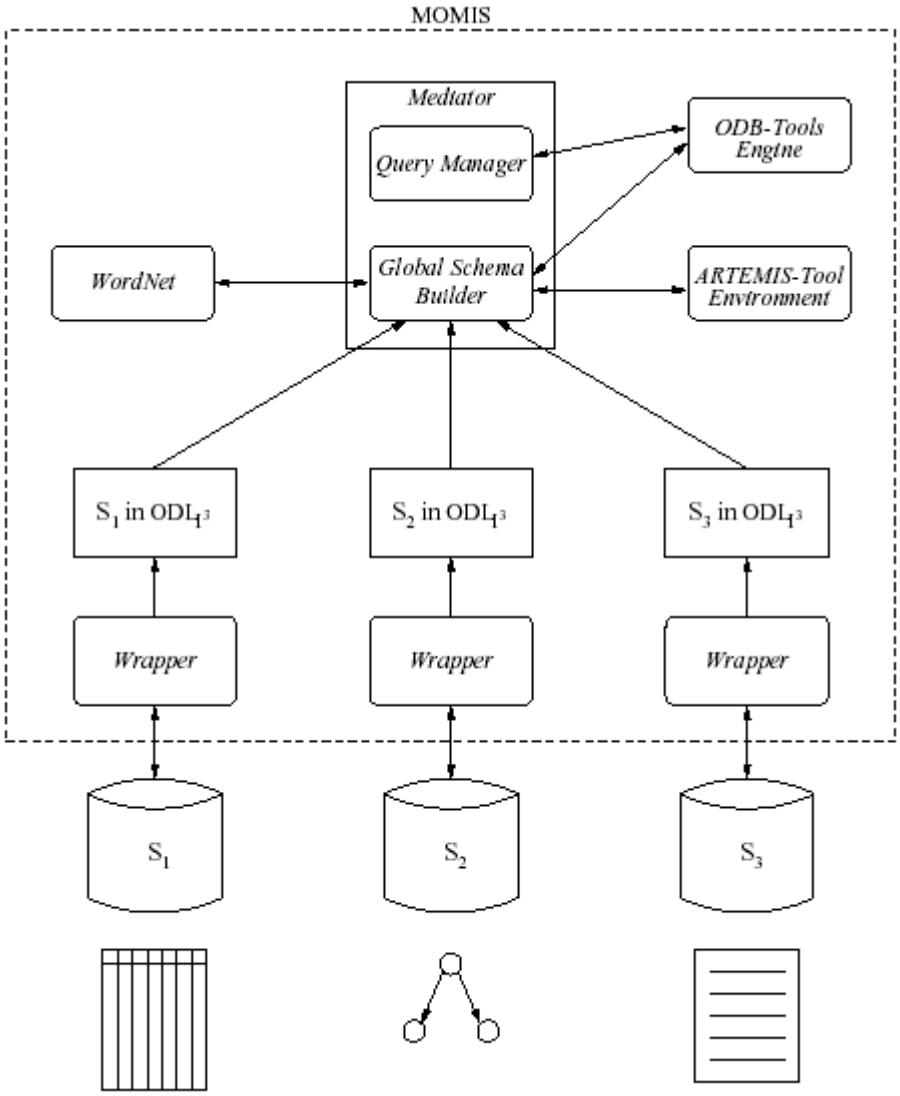


Figura 21:L'architettura MOMIS

Il processo d'integrazione delle informazioni

L'integrazione delle sorgenti informative strutturate e semistrutturate viene compiuta in modo semi-automatico, utilizzando le descrizioni degli schemi locali in linguaggio ODL_{I3} e combinando le tecniche di Description Logics e di clustering.
Come mostrato in figura, le attività compiute sono le seguenti:

1. Generazione del Thesaurus Comune, grazie al supporto di ODB-Tools e di WordNet. Durante questo passo viene costruito un Thesaurus Comune di relazioni terminologiche. Le relazioni terminologiche esprimono la conoscenza inter-schema su sorgenti diverse. Le relazioni terminologiche sono derivate in modo semi-automatico a partire dalle descrizioni degli schemi in ODL_{I3}, attraverso l'analisi strutturale (utilizzando ODB-Tools

e le tecniche di Description Logics) e di contesto (attraverso l'uso di WordNet) delle classi coinvolte.

2. Generazione dei cluster di classi ODL $_{I3}$, con il supporto dell'ambiente ARTEMIS-Tool, le relazioni terminologiche contenute nel Thesaurus vengono utilizzate per valutare il livello di affinit`a tra le classi ODL $_{I3}$ in modo da identificare le informazioni che devono essere integrate a livello globale.
A tal fine, ARTEMIS calcola i coefficienti che misurano il livello di affinità delle classi ODL $_{I3}$ basandosi sia sui nomi delle stesse, sia sugli attributi.
Le classi ODL $_{I3}$ con maggiore affinità vengono raggruppate utilizzando opportune tecniche di clustering.

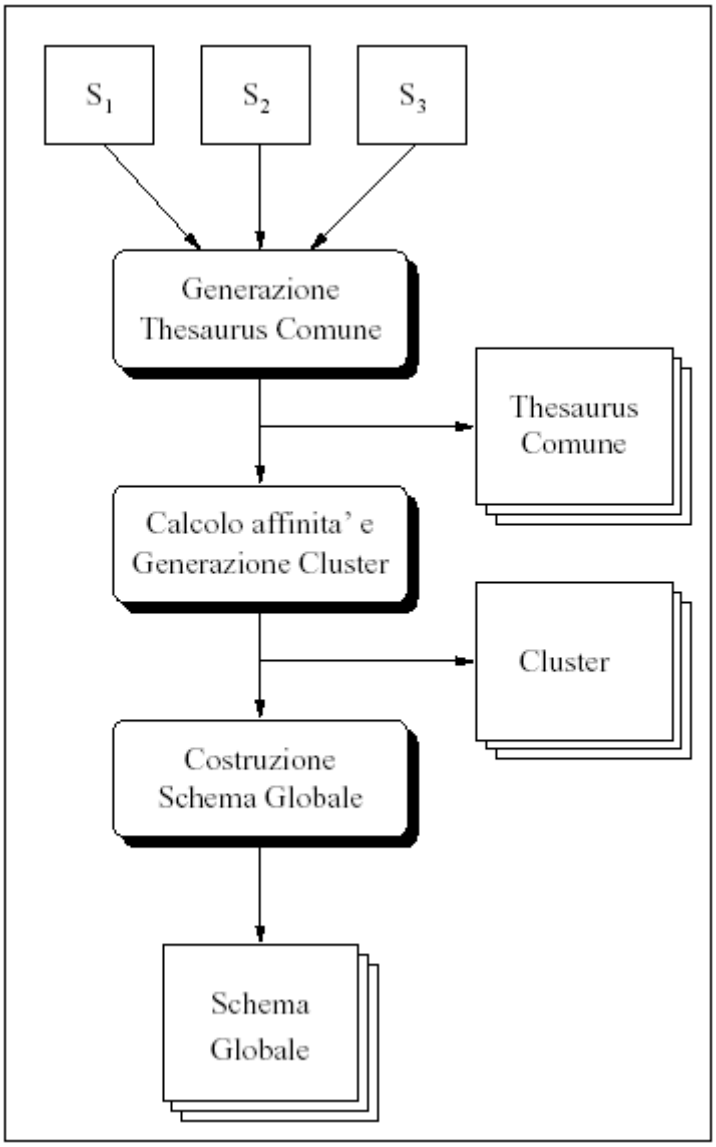


Figura 22:Le Fasi dell'integrazione delle informazioni in MOMIS

3. Costruzione dello schema globale del mediatore, i cluster di classi ODL $_{I3}$ affini sono analizzati per costruire lo schema globale del Mediatore. Per ciascun cluster viene definita una classe globale ODL $_{I3}$ che rappresenta tutte le classi locali che sono riferite al cluster,

e che è caratterizzata dall'unione dei loro attributi. L'insieme delle classi globali definite costituisce lo schema globale del Mediatore che deve essere usato per porre le query alle sorgenti locali integrate.

Il Global Schema Builder

Il Global Schema Builder è la parte del Mediatore di Momis che si preoccupa della costruzione dello schema integrato, secondo le fasi appena elencate. Come si nota dalla figura, è composto da tre componenti:

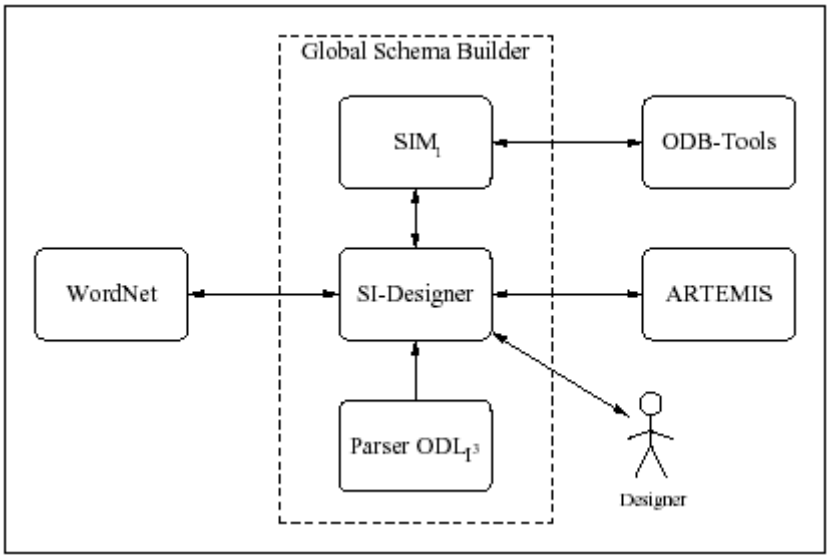


Figura 23: Architettura del Global Schema Builder

1. SIM$_I$: si occupa della generazione del Thesaurus comune, in particolare della estrazione delle relazioni dalla struttura degli schemi sorgenti e, con l'aiuto di ODB-Tools, della validazione delle relazioni integrate dal progettista nonché dell'inferenza di nuove relazioni.

2. SI-Designer: ha il duplice scopo di interfacciarsi fra il sistema e il progettista (fornendo a quest'ultimo un'interfaccia amichevole per l'interazione) e di coordinare l'esecuzione dei diversi software che partecipano all'integrazione.
In particolare:

- interagisce con WordNet per estrarre automaticamente relazioni lessicali tra i termini usati per dare nome a classi ed attributi;
- utilizza Artemis per svolgere il calcolo delle affinità;
- assiste il progettista nelle ultime fasi della costruzione dello schema globale, generando in linguaggio ODL$_{I3}$ il codice che lo descrive.

3. Parser ODL$_{I3}$: effettua l'analisi degli schemi sorgenti, verificandone la consistenza. Inoltre, analizza e archivia tutte le informazioni sullo schema integrato.

Il sistema MOMIS ed il protocollo SOAP

Come già detto, per la gestione dei dati, MOMIS utilizza l'architettura ad oggetti distribuiti CORBA. All'interno di CORBA è possibile sfruttare il protocollo IIOP (basato su TCP/IP) per la trasmissione dei dati, e quindi per l'interazione remota con il sistema.

Il protocollo IIOP presenta tre notevoli svantaggi:

1. È un protocollo nativo per l'architettura CORBA, quindi può essere utilizzato solo all'interno di un sistema in cui tutti i nodi oggetto della comunicazione sono basati sulla stessa architettura CORBA. Questo è un grosso limite al concetto d'interoperabilità.
2. È un protocollo binario e non testuale come SOAP (che infatti è composto da un payload XML, quindi semplice testo). Questo comporta dei problemi nell'attraversamento dei firewall, i quali trovano difficile analizzare gli scopi/le richieste del messaggio e quindi sono propensi a rifiutarlo. Mentre adottando SOAP, in congiunta con HTTP, il firewall si trova ad analizzare un messaggio direttamente leggibile (composto da header HTTP e dal payload informativo XML del messaggio SOAP); questo gli consente di raccogliere rapidamente i contenuti rilevanti delle richieste in entrata e di occuparsi del monitoraggio del traffico in modo semplice.
3. Essendo IIOP un protocollo nativo per l'architettura CORBA richiede l'utilizzo di porte TCP/IP non standard. Anche questa caratteristica comporta delle difficoltà nell'attraversare i firewall che, tendenzialmente, sarà configurato per non permettere comunicazioni su porte TCP/IP non standard. Mentre adottando SOAP, in congiunta con HTTP, è possibile comunicare su porte TCP/IP standard (tipicamente la porta 80), quindi non si ha la necessità di cambiare la configurazione dei firewall con la conseguente diminuzione del livello di sicurezza.

Per questi motivi nasce la necessità di integrare il sistema MOMIS con il protocollo SOAP. Infatti utilizzando SOAP (anziché IIOP), per interagire da remoto con il sistema MOMIS, è possibile superare le difficoltà viste nell'attraversare i firewall ed aprire la strada ad una maggiore interoperabilità con sistemi operativi/linguaggi di programmazione diversi.
L'idea è quella di poter utilizzare tutti i servizi MOMIS, attualmente disponibili, attraverso il protocollo SOAP.
La presente tesi, partendo da questa idea, sviluppa un primo servizio MOMIS utilizzabile in remoto attraverso SOAP (il quale si appoggia al protocollo di trasporto HTTP ed alla porta standard 80).
Nel fare ciò, il client è stato sviluppato attraverso due linguaggi di programmazione diversi: Apache Axis e Visual Basic (estendibile anche ad ASP, per accedere direttamente al servizio da un browser Internet). In questo modo si è messo immediatamente in evidenza il concetto d'interoperabilità, e la possibilità di utilizzare i servizi MOMIS anche attraverso il Visual Basic (o più in generale da un architettura Microsoft proprietaria); questa possibilità era impedita dal protocollo IIOP.
Il server è stato sviluppato con Apache Axis, infatti essendo Axis composto da package Java, si ha la possibilità di mantenere la piena compatibilità con CORBA quindi con il sistema MOMIS già presente. Infatti, non è stato necessario eseguire nessun cambiamento interno al sistema MOMIS, il quale continua ad utilizzare CORBA per accedere alle informazioni. Il server sviluppato, reperisce le informazioni necessarie attraverso CORBA, poi le spedisce al client sfruttando il protocollo SOAP.
Quindi senza la necessità di rivedere un sistema già sviluppato, è stato possibile estendere le sue potenzialità verso una maggiore interoperabilità.

Il servizio ClassiLocali (client Apache Axis)

Nel sviluppare questo servizio si è partiti da un oggetto CORBA Global Schema precedentemente definito e memorizzato all'interno di un file (di tipo mms).
Per accede alle informazioni di questo oggetto si è sfruttato lo stesso procedimento di caricamento presente nella classe SIDesigner.

Il client fornisce il nome del file, dove si trova un oggetto CORBA Global Schema, ed il nome di una classe globale contenuta al suo interno.
Il server recupera questo oggetto CORBA, da questo estrae la sua `MappingTable` e per la classe globale specificata ricava il nome di tutte le classi locali contenute. Infine, invia questi nomi (si tratta di un array di stringhe) al client.

Server ClassiLocali

Come anticipato, il server è stato sviluppato attraverso Apache Axis e pubblicato con il web application server Tomcat.
Il codice del server è il seguente:

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;
import java.io.*;
import java.util.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
//
// --- MOMIS packages
import odli3.*;
import tools.*;
import MomisApplic.*;
import shared.*;
//
// - SIDesigner modules
import SAM.*;
import SIM.*;
import SLIM.*;
import thesRelationEditor.*;
import EXTM.*;
import ARTEMIS.*;
import TUNIM2.*;
import joinMap.*;
import testPackage.*;
import todtd.*;
//
import globalschema.*;
import GlobalSchemaProxy;

public class server_ClassiLocali
{
 private java.util.Vector phases = new Vector();
 //
 // fase attualmente attiva
 private SIDPhase currentPhase = null;
 //
 // ultima fase terminata
 private int lastExecPhase = -1;
 //
 // Informations on SI-Designer configuration.
 private Map sidConfig;
```

```
 //
 // proxy per CORBA object GlobalSchema
 private GlobalSchemaProxy gsProxy;
 //
 // last directory selected in file chooser
 private String lastStatusDir = "";
 //
 // last filename used to save status
 private String lastStatusFileName = "";
 // Stores a reference to the DTD translator dialog
 private DTDProducer dtdDialog;

 //a partire da un GlobalSchemaProxy (memorizzato nel
   file_name),
 //per una sua GlobalClass (specificata da globalClassName)
 //elabora la sua MappingTable e ritorna
 //tutti i nomi  delle classi locali
 public String[] getClassiLocali(String
config_file_name,String opzione,String file_name,String
globalClassName)
 {
  Vector return_vector;
  String[] Args;
  Args= new String[3];
  Args[0]=new String(config_file_name);
  if (opzione != "") {if (file_name != "") {Args[2]=new
String(file_name); Args[1]=new String(opzione);}}
  System.out.println("server_ClassiLocali in azione");

  Map configuration = Tools.readConfFile(Args[0]);

  if (Args.length == 3 && Args[1].equals("-l") )
    {
     // Reads data from a serialized "mms" file
     String fileName=Args[2];
     String fg_serializationType="mms";
     sidConfig = configuration;
     File ftemp = new File(fileName);
     lastStatusFileName = ftemp.getName();
     lastStatusDir = ftemp.getPath();
     // Loads an MMS status
     String status = new String(readFile(fileName));
     if (status != null)
      {
       gsProxy = new GlobalSchemaProxy(sidConfig);
       gsProxy.setStatus(status);
       //String nomegs=gsProxy.getName();
       Schema schema=gsProxy.getLocalSchemata();
       GlobalClass globalClass=(GlobalClass)
       schema.getGlobalClass(globalClassName);
       MappingTable mt;
       return_vector = new Vector();
       mt= (MappingTable) globalClass.getMappingTable();
       try {
             return_vector = mt.getClasses();
```

```
            }
        catch (Exception exc)
            {
              System.out.println(exc.getMessage());
              exc.printStackTrace();
            }
        String[] str=new String[return_vector.size()];
        for(int i=0;i<return_vector.size(); i++)
          str[i]=(String) return_vector.elementAt(i) ;
        return str;

        }
      else
        {
         System.out.println("ERROR: impossible reading file
"+fileName+".");
         String[] str=new String[1];
         str[0]=new String("ERRORE:impossibile leggere il
file");
         return str;
        }

    } else
        {
         String[] str=new String[1];
         str[0]= new String("ERRORE:numero dei parametri
errato");
         return str;
        }

 }

   /**
    * Reads a file.
    *
    * @param fileName File to be read.
    * @return A byte array with the contents of the file or
null if something
    *         wrong.
    */
   private byte[] readFile(String fileName) {
      long fl;      // lunghezza del file da leggere
       try {
          // creo un buffer con le stesse dimensioni del
file
          File f = new File(fileName);
          fl = f.length();
          byte[] buf = new byte[(int)fl];
          // e poi leggo il file nel buffer
          FileInputStream fi = new
FileInputStream(fileName);
          fi.read(buf);
          return buf;
      } catch (IOException e) {
```

```
            System.out.println("ERROR: impossible to read file
"+fileName+".");
            return null;
         };
   }


   /**
    * Return an empty GlobalSchema object.
    */
   private MomisApplic.GlobalSchema getNewGlobalSchema() {
      ORB orb;
      org.omg.CORBA.Object objRef;
      NamingContext ncRef;
      NameComponent nc;
      MomisApplic.GlobalSchema rv = null;
      String gsName = "";
      try {
         //
         // nome del server su cui e' registrato un
MomisFactory
System.out.println((String)sidConfig.get("mfNamingServer"));
         String orbServerName =
((String)sidConfig.get("mfNamingServer")).trim();
         //
         // numero della porta per accedere al server
         String orbPort =
((String)sidConfig.get("mfNamingPort")).trim();
         //
         // nome del momis factory
         String momisFactoryName =
((String)sidConfig.get("mfNamingName")).trim();
         String connArray[] = new String[4];
         connArray[0] = "-ORBInitialHost";
         connArray[1] = orbServerName;
         connArray[2] = "-ORBInitialPort";
         connArray[3] = orbPort;
         System.out.println("- creating orb (" +
connArray[1] + ", " + connArray[3] + ")");
         orb = ORB.init(connArray, null);
         //
         // get the root naming context
         System.out.println("- get root naming context");
         objRef =
orb.resolve_initial_references("NameService");
         ncRef = NamingContextHelper.narrow(objRef);
         //
         // resolve the object reference in naming
         System.out.println("- resolving for momis
factory");
         nc = new NameComponent(momisFactoryName, "");
         NameComponent path[] = {nc};
         MomisApplic.MomisFactory factoryObj =
MomisFactoryHelper.narrow(ncRef.resolve(path));
```

```
            System.out.println("- get new GlobalSchema");
            rv = factoryObj.newGlobalSchema();
         } catch (Exception e) {
        e.printStackTrace();
            if (rv != null) {
               System.out.println("ERRORS: killing new
GlobalSchema object...");
               rv.killObject();
            }
            rv = null;
         }
       return rv;
      }
}
```

In pratica il codice del server per accedere all'oggetto CORBA Global Schema coincide con il codice della classe SIDesigner ripulito di tutte le funzioni GUI (Graphic User Interface) che gestiscono la visualizzazione grafica. Infatti, nel nostro servizio, non è necessario visualizzare sul lato server le informazioni, ma inviarle attraverso SOAP al client.
A questo codice, sono state aggiunte le istruzioni necessarie per recuperare il nome delle classi locali ricercate. Per fare ciò avremo a che fare con i seguenti tipi di oggetti: `Schema`, `GlobalClass` e `MappingTable`.

Analizziamo meglio il codice:

- `public String[] getClassiLocali(String config_file_name,String opzione,String file_name,String globalClassName)`
  La funzione ritorna un array di stringhe contenenti il nome delle classi locali.
  La funzione prevede quattro parametri:
  - `config_file_name`
    Il nome del file di configurazione utilizzato.
  - `opzione`
  - Specifica se il file (`file_name`) da caricare è di tipo mms (`-l`) o di tipo XML (`-x`). Nel servizio sviluppato, è disponibile solo la prima opzione.
  - `file_name`
    Il nome del file in cui è memorizzato l'oggetto CORBA Global Schema, da cui vogliamo ricavare il nome delle classi locali.
  - `globalClassName`
    Il nome della classe globale.

- `Map configuration = Tools.readConfFile(Args[0]);`

  Legge il contenuto del file di configurazione.

- `String status = new String(readFile(fileName));`
  Legge il contenuto del file mms.

- `gsProxy = new GlobalSchemaProxy(sidConfig);`
  Ritorna un nuovo `GlobalSchemaProxy` tenendo in considerazione i parametri di configurazione letti dal file di configurazione (infatti `sidConfig`

= configuration). Il GlobalSchemaProxy rappresenta un proxy per un oggetto CORBA GlobalSchema. Tramite i metodi del GlobalSchemaProxy è possibile accedere o modificare le informazioni contenute nel oggetto CORBA.

- `gsProxy.setStatus(status);`
  Riempie il GlobalSchemaProxy con le informazioni lette dal file mms.

- `Schema schema=gsProxy.getLocalSchemata();`
  Ritorna un oggetto di tipo Schema in cui sono contenute le strutture informative ed il Common Thesaurus dell'oggetto CORBA. Ogni istanza di Schema descrive uno schema ODL_{I3}.

- `GlobalClass globalClass=(GlobalClass) schema.getGlobalClass(globalClassName);`
  Dallo schema ritorna un oggetto GlobalClass individuato dal proprio nome(globalClassName), oppure un null se questo oggetto non esiste.
  Un oggetto di tipo GlobalClass rappresenta la descrizione della singola classe globale. Quindi, gli oggetti di questa classe contengono, un riferimento alla MappingTable, ExtensionalHierarchy e BaseExtension associate alla classe globale.

- `MappingTable mt;`
  Rappresenta un'istanza dell'oggetto di tipo MappingTable.
  La MappingTable, come suggerisce il nome stesso, è una struttura tabellare che deve svolgere due funzioni:
  1) descrizione della classe globale.
  Deve cioè riportare il nome e tipo di tutti gli attributi presenti nella classe globale.
  2) descrizione delle regole di mapping.
  Deve cioè contenere, per ogni classe locale, le regole necessarie per trasformare una query globale in un insieme di query locali.

- ```
  mt= (MappingTable) globalClass.getMappingTable();
   try {
         return_vector = mt.getClasses();
        }
   catch (Exception exc)
       {
         System.out.println(exc.getMessage());
         exc.printStackTrace();
       }
  ```
  Ricava la MappingTable della classe globale e, da questa, tenta di ricavare il vettore di stringhe che specifica il nome delle classi locali (mt.getClasses()).

- ```
  String[] str=new String[return_vector.size()];
  for(int i=0;i<return_vector.size(); i++)
     str[i]=(String) return_vector.elementAt(i) ;
  return str;
  ```
  Ritorna, in un array di stringhe, il nome di tutte le classi locali.

Come si può notare, ancora una volta, non c'è nulla di speciale in questo file Java che lo renda in qualche modo assimilabile ad un servizio web utilizzabile in remoto attraverso SOAP. Le operazioni eseguite, per ottenere le informazioni desiderate, sono esattamente le stesse che dovremmo eseguire dal sistema MOMIS standard.
Le caratteristiche per il supporto del protocollo SOAP sono aggiunte al momento della pubblicazione del servizio attraverso le librerie di Apache Axis.
Questo ci permette di dire che l'estensione SOAP al sistema MOMIS non è un processo di riprogettazione ma un semplice processo d'integrazione.

Per la pubblicazione del servizio è stato utilizzato il seguente documento `deploy_ClassiLocali.wssd`:

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

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

 <service name="ClassiLocali_servizio" provider="java:RPC">
  <parameter name="className" value="server_ClassiLocali"/>
  <parameter name="methodName" value="getClassiLocali"/>
 </service>

</deployment>
```

Il contenuto di questo documento è già stato analizzato nel Capitolo 9.

La pubblicazione avviene fisicamente attraverso la solita istruzione:

```
java
org.apache.axis.client.AdminClient deploy_ClassiLocali.wssd
```

Client ClassiLocali

Il codice del client sviluppato con Apache Axis è il seguente:

```
import org.apache.axis.AxisFault;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.utils.Options;
import org.apache.axis.Constants;
import javax.xml.rpc.ParameterMode;
import javax.xml.rpc.namespace.QName;
import java.util.Vector;
import java.util.*;



public class client_ClassiLocali
{
    public static void main(String [] args) throws Exception
    {
        String endpointURL =
"http://dbgroup.unimo.it:8080/axis/services/ClassiLocali_ser
vizio";
      //controllo sui parametri
      if ( ((args.length != 1) && (args.length != 4)) ||
           ((args.length == 4) && (!args[1].equals("-l")
```

```
          && !args[1].equals("-x")) ) )
        {
         System.out.println("Usage:"+"\n");
         System.out.println("   java SI_Designer
config_file_name [-l file_name]"+" name_GlobalClass \n");
         System.out.println("where:");
         System.out.println("   config_file_name      - name
of the configuration file"+"\n");
         System.out.println(" options:");
         System.out.println("   -l file_name          - start
SIDesigner on the global schema loading its status from
file_name"+"\n");
          System.out.println("   -x file_nameXML      - start
SIDesigner on the global schema loading its status from the
XML file_name"+"\n");
          System.exit(-1);
         }

         Service  service = new Service();
         Call     call    = (Call) service.createCall();
         System.out.println("client_ClassiLocali:preparazione
richiesta all'indirizzo '"+endpointURL+"'");
         String result;
         try {
            call.setTargetEndpointAddress( new
java.net.URL(endpointURL) ); //indirizzo del servizio

            call.setOperationName(new QName
("ClassiLocali_servizio","getClassiLocali"));
            //nome del metodo richiesto al server

            call.addParameter(
"arg0",org.apache.axis.encoding.XMLType.XSD_STRING,Parameter
Mode.IN ); //nome del file di configurazione
            call.addParameter(
"arg1",org.apache.axis.encoding.XMLType.XSD_STRING,Parameter
Mode.IN ); //opzione
            call.addParameter(
"arg2",org.apache.axis.encoding.XMLType.XSD_STRING,Parameter
Mode.IN ); //file opzione
            call.addParameter(
"arg3",org.apache.axis.encoding.XMLType.XSD_STRING,Parameter
Mode.IN ); //nome della classe GlobalClass

            call.setReturnType(Constants.SOAP_ARRAY);
      String[] str;

            //richiesta al server
            str=(String[]) call.invoke( new Object[] {
args[0],args[1],args[2],args[3] } );

            System.out.println("client_ClassiLocali in
azione, risposta ricevuta dal server:");

              for(int i=0;i<str.length; i++)
                     System.out.println(str[i]);
```

```
             System.out.println("Tutto OK");

            } catch (AxisFault fault)
                {
                 result = "Error : " + fault.toString();
                 System.out.println(result);
                }
        }

}
```

Come si può notare il client non contiene nessuna novità rispetto a quanto già visto negli esempi del Capitolo 9. Allo stesso Capitolo 9 possiamo fare riferimento per la l'analisi dettagliata del codice.
I quattro parametri richiesti dal client sono gli stessi che saranno inviati al server per svolgere il servizio (nome del file di configurazione, opzione, nome del file mms e nome della classe globale).

Messaggi SOAP di ClassiLocali

Una volta pubblicato il servizio è possibile eseguire una richiesta utilizzando, ad esempio, la seguente istruzione:

```
java client_ClassiLocali siDesigner -l
 /export/home/sandrobe/prova_class4.mms Global4
```

Il messaggio SOAP di richiesta inviato dal client è il seguente:

```
POST /axis/services/ClassiLocali_servizio HTTP/1.0

Content-Length: 699

Host: dbgroup.unimo.it
Content-Type: text/xml; charset=utf-8

SOAPAction: ""



<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
 SOAP-ENV:encodingStyle=
                "http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
 <SOAP-ENV:Body>
  <ns1:getClassiLocali xmlns:ns1="ClassiLocali_servizio">
   <arg0 xsi:type="xsd:string">siDesigner.conf</arg0>
   <arg1 xsi:type="xsd:string">-l</arg1>
   <arg2 xsi:type="xsd:string">
      /export/home/sandrobe/prova_class4.mms
   </arg2>
   <arg3 xsi:type="xsd:string">Global4</arg3>
  </ns1:getClassiLocali>
 </SOAP-ENV:Body>
```

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

Il messaggio SOAP di risposta fornito dal server è il seguente:

```
HTTP/1.1 200 OK

Content-Type: text/xml; charset=utf-8

Content-Length: 1131

Date: Thu, 04 Jul 2002 14:03:29 GMT

Server: Apache Tomcat/4.0.1 (HTTP/1.1 Connector)



<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <SOAP-ENV:Body>
  <ns1:getClassiLocaliResponse
   SOAP-ENV:encodingStyle=
                "http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:ns1="ClassiLocali_servizio">
   <getClassiLocaliReturn xsi:type="SOAP-ENC:Array"
    SOAP-ENC:arrayType="xsd:string[12]"
    xmlns:SOAP-ENC=
              "http://schemas.xmlsoap.org/soap/encoding/">
    <item>eclass_xml.officesupplies</item>
    <item>eclass_xml.officesuppliesother</item>
    <item>eclass_xml.pad</item>
    <item>eclass_xml.paper</item>
    <item>eclass_xml.paperfilm</item>
    <item>ucec_xml.industrialusepaper</item>
    <item>
          ucec_xml.officeequipmentandaccessoriesandsupplies
    </item>
    <item>ucec_xml.officesupplies</item>
    <item>ucec_xml.paper</item>
    <item>ucec_xml.paperproducts</item>
    <item>ucec_xml.personalpaperproduct</item>
    <item>ucec_xml.printingandwritingpaper</item>
   </getClassiLocaliReturn>
  </ns1:getClassiLocaliResponse>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Come si vedere nella classe globale `Global4` del file `prova_class4.mms` sono contenute 12 classi locali, il cui nome è serializzato come un array di 12 stringhe.

Il servizio ClassiLocali (client Visual Basic/ASP)

A questo punto, per verificare l'interoperabilità del protocollo SOAP, sviluppiamo un client Visual Basic (con il supporto di Microsoft SOAP Toolkit) per accedere al servizio ClassiLocali che presenta lo stesso server (sviluppato con Apache Axis) visto nel precedente paragrafo.
Per fare ciò si è deciso di sfruttare anche i vantaggi del linguaggio WSDL.

Il supporto WSDL di Apache Axis per il servizio ClassiLocali

Attraverso il tool Java2WSDL di Apache Axis (presentato nel Capitolo 9) è possibile ottenere il documento WSDL associato al servizio ClassiLocali.

Digitando l'istruzione

```
java org.apache.axis.wsdl.Java2WSDL -o ClassiLocali.wsdl -l
"http://dbgroup.unimo.it/axis/services/ClassiLocali_servizio
" -m getClassoLocali[10] server_ClassiLocali
```

otteniamo il seguente documento `ClassiLocali.wsdl`:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://DefaultNamespace"
xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:impl="http://DefaultNamespace-impl"
xmlns:intf="http://DefaultNamespace"
xmlns:tns2="http://MomisApplic"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <types>
  <schema targetNamespace="http://DefaultNamespace"
xmlns="http://www.w3.org/2001/XMLSchema">
   <complexType name="ArrayOf_SOAP-ENC_string">
    <complexContent>
     <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType"
wsdl:arrayType="xsd:string[]"/>
     </restriction>
    </complexContent>
   </complexType>
   <element name="ArrayOf_SOAP-ENC_string" nillable="true"
type="intf:ArrayOf_SOAP-ENC_string"/>
  </schema>
 </types>


   <wsdl:message name="getClassiLocaliResponse">

      <wsdl:part name="return" type="intf:ArrayOf_SOAP-
ENC_string"/>
```

[10] Con l'opzione `-m getClassiLocali`, nella creazione del documento WSDL, viene presa in considerazione solo l'operazione specificata. Questo per evitare che operazioni interne per il server siano considerate come operazioni utilizzabili in remoto via SOAP.

```
   </wsdl:message>

   <wsdl:message name="getClassiLocaliRequest">

      <wsdl:part name="in0" type="SOAP-ENC:string"/>

      <wsdl:part name="in1" type="SOAP-ENC:string"/>

      <wsdl:part name="in2" type="SOAP-ENC:string"/>

      <wsdl:part name="in3" type="SOAP-ENC:string"/>

   </wsdl:message>


   <wsdl:portType name="server_ClassiLocali">

      <wsdl:operation name="getClassiLocali"
parameterOrder="in0 in1 in2 in3">

         <wsdl:input message="intf:getClassiLocaliRequest"/>

         <wsdl:output
message="intf:getClassiLocaliResponse"/>

      </wsdl:operation>

   </wsdl:portType>


   <wsdl:binding name="ClassiLocali_servizioSoapBinding"
type="intf:server_ClassiLocali">

      <wsdlsoap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>

      <wsdl:operation name="getClassiLocali">

         <wsdlsoap:operation soapAction=""/>

         <wsdl:input>

            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="getClassiLocali" use="encoded"/>

         </wsdl:input>

         <wsdl:output>

            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://DefaultNamespace" use="encoded"/>

         </wsdl:output>
```

```
      </wsdl:operation>

   </wsdl:binding>


   <wsdl:service name="server_ClassiLocaliService">

      <wsdl:port
binding="intf:ClassiLocali_servizioSoapBinding"
name="ClassiLocali_servizio">


         <wsdlsoap:address
location="http://dbgroup.unimo.it/axis/services/ClassiLocali
_servizio"/>

      </wsdl:port>

   </wsdl:service>

</wsdl:definitions>
```

Tenendo presente le considerazioni fatte per il linguaggio WSDL nel Capitolo 6, da questo documento è possibile ricavare in modo univoco e standard le caratteristiche del servizio. Ad esempio:

- Il metodo richiesto al server si chiama `getClassiLocali`.
- Il metodo richiede quattro parametri d'ingresso di tipo stringa (`in0`, `in1`, `in2` e `in3`).
- Il metodo risponde con un array di stringhe (`return`).
- Il protocollo di trasporto utilizzato è HTTP.
- Il servizio è localizzato all'indirizzo `http://dbgroup.unimo.it/axis/services/ClassiLocali_servizio`.
- Ecc.

Un client che possiede il supporto per WSDL ha la possibilità di accedere ed analizzare questo documento. Così facendo, il client può automaticamente ricavare tutti i parametri necessari per la configurazione della comunicazione SOAP. Questo rappresenta un aiuto per lo sviluppatore che riduce al minimo la programmazione e la presenza di errori.

Client Visual Basic/ASP per ClassiLocali

Per memorizzare il nome delle classi locali è stata creato, in Visual Basic, il seguente oggetto `ClassiLocali`:

```
Option Explicit

Private Array_stringhe() As String

Public Property Get get_array_stringhe() As String()
   get_array_stringhe = Array_stringhe
End Property


Public Property Let let_array_stringhe(ByRef value() As
String)
```

```
   ReDim Array_stringhe(UBound(value))
   Dim i As Integer
   For i = LBound(value) To UBound(value)
     Array_stringhe(i) = value(i)
   Next i
End Property

Public Function get_lenght() As Integer
 get_lenght = LBound(Array_stringhe) + 1
End Function
```

Per poter deserializzare il contenuto dell'array di stringhe ricevuto dal server, è stato necessario definire il seguente deserializzatore nella classe `ClassiLocali_mapper`:

```
Option Explicit

Implements ISoapTypeMapper

Private marray_stringheMapper As ISoapTypeMapper

'inizializzazione
Private Sub ISoapTypeMapper_Init( _
    ByVal pFactory As MSSOAPLib.ISoapTypeMapperFactory, _
    ByVal pSchema As MSXML2.IXMLDOMNode, _
    ByVal xsdType As MSSOAPLib.enXSDType)

    Dim Doc3 As DOMDocument30
    Dim SelectionNamespaces As String
    Dim SelectionLanguage As String
    Dim Node As IXMLDOMNode

    ' Get document that contains schema for complex type.
    Set Doc3 = pSchema.ownerDocument

    ' Save current SelectionLanguage property value, then
change to allow selection using XPath.
    SelectionLanguage= Doc3.getProperty("SelectionLanguage")
    Doc3.setProperty "SelectionLanguage", "XPath"

    ' Save the current SelectionNamepsaces property value,
then change
    ' to allow elements in the XSD namespace to be selected.
    SelectionNamespaces                                    =
Doc3.getProperty("SelectionNamespaces")
    Doc3.setProperty "SelectionNamespaces", _
         "xmlns:XSD='http://www.w3.org/2001/XMLSchema'"

    ' Restore document properties even if an error occurs.
    On Error GoTo Cleanup

    ' Select the element node for the name element, then get
a type mapper
    ' for that element.
```

```
    Set                          Node                          =
pSchema.selectSingleNode("XSD:sequence/XSD:element[@name='ar
ray_stringhe']")
    Set               marray_stringheMapper                =
pFactory.getElementMapper(Node)

Cleanup:
    '  Restore  SelectionNamespaces  and  SelectionLanguage
properties.
    Doc3.setProperty                  "SelectionNamespaces",
SelectionNamespaces
    Doc3.setProperty "SelectionLanguage", SelectionLanguage

End Sub

'deserializzazione
Private   Function   ISoapTypeMapper_read(ByVal   pNode   As
MSXML2.IXMLDOMNode, _
ByVal bstrEncoding As String, ByVal encodingMode As _
MSSOAPLib.enEncodingStyle, ByVal lFlags As Long) As Variant

    Dim Node As IXMLDOMNode

    Dim Array_stringhe() As String
    'Read array_string element's content.
    Array_stringhe = marray_stringheMapper.read( _
            pNode.selectSingleNode("getClassiLocaliReturn"),
_
            bstrEncoding, encodingMode, lFlags)

    Dim cl As New ClassiLocali
    cl.let_array_stringhe = Array_stringhe

    Set ISoapTypeMapper_read = cl
End Function

Private Function ISoapTypeMapper_varType() As Long
    ISoapTypeMapper_varType = vbObject
End Function

'serializzazione
Private Sub ISoapTypeMapper_write(ByVal pSoapSerializer As _
MSSOAPLib.ISoapSerializer,  ByVal  bstrEncoding  As  String,
ByVal _
encodingMode  As  MSSOAPLib.enEncodingStyle,  ByVal  lFlags  As
Long, _
pvar As Variant)
  'non fa nulla, non ci serve il serializzatore
End Sub
```

All'interno di questo codice, si utilizzano i metodi messi a disposizione dalla libreria Microsoft SOAP Type per accedere allo schema creato dal parser DOM nel momento in cui arriva il messaggio di risposta SOAP.
In pratica, il deserializzatore va a catturare, all'interno dello schema, il contenuto del sottonodo `getClassiLocaliReturn`, cioè il risultato dell'elaborazione (l'array di

stringhe). Questi valori vengono assegnati ad `Array_stringhe` della classe `ClassiLocali`.
Per capire in dettaglio il deserializzatore si può far riferimento all'esempio `Indirizzo` del Capitolo 8.

Client Visual Basic di ClassiLocali

Il codice del client richiama le precedenti classi (`ClassiLocali` e `ClassiLocali_mapper`) le quali sono raggruppate nel progetto `ClassiLocaliMapper`.
Il client `client_ClassiLocali` ha è il seguente:

```
Option Explicit
Private soapclient As soapclient

Public Function go(ByVal config_file_name As String, ByVal
opzione As String, ByVal file_name As String, ByVal
globalClass As String) As String()

On Error GoTo fail
Set soapclient = New soapclient soapclient.mssoapinit _
"http://dbgroup.unimo.it/Momis/prototipo/SOAP/ClassiLocali.wsdl", _
"", "", "C:\SOAP_Momis\ClassiLocali\ClassiLocali.wsml"

Dim cl As ClassiLocali

Set cl = soapclient.getComplex(config_file_name, opzione,
file_name, globalClass)

Dim arr_str() As String
ReDim arr_str(cl.get_lenght)
arr_str = cl.get_array_stringhe
go = arr_str

Exit Function

fail:
MsgBox soapclient.faultcode
MsgBox soapclient.faultstring
MsgBox soapclient.faultactor
MsgBox soapclient.detail

End Function
```

Come si può vedere la funzione `go` richiede quattro parametri che saranno inviati al server: il nome del file di configurazione, l'opzione, il nome del file mms ed il nome della classe globale. Dopodiché (come già spiegato nel Capitolo 8) l'oggetto `soapclient` viene inizializzato (`soapclient.mssoapinit`) con il contenuto dei documenti WSDL e WSML specificati.
Il documento WSDL indicato è quello visto nel precedente paragrafo e creato con il tool Java2WSDL di Apache Axis. Il client accede a questo documento (tramite HTTP, essendo memorizzato sulla stessa macchina del server) e si ricava automaticamente tutte le specifiche per accedere al servizi ClassiLocali.

Mentre il documento WSML, in questo caso, contiene il riferimento alla classe che definisce il deserializzatore.
Il documento WSML è stato creato manualmente tenendo presente il contenuto del documento WSDL e del progetto `ClassiLocaliMapper`.
Il contenuto del documento `ClassiLocali.wsdl` è il seguente:

```
<?xml version='1.0' encoding='UTF-8' ?>

<servicemapping name='ClassiLocali_servizio'>
  <service name='server_ClassiLocaliService'>
    <using PROGID='ClassiLocaliMapper.ClassiLocali_mapper'
cachable='0' ID='CLmap' />
    <types>
     <type name='ArrayOf_SOAP-ENC_string'
           targetNamespace='http://DefaultNamespace'
           uses='CLmap' />
    </types>
  </service>
</servicemapping>
```

In poche parole, per il dato di tipo `ArrayOf_SOAP-ENC_string` (con namespace `http://DefaultNamespace`) definito per il service `server_ClassiLocaliService` (si veda il relativo documento WSDL) si deve utilizzare il deserializzatore `ClassiLocali_mapper`.

A questo punto invocando la funzione `go` del codice client (direttamente dall'ambiente Visual Basic o da una pagina ASP, ad esempio) è possibile accedere al servizio gestito dal server Apache Axis visto in precedenza.

Client ASP di ClassiLocali

In questo caso si è deciso di richiamare la funzione `go` del client da una pagina ASP, così da poter accedere al sevizio e successivamente visualizzare il risultato direttamente da un browser Internet (ad esempio Microsoft Explorer).
Il codice della pagina ASP è il seguente:

```
<%@ LANGUAGE = VBScript %>

<HTML>
<HEAD>
<TITLE>ClassiLocali_servizio</TITLE>
</HEAD>
<BODY>

Set objSoap
  = Server.CreateObject("client_ClassiLocali.class")

Response.Write objSoap.go("siDesigner.conf","-
l","/export/home/sandrobe/prova_cass4.mms","Global4")
%>

</BODY>
</HTML>
```

Come si può vedere il codice VBScript della pagina ASP è molto semplice:

- `Set objSoap = Server.CreateObject("client_ClassiLocali.class")`
  Viene istanziato un oggetto della classe contenente il codice del client Visual Basic.

- `Response.Write objSoap.go("siDesigner.conf","-l","/export/home/sandrobe/prova_cass4.mms","Global4")`
  Viene mandata in esecuzione la funzione `go` del client Visual Basic, la quale invia i quattro parametri necessari al server, riceve la risposta e visualizza quest'ultima all'interno della pagina web. In realtà i parametri di chiamata del servizio sono costanti, ma non è difficile richiedere questi all'utente direttamente dalla pagina web.

I messaggi SOAP per il servizio ClassiLocali con client Visual Basic/ASP sono gli stessi già descritti per il client Apache Axis.

Conclusioni

Il servizio ClassiLocali sviluppato è solo un primo semplice servizio per le possibilità offerte da MOMIS in congiunta con SOAP. Ma analizzando questo esempio, appaiono già chiare le potenzialità offerte dal protocollo SOAP:

- Possibilità di integrare l'ambiente MOMIS con il protocollo SOAP senza la necessità di rivedere/riprogettare le caratteristiche base di MOMIS.
- Possibilità di aprire l'utilizzo di MOMIS a diversi linguaggi di programmazione su differenti architetture, ossia rendere possibile l'interoperabilità.
- Possibilità di utilizzare in remoto i servizi SOAP di MOMIS senza incontrare difficoltà nell'attraversare i firewall. Infatti i messaggi SOAP (essendo testuali, e non binari) facilitano notevolmente il compito del firewall nell'analizzare gli scopi/ la destinazione del messaggio stesso.
- Possibilità di utilizzare in remoto i servizi SOAP di MOMIS mantenendo aperta sul firewall la sola porta 80 (una porta sempre aperta), il ché comporta notevoli benefici in termini di sicurezza.

Infine non va sottovalutato il ruolo del linguaggio WSDL, che rappresenta un prezioso alleato per il protocollo SOAP e per lo sviluppatore di applicazioni SOAP.

Conclusioni generali

Nel corso di questa tesi si è potuto acquisire la conoscenza del protocollo SOAP e delle sue potenzialità, andandole a verificare nell'utilizzo pratico. Nel fare ciò sono state prese in considerazioni due diversi tipi d'implementazione: Apache e Microsoft. La prima, essendo basata su Java, si rivolge a diverse architetture; mentre la seconda si rivolge ai linguaggi di programmazione (di conseguenza alle architetture) di casa Microsoft.

Lo sviluppo della tesi è partito dalla definizione di semplici esempi, per acquisire le conoscenze di base, che man mano sono cresciuti di complessità fino a testare diversi gradi d'interoperabilità.
Il passo finale è stato quello di applicare le conoscenze acquisite nel corso della tesi ad un caso reale, già ben definito: il sistema MOMIS. Nella tesi è stato implementato un primo servizio gestito dal sistema MOMIS ed utilizzabile in remoto, attraverso il protocollo SOAP, da diverse tipologie di cliente sostenute da differenti architetture.
Gli sviluppi futuri di questa tesi riguarderanno l'integrazione di tutti i servizi MOMIS attualmente disponibili con il protocollo SOAP. A questo punto il sistema MOMIS potrà essere considerato a tutti gli effetti perfettamente coeso con il concetto d'interoperabilità.

Al termine di questa tesi è possibile esprimere un giudizio, in base all'utilizzo pratico fatto, sui diversi elementi incontrati:

- Il protocollo SOAP.
  Si è potuto apprezzare effettivamente le qualità attese in termini di:
  - **Interoperabilità** fra diverse architetture e/o diversi linguaggi di programmazione. Nel corso della tesi sono state sviluppate diverse combinazioni, una fra tutte la possibilità di accedere ad un servizio del sistema MOMIS attraverso un client Visual Basic (con il supporto Microsoft SOAP Toolkit) installato su un'architettura Windows (2000 oppure XP). Mentre il client, sviluppato con Apache Axis, è installato su un architettura Unix/Solaris.
  - **Facilità nell'attraversamento dei firewall** da parte dei messaggi SOAP. Non sono mai state incontrate difficoltà in questo senso.
  - **Utilizzo di porte standard** per le comunicazioni. Non è mai stato necessario cambiare la configurazione dei firewall incontrati (aprire porte non standard), quindi si è mantenuto un elevato grado di sicurezza per la rete intranet.

- Il linguaggio WSDL.
  Sicuramente è risultato un buon alleato del protocollo SOAP:
  - Permette di descrivere in modo univoco e standard il servizio pubblicato.
  - Permette al client di ricavarsi automaticamente tutte le informazioni necessarie per conoscere le caratteristiche del servizio pubblicato, facilitando anche il compito dello sviluppatore.

- Apache SOAP 2.2.
  Rappresenta una buona implementazione SOAP se pur con due svantaggi:
  - Mancanza del supporto per il linguaggio WSDL.
  - Incontra qualche difficoltà nel comunicare con l'implementazione Microsoft SOAP Toolkit. Queste difficoltà, comunque, sono state risolte all'interno della tesi.

- Microsoft SOAP Toolkit 2.0.
  Il supporto per il linguaggio WSDL è presente ad un livello base. Infatti, nelle situazioni più complesse, è stato necessario intervenire manualmente sul documento WSDL (e WSML) creato in automatico dal tool di supporto.

- Apache Axis.
  Rappresenta l'evoluzione di Apache SOAP 2.2, e come tale elimina tutti i difetti del predecessore. Inoltre fornisce un ottimo supporto al linguaggio WSDL attraverso tool automatici che facilitano il compito dello sviluppatore. La possiamo considerare, sicuramente, la più completa e flessibile implementazione SOAP incontrata in questa tesi.

- Il sistema MOMIS.
  Si è potuto dimostrare la possibilità di integrare il sistema MOMIS con il protocollo SOAP senza l'onere di riorganizzare/riprogettare il sistema. Quindi con un basso costo di sviluppo si ha la possibilità di usufruire di tutti i vantaggi offerti dal protocollo SOAP.

Bibliografia

I. **SOAP Version 1.2 Part 0: Primer**
W3C Working Draft 17 December 2001

II. **SOAP Version 1.2 Part 1: Messaging**
W3C Working Draft 17 December 2001

III. **SOAP Version 1.2 Part 2: Adjuncts**
W3C Working Draft 17 December 2001

IV. **Web Services Description Language (WSDL)**
W3C Note 15 March 2001

V. **Sviluppare Applicazioni con SOAP**
Massimo Canducci (Apogeo)

VI. **www.apache.org**

VII. **www.microsoft.com**

VIII. **www.w3c.org**

IX. **www.xmethods.com**