Laboratorio di Reti
Lezione 7

JAVA.NET:
indirizzi IP, stream sockets

5/11/2020

Laura Ricci

NETWORK APPLICATIONS

applicazioni pervasive e di grande diffusione:

- Web browsers
- SSH
- email
- social networks
- teleconferences (skype, Zoom, GoToMeeeting, Meet, Teams,...)
- P2P File sharing: Bittorrent
- program development environments: GIT
- collaborative work: ShareLatex
- multiplayer games: War of Warcraft
- cryptocurrencies: Bitcoin
- e-commerce

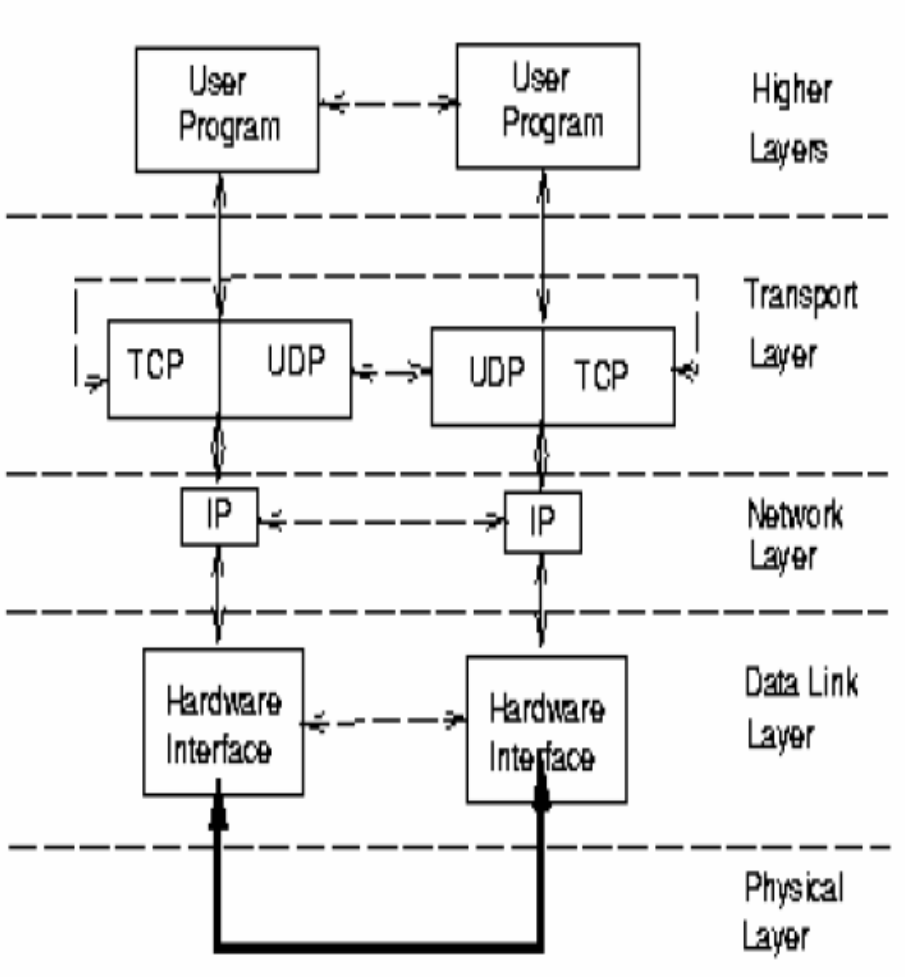
Scopo del corso è mettervi in grado di sviluppare una semplice applicazione di rete.

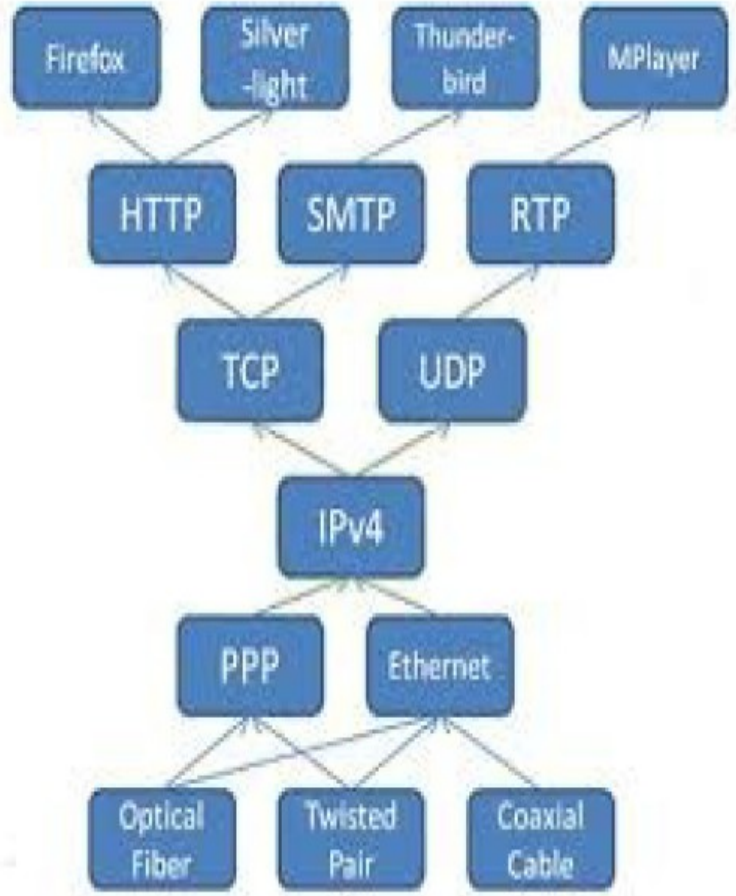
NETWORK APPLICATIONS

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

NETWORK LAYERS: DAL MODULO DI TEORIA



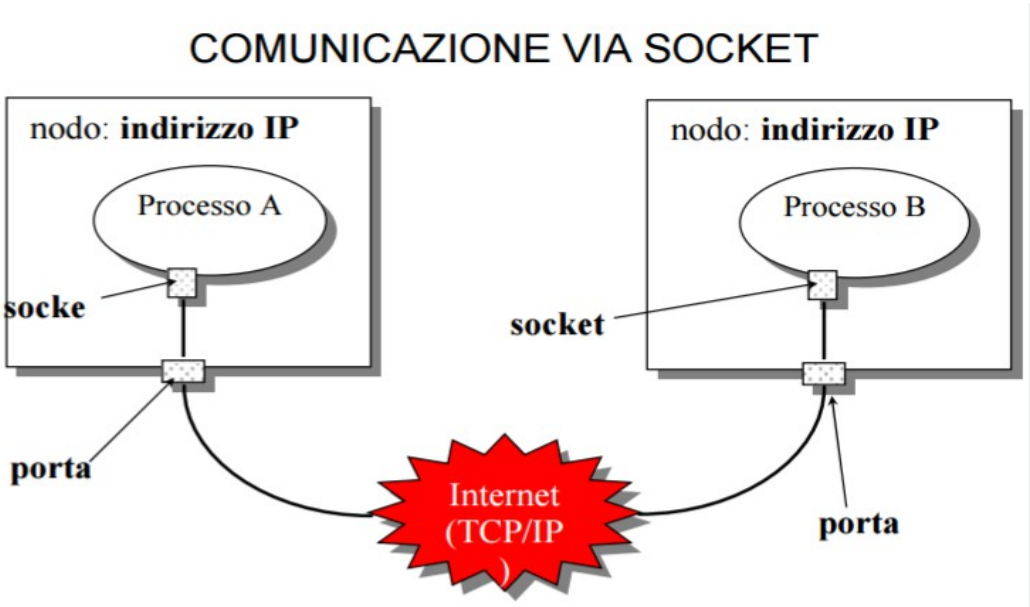


SOCKET: UNO "STANDARD" DI COMUNICAZIONE

- uno standard per connettere dispositivi distribuiti, diversi, eterogenei
- termine utilizzato in tempi remoti in telefonia.
  - la connessione tra due utenti veniva stabilita tramite un operatore
  - l'operatore inseriva fisicamente i due estremi di un cavo in due ricettacoli (sockets)
  - un socket per ogni utente

SOCKET: UNO "STANDARD" DI COMUNICAZIONE

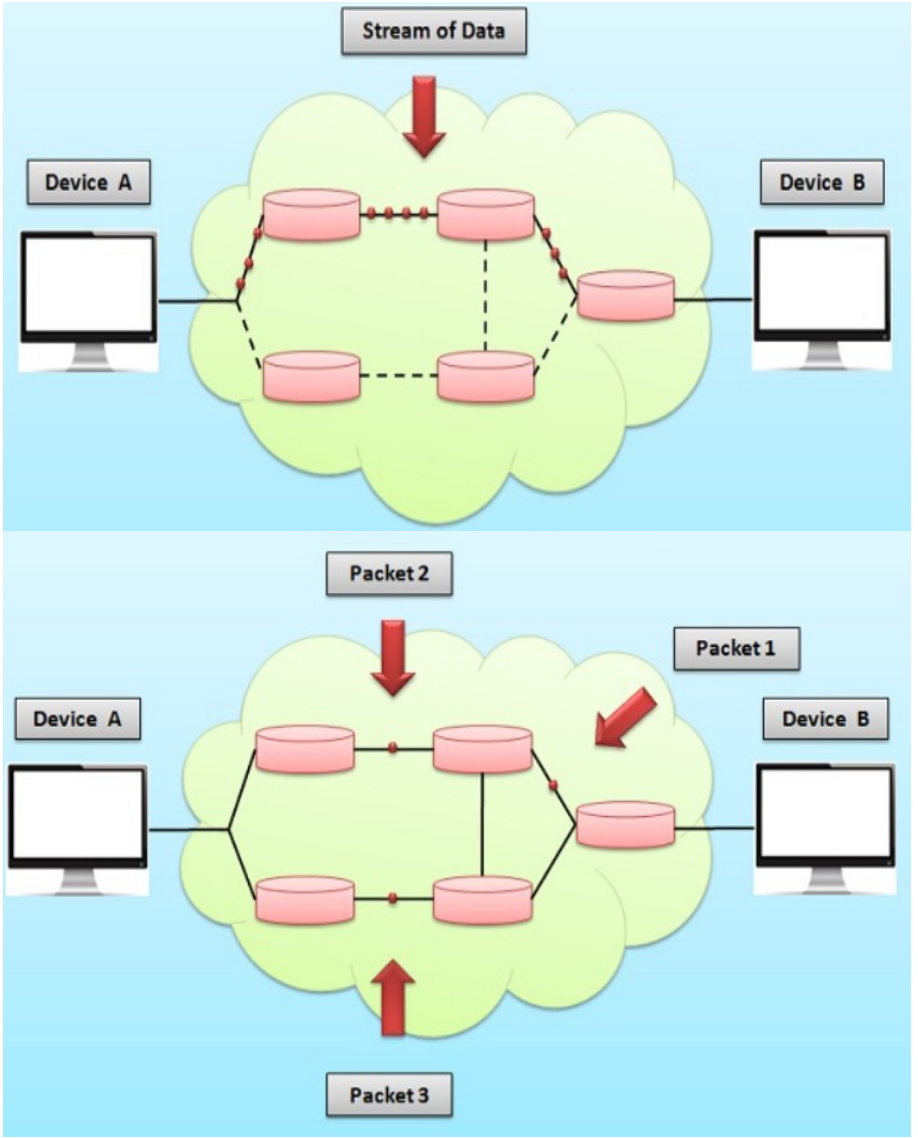
- una presa "standard" a cui un processo si può collegare per spedire dati
- un endpoint sull'host locale di un canale di comunicazione da/verso altri hosts
- introdotti in Unix BSD 4.2
- collegati ad una porta locale



TIPI DI COMUNICAZIONE TRAMITE SOCKET

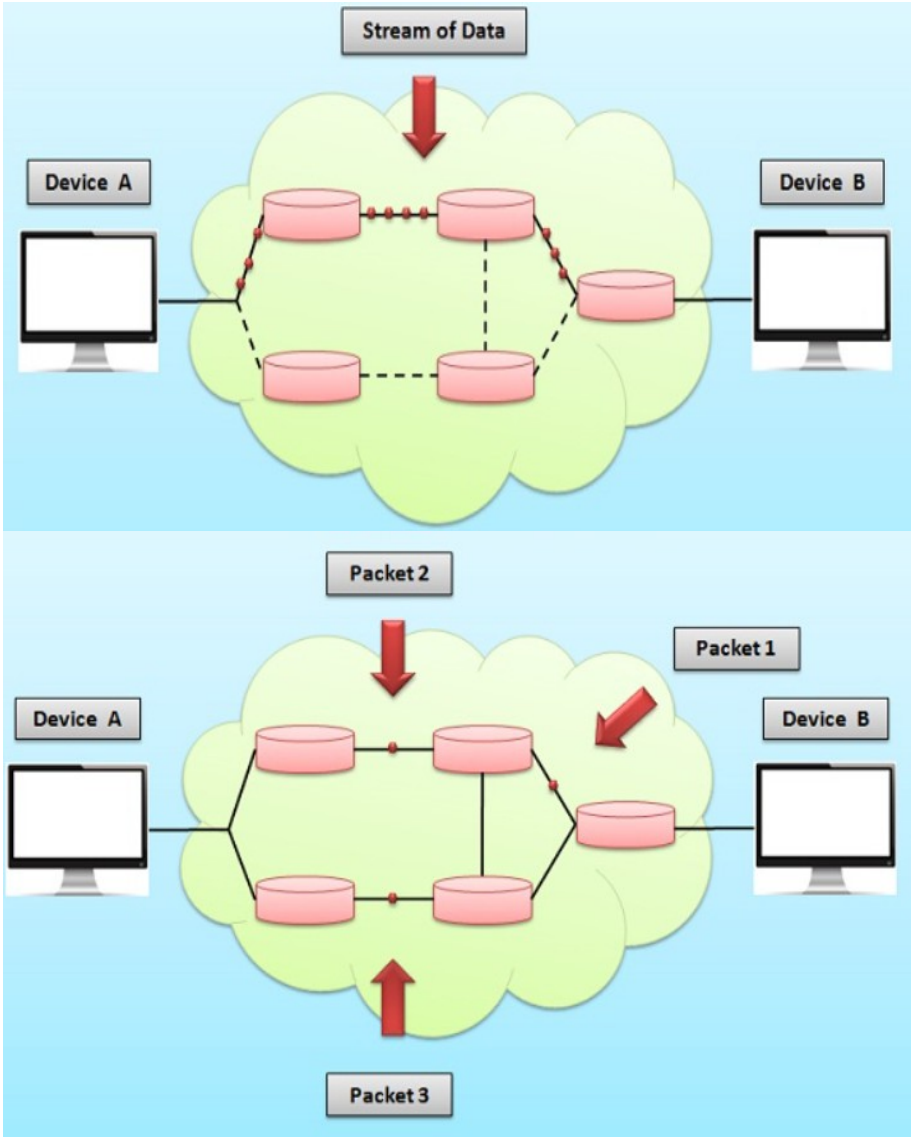
- Connection Oriented (TCP)
  - come una chiamata telefonica
  - una connessione stabile (canale di comunicazione dedicato) tra mittente e destinatario
  - stream socket

- Connectionless (UDP)
  - come l'invio di una lettera
  - non si stabilisce un canale di comunicazione dedicato
  - ogni messaggio viene instradato in modo indipendente dagli altri
  - datagramsocket



TIPI DI COMUNICAZIONE TRAMITE SOCKET

- Connection Oriented
  - l'indirizzo del destinatario (IP+ porta) specificato al momento della apertura connessione
  - ordinamento: garantito
  - utilizzo: reliability, trasmissione di grosse moli di dati
- Connectionless
  - indirizzamento: l'indirizzo del destinatario (IP + porta) viene specificato in ogni pacchetto
  - ordinamento dei dati scambiati: non garantito
  - utilizzo: quando la reliability non è essenziale e si privilegia la velocità di trasmissione



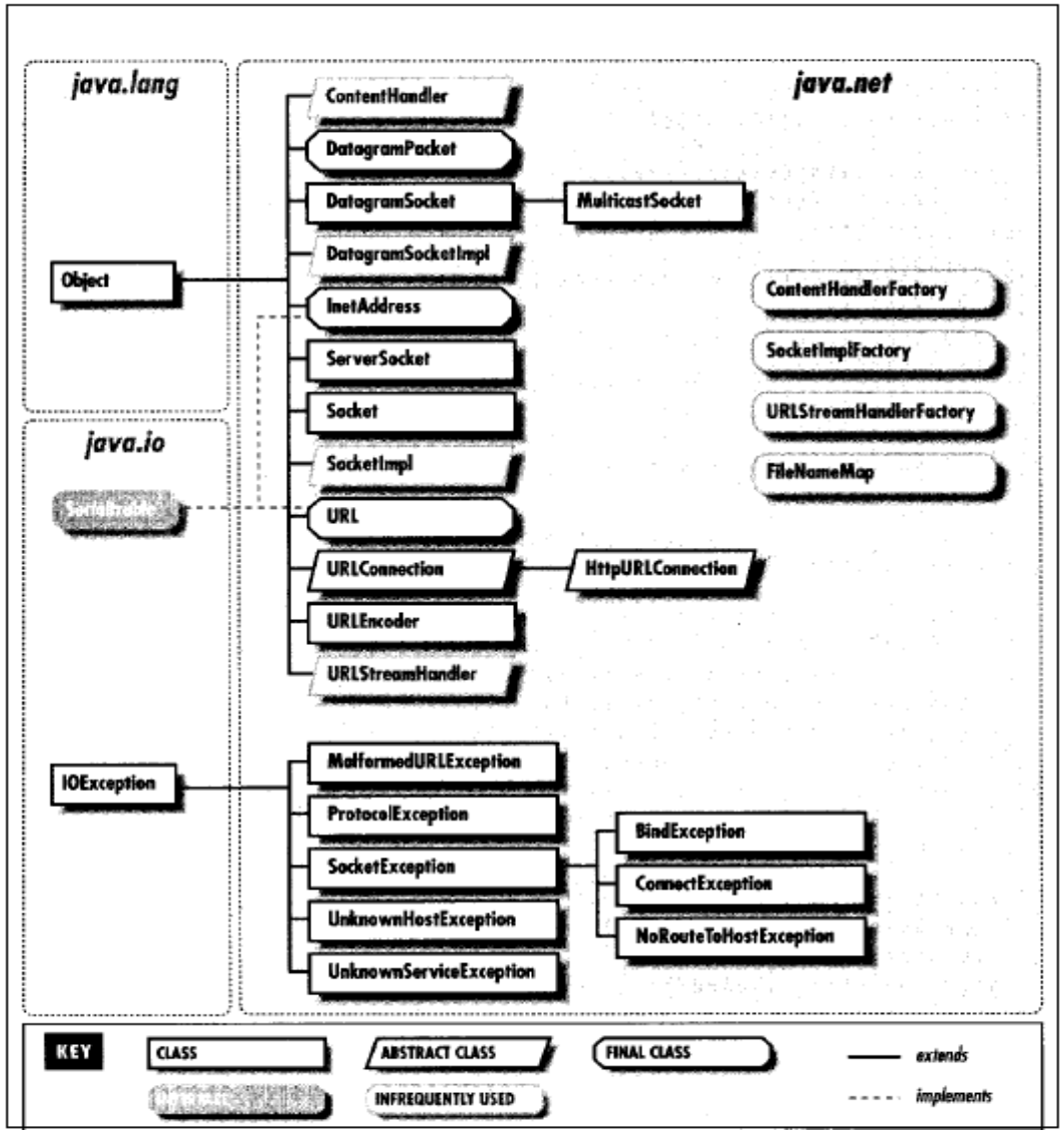
SOCKET IN JAVA: LA GERARCHIA DELLE CLASSI

connection-oriented

- connessione modellata come stream
- asimmetrici
- client side: `Socket` class
- server side:
  - `ServerSocket`
  - `Socket` class

connectionless

- `datagramSocket`
  - simmetrici: sia per il client che per il server



IDENTIFICARE LE APPLICAZIONI

identificazione di un processo con cui comunicare:

- la rete all'interno della quale si trova l'host su cui è in esecuzione il processo
- l'host all'interno della rete
- il processo in esecuzione sull'host

- rete ed host: identificati da di Internet Protocol, mediante indirizzi IP
- processo: identificato da una porta, rappresentata da un 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

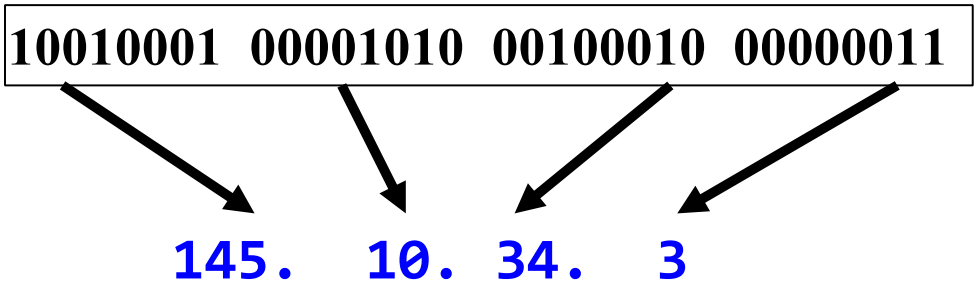
    (client *sky3.cm.deakin.edu.au, 139.130.118.5)*
  - la porta locale esempio: 5101
  - l'indirizzo del computer remoto

    (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



- 4 bytes: ognuno interpretato come un numero decimale senza segno
- valore di ogni byte: 0..255
- 2^{32} indirizzi
- address special blocks in IPV4:
  - loopback address: 127.0.0.0
  - 255.255.255.255: broadcast

un indirizzo IPV6

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

- 16 bytes, 2^{128} indirizzi

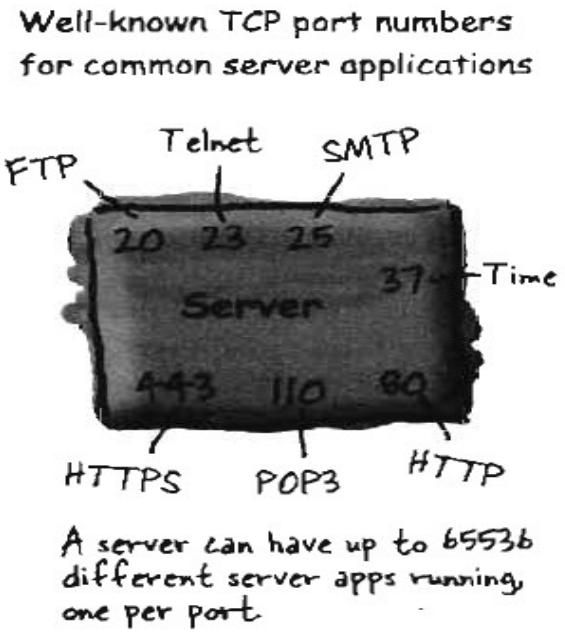
INDIRIZZI IP E NOMI

- gli indirizzi IP semplificano l'elaborazione effettuata dai routers, ma sono poco leggibili per gli utenti della rete
- soluzione
  - assegnare un nome simbolico unico ad ogni host della rete
  - si utilizza uno 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
- indirizzi a lunghezza fissa verso indirizzi a lunghezza variabili
- Domain Name System (DNS) traduce nomi in indirizzi IP

INDIRIZZAMENTO A LIVELLO DI PROCESSI

- porte:...perchè su ogni host possono essere attivi contemporaneamente più servizi (es: e-mail, ftp, http,…)
  - ogni servizio viene incapsulato in un diverso processo
- l'indirizzamento di un processo avviene mediante una porta
  - intero compreso tra 1 e 65535 (per ogni protocollo di trasporto)
  - non un dispositivo fisico, ma un'astrazione per individuare i singoli servizi (processi)
- porte 1–1023: riservate per well-known services.
  - servizio HTTP in genere sulla porta 80.
  - Unix hosts: solo root process possono ascoltare queste porte
  - usare valori di porta > 1024



LA CLASSE INETADDRESS

- rappresenta ad alto livello un indirizzo IP con un oggetto di tipo `InetAddess`, un record che contiene
  - `String` hostName: una stringa che rappresenta il nome simbolico di un host
  - `byte[]` address: un vettore di bytes che rappresenta l'indirizzo IP dell'host
- nessun costruttore, ma tre metodi statici, che definiscono una factory per costruire oggetti di tipo InetAddress

```
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 il cui nome è passato come parametro e restituisce un oggetto di tipo `InetAddress`
- richiede una interrogazione del DNS per risolvere il nome dell'host
  - l'host deve essere connesso in rete
  - può utilizzare una cache locale, in questo caso ricerca prima nella cache
- può sollevare `UnKnownHostException`, se non riesce a risolvere il nome dell'host
- reverse lookup: passo un indirizzo IP come parametro, nella forma dotted quad

LA CLASSE INETADDRESS

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

- per hosts che posseggano 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 example {
  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.amazon.com");
    for (InetAddress ipAddress : inetAddresses)
         { System.out.println(ipAddress);}
    }}
```

OUTPUT DEL PROGRAMMA

```
192.168.1.13
www.java.com/104.83.83.17
www.amazon.com/99.86.160.215
```

OVERRIDING OBJECT METHODS

- come tutte le classi, la classe `InetAddress` eredita da `java.lang.Object`.
- effettua overriding dei 3 metodi base di `Object`
  - `equals()`: due oggetti `InetAddress` sono uguali se e solo se
    - hanno lo stesso indirizzo IP
    - non necessariamente devono avere lo stesso hostname
  - `HashCode()`
    - converte 4 bytes dell'indirizzo IP in un `int`
    - coerente con `equals`, non considera hostname
  - `toString()`
    - restituisce nome dell'host/indirizzo dotted quad
    - se non esiste il nome, stringa vuota + indirizzo dotted quad

INETADDRESS: CACHING

- i metodi descritti effettuano caching dei nomi/indirizzi risolti
  - l'accesso al DNS è una operazione potenzialmente molto costosa
  - anche i tentativi di risoluzione non andati a buon fine in cache
- permanenza dati nella cache:
  - 10 secondi se la risoluzione non ha avuto successo, spesso il primo tentativo di risoluzione fallisce a causa di un time out...
  - tempo illimitato altrimenti. Problemi: indirizzi dinamici.
- `java.security.Security.setProperty` imposta il numero di secondi in cui una entrata nella cache rimane valida,

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

per i tentativi non andati a buon fine: `networkaddress.cache.negative.ttl`

- nomi risolti con i dati nella cache, quando possibile (di default: per sempre)

CACHING DI INDIRIZZI IP: "UNDER THE HOOD"

```
import java.net.InetAddress; import java.net.UnknownHostException;
import java.security.*;
public class Caching {
   public static final String CACHINGTIME="0";
   public static void main(String [] args) throws InterruptedException
    {Security.setProperty("networkaddress.cache.ttl",CACHINGTIME);
     long time1 = System.currentTimeMillis();
     for (int i=0; i<1000; i++){
        try {System.out.println(
             InetAddress.getByName("www.cnn.com").getHostAddress());}
        catch (UnknownHostException uhe)
                { System.out.println("UHE");} }
        long time2 = System.currentTimeMillis();
        long diff=time2-time1; System.out.println("tempo trascorso e'"+diff);}}
```

CACHINGTIME=0 tempo trascorso è 545

CACHINGTIME=1000 tempo trascorso è 85

USARE INETADDRESS: SPAM CHECKING

- diversi servizi monitorano gli spammers: real-time black-hole lists (RTBLs)
  - ad esempio: `sbl.spamhaus.org`
  - mantengono una lista di indirizzi IP che risultano, probabilmente, degli spammers

- per identificare se un indirizzo IP corrisponde ad uno spammer:
  - inversione dei bytes dell'indirizzo IP
  - concatena il risultato a `sbl.spamhaus.org`
  - esegui un DNS look-up
  - la query ha successo se e solo se l'indirizzo IP corrisponde ad uno spammer

- esempio: controlla se l'indirizzo `23.45.65.88` corrisponde ad uno spammer
  - esecuzione di un DNS lookup su `88.65.45.23.sbl.spamhaus.org`

USARE INETADDRESS: SPAM CHECKING

```
import java.net.*;
public class SpamCheck {
  public static final String BLACKHOLE = "sbl.spamhaus.org";
  public static void main(String[] args) throws
                                                UnknownHostException
      { for (String arg: args) {
            if (isSpammer(arg)) {
               System.out.println(arg + " is a known spammer.");
                 } else {
               System.out.println(arg + " appears legitimate.");

                        }
            }
      }
   }
```

USARE INETADDRESS: SPAM CHECKING

```
private static boolean isSpammer(String arg) {
    try { InetAddress address = InetAddress.getByName(arg);
          byte [] quad = address.getAddress();
          String query = BLACKHOLE;
          for (byte octet : quad) {
              int unsignedByte = octet < 0 ? octet + 256 : octet;
              query = unsignedByte + "." + query;
          }
          InetAddress.getByName(query);
          return true;
    } catch (UnknownHostException e) {
      return false;
    }}}
```

```
23.45.65.88 appears legitimate.
1.1.1.1 appears legitimate.
141.250.89.99 appears legitimate.
0.4.2.1 appears legitimate.
207.87.34.17 appears legitimate.
127.0.0.2 is a known spammer
```

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 una interfaccia virtuale legata ad un indirizzo IP fisico
- un host, ad esempio un router, può presentare più interfacce 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 da un insieme di indirizzi IP
  - gateway tra sottoreti IP
  - routers

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

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

INDIRIZZO DI LOOPBACK

```
import java.net.*;
import java.util.*;
public class InterfaceLister {
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)
```

INDIRIZZO DI LOOPBACK

`127.0.0.1` Indirizzo di loopback

- utilizzabile per testare applicazioni
- quando si usa un indirizzo di loopback si possano eseguire client e server in locale, sullo stesso host
- attivati da due shell diverse, o due progetti diversi Eclipse
- 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

ENUMERARE LE INTERFACCE

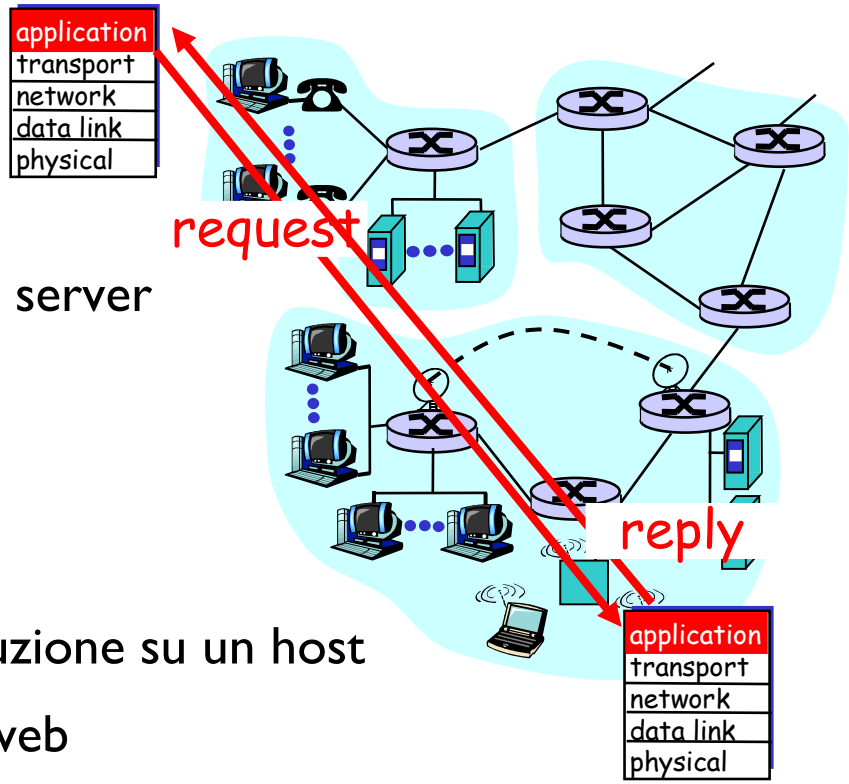
```
import java.net.*;
import java.util.*;

public class InterfaceListener {
 public static void main(String[] args) throws SocketException {
 Enumeration<NetworkInterface>interfaces=NetworkInterface.getNetworkInterfaces();
    while (interfaces.hasMoreElements()) {
      NetworkInterface ni = interfaces.nextElement();
      System.out.println(ni); }}}
```

```
name:lo (Software Loopback Interface 1)
name:wlan0 (Microsoft Wi-Fi Direct Virtual Adapter #2)
name:net0 (Microsoft 6to4 Adapter)
name:net1 (WAN Miniport (SSTP))
name:ppp0 (WAN Miniport (PPPOE))
name:eth0 (Bluetooth Device (Personal Area Network))
name:wlan1 (Microsoft Wi-Fi Direct Virtual Adapter)
name:eth1 (Microsoft Kernel Debug Network Adapter)
name:net2 (Microsoft IP-HTTPS Platform Adapter)
name:eth2 (WAN Miniport (IPv6))
name:net3 (WAN Miniport (L2TP))
name:net4 (Juniper Networks Virtual Adapter Manager)
name:eth3 (WAN Miniport (Network Monitor))
.............
```

IL PARADIGMA CLIENT/SERVER

servizio:

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



client:

- un software che sfrutta servizi forniti dal server

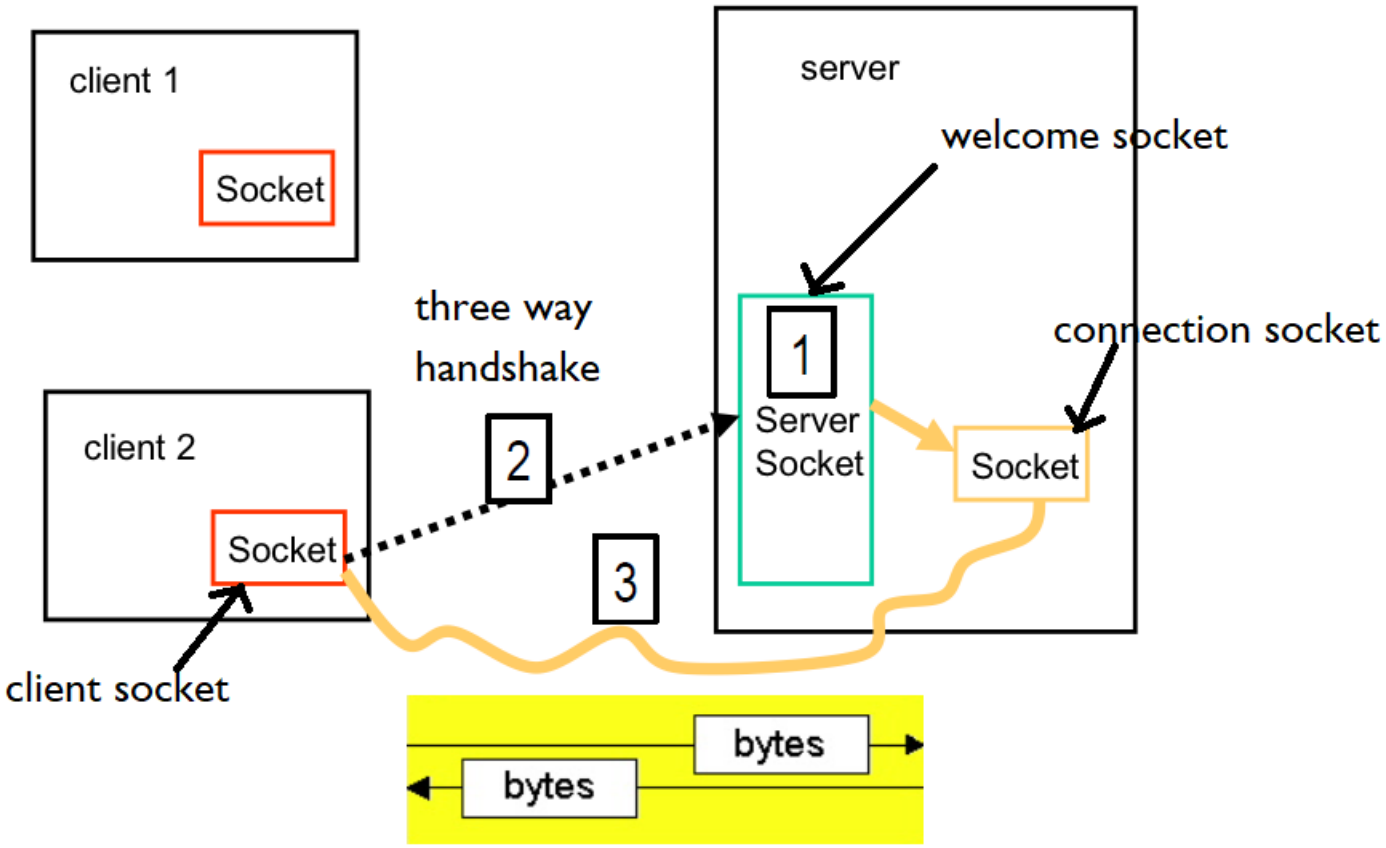
  web client browser

  e-mail client mail-reader

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

IL PARADIGMA CLIENT SERVER: HOW TO DO IN JAVA



IL PARADIGMA CLIENT SERVER: HOW TO DO IN JAVA

- esistono due tipi di socket TCP, lato server:
  - 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 client crea un active socket per richiedere la connessione
- quando il server accetta una richiesta di connessione,
  - crea a sua volta un proprio active socket AS 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

COLLEGARSI AD UN SERVIZIO

- il server pubblica un proprio servizio
  - associato al listening socket, creato sulla porta remota PR, all'indirizzo IP SA
  - usa un oggetto di tipo `ServerSocket`
- il client che intende usufruire del servizio deve conoscere
  - l'indirizzo IP del server, e la porta remota, PR, a cui è associato il servizio
  - usa un oggetto di tipo `Socket`
- la creazione del socket effettuata dal client produce in modo atomico la richiesta di connessione al server
    - completamente gestito dal supporto
    - three way handshake
  - se la richiesta viene accettata,
    - il server crea un socket dedicato per l'interazione con il client
    - tutti i messaggi spediti dal client vengono diretti automaticamente sul nuovo socket creato.

JAVA STREAM SOCKET API: LATO CLIENT

`java.net.Socket` :

costruttori

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

- crea un active socket e tenta di stabilire, tramite esso, 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
                              UnKnownHostException, IOException
```

come il precedente, l'host è individuato dal suo nome simbolico (interroga automaticamente il DNS)

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 un servizio 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
- osservazione: il programma effettua 1024 interrogazioni al DNS, una per ogni socket che tenta di creare
- per ottimizzare il comportamento del programma, utilizzare il costruttore

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

  - il DNS viene interrogato una sola volta, prima di entrare nel ciclo di scanning, dalla `InetAddress.getByName`
  - viene utilizzato l' InetAddress invece del nome dell'host per costruire i sockets

JAVA STREAM SOCKET API: LATO CLIENT

Altri costruttori della Classe `java.net.socket`

```
public Socket (String H, int P, InetAddress IA, int LP)
```

tenta di creare una connessione

- verso l'host H,
- sulla porta P.
- dalla interfaccia locale IA
- dalla porta locale LP

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 p.
- length: lunghezza della coda in cui vengono memorizzate le richieste di connessione.

  se la coda è piena, ulteriori richieste di connessione sono rifiutate

```
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`.

- 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 time-outs)
- se c'è almeno una richiesta, il processo si sblocca e costruisce un nuovo socket S tramite cui avviene la comunicazione effettiva tra cliente server

PORT SCANNER LATO SERVER

- ricerca dei servizi attivi sull'host locale

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

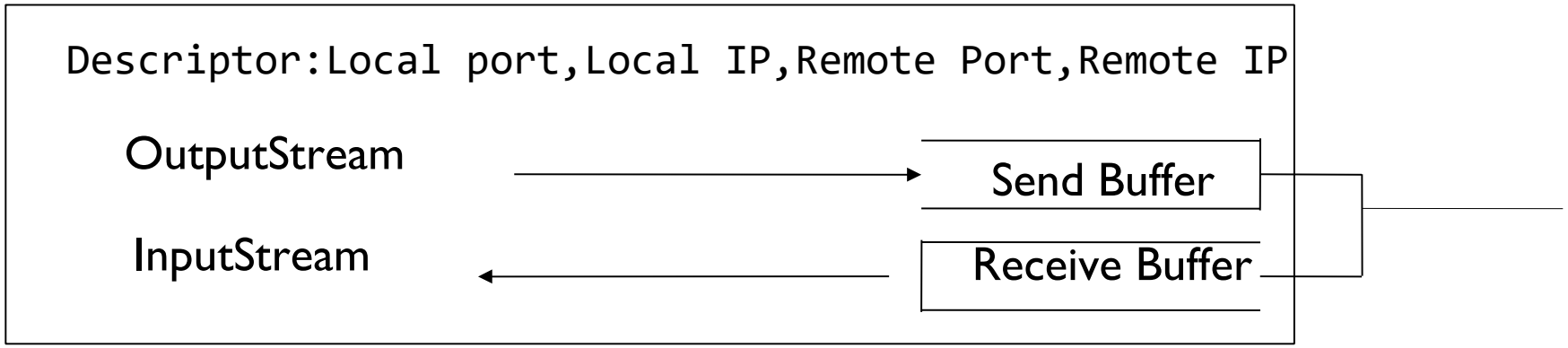
- una volta stabilita una connessione tra client e server come si scambiano i dati?
  - connessione modellata come uno stream
- associare uno stream di input o di output ad un socket:

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

- 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
- ogni valore scritto sullo stream di output associato al socket viene copiato nel *Send Buffer* del livello TCP
- ogni valore letto dallo stream viene prelevato dal *Receive Buffer* del livello TCP

STREAM BASED COMMUNICATION

- dopo che la richiesta di connessione viene accettata, client e server associano streams di bytes di input/output all'active socket
  poichè gli stream sono unidirezionali
- 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

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 and do their
       work
      sock.close();
              }
servSock.close();
```

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 !
  - esempio: un thread che utilizza 1MB di RAM. 1000 thread simultanei possono causare problemi !
- Soluzione, utilizzare
  - i `ServerSocketChannels` di NIO
  - Thread Pooling

A CAPITALIZER SERVICE: SERVICE

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;
import java.util.concurrent.*;
public static void main(String[] args) throws Exception {
        try (ServerSocket listener = new ServerSocket(10000)) {
            System.out.println("The capitalization server is running...");
            ExecutorService pool = Executors.newFixedThreadPool(20);
            while (true) {
                pool.execute(new Capitalizer(listener.accept()));
            }
        }
    }
```

A CAPITALIZER SERVICE: SERVER

```
private static class Capitalizer implements Runnable {
     private Socket socket;
     Capitalizer(Socket socket) {
        this.socket = socket; }
     public void run() {
        System.out.println("Connected: " + socket);
        try (Scanner in = new Scanner(socket.getInputStream());
             PrintWriter out = new PrintWriter(socket.getOutputStream(),
                                                                  true))
            { while (in.hasNextLine()) {
                    out.println(in.nextLine().toUpperCase()); }
            } catch (Exception e) { System.out.println("Error:" + socket); }
           }
      }
     }
```

A CAPITALIZER SERVICE: CLIENT

```
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;
public class CapitalizeClient {
    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            System.err.println("Pass the server IP as the sole command line
                                argument");
            return;
        }
        Scanner scanner=null;
        Scanner in=null;
```

A CAPITALIZER SERVICE: CLIENT

```
try (Socket socket = new Socket(args[0], 10000)) {
            System.out.println("Enter lines of text then EXIT to quit");
            scanner = new Scanner(System.in);
            in = new Scanner(socket.getInputStream());
            PrintWriter out = new PrintWriter(socket.getOutputStream(),
                                                                  true);
            boolean end=false;
            while (!end) {
                    { String line= scanner.nextLine();
                     if (line.contentEquals("exit")) end=true;
                     out.println(line);
                     System.out.println(in.nextLine());}
            }}
        finally {scanner.close(); in.close();}
        }
    }
```

INTERAGIRE CON SERVER PREDEFINITI

- ...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 predefinito (vedi `http://.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();
      StringBuilder time = new StringBuilder();
      InputStreamReader reader = new InputStreamReader(in, "ASCII");
      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'use 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 (45 octal).

  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

```
import java.io.*; import java.net.*; import java.util.Date; import java.nio.*;
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;
            byte[] time = new byte[8];
```

TIME PROTOCOL SERVER

```
        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
     }}}}}
```

ASSIGNMENT 7: 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

ESERCIZIO PREPARAZIONE ASSIGNMENT

- 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, se funziona sula vostra macchina, altrimenti scrivere anche il client in JAVA