Università di Verona, Facoltà di Scienze MM.FF.NN.

Insegnamento di Architetture Hardware di Laboratorio

Comunicazioni in rete tramite Java

Davide Quaglia

Scopo di questa esercitazione è quello di imparare a scrivere programmi Java e farli cooperare attraverso una rete. In particolare verranno prese in considerazione due tecnologie:
1) utilizzo diretto del protocollo TCP;
2) meccanismo dei Web Services per creare e usare servizi distribuiti in maniera object-oriented.

1 Il package java.net

Questo package definisce fondamentalmente:
- le classi Socket e ServerSocket per le connessioni TCP
- la classe DatagramSocket per le connessioni UDP
- la classe URL per le connessioni HTTP

più una serie di classi correlate:
- InetAddress per rappresentare gli indirizzi Internet
- URLConnection per rappresentare le connessioni a un URL
- ...

2 I/O tramite socket

Le socket sono associate a un InputStream e a un OutputStream: quindi, per scrivere e leggere si usano le normali primitive previste per gli stream. È anche possibile incapsulare tali stream in stream più sofisticati, come ampiamente discusso a proposito del package di I/O.

ESEMPIO 1 - apertura di un URL specificato

La classe URL è la base per creare connessioni HTTP. Questo esempio si connette all'URL dato e si visualizza ciò che viene inviato dal server (nell'ipotesi che si tratti di testo).

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

class EsempioURL
{
    public static void main(String args[])
    {
        String indirizzo = "file:///autoexec.bat";
        URL u = null;
        try
        {
            u = new URL(indirizzo);
            System.out.println("URL aperto: " + u);
        }
        catch (MalformedURLException e)
        {
            System.out.println("URL errato: " + u);
        }

        URLConnection c = null;
        DataInputStream istream = null;
```

```
        try
        {
            System.out.print("Connessione in corso...");
            c = u.openConnection();
            c.connect();
            System.out.println("ok.");
            BufferedInputStream b = new
            BufferedInputStream(c.getInputStream());
            istream = new DataInputStream(b);
            System.out.println("Lettura dei dati...");
            String s;
            while( (s = istream.readLine()) != null )
                System.out.println(s);
        }
        catch (IOException e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

ESEMPIO 2 - una mini applicazione client/server

Il client si limita a visualizzare tutto quello che gli è arriva dal server, fino a che non riceve il messaggio "Stop". Il server assegna un numero progressivo, a partire da 1, a ogni client che si connette.

```
// IL CLIENT 1
// Ipotesi: server port (fisso) = 11111

import java.io.*;
import java.net.*;

class Client1
{
    public static void main(String args[])
    {
        Socket s = null;
        DataInputStream is = null;

        try
        {
            s = new Socket("localhost", 11111);
            is = new DataInputStream(s.getInputStream());
        }
        catch (IOException e)
        {
            System.out.println(e.getMessage());
            System.exit(1);
        }

        System.out.println("Socket creata: " + s);

        try
        {
            String line;
            while( (line=is.readLine())!=null )
            {
                System.out.println("Ricevuto: " + line);
                if (line.equals("Stop"))
                    break;
            }
```

```
            is.close(); // chiusura stream
            s.close(); // chiusura socket
        }
        catch (IOException e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

```
// IL SERVER 1

import java.io.*;
import java.net.*;

class Server1
{
    public static void main(String args[])
    {
        ServerSocket serverSock = null;
        Socket cs = null;
        int numero = 1;
        System.out.print("Creazione ServerSocket...");

        try
        {
            serverSock = new ServerSocket(11111);
        }
        catch (IOException e)
        {
            System.err.println(e.getMessage());
            System.exit(1);
        }

        while (numero<3) // condizione arbitraria
        {
            System.out.print("Attesa connessione...");

            try { cs = serverSock.accept(); }
            catch (IOException e)
            {
                System.err.println("Connessione fallita");
                System.exit(2);
            }

            System.out.println("Conness. da " + cs);

            try
            {
                BufferedOutputStream b = new
BufferedOutputStream(cs.getOutputStream());
                PrintStream os = new PrintStream(b,false);
                os.println("Nuovo numero: " + numero);
                numero++;
                os.println("Stop"); os.close();
                cs.close();
            }
            catch (Exception e)
```

```
            {
                System.out.println("Errore: " +e);
                System.exit(3);
            }
        }
    }
}
```

ESEMPIO 3 - Una mini-calcolatrice client / server

Il client invia al server una serie di righe di testo, contenenti numeri interi (uno per riga); l'ultima riga contiene lo zero. Il server effettua la somma dei valori ricevuti e la ritrasmette al client; una riga contenente il valore 0 indica fine sequenza.

```
// IL CLIENT 2

import java.io.*;
import java.net.*;

class Client2
{
    public static void main(String args[])
    {
        Socket c = null;
        DataInputStream is = null;
        PrintStream os = null;

        try
        {
            c = new Socket("localhost", 11111);
            is = new DataInputStream(c.getInputStream());
            os = new PrintStream(new BufferedOutputStream(c.getOutputStream()));
        }
        catch (IOException e)
        {
            System.err.println(e.getMessage());
            System.exit(1);
        }

        System.out.println("Socket creata: " + c);

        // invio valori al server (da linea comandi)
        for (int i=0; i<args.length; i++)
        {
            System.out.println("Sending " + args[i]);
            os.println(args[i]);
        }

        os.println("0"); os.flush();
        System.out.println("Attesa risposta...");
        String line = null;

        try
        {
            line = new String(is.readLine());
            is.close();
            os.close();
            s.close();
        }
        catch (IOException e)
        {
            System.err.println(e.getMessage());
```

```
        }

        System.out.println("Msg dal server: " + line);
    }
}



// IL SERVER 2

import java.io.*;
import java.net.*;

class Server2
{
    public static void main(String args[])
    {
        ServerSocket serverSock = null;
        Socket cs = null;
        System.err.println("Creazione ServerSocket");

        try
        {
            serverSock = new ServerSocket(11111);
        }
        catch (IOException e)
        {
            System.err.println(e.getMessage());
            System.exit(1);
        }

        while (true)
        {
            System.out.println("Attesa connessione...");

            try
            {
                cs = serverSock.accept();
            }
            catch (IOException e)
            {
                System.out.println("Connessione fallita");
                System.exit(2);
            }

            System.out.println("Connessione da " + cs);

            try
            {
                BufferedInputStream ib = new
BufferedInputStream(cs.getInputStream());
                DataInputStream is = new DataInputStream(ib);
                BufferedOutputStream ob = new
BufferedOutputStream(cs.getOutputStream());
                PrintStream os = new PrintStream(ob,false);

                String line;
                int y, x = 0;

                do
                {
                    line = new String(is.readLine());
                    y = Integer.parseInt(line);
```

```
                    System.out.println("Value: " + y);
                    x += y;
                }
                while(y!=0);

                os.println("Somma = " + x);
                os.flush();
                os.close();
                is.close();
                cs.close();
            }
            catch (Exception e)
            {
                System.out.println("Errore: " +e);
                System.exit(3);
            }
        }
    }
}
```

ESEMPIO 4 - Un mini-FTP

Il client invia al server il nome di un file (di testo), preso dalla riga di comando. Il server risponde spedendo al client il contenuto, riga per riga, di tale file. Sono gestite le situazioni particolari (file not found, etc.)

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

class Client3
{
    public static void main(String args[])
    {
        Socket s = null;
        DataInputStream is = null;
        PrintStream os = null;

        try
        {
            s = new Socket("localhost", 11111);
            is = new DataInputStream(s.getInputStream());
            os = new PrintStream(new BufferedOutputStream(s.getOutputStream()));
        }
        catch (IOException e)
        {
            System.err.println(e.getMessage());
            System.exit(1);
        }

        System.out.println("Socket creata: " + s);

        // --- controllo argomenti
        if (args.length==0)
        {
            os.println("Missing file name");
            os.flush();
            os.close();

            try
            {
                is.close();
               s.close();
```

```
            }
            catch (IOException e)
            {
                System.out.println(e.getMessage());
            }
            System.exit(1);
        }

        // --- invio messaggio
        System.out.println("Sending " + args[0]);
        os.println(args[0]); os.flush();

        // --- stampa risposta del server
        System.out.println("Attesa risposta...");
        String line = null;

        try
        {
            while ((line = is.readLine()) != null)
            {
                System.out.println("Messaggio: " + line);
            }
            is.close();
            os.close();
            s.close();
        }
        catch (IOException e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

Il server risponde spedendo al client il contenuto, riga per riga, di tale file. Sono gestite le situazioni particolari (file not found, etc.)

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

class Server3
{
    public static void main(String args[])
    {
        ServerSocket serverSock = null;
        Socket c = null;
        System.err.println("Creazione ServerSocket");
        try
        {
            serverSock = new ServerSocket(11111);
        }
        catch (IOException e)
        {
            System.out.println(e.getMessage());
            System.exit(1);
        }

        while (true)
        {
            System.out.println("Attesa connessione...");
            try
            {
                c = serverSock.accept();
```

```
            }
            catch (IOException e)
            {
                System.out.println("Connessione fallita");
                System.exit(2);
            }

            System.out.println("Connessione da " + c);
            // --- inizio colloquio col client
            DataInputStream is = null;
            PrintStream os = null;

            try
            {
                BufferedInputStream ib = new
BufferedInputStream(cs.getInputStream());
                is = new DataInputStream(ib);
                BufferedOutputStream ob = new
                BufferedOutputStream(c.getOutputStream());
                os = new PrintStream(ob,false);
                // --- ricezione nome file dal client
                String n = new String(is.readLine());
                System.out.println("File: " + n);
                // --- controllo esistenza file
                if (n.equals("Missing file name"))
                {
                    os.flush();
                    os.close();
                    is.close();
                    c.close();
                }

                // --- invio del file al client
                DataInputStream is = new DataInputStream(new FileInputStream(n));

                String r = null;
                while ((r = is.readLine())!=null)
                {
                    os.println(r);
                }

                os.flush();
                os.close();
                is.close();
                cs.close();
            }
            catch (FileNotFoundException e)
            {
                System.out.println("File non trovato");
                os.println("File non trovato");
                os.flush();
                os.close();
            }
            catch (Exception e)
            {
                System.out.println(e);
            }
        }
    }
}
```

3 Architetture orientate ai servizi

L'architettura orientata ai servizi (Service Oriented Architecture o SOA) definisce un nuovo modello logico secondo il quale sviluppare il software. Tale modello è realizzato dai *Web Services*, che si presentano come moduli software distribuiti i quali collaborano fornendo determinati servizi in maniera standard. Un Web Service è un componente SW progettato per interagire automaticamente con altri componenti SW attraverso la rete. Esso ha un'interfaccia descritta in un formato comprensibile da un sistema automatico (cioè il Web Services Description Language – WSDL). Altri componenti SW interagiscono con il web service attraverso messaggi conformi allo standard Simple Object Access Protocol (SOAP) trasportati in connessioni HTTP. E' compito del protocollo SOAP definire una forma serializzata dei parametri attuali da passare al web service e dei valori restituiti da quest'ultimo. WSDL e SOAP seguono il formato XML al fine di essere trattabili da strumenti automatici.

Quella basata sui Web Services è una tecnologia che ha oramai preso piede nell'industria informatica, che può essere usata per esporre sul Web, in modo sicuro e trasparente, delle funzioni di elaborazione (calcolo o accesso a base dati). Attraverso i Web Services, è possibile incapsulare la logica applicativa e presentarla all'esterno come un insieme di chiamate a metodi, oppure cercare e usare servizi messi a disposizione da altri, oppure ancora, costruire un'applicazione distribuita su più macchine per aumentarne l'affidabilità. I Web Services sono un valido strumento per promuovere la collaborazione automatica tra sistemi di elaborazione senza l'intervento umano.

La nuova struttura collaborativa distribuita porta ad una serie di vantaggi che riassumiamo brevemente.

- **Software come servizio**: al contrario del software tradizionale, una collezione di metodi esposta tramite Web Service può essere utilizzata come un servizio accessibile da qualsiasi piattaforma. I Web Services consentono l'incapsulamento: i componenti possono essere isolati in modo tale che solo lo strato relativo al servizio vero e proprio sia esposto all'esterno. Ciò comporta due vantaggi fondamentali: indipendenza dall'implementazione e sicurezza del sistema interno.
- **Interoperabilità**: la logica applicativa incapsulata all'interno dei Web Services è completamente decentralizzata ed accessibile attraverso Internet da piattaforme, dispositivi e linguaggi di programmazione differenti.
- **Semplicità di sviluppo e di rilascio**: sviluppare un insieme di Web Services, intorno ad uno strato di software esistente, è un'operazione semplice che non dovrebbe richiedere cambiamenti nel codice originale dell'applicazione. Lo sviluppo incrementale dei WS avviene in modo semplice e naturale. Inoltre, rilasciare un WS significa solo esporlo al Web.
- **Semplicità di installazione**: la comunicazione avviene grazie allo scambio di informazioni in forma testuale mediante XML all'interno del protocollo HTTP usato per il Web e utilizzabile praticamente su tutte le piattaforme hardware/software. Inoltre, un altro punto di forza è quello legato alla sicurezza: i messaggi testuali viaggiano sul protocolli e porte "aperte" ai firewall proprio perché utilizza lo stesso canale utilizzato per il Web. La comunicazione tra due sistemi non deve essere preceduta da noiose configurazioni ed accordi tra le parti.
- **Standard**: concetti fondamentali che stanno dietro ai Web Service sono regolati da standard approvati dalle più grandi ed importanti società ed enti d'Information Technology al mondo.

4 Architettura e ciclo di vita di un Web Service

Detto ciò, vediamo il processo di pubblicazione (da parte di chi crea il servizio) e di utilizzo (da parte di chi lo vuole utilizzare), spiegando i passaggi e gli standard che sovrintendono queste procedure. Nel testo spesso si indicherà con *client* chi usa il servizio e con *server* chi implementa il servizio. In Figura 1 è rappresentato il ciclo di vita di un Web Service che si può riassumere nelle seguenti fasi.

- 1.a. Il provider (chi crea il web service) crea il servizio la cui descrizione (nome dei metodi, parametri attesi, parametri di ritorno, ecc) è affidata ad un documento WSDL (Web Service Descriptor Language), cioè un formato standard XML. La pubblicazione del servizio viene

affidata ad un registro (terzo rispetto le parti) conforme allo standard Universal Description Discovery and Integration (UDDI).

- 1.b. Il client interroga il registro UDDI, per scoprire (fase di discovery) i servizi di cui ha bisogno (questa fase può essere omessa se si conosce già il servizio che si andrà ad utilizzare).
- 1.c. Nel momento in cui la ricerca ha esito positivo, si chiede al registro il documento WSDL, che definisce la struttura del servizio, e quindi indirizzi e modi su come interrogarlo.
- 2. Il passaggio seguente è un accordo umano tra fornitore del servizio e l'utilizzatore del servizio, ad esempio per stabilire una politica d'uso.
- 3 - 4. L'ultimo passaggio è la messa in produzione del sistema. Il client a partire dal WSDL crea uno strato software (request agent) che interagisce con lo strato software (provider agent, cioè il servizio vero e proprio) fornitore del servizio. Lo scambio di messaggi avviene utilizzando un altro protocollo XML, Simple Object Access Protocol (SOAP), anch'esso standard. Request e provider agent utilizzano SOAP per serializzare i parametri da passare al web service e i valori restituiti da quest'ultimo

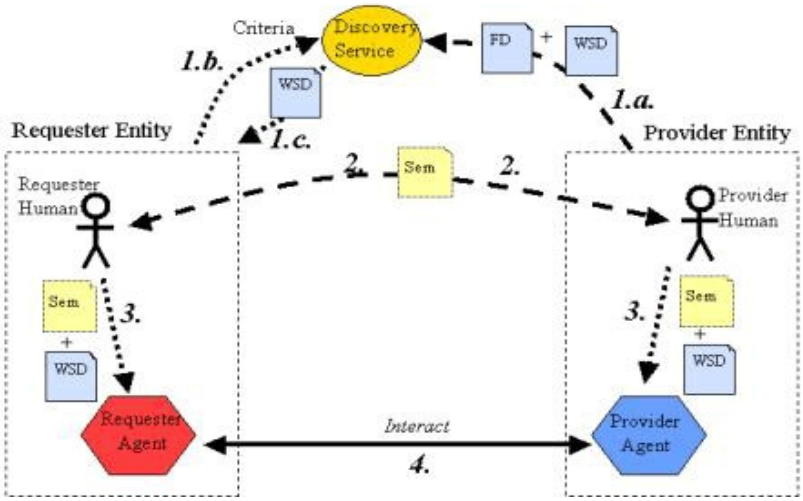

Figura 1. Ciclo di vita di un Web Service.

La cosa interessante, dal punto di vista degli sviluppatori software è che i meccanismi di comunicazione del passaggio 4 vengono creati in maniera semplice a partire dal documento WSDL. Infatti, chi si occupa di sviluppo lato server dovrà preoccuparsi solo di creare le funzioni che implementano il servizio e definire il WSDL del servizio stesso. Chi si occupa di sviluppo lato client dovrà preoccuparsi di chiamare le funzioni all'interno del proprio codice. La creazione dell'infrastruttura di comunicazione sarà automatizzata dalla presenza di opportuni strumenti di sviluppo guidati dal WSDL.

5 Meccanismo di funzionamento di un Web Service

In Figura 2 è rappresentata l'interazione tra client e server per l'utilizzo di un Web Service. Focalizziamo l'attenzione sulla parte sinistra (lato client). Sarà sufficiente conoscere il descrittore WSDL per generare un'interfaccia locale dell'oggetto remoto di cui si vogliono chiamare i metodi. Questa operazione è fatta attraverso un tool pubblicamente disponibile. L'interfaccia generata trasforma la chiamata ai metodi Java in un messaggio XML conforme allo standard SOAP che viene passato come richiesta HTTP al server. La richiesta (in forma testuale) viene estratta dal suo involucro SOAP, portata in forma binaria ed elaborata secondo la logica del programma che fornisce il servizio. La risposta, a sua volta, viene portata in forma testuale, convogliata in un messaggio SOAP ed inoltrata (generalmente su protocollo HTTP) verso il mittente. La risposta nel formato SOAP viene infine trasformata per far restituire i risultati dell'operazione alla funzione chiamata dal client.

Focalizziamo ora l'attenzione sulla parte destra (lato server). Il Web Service deve essere accessibile

attraverso un servizio di rete in attesa di ricevere richieste (esattamente come nel caso di un web server). Quindi il Web Service, dopo essere stato creato, viene installato all'interno di un *Application Container* (ad es. Apache Axis, JBoss o Tomcat). La presenza di un Application Container permette inoltre di avere a disposizione funzionalità avanzate come la gestione delle transazioni, l'eventuale accesso a sorgenti dati, le politiche di sicurezza, l'eventuale distribuzione bilanciata delle richieste su una schiera di server. Tralasciando la procedura di pubblicazione di un servizio, quello che serve alla creazione è:

1) la definizione della logica del servizio (anche una semplice classe),
2) un descrittore WSDL e
3) l'installazione su un web container predisposto a pubblicare web services.

Il middleware dell'Application Server si preoccuperà di configurare opportunamente il servizio per renderlo accessibile. L'implementazione del servizio è indipendente dal modo in cui esso verrà erogato, ad esempio, potrà essere prevista semplicemente una classe Java con relativi metodi. Il descrittore WSDL definisce in che modo si dovrà accedere alla classe che fornisce il servizio. Tutto quello che sta in mezzo (trasformazione del messaggio SOAP in chiamata di metodo Java e viceversa) è a carico del middleware presente lato client e nell'application container (JAX-RPC runtime nella Figura) che dovrà trasformare tipi di dato (primitivi e classi) in messaggi testuali SOAP (serializzazione).

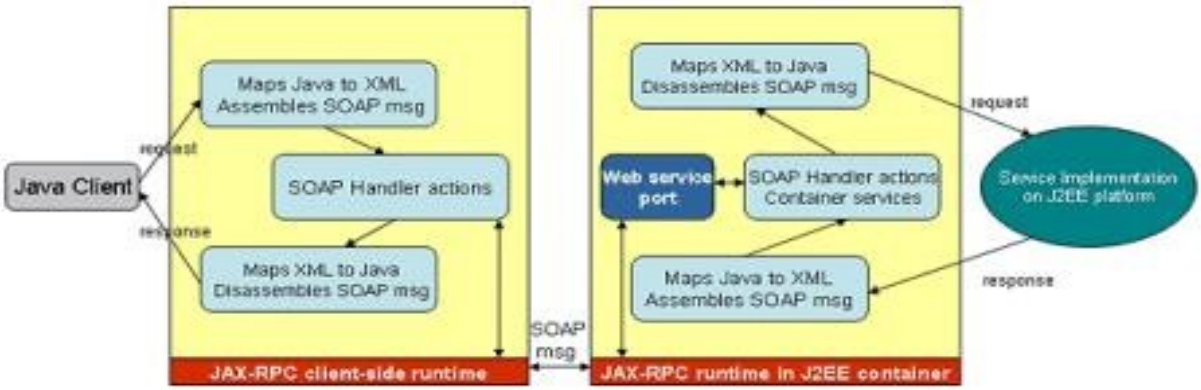

Figura 2. Comunicazione tra client e web service.

6 Creazione di un Web Service

Verrà creato ora un semplice servizio che accetta come parametro di ingresso una stringa e lo restituisce al chiamante all'interno di un messaggio di benvenuto. La semplicità del servizio permetterà di focalizzare l'attenzione sulle procedure necessarie allo sviluppo dello stesso. Tutto verrà realizzato su Application Server Jboss conforme alle specifiche J2EE. Un altro strumento necessario sarà il Java Web Services Developer Pack (JWSDP) di Sun, che serve per la creazione del descrittore WSDL e del codice Java che traduce chiamata e risposta in messaggi SOAP.

Jboss si può scaricare da http://profs.sci.univr.it/~quaglia/temp/jboss-4.0.4.GA.zip e scompattare, ad esempio, in una directory appositamente creata nella propria `/tmp/` e chiamata `jappcont/`

JWSDP si può scaricare da http://profs.sci.univr.it/~quaglia/temp/jwsdp-2_0-unix.sh in `/tmp/jappcont/` e lanciare dopo avergli dato il permesso di esecuzione

```
$>chmod 777 jwsdp-2_0-unix.sh
```

partirà un programma di installazione in cui occorre indicare in sequenza “I accept”, “No container”, “No proxy”, “Default installation” e il percorso di installazione “`/tmp/jappcont/jwsdp-2.0/`”

Prima di tutto occorre creare una struttura di directory come descritto in Figura 3 (ad es. si può creare nella propria home la cartella `hello-servlet.war/`). I file che compaiono in Figura 3 verranno

scritti o generati nel seguito dell'esercitazione. In particolare, i sorgenti Java verranno messi tutti nella cartella `classes/org/jboss/chap12/hello/` al fine di far corrispondere il nome del package.

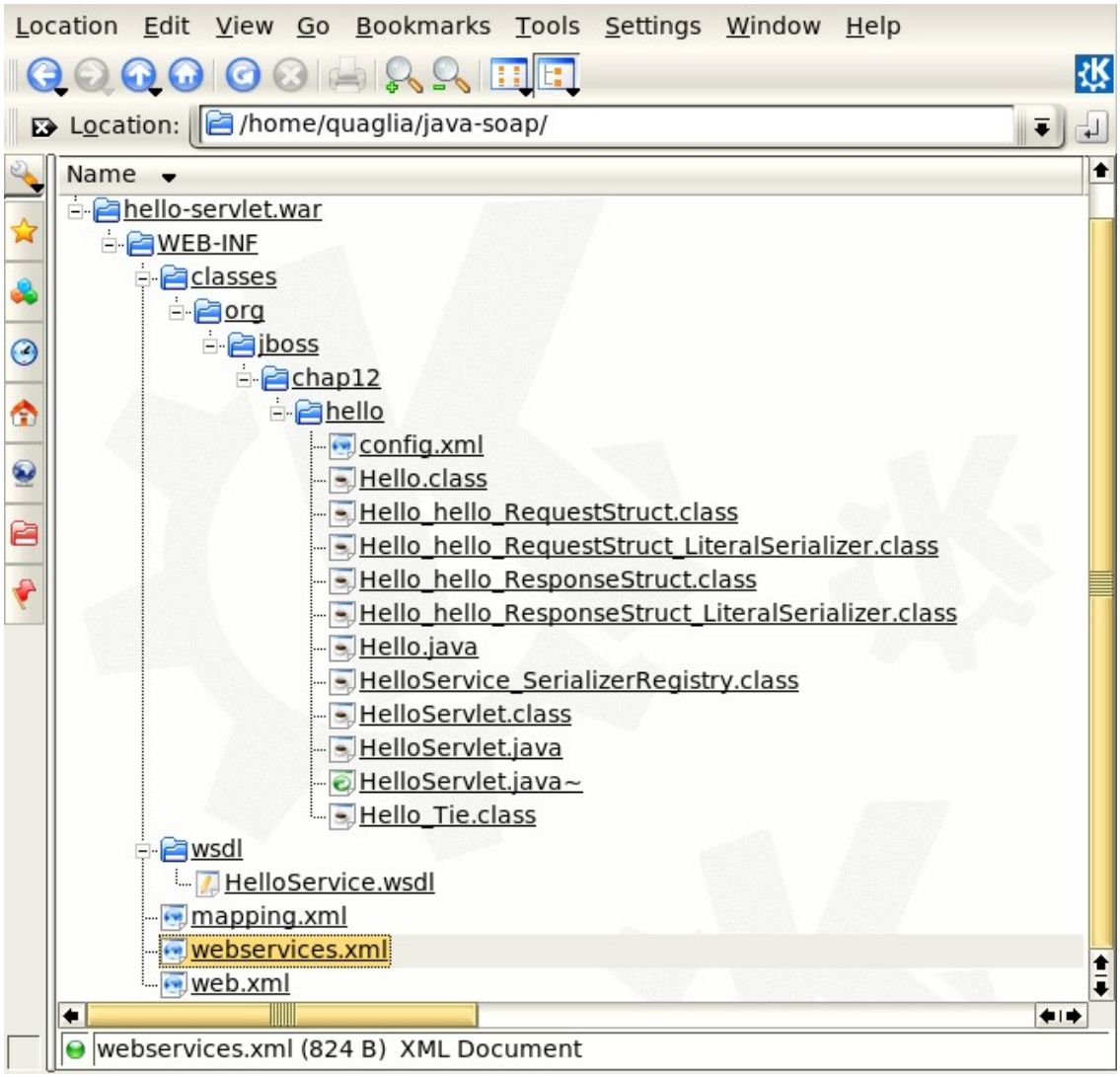

Figura 3. Struttura del package `hello-servlet.war`.

Bisogna ora creare la classe che fornisce il servizio `hello()`.

```
package org.jboss.chap12.hello;

public class HelloServlet
{
    public String hello(String name)
    {
        return "Hello " + name + "!";
    }
}
```

Serve anche un'interfaccia che rappresenti la classe presso i client che la useranno (in realtà in questa esercitazione questa interfaccia verrà usata per generare automaticamente la descrizione WSDL da cui verrà poi generata, ancora automaticamente, l'interfaccia che userà il client; è chiaro che le due interfacce saranno uguali).

```
package org.jboss.chap12.hello;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote
{
    public String hello(String name) throws RemoteException;
}
```

Occorre ora creare il file `web.xml` che contiene il descrittore della classe che servirà per la sua installazione nel Container; esso serve proprio per notificare all'application server il fatto che la classe sarà un web service; occorre mettere il file nella cartella `WEB-INF/` (è possibile recuperare il file al link http://profs.sci.univr.it/~quaglia/temp/web.xml).

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4">


    <servlet>
        <servlet-name>HelloWorldServlet</servlet-name>
        <servlet-class>org.jboss.chap12.hello.HelloServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloWorldServlet</servlet-name>
        <url-pattern>/Hello</url-pattern>
    </servlet-mapping>

</web-app>
```

Il passo successivo è quello di creare il descrittore WSDL, in modo che in fase di installazione, venga messo su il middleware necessario ad accettare le richieste dei client. Si potrebbe tale descrittore a mano, ma è un documento piuttosto complesso. Pertanto utilizzeremo il JWSDP, a cui daremo in ingresso un documento più semplice (sempre XML) più la classe: questo elaborerà i dati e restituirà il documento WSDL.

Il documento più semplice è il file `config.xml` illustrato di seguito, che si deve mettere nella cartella dei sorgenti Java e che si può scaricare al link http://profs.sci.univr.it/~quaglia/temp/config.xml.

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

<configuration xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
    <service name="HelloService"
             targetNamespace="http://hello.chap12.jboss.org/"
             typeNamespace="http://hello.chap12.jboss.org/types"
             packageName="org.jboss.chap12.hello">
        <interface name="org.jboss.chap12.hello.Hello"/>
    </service>
</configuration>
```

Il documento si presenta in maniera molto semplice: nome del servizio, namespace (per definirlo in maniera univoca), package ed interfaccia da utilizzare per generare il WSDL.

Spostiamoci ora su una finestra di comando, a livello della directory di compilazione (in maniera da

semplificare il recupero della classe compilata). E lì inseriamo il file config.xml. Occorre utilizzare ora il tool JAX RPC per creare, lato server, le classi che serviranno a rendere pubblico il servizio tramite SOAP. Procediamo alla compilazione JAX RPC nel modo seguente.

```
$>cd ~/hello-servlet.war/WEB-INF/classes/
$>javac org/jboss/chap12/hello/Hello.java
$>javac org/jboss/chap12/hello/HelloServlet.java
$>/tmp/jappcont/jwsdp-2.0/jaxrpc/bin/wscompile.sh -classpath ./ -gen:server
-f:rpcliteral -mapping mapping.xml org/jboss/chap12/hello/config.xml
```

Se tutto è in ordine verranno creati due file: il suddetto file `HelloService.wsdl` ed il file `mapping.xml`, che servirà al container per associare il servizio correttamente.

Ultimo descrittore da creare sarà il `webservices.xml` in cui si associano le definizioni dei file (wsdl e mapping) al giusto componente. Si può scaricare tale file dal link http://profs.sci.univr.it/~quaglia/temp/webservices.xml.

```
<webservices xmlns="http://java.sun.com/xml/ns/j2ee"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://www.ibm.com/
webservices/xsd/j2ee_web_services_1_1.xsd" version="1.1">
        <webservice-description>
                <webservice-description-name>HelloService</webservice-description-
name>
                <wsdl-file>WEB-INF/wsdl/HelloService.wsdl</wsdl-file>
                <jaxrpc-mapping-file>WEB-INF/mapping.xml</jaxrpc-mapping-file>
                <port-component>
                        <port-component-name>Hello</port-component-name>
                        <wsdl-port>HelloPort</wsdl-port>
                        <service-endpoint-
interface>org.jboss.chap12.hello.Hello</service-endpoint-interface>
                        <service-impl-bean>
                                <servlet-link>HelloWorldServlet</servlet-link>
                        </service-impl-bean>
                </port-component>
        </webservice-description>
</webservices>
```

A questo punto occorre mettere i file `web.xml`, `mapping.xml` e `webservices.xml` sotto la directory `WEB-INF/` ed il file `HelloService.wsdl` sotto `WEB-INF/wsdl/` (Figura 3).

Infine si crea un package `hello-servlet.war` che si installa sull'application server (attenzione al punto finale nella chiamata a `jar`):

```
$>cd ~/hello-servlet.war/
$>jar cvf /tmp/jappcont/jboss-4.0.4.GA/server/default/deploy/hello-servlet.war .
```

Ora si può avviare l'application server:

```
$>cd /tmp/jappcont/jboss-4.0.4.GA/bin
$>./run.sh
```

Per verificare che il servizio sia funzionante si può aprire un web browser sulla pagina del server che pubblica il WSDL del servizio che è stato installato (Figura 4).

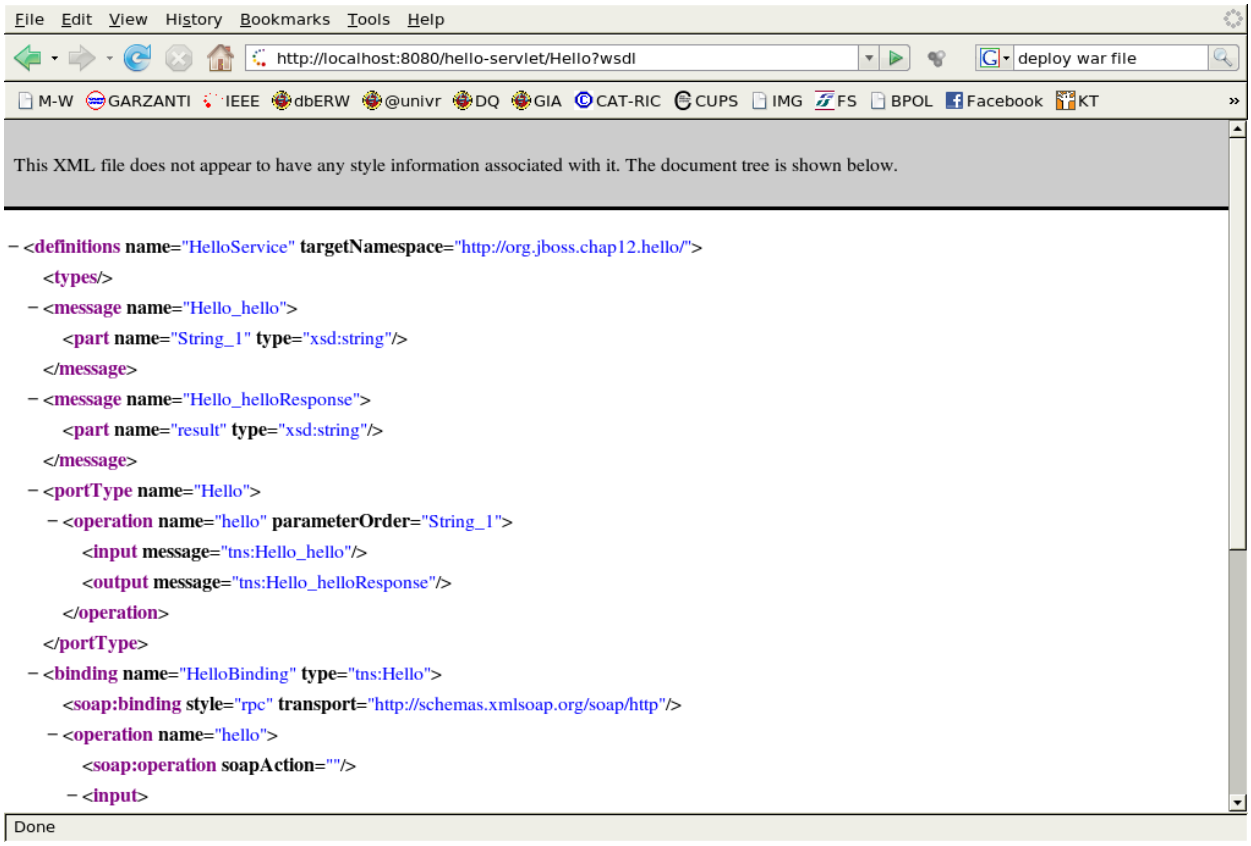

Figura 4. Verifica del funzionamento del servizio.

7 Utilizzo di un Web Service

Una volta installato il web service si può invocarne i metodi. L'ideale, visto l'argomento, sarebbe testare il web service attraverso un altro linguaggio di programmazione, proprio per vedere all'opera l'interoperabilità del componente. Rimaniamo invece in ambiente Java e vediamo un client che effettua la chiamata al Web Service. La cosa interessante sarà creare un programma Java a partire unicamente dal documento WSDL scaricato col browser come mostrato in Figura 4.

Bisogna creare una cartella di lavoro (ad es. `client/`) e copiarvi dentro il file WSDL (ad es. `hello.wsdl`) scaricato dal browser (Figura 4) usando il comando “Save As”. Occorre poi creare un file di configurazione (ad es. `clientconfig.xml`) per il tool JAX RPC che, a partire dal file WSDL e da tale file di configurazione, creerà le classi di supporto per gestire la comunicazione SOAP col server.

Il file di configurazione è il seguente (si può scaricare al link http://profs.sci.univr.it/~quaglia/temp/clientconfig.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
        xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
        <wsdl location="hello.wsdl" packageName="helloclient"/>
</configuration>
```

Esso dice qual è il doc WSDL (invece di un file si poteva mettere direttamente la URL da cui è stato scaricato) e il package Java in cui verranno messe le classi di supporto.

Per generare le classi di supporto occorre mettersi nella cartella in cui c'è il file `clientconfig.xml`

e lanciare il tool JAX RPC.

```
$>cd client/
$>/tmp/jappcont/jwsdp-2.0/jaxrpc/bin/wscompile.sh -gen:client clientconfig.xml -keep
```

Si noti che viene creata una directory `helloclient/` con dentro i file di supporto e si noti in particolare il file `Hello.java` che è praticamente identico all'omonimo file creato all'inizio in `hello-servlet.war/WEB-INF/classes/org/jboss/chap12/hello/` solo che questa volta è stato creato automaticamente !

Ora occorre scrivere la propria applicazione client che richiama il servizio tramite l'interfaccia Hello e quindi la vede come se fosse una classe locale. Per esempio ecco il file `HelloClient.java` in `~/client/` (si può scaricare il file al link http://profs.sci.univr.it/~quaglia/temp/HelloClient.java).

```
import helloclient.*;

public class HelloClient
{
    public static void main(String[] args)
        throws Exception
    {
        String argument = "Davide";

        HelloService_Impl service = new HelloService_Impl();

        Hello hello = (Hello) service.getHelloPort();

        System.out.println("hello.hello(" + argument + ")");
        System.out.println("output:" + hello.hello(argument));
    }
}
```

Occorre scaricare e salvare in `client/`

http://profs.sci.univr.it/~quaglia/temp/j2ee-1.4.jar

http://profs.sci.univr.it/~quaglia/temp/FastInfoset.jar

Ora si può compilare il client:

```
$>cd ~/client/
$>javac -classpath "./:/home/quaglia/work/jappcont/jwsdp-2.0/jaxrpc/lib/jaxrpc-
api.jar:/home/quaglia/work/jappcont/jwsdp-2.0/jaxrpc/lib/jaxrpc-
impl.jar:./j2ee-1.4.jar:/home/quaglia/work/jappcont/jwsdp-2.0/saaj/lib/saaj-
impl.jar:./FastInfoset.jar" HelloClient.java
```

Infine è sufficiente lanciare il client (ovviamente l'application server Jboss deve essere running).

```
$>cd ~/client/helloclient/
$>java HelloClient
```