UNIVERSITÀ DI PISA

Programmazione di reti
Corso B

18 Ottobre 2016
Lezione 5

Contenuti

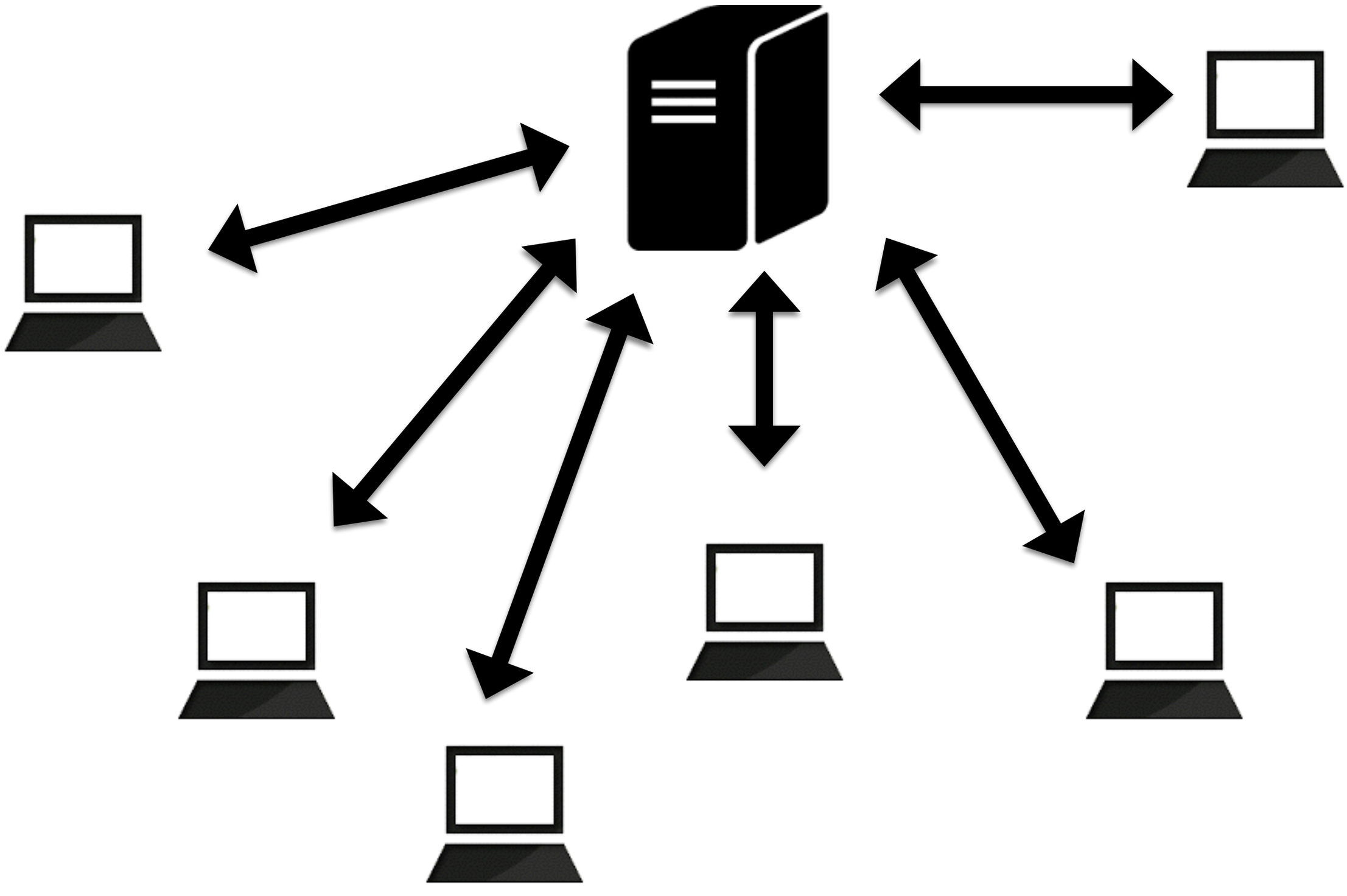
- Programmazione di reti
- Indirizzi di rete
- Java TCP *socket*

Programmazione di reti

- Programmi che comunicano usando la rete
- Paradigmi
  - *Client-server*
  - *Peer-to-peer*

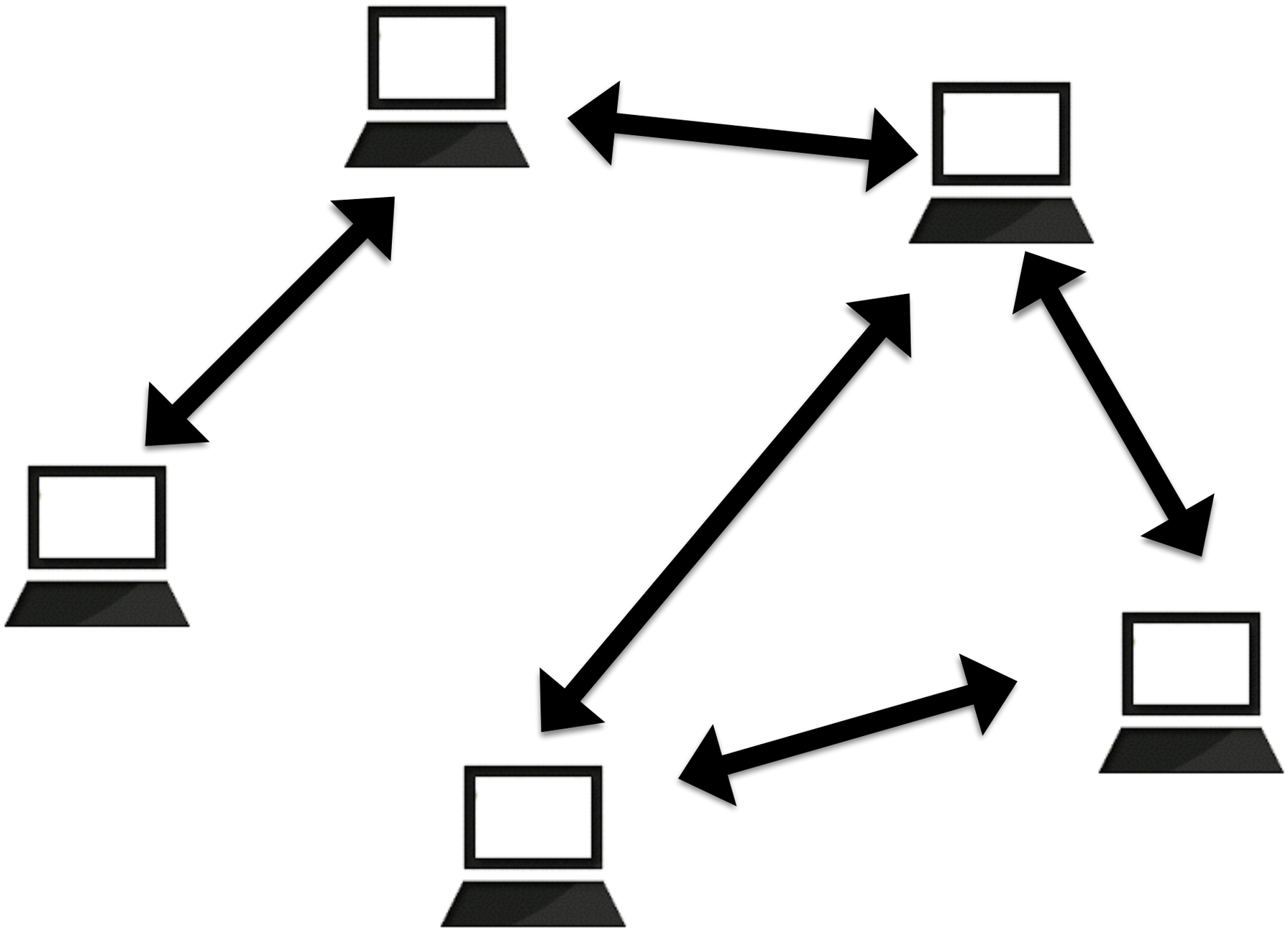
Paradigma *client-server*

- *Server*
  - offre un servizio ai clienti
  - gestisce la logica del programma - ha una visione completa del sistema
- Cliente
  - usa il servizio offerto
  - ha una conoscenza limitata del sistema



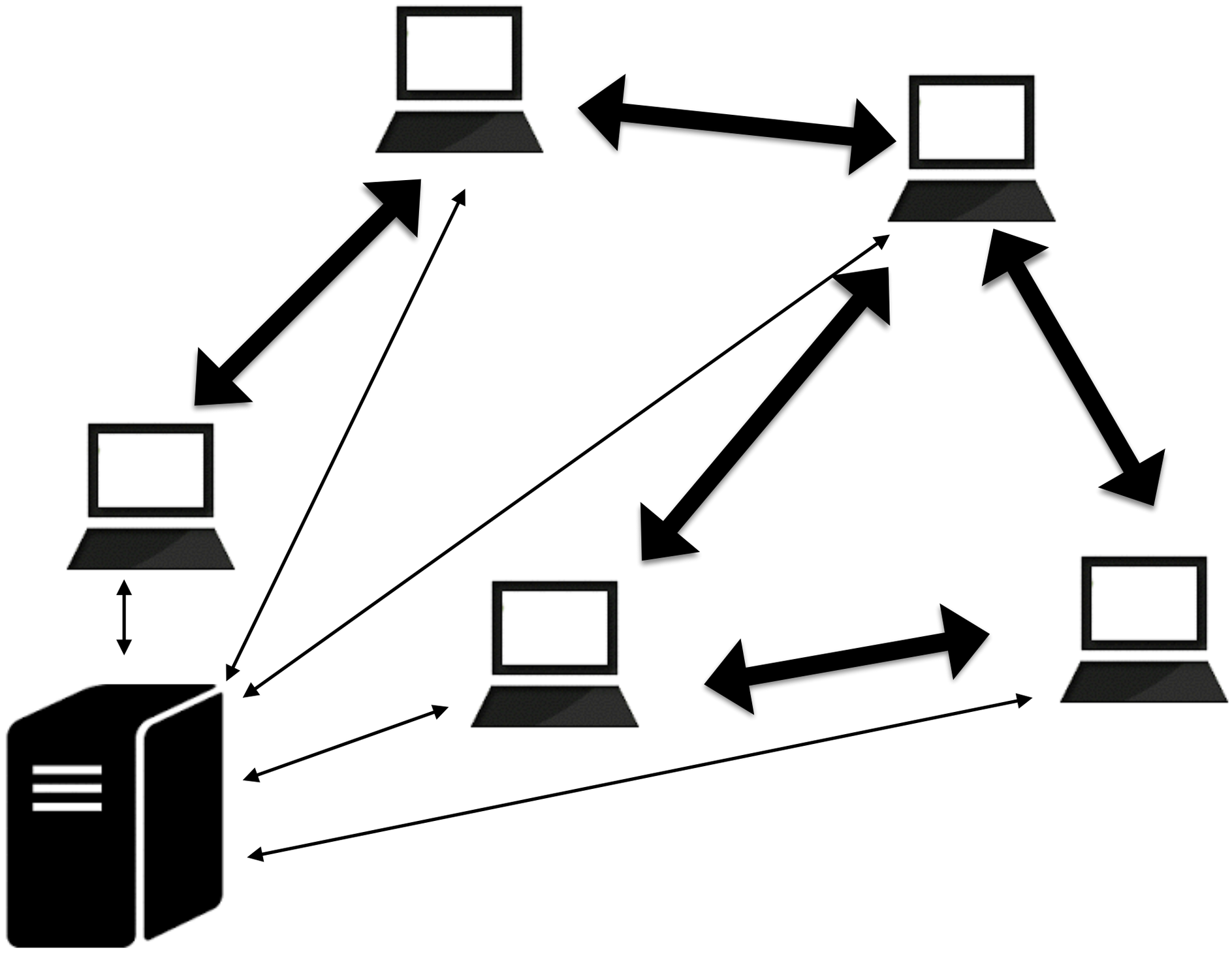
Paradigma *peer-to-peer*

- Tutti i nodi sono uguali
- Applicazioni comunicano direttamente, senza un server
- Tutti i nodi hanno una conoscenza parziale del sistema



Sistemi ibridi

- Un *server* aiuta a trovare i *peer* con cui comunicare
  - La comunicazione avviene *peer-to-peer*
- Super-nodi che fanno anche da *server*



Comunicazione tra applicazioni

- Usando *Socket* - a distanza - usando la rete
- Primo passo : identificare le applicazioni nella rete
- Secondo passo: inviare i dati

- Si usano pacchetti chiamati *datagram* - contengono indirizzo e dati
- *Connection-based* : TCP - in ordine - usando s*ocket*
- *Packet-based* : UDP - messaggi indipendenti, più veloce, meno affidabile

Indirizzi

- ogni host della rete viene identificata dall'IP (IPv4 o IPv6)

- difficili da ricordare => *hostname* più facili (e.g. elearning.unipi.it)

- DNS (*domain name system*) - mappa gli indirizzi IP alla *hostname*

- in Java : `InetAddress`

InetAddress

- Memorizzano l'IP e il *hostname*
- Creati usando metodi statici *factory*

`InetAddress getLocalHost()`
restituisce l'indirizzo e nome del *local host*

`InetAddress  getLoopbackAddress()`
indirizzo 127.0.0.1, nome "localhost"

InetAddress

- `static InetAddress getByName(String name)`

```
InetAddress address =
InetAddress.getByName(“elearning.unipi.it”);
```

Lancia eccezione se hostname non esiste. Se **`name`** è **`null`**, restituisce indirizzo *loopback*.

```
InetAddress address =
InetAddress.getByName(“131.114.18.105”);
```

Non lancia eccezione, mette *hostname*=“131.114.18.105”

- `static InetAddress  getByAddress(byte[] addr)`

```
byte[] rawAddress={(byte)131,114,18,105};
InetAddress address = InetAddress.getByAddress(rawAddress);
```

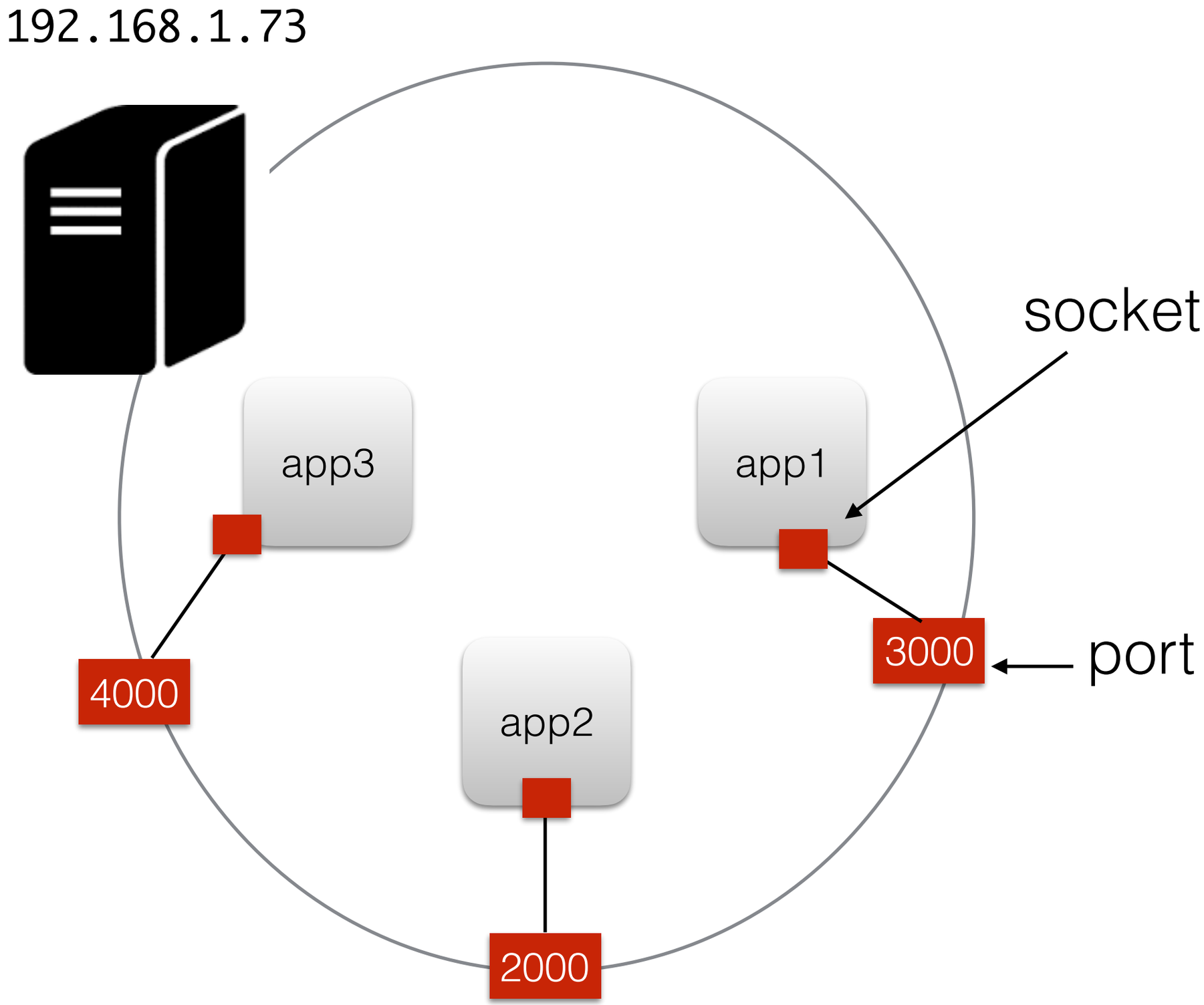
Non lancia eccezione, mette *hostname*=“131.114.18.105”

```
public class Address {
   public static void main(String[] args) {
      String stringAddress="131.114.18.105";
      String name="elearning.unipi.it";
      byte[] rawAddress={(byte)216,58,(byte)212,68};
      try {
         InetAddress address = InetAddress.getByName(name);
         System.out.println("Found "+address.getHostName()+
               " with IP "+address.getHostAddress());
         address = InetAddress.getByName(stringAddress);
         System.out.println("Found "+address.getHostName()+
               " with IP "+address.getHostAddress());
         address = InetAddress.getByAddress(rawAddress);
         System.out.println("Found "+address.getHostName()+
               " with IP "+address.getHostAddress());
         address = InetAddress.getLocalHost();
         System.out.println("Running on "+address.getHostName()+
               " with IP "+address.getHostAddress());
         address = InetAddress.getLoopbackAddress();
         System.out.println("Local "+address.getHostName()+
               " with IP "+address.getHostAddress());
      } catch (UnknownHostException e) {
         System.out.println("No such host: "+name);
      } catch (IOException e) {e.printStackTrace();}
   }
}
```

```
Found elearning.unipi.it with IP 131.114.18.105
Found hosting1.sid.unipi.it with IP 131.114.18.105
Found mil01s24-in-f68.1e100.net with IP 216.58.212.68
Running on Alinas-Air.lan with IP 192.168.1.73
Local localhost with IP 127.0.0.1
```

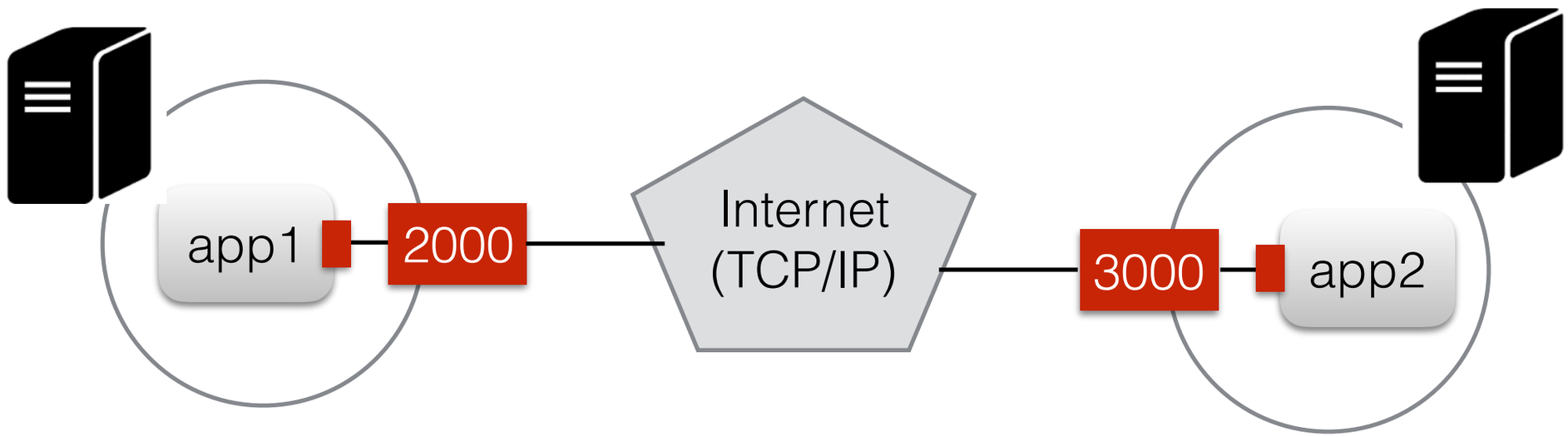
Indirizzi

- le applicazioni di rete usano i *socket* per comunicare
- un *socket* viene identificato da un numero di *port* sulla *host* (numero intero tra 0 e 65535)
- qualche *port* sono riservati per vari servizi/protocolli - si possono comunque usare se non già usati sulla machina
- su sistemi UNIX solo l'utente *root* può usare i port fino a 1023
- un IP + un numero di *port* identificano un'applicazione su internet!!!


192.168.1.73
app3
app1
app2
4000
3000
2000
socket
port

Comunicazione TCP

- Tramite *socket*
- un *socket* è una estremità (end-point) di una connessione bidirezionale tra due programmi eseguiti nella rete
- Classe `java.net.Socket` usata da cliente e *server*
- Classe `java.net.ServerSocket` usata dal *server*



Comunicazione TCP

- Passi:
  - Il *server* apre un `ServerSocket` e aspetta connessioni dai clienti
  - Il cliente apre un `Socket` e si connette al *server*
  - Quando un cliente arriva, il *server* crea un `Socket` dedicato a questo cliente
  - *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)

Socket

- Creare un *socket* e aprire una connessione

```
Socket(InetAddress address, int port)

Socket(InetAddress address, int port, InetAddress
localAddr, int localPort)
```

  - localAddr=null => qualsiasi indirizzo locale, localPort=0 => qualsiasi port locale

  - lancia eccezioni (`UnknownHostExceptio, IOException`)

```
Socket(String address, int port)

Socket(String address, int port, InetAddress localAddr,
int localPort)
```

Socket

- Creare un *socket* senza aprire una connessione

`Socket()`

- Aprire la connessione con metodo **`connect()`** - dopo aver creato un **`SocketAddress`**

```
Socket socket = new Socket();
SocketAddress address = new
InetSocketAddress(“www.google.com”, 80);
socket.connect(address);
```

```
public class SocketProps{
   public static void main(String[] args) {
      Socket socket = new Socket();
      try {
         SocketAddress address = new InetSocketAddress("www.google.com", 80);
         socket.connect(address);
         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 teh host ");
      } catch (IOException ex) {
      System.err.println(ex);
      } finally {
         try {
            socket.close();
         } catch (IOException e) {}
      }
   }
}
```

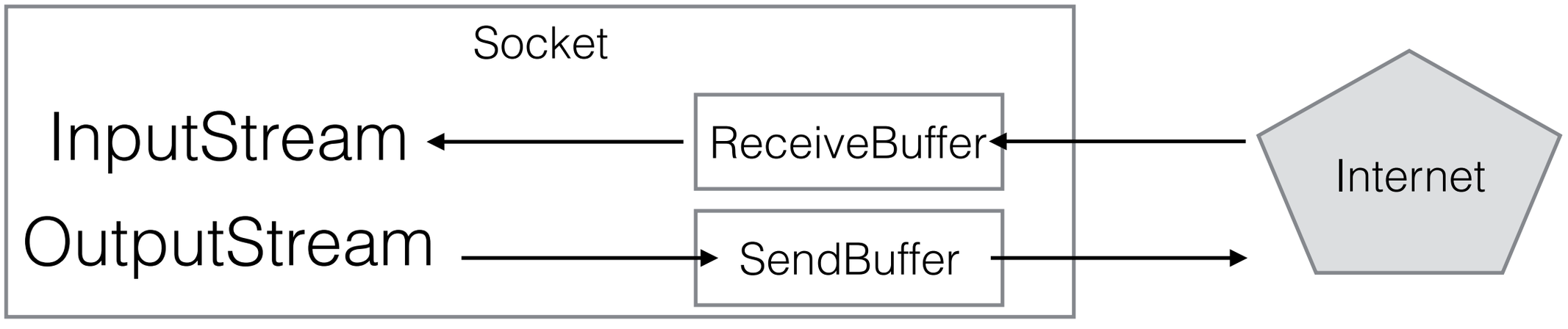
```
Connected to www.google.com/172.217.16.4 on port 80 from port 58386 of /131.114.218.134
```

Socket

- Ottenere i *stream* per inviare e ricevere messaggi

```
InputStream  getInputStream()
OutputStream getOutputStream()
```

- Lanciano `IOException` se *socket* chiuso o sconnesso.



Socket

- Connessione è *full duplex* - può diventare non valida (e.g. *connection reset*) o chiusa : le operazioni *read/write* lanciano `IOException`.

- Gli *input/output stream* sono standard: usati come prima, abbinandoli con *filter* o *reader/writer*

- Protocollo viene definito dal programmatore

Socket

- Fissare un *timeout* : evita blocchi causati da un server non-responsive

```
void setSoTimeout(int timeout)
```

  - lancia `java.net.SocketTimeoutException` se un'operazione (read/write) prende più di timeout, pero il *socket* rimane valido

Socket

- Chiudere la connessione - rilascia il *port* sulla *host* - chiude gli *stream* associati

`void close() throws IOException`

- Se un altro thread e bloccato in read/write su questo socket lancia `SocketException`

- Una volta chiuso un socket non può essere riaperto.

Socket

- Chiudere parzialmente la connessione - rilascia il *port* sulla *host*

```
void shutdownInput() throws IOException
void shutdownOutput() throws IOExceptio
```

- L'input stream viene fissato su end of stream
- L'output stream lancia `IOException`

Socket

- Altre opzioni:

`void setTcpNoDelay(boolean on) throws IOException`
invia dati subito dopo `send()` senza aspettare che pacchetto sia pieno

`void setSoLinger(boolean on, int seconds) throws IOException`
se ci sono pacchetti da inviare dopo `Socket.close()`, questi vengono inviati comunque se *linger* e OFF. Se ON, i pacchetti vengono scartati dopo aver aspettato il numero di secondi (metodo `close()` si blocca).

`void setReceiveBufferSize(int size)`
`throws IOException, IllegalArgumentException`
`void setSendBufferSize(int size)`
`throws IOException, IllegalArgumentException`
cambia la dimensione dei buffer - sono sempre uguali - può essere ignorato

Socket

- Altre opzioni:

`public void setKeepAlive(boolean on) throws IOException`

Attiva verifiche di *keepalive*. Se *server* non risponde dopo qualche minuto il *socket* viene chiuso

`public void setReuseAddress(boolean on) throws IOException`

Dopo aver chiuso il *socket*, il *port* diventa riutilizzabile da subito. Deve essere richiamato prima di `connect()`

Verificare connettività

`boolean isClosed()`

`true` se il *socket* e stato chiuso

`boolean isConnected()`

`true` se il *socket* è mai stato connesso

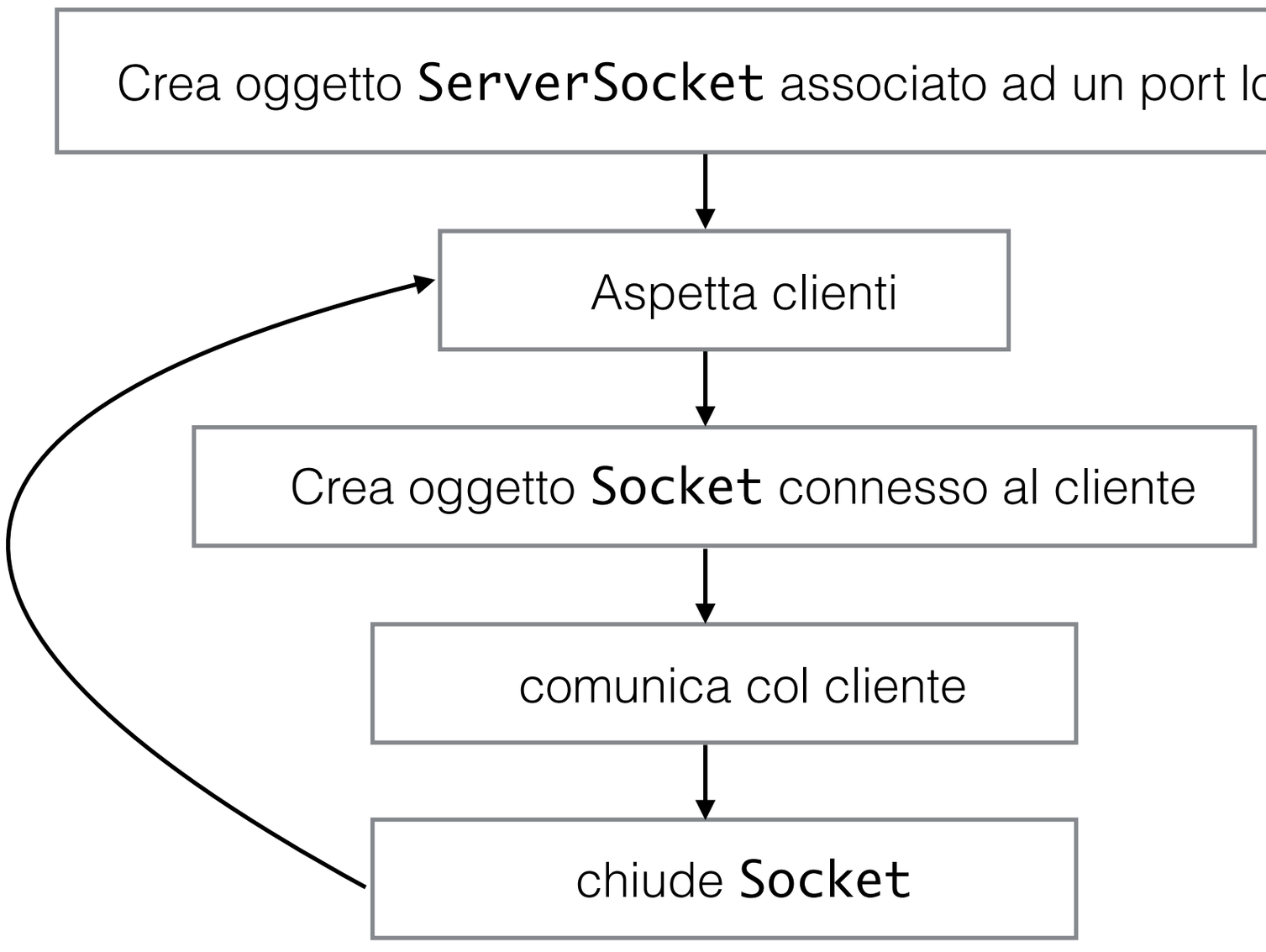
per verificare se un *socket* ha una connessione attiva:
`isConnected() && ! isClosed()`

ServerSocket

- Un *socket* speciale che aspetta delle connessioni
- Non sa in anticipo ne chi sono i clienti ne quando arriveranno
- Con `Socket` se l'altra parte della connessione non è pronta, il *socket* non si connette
- il `ServerSocket` esiste ed è attivo anche senza nessun cliente
- i `ServerSocket` non sono usati per trasmettere dati dall'applicazione, servono solo ad aspettare, negoziare la connessione con i nuovi clienti e creare un `Socket` normale per usare nell'applicazione.

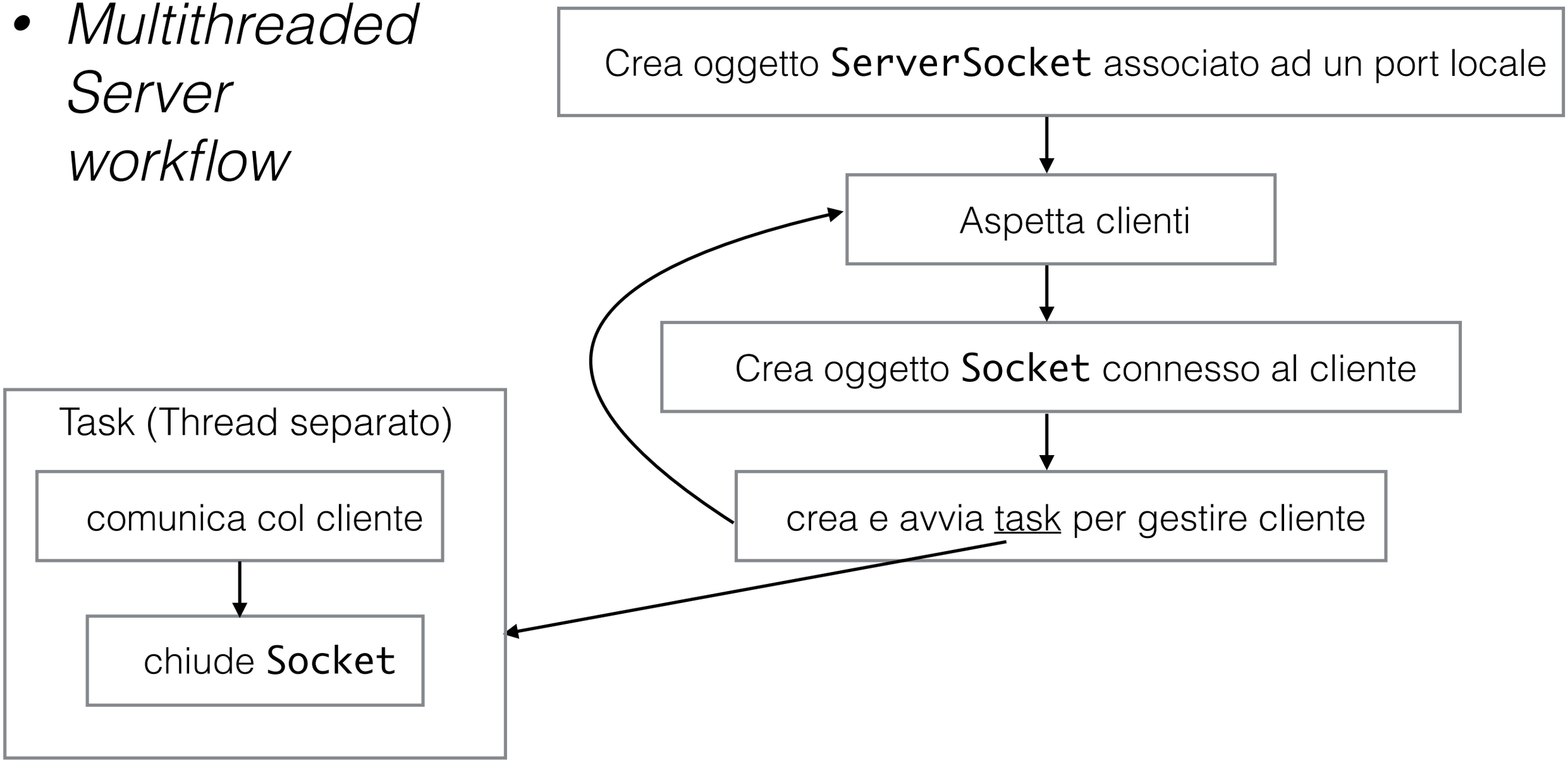
ServerSocket

- *Server workflow*



ServerSocket

- *Multithreaded Server workflow*



Creare `ServerSocket`

- *Socket* già associato con un port (bound)

  `public ServerSocket(int port)`

  - usare 0 per scegliere un port a caso

  `public ServerSocket(int port, int backlog)`

  - *backlog* indica il numero di richieste da mettere in coda (le altre vengono respinte). Il numero può essere ignorato o limitato.

  `public ServerSocket(int port,int backlog, InetAddress localAddress)`

  - associa al *socket* solo l'interfaccia specificata (altrimenti il *server* ascolta su tutte le interfacce attive)

Creare `ServerSocket`

- Socket non associato con un port (unbound) - bisogna richiamare metodo `bind()`
  - `public ServerSocket()`
  - `public void bind(SocketAddress endpoint)`

Aspettare connessioni

`public Socket accept()`

- si blocca fin che un cliente arriva
- restituisce un `Socket` connesso al cliente
- Il nuovo *socket* funziona esattamente come il *socket* cliente di prima, può essere utilizzato per inviare o ricevere dati

Metodi

`public void close()`
chiude il *server socket* e anche i *client socket* generati - gli socket sono `AutoCloseable`

`public boolean isClosed()`
`true` se il *socket* e chiuso

`public boolean isBound()`
`true` se *socket* ha mai fatto `bind` (anche nel costruttore)

Per verificare se un *server socket* è aperto e attivo, dobbiamo usare `isBound() && !isClosed()`

Metodi

```
public InetAddress getInetAddress()
public int getLocalPort()

public void setSoTimeout(int timeout) throws
SocketException
```

Imposta il tempo per cui `accept()` si blocca. Lancia `SocketTimeoutException` se nessun cliente arriva , però il *socket* rimane attivo.

```
public void setReuseAddress(boolean on) throws
SocketException
```

Se attivata, dopo aver chiuso il *socket*, il port diventa riutilizzabile da subito. Deve essere richiamato prima di `bind()`

Eccezioni

Vari metodi dichiarano di lanciare `IOException`. In realtà, lanciano sottoclassi di `IOException`

`class SocketException extends IOException`

`class BindException extends SocketException`
se *socket* e già usato o in (0,1024) e non sono root (su Unix)

`class ConnectException extends SocketException`
`class NoRouteToHostException extends SocketException`
`class ProtocolException extends IOException`

`class SocketTimeoutException extends InterruptedIOException`
timeout su `ServerSocket.accept()`

si possono gestire tutti i tipi di eccezioni per avere un controllo dettagliato

```
public class EchoServer {
   public static void main(String[] args) {
      try (ServerSocket server= new ServerSocket();) {
         server.setReceiveBufferSize(100);
         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()));){
               while ((message= reader.readLine() )!=null){
                     System.out.println("Client sent: "+message);
                     writer.write(message+"\r\n");
                     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();
   }}}
```

```
public class EchoClient {
   public static void main(String[] args) {
      Socket socket=new Socket();
      BufferedReader reader=null;
      BufferedWriter writer=null;
      try{
         socket.setSoTimeout(100000);
         socket.setTcpNoDelay(true);
         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;
while (!(choice= localReader.readLine().trim()).equals("exit"))
{
      writer.write(choice+"\r\n");
      writer.flush();
      reply= reader.readLine();
      System.out.println("Server sent back: "+reply);
}
System.out.println("Communication ended by client “);
      } 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.");
      }
   }
}
```

Tempo di risposta del server

- I clienti sono gestiti uno alla volta
- Se un cliente invia messaggi all'infinito, gli altri non avranno mai risposta (vanno in *timeout*)
- Soluzione: *multithreaded server*

```java
public class MultithreadedEchoServer {

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

```
public 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()));){
      this.client.setSoTimeout(200000);
      String message="";
      while ((message= reader.readLine())!=null){
            System.out.println("Client sent: "+message);
            writer.write(message+"\r\n");
            writer.flush();
      }
   } catch (IOException e){
      System.out.println("Client closed connection or an error appeared.");
   } finally{
      try {this.client.close();
      } catch (IOException e) {}
   }}}
```

DoS attack

- Facendo un *thread* per ogni cliente, il numero di *thread* può crescere velocemente

- il *server* diventa lento o non risponde più

- soluzione: *thread pool*

- in questo caso il nostro *server* è comunque suscettibile a DoS- basta creare dei clienti che non si fermano mai di inviare messaggi

- si può risolvere impostando un numero massimo di messaggi per cliente

```
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(100);
         while (true){
            try { //not try with resources
               Socket client=server.accept();
               EchoClientHandler handler = new EchoClientHandler(client);
               es.submit(handler);
            } catch (IOException e){
               System.out.println("Some error appeared");
            }
         }
      } catch (UnknownHostException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      } finally{
         if (es != null)
            es.shutdown();
      }
   }
}
```

```
public class EchoClientHandler implements Runnable{
    static int MAX_MSG=10;
    Socket client;
    int messageCount;

    public EchoClientHandler(Socket client) {
        this.client=client;
        this.messageCount=0;
    }
    @Override
    public void run() {
        try (BufferedReader reader= new BufferedReader(
                new InputStreamReader(this.client.getInputStream()));
                BufferedWriter writer= new BufferedWriter(
                        new OutputStreamWriter(this.client.getOutputStream()));){
        this.client.setSoTimeout(200000);
        String message="";
        while (this.messageCount<MAX_MSG && (message= reader.readLine())!=null){
                this.messageCount++;
                System.out.println("Client sent: "+message);
                String responseFooter="";
                if(this.messageCount==MAX_MSG)
                    responseFooter=" This was your last message";
                writer.write(message+responseFooter+"\r\n");
                writer.flush();
        }
    } catch (IOException e){
        System.out.println("Client closed connection or an error appeared.");
    } finally{ try {this.client.close();
        } catch (IOException e) {}}}}
```

Inviare oggetti

- Si usano `ObjectInputStream` e `ObjectOutputStream` sopra gli *stream* dei socket.

- Sia il *server* che il cliente devono avere una copia della definizione della classe dell'oggetto

- Metodo più facile: includere nella distribuzione del codice cliente una copia della classe.

Possibilità di *deadlock*

- Aprendo un `ObjectOutputStream` sopra un *stream* di un *socket* (in rete), si invia un *header* con delle informazioni.
- Aprendo un `ObjectInputStream` sopra un stream di un *socket* (in rete), si aspetta il header dall'altra parte.
- Quando si usa `ObjectInputStream` insieme a `ObjectOutputStream`, da entrambe le applicazioni (read e write sul *socket*), è possibile generare *deadlock:*
  - cliente e *server* aprono gli `ObjectInputStream` è si bloccano tutti e due aspettando il header
- Una delle parti deve aprire prima l'*output stream*

Esempio

- Server mantiene un registro studenti
- Cliente chiede al server la lista di studenti con lo stesso cognome
- Server invia la lista

```
public class StudentServer {

    public static void main(String[] args) {
        ArrayList<Student> students= new ArrayList<Student>();
        students.add(new Student("Robert","Brown","Dawson",12));
        students.add(new Student("Michael","Reds","Pearse",40));
        students.add(new Student("Joanna","Moore","Collins",62));
        students.add(new Student("Ann","Brown","Buffallo",132));

        ExecutorService es= Executors.newFixedThreadPool(20);

        try(ServerSocket server= new ServerSocket(1500)){
            while(true){
                System.out.println("Waiting for clients...");
                Socket client=server.accept();
                System.out.println("Client arrived.");
                StudentClientHandler handler= new StudentClientHandler(client,students);
                es.submit(handler);
            }
        } catch (IOException e) {
            System.err.println("Error: "+ e.getMessage());
        } finally {
            es.shutdown();
        }
    }
}
```

```
public class StudentClientHandler implements Runnable {
    Socket client;
    ArrayList<Student> students;
    public StudentClientHandler(Socket client, ArrayList<Student> students) {
        this.students=students;
        this.client=client;
    }
    public void run() {
        System.out.println("Handling client");
        try(ObjectInputStream in= new ObjectInputStream(client.getInputStream());
                ObjectOutputStream out= new ObjectOutputStream(client.getOutputStream());){
            while(true){
                System.out.println("Ready for a new name...");
                String lname=(String) in.readObject();
                System.out.println("Received request for "+lname);
                ArrayList<Student> response= new ArrayList<>();
                for (Student student : this.students){
                    if (student.getLname().trim().toLowerCase().equals(
                            lname.trim().toLowerCase())){
                        response.add(student);
                    }
                }
                out.writeObject(response);
                out.flush();
            }
        } catch (IOException e) {System.err.println("Error: "+ e.getMessage());
        } catch (ClassNotFoundException e) {System.err.println("Class not found: "+
e.getMessage());
        } finally{try {client.close();} catch (IOException e) {}}}}
```

```
public class StudentClient {
    public static void main(String[] args) {
        System.out.println("Connecting to server...");
        try(Socket socket= new Socket(InetAddress.getLocalHost(),1500);
                ObjectInputStream in= new ObjectInputStream(socket.getInputStream());
                ObjectOutputStream out= new ObjectOutputStream(socket.getOutputStream());
                BufferedReader localIn= new BufferedReader(
                                        new InputStreamReader(System.in));)
        {
            System.out.println("Connected.");
            String option=null;
            ArrayList<Student> result=null;
            System.out.println("Insert last name to query server, 'exit' to quit.");
            while( !(option=localIn.readLine()).equals("exit")){
                System.out.println("Asking for students with last name "+option);
                out.writeObject(option);
                out.flush();
                result=(ArrayList<Student>) in.readObject();
                System.out.println("Received "+result.size()+" students with last name "+
option);
                for (Student student: result){
                    System.out.println(student);
                }
                System.out.println("Insert last name to query server, 'exit' to quit.");
            }
        } catch (UnknownHostException e) {System.err.println("Unknown host");
        } catch (IOException e) {System.err.println("Error: "+ e.getMessage());
        } catch (ClassNotFoundException e) {System.err.println("Class not found: "+
e.getMessage());}}}
```

```
Waiting for clients...
Client arrived.
Waiting for clients...
Handling client
```

```
Connecting to server...
```

DEADLOCK

```
public class StudentClient {
    public static void main(String[] args) {
        System.out.println("Connecting to server...");
        try(Socket socket= new Socket(InetAddress.getLocalHost(),1500);
            ObjectOutputStream out= new ObjectOutputStream(socket.getOutputStream());
            ObjectInputStream in= new ObjectInputStream(socket.getInputStream());
            BufferedReader localIn= new BufferedReader(
                                new InputStreamReader(System.in));)
        {
            System.out.println("Connected.");
            String option=null;
            ArrayList<Student> result=null;
            System.out.println("Insert last name to query server, 'exit' to quit.");
            while( !(option=localIn.readLine()).equals("exit")){
                System.out.println("Asking for students with last name "+option);
                out.writeObject(option);
                out.flush();
                result=(ArrayList<Student>) in.readObject();
                System.out.println("Received "+result.size()+" students with last name "+
option);
                for (Student student: result){
                    System.out.println(student);
                }
                System.out.println("Insert last name to query server, 'exit' to quit.");
            }
        } catch (UnknownHostException e) {System.err.println("Unknown host");
        } catch (IOException e) {System.err.println("Error: "+ e.getMessage());
        } catch (ClassNotFoundException e) {System.err.println("Class not found: "+
e.getMessage());}}}
```

Swap ->

```
Waiting for clients...
Client arrived.
Waiting for clients...
Handling client
Ready for a new name...
Received request for sirbu
Ready for a new name...
Received request for brown
Ready for a new name...
Received request for Moore
Ready for a new name...
Error: null
```

```
Connecting to server...
Connected.
Insert last name to query server, 'exit' to quit.
sirbu
Asking for students with last name sirbu
Received 0 students with last name sirbu
Insert last name to query server, 'exit' to quit.
brown
Asking for students with last name brown
Received 2 students with last name brown
Robert Brown living at 12 Dawson street.
Ann Brown living at 132 Buffallo street.
Insert last name to query server, 'exit' to quit.
Moore
Asking for students with last name Moore
Received 1 students with last name Moore
Joanna Moore living at 62 Collins street.
Insert last name to query server, 'exit' to quit.
exit
```

Esercizi

- MiniFTP
  - Scrivere un servizio di trasferimento di file:
  - Il cliente invia al server il nome di un file (di testo), preso dalla linea di comando.
  - Il server risponde spedendo al cliente il contenuto del file, riga per riga.
  - Il cliente salva il contenuto in un file locale con lo stesso nome.
  - Gestire situazioni di errore (file not found, etc)

Esercizi

- Trovaprezzi: sviluppare un programma che implementa un servizio trova prezzi e i suoi clienti.
    - 3 negozi web
        - vendono telefonini e smartphone. Ogni negozio ha una lista di prodotti. Ogni prodotto è descritto da: nome del produttore, modello, prezzo. Ogni negozio mette a disposizione un server che offre la lista completa dei prodotti, su richiesta.
    - Un server trova prezzi
        - mantiene una lista di prodotti ed il negozio in cui sono venduti - scaricati da ogni negozio e aggiornati ogni 24 ore.
        - offre un API per i clienti che cercano un prodotto (per il nome).
    - Dei clienti - interrogano il server trova prezzi
    - Simulare l'intervallo di aggiornamento della lista con un valore introdotto in input.
    - Attenzione: l’accesso alla lista di prodotti del server trova prezzi deve essere thread safe. Usare serializzazione per scambiare i prodotti.

Ultimo esempio

- Implementiamo insieme una chat room anonima usando socket TCP