

Comunicazioni in rete tramite Java

Davide Quaglia
Matteo Corsini
Simone Bronuzzi

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 dei protocolli TCP e UDP;
- 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 passato su linea di comando e visualizza ciò che viene inviato dal server (nell'ipotesi che si tratti di testo).

La sintassi generale di un URL è:

```
protocol://username:password@domain:port/path?query_string#fragment_id
```

dove la porta è sottointesa se si usa quella di default per il protocollo specificato.

Esempio di URL web

```
http://www.univr.it
```

Esempio di URL riferita ad un file sulla propria macchina (non serve essere connessi in rete)

```
file:///home/davide/output.txt
```

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

class EsempioURL
{
    public static void main(String args[])
    {
```

```

String indirizzo;

if (args.length > 0)
{
    indirizzo = args[0];
}
else
{
    System.out.println("Uso: EsempioURL URL");
    return;
}

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. Bisogna lanciare client e server da due finestre di Shell differenti.

```

// CLIENT
// 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); // IP e porta del server
        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());
    }
}
}

```

// SERVER

```

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("Connessione 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);
        }
    }
}

```

Esercizio 1

- 1) Cosa succede se lancio da una finestra diversa una nuova istanza di server quando è ancora attiva la prima ?
- 2) Modificare il sorgente del server in modo da prendere da linea di comando la porta su cui attendere. Si utilizzi il seguente frammento di codice come fonte di ispirazione.

```

int firstArg;
if (args.length > 0) {
    try {
        firstArg = Integer.parseInt(args[0]);
    } catch (NumberFormatException e) {
        System.err.println("Argument must be an integer");
        System.exit(1);
    }
}

```

- 3) Modificare il sorgente del client in modo da prendere da linea di comando sia l'indirizzo IP del server sia la porta su cui attende. Infatti l'indirizzo "localhost" significa "la propria interfaccia di rete". Suggerimento: bisogna modificare la chiamata al costruttore di Socket come segue dove IP è di tipo String e porta è di tipo int.

```
s = new Socket(InetAddress.getByName(IP), porta);
```

- 4) Scoprire l'indirizzo IP della propria interfaccia di rete usando il comando

```
$ /sbin/ifconfig
```

5) Connettere il proprio client al server del vicino.

ESEMPIO 3 - Un mini servizio di trasferimento file

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.)

```
// CLIENT

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); // IP e porta del server
            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());
    }
}

// SERVER

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(c.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
        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);
    }
}
}
}
}

```

Esercizio 2

1) Modificare client e server in modo che prendano da riga di comando anche IP/porta e porta, rispettivamente.

ESEMPIO 4 - Una mini-calcolatrice client / server

Il client invia al server una serie di numeri in linea di comando separati da spazio; l'ultimo numero deve essere lo zero. Il server effettua la somma dei valori ricevuti e la ritrasmette al client.

// CLIENT

```

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); // IP e porta del server
        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();
        c.close();
    }
    catch (IOException e)
    {
        System.err.println(e.getMessage());
    }

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

```

// SERVER

```

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);
    }
}
}
}
}

```

Esercizio 3

1) Modificare client e server in modo che prendano da riga di comando anche IP/porta e porta, rispettivamente.

2) Lanciare il client antepoendo il comando time

```
$ time java Client2 IP porta 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 0
real XXXX
user YYYY
system ZZZZ
```

dove “real” è il tempo lordo che non considera il multitasking, “user” è il tempo netto di utilizzo della CPU in modalità utente (ad esempio per fare delle operazioni matematiche), “system” è il tempo netto di utilizzo della CPU impegnata in attività di input/output.

3) Che differenza di tempi si ottiene con server locale (sulla stessa macchina del client) o remoto ? Trovare i tempi sia passando pochi numeri sia tanti numeri e compilare una tabella simile. Si possono fare ipotesi sulle ragioni dei risultati ?

Time	N=5		N=20	
	Locale	Remoto	Locale	Remoto
real				
user				
system				

ESEMPIO 5 – una semplice applicazione client/server UDP

Il client invia al server una frase. Il server risponde spedendo al client il contenuto ricevuto convertito in maiuscolo. Il protocollo che viene utilizzato per la comunicazione è UDP.

```
// CLIENT

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

class UDPClient
{
    public static void main(String args[]) throws Exception
    {
        try{
            BufferedReader inFromUser = new BufferedReader(new
                InputStreamReader(System.in));

            DatagramSocket clientSocket = new DatagramSocket();
            InetAddress IPAddress = InetAddress.getByName("localhost");// IP destinazione
            byte[] sendData = new byte[1024];
            byte[] receiveData = new byte[1024];
            String sentence = inFromUser.readLine();
            sendData = sentence.getBytes();
            DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length,
                IPAddress, 9876); // IP e PORTA destinazione
            clientSocket.send(sendPacket);
            DatagramPacket receivePacket = new DatagramPacket(receiveData,
                receiveData.length);
            clientSocket.receive(receivePacket);
            String modifiedSentence = new String(receivePacket.getData());
            System.out.println("FROM SERVER:" + modifiedSentence);
        }
    }
}
```

```

        clientSocket.close();
    }catch(Exception e){}
}
}

// SERVER

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

class UDPServer
{
    public static void main(String args[]) throws Exception
    {
        try{
            DatagramSocket serverSocket = new DatagramSocket(9876);
            byte[] receiveData = new byte[1024];
            byte[] sendData = new byte[1024];
            while(true)
            {
                try{
                    DatagramPacket receivePacket = new DatagramPacket(receiveData,
                                                                    receiveData.length);
                    serverSocket.receive(receivePacket);
                    String sentence = new String( receivePacket.getData());
                    System.out.println("RECEIVED: " + sentence);
                    InetAddress IPAddress = receivePacket.getAddress();
                    int port = receivePacket.getPort();
                    String capitalizedSentence = sentence.toUpperCase();
                    sendData = capitalizedSentence.getBytes();

                    DatagramPacket sendPacket =
                        new DatagramPacket(sendData, sendData.length, IPAddress, port);
                    serverSocket.send(sendPacket);
                }
                catch(UnknownHostException ue){
                }
            }
            catch(java.net.BindException b){
            }
        }
    }
}

```

Esercizio 4

1) Si prenda in considerazione il client: dove vengono impostati IP e porta del server a cui mandare la richiesta ? Rispetto al codice dei client precedenti si nota una differenza a questo proposito ? Perché ?

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 Service**, che si presentano come moduli software distribuiti i quali collaborano fornendo determinati servizi in maniera standard. Il Web Service è una funzionalità messa a disposizione in modalità server ad altri moduli software implementati come client. I componenti software **interagiscono tra loro attraverso protocolli Web Service (ad es. REST, SOAP) trasportati in connessioni HTTP (da cui il nome Web Service)**. E' compito del protocollo Web Service definire una forma serializzata dei parametri attuali da passare al web service e dei valori restituiti da quest'ultimo.

Quando uno sviluppatore crea un Web Service, si deve preoccupare di definire:

1. la **logica di funzionamento del servizio**, ovvero quello che dovrà fare; questo può spaziare da una semplice classe ad un'applicazione molto complessa;
2. il **web container** su cui verrà installato (ad es. Apache, Tomcat, Jboss, Glassfish) e che ne consentirà l'uso da parte dei client.

Motivazioni

Si sta osservando un sempre maggiore utilizzo dell'approccio SOA nella creazione di applicativi software per i seguenti motivi:

- **Protezione della Proprietà Intellettuale.** Il cuore dell'applicazione rimane sul server e non viene distribuito neanche come eseguibile agli utenti.
- **Requisiti di potenza di calcolo.** La potenza di calcolo richiesta per l'applicazione può essere soddisfatta dal server senza gravare sulla CPU e il consumo energetico del client (ad es. smartphone)
- **Requisiti di memoria di massa.** La grande memoria di massa richiesta per l'applicazione può essere soddisfatta dal server senza gravare sulle risorse del client.
- **Comodità di distribuzione agli utenti.** Ogni utente usa un software client minimale (eseguibile per PC, plugin nel browser web, app per smartphone) che si connette al server. Questo comporta le seguenti conseguenze:
 - **Aggiornamento istantaneo.** La logica applicativa interna al server può cambiare in qualsiasi momento (ad es. per eliminare bug, adeguamenti normativi di un SW legale, ecc.) senza dover re-distribuire aggiornamenti agli utenti che si trovano istantaneamente ad utilizzare il software aggiornato.
 - **Sviluppo e mantenimento di una sola versione del prodotto** a fronte di molteplici piattaforme utente (ad es. Windows, Linux, MAC, smartphone).
 - **Maggiori ritorni economici** col nuovo approccio **pay-per-use** rispetto al tradizionale approccio della distribuzione e installazione dell'applicativo sui PC degli utenti. Eliminazione del fenomeno della pirateria.

Vantaggi tecnologici

La nuova struttura collaborativa distribuita porta ad una serie di **vantaggi tecnologici**:

- **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 client.
- **Interoperabilità:** I Web Service 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. la logica applicativa incapsulata all'interno dei Web Service è completamente decentralizzata ed accessibile attraverso Internet da piattaforme, dispositivi, sistemi operativi e linguaggi di programmazione differenti.

- **Semplicità di sviluppo e di rilascio:** Un'applicazione è costituita da moduli indipendenti, interagenti tramite la rete; la modularità semplifica lo sviluppo. Rilasciare un WS significa solo esporlo al Web.
- **Semplicità di installazione:** la comunicazione avviene grazie allo scambio di informazioni in forma testuale all'interno del protocollo HTTP usato per il Web e utilizzabile praticamente su tutte le piattaforme hardware/software. I messaggi testuali viaggiano sullo stesso canale utilizzato per il Web e quindi sono già abilitati dai firewall. 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 SOAP

Vediamo ora uno dei protocolli per scambiare messaggi relativi a Web Service che utilizza l'approccio della **chiamata remota di metodi esposti da oggetti residenti su altri nodi della rete**.

I componenti software interagiscono 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 metodo remoto e dei valori restituiti da quest'ultimo. Viene inoltre introdotto un **descrittore WSDL (Web Service Description Language)** che definisce:

- le operazioni messe a disposizione dal servizio;
- il modo per utilizzare il servizio (protocollo da utilizzare, formato dei messaggi ecc);
- l'endpoint dove utilizzare il servizio (l'indirizzo che permette di raggiungerlo).

WSDL e SOAP seguono il formato XML al fine di essere trattabili da strumenti automatici.

4.1 Architettura e ciclo di vita di un Web Service SOAP

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.

FASE 1:

- (a) Il provider (chi crea il web service) crea il servizio la cui descrizione (nome dei metodi, parametri attesi, valori 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).
- (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).
- (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 con cui interrogarlo.

FASE 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.

FASE 3: 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. Tali componenti, chiamati *stub* e *skeleton* (rispettivamente lato client e lato server), si occupano della gestione dei messaggi SOAP, assemblando e disassemblando messaggi nel formato XML per permettere la comunicazione automatica tra due sistemi informatici.

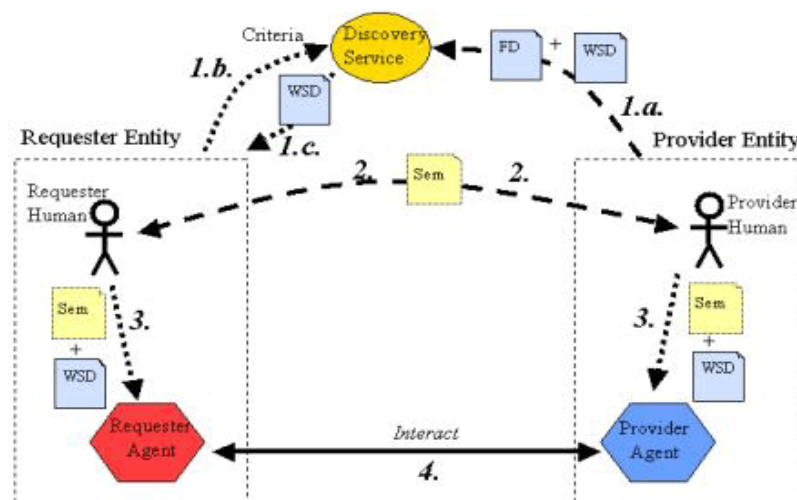


Figura 1. Ciclo di vita di un Web Service.

La cosa interessante, dal punto di vista degli sviluppatori software è che i meccanismi di comunicazione della FASE 3 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.

Nella Figura 2 è rappresentato in dettaglio la comunicazione tra client e server. Le fasi sono:

- 1) l'interfaccia lato client chiama un metodo del Web Service attraverso lo stub, che si occupa di trasformare la richiesta del client in un messaggio XML conforme al protocollo SOAP;
- 2) il messaggio viene spedito al server tramite protocollo SOAP (REQUEST);
- 3) lo skeleton riceve il messaggio SOAP, lo trasforma in chiamate al codice Java del servizio lato server;
- 4) il servizio lato server riceve la chiamata, elabora i dati e tramite lo skeleton assembla un messaggio SOAP pronto per essere spedito al client;
- 5) la risposta (RESPONSE) viene inviata al client e ricevuta dallo stub;
- 6) lo stub estrae la risposta dal messaggio SOAP e la restituisce all'interfaccia che ha chiamato il servizio.

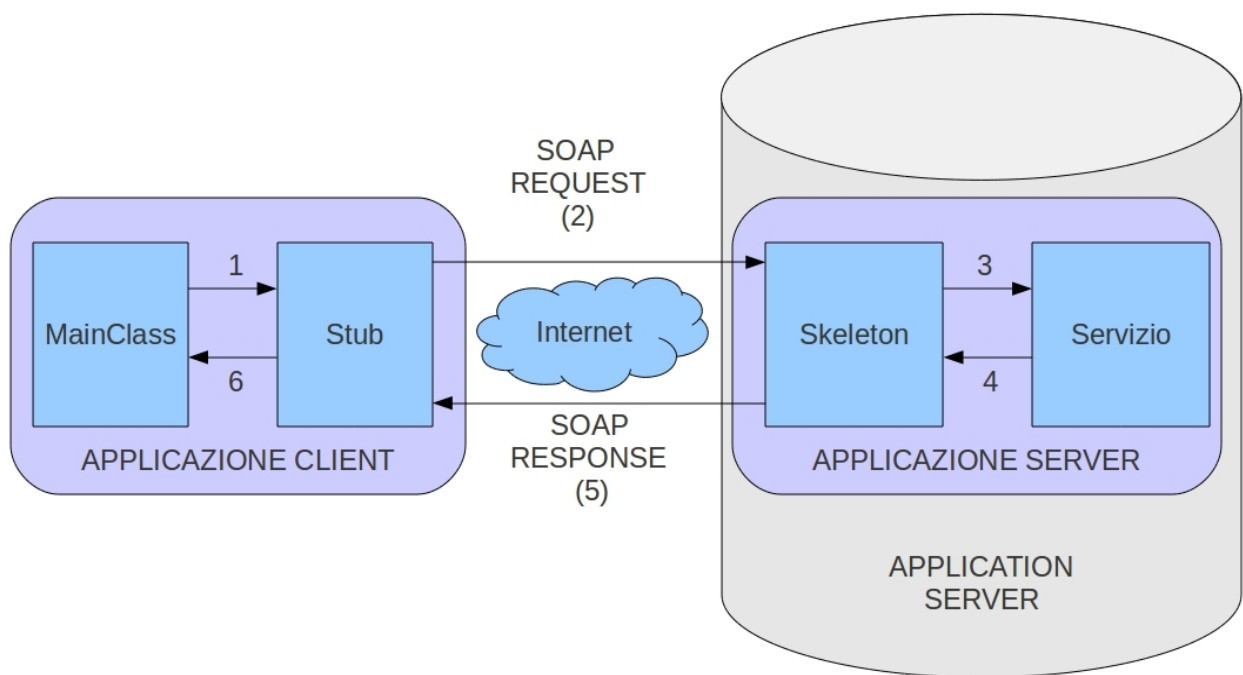


Figura 2: dettaglio comunicazione tra client e server

5 Esercizio su creazione di un web service SOAP

In questa sezione impareremo a costruire, pubblicare ed eseguire un Web Service a partire da una semplice classe Java e la relativa interfaccia. Il servizio che si vuole offrire è una semplice calcolatrice.

Il software che sarà utilizzato (su sistema operativo Ubuntu) è:

- Apache Tomcat 6.0, come application container;
- Apache Axis 1.4, per gestire SOAP;
- Openjdk-6, per compilare ed eseguire programmi Java.

Vediamo passo per passo come procedere.

5.1 Scaricare e decomprimere il pacchetto soap.tar.gz

Il pacchetto soap.tar.gz (circa 11 MB) contiene tutti i file necessari per questa esercitazione. Nel caso nella propria home non ci sia spazio sufficiente, scaricare e scompattare il file in /tmp/ (attenzione che la cartella viene svuotata ad ogni riavvio della macchina). Vedere la tabella sottostante per maggiori dettagli sul contenuto dell'archivio.

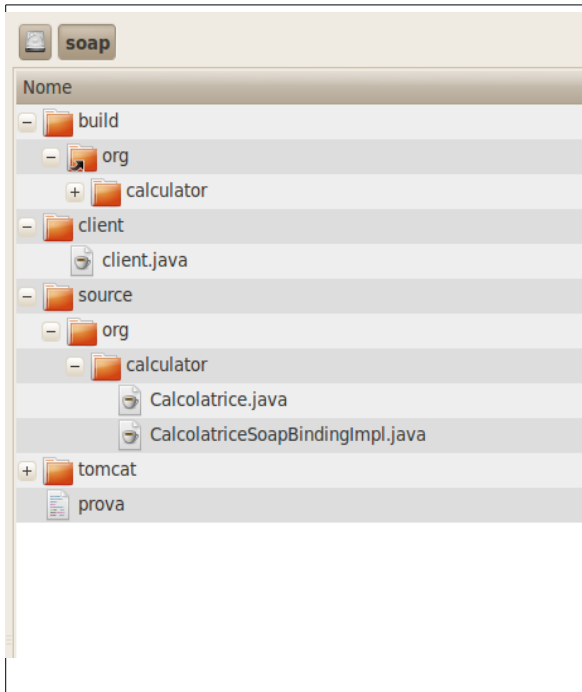
	CARTELLA	CONTENUTO
	build/org	link simbolico alla cartella /tomcat/webapps/axis/WEB-INF/classes/org, dove sono contenuti i file necessari per eseguire l'applicazione lato server
	client	Sorgente del client
	source/org/calculator	Sorgenti dell'interfaccia (Calcolatrice.java) e della relativa implementazione (CalcolatriceSoapBindingImpl.java)
	tomcat	Application container con AXIS già installato

Tabella 1: contenuto del pacchetto soap.tar.gz

➤ Nel presente documento, da questo punto in poi quando viene indicato un comando da digitare a terminale, è inteso che ci si trovi nella cartella principale del pacchetto soap.tar.gz (in questa esercitazione la cartella è soap/).

5.2 Impostazione del classpath

E' necessario impostare correttamente il classpath affinché il compilatore Java possa trovare correttamente le librerie necessarie. Per eseguire questa operazione, digitare i seguenti comandi:

```
matteo@juno:/soap$ export CLASSPATH="";  
matteo@juno:/soap$ for i in tomcat/webapps/axis/lib/*; do export  
CLASSPATH=$CLASSPATH:$PWD/$i; done  
matteo@juno:/soap$ export CLASSPATH=$CLASSPATH:..$PWD/build/
```

abbiamo così impostato Java per raggiungere tutte le librerie di AXIS nonché il client.

Prima di procedere, è inoltre opportuno avviare l'application container Tomcat digitando:

```
matteo@juno:/soap$ ./tomcat/bin/startup.sh
```

Se si vuole spegnere il l'application container usare il comando:

```
matteo@juno:/soap$ ./tomcat/bin/shutdown.sh
```

5.3 Il codice del Web Service

Per iniziare, abbiamo bisogno di almeno due elementi:

- una classe contenente il codice del programma che vogliamo eseguire (source/org/calculator/CalcolatriceSoapBindingImpl.java);
- l'interfaccia che rappresenterà la classe (source/org/calculator/Calcolatrice.java).

5.4 L'interfaccia Calcolatrice

```
package org.calculator;  
public interface Calcolatrice  
{  
    String calcola(String operazione, Double x, Double y);  
}
```

questa interfaccia contiene la chiamata ai metodi della classe.

5.5 La classe CalcolatriceSoapBindingImpl

```
package org.calculator;  
  
public class CalcolatriceSoapBindingImpl implements org.calculator.Calcolatrice{  
    public java.lang.String calcola(String operazione, Double x, Double y) throws  
        java.rmi.RemoteException {  
  
        Double risultato = 0.0;  
        String simbolo = "";  
  
        if (operazione.equalsIgnoreCase("somma"))  
        {  
            risultato = x+y;  
            simbolo = "+";  
        }  
        else if (operazione.equalsIgnoreCase("sottrazione"))  
        {  
            risultato = x-y;  
            simbolo = "-";  
        }  
        else if (operazione.equalsIgnoreCase("prodotto"))  
        {  
            risultato = x*y;  
            simbolo = "x";  
        }  
        else if (operazione.equalsIgnoreCase("divisione"))  
        {  
            risultato = x/y;  
            simbolo = "/";  
        }  
        if (simbolo != "")  
        {  
            return x + " " + simbolo + " " + y + " = " + risultato;  
        }  
        else  
        {  

```

```

        return "ERRORE";
    }
}
}

```

questa classe è il codice vero e proprio del servizio. Quando verrà eseguita l'interfaccia, questa chiamerà a sua volta i metodi presenti in questa classe.

➤ *Attenzione: il programma WSDL2Java creerà sempre questo file nella cartella di destinazione (~/.build/org/calculator), a meno che non sia già presente (dovrebbe essere il nostro caso). Prestare attenzione a trasferire in quella cartella il file corretto; in caso contrario, il file sarà ricreato da zero e il client restituirà sempre NULL.*

In questa fase possiamo apportare eventuali modifiche ai sorgenti. Una volta terminato, li spostiamo nella cartella di AXIS, in modo da preservarli da eventuali modifiche accidentali, e compiliamo l'interfaccia:

```

matteo@juno:/soap$ cp source/org/calculator/* build/org/calculator/
matteo@juno:/soap$ javac build/org/calculator/Calcolatrice.java

```

5.6 Creazione del WSDL

Una volta creata l'interfaccia, possiamo procedere alla creazione del WSDL, che servirà per definire il funzionamento di SOAP. Per procedere alla creazione, digitiamo il seguente comando:

```

matteo@juno:/soap$ java org.apache.axis.wsdl.Java2WSDL -o
./build/Calcolatrice.wsdl -lhttp://localhost:8080/axis/services/Calcolatrice -i
org.calculator.Calcolatrice -n "Calcolatrice" -p"org.calculator" "Calcolatrice"
org.calculator.Calcolatrice

```

dove:

- -o: indica il percorso dove vogliamo creare il file WSDL. E' consigliabile indicare il path assoluto;
- -l: indica l'endpoint, ovvero a quale indirizzo sarà raggiungibile il Web Service;
- -n: indica il nome del Web Service;
- -p: indica la mappatura dal package al namespace.

Se tutto ha funzionato a dovere, otterremo il file *Calcolatrice.wsdl* nella cartella *build/*.

Esercizio 5

Aprire *Calcolatrice.wsdl* e *build/org/calculator/Calcolatrice.java* con un editor di testo e identificare gli elementi dell'interfaccia Java che sono ripresi nel file WSDL.

5.7 Creazione dello stub e dello skeleton

Questi due elementi sono indispensabili per permettere al servizio di funzionare¹. In particolare:

- lo **stub**: risiede nel client e fa da tramite tra esso e internet;
- lo **skeleton**: risiede nel server e, interponendosi tra internet e il servizio vero e proprio, ne permette l'esecuzione da remoto.

Per la loro creazione, si utilizza il programma WSDL2Java incluso in AXIS, dandogli in ingresso il WSDL creato nel passo precedente. Il comando è il seguente:

```

matteo@juno:/soap$ java org.apache.axis.wsdl.WSDL2Java -o ./build -d Session -s
-S true -Ncalcolatrice org.calculator ./build/Calcolatrice.wsdl

```

¹ Vedere Figura 2.

In questo modo avremo ottenuto tutti i file necessari per l'installazione del nostro web service all'interno di AXIS.

➤ *Attenzione: sono file creati in automatico che non vanno modificati; se si ha la necessità di variare il codice e/o il funzionamento del programma, è necessario modificare i file originali (Calcolatrice.java e CalcolatriceSoapBindingImpl.java) e ripetere la procedura dal punto 1).*

I file che sono stati creati in questo passo sono:

- **deploy.wsdd** e **undeploy.wsdd**: permettono rispettivamente l'installazione e la disinstallazione del Web Service all'interno di AXIS;
- **Calcolatrice.java**: nuova interfaccia che contiene le chiamate a java.rmi.Remote;
- **CalcolatriceService.java**: l'interfaccia di servizio lato client ;
- **CalcolatriceServiceLocator.java**: implementazione del servizio lato client;
- **CalcolatriceSoapBindingImpl.java**: implementazione del servizio lato server;
- **CalcolatriceSoapBindingStub.java**: è lo stub;
- **CalcolatriceSoapBindingSkeleton.java**: è lo skeleton.

Possiamo ora compilare tutti i file .java con il comando

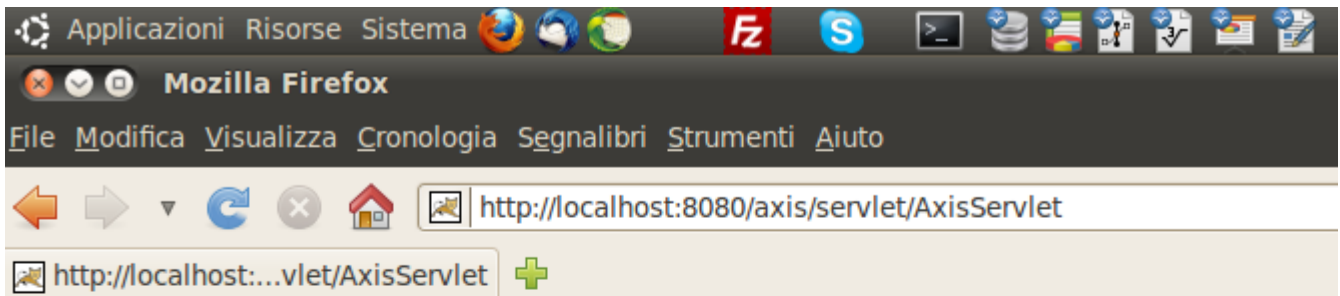
```
javac build/org/calculator/*.java
```

5.8 Installazione del Web Service nell'application container

Possiamo ora procedere all'installazione del nostro web service all'interno di AXIS. Per procedere, dobbiamo eseguire il comando

```
matteo@juno:/soap$ java org.apache.axis.client.AdminClient  
tomcat/webapps/axis/WEB-INF/classes/org/calculator/deploy.wsdd
```

possiamo verificare l'avvenuta installazione aprendo un browser web e digitando l'URL <http://localhost:8080/axis/servlet/AxisServlet>. Dovremmo vedere qualcosa di simile:



And now... Some Services

- AdminService ([wsdl](#))
 - AdminService
- Calcolatrice ([wsdl](#))
 - somma
 - sottrazione
 - prodotto
 - divisione
- asyncService ([wsdl](#))
 - process
- Version ([wsdl](#))
 - getVersion

5.9 Creazione del client

L'insieme di file che abbiamo creato nel punto 2) è di per sé sufficiente per eseguire il nostro web service da remoto; bisogna solo avere l'accortezza di modificare il package a cui puntano (*org.calculator* anziché *Calcolatrice_pkg*), ma risulterebbe un'operazione scomoda e suscettibile di errori.

Inoltre, il bello di SOAP consiste anche nella possibilità di creare tutti i file necessari a partire dal solo WSDL, senza compiere macchinose modifiche manuali.

Per farlo, è necessario posizionarsi nella cartella in cui vogliamo che vengano creati i file (nel nostro caso la cartella *client*) ed eseguire il seguente comando:

```
matteo@juno:/soap$ cd client
matteo@juno:/soap/client$ java org.apache.axis.wsdl.WSDL2Java
http://localhost:8080/axis/services/Calcolatrice?wsdl
```

avremo così ottenuto la cartella *Calcolatrice_pkg*, che contiene tutti i sorgenti lato client.

Ora non ci resta che compilarli:

```
matteo@juno:/soap/client$ javac Calcolatrice_pkg/*.java
matteo@juno:/soap/client$ javac client.java
```

5.10 Il codice del client

```
import Calcolatrice_pkg.*;

public class client
{
    public static void main(String args[] ) throws Exception
    {
        CalcolatriceService service = new CalcolatriceServiceLocator();
```

```

        Calcolatrice handler = service.getCalcolatrice();

        String operazione = args[0];
        Double x = Double.parseDouble(args[1]);
        Double y = Double.parseDouble(args[2]);

        System.out.println(handler.calcola(operazione, x,y));
    }
}

```

5.11 Esecuzione del client

Ora possiamo finalmente verificare il funzionamento di SOAP e dei Web Services!

Digitiamo:

```
matteo@juno:/soap/client$ java client operazione x y
```

dove dobbiamo sostituire:

- *operazione* con l'operazione che preferiamo (somma, sottrazione, prodotto o divisione);
- *x* e *y* con i due numeri che dobbiamo elaborare

Una volta dato l'invio, attendiamo un paio di secondi per permettere al server di ricevere la nostra richiesta, elaborarla, e rispedirci la risposta, il tutto tramite HTTP!

Come abbiamo detto nella parte teorica, la nostra calcolatrice è implementata nel server. Il client si limita a chiamare l'interfaccia locale, passare i tre parametri operazione, x e y ed attendere il risultato, ma non sa assolutamente quale calcolo verrà svolto sul server. Questo porta agli sviluppatori il grande vantaggio di poter liberamente modificare il servizio offerto e, nel momento in cui rieseguiranno il deploy dell'applicazione sull'application container, la nuova implementazione sarà immediatamente disponibile all'esterno, senza che gli utenti debbano eseguire alcuna operazione.

Esercizio 6

1) Provare cosa succede se si invoca il client dopo aver spento l'application container (Suggerimento: spegnere il server con la procedura di shutdown in tomcat/bin/).

2) Provare a invocare con il client il servizio in esecuzione su un altro PC.

Suggerimento: il file WSDL e il codice stub/skeleton devono essere rigenerati e ricompilati. In questa fase deve essere cambiata l'URL su cui si trova il servizio.

3) Modificare l'implementazione della calcolatrice (ad esempio restituendo una frase scherzosa invece del risultato) all'insaputa dell'altro studente che sta invocando il servizio da un'altro PC.

Suggerimento: occorre ricompilare la classe modificata e occorre re-installare il servizio nell'application server.