Università degli Studi di Pisa
Dipartimento di Informatica

Laboratorio di Reti
Lezione 6

JAVA.NET:
indirizzi IP, stream sockets

19/10/2021

Federica Paganelli

NETWORK APPLICATIONS

applicazioni pervasive e di grande diffusione:

- Web browsers
- SSH
- email
- Social networks
- P2P File sharing: Bittorrent
- program development environments: GIT
- collaborative work: OverLeaf
- multiplayer games: War of Warcraft
- cryptocurrencies: Bitcoin
- e-commerce
- POS terminals

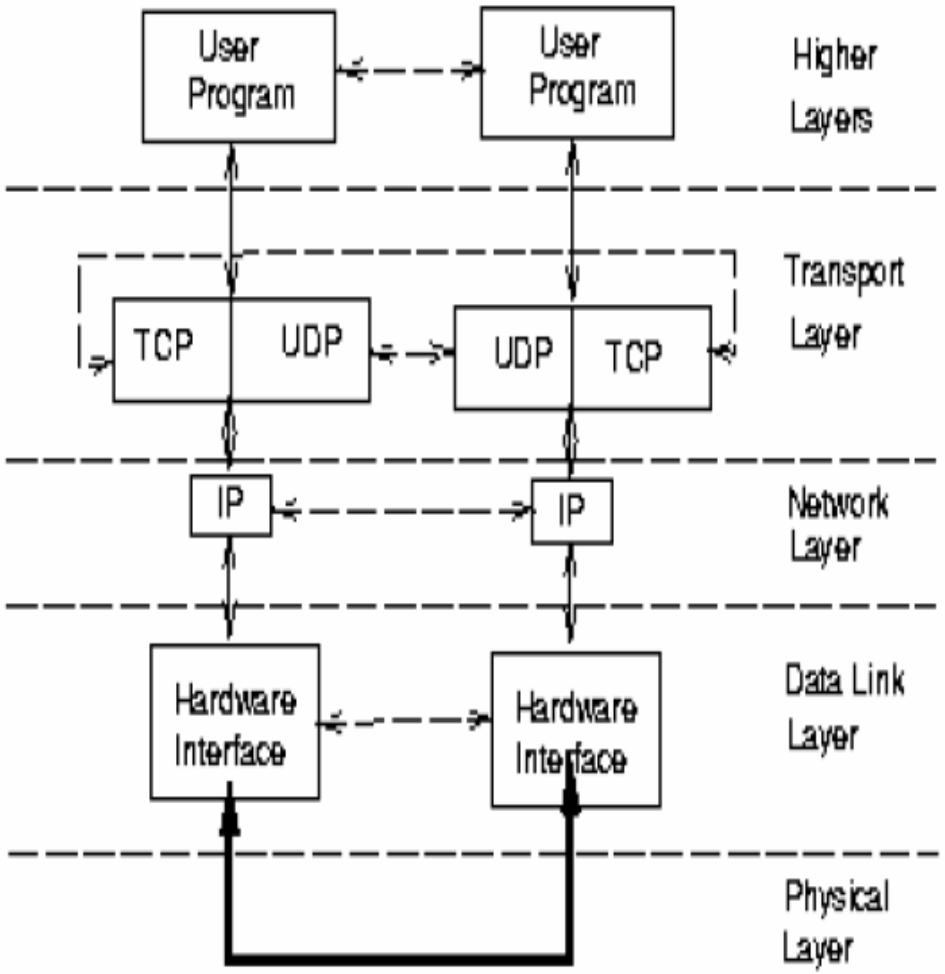
Scopo del corso è mettervi in grado di sviluppare una semplice applicazione di rete.

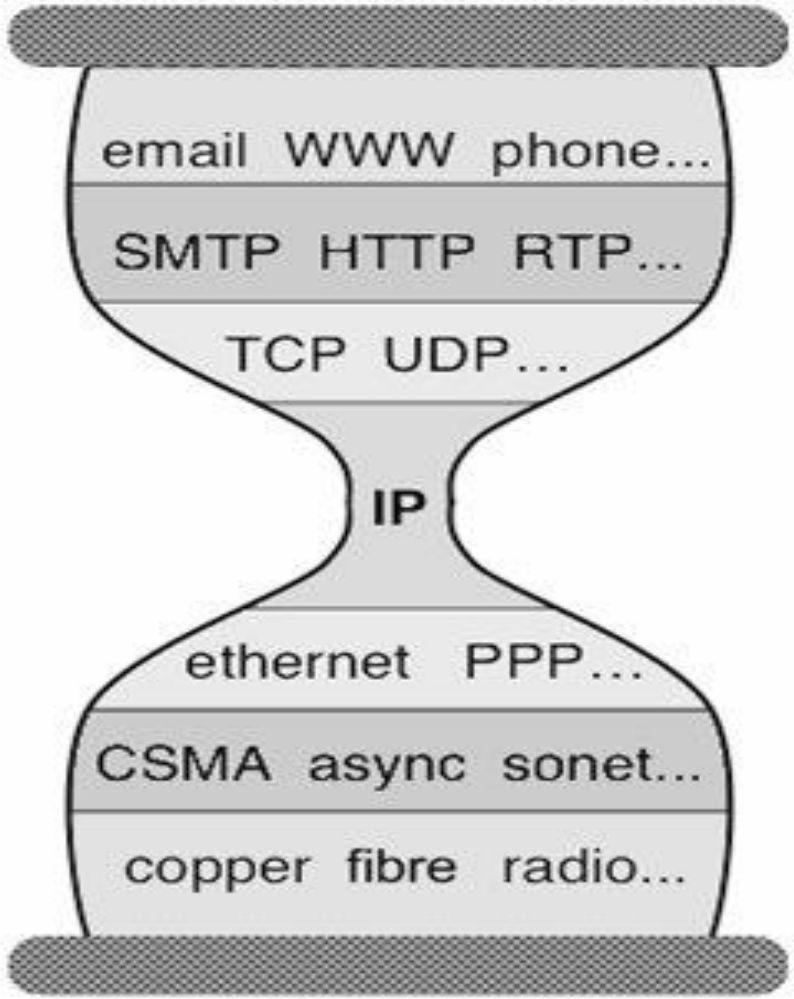
NETWORK APPLICATIONS

- In una applicazione di rete:
  - due o più processi (non thread!) in esecuzione su hosts diversi, distribuiti geograficamente sulla rete, comunicano e cooperano per realizzare una funzionalità globale
  - cooperazione: scambio informazioni utile per perseguire l'obiettivo globale, quindi implica comunicazione
  - comunicazione: utilizza protocolli, ovvero insieme di regole che i partners devono seguire per comunicare correttamente.

- in questo corso utilizzeremo i protocolli di livello trasporto:
  - `connection-oriented`: `TCP, Trasmission Control Protocol`
  - `connectionless`: `UDP, User Datagram Protocol`

ARCHITETTURA E LAYERS





SOCKET: UNO “STANDARD” DI COMUNICAZIONE

socket: uno standard per connettere dispositivi distribuiti, diversi, eterogenei

- termine utilizzato in tempi remoti in telefonia.
- la connessione tra due utenti stabilita tramite un operatore che inseriva fisicamente i due estremi di un cavo in due ricettacoli (sockets), ognuno dei quali era assegnato ai due utenti.

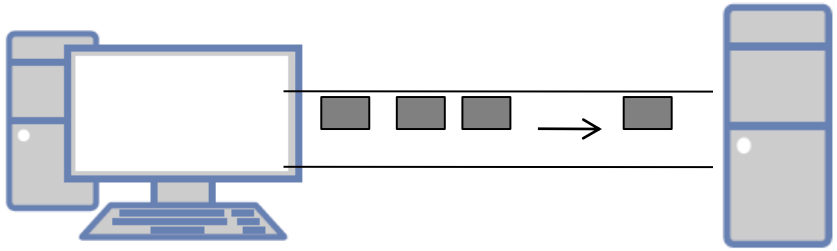
SOCKET: UNO “STANDARD” DI COMUNICAZIONE

- un modo di standardizzare la comunicazione: “presa” a cui un processo si può collegare per spedire dati
  - un endpoint sull'host locale di una canale di comunicazione da/verso altri hosts
  - creato su richiesta dell’applicazione e controllato dal SO dell’host locale
- Un processo applicativo può sia spedire che ricevere messaggi a/da un altro processo applicativo mediante la propria socket
- introdotta in Unix BSD 4.2

TIPI DI COMUNICAZIONE

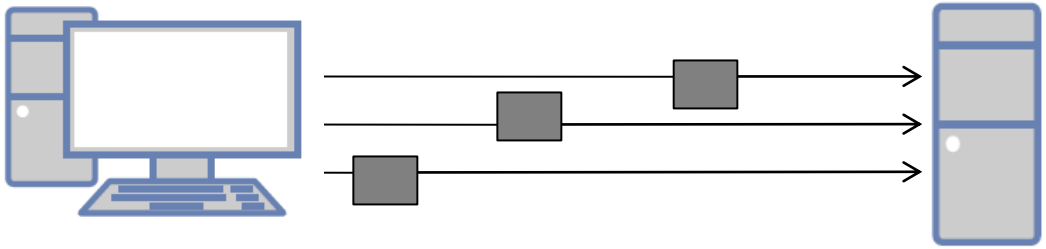
Connection Oriented

- analogia: chiamata telefonica
- creazione di una connessione logica end-to-end tra mittente e destinatario
- Tre fasi: Apertura della connessione, invio dei dati sulla connessione, chiusura
- stream socket
- esempio: Web



Connectionless

- analogia: invio di lettere
- non si stabilisce un canale di comunicazione logico dedicato
- one-way message: ogni messaggio viene instradato in modo indipendente dagli altri
- datagramsocket
- esempio: richieste DNS



CONNECTION ORIENTED/LESS

Indirizzamento (a livello di applicativo software)

- *Connection Oriented*: l'indirizzo del destinatario (indirizzo IP+ porta) è specificato al momento della apertura connessione
- *Connectionless*: l'indirizzo del destinatario (indirizzo IP + porta) viene specificato in ogni pacchetto (per ogni send)

Ordinamento dei dati scambiati

- *Connection Oriented*: garantito ordinamento dei messaggi da parte del protocollo di trasporto
- *Connectionless*: nessuna garanzia sull'ordinamento dei messaggi da parte del protocollo di trasporto

Utilizzo

- *Connection Oriente*d: è richiesta la trasmissione affidabile dei dati
- *Connectionless*: invio di un numero limitato di dati/non è importante che l'invio sia affidabile

SOCKET CLASSES IN JAVA

connection-oriented (protocollo TCP)

- si sfruttano gli stream per modellare la connessione
- Client side: `Socket class`
- Server side:
  - `ServerSocket class`
  - `Socket class`

Perché servono due classi diverse per il server?

connectionless (protocollo UDP)

- nessuna garanzia su consegna dei dati/ordinamento
- `DatagramSocket class` sia per il client che per il server

Indirizzamento a livello di processo

- la rete all'interno della quale si trova l'host su cui è in esecuzione il processo e l'host all'interno della rete
    - Identificativo a livello di Internet Protocol, mediante indirizzi IP
- il processo in esecuzione sull'host
    - al processo viene assegnata una porta: intero da 0 a 65535
- ogni «comunicazione» è quindi individuata dalla seguente 5-upla:
    - il protocollo (TCP o UDP)
    - l'indirizzo IP del computer locale (client *sky3.cm.deakin.edu.au, 139.130.118.5)*
    - la porta locale esempio: 5101
    - l'indirizzo del computer remoto (server *res.cm.deakin.edu.au* 139.130.118.102),
    - la porta remota: 5100

{tcp,139.130.118.102,5100,139.130.118.5,5101}

INDIRIZZI IP

un indirizzo IPV4

10010001	00001010	00100010	00000011

145. 10. 34. 3

- 4 bytes: ognuno interpretato come un numero decimale senza segno
- valore di ogni byte: `0..255`
- 2^{32} indirizzi, ma stanno per esaurirsi....
- address special blocks in IPV4:
  - loopback address: 127.0.0.1
  - 255.255.255.255: broadcast

un indirizzo IPV6

FE80:0000:0000:0000:02A0:24FF:FE77:4997

- 16 bytes, 2^{128} indirizzi

INDIRIZZI IP E NOMI

- gli indirizzi IP semplificano l'elaborazione effettuata dai routers, ma sono poco leggibili per gli utenti della rete

  -> identificativo di livello 3

- Identificativi di risorse a livello applicativo: nome simbolico ad ogni host della rete
  - spazio di nomi gerarchico

fujih0.cli.di.unipi.it

host fuji presente nell'aula H alla postazione 0, nel dominio cli.di.unipi.it

  - Livelli della gerarchia separati dal punto.
  - nomi interpretati da destra a sinistra (dal più specifico al meno specifico)

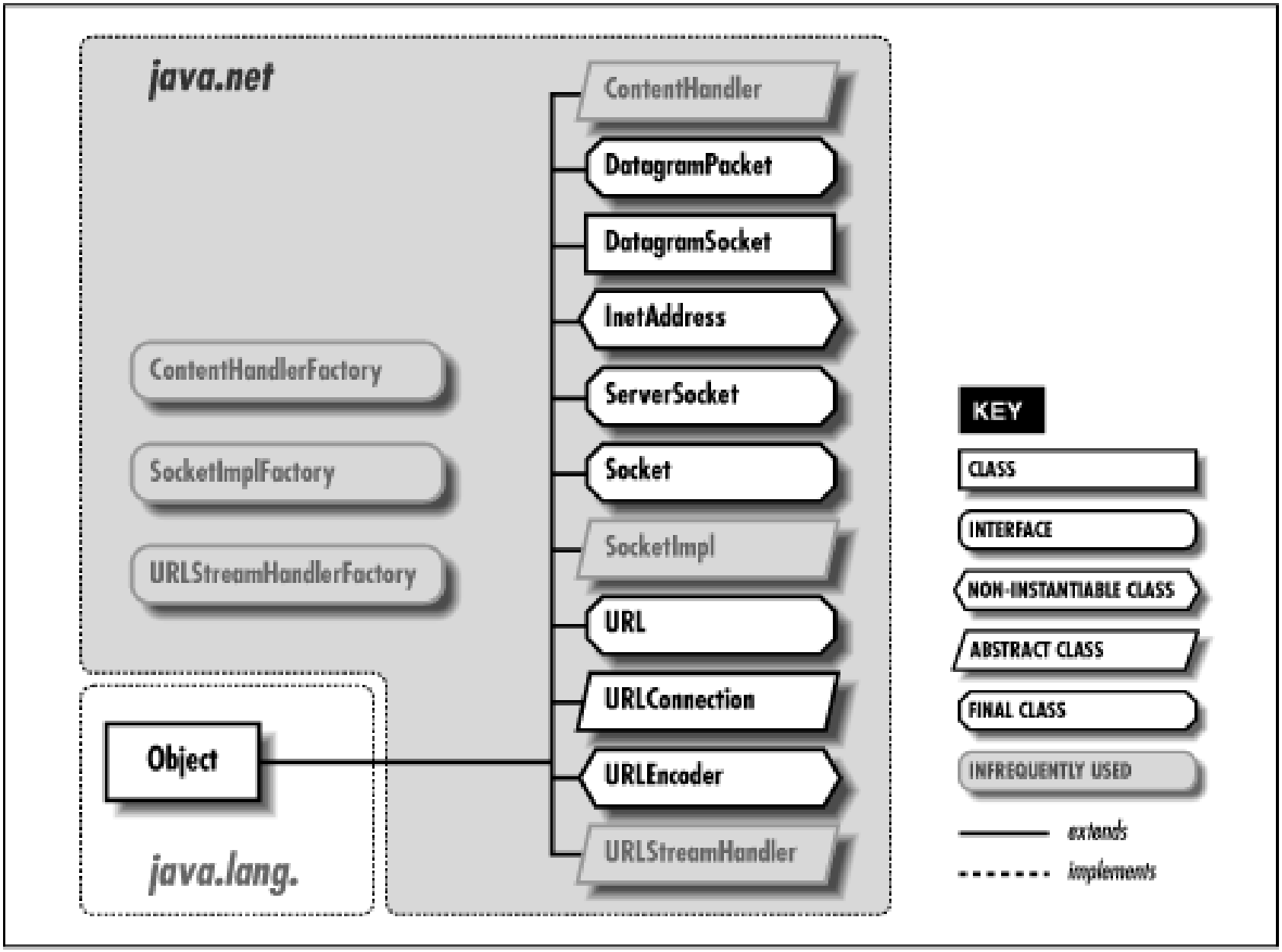
- Domain Name System (DNS) traduce nomi in indirizzi IP

INDIRIZZAMENTO A LIVELLO DI PROCESSI

- Su ogni host possono essere attivi contemporaneamente più servizi (es: `e-mail`, `ftp`, `http`,…)
  - ogni servizio viene incapsulato in un diverso processo
  - Al servizio è assegnato un numero di porta
  - intero compreso tra 1 e 65535 (per ogni protocollo di trasporto)
  - non un dispositivo fisico, ma un'astrazione

- Porte 1–1023: riservate per *well-known* services
  - servizio HTTP in genere sulla porta 80.
  - Unix hosts: solo root process possono ascoltare su queste porte
  - Usare valori di porta > 1024

Java.net classes



InetAddress e NetworkInterface

LA CLASSE INETADDRESS

- Rappresentazione di alto livello di un indirizzo IP (IPv4 e IPV6)
- utilizzata dalle altre classi di networking (`Socket, ServerSocket, URL, …`)
- rappresenta gli indirizzi Internet come
  - `String hostName`: una stringa che rappresenta il nome simbolico di un host
  - `byte[] address`: raw IP address
  - non definisce costruttori, ma fornisce tre metodi statici per costruire oggetti di tipo `InetAddress` (anche utilizzando il servizio DNS)

```
public static InetAddress.getByName (String hostname)
                                        throws UnKnownHostException

public static InetAddress.getAllByName(String hostname)
                                        throws UnKnownHostException

public static InetAddress.getLocalHost() throws UnKnownHostException
```

LA CLASSE INETADDRESS

```
public static InetAddress getByName (String hostname)
   throws UnKnownHostException
```

- cerca l'indirizzo IP corrispondente all'host di nome hostname e restituisce un oggetto di tipo `InetAddress`
- richiede una interrogazione del DNS per risolvere il nome dell'host

```
InetAddress addr1 =
 InetAddress.getByName("www.codejava.net");
System.out.println(address1.getHostAddress());
```

- può sollevare una eccezione (`UnknownHostException`, `sottoclasse di IOException`) se non riesce a risolvere il nome dell'host

LA CLASSE INETADDRESS

public static InetAddress[] *getAllByName (String hostname)* **throws** *UnKnownHostException*

- per hosts che possiedono piu indirizzi (es: web servers)

public static InetAddress *getLocalHost()* **throws** *UnKnownHostException*

- reperire nome simbolico ed indirizzo IP del computer locale

metodi getter: reperire i campi di un oggetto di tipo `InetAddress` (non effettuano collegamenti con il DNS, non sollevano eccezioni)

```
public String  getHostName ( )
public byte[] getAddress ( )
public String  getHostAddress ( )
```

LA CLASSE INETADDRESS

```
import java.net.InetAddress;
import java.net.UnknownHostException;
public class GetIpAddress {
  public static void main(String[] args) throws
                                              UnknownHostException {
   // print the IP Address of your machine (inside your local network)
   System.out.println(InetAddress.getLocalHost().getHostAddress());
                  // print the IP Address of a web site
   System.out.println(InetAddress.getByName("www.java.com"));
   // print all the IP Addresses that are assigned to a certain domain
   InetAddress[]inetAddresses=InetAddress
                                .getAllByName("www.repubblica.it");
   //System.out.println(inetAddresses);
   for (InetAddress ipAddress : inetAddresses)
        System.out.println(ipAddress);
  }
}
```

LA CLASSE INETADDRESS

OUTPUT DEL PROGRAMMA

```
192.168.1.10
www.java.com/72.247.209.10
www.repubblica.it/213.92.16.191
www.repubblica.it/213.92.16.171
```

OVERRIDING OBJECT METHODS

- come tutte le classi, la classe `InetAddress` eredita da `java.lang.Object`
- effettua overriding dei 3 metodi base di `Object`
  - `equals()`: un oggetto `o` è uguale ad un oggetto `InetAddress ia` se e solo se
    - `o` è un oggetto `InetAddress` e rappresenta lo stesso indirizzo IP, ovvero:
      - i byte array restituiti dalla `getAddress()` per `o` ed `ia` hanno la stessa lunghezza e le componenti, byte a byte, sono uguali.
      - Non è richiesto che abbiano lo stesso hostname!!!
  - `hashCode()`: calcola l'hash dai 4 bytes dell'indirizzo e restituisce un int
  - `toString()`: restituisce *<nome dell'host/indirizzo dotted quad>*, se non esiste il nome, stringa vuota + indirizzo dotted quad

CACHING

- i metodi descritti effettuano caching dei nomi/indirizzi risolti, poiché l'accesso al DNS è una operazione potenzialmente molto costosa

- nella cache vengono memorizzati anche i tentativi di risoluzione non andati a buon fine
  - validi per un intervallo breve (tipicamente 10 secondi)

- `java.security.Security.setProperty` consente di impostare il numero di secondi in cui una entrata nella cache rimane valida, tramite le proprietà
  - `networkaddress.cache.ttl`
  - `networkaddress.cache.negative.ttl`

esempio

```
java.security.Security.setProperty("networkaddress.cache.ttl", "0");
```

Caching

```
import java.net.InetAddress; import java.net.SocketException;
import java.net.UnknownHostException; import java.security.*;

public class InetAddressCacheUsage {
 public static void main(String[] args) throws UnknownHostException,
                                                SocketException {
     final String CACHINGTIME="0";
     Security.setProperty("networkaddress.cache.ttl",CACHINGTIME);
     long time1 = System.currentTimeMillis();
     for (int i=0; i<1000; i++) {
         try {
             System.out.println( i + " "+
                     InetAddress.getByName("www.cnn.com").getHostAddress());}
         catch (UnknownHostException uhe) {
             System.out.println("UHE"); }
         }
     long diff=System.currentTimeMillis();
     System.out.println("tempo trascorso e'"+diff);
     }
}
```

CACHING

Output prodotto dal programma:

```
CACHINGTIME=0
tempo trascorso è 436

CACHINGTIME=1000
tempo trascorso è 50
```

Chiaramente visibili gli effetti del caching sulle prestazioni del programma

NETWORK INTERFACE

Ogni host di una rete IPV4 o IPV6 è connesso alla rete mediante una o più interfacce

- ogni interfaccia è caratterizzata da un indirizzo IP
- può essere un'interfaccia virtuale non associata direttamente ad un dispositivo fisico

se un host, ad esempio un router, presenta più di una interfaccia sulla rete, allora si hanno più indirizzi IP per lo stesso host, uno per ogni interfaccia

multi-homed hosts: un host che possiede un insieme di interfacce verso la rete, e quindi un insieme di indirizzi IP

- esempio: router

```
public static NetworkInterface getByAddress (InetAddress address)
throws SocketException
```

restituisce un oggetto `NetworkInterface` che rappresenta la network interface collegata ad un indirizzo IP (o null)

Enumerare le interfacce

```
Enumeration<NetworkInterface> nets =
NetworkInterface.getNetworkInterfaces();

for (NetworkInterface netint : Collections.list(nets)) {

        System.out.printf("*****\n");

        System.out.printf("Display name: %s\n",
                                        netint.getDisplayName());

        System.out.printf("Name: %s\n", netint.getName());

}
```

OUTPUT

```
*****
Display name: Software Loopback Interface 1
Name: lo
*****
Display name: Microsoft 6to4 Adapter
Name: net0
*****
Display name: Killer Wireless-n/a/ac 1535 Wireless Network Adapter
Name: wlan1
*****
```

Indirizzo di Loopback

- **`127.0.0.1` Indirizzo di Loopback**
- può essere utilizzato da un nodo solo per inviare i pacchetti a se stesso.
- utilizzabile per testare applicazioni
- quando si usa un indirizzo di loopback si possano eseguire client e server in locale, sullo stesso host
- ogni dato spedito utilizzando l'indirizzo di loopback in realtà non lascia l'host locale

- il dato viene restituito all'host locale stesso, sulla porta opportuna

Indirizzo di Loopback

```
import java.net.*;
import java.util.*;
public class InterfaceListener {
public static void main(String[] args) throws Exception {
    try {
      InetAddress local = InetAddress.getByName("127.0.0.1");
      NetworkInterface ni = NetworkInterface.getByInetAddress(local);
      if (ni == null) { System.err.println(
                        "That's weird. No local loopback address.");
        }
      else System.out.println(ni); }
    catch (UnknownHostException ex) {
    System.err.println("That's weird. No local loopback address.");
     }
  }
}
```

```
name:lo (Software Loopback Interface 1)
```

TCP socket programming

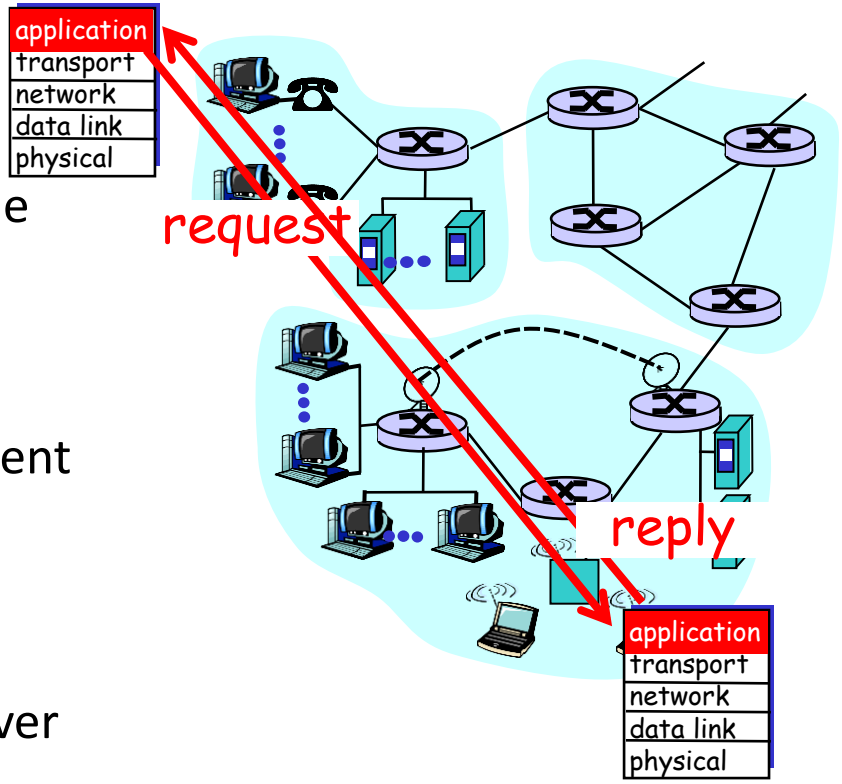
IL PARADIGMA CLIENT/SERVER

Servizio

- software in esecuzione su una o più macchine.
- fornisce l'astrazione di un insieme di operazioni

Server

- istanza di un particolare servizio in esecuzione su un host
- ad esempio: server Web invia la pagina web
- richiesta, mail server consegna la posta al client



Client

- Un software che sfrutta servizi forniti dal server

web client = browser

e-mail client = mail-reader

TCP SOCKET PROGRAMMING

Il client

contatta il server che deve essere in esecuzione al momento della richiesta di contatto

crea un TCP socket specificando l'indirizzo IP e la porta associata al processo (servizio) in esecuzione sul Server

quando il client crea il socket, viene inviata una richiesta di connessione con il server

Il server

deve aver creato un welcoming socket che consente di ricevere la richiesta di contatto del client

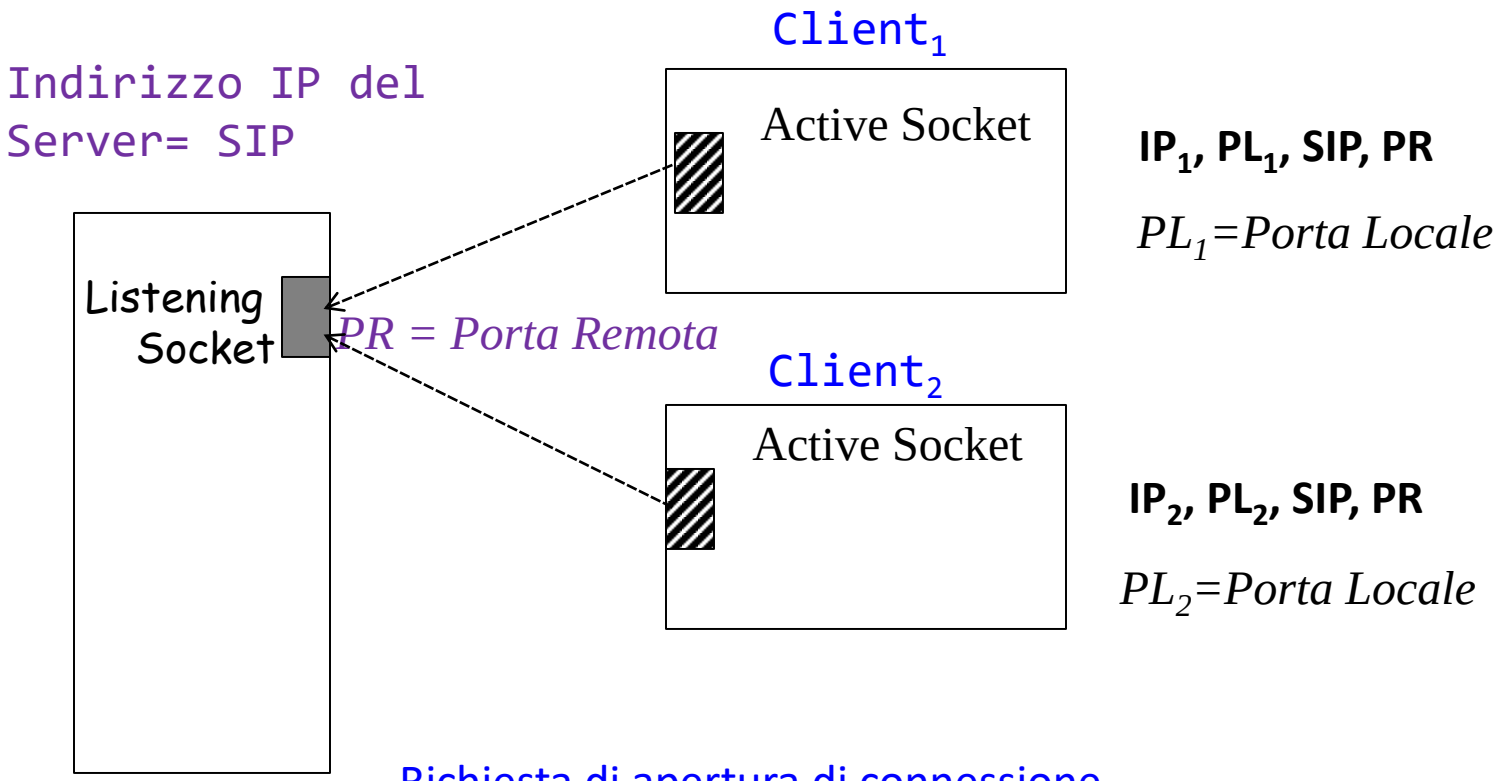
quando accetta la richiesta di connessione crea un nuovo connection socket per comunicare con quel client

IL PROTOCOLLO TCP: JAVA STREAM SOCKET

- il client crea una socket per richiedere la connessione
- Lato server esistono due tipi di socket TCP:
  - **welcome (passive, listening) sockets**: utilizzati dal server per accettare le richieste di connessione
  - **connection (active) sockets**: supportano lo streaming di byte tra client e server

- Il server ha una welcome socket attiva per stare in ascolto
- quando il server accetta una richiesta di connessione,
  - crea a sua volta un proprio active socket che rappresenta il punto terminale della sua connessione con il client
  - la comunicazione vera e propria avviene mediante la coppia di active socket presenti nel client e nel server

IL PROTOCOLLO TCP: JAVA STREAM SOCKET

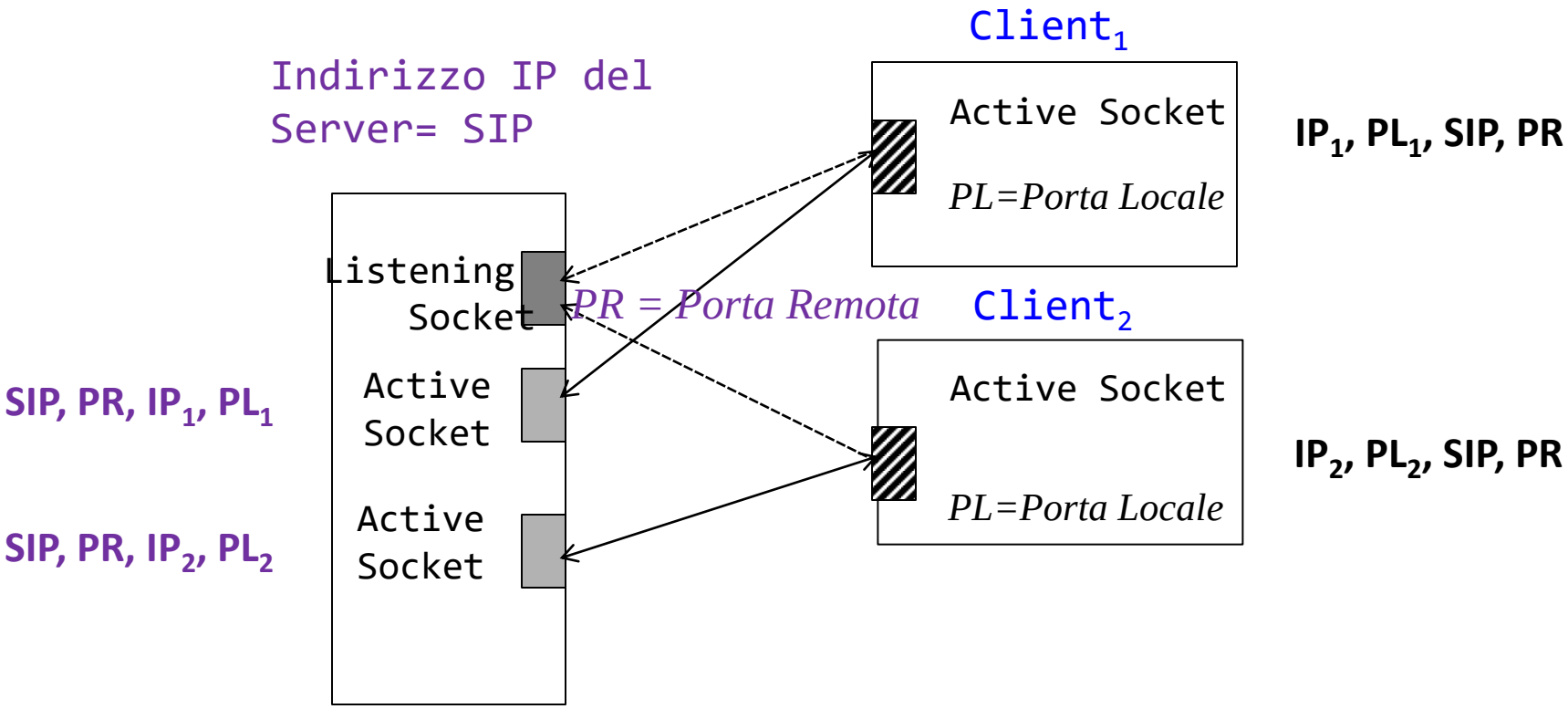


Richiesta di apertura di connessione.

i) Il client crea una active socket S

ii) lo associa alla sua porta locale PL e (binding) alla listening socket presente sul server pubblicato all'indirizzo (SIP,PR)

IL PROTOCOLLO TCP: JAVA STREAM SOCKET



Apertura di una connessione

il server accetta la richiesta di connessione e crea una propria active socket che rappresenta il suo punto terminale della connessione

JAVA SOCKETS

- Il server pubblica un proprio servizio associandolo alla listening socket, creata sulla porta remota PR
- il client C che intende usufruire del servizio deve conoscere l'indirizzo IP del server, SIP, ed il riferimento alla porta remota, PR, a cui è associato il servizio
- la richiesta di creazione della socket produce la richiesta di connessione al server
  - il protocollo di richiesta della connessione viene completamente gestito dal supporto
  - quando la richiesta di connessione viene accettata dal server, il supporto in esecuzione sul server crea in modo automatico una nuova active socket AS.
  - Questa AS è utilizzato per l'interazione con il client. Tutti i messaggi spediti dal client vengono diretti automaticamente sulla nuova socket creata.

TCP connection

Passi:

- Il *server* apre una `ServerSocket` e aspetta connessioni dai clienti
- Il cliente apre una `Socket` e si connette al *server*
- Quando arriva una richiesta da un client, il *server* crea una `Socket` dedicato a questo client
- *Server* e client comunicano in base al protocollo stabilito, usando sempre `Socket`
- `ServerSocket` serve solo per aspettare clienti e iniziare la connessione (3 way handshake)

JAVA SOCKET API: LATO CLIENT

java.net.Socket

costruttori

```
public Socket(InetAddress host, int port) throws
   IOException
```

crea una active socket e tenta di stabilire, tramite essa, una connessione con l'host individuato da `InetAddress`, sulla porta `port`.

Se la connessione viene rifiutata, lancia una eccezione di IO

```
public Socket (String host, int port) throws IOException
```

host può essere individuato dal suo nome simbolico `host`

viene interrogato automaticamente il DNS

JAVA STREAM SOCKET API: LATO CLIENT

Altri costruttori della Classe `java.net.socket`

```
public Socket (String host, int port, InetAddress
   localIA, int localPort)
```

```
public Socket (InetAddress host, int Port, InetAddress
   localIA, int localPort)
```

tenta di creare una connessione

- verso *host*,
- sulla porta *port*.
- dalla interfaccia locale *localIA*
- dalla porta locale *localPort*

PORT SCANNER: INDIVIDUAZIONE SERVIZI SU SERVER

```
import java.net.*;  import java.io.*;
public class PortScanner {
 public static void main(String args[]) {
   String host;
  try { host = args[0]; }
  catch (ArrayIndexOutOfBoundsException e) {
    host="localhost";}
  for (int i = 1; i< 1024; i++) {
    try {
        Socket s = new Socket(host, i);
        System.out.println("Esiste un servizio sulla porta "+i);}
        catch (UnknownHostException ex) {
          System.out.println("Host Sconosciuto"); break; }
        catch (IOException ex)  {
            System.out.println("Non esiste un servizio sulla porta
  "+i);
         }  }}}
```

PORT SCANNER: ANALISI

- il client richiede la connessione tentando di creare un socket su ognuna delle prime 1024 porte di un host
- nel caso in cui non vi sia alcun servizio attivo, il socket non viene creato e viene invece sollevata un'eccezione

- per ottimizzare il comportamento del programma, utilizzare il costruttore

```
public Socket(InetAddress host, int port) throws IOException
```

- viene utilizzato **`InetAddress`** invece del nome dell'host per costruire i sockets
- Cache locale dei nomi

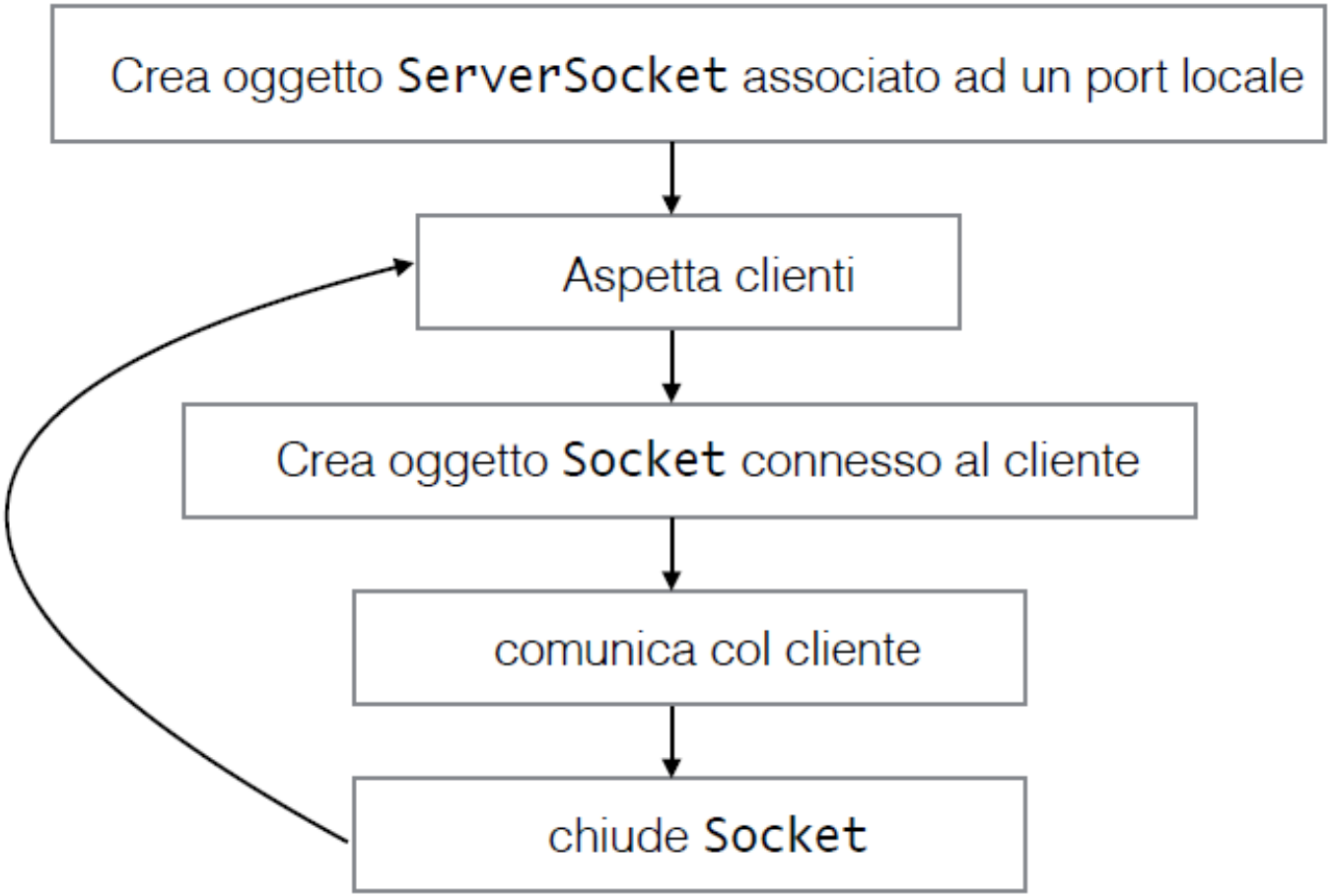
Example

```
public class SocketProps {
public static void main(String[] args) {
                                          // crea una socket non connessa
 try (Socket socket = new Socket()) {
     SocketAddress address = new InetSocketAddress("www.google.com", 80);
     socket.connect(address);   // connette la socket ad un server
     System.out.println("Connected to " + socket.getInetAddress()
         + " on port " + socket.getPort() + " from port "
         + socket.getLocalPort() + " of " + socket.getLocalAddress());
 }
 catch (UnknownHostException ex) {
     System.err.println("I can't find the host ");}
 catch (IOException ex) {
     System.err.println(ex);
  }
 }
}
```

ServerSocket

- Una *socket* speciale che aspetta delle connessioni
    - Non sa in anticipo chi sono i clienti e quando arriveranno
- il ServerSocket esiste ed è attivo anche senza nessun cliente
- i ServerSocket non sono usati per trasmettere dati all'applicazione, servono solo ad aspettare, negoziare la connessione con i nuovi clienti e creare una Socket normale da usare nell'applicazione.

ServerSocket workflow



JAVA STREAM SOCKET API: LATO SERVER

java.net.ServerSocket: costruttori

```
public ServerSocket(int port) throws BindException, IOException

public ServerSocket(int port, int length) throws BindException,
                                                  IOException
```

- costruisce un listening socket, associandolo alla porta port
- length: lunghezza della coda in cui vengono memorizzate le richieste di connessione. Se la coda è piena, ulteriori richieste di connessione sono rifiutate. La semantica esatta dipende dall'implementazione.

```
public ServerSocket(int port, int length, Inetaddress bindAddress)
```

…

- permette di collegare il socket ad uno specifico indirizzo IP locale.
- utile per macchine dotate di più schede di rete, ad esempio un host con due indirizzi IP, uno visibile da Internet, l'altro visibile solo a livello di rete locale.
- Se voglio servire solo le richieste in arrivo dalla rete locale, associo il connection socket all'indirizzo IP locale

JAVA STREAM SOCKET API: LATO SERVER

accettare una nuova connessione dal connection socket

```
public Socket accept() throws IOException
```

metodo della classe **`ServerSocket`**.

Comportamento:

- quando il processo server invoca il metodo `accept()`, pone il server in attesa di nuove connessioni.
- se non ci sono richieste, il server si blocca (possibile utilizzo di timeout)
- All'arrivo di una richiesta, il processo si sblocca e costruisce un nuovo socket S tramite cui avviene la comunicazione effettiva tra client e server

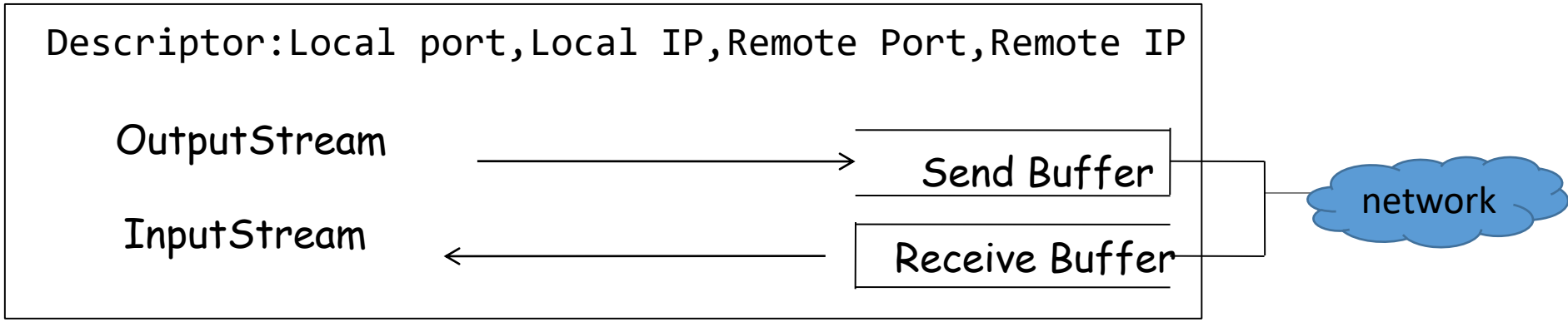
PORT SCANNER su un host

Esempio: ricerca dei servizi attivi sull'host locale

```
import java.net.*;
public class SocketLauncher {
    public static void main(String args[])
    for (int port= 1; port<= 1024; port++) {
        try   {
          ServerSocket server = new ServerSocket(port);
        }
          catch (BindException ex) {
             System.out.println(port + "occupata");}
         catch (Exception ex) {System.out.println(ex);}
    }
}
```

MODELLARE UNA CONNESSIONE MEDIANTE STREAM

- Dopo che la richiesta di connessione viene accettata, client e server associano stream di byte di input/output all'active socket
  - poiché gli stream sono unidirezionali si hanno uno stream di input ed uno di output per lo stesso socket
  - la comunicazione avviene mediante lettura/scrittura di dati sullo stream
  - eventuale utilizzo di filtri associati agli stream



Struttura del Socket TCP

SendBuffer e Receive buffer in figura rappresentano le implementazioni dei buffer TCP nello stack di rete del S.O.

MODELLARE UNA CONNESSIONE MEDIANTE STREAM

Connessioni modellate come stream di input o di output

Per associare uno stream di input o di output ad un socket:

```
public InputStream getInputStream() throws IOException
Public OutputStream getOutputStream() throws IOException
```

- ogni valore scritto sullo stream di output associato al socket viene quindi copiato nel `Send Buffer` del TCP
- ogni valore letto dallo stream viene prelevato dal `Receive Buffer` del TCP
- invio di dati: client/server leggono/scrivono dallo/sullo stream
  - un byte/una sequenza di bytes
  - dati strutturati/oggetti. In questo caso è necessario associare dei filtri agli stream

CICLO DI VITA TIPICO DI UN SERVER

```
// instantiate the ServerSocket
ServerSocket servSock = new ServerSocket(port);
 while (! done) // oppure while(true) {
     // accept the incoming connection
     Socket sock = servSock.accept();
    // ServerSocket is connected ... talk via sock
     InputStream in = sock.getInputStream();
     OutputStream out = sock.getOutputStream();

     //client and server communicate via in and out
     sock.close();
 }
servSock.close();
```

Esempio: echoserver e echoclient

EchoServer

```
public class EchoServer {

public static void main(String[] args) {

try (ServerSocket server= new ServerSocket();) {

        server.bind(new InetSocketAddress(InetAddress.getLocalHost(),
                                                          1500));

        while (true){

            System.out.println("Waiting for clients");

            String message;

            try (Socket client=server.accept();

                BufferedReader reader=new BufferedReader(new
                InputStreamReader(client.getInputStream()));

                BufferedWriter writer= new BufferedWriter(new
                OutputStreamWriter(client.getOutputStream()));){
```

EchoServer

```
…
   while ((message= reader.readLine() )!=null){
        System.out.println("Client sent: "+message);
        writer.write(message+"\r\n");
        //writer.newLine();
        writer.flush(); } }
   catch (IOException e){
        System.out.println("Client closed connection or
some error     appeared");
        } } }
catch (UnknownHostException e) {
        e.printStackTrace();
}
catch (IOException e) {
e.printStackTrace(); }}}
```

EchoClient

```
import java.io.BufferedReader;

public class EchoClient {
    public static void main(String[] args) {
        Socket socket=new Socket();
        BufferedReader reader=null;
        BufferedWriter writer=null;
        try{
            socket.connect(new InetSocketAddress(
                    InetAddress.getLocalHost(),1500));
            reader= new BufferedReader(new InputStreamReader(
                    socket.getInputStream()));
            writer= new BufferedWriter(new OutputStreamWriter(
                    socket.getOutputStream()));

            BufferedReader localReader= new BufferedReader(
                    new InputStreamReader(System.in));

            System.out.println("Type 'exit' to stop, any message to send to server:")

            String reply="";
            String choice;
```

EchoClient

```
            while (!(choice= localReader.readLine().trim()).equals("exit"))
            {
                writer.write(choice+"\r\n");
                writer.flush();
                reply= reader.readLine();
                System.out.println("Server sent back: "+reply+"");
            }

            socket.close();

        }
        catch (SocketException e) {
            System.out.println("Server closed connection or an error appeared.");
        } catch (UnknownHostException e) {
            e.printStackTrace();
        } catch (IOException e) {
            System.out.println("Server closed connection or an error appeared.");
        }
    }
}
```

HALF CLOSED SOCKETS

- `close( )`: chiusura del socket in entrambe le direzioni
- half closure: chiusura del socket in una sola direzione
  - `shutdownInput()`
  - `shutdownOutput()`
- in molti protocolli: il client manda una richiesta al server e poi attende la risposta

```
try ( Socket connection = new Socket("www.somesite.com", 80)){
      Writer out = new OutputStreamWriter(
                              connection.getOutputStream(), "8859_1");
      out.write("GET / HTTP 1.0\r\n\r\n");;
      out.flush();
      connection.shutdownOutput();
      // read the response
    } catch (IOException ex) { ex.printStackTrace(); }
```

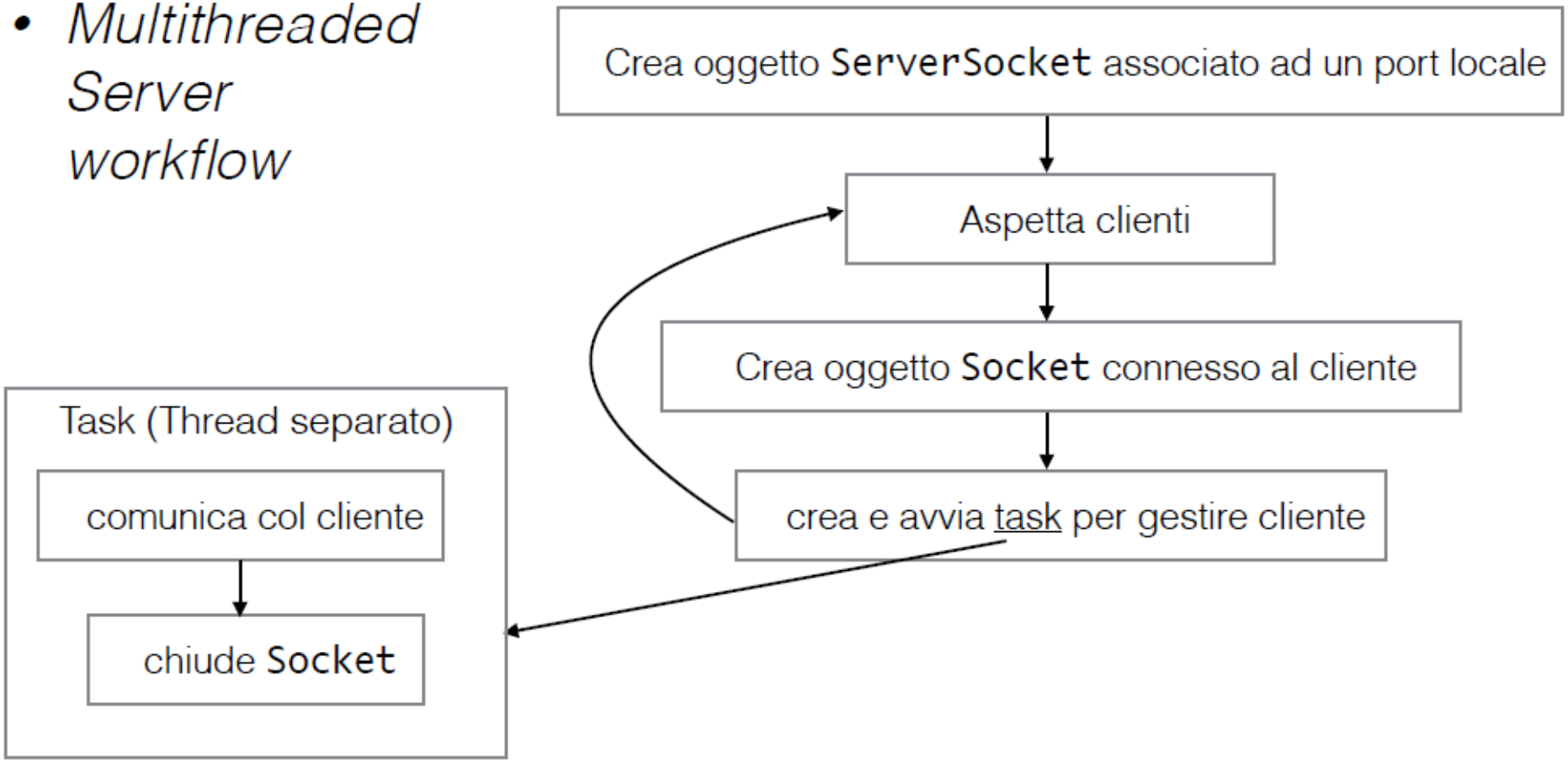
- scritture successive sollevano una IOException

MULTITHREADED SERVER

- nello schema del lucido precedente, la fase “communicate and work” può essere eseguita in modo concorrente da più threads
- un thread per ogni client, gestisce le interazioni con quel particolare client
- il server può gestire le richieste in modo più efficiente
- tuttavia.....threads: anche se processi lightweigth ma tuttavia utilizzano risorse !
- Soluzione 1 -> utilizzare i threadpool

MULTITHREADED SERVER

- *Multithreaded Server workflow*



```
public class ThreadPoolEchoServer {
public static void main(String[] args) {
        ExecutorService es=null;
        try (ServerSocket server = new ServerSocket()) {
        server.bind(new InetSocketAddress(InetAddress.getLocalHost(),
        1500));
        es= Executors.newFixedThreadPool(10);
        while (true){
        try { //not try with resources
                Socket client=server.accept();
                EchoClientHandler handler = new
                          EchoClientHandler(client);
                es.execute(handler);
        } catch (IOException e){
        System.out.println("Some error appeared");
        } } } catch (UnknownHostException e) {
                e.printStackTrace();
        } catch (IOException e) {
                e.printStackTrace();
```

```
class EchoClientHandler implements Runnable{
        Socket client;
        public EchoClientHandler(Socket client) {
        this.client=client; }
        @Override
        public void run() {
        try (BufferedReader reader= new BufferedReader(
            new InputStreamReader(this.client.getInputStream()));
            BufferedWriter writer= new BufferedWriter(
            new OutputStreamWriter(this.client.getOutputStream()));){
        String message="";
        while ((message= reader.readLine())!=null){
                System.out.println("Client sent: "+message);
                writer.write(message);
                writer.flush(); }
        } catch (IOException e){
                System.out.println("Client closed connection or an
                        error appeared.");
        }
```

Esempio TimeServer

INTERAGIRE CON SERVER PREDEFINITI

- ...ma cosa accade se client e server non sono entrambe scritti in JAVA?
- è sufficiente rispettare il protocollo ed il formato dei dati scambiati, codificati in un formato interscambiabile
  - `testo`
  - `JSON`
  - `XML`
- esempio: **`NIST: National Institute of Standards and Technology`**
- aprire una connessione sulla porta `13`, verso il servizio `time.nist.gov`
- il server risponde inviando informazioni su data, ora etc, secondo un formato pre definito (vedi hytp://.nist.gov, per informazioni sui vari campi)

```
58048 17-10-22 14:01:15 15 0 0 667.9 UTC(NIST) *
```

TIME CLIENT NIST

```
public class TimeClient {
  public static void main(String[] args) {
    String hostname = args.length > 0 ? args[0] : "time.nist.gov";
    Socket socket = null;
    try {
      socket = new Socket(hostname, 13);
      socket.setSoTimeout(15000);
      InputStream in = socket.getInputStream();
      InputStreamReader reader = new InputStreamReader(in, "ASCII");
      StringBuilder time = new StringBuilder();
      for (int c = reader.read(); c != -1; c = reader.read()) {
              time.append((char) c);  }
      System.out.println(time);
        } catch (IOException ex) {
    System.out.println(ex);
```

TIME CLIENT NIST

```
  } finally {
      if (socket != null) {
        try {
          socket.close();
        } catch (IOException ex) {
          // ignore
        }
      }
    }
  }
}
```

- notare l'uso del Reader che consente di specificare la codifica dei caratteri che il server invia

TIME PROTOCOL: RFC 868

- `RFC 868: Time Protocol`:
- può essere implementato su TCP o UDP

- funzionamento:
  - Server : `Listen` on port 37.
  - Client: `Connect` to port 37.
  - Server: `Send` the time as a 32 bit binary number.
  - Client: `Receive` the time.
  - Client: `Close` the connection.
  - Server: `Close` the connection.

TIME PROTOCOL: RFC 868

- Tempo: numero di secondi dalle 00:00 (midnight) 1 January 1900 GMT,
- Il tempo 1 è 12:00:01 del 1 January 1900 GMT; questa base sarà utilizzata fino al 2036.

- Ad esempio :

```
 2,208,988,800 corrisponde a 00:00  1 Jan 1970 GMT,
 2,398,291,200 corrisponde a 00:00  1 Jan 1976 GMT,
 2,524,521,600 corrisponde a 00:00  1 Jan 1980 GMT,
 2,629,584,000 corrisponde a 00:00  1 May 1983 GMT,
-1,297,728,000 corrisponde a 00:00 17 Nov 1858 GMT.
```

TIME PROTOCOL SERVER

```
import java.io.*;
import java.net.*;
import java.util.Date;
import java.nio.*;
public class TimeServer {

  public final static int PORT = 6000;

  public static byte[] longToBytes(long x) {
    ByteBuffer buffer = ByteBuffer.allocate(Long.BYTES);
    buffer.putLong(x);
    return buffer.array();
                                                }
```

TIME PROTOCOL SERVER

```
public static void main(String[] args) {
   // The time protocol sets the epoch at 1900,
   // the Date class at 1970. This number
   // converts between them.
   long differenceBetweenEpochs = 2208988800L;
   try (ServerSocket server = new ServerSocket(PORT)) {
     while (true) {
       try (Socket connection = server.accept()) {
            OutputStream out = connection.getOutputStream();
            Date now = new Date();
            long msSince1970 = now.getTime();
            long secondsSince1970 = msSince1970/1000;
            long secondsSince1900 = secondsSince1970  +
                                    differenceBetweenEpochs;
```

TIME PROTOCOL SERVER

```
            byte[] time = new byte[8];
            time = longToBytes(secondsSince1900);
          out.write(time);
          System.out.println(secondsSince1900);
          out.flush();
        } catch (IOException ex) {
          System.out.println(ex.getMessage());
        }
      }
    } catch (IOException ex) {
      System.out.println(ex); }
  }
}
```

TIME PROTOCOL CLIENT

```
import java.net.*;
import java.io.*;
import java.nio.*;
public class TimeClient {
static public long bytesToLong(byte[] bytes) {
    ByteBuffer buffer = ByteBuffer.allocate(Long.BYTES+10);
    buffer.put(bytes);
    buffer.flip();//need flip
    return buffer.getLong();}

  public static void main(String[] args) {
    String hostname = "localhost";
    Socket socket = null;
    byte[] time = new byte[8];
```

TIME PROTOCOL CLIENT

```
    try {socket = new Socket(hostname, 6000);
         InputStream in = socket.getInputStream();
         int x = in.read(time);
         System.out.println(x);
         Long timeL=bytesToLong(time);
         System.out.println(timeL);
     } catch (IOException ex) { System.out.println(ex);}
       finally {
       if (socket != null) {
         try {
           socket.close();
         } catch (IOException ex) {
           // ignore   }
    }
  }
}
```

Esercizio (laboratorio)

Esercizio Socket

Scrivere un programma JAVA che implementi un server che apre una ServerSocket su una porta e sta in attesa di richieste di connessione.

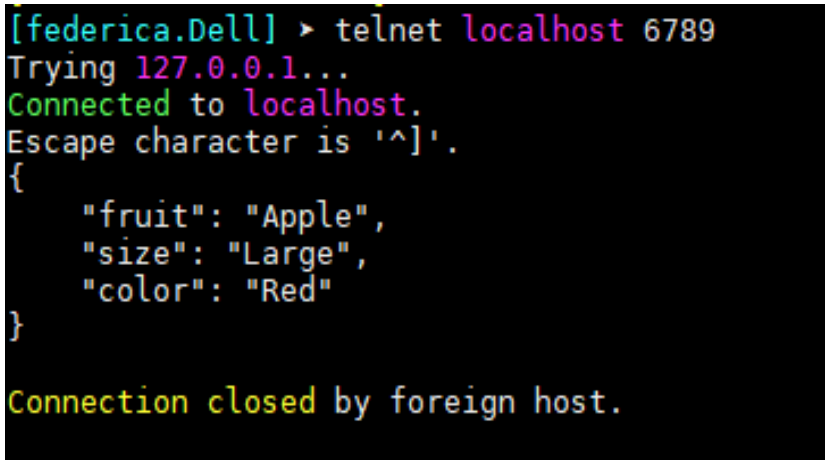
Quando arriva una richiesta di connessione, accetta la connessione, trasferisce al client un file e poi chiude la connessione.

Ulteriori dettagli:

- Il server gestisce una richiesta per volta

- Il server invia sempre lo stesso file, usate un file di testo

Per il client potete usare telnet. Qui sotto un esempio di utilizzo:



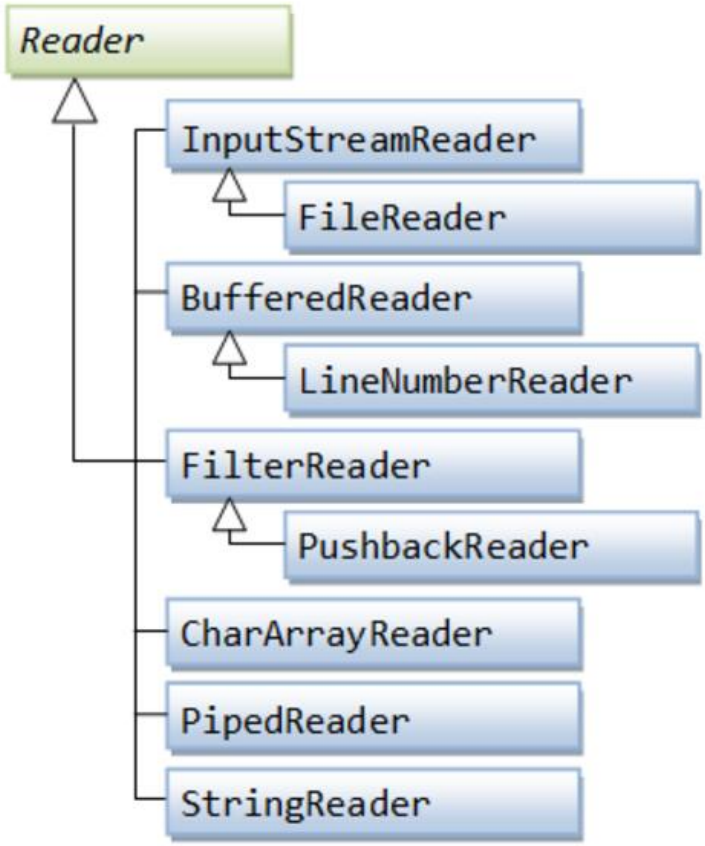
ASSIGNMENT 6: HTTP-BASED FILE TRANSFER

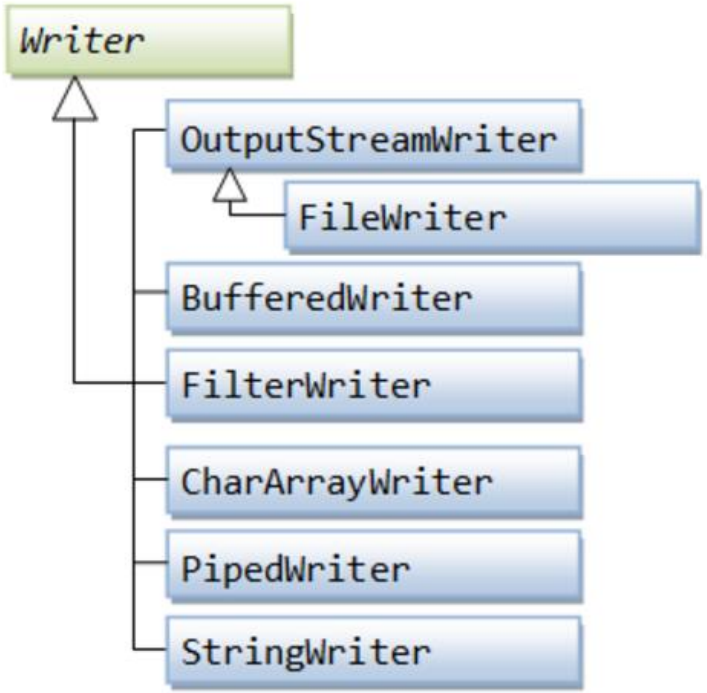
Scrivere un programma JAVA che implementi un server HTTP che gestisca richieste di trasferimento di file di diverso tipo (es. immagini jpeg, gif) provenienti da un browser web.

- il server
  - sta in ascolto su una porta nota al client (es. 6789)
  - gestisce richieste HTTP di tipo GET alla Request URL *localhost:port/filename*

- le connessioni possono essere non persistenti.
- usare le classi Socket e ServerSocket per sviluppare il programma server
- per inviare al server le richieste, utilizzare un qualsiasi browser

CHARACTER-BASED I/O & CHARACTER STREAMS





DA STREAM DI BYTE A STREAM DI CARATTERI

Reader

Reader è una classe astratta, usata polimorficamente attraverso una delle sue sottoclassi

```
public abstract int read(char[] text, int offset, int length)
 throws IOException
public int read() throws IOException
```

Restituisce un singolo carattere Unicode character come int (da 65535 a 0) o -1 (fine dello stream)

```
public int read(char[] text) throws IOException
```

prova riempire l'array e restituisce il numero di char letti o -1

```
read(char[] text, int offset, int length)
```

attempts to read length characters into the subarray of text beginning at offset and continuing for length characters. It also returns the actual number of characters read or -1 on end of stream

```
public abstract void close(  ) throws IOException
```

…

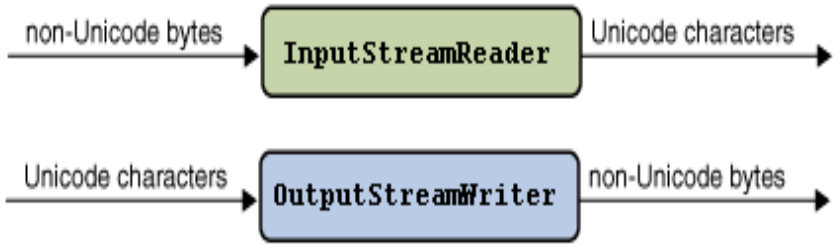
DA STREAM DI BYTE A STREAM DI CARATTERI

InputStreamReader

- Legge byte da un input stream (es. FileInputStream) e traduce questi byte in caratteri Unicode a 16 bit.
- permette di specificare un encoding particolare, così che possa essere utilizzato un insieme di diverso da quello di default (es. file di testo generati con altri tool)
- se non specificato diversamente, assume che i caratteri nel file siano codificati usando la codifica di default adottata dalla piattaforma locale
- Nel costruttore si può specificare l'input stream da cui leggere e la codifica da usare

```
public InputStreamReader(InputStream in)

public InputStreamReader(InputStream in, St
UnsupportedEncodingException
```



Writer

```
public abstract void write(char[] text, int offset, int length)
 throws IOException
public void write(char[] text) throws IOException
public abstract void flush(  ) throws IOException
public abstract void close(  ) throws IOException
…
```

OutputStreamWriter

riceve caratteri Unicode da un programma in esecuzione. Quindi traduce quei caratteri in byte usando una codifica specificata e scrive i byte su un output stream sottostante

The write(char[] text, int offset, int length): scrive una porzione dell'array di caratteri (length caratteri dalla posizione offset)

```
OutputStreamWriter w = new OutputStreamWriter(
      new FileOutputStream("OdysseyB.txt"), "Cp1252");
```

Buffered readers and writers

Usano un buffer interno di chars.

Quando un programma legge da un BufferedReader, il testo viene preso dal buffer anziché direttamente dall'input stream sottostante o da un'altra sorgente. Quando il buffer si svuota, viene riempito di nuovo con quanto più testo possibile, anche se non tutto è richiesto immediatamente.

Quando un programma scrive in un BufferedWriter, il testo viene inserito nel buffer. Il testo viene scritto nell'output stream sottostante o in un'altra destinazione solo quando il buffer si riempie o quando il writer viene svuotato (flushed) in modo esplicito. La classe BufferedReader ha un metodo readLine( ) che legge un singola riga

- Filter Stream: vedi esempio
  - `System.in` e `System.out` sono istanze di Inputstream ed OutputStream, rispettivamente

```
BufferedReader myIn =
    new BufferedReader(new InputStreamReader(System.in));
```

READERS E WRITERS

```
import java.io.*;

public class Encoding {

  public static void main(String[] args) throws
java.io.UnsupportedEncodingException {

  try (BufferedReader rdr = new BufferedReader(

                          new InputStreamReader(

                           new FileInputStream("filename"),"UTF-8"));){

   String line = rdr.readLine();

   System.out.println(line);

         } catch (IOException exc) {

                  System.err.println("I/O error"); }

    }

}
```