

Java

accesso alla rete -- UDP

G. Prencipe
prencipe@di.unipi.it

Sulle esercitazioni....

1. Ripasso del codice visto nella lezione precedente
2. Ripasso della struttura complessiva di client e server tipici
3. Soluzioni agli esercizi

Apertura socket (client)

```
import java.net.*;
/* ... */
void client(String server, int port) {
    Socket socket;
    socket=new Socket(server,port);
    /* ... usa il socket ... */
    socket.close();
}
```

Apertura socket (server)

```
import java.net.*;
/* ... */
void server(int port) {
    ServerSocket ssocket;
    ssocket=new ServerSocket(port);
    while (!serverFinito) {
        Socket socket=ssocket.accept()
        /* ... usa il socket ... */
        socket.close();
    }
    ssocket.close();
}
```

I/O con i socket

```
import java.net.*;
import java.io.*;
/* ... */
BufferedReader in =
    new BufferedReader(
        new InputStreamReader(
            socket.getInputStream()));
PrintWriter out =
    new PrintWriter(
        socket.getOutputStream());
/* ... usa in e out ... */
in.close();
out.close();
```

Scambio dati

```
String input, output;
/* ... */
while (!transazioneFinita) {
    input=in.readLine();
    /* ... */
    out.println(output);
    /* ... */
}
```

Struttura complessiva (client)

```
import java.net.*;
import java.io.*;
/* ... */
void client(String server, int port) {
    Socket socket;
    socket=new Socket(server,port);

    BufferedReader in =
        new BufferedReader(
            new InputStreamReader(
                socket.getInputStream()));
    PrintWriter out =
        new PrintWriter(
            socket.getOutputStream());

    String input, output;
    while (!transazioneFinita) {
        input=in.readLine();
        /* ... */
        out.println(output);
        /* ... */
    }
    in.close();
    out.close();

    socket.close();
}
```

Struttura complessiva (server)

```
import java.net.*;
import java.io.*;
/* ... */
void server(int port) {
    ServerSocket ssocket;
    ssocket=new ServerSocket(port);
    while (!serverFinito) {
        Socket socket=ssocket.accept()

        BufferedReader in =
            new BufferedReader(
                new InputStreamReader(
                    socket.getInputStream()));
        PrintWriter out =
            new PrintWriter(
                socket.getOutputStream());

        String input, output;
        while (!transazioneFinita) {
            input=in.readLine();
            /* ... */
            out.println(output);
            /* ... */
        }
        in.close();
        out.close();
        socket.close();
    }
    ssocket.close();
}
```

Esercizio 0

- Scrivere un server **TCPEchoServer** che accetta come messaggi linee di testo dal cliente, conta i messaggi ricevuti, e manda indietro al cliente il messaggio (numerato)
- Scrivere il relativo cliente **TCPEchoClient**

Soluzione per l'es. 0

```
public class EchoServer {
    public static void main(String[] args) {
        int count=1;
        try{
            ServerSocket ss = new ServerSocket(2222);
            Socket s = ss.accept();
            BufferedReader in = ...;
            PrintWriter out = new ...(...,true);
            String letto=in.readLine();
            while(!letto.equals("FINE")){
                out.println(letto+" "+count);
                count++;
                letto=in.readLine();
            }
            ss.close();
            s.close();
        }catch (IOException e){e.printStackTrace();}
    }
}
```

Soluzione per l'es. 0

```
public class EchoClient {
    public static void main(String[] args) {
        String host = "localhost";
        String inConsole;
        try{
            Socket s = new Socket (host,2222);
            System.out.println("Inserire frase...");
            BufferedReader in = ...;
            PrintWriter out = ...(...,true);
            BufferedReader bR = ... (System.in);
            inConsole = bR.readLine();
            while(!inConsole.equals("FINE")){
                out.println(inConsole);
                String letto = in.readLine();
                System.out.println(letto);
                inConsole = bR.readLine();
            }
            out.println("FINE");
            s.close();
        }catch (IOException e){e.printStackTrace();} } }
```

Esercizio 1

- Scrivere un server **DaytimeServer** che, ad ogni connessione, risponda inviando al client la data e l'ora corrente (a questo provvede la classe **Date**)
- Il server deve chiudere la connessione subito dopo aver inviato la data al client
- Scrivere il relativo cliente **DaytimeClient**

Soluzione (base) per l'es. 1

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

class DaytimeServer {
    public void server(int port) {
        ServerSocket ssocket;
        ssocket = new ServerSocket(port);
        while (true) {
            Socket socket = ssocket.accept();
            PrintWriter out = new PrintWriter(socket.getOutputStream());
            out.println(new Date());
            out.close();
            socket.close();
        }
        ssocket.close();
    }
    public static void main(String[] args) {
        (new TimeServer()).server(5000);
    }
}
```

Eccezioni nell'es. 1

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

class DaytimeServer {
    public void server(int port) {
        ServerSocket ssocket;
        ssocket = new ServerSocket(port);
        while (true) {
            Socket socket = ssocket.accept();
            PrintWriter out = new PrintWriter(socket.getOutputStream());
            out.println(new Date());
            out.close();
            socket.close();
        }
        ssocket.close();
    }
    public static void main(String[] args) {
        (new TimeServer()).server(5000);
    }
}
```

Grave: il server non può partire!

*Sarebbe grave, ma:
1. non ci possiamo fare nulla
2. e comunque, non ci arriviamo mai!*

Non possiamo servire un cliente: pazienza!

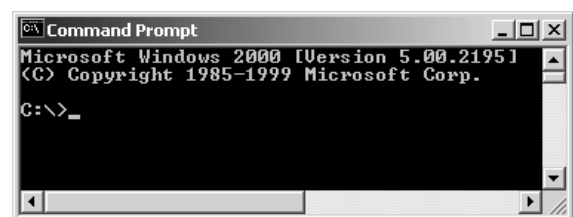
Soluzione (completa) per l'es. 1

```
public void server(int port) {
    ServerSocket ssocket;
    try {
        ssocket = new ServerSocket(port);
        while (true) {
            try {
                Socket socket = ssocket.accept();
                PrintWriter out = new PrintWriter(socket.getOutputStream());
                out.println(new Date());
                out.close();
                socket.close();
            } catch (IOException e) {
                ;
            }
        }
        ssocket.close();
    } catch (IOException e) {
        System.err.println("Impossibile partire: " + e);
    }
}
```

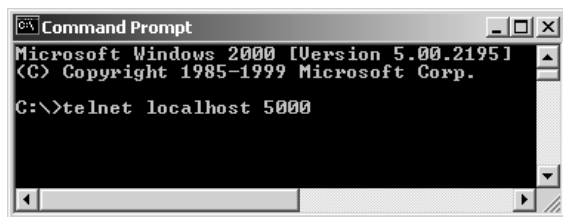
Non possiamo servire un cliente: pazienza!

Grave: il server non può partire!

Ma funziona?



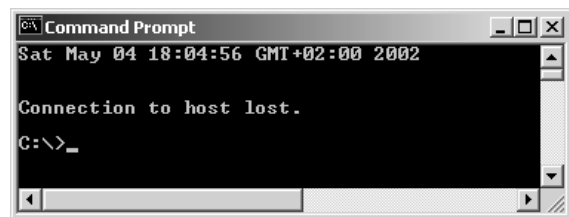
Ma funziona?



```
Microsoft Windows [Version 5.00.2195]
(C) Copyright 1985-1999 Microsoft Corp.

C:\>telnet localhost 5000
```

Ma funziona?



```
Sat May 04 18:04:56 GMT+02:00 2002

Connection to host lost.

C:\>_
```

Esercizio 2

- Scrivere un server **DumpServer** che si limiti a stampare su **System.out** tutto ciò che arriva alla porta 4040
- Scrivere un client **DumpClient** che mandi sulla porta 4040 di un server dato una stringa passata come argomento
 - **DumpServer**: la porta deve essere fornita come argomento
 - **DumpClient**: il server e la porta a cui collegarsi devono essere passati come argomento
- Possiamo usare questo programma per guardare meglio cosa succede durante una connessione web?

Soluzione (completa) per l'es. 2

Codice del server

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

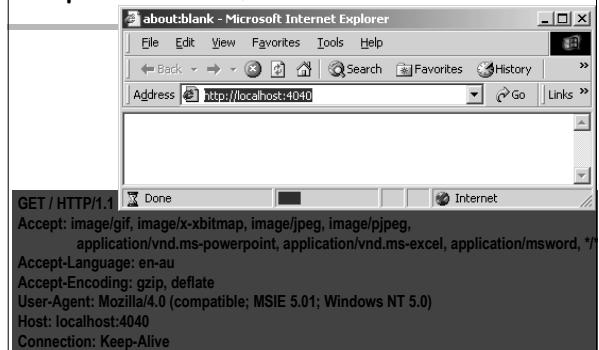
class DumpServer {
    public static void main(java.lang.String[] args) {
        (new DumpServer()).server(4040);
    }

    public void server(int port) {
        /* prossimo lucido */
    }
}
```

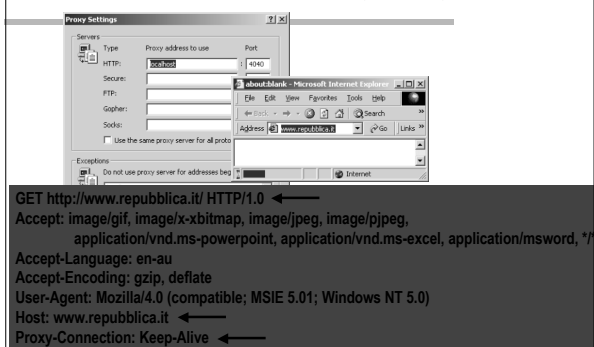
Soluzione (completa) per l'es. 0

```
public void server(int port) {
    ServerSocket ssocket;
    try {
        ssocket = new ServerSocket(port);
        while (true) {
            try {
                Socket socket = ssocket.accept();
                BufferedReader in =
                    new BufferedReader(new InputStreamReader(socket.getInputStream()));
                try {
                    while (true) {
                        String line = in.readLine();
                        System.out.println(line);
                    }
                } catch (IOException e) { ; }
                finally {
                    try {
                        in.close();
                        socket.close();
                    } catch (Exception e) { ; }
                }
            } catch (IOException e) { ; }
        }
    } catch (IOException e) { ; }
    ssocket.close();
} catch (IOException e) {
    System.err.println("Impossibile partire: " + e);
}
```

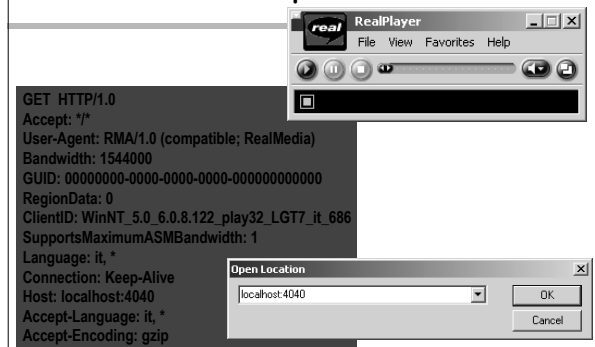
E quest'altro, funziona?



E se lo usiamo come proxy?



Wow! Proviamo qualcos'altro...



Soluzione (completa) per l'es. 2

Codice del client

- Banale!
- Si prende il codice del client generico e si inserisce
`out.println(args[0])`
nella parte “usa in e out”
- `in` non serve
- `args` va passato come argomento a `client()`

Esercizio 3

- Scriviamo un **server web** (semplificato)
- Il server riceve dal client un comando come
`GET /un_certo_path/un_certo_file.html`
- Il server risponde inviando al client un codice di successo (si veda il protocollo HTTP), seguito dal contenuto del file indicato da GET
- Il ciclo si può ripetere; la connessione viene abbattuta dal client quando non vuole più chiedere altri file, oppure dopo 30 secondi senza comandi GET

Soluzione per l'es. 3

- Apro il server socket; accetto la connessione e costruisco `in` e `out`
- Leggo il comando con `in.readLine()`, lo analizzo e prendo il nome del file (se il comando non è GET, mando un codice d'errore)
- Provo ad aprire il file, se ci riesco mando il codice per “OK”, altrimenti un codice d'errore
- Leggo il file riga per riga e mando ogni riga sul socket con `out.println()`
- Quando il file è finito, chiudo il file, `in`, `out` e il socket, e torno alla `ssocket.accept()`

Ora vedremo....



- Prossimi argomenti:
 - connessioni Datagram
 - le classi `URL` e `URLConnection`
- Poi:
 - altri esercizi!

Datagram



- Un *datagram* è un pacchetto di informazioni trasmesso via rete
 - indipendente
 - auto-contenuto
- la cui consegna, tempo di percorrenza, e contenuto all'arrivo **non sono garantiti**
- In Java: classi `DatagramSocket` e `DatagramPacket`

TCP vs Datagram

- **TCP** garantisce la consegna delle informazioni
 - Questa garanzia si paga in termini di overhead
- **UDP** invece non garantisce che i pacchetti verranno consegnati e non garantisce che verranno consegnati nell'ordine in cui sono stati inviati
 - File audio, telefonia, streaming....

DatagramSocket



- Quando si usano i datagram, non c'è distinzione fra client e server: le due parti sono paritarie nella comunicazione (anche se *logicamente* i ruoli di client e server possono permanere)
- Di conseguenza, c'è una sola classe per i socket datagram: `DatagramSocket`
- Su `DatagramSocket` si mandano e ricevono singoli pacchetti UDP

DatagramSocket



■ Costruttori

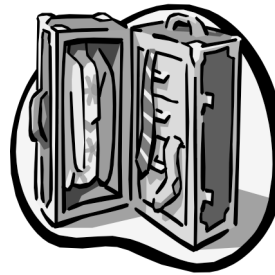
<code>DatagramSocket ()</code>	Crea un socket su una porta qualunque
<code>DatagramSocket (porta)</code>	Crea un socket sulla porta specificata
<code>DatagramSocket (porta, inetaddress)</code>	Crea un socket sulla porta specificata, che risponde all'indirizzo indicato

Dettaglio importante...



- Nel costruttore di **Socket**, si indica la porta **del partner** a cui ci si vuole connettere
 - `sock = new Socket(host,p);`
vuol dire: mi collego alla porta p di host
- Invece, nel costruttore di **DatagramSocket**, si indica la porta **propria** che si vuole aprire
 - `sock=new DatagramSocket(p);`
vuol dire: apro la mia porta p
- Il server logico di solito specifica la porta, il client logico no (usa il costruttore senza argomenti)

DatagramPacket



- La classe **DatagramPacket** funge da “contenitore” per i dati utili di un datagram
- I dati vengono trattati come un array di byte
- Sta alle applicazioni interpretare questi byte nel modo giusto

Inizializzazione

- Si dichiara un array di byte
- Si chiama il costruttore di **DatagramPacket**
 - Argomenti: il buffer e la sua lunghezza
- La lunghezza del buffer è importante!

```
byte[] buf = new byte[256];  
DatagramPacket packet =  
    new DatagramPacket(buf, buf.length);
```

Ricezione

- Si costruisce un **DatagramPacket**
- Si chiama il metodo **receive()** del socket
- Il programma si ferma in attesa di un pacchetto
- Quando il pacchetto arriva, **receive()** ritorna
- I dati ricevuti sono nel **DatagramPacket**
 - se il buffer non è abbastanza lungo, si perdono i dati in eccesso
 - il campo **length** del pacchetto dice quanti dati sono stati ricevuti

Ricezione

- Una volta ricevuto il pacchetto, si possono estrarre tante informazioni chiamando gli opportuni metodi di **DatagramPacket**

```
socket.receive(packet);
```

```
InetAddress address = packet.getAddress();  
int port = packet.getPort();  
byte[] buf = packet.getData();  
int len = packet.getLength();  
int offset = packet.getOffset();
```

Classe InetAddress

- La classe **InetAddress** rappresenta un indirizzo IP
- Ha diversi metodi utili per manipolare indirizzi IP
 - Ad esempio, il metodo **statico** **InetAddress.getByName(host)** restituisce l'indirizzo IP associato a **host** (una **Stringa**)

Trasmissione

- Si costruisce un **DatagramPacket**
- Si riempie il pacchetto con i dati desiderati
- Si impostano l'indirizzo e la porta di destinazione del pacchetto
 - Oppure: si crea un nuovo pacchetto passando al costruttore i dati, l'indirizzo e la porta di destinazione
- Si chiama il metodo **send()** del socket

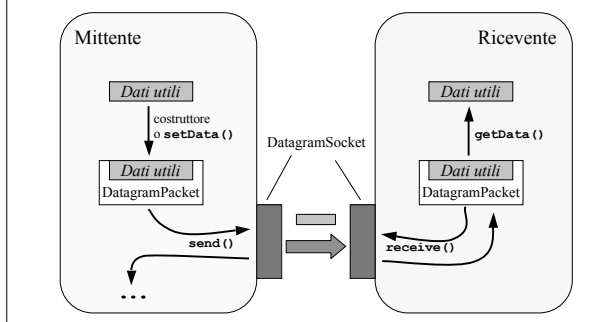
Trasmissione

- Quando si effettua la **send()**, il pacchetto viene inviato (verrà recuperato dalla **receive()** del partner)

```
byte[] buf = new byte[256];  
InetAddress address = InetAddress.getByName(host);  
DatagramPacket packet =  
    new DatagramPacket(buf, buf.length, address, 4445);
```

```
socket.send(packet);
```

Comunicazione Datagram



Sembra complicato?

■ Facciamo un esempio:

- Vogliamo scrivere un server che invii la quotazione corrente di un certo titolo ad ogni client che ne faccia richiesta
- Usiamo UDP: forse va bene, forse va male
- Se il client non riceve risposta, ritenterà...



QuoteServer UDP

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

public class QuoteServer {
    DatagramSocket socket = new DatagramSocket(4445);
    boolean errore=false;

    public void run() {
        while (!errore) {
            try {
                byte[] buf = new byte[256];
                DatagramPacket packet = new DatagramPacket(buf, buf.length);
                socket.receive(packet);
                String q = quotazioneCorrente();
                buf = q.getBytes();
                InetAddress address = packet.getAddress();
                int port = packet.getPort();
                packet = new DatagramPacket(buf, buf.length, address, port);
                socket.send(packet);
            } catch (IOException e) {
                e.printStackTrace();
                errore = true;
            }
        }
        socket.close();
    }
}
```



QuoteClient UDP

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

public class QuoteClient {
    {
        public static void main(String[] args) throws IOException
        {
            DatagramSocket socket = new DatagramSocket();
            byte[] buf = new byte[256];
            InetAddress addr = InetAddress.getByName(args[0]);
            DatagramPacket packet = new DatagramPacket(buf, buf.length, addr, 4445);
            socket.send(packet);
            packet = new DatagramPacket(buf, buf.length);
            socket.receive(packet);
            String ricevuto = new String(packet.getData());
            System.out.println("Quotazione corrente: " + ricevuto);
            socket.close();
        }
    }
}
```



Alcune note...



- In **QuoteServer** va aggiunto un metodo **main()** (come al solito)
- In **QuoteClient** andrebbe impostato un timeout:

```
socket.setSoTimeout(1000);
```


altrimenti, se si perde la risposta (o anche la domanda) il client si blocca!
- In caso di timeout si verifica un'eccezione: va bene la gestione che già abbiamo.

Esercizio 4

- Usando UDP, scrivere un server **UDPEchoServer** che accetta una frase da un cliente e la stampa
- Scrivere il relativo cliente **UDPEchoClient**

Esercizio 5

- Partendo dall'Esercizio 4, svolgere l'Esercizio 0 (**UDPEchoServer** che numera i messaggi) utilizzando UDP

MulticastSocket

- È possibile usare UDP per trasmettere gli stessi dati *simultaneamente* a più clienti (**multicast**):
 - Il server manda dati a un indirizzo "di gruppo" (per esempio, 131.4.0.1)
 - I client aprono un **MulticastSocket** sull'indirizzo di gruppo e si iscrivono fra gli ascoltatori:

```
MulticastSocket socket=new MulticastSocket(porta);  
InetAddress grp=InetAddress.getByName("131.4.0.1");  
socket.joinGroup(grp);
```
 - Si invia e si riceve come al solito; quando un client ha finito, esce dal gruppo con `socket.leaveGroup(grp);`

Quando usare il multicast

- È utile per i servizi tradizionalmente in stile *broadcast*: trasmissioni TV o radio
- È utile anche per i servizi *push* di altro tipo: il quote server ne è un esempio
- Ma anche per altri dati: notizie, aggiornamenti sul traffico, previsioni meteo....

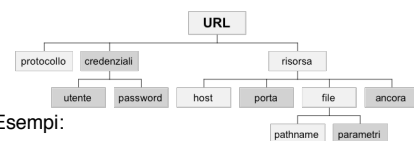
Accesso al Web

- I programmi Java hanno un rapporto particolarmente stretto con il Web
- Infatti, esistono classi per facilitare l'accesso a risorse sul Web in maniera *semi-trasparente*
- Le applicazioni Java possono usare queste classi per interagire con altre macchine (usando vari protocolli: HTTP, FTP, ecc.)

URL e URL

- URL = *Uniform Resource Locator*
- È un **riferimento** a una risorsa sul Web
 - (esattamente come il nome di un file è un riferimento al contenuto del file su disco)
 - Però è più generale: una risorsa può anche essere un programma da eseguire, una interrogazione su un motore di ricerca, un ordine di acquisto per una biblioteca on-line...
- **URL** è una classe nel package **java.net**
- Serve a rappresentare le URL all'interno di un programma Java

Struttura di una URL



- Esempi:
 - <http://www.repubblica.it/>
 - <ftp://pino:segreto@download.com/pub/warez/fifa2005.zip>
 - <http://www.google.it/cgi-bin/query.exe?q=De+Curtis>
 - <http://localhost:4040/~gervasi/index.html#linkUtili>
 - <mailto:prencipe@di.unipi.it>

Creare una URL

- Esistono diversi costruttori per URL. Eccone alcuni
 - **URL(String s)**
 - Accetta una stringa che *deve* rappresentare una URL
 - **URL(String protocol, String hostname, String file)**
 - Costruisce una URL a partire dalle sue componenti
 - **URL(URL base, String URLrelativa)**
 - Costruisce una URL a partire da una URL di base e una URL relativa

Costruire una URL

<code>url=new URL("http://www.go.it");</code>	da una URL testuale
<code>base=new URL("http://www.go.it"); url=new URL(base,"/index.html");</code>	relativa rispetto a una URL di base data
<code>url=new URL("http","www.go.it", 80,"index.html");</code>	pezzo per pezzo (in varie combinazioni)

- ⚠ Se i parametri sono errati, il costruttore fallisce e genera una **MalformedURLException**
- ⚠ Una volta creata, una **URL** non può essere cambiata

Altri metodi di URL

<code>getProtocol() getHost() getPort() getFile() getRef()</code>	restituiscono i vari componenti della URL rappresentata dalla URL
<code>toString()</code>	restituisce la versione testuale della URL (<code>http://.../...</code>)
<code>sameFile(URL altra)</code>	restituisce true se l'altra URL si riferisce allo stesso file a cui fa riferimento l'URL corrente

Leggere da una URL

- Una volta costruita una **URL**, si possono leggere i dati a cui la **URL** fa riferimento come se fossero locali!
- Il metodo più semplice è chiamare il metodo **openStream()** della **URL**
 - Si connette alla risorsa indicata dalla URL, e restituisce un **InputStream**
- Si possono quindi leggere i dati dall'**InputStream** (come per un file o per i socket TCP)

Leggere da una URL

- I dati ottenuti sono semplicemente il contenuto del file indicato nella URL (in altre parole, *non sono interpretati*)
 - ASCII, se leggiamo ASCII
 - HTML, se leggiamo un file HTML
 - Dati binari, se accediamo una immagine
- Non sono inclusi gli header HTTP, né altre informazioni legate al protocollo


Leggere da una URL

- Un altro metodo per leggere da una URL è fornito da **openConnection()**
- Questo metodo apre una socket verso la URL specificata e restituisce un oggetto della classe **URLConnection**
- Permette di fare cose più sofisticate della **openStream()**
- Lo vedremo più in dettaglio in seguito

Leggere da una URL

- Per applicazioni non-testuali, o più sofisticate, è meglio usare il metodo **getContent()**
- Il metodo legge i dati, esamina il loro tipo, e li usa per costruire un **oggetto Java** equivalente
 - per esempio, chiamando **getContent()** sulla URL <http://www.di.unipi.it/cherub.gif> viene restituito un oggetto di classe **Image**, che poi può essere visualizzato sullo schermo, elaborato digitalmente, stampato su carta, ecc.
- È facile scrivere un Web browser in questo modo!

Tipi MIME riconosciuti



audio/aiff	Formati audio
audio/basic	
audio/wav	
audio/x-aiff	
audio/x-wav	
image/gif	Formati grafici
image/jpeg	
image/x-bitmap	
image/x-pixmap	
text/plain	Formati testo
text/*	

Esercizio 6

- Si vuole scrivere un programma **DumpURL** che, data una URL passata come argomento, stampi su video i dati riferiti da quella URL
- Notare la differenza con il client dell'Esercizio 2 sui socket TCP:
 - In quel caso, si stampavano tutti i dati scambiati sul socket, incluse le informazioni di protocollo; i dati passavano in forma codificata
 - Ora vogliamo leggere solo i dati "veri", senza header HTTP e in forma decodificata

Esercizio 6 -- soluzione

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

public class DumpURL {
    public static void main(String[] args) throws Exception {
        URL url = new URL(args[0]);
        BufferedReader in = new BufferedReader(
            new InputStreamReader(
                url.openStream()));

        String input;

        while ((input = in.readLine()) != null)
            System.out.println(input);

        in.close();
    }
}
```

Esercizio 6 -- soluzione

- Notare la differenza con il client dell'Esercizio 2 sui socket TCP:
- In quel caso, si stampavano tutti i dati scambiati sul socket, incluse le informazioni di protocollo; i dati passavano in forma codificata
- Ora leggiamo solo i dati "veri", *senza* header HTTP e in forma decodificata

DumpURL in azione

- Compiliamo **DumpURL** e proviamo ad eseguirlo con URL basate su diversi protocolli:
 - http:
 - ftp:
 - https:
 -
- Quali si riescono a leggere? Comparete i risultati con quelli ottenuti fornendo la stessa URL a un browser web (Netscape, Mozilla, Internet Explorer)

Vantaggi e svantaggi di `openStream()`

- Quello appena mostrato è il metodo più semplice per accedere ai dati di una URL
- Consente di *leggere* i dati con pochissimo codice
- Però *non consente di scriverli*: operazione che talvolta è utile, soprattutto quando si ha a che fare con i <FORM> o con script CGI
- Per operazioni più sofisticate, è necessario accedere direttamente alla *connessione* sottostante all'URL

Connessioni a URL

- La classe `URLConnection` fa al caso nostro
 - Implementa la connessione fra la nostra macchina e quella indicata dalla parte *host* di una URL
 - È usata internamente da `URL.openStream()`
- Uso tipico:

```
try {
    URL url = new URL("http://www.di.unipi.it/");
    URLConnection connessione = url.openConnection();
} catch (MalformedURLException e) {
    // errore durante new URL(...)
} catch (IOException e) {
    // errore durante openConnection(...)
}
```

Connessioni a URL

- Una volta ottenuta una `URLConnection`, la si può usare per estrarre uno stream di input e uno di output:
 - `URLConnection.getInputStream()`
 - `URLConnection.getOutputStream()`
- Lo stream di input si può usare normalmente
 - È analogo a `URL.openStream()`
- Lo stream di output richiede qualche cautela in più...
- Vediamo un esempio: **DumpURL** con `URLConnection`

DumpURL

usando `URLConnection`

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

public class DumpURL {
    public static void main(String[] args) throws Exception {
        URL url = new URL(args[0]);
        URLConnection uc = url.openConnection();
        BufferedReader in = new BufferedReader(
            new InputStreamReader(
                uc.getInputStream()));
        String input;

        while ((input = in.readLine()) != null)
            System.out.println(input);

        in.close();
    }
}
```

Aggiunto

Era:

Scrivere su una URL

- La scrittura richiede un passo in più:
 1. si crea una URL
 2. si apre la sua connessione
 3. si abilita la connessione all'output
 4. si ottiene un `OutputStream` dalla connessione
 5. si scrive su questo stream
 6. si chiude lo stream
- Per il passo 3:
 - `connessione.setDoOutput(true);`

Effetto della scrittura

- Su connessioni legate a URL con protocollo FTP, equivale a scrivere nel file indicato dall'URL
 - Naturalmente, bisogna avere i permessi giusti: il server FTP dall'altra parte farà i controlli del caso...
- Su connessioni legate a URL con protocollo HTTP, equivale a usare il metodo POST del protocollo
 - Dunque, non è possibile scrivere dati usando il metodo PUT: per questo, servono i socket diretti

Esempio: CGI

- Sul sito della Sun, c'è (c'era?) uno script CGI di test che riceve come argomento una stringa, e restituisce la stessa stringa invertita
- Normalmente, si accede agli script CGI attraverso una pagina HTML contenente un `<FORM>` che specifica lo script come propria ACTION
- Possiamo fare lo stesso in Java?

Esempio: CGI

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

public class Reverse {
    public static void main(String[] args) throws Exception {
        String daInvertire = URLEncoder.encode(args[0]);

        URL url = new URL("http://java.sun.com/cgi-bin/backwards");
        URLConnection connessione = url.openConnection();
        connessione.setDoOutput(true);

        PrintWriter out = new PrintWriter(connessione.getOutputStream());
        out.println("string=" + daInvertire);
        out.close();

        BufferedReader in = new BufferedReader(
            new InputStreamReader(
                connessione.getInputStream()));
        String risposta;
        while ((risposta = in.readLine()) != null)
            System.out.println(risposta);
        in.close();
    }
}
```

Altri metodi di URLConnection



- Oltre alla possibilità di estrarre `InputStream` e `OutputStream`, `URLConnection` offre molti metodi di utilità
- Possono servire per manipolare con precisione i trasferimenti di oggetti identificati da `URL`
- Sono tecniche avanzate...

Schema generale

1. Si crea una `URLConnection` (chiamando `openConnection()` su una `URL`)
2. Si impostano i parametri iniziali e le proprietà della connessione
3. Si effettua la connessione, chiamando `connect()`: a questo punto, l'oggetto remoto diventa accessibile
4. Ora si possono ispezionare gli header HTTP del trasferimento e il contenuto dell'oggetto remoto

Parametri iniziali

acceduti via `set/get...()`

<code>AllowUserInteraction</code>	La connessione è interattiva; ha senso, per esempio, aprire una finestra per chiedere una password
<code>DoInput</code>	È possibile leggere da questa connessione (vero per default)
<code>DoOutput</code>	È possibile scrivere su questa connessione (falso per default)
<code>IfModifiedSince</code>	Legge l'oggetto remoto solo se è stato modificato dopo il tempo indicato (1 gennaio 1970 per default)
<code>UseCaches</code>	È possibile usare tutte le cache previste dal protocollo

Header HTTP

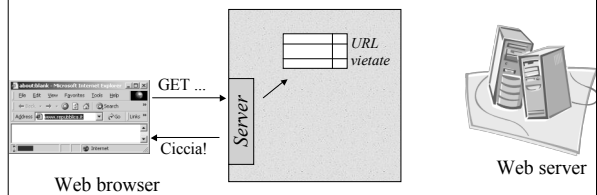
leggibili via `get...()`

<code>ContentEncoding</code>	Tipo di codifica usato per i dati
<code>ContentLength</code>	Lunghezza dei dati
<code>ContentType</code>	Tipo dei dati
<code>Date</code>	Data di creazione o ricezione della risorsa
<code>Expiration</code>	Data oltre la quale la risorsa non è più valida
<code>LastModified</code>	Data dell'ultima modifica alla risorsa
<code>getHeaderField(hdr)</code>	Accesso ad altri campi per nome o per posizione
<code>getHeaderFieldKey(i)</code>	Accesso al nome dell' <i>i</i> -esimo campo

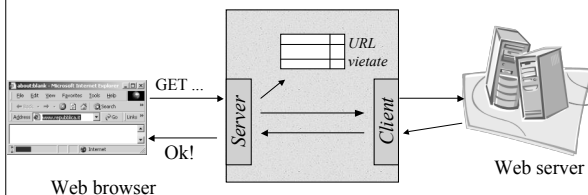
Esercizio 7 -- generale

- L'idea di base:
 - Si vuole realizzare un **proxy con filtraggio**, ovvero un proxy web che si rifiuta di passare le richieste per certe URL
 - Si può usare per varie cose:
 - proteggere i consumatori dall'eccesso di pubblicità,
 - proteggere i bimbi dai siti porno,
 - impedire ai dipendenti di perdere tempo sul sito della Gazzetta,
 - "proteggere" gli elettori dalla propaganda faziosa dell'opposizione...
- Ci concentreremo sulla rimozione dei *banner*

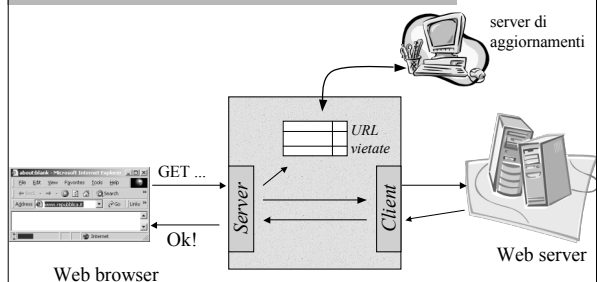
Caso base



Caso base



Caso avanzato



Soluzione

- Ci *accontenteremo* di una soluzione con queste caratteristiche:
 - Server *multi-threaded*
 - Supporta **solo** il comando **GET** (blocca **PUT** e **POST**)
 - Non supporta i trasferimenti binari (per semplicità)
 - Aggiorna la lista nera da un server di aggiornamenti
- ... facciamo tutto in maniera sempliciotta!
- HTTP introduce altre complessità
 - cache, Keep-Alive, HTTP/1.0 vs. HTTP/1.1, ...

Buon lavoro!

- ✳ L'esercizio è **difficile**
- ✳ Soluzioni imperfette vanno bene

Classi della soluzione



- **MainServer**
 - è il server principale; accetta le connessioni e lancia Server1
- **Server1**
 - tratta una singola connessione HTTP
 - controlla la URL e blocca o fa da proxy
- **ListaURL**
 - implementa la lista delle URL vietate
 - estende Vector
 - implementa l'aggiornamento remoto

MainServer.java



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

/**
 * Server principale; accetta le connessioni e fa partire i thread
 * per servire le richieste dei client.
 * @author: V. Gervasi
 */

public class MainServer {
    private ServerSocket ssocket = null;
    private ListaURL listaNera = new ListaURL();

    /**
     * Avvia il proxy server.
     */
    public static void main(java.lang.String[] args) throws IOException {
        (new MainServer()).server(Integer.parseInt(args[0]));
    }
}
```

MainServer.java



```
/**
 * Metodo principale del server. Accetta una connessione, e lancia
 * un nuovo thread su Server1, passando il socket assegnato al cliente
 * e una lista nera di URL da bloccare.
 * @exception java.io.IOException se non e' possibile creare il ServerSocket.
 */
public void server(int port) throws IOException {
    ssocket = new ServerSocket(port);
    while (true) {
        (new Server1(ssocket.accept(), listaNera)).start();
    }
}
```

Server1.java



```
package proxy;

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

/**
 * Server di una singola connessione; controlla l'URL di un comando GET
 * se e' nella lista nera, la respinge, altrimenti fa da proxy
 * @author: V. Gervasi
 */
public class Server1 extends Thread {
    private BufferedReader dalBrowser, dalServer;
    private PrintWriter alBrowser, alServer;
    private ListURL listaNera;
    private Socket browserSocket = null;
    private Socket webServerSocket = null;

    /**
     * Costruttore di Server1.
     */
    public Server1(Socket socket, ListURL lista) {
        this.browserSocket = socket;
        this.listaNera = lista;
    }
}
```

Server1.java



```
/**
 * Metodo principale del server per una singola richiesta.
 * Se il comando non e' GET, respinge la richiesta.
 * Se e' GET, ma la URL e' inclusa nella lista nera, risponde 404.
 * Altrimenti, fai da proxy fra il browser e il server.
 */
public void run() {
    try {
        dalBrowser =
            new BufferedReader(new InputStreamReader(
                browserSocket.getInputStream()));
        alBrowser = new PrintWriter(browserSocket.getOutputStream());
        String primalinea = dalBrowser.readLine();
        StringTokenizer st = new StringTokenizer(primalinea);
        String cmd = st.nextToken();
        String url = st.nextToken();
        String protocollo = st.nextToken();
        URL u = new URL(url);

        if (!cmd.equals("GET"))
            alBrowser.println("HTTP/1.0 400 Blocked by Proxy");
    }
}
```

Server1.java



```
segue run()
{
    if (!cmd.equals("GET"))
        alBrowser.println("HTTP/1.0 400 Blocked by Proxy");
    if (listaNera.contiene(u)) {
        alBrowser.println("HTTP/1.0 404 File Not Found");
    }
    else {
        String server = u.getHost();
        int port = u.getPort() > 0 ? u.getPort() : 80;
        webServerSocket = new Socket(server, port);
        dalServer =
            new BufferedReader(new InputStreamReader(
                webServerSocket.getInputStream()));
        alServer =
            new PrintWriter(webServerSocket.getOutputStream());
        primalinea = cmd + " " + u.getFile() + " " + protocollo;
        alServer.println(primalinea);
        copiaTutto();

        alBrowser.flush();
    }
    catch (IOException e) {
        System.err.println("Errore: " + e);
    }
}
```

Server1.java



```
segue run()
{
    catch (IOException e) {
        System.err.println("Errore: " + e);
    } finally {
        try {
            if (alServer != null) alServer.close();
            if (dalServer != null) dalServer.close();
            if (webServerSocket != null) webServerSocket.close();
            if (alBrowser != null) alBrowser.close();
            if (dalBrowser != null) dalBrowser.close();
            if (browserSocket != null) browserSocket.close();
        } catch (Exception e) {
            System.err.println("Errore nella chiusura: " + e);
        }
    }
}
```

Server1.java



```
/**
 * Copia il resto dello stream HTTP dalBrowser->alServer
 * poi copia la risposta dalServer->alBrowser
 */

public void copiaTutto() throws IOException {
    String linea;
    while ((linea = dalBrowser.readLine()) != null) {
        alServer.println(linea);
        if (linea.length() < 1) break;
    }
    alServer.flush();

    // NOTA: questo codice funziona per HTML, ma non per il
    // trasferimento di dati binari, come le immagini.

    while ((linea = dalServer.readLine()) != null) {
        alBrowser.println(linea);
        alBrowser.flush();
    }
}
```

ListaURL.java



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

/**
 * Lista nera di URL; supporta l'aggiornamento da un server remoto
 * @author: V. Gervasi
 */
public class ListaURL extends Vector {
    /**
     * Verifica se un prefisso di una certa URL compare nella lista nera
     * @return boolean vero se la URL (o un suo prefisso) e' nella lista nera
     * @param u java.net.URL la URL da controllare
     */
    public boolean contiene(URL u) {
        String url = u.toString();
        for (int i = 0; i < size(); i++)
            if (url.startsWith((String)elementAt(i)))
                return true;
        return false;
    }
}
```

ListaURL.java



```
/**
 * Si collega al server degli aggiornamenti e scarica la lista
 * nera aggiornata.
 */

public void aggiorna() {
    try {
        Socket sock = new Socket("aggiornamenti.filterproxy.it", 8020);
        BufferedReader in =
            new BufferedReader(
                new InputStreamReader(sock.getInputStream()));
        clear();
        String linea;
        while ((linea = in.readLine()) != null)
            add(linea);
    } catch (IOException e) {
        System.err.println("Impossibile aggiornare la lista nera: " + e);
    }
}
```

