Laboratorio di Reti – A (matricole pari)

Autunno 2021,
instructor: Laura Ricci

laura.ricci@unipi.it

Lezione 6

InetAddress, Stream Sockets

19/10/2021

NETWORK APPLICATIONS

applicazioni pervasive e di grande diffusione:

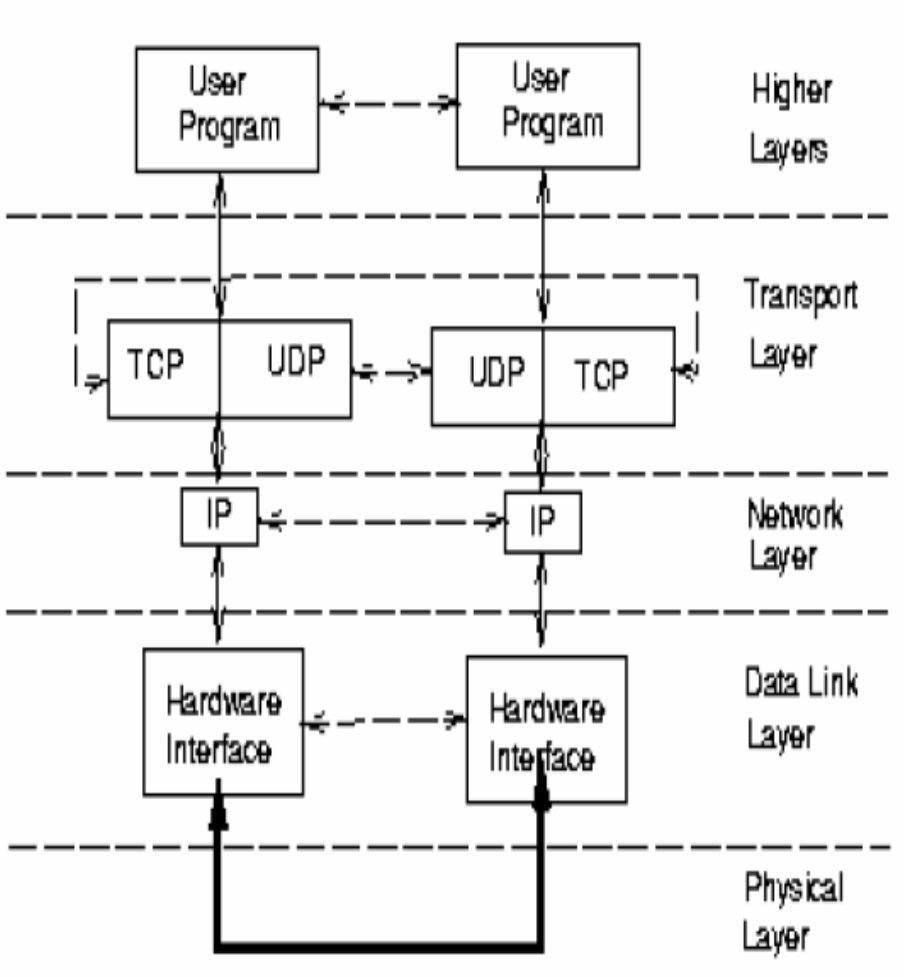
- web browsers
- SSH
- email
- social networks
- teleconferences (skype, Zoom, GoToMeeeting, Meet, Teams,...)
- P2P File sharing: Bittorrent
- program development environments: GIT
- collaborative work: Overleaf
- multiplayer games: War of Warcraft
- blockchain: cryptocurrencies (Bitcoin), supply chain,...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

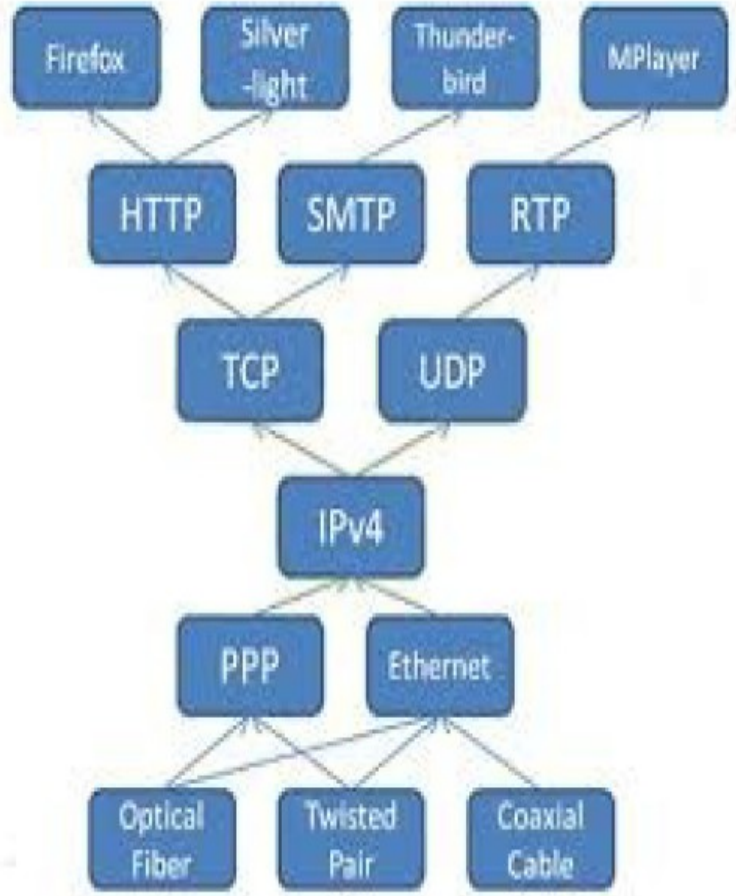
Scopo del corso è mettervi in grado di sviluppare una semplice applicazione di rete.

NETWORK APPLICATIONS

- 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

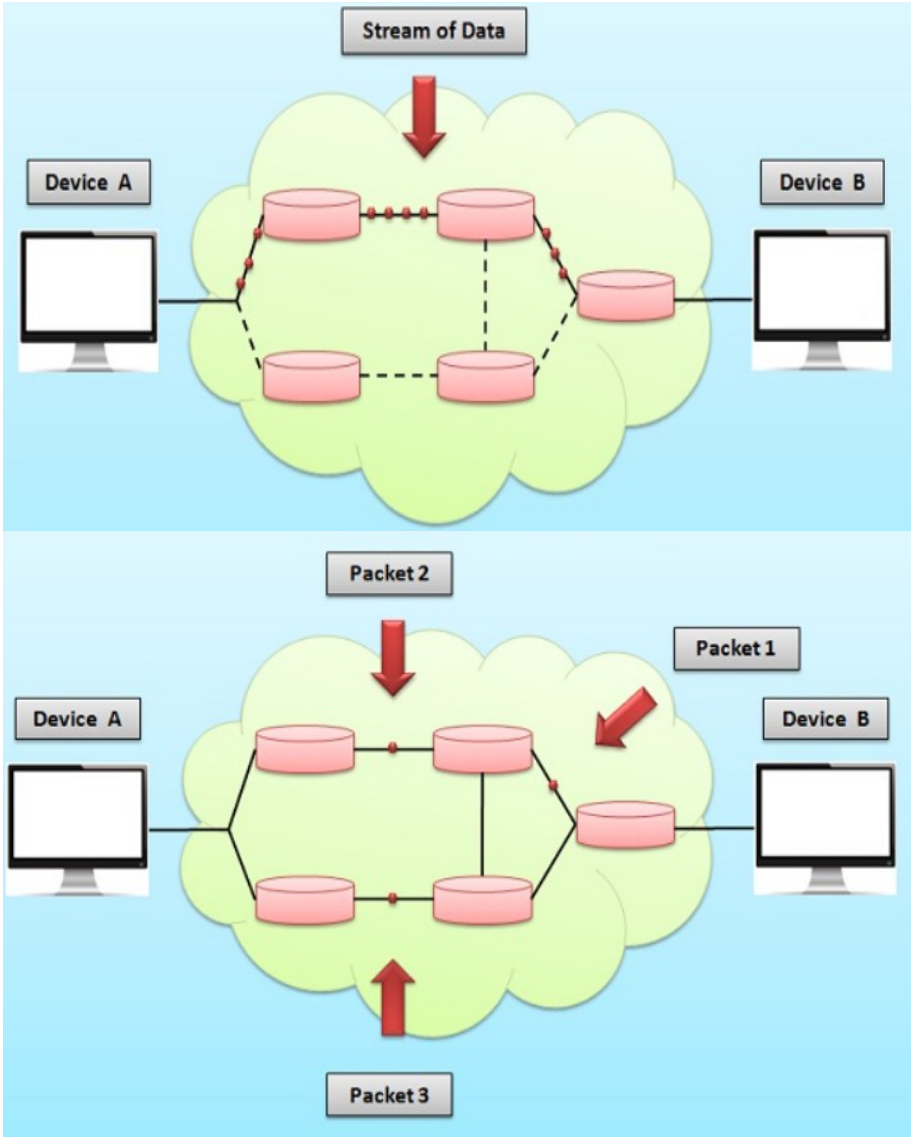
NETWORK LAYERS: DAL MODULO DI TEORIA





TIPI DI COMUNICAZIONE

- 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



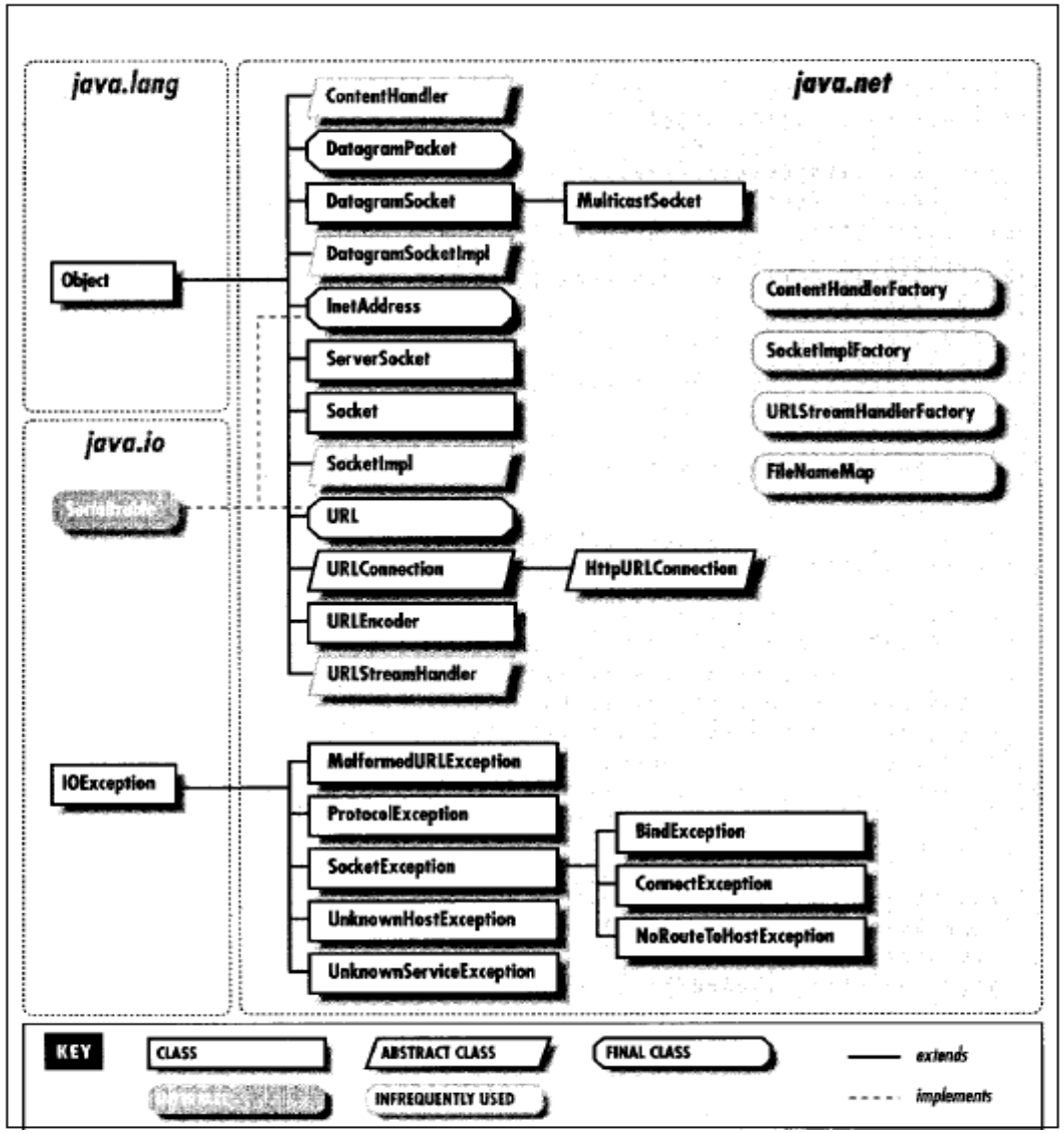
NETWORKING IN JAVA: JAVA.NET

connection-oriented

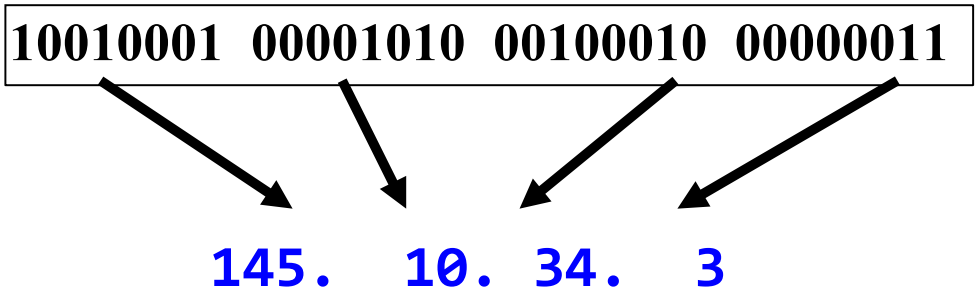
- connessione modellata come stream
- asimmetrici
  - client side: `Socket` class
  - server side:
    - `ServerSocket`
    - `Socket` class

connectionless

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



IP (INTERNET PROTOCOL) ADDRESS



- IPV4, 4 bytes: 2^{32} indirizzi, ogni byte interpretato come un numero decimale senza segno
- dotted quad form
- alcuni indirizzi riservati, loopback address: `127.0.0.0`, broadcast `255.255.255.255`

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

- IPV6, 16 bytes, 2^{128} indirizzi, 8 blocchi di 4 cifre esadecimali

`2400:cb00:2048:0001:0000:0000:6ca2:c665` → `2400:cb00:2048:1::6ca2:c665`

DOMAIN NAMES

- 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
  - utilizzare 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
  - un nome può essere mappato a più indirizzi IP
- indirizzi a lunghezza fissa verso nomi a lunghezza variabili
- Domain Name System (DNS) traduce nomi in indirizzi IP

LA CLASSE INETADDRESS

- oggetto di tipo `InetAddress`
  - coppia indirizzi IP (`byte[] address`) e nomi simbolici di dominio (`String`)
  - nessun costruttore, una factory con metodi statici
  - i metodi si connettono al DNS per risolvere un hostname, ovvero trovare l'indirizzo IP ad esso corrispondente: necessaria connessione di rete
  - possono sollevare `UnKnownHostException`, se non riescono a risolvere il nome dell'host

```
import java.net.*;
public class FindIP {
 public static void main (String[] args) {
    try {
        InetAddress address = InetAddress.getByName("www.unipi.it");
        System.out.println(address);
    } catch (UnknownHostException ex) {
      System.out.println("Could not find www.unipi.it"); }} }
```

```
$ java FindIP
www.unipi.it/131.114.21.42
```

ALTRI METODI

- `getAllByName()` lookup di tutti gli indirizzi di un host

```
$ java FindAllIP
www.repubblica.it/18.66.196.45
www.repubblica.it/18.66.196.118
www.repubblica.it/18.66.196.94
www.repubblica.it/18.66.196.112
```

```
import java.net.*;
public class FindAllIP {
 public static void main (String[] args) {
  try { InetAddress [] addresses = InetAddress.getAllByName("www.repubblica.it");
    for(InetAddress address:addresses)
        { System.out.println(address); }
 } catch (UnknownHostException ex) {
                    System.out.println("Could not find www.repubblica.it");}}}
```

- `getLocalHost()` restituisce l' InetAddress del local host

```
import java.net.*;
public class MyAddress {
 public static void main (String[] args) {
 try {
   InetAddress address = InetAddress.getLocalHost();
   System.out.println(address);
 } catch (UnknownHostException ex)
          {System.out.println("Could not find this computer address"); }}}
```

INETADDRESS: CACHING

- i metodi descritti effettuano caching dei nomi/indirizzi risolti
  - l'accesso al DNS è una operazione potenzialmente molto costosa
  - nomi risolti con i dati nella cache, quando possibile (di default: per sempre)
  - 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.
- controllo dei tempi di permanenza in cache

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

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

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

UN PROGRAMMA UTILE: SPAM CHECKER

- 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
- `SpamCheck` richiede a `sbl.spamhaus.org` se un indirizzo IPv4 è uno spammer noto
  - es una query DNS su `17.34.87.207.sbl.spamhaus.org` ha successo se l'indirizzo è uno spammer

UN PROGRAMMA UTILE: SPAM CHECKER

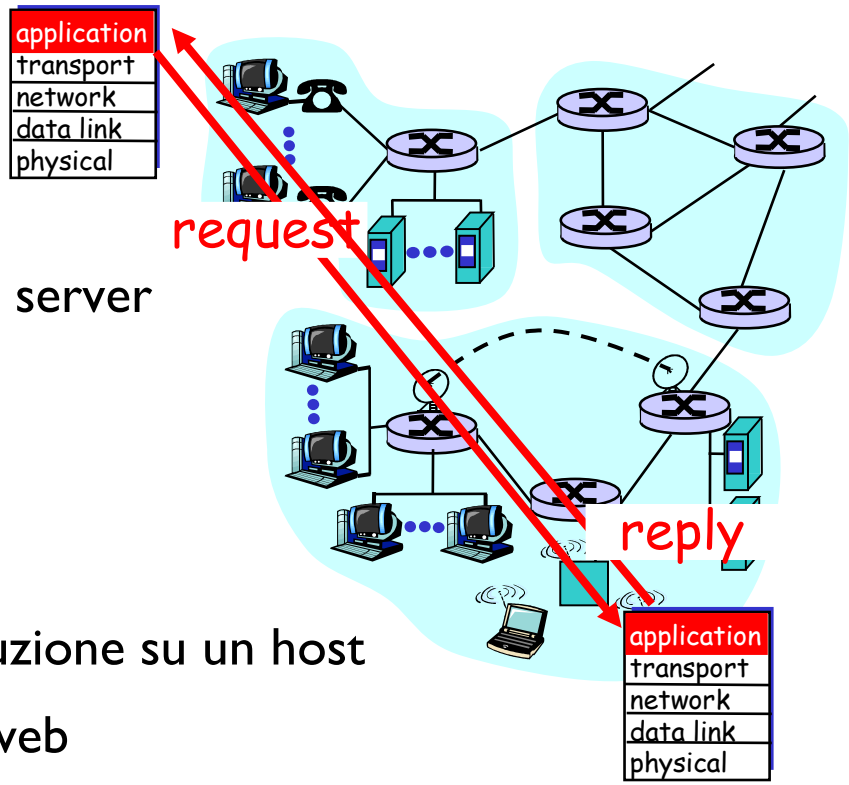
```
import java.net.*;
public class SpamCheck {
  public static final String BLACKHOLE = "sbl.spamhaus.org";
  public static void main(String[] args) throws  UnknownHostException
       { for (String arg: args) {
                if (isSpammer(arg)) {
                    System.out.println(arg + " is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

```
$java SpamCheck  23.45.65.88 141.250.89.99
127.0.0.2

23.45.65.88 appears legitimate.
141.250.89.99 appears legitimate.
127.0.0.2 is a known spammer
```

IL PARADIGMA CLIENT/SERVER

servizio:

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



client:

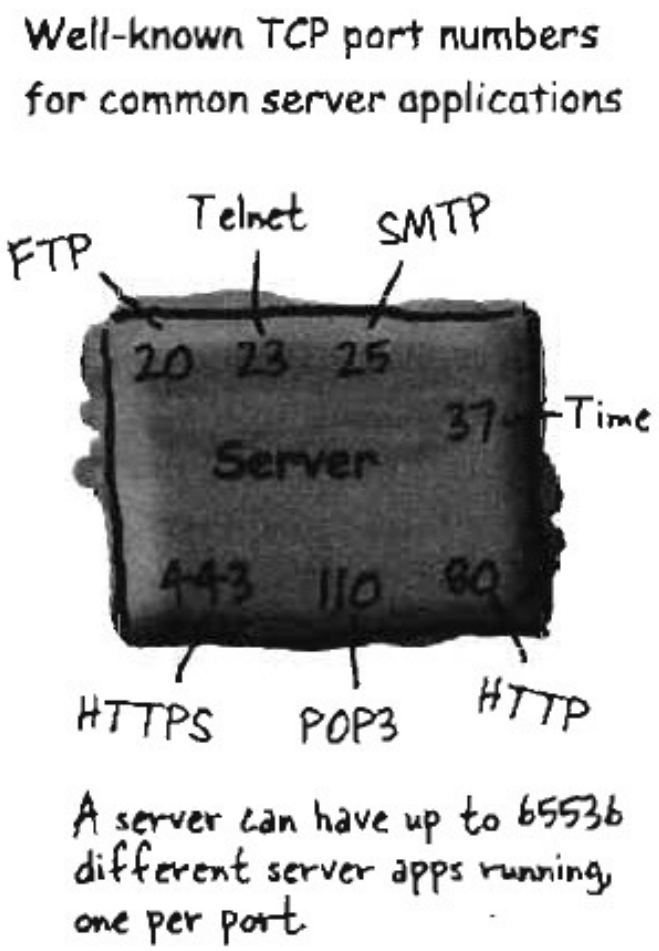
- un software che sfrutta servizi forniti dal server
  - web client browser
  - e-mail client mail-reader

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

IDENTIFICARE I SERVIZI

- occorre specificare:
  - l'host, tramite indirizzo IP (la rete all'interno della quale si trova l'host + l'host all'interno della rete)
  - la porta individua un servizio tra I tanti servizi (es: e-mail, ftp, http,…) attivi su un host
- ogni servizio individuato da una porta
  - intero tra 1 e 65535 (per TCP ed UDP)
  - non un dispositivo fisico, ma un'astrazione per individuare i singoli servizi (processi)
- porte 1–1023: riservate per well-known services.

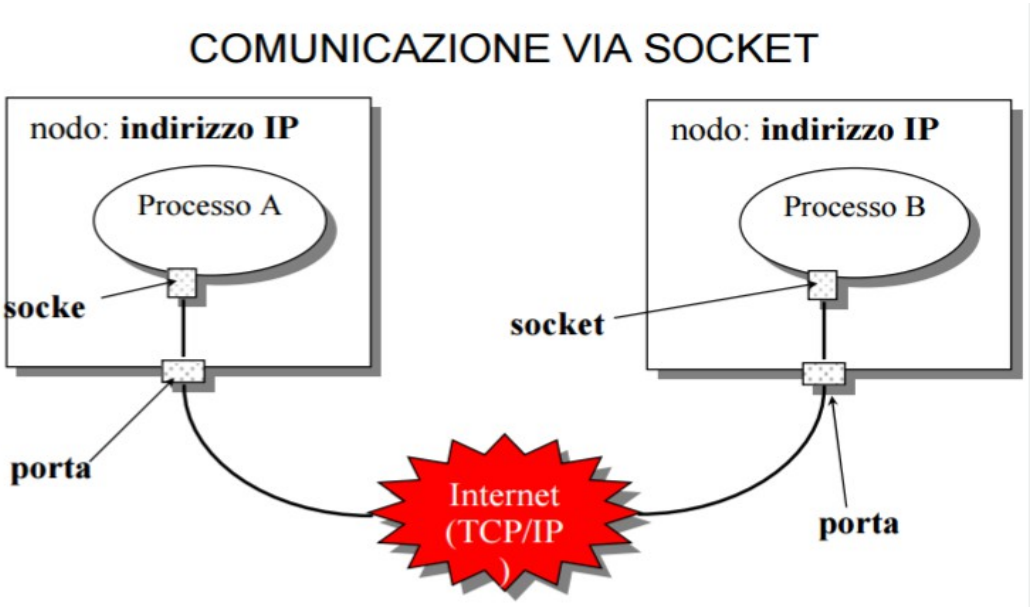


CONNETTERSI AD UN SERVIZIO

- socket: 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



COME IL CLIENT ACCEDE AD UN SERVIZIO

- per usufruire di un servizio, il client apre un socket individando
  - host + porta che identificano il servizio
  - invia/riceve messaggi su/da uno stream
- in JAVA: `java.net.Socket`
  - usa codice nativo per comunicare con lo stack TCP locale

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

- crea un socket su una porta effimera 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

- ricerca quale delle prime 1024 porte di un host è associata ad un servizio

```
import java.net.*;
import java.io.*;
public class LowPortScanner {
  public static void main(String[] args) {
    String host = args.length > 0 ? args[0] : "localhost";
    for (int i = 1; i < 1024; i++) {
      try {
          Socket s = new Socket(host, i);
          System.out.println("There is a server on port " + i + " of " + host);
          s.close();
      } catch (UnknownHostException ex) {
                  System.err.println(ex);
                  break;
      } catch (IOException ex) {
      // must not be a server on this port
     }}}}
```

```
$java LowPortScanner

There is a server on port 80  of localhost
There is a server on port 135 of localhost
There is a server on port 445 of localhost
There is a server on port 843 of localhost
```

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
- il programma precedente può effettuare 1024 interrogazioni al DNS, a meno di meccanismi di caching nel sistema operativo
- soluzione alternativa:

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

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

MODELLARE UNA CONNESSIONE MEDIANTE STREAM

- una volta stabilita una connessione tra client e server devono scambiarsi dei dati. La connessione è 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

INTERAGIRE CON IL SERVER TRAMITE SOCKET

- client implementato in JAVA, server in qualsiasi altro linguaggio
  - aprire un socket `sock` sulla porta su cui è attivo il servizio
  - utilizzare gli stream per la comunicazione con il servizio
- occorre conoscere il protocollo ed il formato dei dati scambiati, che sono codificati in un formato interscambiabile
  - testo
  - `JSON`
  - `XML`
- possibile conoscere il formato dei dati scambiati interagendo con il server tramite il protocollo `telnet`

DAYTIME PROTOCOL (RFC 867)

- aprire una connessione sulla porta 13, verso il servizio `time.nist.gov (NIST: National Institute of Standards and Technology)`

```
$ telnet time.nist.gov 13
Trying 129.6.15.28...
Connected to time.nist.gov.
Escape character is '^]'.

56375 13-03-24 13:37:50 50 0 0 888.8 UTC(NIST) *
Connection closed by foreign host.
```

Format: `JJJJJ YY-MM-DD HH:MM:SS TT L H msADV UTC(NIST) OTM`

- `JJJJJ`: Modified Julinan Date (days since Nov 17, 1858)
- `TT`: 00 means standard time and 50 means daylight savings time
- `L`: indicates whether a leap second will be added (1) or subtracted (2)
- `H`: health of the server (0: healthy; 1: up to 5 seconds off; …)
- `msADV`: how long (ms) it estimates it's going to take for the response to return
- `UTC (NIST)`: time-zone constant string
- `OTM`: almost a constant (an asterisk)

DAYTIME PROTOCOL CLIENT

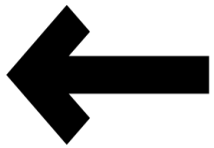
```
public class TimeClient {
  public static void main(String[] args) {
    String hostname = args.length > 0 ? args[0] : "time.nist.gov";
    Socket socket = null;
    try {
      socket = new Socket(hostname, 13);
      socket.setSoTimeout(15000);
      InputStream in = socket.getInputStream();
      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("could not connect to
                                   time.nist.gov");
 } finally {
      if (socket != null) {
        try {
          socket.close();
        } catch (IOException ex) {// ignore }}}}}
```

setSoTimeout(<ms>): setta un timeout sul socket

- previene attese indeterminate di risposte dal server
- solleva SocketTimeoutException (è una IOException)

DAYTIME CON TRY WITH RESOURCES

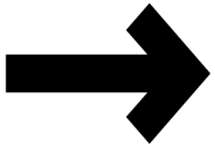
```
try { socket = new Socket(hostname, 13);
      //read from the socket
 } catch (IOException ex)
      {System.out.println("could not connect to time.nist.gov");}

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



Java 6-: rilascio esplicito delle risorse

Java 7+: autochiusura tramite try with resources

clausola finally non necessaria



```
try (Socket socket = new Socket("time.nist.gov", 13))
        {//read from the socket
        } catch (IOException ex)
      { System.out.println("could not
           connect to time.nist.gov");}
```

TRY WITH RESOURCES

- introdotto in `JAVA 7`, aggiornato in `JAVA 9`
- chiusura sistematica ed automatica delle risorse usate da un programma
  - `Inputstream`, netwok connection, database connection
- un blocco `try` con uno o più argomenti tra parentesi.
  - argomenti = risorse che devono essere chiuse quando il `try block` termina
  - le variabili che rappresentano le risorse non devono essere riutilizzate
- suppressed exceptions:
  - quando si verificano delle eccezioni sia nel blocco try-with-resources sia durante la chiusura, la `JVM` sopprime l'eccezione generata nella chiusura automatica.
- generalizzazione: implementazione della `AutoCloseable` interface

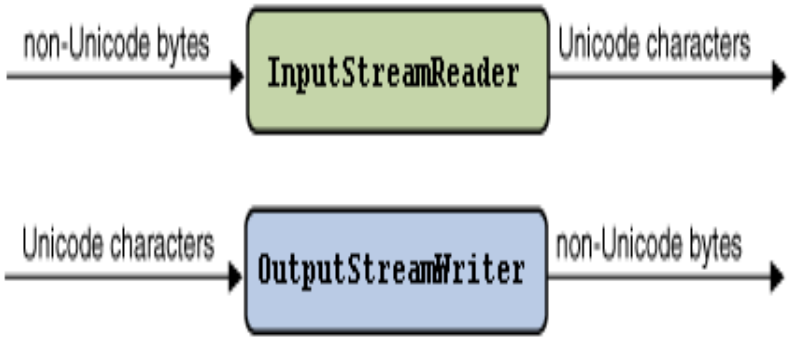
TRY WITH RESOURCES

```
import java.io.*;
public class trywithresources
  { public static void main (String args[])throws IOException {
    try(FileInputStream input = new FileInputStream(new  File("immagine.jpg"));
        BufferedInputStream bufferedInput = new  BufferedInputStream(input))
        {
          int data = bufferedInput.read();
          while(data != -1){
             System.out.print((char) data);
              data = bufferedInput.read();
          }}}}
```

- risolve il problema delle "suppressed exceptions"
  - eccezioni possono essere sollevate nel blocco `try`, oppure nel blocco `finally`,
  - un'eccezione rilevata nella `finally` sopprime l'eccezione rilevata nel blocco `try`
- con il try with resources viene propagata l'eccezione rilevata nel blocco `try`

DAYTIME: LEGGERE CARATTERI

- utilizza `InputStreamReader`
- istanziato su un `InputStream`
- parametro
  - codifica dei caratteri presenti sullo stream di byte (`ASCII`, `UTF-8`, `UTF-16`,...)
- traduce caratteri esterni nella codifica interna Unicode




```
                    …...
InputStream in =
          socket.getInputStream();
StringBuilder time = new
                    StringBuilder();
InputStreamReader reader = new
    InputStreamReader(in, "ASCII");
for (int c=reader.read();c != -1;
         {
     time.append((char) c);  }
                    …...
```

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`

COSTRUZIONE SOCKET SENZA CONNESSIONE

- costruttore senza argomenti e connessione successiva

```
try {
    Socket socket = new Socket();
    // setta opzioni Socket, ad esempio timeout
     SocketAdrress = new InetSocketAddress ("time.nist.gov", 13);
     socket.bind(conect(address));
    // utilizza il socket
    } catch (IOException ex) {System.out.println(err); }}
```

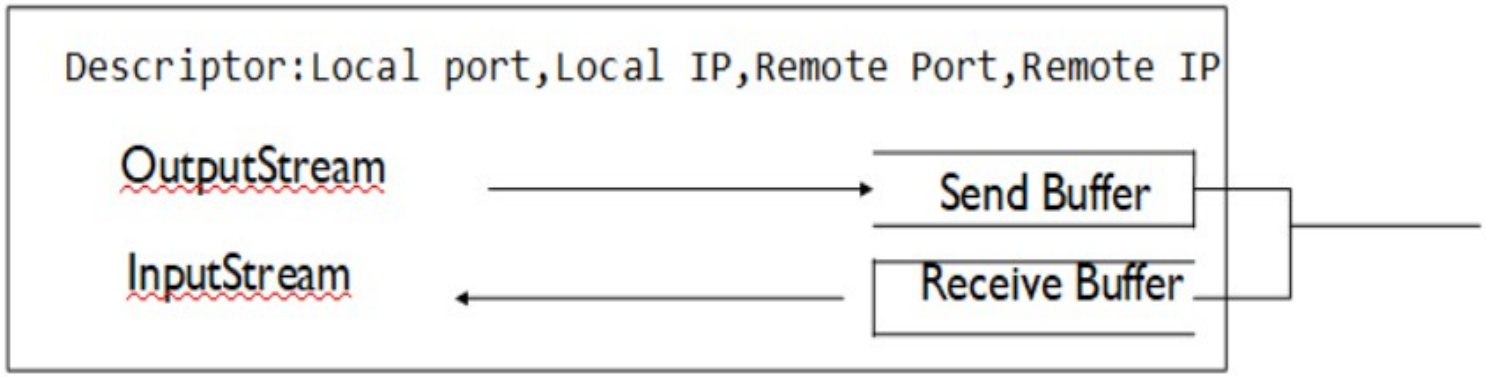
- scritture successive sollevano una `IOException`
- `InetSocketAddress`: costruttori

```
public InetSocketAddress (InetAddress address, int port);
public InetSocketAddress(String host, int port);
public InetSocketAddress (int port);
```

REPERIRE INFORMAZIONI SU UN SOCKET

- metodi getter

```
public InetAddress getInetAddress()   } indirizzo e porta
public int getPort()                  } host remoto
public InetAddress getLocalAddress(   } indirizzo e porta
public int getLocalPort()             } host locale
```



Struttura del Socket TCP

REPERIRE INFORMAZIONI SU UN SOCKET

```
import java.net.*;
import java.io.*;
public class SocketInfo {
public static void main(String [] args)
{ for (String host: args) {
   try {
     Socket theSocket = new Socket (host, 80);
     System.out.println("Connected to "+theSocket.getInetAddress()
     +" on port"+ theSocket.getPort()+ " from port "
     + theSocket.getLocalPort() + " of"
      + theSocket.getLocalAddress());
   } catch(UnknownHostException ex) {
     System.out.println("I cannot find"+host);}
    catch(SocketException ex) {
     System.out.println("Could not connect to"+host);}
    catch(IOException ex) { System.out.println(ex);}}}}
```

```
$ java SocketInfo www.repubblica.it www.google.com
Connected to www.repubblica.it/18.66.196.94
on port 80 from port 56261 of/192.168.1.146
Connected to www.google.com/142.250.180.164
on port 80 from port 56262 of/192.168.1.146
```

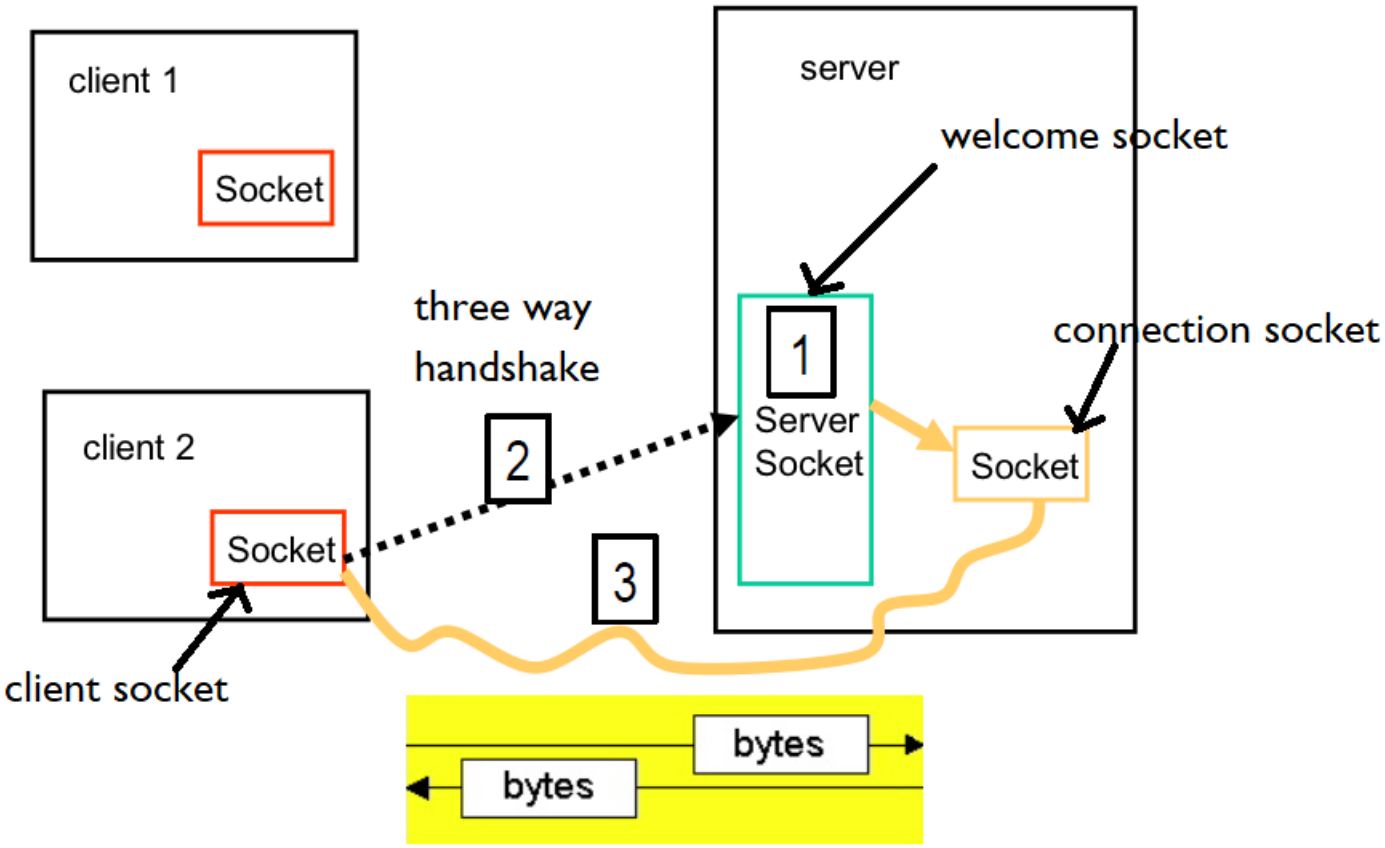
RIASSUNTO

identificazione di un servizio con cui comunicare, occorre individuare:

- 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}

ANCHE IL SERVER IMPLEMENTATO IN JAVA



ANCHE IL SERVER IMPLEMENTATO 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

ANCHE IL SERVER IMPLEMENTATO IN JAVA

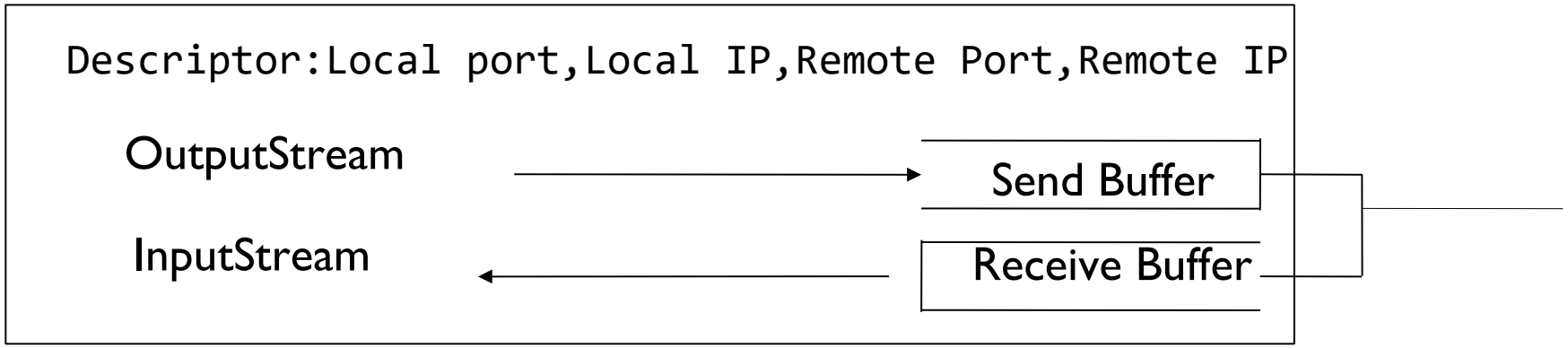
- il server pubblica un proprio servizio
  - gli associa un welcome socket, sulla porta remota PS, all'indirizzo IP IPS
  - usa un oggetto di tipo ServerSocket
- il client crea un Socket su IPS + PS
- la creazione del socket effettuata dal client produce in modo atomico la richiesta di connessione al server
  - three way handshake col server completamente gestito dal supporto
  - se la richiesta viene accettata,
    - il server crea un socket dedicato per l'interazione con quel client
    - tutti i messaggi spediti dal client vengono diretti automaticamente sul nuovo socket creato.

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

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.
- bloccante: se non ci sono richieste, il server si blocca (possibile utilizzo di time-outs)
- quando 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);}
         } }
```

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 wotk
      sock.close();
              }
servSock.close();
```

DAYTIME SERVER

```
import java.net.*;
import java.io.*;
import java.util.Date;
public class DayTimeServer {
public final static int PORT = 1313;
  public static void main(String[] args) {
   try (ServerSocket server = new ServerSocket(PORT)) {
     while (true) {
       try (Socket connection = server.accept()) {
         Writer out = new OutputStreamWriter(connection.getOutputStream());
         Date now = new Date();
         out.write(now.toString() +"\r\n");
         out.flush();
         connection.close();
       } catch (IOException ex) {}
     }
   } catch (IOException ex) {
     System.err.println(ex); } } }
```

porte 0-1023 privilegiate

si ferma qui ed aspetta, quando un client si connette restituisce un nuovo Socket

servizio della richiesta

chiude la connessione e torna ad accettare nuove richieste

try-with-resource: autoclose

DAYTIME SERVER: CONNETTERSI CON TELNET

```
import java.net.*;
import java.io.*;
import java.util.Date;
public class DayTimeServer {
 public final static int PORT = 13;
 public static void main(String[] args) {
   try (ServerSocket server = new ServerSocket(PORT)) {
     while (true) {
       try (Socket connection = server.accept()) {
         Writer out = new OutputStreamWriter(connection.getOutputStream());
         Date now = new Date();
         out.write(now.toString() +"\r\n");
         out.flush();
         connection.close();
       } catch (IOException ex) {}
     }  } catch (IOException ex) {System.err.println(ex);}}}
```

```
$ telnet localhost 1333
trying 127.0.0.1....
connected to localhost
San Oct 17 23:16:12 CEST 2021
```

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: SERVER

```
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();}
        }
    }
```

ASSIGNMENT: 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