

# Scrittura di applicazioni di rete

## Indice

|  |    |
|--|----|
| Java Socket.....                               | 1  |
| 1 Introduzione.....                            | 1  |
| 2 Apertura di un URL.....                      | 2  |
| Socket in TCP.....                             | 4  |
| 1 Echo server UPPERCASE.....                   | 4  |
| 2 Servizio di trasferimento file.....          | 6  |
| 3 Calcolatrice client / server.....            | 8  |
| 4 Java WEB Server.....                         | 10 |
| 5 Chat client/server.....                      | 11 |
| UDP vs TCP.....                                | 13 |
| 1 UDP echo server UPPERCASE.....               | 13 |
| Socket in C.....                               | 15 |
| Architetture orientate ai servizi.....         | 19 |
| 1 Vantaggi tecnologici.....                    | 19 |
| 2 Motivazioni.....                             | 19 |
| REST.....                                      | 21 |
| 1 Note tecniche.....                           | 21 |
| 2 Esercizi.....                                | 21 |
| SOAP.....                                      | 27 |
| 1 Ciclo di vita di un Web Service SOAP.....    | 27 |
| 2 Architettura di un sistema che usa SOAP..... | 28 |
| 3 Creazione di un web service.....             | 30 |
| 4 Creazione del WSDL.....                      | 32 |
| 5 Creazione dello stub e dello skeleton.....   | 32 |
| 6 Esecuzione del client.....                   | 35 |
| Appendice A.....                               | 36 |
| Appendice B.....                               | 38 |
| Appendice C.....                               | 39 |

## Java Socket

### 1 Introduzione

#### Il package *java.net*

Il package [java.net] fornisce interfacce e classi per l'implementazione di applicazioni di rete.

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

## I/O tramite socket

Normalmente, un server in esecuzione ha un socket legato ad una specifica porta. Il server aspetta e ascolta eventuali connessioni provenienti dai client.

Il client conosce l'hostname della macchina su cui il server è in esecuzione e il numero della porta sul quale il server è in ascolto.

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.

## 2 Apertura di un URL

La classe URL rappresenta un Uniform Resource Locator, ovvero un puntatore verso una risorsa nel World Wide Web. In questo esempio scriverete una classe Java la quale, dato un indirizzo URL passato tramite linea di comando, visualizza ciò che viene inviato dal server (nell'ipotesi che si tratti di testo).

### Ricorda

La sintassi generale di un URL è:

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

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

Esempio di URL web (qual'è la porta di default?)

`http://www.univr.it`

Esempio di URL riferita ad un file sulla propria macchina

(non serve essere connessi in rete ... perché?)

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

In questa lezione vengono utilizzate (tra le altre) le classi URL, URLConnection e MalformedURLException.

La Javadoc è vostra amica, trovate i dettagli delle classi ai seguenti link:

URL (<http://docs.oracle.com/javase/7/docs/api/java/net/URL.html>)

URLConnection (<http://docs.oracle.com/javase/7/docs/api/java/net/URLConnection.html>)

MalformedURLException

(<http://docs.oracle.com/javase/7/docs/api/java/net/MalformedURLException.html>)

**Attenzione:** copia\incolla, a differenza della Javadoc, è, spesso, tuo nemico...

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

class EsempioURL
{
    public static void main(String args[])
    {
        String indirizzo=null;
```

```

    if (args.length > 0)
    {
        indirizzo = args[0];
    }
    else
    {
        System.out.println("Uso: java EsempioURL URL");
        System.exit(-1);
    }

    URL url = null;
    try
    {
        url = new URL(indirizzo);
        System.out.println("URL aperto: " + url);
    }
    catch (MalformedURLException e)
    {
        System.out.println("URL errato: " + url);
    }

    URLConnection connection = null;

    try
    {
        System.out.print("Connessione in corso...");
        connection = url.openConnection();
        connection.connect();
        System.out.println("ok.");
        BufferedReader reader = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
        System.out.println("Lettura dei dati...");
        String str;
        while( (str = reader.readLine()) != null )
            System.out.println(str);
    }
    catch (IOException e)
    {
        System.out.println(e.getMessage());
    }
}

```

# Socket in TCP

## 1 Echo server UPPERCASE

In questo esempio andremo ad implementare un echo server che una volta ricevuta una, le converte in maiuscolo, e le rimanda al client. Lanciare client e server da due diverse shell.

### Documentazione:

All about sockets (<https://docs.oracle.com/javase/tutorial/networking/sockets/index.html>)  
Socket (<http://docs.oracle.com/javase/7/docs/api/java/net/Socket.html>)  
ServerSocket (<http://docs.oracle.com/javase/7/docs/api/java/net/ServerSocket.html>)  
BufferedOutputStream (<https://docs.oracle.com/javase/7/docs/api/java/io/BufferedOutputStream.html>)  
BufferedReader (<https://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html>)  
PrintWriter (<https://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html>)  
InputStreamReader (<http://docs.oracle.com/javase/7/docs/api/java/io/InputStreamReader.html>)

### Client:

```
//CLIENT

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

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

        // socket
        Socket clientSocket = null;

        try {
            clientSocket = new Socket("localhost", 11111); // IP e porta del server
            System.out.println("Socket creata: " + clientSocket);

            // creazione del BufferedOutputStream e PrintWriter per
            // inviare la stringa al server
            BufferedOutputStream outBuffer = new
BufferedOutputStream(clientSocket.getOutputStream());
            PrintWriter outWriter = new PrintWriter(outBuffer, true);

            // creazione del BufferedReader per leggere la risposta del server
            BufferedReader reader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

            // Invio messsaggio
            outWriter.println("echo up");

            // ricezione risposta dal server
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println("Ricevuto: " + line);
                if (line.equals("Stop"))
                    break;
            }
            // chiudo il Socket (client)
            clientSocket.close();
            // chiudo PrintWriter
            outWriter.close();
            // chiudo BufferedReader
            reader.close();
        }
    }
}
```

```

    } catch (IOException e) {
        System.out.println("Error:" + e.getMessage());
    }
}

```

Server:

```

// SERVER

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

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

        // sockets
        ServerSocket serverSocket = null;
        Socket clientSocket = null;

        try {
            System.out.print("Creazione ServerSocket...");
            serverSocket = new ServerSocket(11111);

            System.out.print("Attesa connessione...");
            clientSocket = serverSocket.accept();
            System.out.println("Connessione da " + clientSocket);

            // creazione BufferedReader per leggere stringa in arrivo
            BufferedReader reader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

            // creazione del BufferedOutputStream per inviare la stringa maiuscola
            BufferedOutputStream outBuffer = new
BufferedOutputStream(clientSocket.getOutputStream());
            PrintWriter outWriter = new PrintWriter(outBuffer, true);

            // ricezione della stringa
            String text = new String(reader.readLine());
            System.out.println(text);

            // invio della nuova stringa in maiuscolo
            outWriter.println(text.toUpperCase());
            outWriter.println("Stop");

            // chiudo BufferedReaser
            reader.close();
            // chiudo PrintWriter
            outWriter.close();
            // chiudo il Socket (client)
            clientSocket.close();
            // chiudo ServerSocket
            serverSocket.close();
        } catch (Exception e) {
            System.out.println("Errore: " + e);
            System.exit(3);
        }
    }
}

```

## Esercizio

1. Cosa succede se lancio da una finestra diversa una nuova istanza del 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.

3. Modificare il sorgente del client in modo da prendere da linea di comando sia: IP del server a cui connettersi, porta alla quale connettersi, stringa da inviare per la conversione.
4. Scoprire l'indirizzo IP della propria interfaccia di rete usando il comando (da usare per collegarsi al proprio server)

```
$ ifconfig
```

5. Connettere il proprio client al server del vicino.
6. Notare l'ordine di chiusura di stream e socket alla fine dei due programmi. Che differenze ci sono e perché?
7. Modificare il codice del server e del client in modo che sia possibile inviare più stringhe da convertire. Il server aspetta una stringa dal client, la converte in maiuscolo e la rimanda al client. Il server continua ad aspettare una nuova stringa finché il client non chiude il socket (come fa ad accorgersene?). Il client permette di inviare più stringhe lette da standard input e chiude il socket quando l'utente digita la stringa "Stop".

## 2 Servizio di trasferimento file

Il client richiede al server un file, il cui nome viene specificato tramite la riga di comando, e il server risponde inviando al client il contenuto del file riga per riga. Gestire le eccezioni (file not found, etc.)

Client:

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

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

        // socket
        Socket clientSocket = null;

        // streams
        BufferedReader reader = null;
        PrintStream outputStream = null;

        if (args.length == 0) {
            System.out.println("Missing file name");
            System.exit(-1);
        }

        try {
            clientSocket = new Socket("localhost", 11111); // IP e porta del server
        } catch (IOException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }

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

        try{
            reader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
            outputStream = new PrintStream(new
BufferedOutputStream(clientSocket.getOutputStream()), true);
        }catch(IOException e){
            e.printStackTrace();
        }
    }
}
```

```

        // --- invio messaggio
        System.out.println("Invio richiesta per: " + args[0]);
        outputStream.println(args[0]);

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

        try {
            while ((line = reader.readLine()) != null) {
                System.out.println("Messaggio: " + line);
            }

            // chiusure
            clientSocket.close();
            outputStream.close();
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Server:

```

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

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

        // sockets
        ServerSocket serverSocket = null;
        Socket clientSocket = null;

        // streams
        BufferedReader reader = null ;
        PrintStream outputStream = null ;
        BufferedReader fileReader = null;

        try {
            System.err.println("Creazione ServerSocket");
            serverSocket = new ServerSocket(11111);
        } catch (Exception e){
            System.out.println("Impossibile creare ServerSocket");
            e.printStackTrace();
            System.exit(-1);
        }

        System.out.println("Attesa connessione...");
        try {
            clientSocket = serverSocket.accept();
        } catch (IOException e) {
            System.out.println("Connessione fallita");
            e.printStackTrace();
            System.exit(2);
        }

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

        try {
            reader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
            BufferedOutputStream outBuffer = new
BufferedOutputStream(clientSocket.getOutputStream());
            outputStream = new PrintStream(outBuffer, true);

            // --- ricezione nome file dal client
            String fileName = new String(reader.readLine());

```

```

        System.out.println("File richiesto dal client: " + fileName);

        // --- invio del file al client
        fileReader = new BufferedReader(new InputStreamReader(new
FileInputStream(fileName)));

        String tmp = null;
        while ((tmp = fileReader.readLine()) != null) {
            outStream.println(tmp);
        }
        // chiusure
        try{
            serverSocket.close();
            clientSocket.close();
            reader.close();
            outStream.close();
            fileReader.close();
        }catch(IOException e){
            e.printStackTrace();
        }
    } catch (FileNotFoundException e) {
        System.out.println("File non trovato");
        outStream.println("File non trovato");
    } catch (Exception e) {
        System.out.println(e);
    }
}
}

```

## Esercizio

1. Modificare client e server in modo che prendano da riga di comando IP/porta e porta, rispettivamente.
2. Modificare il client in modo che il contenuto del file ricevuto non venga visualizzato a video ma venga salvato su un file di testo. Il nome file è lo stesso di quello richiesto se non viene passato come parametro via riga di comando.
3. Cosa succede se provate a richiedere al server il file `/etc/shadow` ? Perché?

## 3 Calcolatrice client / server

In questo esempio vediamo come sia possibile eseguire operazioni su un server remoto con l'obiettivo di non gravare il client. 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 CalcClient {
    public static void main(String args[]) {

        // socket
        Socket clientSocket = null;

        // streams
        BufferedReader reader = null;
        PrintStream outStream = null;

        try {
            clientSocket = new Socket("localhost", 11111); // IP e porta del server
            System.out.println("Socket creata: " + clientSocket);
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}

```



```

        System.exit(1);
    }

    try {
        reader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
        outputStream = new PrintStream(new
BufferedOutputStream(clientSocket.getOutputStream()), true);
    } catch (IOException e) {
        e.printStackTrace();
    }

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

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

    try {
        line = new String(reader.readLine());
        System.out.println("Msg dal server: " + line);

        // chiusure
        reader.close();
        outputStream.close();
        clientSocket.close();
    } catch (IOException e) {
        System.err.println(e.getMessage());
    }
}
}

```

```

// SERVER

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

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

        // sockets
        ServerSocket serverSocket = null;
        Socket clientSocket = null;

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

        System.out.println("Attesa connessione...");

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

        System.out.println("Connessione da " + serverSocket);
    }
}

```

```

        try {
            BufferedReader reader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
            BufferedOutputStream outBuffer = new
BufferedOutputStream(clientSocket.getOutputStream());
            PrintStream outStream = new PrintStream(outBuffer, true);

            String line;
            int y, x = 0;

            do {
                line = new String(reader.readLine());
                y = Integer.parseInt(line);
                System.out.println("Value: " + y);
                x += y;
            } while (y != 0);

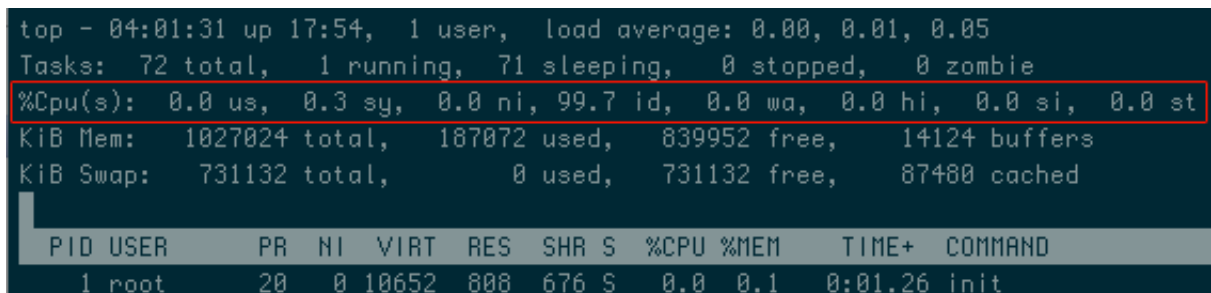
            outStream.println("Somma = " + x);

            // chiusura
            outStream.close();
            reader.close();
            clientSocket.close();
            serverSocket.close();
        } catch (Exception e) {
            System.out.println("Errore: " + e);
            System.exit(3);
        }
    }
}

```

## Esercizi

1. Modificare client e server in modo che prendano da riga di comando IP/porta e porta, rispettivamente.
2. Creare una versione del server che calcoli i primi N numeri primi e una versione del client che interroghi il server chiedendo i primi N numero primi.
3. Aprite una shell e lanciate il comando `top` per visualizzare la lista dei processi e l'uso del processore. Provate a verificare il carico di lavoro della vostra CPU a seconda che abbiate client e server sulla stessa macchina e client e server su due macchine separate. Cosa si osserva? Perché?



```

top - 04:01:31 up 17:54, 1 user, load average: 0.00, 0.01, 0.05
Tasks: 72 total, 1 running, 71 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.0 us, 0.3 sy, 0.0 ni, 99.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 1027024 total, 187072 used, 839952 free, 14124 buffers
KiB Swap: 731132 total, 0 used, 731132 free, 87480 cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU  %MEM    TIME+  COMMAND
    1 root        20   0 10652   808  676 S   0.0   0.1   0:01.26 init

```

*Illustration 1: Esempio di visualizzazione del comando top*

## 4 Java WEB Server

Esercizio: con le conoscenze sui socket acquisite, create un piccolo WEB server in Java che sia possibile interrogare da browser. Il server dovrà essere in ascolto sulla porta 80 e, una volta ricevuta

una connessione da parte del client, dovrà leggere la richiesta del client, stamparla a video, spedire un header HTTP + il contenuto di una pagina HTML e chiudere il socket.

Un esempio di header HTTP è:

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: My Java Web Server
```

**Attenzione** all'ultima riga vuota, che serve da delimitatore tra header HTTP e pagina HTML della risposta del server.

Una volta che il server è in ascolto, usate il browser per navigare sull'indirizzo localhost per visualizzare la risposta del server.

## 5 Chat client/server

In questo esempio andremo ad implementare una chat testuale. La principale novità introdotta in questo esempio sarà l'uso delle thread con l'obiettivo di introdurre interattività.

Per testare il server utilizzate il programma di sistema TELNET in questo modo:

```
telnet localhost 1025
```

Quando Telnet sarà connesso al server, ogni volta che scriverete qualcosa e darete invio, lo vedrete apparire sul server e replicato sul client. Per chiudere la sessione in Telnet occorre digitare il comando *close*, oppure *quit* oppure CTRL-C.

Server

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

class ChatServer {
    public static void main(String args[]) throws Exception {
        ServerSocket conn = new ServerSocket(1025);
        new Server(conn.accept()).run();
    }
}

class Server {
    Socket socket;

    public Server(Socket s) {
        this.socket = s;
    }

    public void run() {
        String from;
        BufferedReader reader = null;
        PrintStream outputStream = null;

        try {
            reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            outputStream = new PrintStream(socket.getOutputStream());

            System.out.println("Connected");
            while ((from = reader.readLine()) != null && !from.equals("")) {
                System.out.println(from);
                outputStream.print(from + "\r\n");
            }
            socket.close();
        } catch (IOException e) {
```

```
        }  
        System.out.println("Disconnected");  
    }  
}
```

## Esercizi

1. Un problema di questo server è che termina non appena il client si disconnette, invece di aspettare una nuova connessione. Cosa succede se provate a connettere due client Telnet contemporaneamente? NOTA: dovete usare due shell diverse.
2. Modificare il server in modo che, una volta terminata una connessione con un client, resti in ascolto di altre connessioni. Cosa succede adesso se provate a connettere due client Telnet contemporaneamente?
3. Modificare il server in modo che accetti connessioni simultanee da più client. Ciascun client deve rimanere connesso e funzionare indipendentemente dagli altri.
4. Ancora non abbiamo implementato una vera chat, infatti i client si connettono al server ma non sono in grado di comunicare tra di loro. Come possiamo fare in modo che i messaggi ricevuti dal server vengano inviati alla lista di tutti i client che sono connessi?
5. Cosa succede col server del punto precedente quando un client qualsiasi di quelli connessi si disconnette? Come fare perché il server continui a propagare sui client rimanenti i messaggi ricevuti?
6. Implementare un client in java che si colleghi al server e che invii e riceva i messaggi. Vogliamo poter continuare ad inviare messaggi senza dover bloccare il client fino a quanto non riceve un messaggio di risposta. L'utilizzo delle thread consente di gestire in parallelo invio e ricezione di messaggi senza bloccare il client. Il client dovrà implementare una thread che riceve i messaggi del server e una interazione con l'utente che scrive i messaggi.
7. ...e se volessimo implementare anche la gestione dei nickname?

# UDP vs TCP

User Datagram Protocol (UDP) e Transmission Control Protocol (TCP) sono protocolli di livello trasporto a pacchetto. La differenza tra UDP e TCP è che UDP è un protocollo di tipo connectionless (ovvero senza connessione), non gestisce il riordinamento dei pacchetti né la ritrasmissione di quelli persi è per ciò considerato meno affidabile del TCP.

Il TCP, invece, si occupa di “controllo di trasmissione” ovvero rende affidabile la comunicazione, fornendo un servizio di ritrasmissione di pacchetti persi o corrotti.

## 1 UDP echo server UPPERCASE

In questa esercitazione implementeremo una versione dell'echo server tramite protocollo UDP per poi confrontare cosa succede a livello di rete tra la versione TCP e la versione UDP. Per questa esercitazione occorre utilizzare la virtual machine Linux che vi è stata messa a disposizione in laboratorio. Il codice Java può essere scritto e modificato dal PC host (quello vero) usando il solito editor di preferenza. La compilazione si può effettuare sul PC host. **Esecuzione di programmi Java, Wireshark e IPTABLE deve avvenire solo sul PC virtuale quando si è in laboratorio o sul proprio PC di proprietà.**

### Documentazione:

DatagramSocket (<http://docs.oracle.com/javase/7/docs/api/java/net/DatagramSocket.html>)

DatagramPacket (<http://docs.oracle.com/javase/7/docs/api/java/net/DatagramPacket.html>)

InetAddress (<http://docs.oracle.com/javase/7/docs/api/java/net/InetAddress.html>)

```
// CLIENT

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

class UDPCClient {
    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];
            System.out.println("Inserisci il messaggio per il server\n");
            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];
            System.out.print("Attesa connessione...");
            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) {
            System.err.println("Impossibile avviare il server.");
        }
    }
}

```

## Esercizio 4

1. A questo punto vediamo la differenza tra TCP e UDP:
  1. avviate Wireshark (mediante il comando da shell `sudo wireshark`) e fate partire una cattura sull'interfaccia di loopback;
  2. Per filtrare la versione TCP applica il filtro `tcp.port == 11111`;
  3. Avviate il server e poi avviate il client per vedere cosa succede a livello di rete.

Per catturare la comunicazione UDP seguite la stessa procedura cambiando il filtro in `udp.port == 9876`

Dalla comunicazione TCP vediamo che i messaggi scambiati sono stati >>2 (quanti?) mentre dalla comunicazione UDP vediamo che i messaggi scambiati sono stati solo 2.

Il protocollo TCP è famoso per fornire affidabilità della comunicazione implementando un ritrasmissione automatica basato su acknowledge per segnalare quando un pacchetto è stato ricevuto. Come mostra l'esempio dell'echo server UPPERCASE, la gestione della ritrasmissione dei pacchetti comporta un costo in termini di pacchetti scambiati sulla rete non indifferente. La versione TCP ha trasmesso un totale di >>2 pacchetti mentre la versione UDP ha trasmesso solo 2 pacchetti. Il protocollo UDP è da preferire al protocollo TCP ogniqualvolta non è essenziale avere un sistema di ritrasmissione. Usare il TCP per l'esempio dell'echo server UPPERCASE è uno spreco di risorse. Il fatto stesso che il client riceva (o non riceva) una risposta con il testo in UPPERCASE è un acknowledge implicito.

## Esercizio 5

1. Provate ad usare il servizio di sistema IPTABLES per bloccare i pacchetti in ricezione sulla porta 11111 del server TCP e vedere cosa succede a livello di rete sempre con l'ausilio di Wireshark.
2. Provate a bloccare il server, mandare una richiesta al server e poi sbloccarlo dopo un brevissimo tempo di attesa (eventualmente provare con diversi tempi di attesa).

Iptables è un firewall per Linux che consente di specificare regole per il traffico di rete. Per bloccare la ricezione sulla porta 11111 del server eseguire il comando:

```
sudo iptables -A INPUT -p tcp --dport 11111 -j DROP
```

Per riabilitare la ricezione di pacchetti eseguire il comando:

```
sudo iptables -D INPUT -p tcp --dport 11111 -j DROP
```

## Esercizio 6

1. Implementate una variante dell'esempio della Calcolatrice TCP vista nelle lezioni precedenti utilizzando il protocollo UDP (chiamiamola Calcolatrice UDP).
2. Provare a bloccare la ricezione e a sbloccarla subito dopo usando IPTABLES in modo da perdere un addendo. Cosa si può notare?

Per bloccare la ricezione sulla porta 9876 del server UDP eseguire il comando:

```
sudo iptables -A INPUT -p udp --dport 9876 -j DROP
```

Per riabilitare la ricezione di pacchetti eseguire il comando:

```
sudo iptables -D INPUT -p udp --dport 9876 -j DROP
```

3. Implementare una nuova variante della Calcolatrice UDP con la gestione del rinvio di pacchetti. Ogni volta che client o server inviano un pacchetto, dovranno aspettare di ricevere un acknowledge come in TCP. Se entro un certo tempo l'acknowledge non viene ricevuto, bisogna ritrasmettere il dato (chiamiamo questa versione Calcolatrice UDP+ACK).

NOTA: Per implementare il timeout in ricezione su un `DatagramSocket` utilizzate `setSoTimeout` dentro un blocco `try ... catch (SocketTimeoutException)`. Per la documentazione fare riferimento a:

[http://docs.oracle.com/javase/7/docs/api/java/net/DatagramSocket.html#setSoTimeout\(int\)](http://docs.oracle.com/javase/7/docs/api/java/net/DatagramSocket.html#setSoTimeout(int))

4. Provare a bloccare la ricezione e a sbloccarla subito dopo usando IPTABLES in modo da

perdere un addendo. Cosa si può notare?

5. Confrontate i sorgenti di Calcolatrice UDP+ACK, Calcolatrice UDP e Calcolatrice TCP:
  - qual è il codice Java più complesso?
  - osservando con Wireshark, il traffico generato dal codice UDP+ACK è paragonabile a quello generato dalla versione TCP?
6. Ha senso scrivere questa applicazione in TCP? I pacchetti ACK non sono necessari dato il tipo di applicazione, perché? Qual è la versione ottimale di codice Java per questa applicazione tra UDP, UDP+ACK e TCP ?

## Socket in C

Adesso vedremo brevemente la programmazione via socket in C. Andremo ad implementare un server sempre in esecuzione e che, appena un client si connette, invia data e ora.

### Documentazione:

socket() (<http://man7.org/linux/man-pages/man2/socket.2.html>)

bind() (<http://unixhelp.ed.ac.uk/CGI/man-cgi?bind+2>)

struct sockaddr\_in ([http://www.gta.ufjf.br/ensino/eel878/sockets/sockaddr\\_inman.html](http://www.gta.ufjf.br/ensino/eel878/sockets/sockaddr_inman.html))

connect() (<http://linux.die.net/man/2/connect>)

### Server:

- La chiamata alla funzione socket() crea un socket nel kernel e restituisce un intero chiamato descrittore del socket. A questo passiamo come argomento AF\_INET per indicare che vogliamo utilizzare il protocollo IPv4. Il secondo parametro SOCK\_STREAM dice che vogliamo che il canale sia affidabile e che quindi implementi l'invio di ack. Il terzo parametro è generalmente lasciato a 0 per indicare al kernel si scegliere il protocollo (TCP in questo caso) (riferirsi alla documentazione per ulteriori dettagli)
- La struttura sockaddr\_in contiene le informazioni IP porta e dettagli della connessione
- La funzione bind() assegna i dettagli della struttura sockaddr\_in al socket precedentemente creato.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>

// SERVER
int main(int argc, char *argv[])
{
    int listenfd = 0, connfd = 0;
    struct sockaddr_in serv_addr;
```



```

char sendBuff[1025];
time_t ticks;

listenfd = socket(AF_INET, SOCK_STREAM, 0);
memset(&serv_addr, '0', sizeof(serv_addr));
memset(sendBuff, '0', sizeof(sendBuff));

serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(5000);

bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));

listen(listenfd, 10);

while(1)
{
    connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);

    ticks = time(NULL);
    snprintf(sendBuff, sizeof(sendBuff), "%.24s\r\n", ctime(&ticks));
    write(connfd, sendBuff, strlen(sendBuff));

    close(connfd);
    sleep(1);
}

```

#### Client:

- Anche qui viene creato un socket tramite la chiamata alla funzione socket() e viene utilizzata la struttura sockaddr\_in per specificare i dettagli del server a cui connettersi.
- A questo punto viene effettuata la connessione tramite la chiamata alla funzione connect() che tenta di connettere il socket creato, con il socket remoto del server.
- Una volta che il socket è connesso, il client riceve le informazioni e le visualizza a video, e poi termina.

```

#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <arpa/inet.h>

//CLIENT
int main(int argc, char *argv[])
{
    int sockfd = 0, n = 0;
    char recvBuff[1024];
    struct sockaddr_in serv_addr;

    if(argc != 2)
    {
        printf("\n Usage: %s <ip of server> \n", argv[0]);
        return 1;
    }
}

```

```

memset(recvBuff, '0', sizeof(recvBuff));
if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
    printf("\n Error : Could not create socket \n");
    return 1;
}

memset(&serv_addr, '0', sizeof(serv_addr));

serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(5000);

if(inet_pton(AF_INET, argv[1], &serv_addr.sin_addr)<=0)
{
    printf("\n inet_pton error occured\n");
    return 1;
}

if( connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
{
    printf("\n Error : Connect Failed \n");
    return 1;
}

while ( (n = read(sockfd, recvBuff, sizeof(recvBuff)-1)) > 0)
{
    recvBuff[n] = 0;
    if(fputs(recvBuff, stdout) == EOF)
    {
        printf("\n Error : Fputs error\n");
    }
}

if(n < 0)
{
    printf("\n Read error \n");
}

return 0;
}

```

# 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.

## 1 Vantaggi tecnologici

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

## 2 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.

# REST

In questa esercitazione vedremo come realizzare dei programmi Java che si interfaccino all'archivio genomico KEGG (<http://www.kegg.jp/>) allo scopo di aumentare l'automatizzazione nella ricerca biotecnologica.

Per fare ciò KEGG mette a disposizione delle API che, tramite metodologia REST, forniscono una serie di metodi per recuperare le informazioni che si trovano sul database attraverso programmi di interrogazione scritti dall'utente (ad es. noi useremo il linguaggio Java) in alternativa alla consultazione manuale tramite form web. Questa interfaccia al server KEGG è fondamentale per costruire programmi che automatizzano il recupero di informazioni evitando noiose e lente operazioni di interrogazione manuale.

## 1 Note tecniche

Un Web Service (servizio web), secondo la definizione data dal World Wide Web Consortium (W3C), è un sistema software progettato per **l'interazione tra diversi elaboratori in rete**; tale caratteristica si ottiene associando all'applicazione un'interfaccia software che espone all'esterno il servizio associato (**server**) e utilizzando la quale altri sistemi (**client**) possono interagire con l'applicazione stessa attivando le operazioni descritte nell'interfaccia (servizi o richieste di procedure remote) tramite appositi messaggi di richiesta; tali messaggi di richiesta possono avere un formato particolare (il più famoso è SOAP) e sono sempre incapsulati e trasportati tramite i protocolli del Web (solitamente HTTP), da cui deriva appunto il nome web service.

REST è un approccio alternativo a SOAP per la realizzazione di web services. SOAP si basa sul concetto di chiamata remota e utilizza una complessa infrastruttura di messaggi XML, stub/skeleton e application container. Al contrario, REST è molto più leggero perché si basa sulla corrispondenza diretta tra:

- oggetti remoti da leggere/scrivere → Uniform Resource Identifiers (URI) cioè i comuni link web
- comandi da applicare agli oggetti → comandi HTTP (GET, PUT, POST, DELETE)

Se ad esempio vogliamo leggere la lista dei prodotti presenti su un catalogo, verrà semplicemente invocato il metodo HTTP GET sulla URI che rappresenta il catalogo prodotti; siccome il metodo GET è quello usato per default dai browser web nell'accesso alle pagine, sarà quindi sufficiente aprire la pagina

<http://www.mionegozio.it/lista/prodotti>

Sarà stata ovviamente cura del progettista del server far corrispondere l'URI lista/prodotti all'opportuna query sul database.

Se il server lo supporta, la stessa risorsa può essere richiesta in formati diversi, come ad esempio TXT, HTML, XML o JSON.

REST assume che le interazioni tra client e server devono avvenire senza memorizzazione di informazioni di stato sul server. Cioè **le interazioni tra client e server devono essere senza stato (stateless)**. È importante sottolineare che sebbene REST assuma la comunicazione stateless, non vuol dire che un'applicazione non deve avere stato. Tuttavia la responsabilità della gestione dello stato dell'applicazione non deve essere conferita al server, ma rientra nei compiti del client.

## 2 Esercizi

NOTA: in laboratorio si consiglia di operare nella cartella /tmp/ in cui vi è tutto lo spazio necessario; tale cartella viene svuotata ad ogni avvio della macchina per cui sui propri PC personali se ne

sconsiglia l'uso.

Occorre creare una cartella di lavoro (ad esempio basta copiare la cartella `rest/` dell'archivio) all'interno della quale andremo a mettere i seguenti files:

- **Definition.java** (classe corrispondente alla definizione di organismo nel DB di KEGG)
- **RESTCall.java** (classe che incapsula le chiamate REST per il KEGG, utilizza gli oggetti `httpClient`)
- **httpClient-4.2.5.jar** (jar scaricabile dal sito apache foundation, serve per eseguire una richiesta HTTP)
- **httpcore-4.2.4.jar** (jar scaricabile dal sito apache foundation, serve per eseguire una richiesta HTTP)
- **commons-logging-1.1.1.jar** (libreria che fornisce un sistema evoluto di logging)

I file jar sono anche scaricabili da questo link: <http://hc.apache.org/downloads.cgi>

Infine occorre mettere nella cartella di lavoro tutti i programmi Java che si vogliono compilare ed eseguire. Al momento della compilazione e della esecuzione dei nostri programmi Java, dovremmo aver cura di comprendere i file di supporto elencati; per farlo dobbiamo includere nel classpath le librerie come spiegato di seguito.

Portarsi nella cartella dei sorgenti

### **Compilazione:**

```
javac -classpath ../httpClient-4.2.5.jar:../httpcore-4.2.4.jar:../commons-logging-1.1.1.jar *.java
```

### **Esecuzione:**

```
java -classpath ../httpClient-4.2.5.jar:../httpcore-4.2.4.jar:../commons-logging-1.1.1.jar classe [parametri]
```

Vediamo quindi subito un esempio:

```
public class KeggOrganismsList {
    public static void main(String[] args) {
        // creazione di un oggetto RESTCall che effettua le chiamate al KEGG
        RESTCall rest= new RESTCall();
        // richiediamo la lista degli organismi presenti in KEGG
        Definition[] def= rest.list_organisms();
        // tramite un ciclo for stampiamo la lista degli organismi
        for (int i = 0; i < def.length; i++) {
            System.out.println(i+" "+def[i].getDefinition());
        }
    }
}
```

Compilate ed eseguite il codice; come si evince dai commenti, questo semplice programma recupera la lista degli organismi contenuti nel database KEGG e ne stampa la descrizione a video.

Se si analizza l'implementazione del metodo `list_organisms()` in `RESTCall.java` si vede che la chiamata REST è <http://rest.kegg.jp/list/organism>

**Mini esercizio:** provare a copiare e incollare in un web browser (ad es. Firefox) la URI corrispondente alla chiamata REST e vedere cosa succede.

**Definition** è una classe che contiene informazioni su un organismo; di conseguenza un array di **Definition** contiene un organismo per ogni valore dell'indice. La descrizione della classe **Definition** è in Appendice C; si chiede di analizzarne i metodi.

### Esercizio 1

Si voglia verificare la presenza di un certo organismo all'interno del database e stamparne anche l'**Entry\_id** oltre che al nome (vedere metodo corrispondente). Modificate quindi il codice in modo da ricercare un particolare organismo passato come input sulla riga di comando.

*Suggerimento:* ricavare la stringa con `def[i].getDefinition()` e su questa chiamare il metodo `indexOf(args[0])`.

Si noti che passando sulla linea di comando un nome di organismo senza delimitarlo da doppi apici solo la prima stringa viene messa in `args[0]` e quindi possono essere associati più risultati che contengono tale stringa.

Vediamo ora un altro esempio:

```
public class GetGenesByEnzyme {
    public static void main(String[] args) {
        // come sopra
        RESTCall serv= new RESTCall();
        String[] result;
        // recuperiamo il numero di geni dato l'organismo
        result = serv.get_genes_by_enzyme("ec:2.7.1.6", "ljo");
        System.out.println(result[0]);
    }
}
```

Il metodo `get_genes_by_enzyme()` accetta come parametro due stringhe, una corrispondente all'enzima e l'altra corrispondente all'id dell'organismo. Se si va ad analizzarne l'implementazione si vede che la chiamata REST corrispondente è <http://rest.kegg.jp/link/ljo/ec:2.7.1.6>

I due sorgenti appena visti non avrebbero molto senso se eseguiti da soli, ma componendo il codice del primo con il codice del secondo si incomincia a vedere l'utilità di applicazioni che automatizzano la ricerca.

### Esercizio 2

Provate quindi a scrivere un programma che dato il nome di un organismo ne ricavi il suo id e dall'id ricavi il gene associato all'enzima `ec:2.7.1.6` (la spiegazione della scrittura `ec:2.7.1.6` si trova più avanti nella dispensa).

Provatelo con input: "Lactobacillus johnsonii NCC 533", il vostro codice dovrebbe restituire:

`ljo:LJ0859` (dove `ljo` è l'organismo e `LJ0859` il gene, questa scrittura sarà spiegata più avanti nella dispensa). Si noti che passando sulla linea di comando un nome di organismo senza delimitarlo da doppi apici solo la prima stringa viene messa in `args[0]` e quindi possono essere associati più risultati che contengono tale stringa. Questo caso deve essere rilevato (quanti parametri vedo sulla linea di comando?) e bisogna avvisare l'utente di delimitare il nome dell'organismo con doppi apici.

Passiamo ora a vedere una struttura diversa chiamata **DBGET**. Si tratta di un metodo di recupero delle informazioni formattate come file di testo, detti anche flat-file. La definizione di flat-file di KEGG non è limitata ai soli file di testo ma si estende a immagini GIF (per i pathway di KEGG), grafica 3D per la

struttura di proteine, ecc.

La maggior parte degli attuali database biologici possono essere utilizzati in questa specifica.

Quando si passa un input ad un metodo di DBGET bisogna seguire un certo pattern:

**dbname:identifier**

ovvero nome del database (abbreviato) seguito dall'identificatore di ciò che ci interessa.

(per una lista completa dei database si rimanda a: <http://www.genome.jp/dbget/>)

Il catalogo dei geni in KEGG considera anche come identificatore la combinazione organismo e gene:

**organismo:gene**

Vediamo subito un esempio di codice e cosa ci restituisce:

```
public class KeggEsDBGET {
    public static void main(String[] args) {
        RESTCall serv= new RESTCall();
        String result = serv.bget(args[0]);
        System.out.println(result);
    }
}
```

Provate a compilare ed eseguire questo codice con input “eco:b0004” dove in questo caso “eco” è l'organismo Escherichia coli e “b0004” è l'id del gene. Se si va ad analizzarne l'implementazione si vede che la chiamata REST corrispondente è <http://rest.kegg.jp/get/eco:b0004>

Il metodo bget() restituisce una stringa che contiene tutte le informazioni relative all'argomento della ricerca.

|            |   |     |        |
|------------|---|-----|--------|
| ENTRY      | b0004   | CDS | E.coli |
| NAME       | thrC, ECK0004, JW0003   |     |        |
| DEFINITION | threonine synthase (EC:4.2.3.1)   |     |        |
| ORTHOLOGY  | K01733 threonine synthase [EC:4.2.3.1]  |     |        |
| PATHWAY    | eco00260 Glycine, serine and threonine metabolism   |     |        |
|            | eco00750 Vitamin B6 metabolism  |     |        |
|            | eco01100 Metabolic pathways   |     |        |
| CLASS      | Metabolism; Amino Acid Metabolism; Glycine, serine and threonine metabolism [PATH:eco00260] |     |        |
|            | Metabolism; Metabolism of Cofactors and Vitamins; Vitamin B6 metabolism [PATH:eco00750]     |     |        |
| POSITION   | 3734..5020  |     |        |
| MOTIF      | Pfam: PALP  |     |        |
|            | PROSITE: DEHYDRATASE_SER_THR  |     |        |
| DBLINKS    | NCBI-GI: 16127998   |     |        |
|            | NCBI-GeneID: 945198   |     |        |
|            | RegulonDB: B0004  |     |        |
|            | EcoGene: EG11000  |     |        |
|            | UniProt: P00934   |     |        |
| STRUCTURE  | PDB: 1VB3   |     |        |
| AASEQ      | 428   |     |        |
|            | MKLYNLKDHNEQVSFAQAVTQGLGKNQGLFFPHDLPEFSLTEIDEMLKLDVTRSAKILS                                 |     |        |
|            | AFIGDEIPQEILEERVRAAFAPVANVESDVGCLELFHGPTLAFKDFGGRFMAQMLTH                                   |     |        |
|            | IAGDKPVTILTATSGDTGA AVAHAFYGLPNVKVVI LYPRGKISPLQEKL FCTLGNIETV                              |     |        |
|            | AIDGDFDACQALVKQAFDDEELKVALGLNSANSINISRLLAQICYFEAVAQLPQETRQ                                  |     |        |
|            | LVVSVPSGNFGDLTAGLLAKSLGLPVKRFIAATNVNDTVPRFLHDGQWSPKATQATLSNA                                |     |        |
|            | MDVSQPNNWPRVEELFRRKIWQLKELGYAAVDDETTQQTRELKELGYTSEPHAAYAYRA                                 |     |        |
|            | LRDQLNPGEYGLFLGTAHPAKFKESVEAILGETLDLPKELAERADLPLLSHNLPA DFAAL                               |     |        |



```

NTSEQ      RKLMNHQ
1287
atgaaactctacaatctgaaagatcacacgagcaggtcagctttgcgcaagccgtaacc
caggggttgggcaaaaatcaggggtgttttttccgcacgacctgccggaattcagcctg
actgaaattgatgagatgctgaagctggattttgtcaccgcagtgccaagatcctctcg
gcgtttattggtgatgaaatcccacaggaaatcctggaagagcgcgtgcgcgcggtgtt
gccttcccgggtccgggtcgccaatgttgaaagcgatgtcggttgcttggaattgttccac
gggccaacgctggcatttaagatttcggcggtcgctttatggcacaatgctgacccat
attgcggtgataagccagtgaccattctgaccgcgacctccggtgataccggagcggca
gtggctcatgctttctacgggtttaccgaatgtgaaagtgggtatcctctatccacgaggc
aaaatcagtcactgcaagaaaaactgttctgtacattggcggaatatcgaaactgtt
gccatcgacggcgatttcgatgcctgtcaggcgctgggtgaagcaggcggttgatgatgaa
gaactgaaagtggcgctagggtaaaactcggctaactcgattaacatcagccgtttgctg
gcgcagatttgctactactttgaagctgttgcgagctgccgcaggagacgcgcaaccag
ctggttgctcggtgccaagcggaaacttcggcgatttgacggcggtgtgctggcgaag
tcactcgggtcggcggtgaaacgttttattgctgacgaacgtgaacgataccgtgcca
cgtttctgcacgacggtcagtggtcacccaaagcgactcaggcgacgttatccaacgcg
atggacgtgagtcagccgaacaactggccgcgtgtggaagagttgttccgcgcgaaaatc
tggaactgaaagagctgggttatgcagccgtggatgatgaaaccacgcaacagacaatg
cgtgagttaaaagaactgggtacacttcggagccgcacgctgccgtagcttatcgtgcg
ctgctgatcagttgaatccaggcgaatatggcttggttctcggcaccgcgcacccggcg
aaatttaaagagagcgtggaagcgattctcggtgaaacgttggtatcgccaaaagagctg
gcagaacgtgctgatttacccttgctttcacataatctgcccgcgattttgctgcttg
cgtaaattgatgatgaatcatcagtaa
///

```

Chiaramente questa rappresentazione dei dati contiene un po' troppe informazioni e nessuna utilizzabile per qualche elaborazione successiva, è quindi necessario suddividere questa grossa stringa “ritagliando” solo le parti che ci interessano.

### Esercizio 3

Recuperare il valore di POSITION.

**SUGGERIMENTO:** utilizzate il metodo `indexOf()` per recuperare l'indice della parola POSITION e la posizione del primo punto dopo la parola POSITION con il metodo `indexOf(int ch, int fromIndex)`, poi recuperate la sottostringa con il metodo `substring(start_index+12, end_index)`.

### Esercizio finale

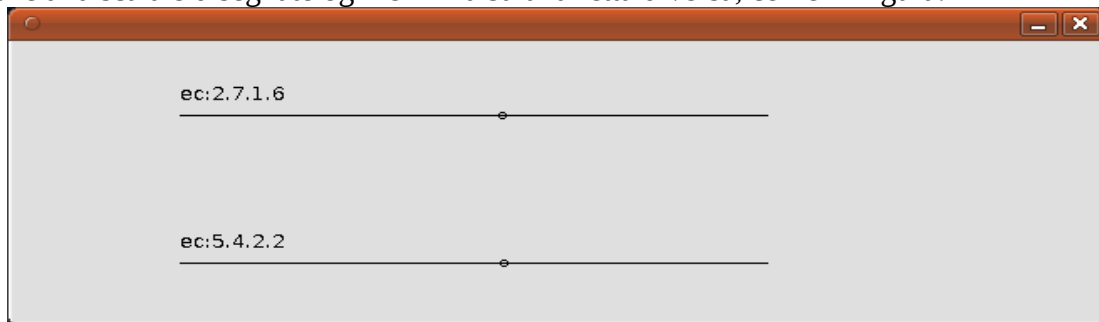
Scrivere un programma che:

- 1) Riceva il nome di un organismo come primo parametro sulla linea di comando e ne recuperi l'ID.  
**ATTENZIONE:** abbiamo già visto un possibile codice che lo fa, ma ad una voce inserita dall'utente possono essere associati più risultati, sarà quindi necessario gestire questa evenienza (anche semplicemente prendendo il primo risultato).
- 2) Dato l'ID dell'organismo e una serie di enzimi, restituisca la posizione dei geni corrispondenti. Provate con i seguenti enzimi:
  - ec:2.7.1.6
  - ec:5.4.2.2**SUGGERIMENTO:** usate il metodo `get_genes_by_enzyme()` per recuperare il gene e passate ciò che viene restituito al metodo `bget()` già visto precedentemente. Notare che la stringa ottenuta va convertita in numero.
- 3) Recuperi la lunghezza del genoma:  
**SUGGERIMENTO:** usate sempre `bget()` con la stringa “gn:”+id\_organismo (vedere in Appendice B un esempio di output) e recuperate il numero che segue la stringa “LENGTH”

estraendo la sotto-stringa da LENTGH+10 alla stringa “\n”. Notare che la stringa ottenuta va convertita in numero.

- 4) Tramite l'oggetto Canvas disegni una retta che identifica il genoma e un pallino che identifica la posizione degli enzimi sul genoma (vedere Appendice A).

Attenzione alla scala e disegnate ogni enzima su una retta diversa, come in figura:



Provate usando come input “Lactobacillus johnsonii NCC 533” tra doppi apici. Se non usate i doppi apici o passate solo “johnsonii”, il vostro programma dovrebbe farvi scegliere attraverso un menu composto da indice e nome dell’organismo, tra altri organismi che contendono quella parola.

NOTA: il programma è solo un esempio prototipale e funziona bene solo con “Lactobacillus johnsonii NCC 533”; altri organismi potrebbero non contenere gli enzimi specificati oppure generare l'informazione della posizione con una sintassi diversa da quella vista e parserizzata negli esercizi.

# SOAP

Vediamo ora un altro protocollo 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.

## 1 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.

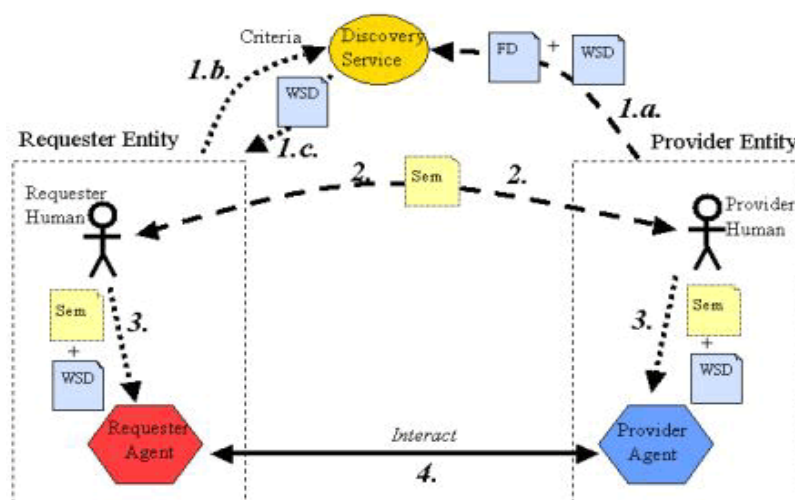


Figura 1. Ciclo di vita di un Web Service.

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.

## 2 Architettura di un sistema che usa SOAP

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.

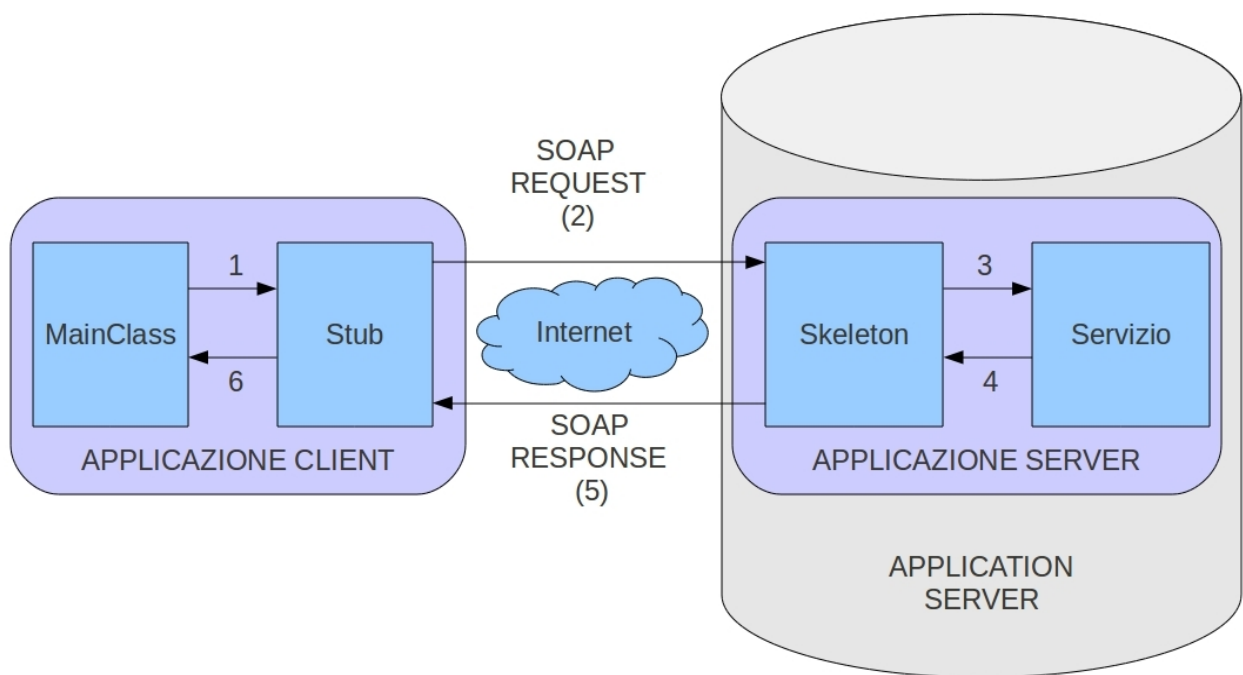


Figura 2: architettura di un webservice basato su SOAP e dettaglio di comunicazione tra client e server.

### 3 Creazione di un web service

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.

Vediamo passo per passo come procedere.

#### 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.

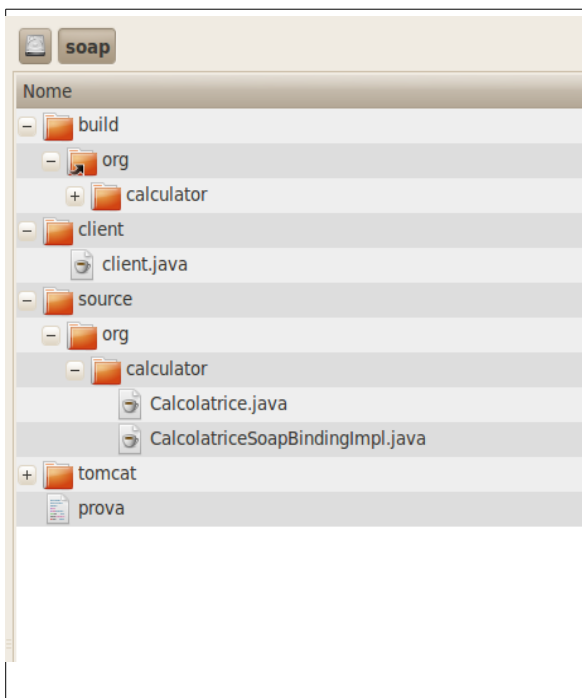
|  | CARTELLA              | CONTENUTO   |
|--|-----------------------|---|
|  | build/org             | link simbolico alla cartella /tomcat/webapps/axis/WEB-INF/classes/org, dove sono contenuti i file necessari per eseguire l'applicazione lato server |
|  | client                | Sorgente del client   |
|  | 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/).

#### 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 l'application container usare il comando:

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

## 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).

### L'interfaccia Calcolatrice.java

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

### La classe CalcolatriceSoapBindingImpl.java

```
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

```

## 4 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

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 Creazione dello stub e dello skeleton

Questi due elementi sono indispensabili per permettere al servizio di funzionare<sup>1</sup>. 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

```

---

<sup>1</sup> 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

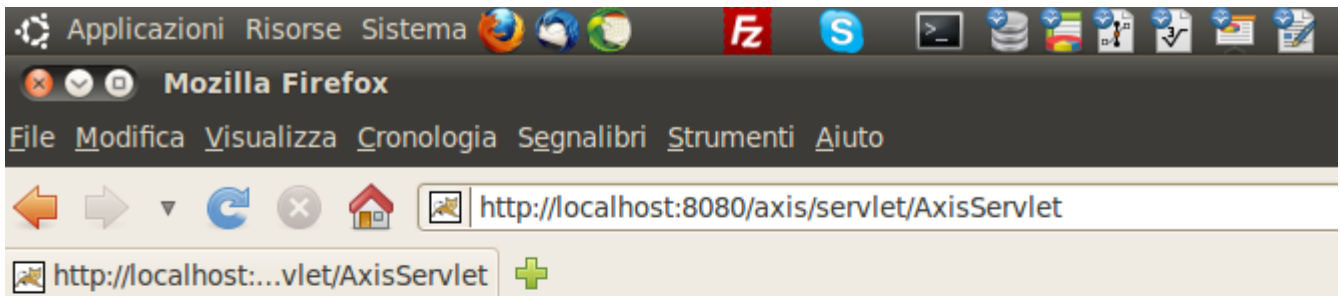
```
javac build/org/calculator/*.java
```

### Installazione del Web Service nell'application container

Possiamo ora procedere all'installazione del nostro web service all'interno di AXIS:

```
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

## Creazione del client

L'insieme di file che abbiamo creato nel passo precedente, 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
```

## 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));
    }
}

```

## 6 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

1. 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:** in questa fase deve essere cambiata l'URL su cui si trova il servizio (in quale comando di shell era stata impostata?). Di conseguenza anche il codice dello stub deve essere rigenerato e ricompilato.
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.

# Appendice A

## Java e le primitive grafiche

Introduciamo brevemente le primitive che Java mette a disposizione del programmatore per la gestione della grafica elementare. Queste primitive consentono da un lato di implementare GUI (Graphical User Interface) interattive con l'utente, dall'altro di giocare con i costrutti geometrici fondamentali e realizzare ogni sorta di disegno e rappresentazione grafica. Vedremo dunque come implementare a livello applicativo la visualizzazione di un semplice grafico per la rappresentazione di figure geometriche elementari, al fine di visualizzare semplici rappresentazioni grafiche dei risultati.

### Panoramica sul package AWT

Di seguito faremo riferimento all'Abstract Window Toolkit, la libreria contenente le classi e le interfacce fondamentali per la creazione di elementi grafici, inserita nelle API standard di Java.

I package fondamentali in cui si articola sono i seguenti:

- `java.awt` package principale che contiene le classi di tutti i componenti che possiamo utilizzare nella GUI, quali ad esempio `Frame`, `Panel`, `Button`, `Label`, `Checkbox`, `Canvas`, `Menu`, `MenuBar`, `TextArea` e molti altri
- `java.awt.event` fornisce le classi per la gestione degli “eventi”, ovvero il sistema che awt utilizza per passare il controllo al programmatore in seguito ad azioni avvenute sui componenti, come la pressione di un bottone, il passaggio del mouse su un componente, l'apertura di una finestra

La classe che rappresenta una “finestra” di interazione grafica con l'utente è la `java.awt.Frame`. Essa presenta fondamentalmente la classica barra del titolo e un bordo.

Ci concentreremo sull'uso di oggetti `Canvas`. `Canvas` è un componente che rappresenta un rettangolo vuoto dello schermo all'interno del quale è possibile disegnare oggetti geometrici elementari.

Vediamo subito un esempio per spiegare come utilizzare questo componente.

```
import java.awt.*;
import java.awt.event.*;

public class DrawExample extends Canvas {

    public void paint(Graphics g){
        g.drawRect(10, 50, 400, 1);
    }

    public static void main (String args[]){
        Canvas frame = new DrawExample();
        Frame finestra = new Frame();
        finestra.add(frame);
        finestra.setSize(450, 90);
        finestra.setResizable(false);
        finestra.setVisible(true);
    }
}
```

Come vedete la classe `DrawExample` estende `Canvas`, infatti per poter disegnare qualcosa sul canvas è necessario ridefinire il metodo `paint()`. L'oggetto passato come parametro al metodo `paint()` è un oggetto `Graphics`, che definisce un contesto grafico con il quale disegnare sul `Canvas`.

`Graphics` mette a disposizione molti metodo per disegnare delle figure geometriche elementari quali rettangoli, cerchi, rette ecc. Nell'esempio si utilizza il metodo `drawRect(int x, int y, int width, int`

height) i cui argomenti sono abbastanza autoesplicativi.

Analizziamo ora il main del progetto:

- Come prima cosa si crea un oggetto della classe che estende Canvas e quindi definisce cosa bisogna disegnare;
- L'oggetto Canvas appena creato deve essere messo in un Frame per poterlo visualizzare a video e quindi creiamo un oggetto di tipo Frame;
- Una volta creato il Frame basta impostare le dimensioni, togliere il resize della finestra e settare la visibilità.

Come già detto Graphics mette a disposizione un metodo per quasi tutte le forme geometriche elementari, si rimanda quindi alla Javadoc di Graphics per una lista completa:

(Graphics: <http://java.sun.com/j2se/1.4.2/docs/api/java/awt/Graphics.html>)

## Appendice B

### Esempio di output della chiamata bget() con argomento “gn:”+id\_organismo

```
ENTRY      T00158          Complete Genome
NAME       ljo, L.johnsonii, LACJ0, 257314
DEFINITION Lactobacillus johnsonii NCC 533
ANNOTATION manual
TAXONOMY   TAX:257314
  LINEAGE  Bacteria; Firmicutes; Lactobacillales; Lactobacillaceae;
           Lactobacillus
DATA_SOURCE RefSeq
ORIGINAL_DB Nestle
CHROMOSOME Circular
  SEQUENCE RS:NC_005362
  LENGTH   1992676
STATISTICS Number of nucleotides:      1992676
           Number of protein genes:    1821
           Number of RNA genes:        97
REFERENCE  PMID:14966310
  AUTHORS  Pridmore RD, et al.
  TITLE    The genome sequence of the probiotic intestinal bacterium
           Lactobacillus johnsonii NCC 533.
  JOURNAL  Proc Natl Acad Sci U S A : (2004)
///
```

# Appendice C

## Descrizione della classe Definition

Definition è una classe che contiene informazioni su un organismo; di conseguenza un array di Definition contiene un organismo per ogni valore.

```
public class Definition {
    private String definition=null;
    private String Entry_id;
    private String org;

    public String getDefinition() {
        return definition;
    }
    public void setDefinition(String definition) {
        this.definition = definition;
    }
    public String getEntry_id() {
        return Entry_id;
    }
    public void setEntry_id(String entry_id) {
        Entry_id = entry_id;
    }
    public String getOrg() {
        return org;
    }
    public void setOrg(String org) {
        this.org = org;
    }
}
```