java.net.*

Basic Network Tutorial

aprile 2005

Stefano Sanna - gerda@crs4.it

Agenda

- Java e network programming
- Low-level programming, network programming e distributed computing
- UDP e TCP
- Client e Server
- Applicazioni
- Bibliografia
- Conclusioni

Java

Protocol stack

Applicazione (HTTP, FTP...)	↔	Applicazione (HTTP, FTP...)
Trasporto (UDP, TCP, RTP)	↔	Trasporto (UDP, TCP, RTP)
Rete (IP)	↔	Rete (IP)
Hardware	↔	Hardware

Protocolli di rete

- E' importante conoscere i diversi protocolli a disposizione: caratteristiche e finalità, i limiti, le potenzialità, l'affidabilità, la robustezza, l'overhead introdotto, la sicurezza, i requisiti di memoria e calcolo
- La scelta del protocollo giusto deve essere fatta in base a:
  - banda disponibile
  - affidabilità della rete (wired/wireless)
  - volume dei dati scambiati
  - ...

Tre livelli di sviluppo

- L'evoluzione dei protocolli e delle infrastrutture di rete ha portato “idealmente” a tre livelli distinti di programmazione e sviluppo network-oriented:
  - Low-level Network Programming
  - Network Programming
  - Distributed Computing

Low-level network programming

- Per low-level network programming intendiamo la possibilità di scambiare dati primitivi e complessi attraverso stream su trasporto TCP e UDP.
- Attraverso le classi Socket e ServerSocket è possibile gestire la comunicazione client/server per l'invio di byte[], tipi base, oggetti serializzati
- La logica di comunicazione è responsabilità del programmatore che dovrà prevedere opportuni codici di controllo

Network Programming

- Per network programming intendiamo l'utilizzo di uno o più protocolli applicativi, che si occupano di gestire la logica di comunicazione tra client e server
- A seconda del protocollo scelto, il programmatore dispone di strumenti più o meno complessi che nascondono i dettagli implementativi della specifica
- Esempi di protocolli applicativi: Telnet, FTP, HTTP, SSH, NFS, NTP...

Distributed Computing

- Il distributed computing rappresenta la possibilità di delocalizzare l'elaborazione sui nodi di una rete in modo trasparente per il programmatore ed il codice applicativo
- Attraverso opportuni meccanisimi di *binding*, *lookup* e serializzazione di oggetti, l'invocazione di metodi e l'accesso a funzioni remote avviene come se risorse fossero locali: la presenza della rete è totalmente mascherata dalle librerie applicative

Distributed Computing

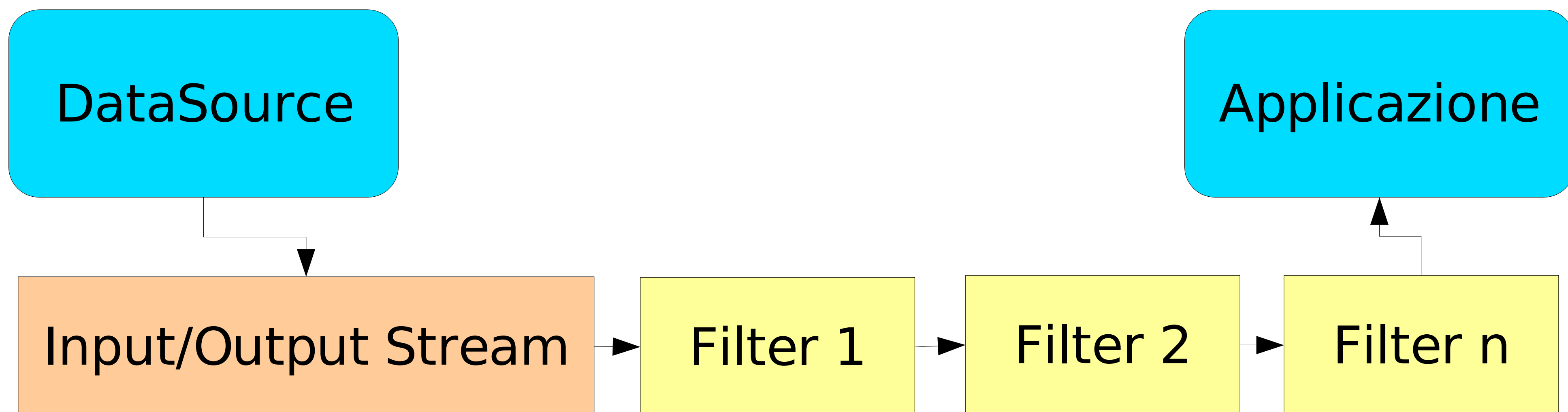
- Esempi di distributed computing sono:
  - Java RMI: permette l'invocazione remota di metodi in maniera trasparente per il programmatore. Oggetti locali e remoti sono trattati allo stesso modo
  - CORBA: permette l'interoperabilità tra componenti software scritti in linguaggi diversi
  - Web Service: usano l'invocazione remota di metodi attraverso scambio di documenti XML su protocollo HTTP
    - XML-RPC
    - SOAP

Stream model

- Il sistema di I/O di Java è basato sul modello a *stream*: opportuni oggetti permettono di gestire i *flussi* di byte in ingresso e in uscita
- Il modello a stream permette una gestione omogenea delle sorgenti di dati, a prescindere dal supporto fisico (file, rete, porta seriale, link Bluetooth...)
- Attraverso un insieme componibile di filtri (*decorator*) e adattatori è possibile elaborare i byte in transito

Input/Output Stream

- InputStream:
  - Gestisce i flussi di byte in ingresso
  - Fornisce facility di buffering elementari
- OutputStream:
  - Gestisce i flussi di byte in uscita (con buffering)



Data Stream

- I tipi base quali numeri interi, numeri in virgola mobile, valori logici e stringhe (ASCII, Unicode, UTF) sono codificati con più byte (non sempre indipendenti dalla piattaforma)
- Al fine di rendere la trasmissione di tali dati trasparente rispetto alla effettiva codifica (si pensi alla variabilità del numero di bit di un valore numerico intero) si utilizzano i DataStream, che offrono i metodi per leggere/scrivere int, short, long, float, double, String...

Gestione caratteri: reader

- L'introduzione di specifiche quali Unicode (1.0 e 2.0) e UTF-8 ha portato alla necessità di trattare in maniera dedicata il trasferimento di stream di caratteri: non è garantita, infatti, la corrispondenza “classica” 1byte == 1char! :-(

- Reader e Writer sono gli strumenti utilizzati da Java per accedere a flussi di caratteri indipendentemente dalla codifica utilizzata

Buffering

- Alcuni protocolli si basano su scambio sequenze di stringhe separate da CR+LF: il protocollo HTTP, ad esempio, descrivere le intestazioni (header) delle HTTP Request e Response

- La gestione dei flussi “per righe” può essere fatta agevolmente attraverso l'uso di classi quali il BufferedReader, che offre metodi per la lettura “line by line”

Low-level network programming

- Introduciamo ora i concetti base per lo sviluppo di applicazioni che utilizzano i protocolli di comunicazione a basso livello:
  - IP
  - UDP (Datagram Packet e Datagram Socket)
  - TCP (Socket e ServerSocket)

IP

- La libreria standard di Java offre classi per:
  - Selezione di una specifica interfaccia di rete sulla macchina host
  - Recuperare informazioni su un indirizzo Internet (IPv4 e IPv6)

- InetAddress: è la classe che incapsula un indirizzo IP. E' utilizzata dalle classi per la gestione dei protocolli UDP e TCP.

InetAdress

- Metodi principali dell'interfaccia pubblica:
  - **InetAddress.getByName(String host)**
  - **InetAddress.getAllByName(String host)**
  - **String getHostName()**
  - **byte[] getAddress()**
  - **String getHostAddress()**

nslookup...

```
import java.net.*;
public class NSLookup {

  public static void main(String[] args) {
    try {
      String addr = InetAddress.getByName(args[0]).getHostAddress();
      System.out.println("Host " + args[0] + " has IP: " + addr);
    }
    catch(Exception e) {
      System.out.println("Error: " + args[0]);
    }
  }
}
```

UDP

- UDP: User Datagram Protocol
  - Permette di inviare/ricevere pacchetti di byte (datagram) ad/da un host su una porta nota
  - E' un protocollo connection-less, non orientato alla connessione (in senso stretto); non è garantito l'ordine di arrivo dei pacchetti: è compito dell'applicazione marcare i dati per ricostruirne la sequenza
  - Non offre meccanismi di controllo sul corretto invio dei dati: lo strato applicativo deve gestire tale controllo
  - Protocolli applicativi basati su UDP: DNS, NFS...

Datagram

- Java offre alcune classi per la ricezione e l'invio di pacchetti UDP:
  - Datagram Packet: è un classe “double-face” usato come contenitore per l'invio e la ricezione dei pacchetti
  - Datagram Socket: è la classe responsabile dell'invio dei Datagram Packet ad un server remoto e l'ascolto su una porta UDP locale

Datagram Packet

- Pacchetti in uscita
  - Sono costruiti a partire da un byte[] contenente i dati da inviare:

```
DatagramPacket(byte[] buf, int length, InetAddress address, int port)
```

- Pacchetti in ingresso
  - Sono costruiti a partire da un byte[] usato come buffer per i dati in arrivo:

```
DatagramPacket(byte[] buf, int length)
```

Datagram Packet

- Metodi principali dell'interfaccia pubblica:
  - `byte[] getData()`
  - `InetAddress getAddress()`
  - `int getLength()`
  - `int getPort()`

Datagram Socket

- E' responsabile dell'invio e della ricezione dei Datagram Packet. Metodi principali dell'interfaccia pubblica:
  - `DatagramSocket(int port)`
  - `int getLocalPort()`
  - `void send(DatagramPacket p)`
  - `void receive(DatagramPacket p)`
- E' possibile assegnare staticamente un Datagram Socket ad un host remoto (ferme restando tutte le caratteristiche di UDP)

TCP/IP

- Il TCP/IP è un protocollo a livello di trasporto:
  - Orientato alla connessione: si stabilisce un canale virtuale tra client e server che rimane attivo sino alla chiusura della connessione
  - Trasparente: l'implementazione si occupa di assemblare la corretta sequenza dei pacchetti in arrivo
  - Affidabile: in caso di pacchetti persi o corrotti, lo stack si occupa di chiedere la ritrasmissione dei pacchetti mancanti

Socket

- La classe Socket offre i meccanismi base per aprire una connessione TCP verso un server remoto

- Attraverso il modello basato su stream è possibile ricevere e inviare dati in modo trasparente rispetto alla rete fisica utilizzata e ai meccanismi di connessione di quest'ultima

Socket

- Metodi principali dell'interfaccia pubblica:
  - `Socket(String host, int port)`
  - `InputStream getInputStream()`
  - `OutputStream getOutputStream()`
  - `int getPort()`
  - `int getLocalPort()`
  - `InetAddress getInetAddress()`
  - `InetAddress getLocalInetAddress()`
  - `void close()`

ServerSocket

- La classe ServerSocket è utilizzata creare ed assegnare server TCP ad una porta nota

- Il modello di comunicazione è simmetrico: per ogni connessione, la classe ServerSocket restituisce una istanza di Socket, i cui input e output stream forniscono i canali di comunicazione verso il client remoto

ServerSocket

- Metodi principali dell'interfaccia pubblica:
  - `ServerSocket(int port)`
  - `Socket accept();`
  - `int getLocalPort()`
  - `InetAddress getInetAddress()`
  - `void close()`

Network Games

Network games :-)

- Il miglior modo per acquisire praticità con lo sviluppo di applicazioni network-oriented e con i protocolli più comuni è... giocarci! :-)

- Sistemi asincroni, multithreading, sicurezza, accesso concorrente ai dati sono problematiche comuni a molte applicazioni di rete...

Echo

- Vogliamo scrivere una semplicissima applicazione che riceve una stringa attraverso Socket e la rimanda indietro.
- Client
  - Apre un socket verso il server
  - Invia e riceve una stringa
- Server
  - Apre un serversocket e accetta una connessione
  - Restituisce le stringhe ricevute

Echo server

```
import java.io.*;
import java.net.*;
public class EchoServer {
    public static void main(String[] args) throws Exception {
        ServerSocket serverSocket = new ServerSocket(8899);
        Socket socket = serverSocket.accept();
        Reader reader = new InputStreamReader(socket.getInputStream());
        BufferedReader buffer = new BufferedReader(reader);
        PrintStream ps = new PrintStream(socket.getOutputStream());
        String read = "";
        while ((read = buffer.readLine()) != null) {
            System.out.println("Ricevuto: " + read);
            ps.println(read);
        }
}   }
```

Echo client

```
import java.io.*;
import java.net.*;
import java.util.Date;
public class EchoClient {
    public static void main(String[] args) throws Exception {
        Socket clientSocket = new Socket("127.0.0.1", 8899);
        Reader reader = new InputStreamReader(clientSocket.getInputStream());
        BufferedReader buffer = new BufferedReader(reader);
        PrintStream ps = new PrintStream(clientSocket.getOutputStream());
        while (true) {
            ps.println(new Date(System.currentTimeMillis()));
            System.out.println("Tornato indietro: " + buffer.readLine());
            Thread.sleep(200);
        }
} }
```

Esplorazione di un protocollo

- I Socket costituiscono un ottimo strumento didattico per imparare a gestire protocolli di complessità crescente
- In un modello client/server è possibile:
  - Creare un server fittizio per intercettare e tracciare il formato delle chiamate effettuate da un client
  - Create un client fittizio che, a fronte di una richiesta “elementare” ad un server reale, intercetti e descriva nel dettaglio il formato dei dati restituiti dal server

Cosa manda un browser?

- Si supponga di voler esaminare il formato di una richiesta HTTP inviata da un browser al server web
- Sarà necessario:
  - Creare un serversocket (verosimilmente sulla porta 80)
  - Leggere i dati inviati dal client riga per riga

HTTP Request sniffer...

```
import java.io.*;
import java.net.*;
public class HTTPRequestSniffer {
    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(80);
            Socket clientSocket = serverSocket.accept();
            InputStream input = clientSocket.getInputStream();
            BufferedReader buffer = new BufferedReader(new InputStreamReader(input));
            String read = "";
            int line = 0;
             while ((read = buffer.readLine()) != null) {
                System.out.println(Integer.toString(line++) + " " + read);
            }
        } catch(Exception e) { e.printStackTrace(); }
  } }
```

HTTP Request: Mozilla

```
GET / HTTP/1.1

Host: localhost

User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US;
   rv:1.0.1) Gecko/20020823 Netscape/7.0

Accept:
   text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,te
   xt/plain;q=0.8,video/x-
   mng,image/png,image/jpeg,image/gif;q=0.2,text/css,*/*;q=0.1

Accept-Language: en-us, en;q=0.50

Accept-Encoding: gzip, deflate, compress;q=0.9

Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66

Keep-Alive: 300

Connection: keep-alive
```

HTTP Request: Internet Explorer

```
GET / HTTP/1.1

Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
   application/vnd.ms-powerpoint, application/vnd.ms-excel,
   application/msword, application/x-shockwave-flash, */*

Accept-Language: it

Accept-Encoding: gzip, deflate

User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET
   CLR 1.1.4322)

Host: localhost

Connection: Keep-Alive
```

Cosa invia un server web?

- Conoscendo il formato di una richiesta elementare inviata da un browser web, possiamo preparare un client fittizio con cui tracciare la risposta restituita dal server

- Sarà necessario:
  - Create un client socket verso un server web attivo
  - Inviare la richiesta minima (**`GET / HTTP/1.x`**)
  - Leggere i dati inviati dal server riga per riga

HTTP Response sniffer...

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

public class HTTPResponseSniffer {

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

        Socket clientSocket = new Socket("java.sun.com", 80);

        InputStream input = clientSocket.getInputStream();

        BufferedReader buffer = new BufferedReader(new InputStreamReader(input));

        PrintStream ps = new PrintStream(clientSocket.getOutputStream());

        ps.println("GET / HTTP/1.0\n\n");

        String read = "";

        int line = 0;

        while ((read = buffer.readLine()) != null) {

            System.out.println(Integer.toString(line++) + " " + read);

        }

    }

}
```

HTTP Response di *java.sun.com*

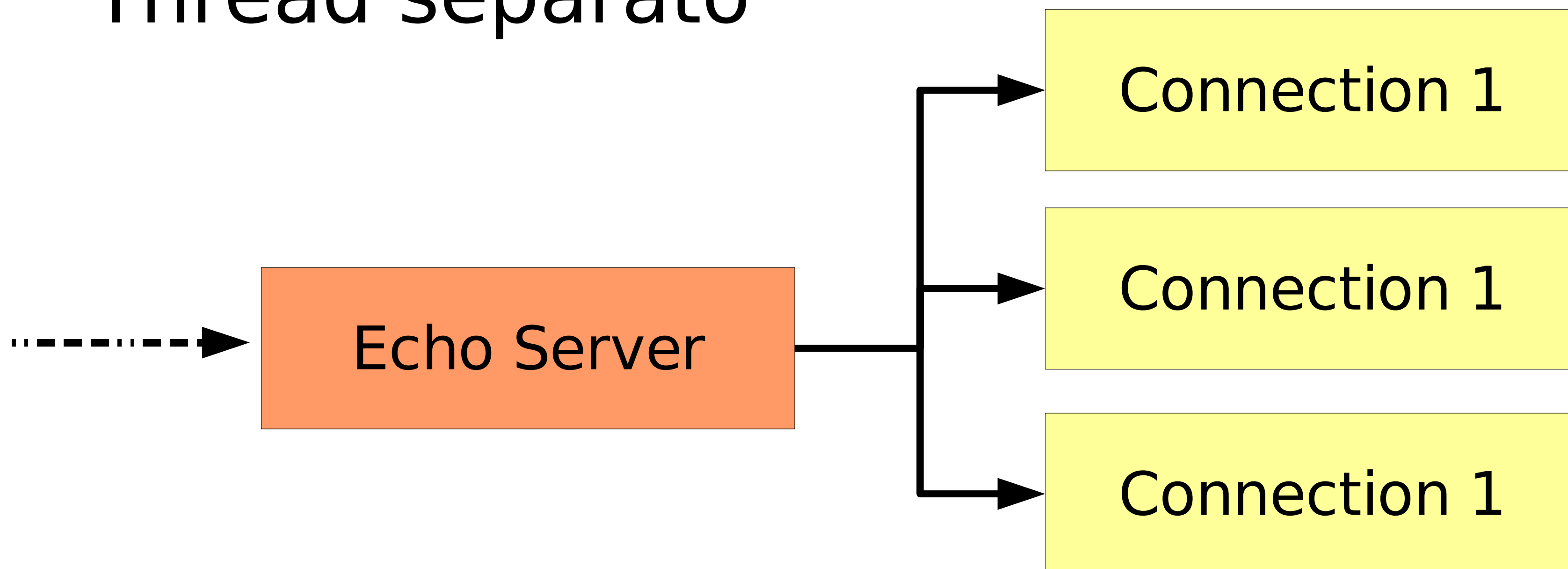
```
HTTP/1.1 200 OK

Server: Netscape-Enterprise/6.0

Date: Tue, 18 May 2004 12:53:33 GMT

Content-type: text/html;charset=ISO-8859-1

Set-cookie: JSESSIONID=java.sun.com-987f%253A40aa074a%
   253A391e72d077d4b9a;path=/;expires=Tue, 18-May-2004 13:23:31 GMT

Connection: close



<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>

<head>

<title>Java Technology</title>

<meta name="keywords" content="Java, platform" />



[...]
```

Multithreading

- Rispetto ad altri linguaggi (Pascal, C/C++...), Java nasce con la multiprogrammazione “in mente”
- La possibilità di gestire più flussi di esecuzione concorrenti in maniera nativa nel linguaggio è fondamentale per facilitare lo sviluppo di applicazioni network-oriented
- Il server deve gestire in modo concorrente più connessioni, delegando a ciascun thread la gestione di un client

Multithreading

- L'esempio di “Echo server” precedentemente illustrato... funziona una sola volta e con un solo client!
- E' necessario incapsulare la gestione di una singola connessione all'interno di un Thread separato



Multithreading

- Un Thread gestisce la connessione:

```
private static class Connection extends Thread {
        private Socket clientSocket;
        public Connection(Socket socket) {
            this.clientSocket = socket;
            start();
        }
        public void run() {
            try {
             // CODICE DI LETTURA:..
            }
            catch(Exception e) { e.printStackTrace(); }
        }
    }
```

Multithreading

- Il server istanzia un Thread per ogni connessione ricevuta:

```
public static void main(String[] args) throws Exception {
    ServerSocket serverSocket = new ServerSocket(8899);
    while (true) {
        Socket clientSocket = serverSocket.accept();
        new Connection(clientSocket);
    }
}
```

Protocolli applicativi

- L'API Java offre classi per la gestione di protocolli a livello applicativo (ad esempio HTTP), che permettono di sviluppare in modo semplice e rapido potenti applicazioni network oriented

- Il network programming permette di trascurare tutti i dettagli del protocollo sottostante per concentrarsi solamente sullo scambio dei contenuti

URL e URLConnection

- La classe URL rappresenta un Uniform Resource Locator, cioè un riferimento formale ad una risorsa reperibile su Internet
- La classe URLConnection rappresenta la connessione al servizio su cui risiede la risorsa individuata da un URL
- Attenzione:
  - URL: non più "indirizzo IP"!
  - URLConnection: non più Socket!

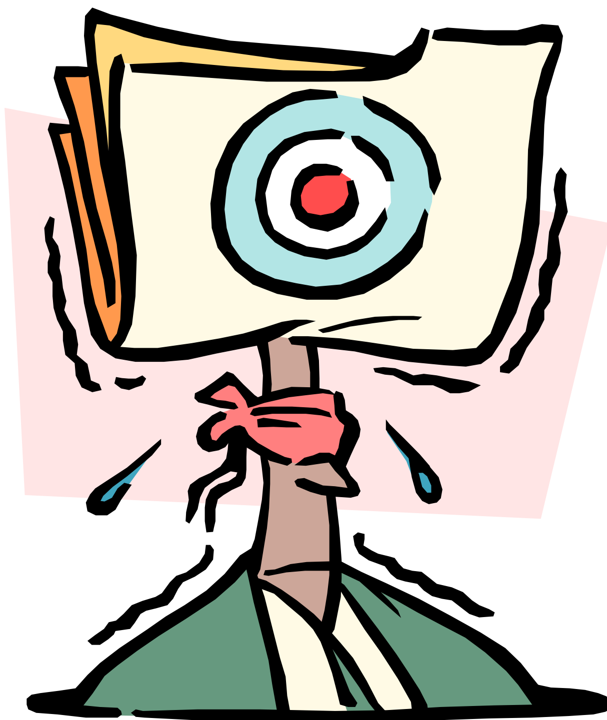
HttpURLConnection

- E' la classe che incapsula la gestione di una connessione HTTP ad un server web
- Nasconde tutti i dettagli del protocollo, gestendo l'invio degli header, la codifica dei dati binari e la notifica degli errori di elaborazione
- Si ottiene a partire da una URL valida

HttpURLConnection

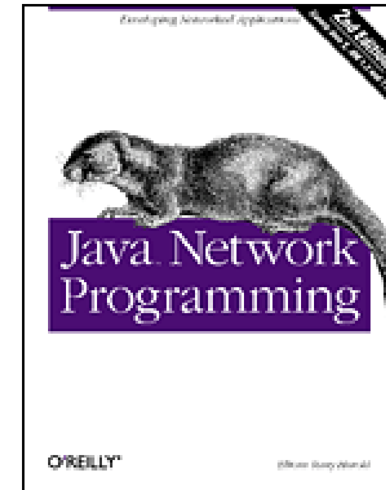
- Metodi principali dell'interfaccia pubblica:
  - `URL.openConnection()`
  - `Object getContent()`
  - `int getContentLength()`
  - `String getHeaderField(String name)`
  - `String getRequestProperty(String name)`
  - `InputStream getInputStream()`
  - `OutputStream getOutputStream()`

Domande?



Bibliografia

- Java Network Programming
  E. Rusty Harold
  O'Reilly 2000
  ISBN: 1565928709



  XML-RPC
  F. Caboni e S. Sanna
  Gruppo Utenti Linux Cagliari
  LinuxDay 2003
  http://linuxday.gulch.crs4.it

Il protocollo
XML-RPC
e le sue applicazioni

Federico Caboni & Stefano Sanna

GULCh – Gruppo Utenti Linux Cagliari

Conclusioni

- Java offre una potente libreria per la gestione delle connessioni in rete
- Qualsiasi sia la versione utilizzata (Enterprise, Standard, Micro), vi è sempre la possibilità di connettersi ad un server remoto per inviare e ricevere dati
- java.net.* è una buona palestra per prendere dimestichezza con i più diffusi protocolli di comunicazione!

Grazie... :-)

Copyright (c) 2004-2005 CRS4

Scritto da Stefano Sanna (gerda@crs4.it)

è garantito il permesso di copiare, distribuire e/o modificare questo documento seguendo i termini della Licenza per Documentazione Libera GNU, Versione 1.1 o ogni versione successiva pubblicata dalla Free Software Foundation. Una copia della licenza in lingua italiana è disponibile presso:

http://www.softwarelibero.it/gnudoc/fdl.it.html