Università degli Studi dell'Insubria
Dipartimento di Scienze Teoriche e Applicate

Programmazione Concorrente e Distribuita
Socket

Luigi Lavazza
Dipartimento di Scienze Teoriche e Applicate
luigi.lavazza@uninsubria.it

Applicazioni distribuite

- Applicazione: un insieme di programmi coordinati per svolgere una data funzione.
- Un'applicazione è distribuita se prevede più programmi eseguiti (o processi) su differenti calcolatori connessi tramite una rete.
  - Es: Web Browser (Firefox, IE, Chrome, Safari, Opera …) e Web Server (server http)

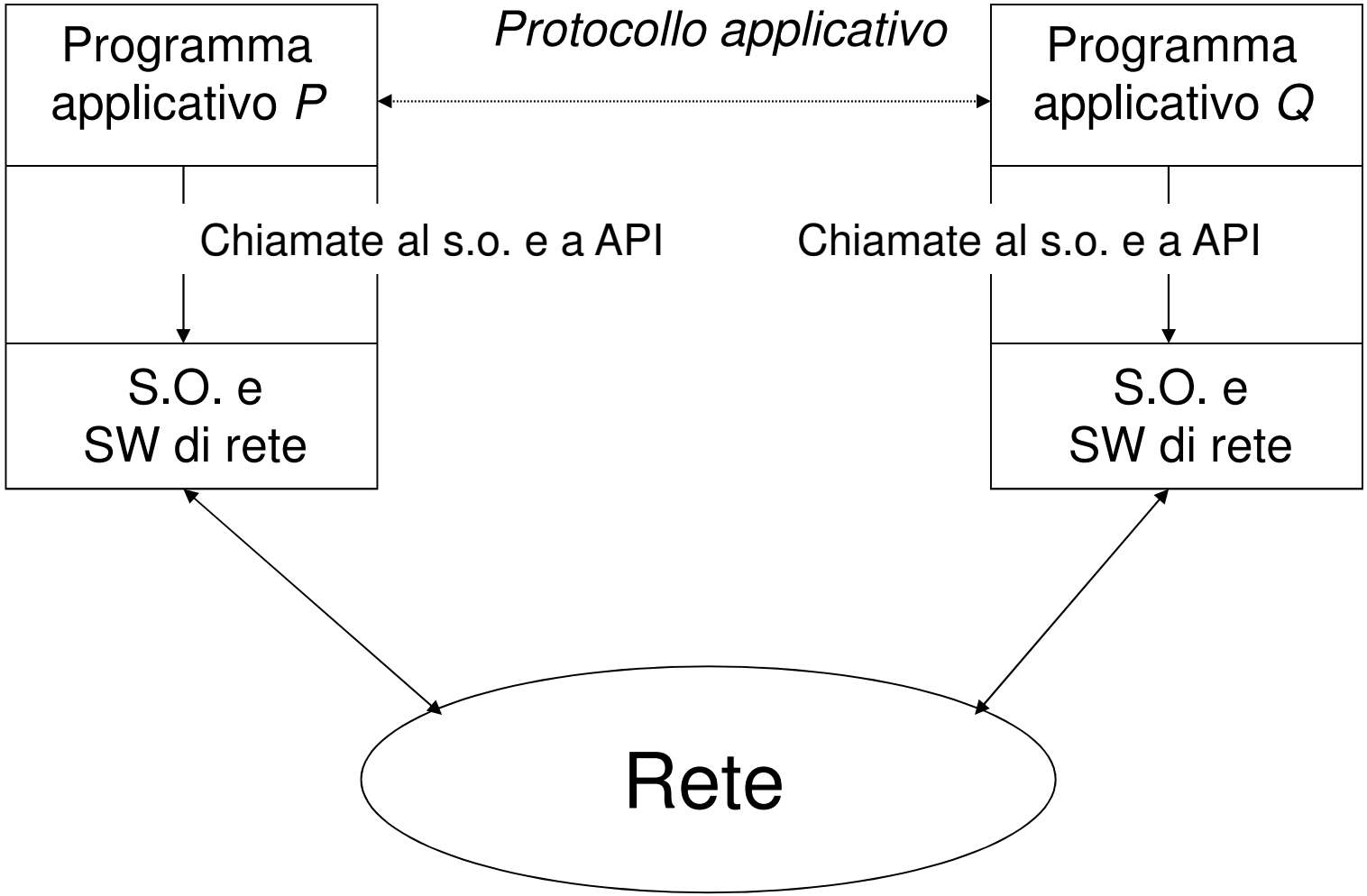
Protocollo applicativo

- Le regole per la comunicazione in una applicazione distribuita sono dette protocollo applicativo.
  - Es. il protocollo applicativo della navigazione Web è detto HyperText Transfer Protocol - http.
- Il protocollo applicativo deve essere definito opportunamente e comune a tutti i programmi dell'applicazione.
  - Es. ogni messaggio scambiato è terminato dalla stringa "\0 \0 \0".

Interfacce e protocolli

- I programmi applicativi utilizzano opportune interfacce (API - Application Programming Interface), fornite dal sistema operativo e dal software di rete, per accedere ai servizi di comunicazione
  - Nascondono i dettagli dei livelli inferiori.
- Il protocollo applicativo rappresenta le regole di comunicazione, e considera il contenuto della comunicazione.
  - Realizzabile usando, attraversi API, i servizi disponibili

Interfacce e protocolli

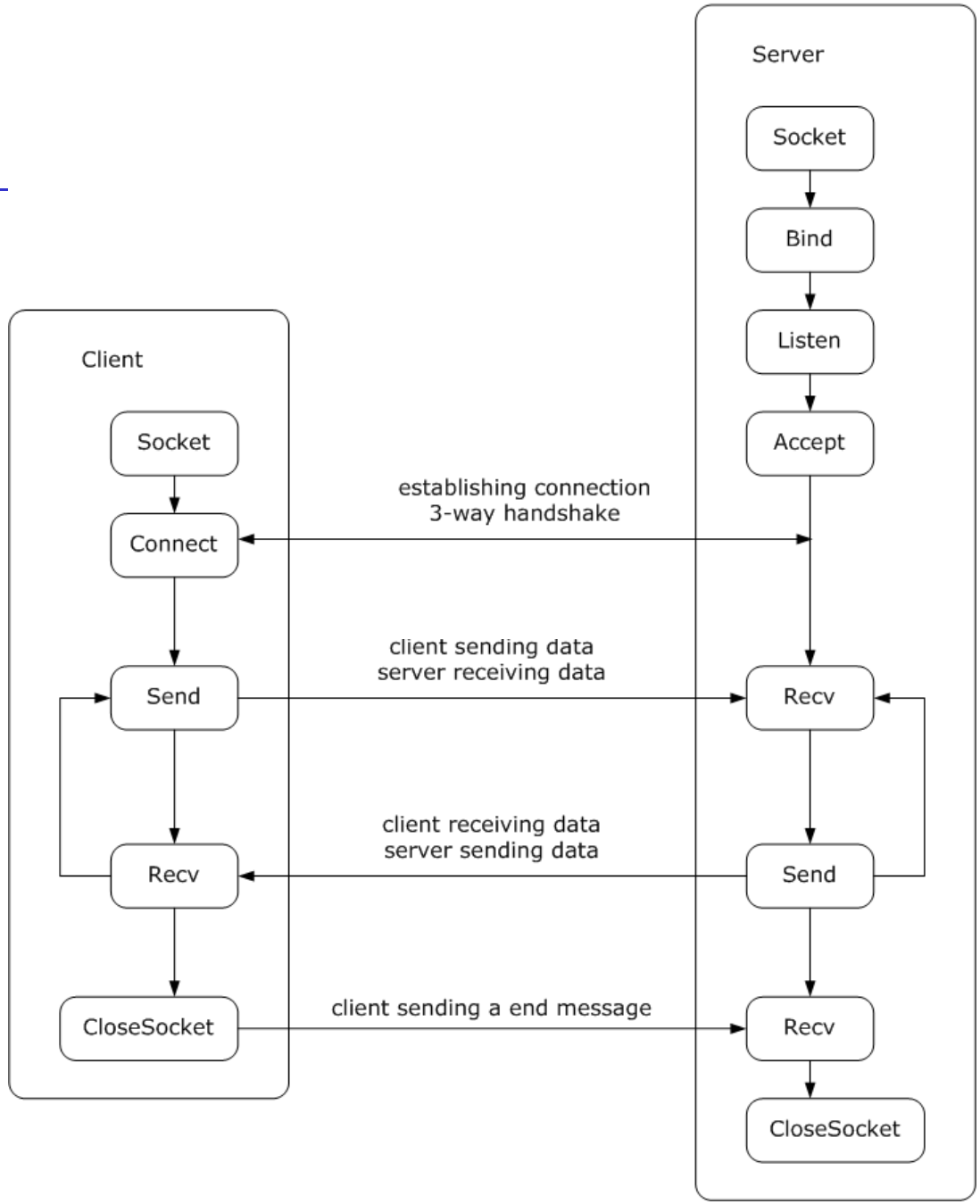


Programmare la comunicazione via API

- Fortunatamente esiste uno standard de facto per usare TCP/IP (e UDP/IP) : l'interfaccia socket.
  - Un'API (Application Programming Interface)
  - Disponibile per i linguaggi C, Java e molti altri

Socket

- I Socket di Berkeley hanno origine dal sistema operativo Unix BSD 4.2 (rilasciato nel 1983) come una API.
- Tutti i moderni sistemi operativi ne hanno ora almeno un'implementazione: l'interfaccia socket di Berkeley è di fatto diventata l'interfaccia standard per la connessione a Internet.
- TCP sockets flow diagram ⇒

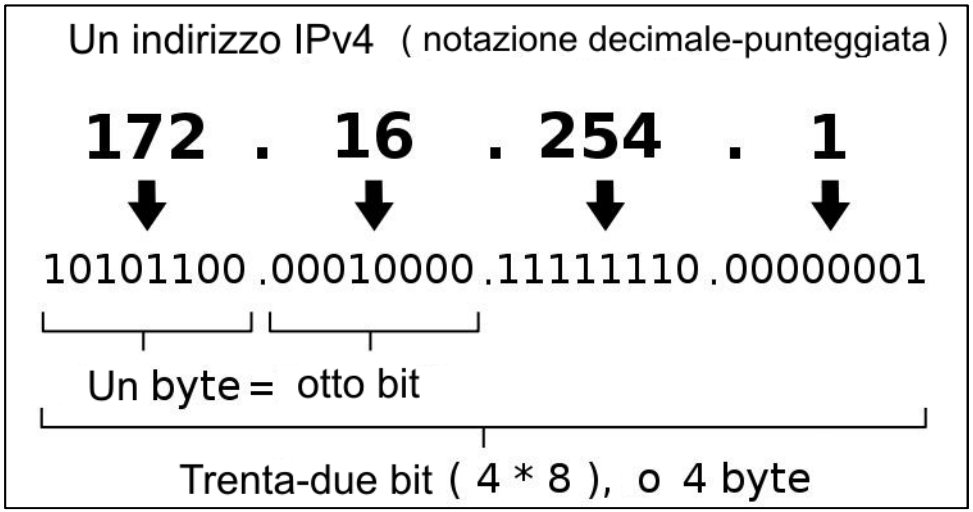


API socket

- L'interfaccia (API) socket è disponibile su Unix, Windows e altre piattaforme
- La disponibilità universale dell'interfaccia socket rende possibile la portabilità dei programmi di rete
- Due applicazioni basate su uno stesso protocollo di Trasporto possono interagire
  - ad esempio un'applicazione Java che utilizza TCP tramite l'API socket-Java può interagire senza problemi con una applicazione C che utilizza TCP tramite l'API socket C.
- Due applicazioni basate su diversi protocolli di Trasporto non possono interagire: ad esempio un'applicazione che utilizza UDP non può interagire con una applicazione che utilizza TCP.
- In effetti è insito nella nozione di protocollo che un insieme di programmi usino lo stesso protocollo.

Come identificare una macchina

- Serve un metodo per identificare in modo univoco una macchina e distinguerla da tutte le altre nel mondo
- Ciò si ottiene con l'indirizzo IP (Internet Protocol address), che può esistere in due forme:
  - Indirizzo numerico a 32 bit separati da punti, ad esempio 172.16.254.1



  - Nome simbolico: un nome testuale facile per identificare un server, ad esempio dista.uninsubria.it
    - Attribuito da DNS (Domain Name System)

Come identificare una macchina

- L'indirizzo IP si può ottenere con **`InetAddress.getByName()`**
  - utilizzabile per costruire un socket

Come identificare l'indirizzo IP dal nome di dominio in Java

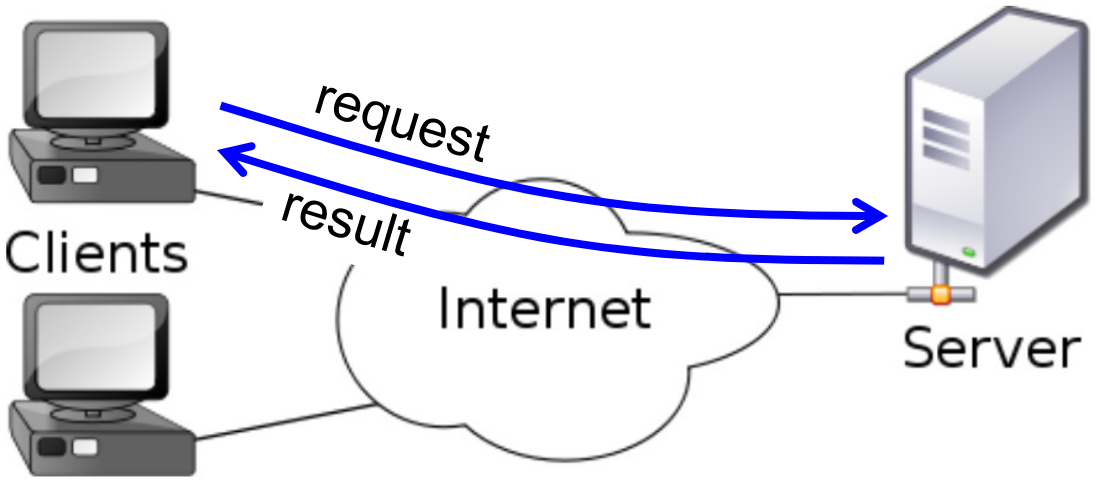
- L'indirizzo IP si può ottenere con **`InetAddress.getByName()`**
  - utilizzabile poi per costruire un socket

```
import java.net.*;
public class IPaddressSolver{
  public static void main(String[] args) throws Exception {
    if (args.length!=1) {
      System.err.println("Usage: WhoAmI MachineName ");
      System.exit(1);
    }
    InetAddress a=InetAddress.getByName(args[0]);
    System.out.println(a);
  }
}
```

Scrivendo: **`java IPaddressSolver google.com`**
ottengo: google.com/216.58.208.174
scrivendo: **`java IPaddressSolver www.cefriel.com`**
ottengo: www.cefriel.com/151.101.1.195

Client e Server in Java

- Il server deve rimanere in ascolto di richieste di connessione, e questo viene fatto dall'oggetto speciale **`ServerSocket`**.
- Il client cerca di stabilire una connessione con un server usando un oggetto di tipo **`Socket`**.
- Effettuata la connessione, questa viene trasformata in un I/O stream, e da allora in poi si può trattare la connessione come se si stesse leggendo da e scrivendo su un file.
- Le regole con cui scrivere/leggere sono dettate dai protocolli di livello più alto.



127.0.0.1

- 127.0.0.1 is the loopback Internet protocol (IP) address also referred to as the localhost.
- The address is used to establish an IP connection to the same machine or computer being used by the end-user.

- Utile per fare prove, quando non si hanno diverse macchine in rete.
- Infatti i processi che comunicano possono stare sulla stessa macchina

- In effetti i socket sono comunemenmte usati anche per fare inter-process communication in locale.

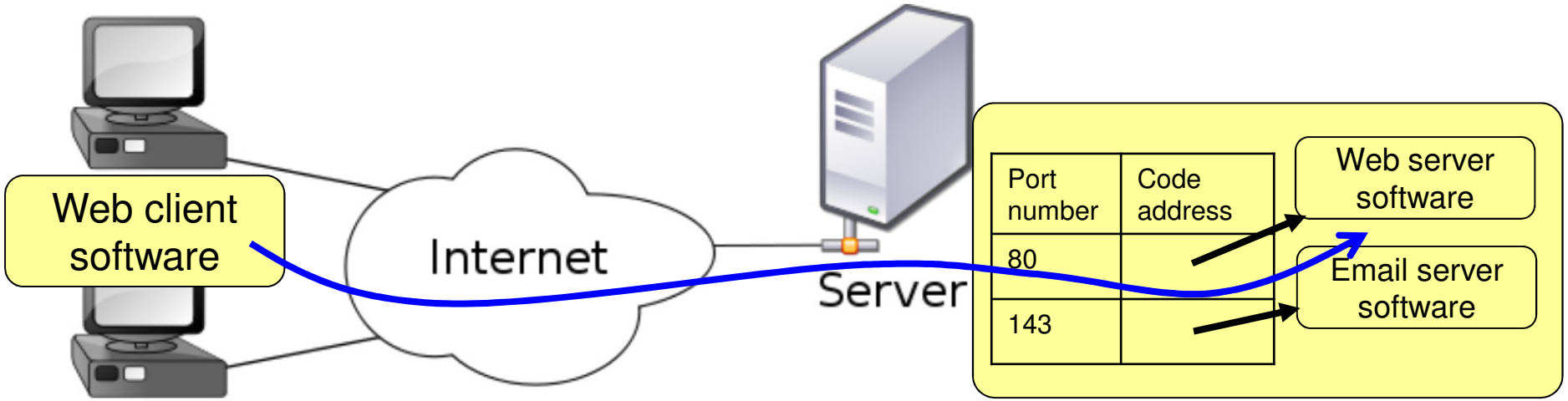
Client e Server: test senza rete

- Per molte ragioni, si potrebbe non avere una macchina client, una macchina server e una rete a disposizione per testare i programmi
- In questo caso si può usare l'indirizzo speciale localhost che corrisponde all'indirizzo IP "127.0.0.1" pensato per test senza la rete.
- Queste tre istruzioni producono tutte lo stesso risultato.

```
InetAddress addr=InetAddress.getByName(null);
InetAddress addr=InetAddress.getByName("localhost");
InetAddress addr=InetAddress.getByName("127.0.0.1");
```

Client e Server: l'indirizzo IP non basta!

- Un indirizzo IP identifica una macchina, ma non è sufficiente per identificare un processo server in modo univoco, visto che su una sola macchina possono essere attivi molti processi server.
- Ogni macchina IP contiene anche le porte, e quando si sta impostando un client o un server è necessario scegliere una porta attraverso la quale il client e il server si connettono.
- Attenzione: le porte da 1 a 1024 solitamente sono riservate dal sistema



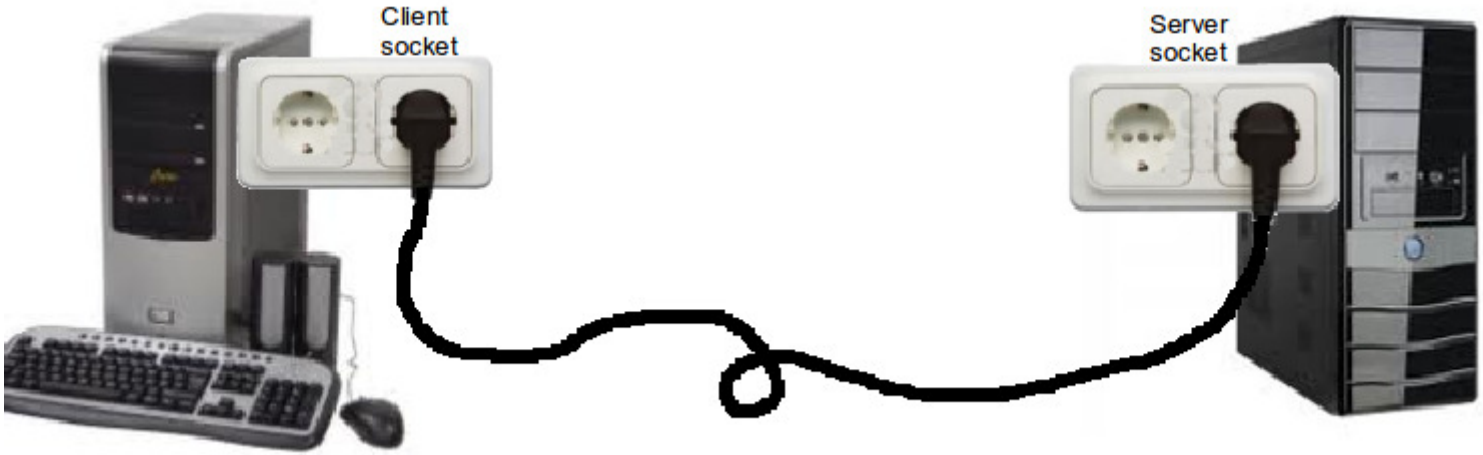
Indirizzamento in Java

- java.net fornisce le seguenti classi di indirizzamento correlate:

- InetAddress: indirizzo IP
  - Inet4Address: indirizzo a 32-bit IPv4 (circa 4.3×10^9)
  - Inet6Address: indirizzo a 128-bit IPv6 (circa 3.4×10^{38})
- SocketAddress: indirizzo Socket
  - InetSocketAddress: IP Socket Address (indirizzo IP + porta)

Socket

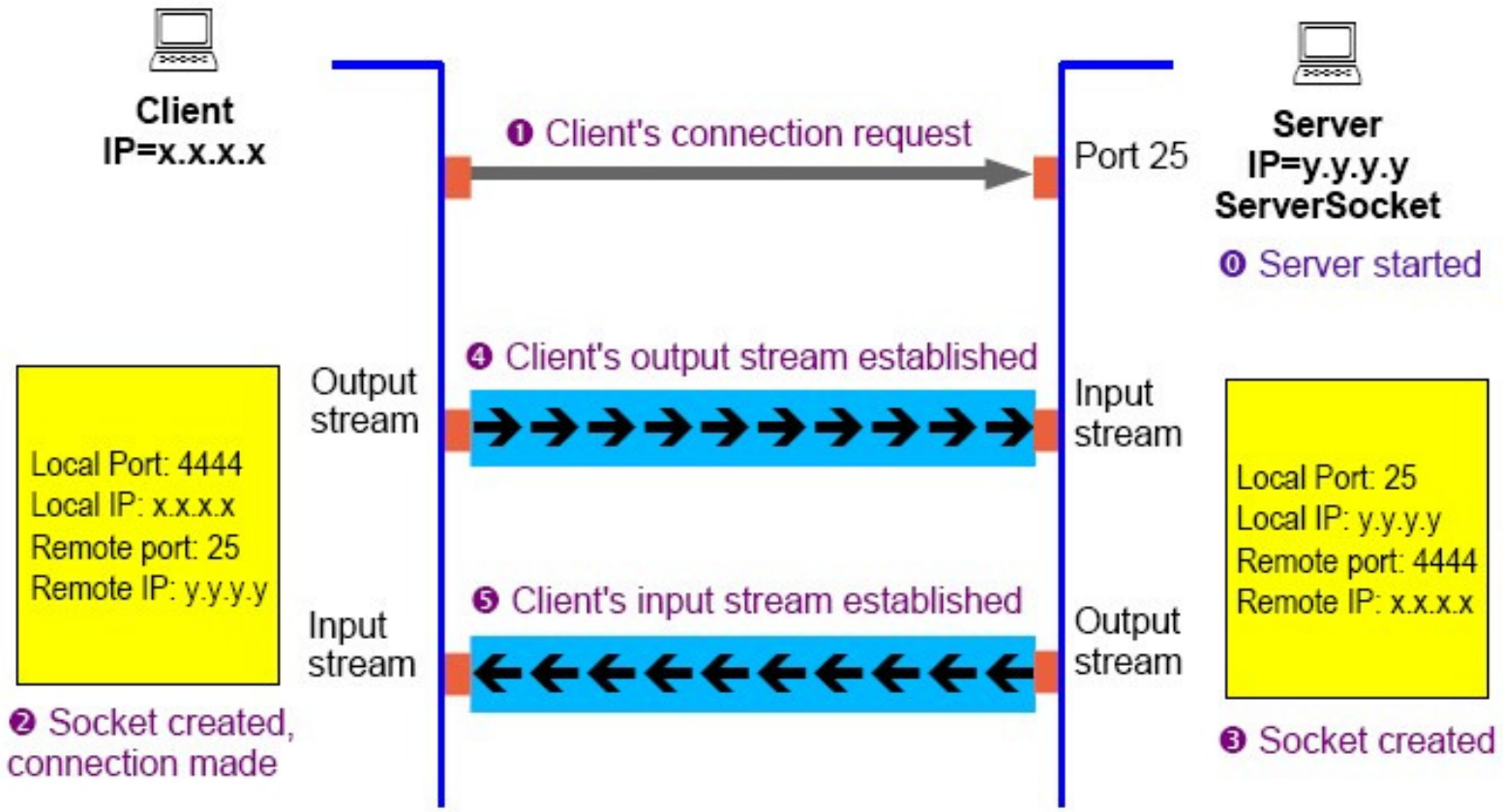
- Un socket rappresenta i “terminali” di un collegamento tra due macchine.
- Per una determinata connessione, c'è un socket su ogni macchina
- Si può immaginare un ipotetico “cavo” (InputStream/OutputStream) tra le due macchine con ciascuna estremità collegata ad un “socket” che persiste finché non viene esplicitamente disconnesso.



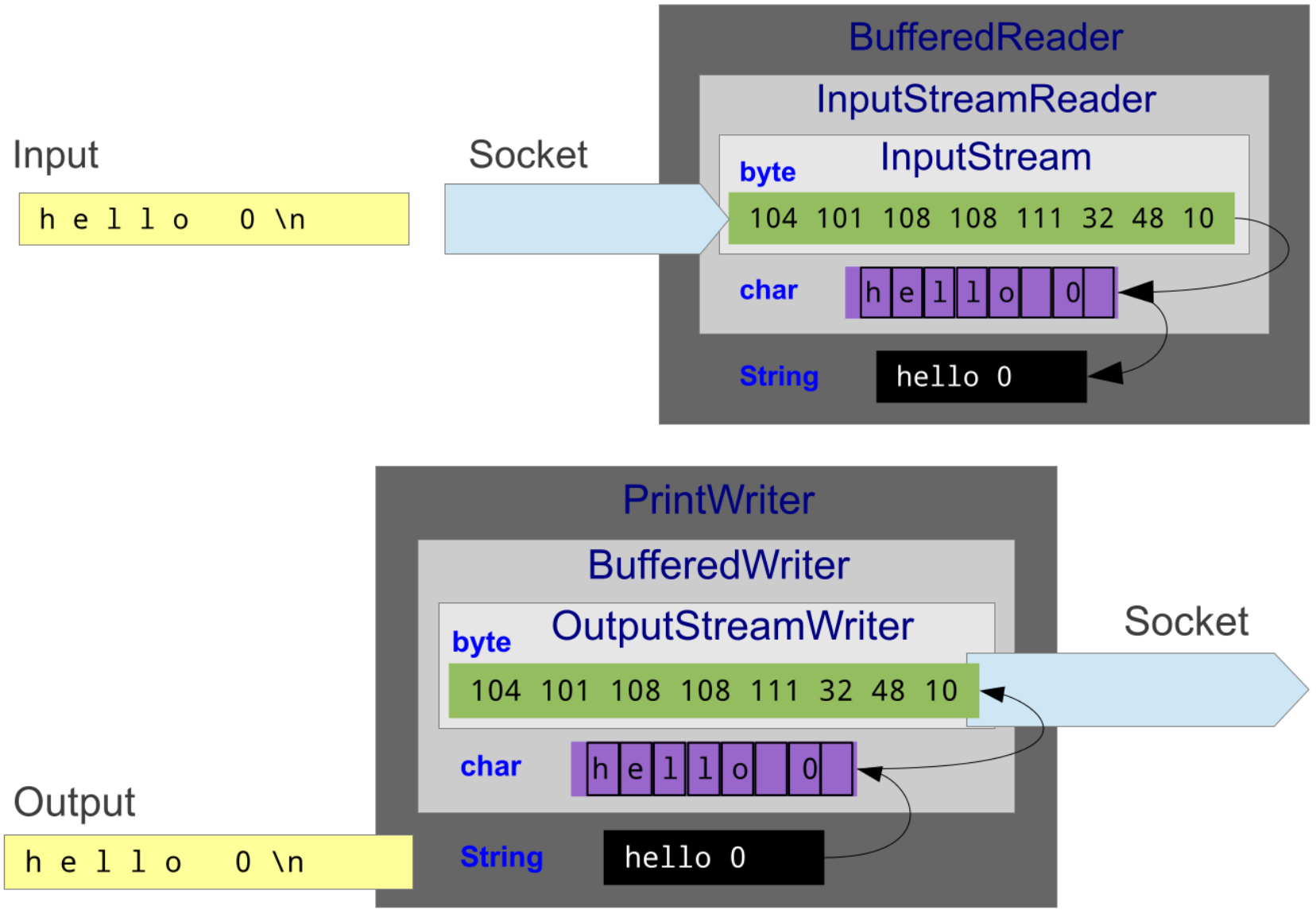
Socket

- In Java esistono due classi di socket stream-based:
  - **`ServerSocket`**, che un server utilizza per ”ascoltare” le connessioni in ingresso
  - **`Socket`**, che un client utilizza, al fine di avviare una connessione.
- **`ServerSocket`** usa il metodo **`accept()`** per ritornare un Socket nel momento in cui il client si connette
- Dopo di che si ha una connessione Socket-to-Socket
- A questo punto, è possibile utilizzare i metodi **`getInputStream()`** e **`getOutputStream()`** su ogni Socket.
- Quando si crea un **`ServerSocket`**, si dà solo il numero di porta su cui si accetteranno le connessioni.
- Quando si crea un **`Socket`** è necessario dare sia l’indirizzo IP che il numero di porta in cui si sta tentando di connettersi.

Stream Socket overview



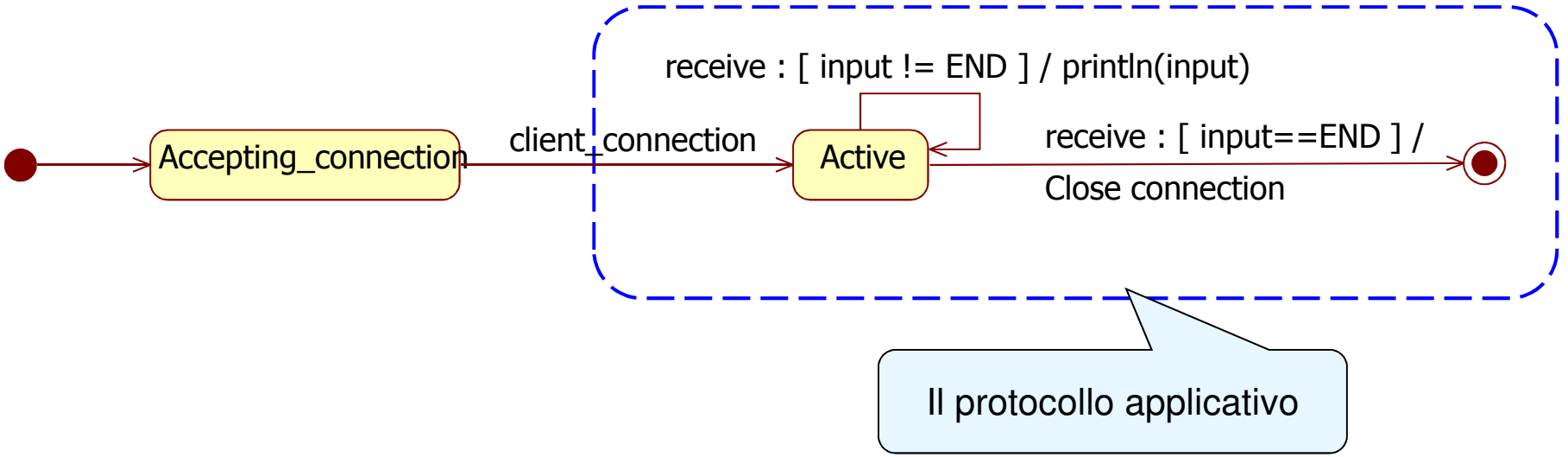
InputStream e OutputStream



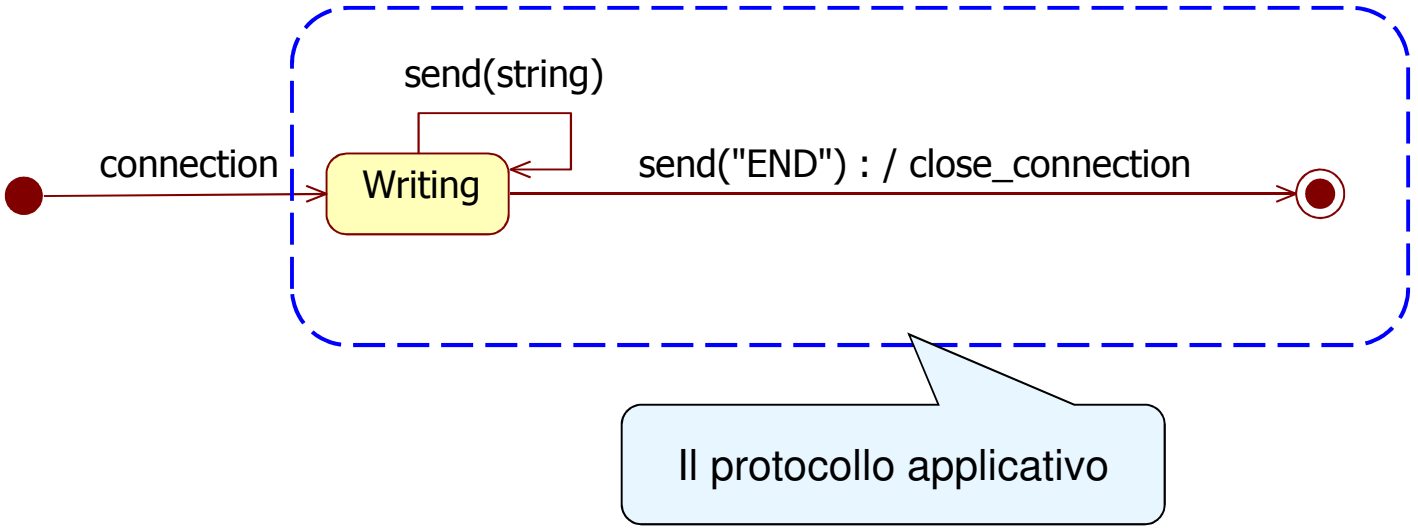
Esempio

- Programmiamo una semplice applicazione distribuita
- Il client
  1. Si connette al server
  2. Manda al server una sequenza di stringhe, terminate dalla stringa “END”
  3. Si sconnette
- Il server
  1. Accetta connessioni
  2. Iterativamente:
  3. Riceve una stringa
     - Se la stringa ricevuta non e` “END” la visualizza
     - Se la stringa ricevuta e` “END” esce dal ciclo
  4. Chiude la connessione

Server: stati e protocollo applicativo



Client: stati e protocollo applicativo



Un semplice Server che ritorna qualunque cosa arrivi dal client

```
public class JabberServer {
  public static final int PORT=8080;
  public static void main(String[] args) throws IOException {
    ServerSocket s = new ServerSocket(PORT);
    System.out.println("Started: "+s);
    try {
      Socket mySocket=s.accept();
      try {
        System.out.println("Connection accepted: "+mySocket);
        BufferedReader in=
          new BufferedReader(new InputStreamReader(
                                  mySocket.getInputStream()));
        PrintWriter out=new PrintWriter(new BufferedWriter(
          new OutputStreamWriter(mySocket.getOutputStream())),
                                        true);
        while(true) {
          String str=in.readLine();
          if(str.equals("END")) break;
          System.out.println("Echoing:"+str);
          out.println(str);
        }
      }
```

Un semplice Server che ritorna qualunque cosa arrivi dal client

```
        finally {
          System.out.println("closing...");
          mySocket.close();
        }
    } finally {
      s.close();
    }
}
```

Client semplice che invia testo al server e legge le sue risposte

```
public class JabberClient {
  public static void main(String[] args) throws IOException {
    InetAddress addr = InetAddress.getByName(null);
    System.out.println("addr = " + addr);
    Socket socket = new Socket(addr, 8080);
    try {
      System.out.println("socket = " + socket);
      BufferedReader in = new BufferedReader(
              new InputStreamReader(socket.getInputStream()));
      PrintWriter out = new PrintWriter(new BufferedWriter(
        new OutputStreamWriter(socket.getOutputStream())), true);
      for (int i = 0; i < 10; i++) {
        out.println("hello " + i);
        String str = in.readLine();
        System.out.println(str);
      }
      out.println("END");
    } finally {
      System.out.println("closing...");
      socket.close();
}}}
```

La connessione potrebbe non andare a buon fine

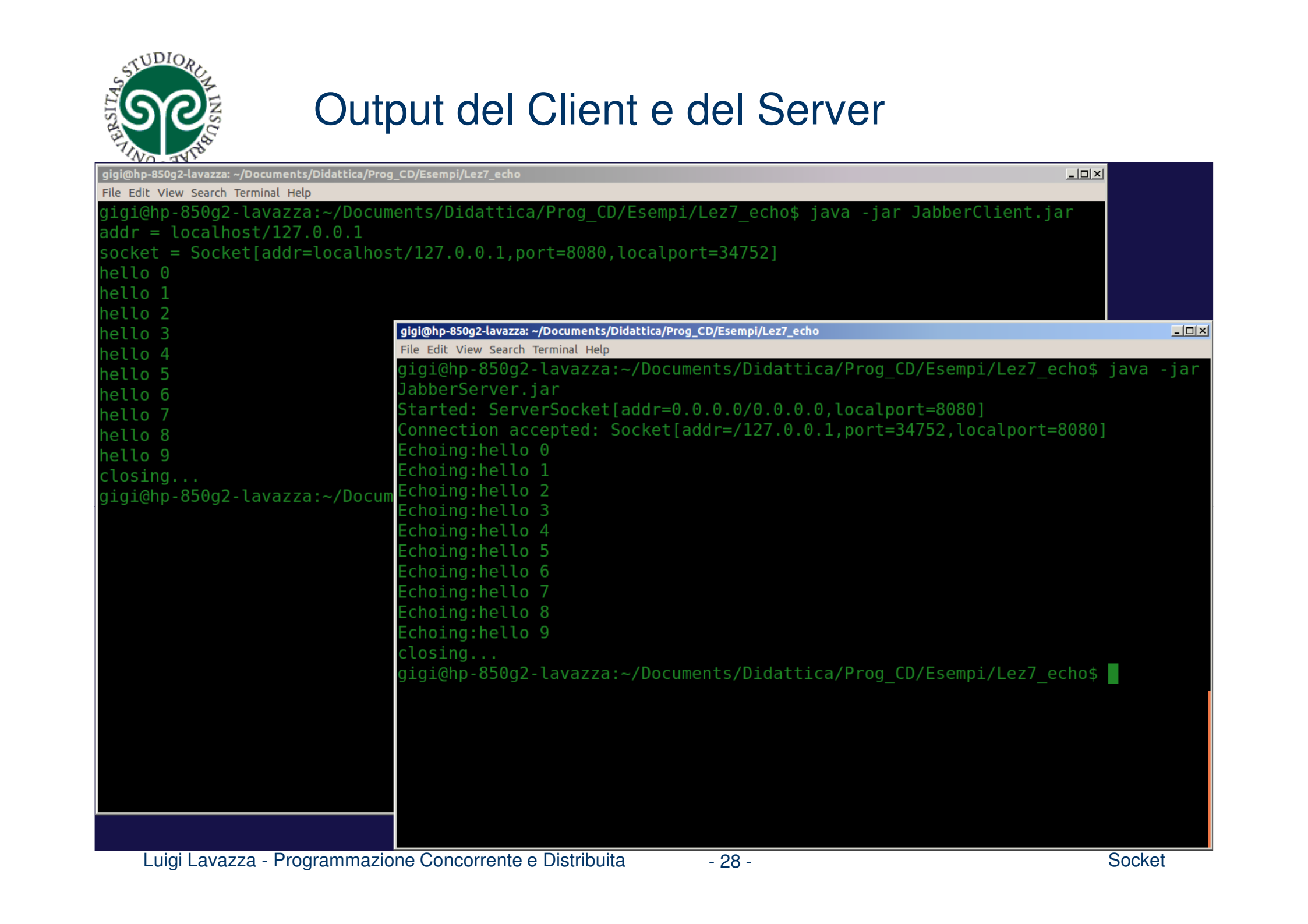
Output del Client e del Server

Client

```
addr = localhost/127.0.0.1
socket = Socket[
  addr=localhost/127.0.0.1,
  port=8080,
  localport=48904]
hello 0
hello 1
hello 2
hello 3
hello 4
hello 5
hello 6
hello 7
hello 8
hello 9
closing...
```

Server

```
Started: ServerSocket[
  addr=0.0.0.0/0.0.0.0,
  port=0,
  localport=8080]
Connection accepted: Socket[
  addr=/127.0.0.1,
  port=48904,
  localport=8080]
Echoing: hello 0
Echoing: hello 1
Echoing: hello 2
Echoing: hello 3
Echoing: hello 4
Echoing: hello 5
Echoing: hello 6
Echoing: hello 7
Echoing: hello 8
Echoing: hello 9
closing...
```



Output del Client e del Server

gigi@hp-850g2-lavazza: ~/Documents/Didattica/Prog_CD/Esempi/Lez7_echo

File Edit View Search Terminal Help

```
gigi@hp-850g2-lavazza:~/Documents/Didattica/Prog_CD/Esempi/Lez7_echo$ java -jar JabberClient.jar
addr = localhost/127.0.0.1
socket = Socket[addr=localhost/127.0.0.1,port=8080,localport=34752]
hello 0
hello 1
hello 2
hello 3
hello 4
hello 5
hello 6
hello 7
hello 8
hello 9
closing...
gigi@hp-850g2-lavazza:~/Docum
```

gigi@hp-850g2-lavazza: ~/Documents/Didattica/Prog_CD/Esempi/Lez7_echo

File Edit View Search Terminal Help

```
gigi@hp-850g2-lavazza:~/Documents/Didattica/Prog_CD/Esempi/Lez7_echo$ java -jar
JabberServer.jar
Started: ServerSocket[addr=0.0.0.0/0.0.0.0,localport=8080]
Connection accepted: Socket[addr=/127.0.0.1,port=34752,localport=8080]
Echoing:hello 0
Echoing:hello 1
Echoing:hello 2
Echoing:hello 3
Echoing:hello 4
Echoing:hello 5
Echoing:hello 6
Echoing:hello 7
Echoing:hello 8
Echoing:hello 9
closing...
gigi@hp-850g2-lavazza:~/Documents/Didattica/Prog_CD/Esempi/Lez7_echo$
```

Daytime Server

- Un esempio di Server Socket che in un loop infinito accetta delle connessioni Socket per inviare la data corrente al Client.

Daytime Server

```
public class DaytimeServer {
  public final static int DayTime_PORT= 1333;
  public static void main(String[] args) {
    try {
      ServerSocket server = new ServerSocket(DayTime_PORT);
      Socket conn = null;
      while (true) {
        try {
          conn = server.accept();
          Writer out = new OutputStreamWriter(conn.getOutputStream());
          Date now = new Date();
          out.write(now.toString() +"\r\n"); out.flush();
          conn.close();
        } catch (IOException ex) {}
        finally {
          try {
            if (conn != null) conn.close();
          } catch (IOException ex) {}
        }
      }
    } catch (IOException ex) {}
}}
```

Fallisce se il client ha già chiuso la connessione

Daytime client

```
import java.net.*;
import java.io.*;
public class DayTimeClient {
  public static void main(String[] args) throws IOException {
    String str;
    InetAddress addr = InetAddress.getByName(null);
    System.out.println("addr = " + addr);
    Socket connection = new Socket(addr, 1333);
    try {
      System.out.println("socket = " + connection);
      BufferedReader in = new BufferedReader(new
           InputStreamReader(connection.getInputStream()));
      str = in.readLine();
      System.out.println(str);
    }
    finally {
      System.out.println("closing...");
      connection.close();
}}}
```

DayTime: client output

```
addr = localhost/127.0.0.1
socket = Socket[addr=localhost/127.0.0.1,port=1333,localport=52696]
Wed Apr 14 12:27:59 CEST 2021
closing...
```

Transmission Control Protocol (TCP)

- Gli esempi visti fino ad ora utilizzano TCP, noto anche come stream-based sockets, progettato per la massima affidabilità e garantisce che i dati arrivino a destinazione.
- Le caratteristiche di TCP sono
  - consentire la ritrasmissione dei dati persi
  - fornire percorsi multipli attraverso diversi router nel caso in cui uno di questi diventi indisponibile
  - consegna dei byte nell'ordine in cui sono stati inviati.
- Per garantire tutto questo controllo e affidabilità TCP ha un grande overhead come costo da pagare.
- Si può usare un secondo protocollo, chiamato User Datagram Protocol (UDP), che non garantisce che i pacchetti saranno consegnati e nemmeno che arriveranno nell'ordine in cui sono stati inviati.

TCP vs. UDP

- Clients and servers that communicate via a TCP socket, have a dedicated point-to-point channel.
- To communicate, the parties
  - establish a connection,
  - transmit the data, and then
  - close the connection.
- All data sent over the channel is received in the same order in which it was sent. This is guaranteed by the channel.

- In contrast, applications that communicate via datagrams send and receive completely independent packets of information.
- These clients and servers do not have and do not need a dedicated point-to-point channel.
- The delivery of datagrams to their destinations is not guaranteed.
- Nor is the order of their arrival.

User Datagram Protocol (UDP)

- Datagram definition

  A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed.
- The java.net package contains three classes to help you write Java programs that use datagrams to send and receive packets over the network:
- **`DatagramSocket`**, **`DatagramPacket`**: An application can send and receive DatagramPackets through a DatagramSocket.
- **`MulticastSocket`**: DatagramPackets can be broadcast to multiple recipients all listening to a MulticastSocket.

Esempio

- Il server
  - riceve un pacchetto di dati,
  - ne estrae una stringa e la mostra a video
  - la rimanda al client, dopo averla convertita in lettere maiuscole
- Il client:
  - legge una stringa da terminale
  - confeziona un pacchetto contenente la stringa e lo spedisce al server
  - riceve un pacchetto dal server, estrae la stringa contenuta, mostra e termina

Esempio dimostrativo di UDP

```
class UDPServer {
  public static void main(String args[]) throws Exception {
    DatagramSocket serverSocket = new DatagramSocket(9876);
    byte[] receiveData = new byte[1024];
    byte[] sendData = new byte[1024];
    DatagramPacket receivePacket = new
          DatagramPacket(receiveData, receiveData.length);
    while (true) {
      serverSocket.receive(receivePacket);
      String sentence = new String(receivePacket.getData());
      System.out.println("RECEIVED: " + sentence);
      InetAddress IPAddress = receivePacket.getAddress();
      int port = receivePacket.getPort();
      String capitalizedSentence = sentence.toUpperCase();
      sendData = capitalizedSentence.getBytes();
      DatagramPacket sendPacket = new
        DatagramPacket(sendData,sendData.length,IPAddress,port);
      serverSocket.send(sendPacket);
    }
}}
```

Esempio dimostrativo di UDP

```
class UDPClient {
  public static void main(String args[]) throws Exception {
    BufferedReader inFromUser = new BufferedReader(
                          new InputStreamReader(System.in));
    DatagramSocket clientSocket = new DatagramSocket();
    InetAddress IPAddress = InetAddress.getByName("localhost");
    byte[] sendData = new byte[1024];
    byte[] receiveData = new byte[1024];
    System.out.println("type a string: ");
    String sentence = inFromUser.readLine();
    sendData = sentence.getBytes();
    DatagramPacket sendPacket = new DatagramPacket(sendData,
                              sendData.length, IPAddress, 9876);
    clientSocket.send(sendPacket);
    DatagramPacket receivePacket =
            new DatagramPacket(receiveData, receiveData.length);
    clientSocket.receive(receivePacket);
    String modifiedSentence=new String(receivePacket.getData());
    System.out.println("FROM SERVER:" + modifiedSentence);
    clientSocket.close();
  }
}
```

Esecuzione (lato client)

type a string:

bubu

FROM SERVER:BUBU

UDP Image Server

- Un esempio di Server Socket che usa il protocollo UDP per inviare un'immagine al Client. Ogni immagine viene scomposta in tanti DatagramPacket da 1024 byte.
- Questo esempio si può estendere all'invio dei frame di un video. . .

UDP Image Server

```
class UDPImageServer {
  public static void main(String args[]) throws Exception {
    DatagramSocket serverSocket = new DatagramSocket(9876);
    while (true) {
      // RECEIVE
      byte[] receiveData = new byte[1024];
      DatagramPacket receivePacket =
            new DatagramPacket(receiveData, receiveData.length);
      System.out.println("waiting ");
      serverSocket.receive(receivePacket);
      String sentence = new String(receivePacket.getData());
      System.out.println("RECEIVED: " + sentence);
      // SEND...
      }
   }
}
```

UDP Image Server

```
class UDPImageServer {
  public static void main(String args[]) throws Exception {
  DatagramSocket serverSocket = new DatagramSocket(9876);
  while (true) {
    // RECEIVE ...
    // SEND
    InetAddress IPAddress = receivePacket.getAddress();
    int port = receivePacket.getPort();
    File file = new File("race-condition.png");
    System.out.println("Starting to send file: " + file);
    FileInputStream fis = new FileInputStream(file);
    int size = 0;
    while (true) {
      ...
    }
  }
}}
```

UDP Image Server

```
while (true) {
  byte[] sendData = new byte[1024];
  size = fis.read(sendData);
  if(size == -1) {
    byte[] sendEOFData = "END_FILE".getBytes();
    System.out.println("Send: " + new String(sendEOFData));
    DatagramPacket sendPacket = new DatagramPacket(
         sendEOFData, sendEOFData.length, IPAddress, port);
    serverSocket.send(sendPacket);
    break;
  } else {
    DatagramPacket sendPacket = new DatagramPacket(
         sendData, size, IPAddress, port);
    serverSocket.send(sendPacket);
  }
  Thread.sleep(1);
}
```

UDP Image Client

```
class UDPImageClient {
  public static void main(String args[]) throws Exception {
    DatagramSocket clientSocket = new DatagramSocket();
    InetAddress IPAddress = InetAddress.getByName("localhost");
    byte[] sendData;
    sendData = "START".getBytes();
    DatagramPacket sendPacket = new DatagramPacket(
       sendData, sendData.length, IPAddress, 9876);
    clientSocket.send(sendPacket);
    File file = new File("./race-condition-client.png");
    FileOutputStream fos = new FileOutputStream(file);
```

UDP Image Client

```
    while (true) {
      byte[] receiveData = new byte[1024];
      DatagramPacket receivePacket =
         new DatagramPacket(receiveData, receiveData.length);
      clientSocket.receive(receivePacket);
      String modifiedSentence =
           new String(receivePacket.getData());
      if (modifiedSentence.startsWith("END_FILE")) {
        System.out.println("RECEIVED FROM SERVER: " +
                           modifiedSentence);
        fos.close();
        break;
      }
      fos.write(receivePacket.getData(), 0,
                receivePacket.getLength());
   }
   clientSocket.close();
   System.out.println("CLIENT: finished");
}}
```

Server che gestiscono più di un client

- Gli esempi di Server visti finora sono in grado di gestire un solo client alla volta.
- Un tipico server deve poter gestire diversi client in parallelo.
- Possiamo ottenere questo risultato attraverso il multithreading
  1. si crea un unico **`ServerSocket`** nel server
  2. si attende una nuova connessione attraverso **`accept()`**
  3. quando **`accept()`** ritorna un **`Socket`** si crea un nuovo thread il cui compito è "servire" unicamente quel particolare client.
     - A tale scopo si passa al nuovo thread il socket connesso al client.
     - Quando il tread ha finito di servire il suo client termina
  4. Appena creato il thread ci si mette di nuovo in attesa di un nuovo client con **`accept()`** (tornando al punto 2)

Server che accetta più connessioni contemporanee

```
public class MultiJabberServer {
  static final int PORT = 8080;
  public static void main(String[] args) throws IOException {
    ServerSocket s = new ServerSocket(PORT);
    System.out.println("Server Started");
    try {
      while (true) {
        Socket socket = s.accept();
        try {
          new ServeOneJabber(socket);
        } catch (IOException e) {
        // If it fails, close the socket,
        // otherwise the thread will close it:
        socket.close();
      }
    }
    } finally {
      s.close();
    }
}}
```

Server che accetta più connessioni contemporanee

```
class ServeOneJabber extends Thread {
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;
  public ServeOneJabber(Socket s) throws IOException {
    socket = s;
    in = new BufferedReader(
             new InputStreamReader(socket.getInputStream()));
    out = new PrintWriter(new BufferedWriter(new
         OutputStreamWriter(socket.getOutputStream())),true);
    start();
  }
```

auto-flush

Server che accetta più connessioni contemporanee

```
  public void run() {
    try {
      while (true) {
        String str = in.readLine();
        if (str.equals("END"))
          break;
        System.out.println("Echoing: " + str);
        out.println(str);
      }
      System.out.println("closing...");
    } catch (IOException e) {
      System.err.println("IO Exception");
    } finally {
      try {
        socket.close();
      } catch (IOException e) {
        System.err.println("Socket not closed");
      }
    }
}}
```

Client **JabberClient**

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

public class JabberClient {
  InetAddress addr;
  Socket socket;
  JabberClient(){
    try {
      addr = InetAddress.getByName(null);
    } catch (UnknownHostException e) {
      System.err.println("JabberClient: no IP address");
      System.exit(0);
    }
    System.out.println("addr = " + addr);
  }
    public static void main(String[] args) {
    new JabberClient().exec(args.length==0?"anonymous":args[0]);
  }
```

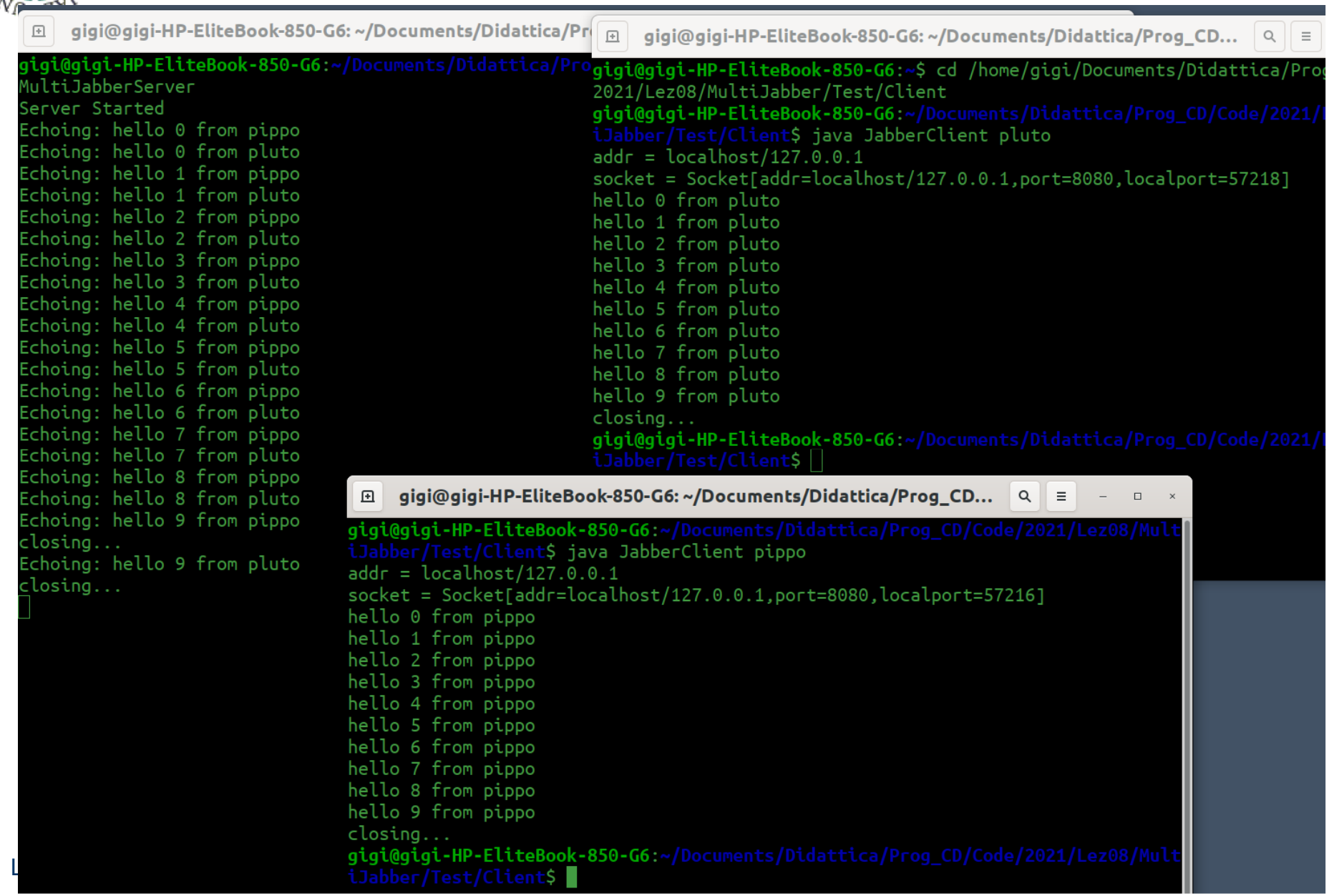
Client **JabberClient**

```
void exec(String myName) {
  try {
    socket = new Socket(addr, 8080);
    System.out.println("socket = " + socket);
    BufferedReader in = new BufferedReader(new
        InputStreamReader(socket.getInputStream()));
    PrintWriter out = new PrintWriter(new BufferedWriter(new
        OutputStreamWriter(socket.getOutputStream())), true);
    for (int i = 0; i < 10; i++) {
      Thread.sleep(1500);
      out.println("hello " + i +" from "+myName);
      String str = in.readLine();
      System.out.println(str);
    }
    out.println("END");
  } catch (IOException | InterruptedException e) {
    System.err.println("JabberClient: IO problems");
  }
  System.out.println("closing...");
  try { socket.close(); } catch (IOException e) {}
}}
```

Testing di un server multithread

- Per testare il server multi-thread occorre stabilire più connessioni contemporanee.
- Per fare questo basta creare diversi processi che eseguono tutti il programma JabberClient.
  - Si aprono tanti terminali
  - In ciascuno si esegue **`java JabberClient.class`**

Test con più clienti



Client multipli per testare il server

- La creazione manuale di diversi processi client può essere scomoda.
- Si può pensare di automatizzare il test creando un programma multi-thread in cui ogni thread si comporta come un client che si collega al stesso da testare.
- Questa modalità è un po' meno realistica della precedente, perché i diversi processi potrebbero trovarsi su macchine diverse, mentre i diversi thread eseguono necessariamente sulla medesima macchina.

Client multi-thread per testare il server

- Ogni client è un **`Thread`** che crea un **`Socket`** per connettersi con lo stesso server usando **`InetAddress`**
- Ogni thread ha una durata limitata e alla fine chiude il **`Socket`**.
- Una costante **`MAX_THREADS`** nel main verrà usata per limitare il numero massimo di Thread contemporaneamente in esecuzione.
  - Cambiando il valore di questa costante si può vedere dove il vostro particolare sistema comincia ad avere problemi con troppe connessioni.

Multi-Client per più connessioni

```
public class MultiJabberClient {
  static final int MAX_THREADS = 4;
  public static void main(String[] args) throws IOException,
InterruptedException {
    InetAddress addr = InetAddress.getByName(null);
    while (JabberClientThread.threadTotalCount()<20) {
      if (JabberClientThread.threadCount() < MAX_THREADS) {
        new JabberClientThread(addr);
        Thread.sleep(ThreadLocalRandom.current().nextInt(100));
      } else {
        Thread.sleep(ThreadLocalRandom.current().nextInt(10));
      }
    }
  }
}
```

Multi-Client per più connessioni

```
public class JabberClientThread extends Thread {
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;
  private static int counter = 0;       // client totali
  private int id;
  private static int threadcount = 0;   // client correnti

  public static int threadCount() {
    return threadcount;
  }
  public static int threadTotalCount() {
    return counter;
  }
  static synchronized void addToThreadCount(int d) {
    threadcount+=d;
  }
```

Multi-Client per più connessioni

```
public JabberClientThread(InetAddress addr) {
    synchronized(JabberClientThread.class) { id = counter++; }
    System.out.println("Making client " + id);
    addToThreadCount(1);
    try {
      socket = new Socket(addr, MultiJabberServer.PORT);
    } catch (IOException e){System.err.println("Socket failed");}
    try {
      in = new BufferedReader(
        new InputStreamReader(socket.getInputStream()));
      out = new PrintWriter(new BufferedWriter(
        new OutputStreamWriter(socket.getOutputStream())), true);
      start();
    } catch (IOException e) {
      System.err.println("Client thread "+id+" failure");
      try {
        socket.close();
      } catch (IOException e2) {
        System.err.println("Socket not closed");
}}}
```

Multi-Client per più connessioni

```
public void run() {
   try {
     for (int i = 0; i < 10; i++) {
       out.println("Client " + id + ": " + i);
       try { Thread.sleep(
               ThreadLocalRandom.current().nextInt(200,800));
       } catch (InterruptedException e) {  }
       String str = in.readLine();
       System.out.println(str);
     }
     out.println("END");
   } catch (IOException e) {
     System.err.println("IO Exception");
   } finally { // Always close it:
     try {  socket.close();
     } catch (IOException e) {
       System.err.println("Socket not closed");
     }
      addToThreadCount(-1);
} } }
```

Output del Client e del Server

- Client:

```
Making client 0
Making client 1
Making client 2
Making client 3
Client 3: 0
Client 0: 0
Client 1: 0
Client 2: 0
Client 1: 1
Client 2: 1
Client 0: 1
Client 3: 1
Client 2: 2
Client 3: 2
Client 1: 2
Client 0: 2
Client 1: 3
Client 2: 3
Client 3: 3
Client 2: 4
Client 0: 3
Client 1: 4
Client 3: 4
Client 0: 4
```

- Server:

```
Server Started
Echoing: Client 0: 0
Echoing: Client 1: 0
Echoing: Client 2: 0
Echoing: Client 3: 0
Echoing: Client 3: 1
Echoing: Client 0: 1
Echoing: Client 1: 1
Echoing: Client 2: 1
Echoing: Client 1: 2
Echoing: Client 2: 2
Echoing: Client 0: 2
Echoing: Client 3: 2
Echoing: Client 2: 3
Echoing: Client 3: 3
Echoing: Client 1: 3
Echoing: Client 0: 3
Echoing: Client 1: 4
Echoing: Client 2: 4
Echoing: Client 3: 4
Echoing: Client 2: 5
Echoing: Client 0: 4
```

Caratteristiche del programma appena visto

- Nel **`main`** della classe **`MultiJabberClient`** la creazione di un nuovo Thread può essere rallentata per più o meno millisecondi.
- Ad esempio
  - **`Thread.currentThread().sleep(1000)`**: pochissimi Thread client saranno in esecuzione contemporaneamente
  - **`Thread.currentThread().sleep(1)`**: molti Thread client potrebbero essere in esecuzione contemporaneamente
  - **`//Thread.currentThread().sleep(1)`**: sicuramente viene raggiunto il numero massimo di Thread in esecuzione
- ATTENZIONE: quando si scrive(legge) su un socket, se non c'è abbastanza spazio(dati) disponibile nel buffer gestito dal S.O., la chiamata di scrittura **`out.println()`** o di lettura **`in.readLine()`** si blocca.
  NB: Se un thread si blocca in lettura o scrittura, allora non è più in grado di fare altro

Esempio: Client che risponde alle domande del Server

- Questo esempio mostra un Client/Server (TCP) che si scambiano domande e risposte.
- Creiamo un file QnA.txt con domande seguite da risposte, un per ogni riga. Ad esempio

```
What caused the craters on the moon?
meteorites
How far away is the moon (in miles)?
239000
How far away is the sun (in millions of miles)?
93
Is the Earth a perfect sphere?
no
What is the internal temperature of the Earth (in degrees F)?
9000
```

Client che risponde alle domande del Server

```
public class QAClient {
  Socket socket = null;
  BufferedReader in = null;
  PrintWriter out = null;
  QAClient(String address){
    try {
      socket = new Socket(address, QAServer.PORTNUM);
      in = new BufferedReader(new
                   InputStreamReader(socket.getInputStream()));
      out = new PrintWriter(new BufferedWriter(new
                   OutputStreamWriter(socket.getOutputStream())), true);
    } catch (IOException e) {
      System.err.println("Couldn't create stream socket");
      System.exit(1);
    }
  }
  public static void main(String[] args) {
    if (args.length != 1) {
      new QAClient("localhost").exec();
    } else
      new QAClient(args[0]).exec();
  }
}
```

Client che risponde alle domande del Server

```
void exec() {
   StringBuffer userAnswer = new StringBuffer(128);
   String serverQuestion;
   int c;
   try {
     while ((serverQuestion = in.readLine()) != null) {
       System.out.println("Server: " + serverQuestion);
       if (serverQuestion.equals("END"))
         break;
       while ((c = System.in.read()) != '\n')
         userAnswer.append((char) c);
       System.out.println("Client: " + userAnswer);
       out.println(userAnswer.toString());
       userAnswer.setLength(0);
     }
     out.close();
     in.close();
     socket.close();
   } catch (IOException e) {
     System.err.println("I/O error trying to talk to server");
   }
}
```

Server che fa le domande al Client

```
public class QAServer extends Thread{
  public static final int PORTNUM = 1234;
  enum QAserverState {WAITFORCLIENT, WAITFORANSWER, WAITFORCONFIRM};
  private ArrayList<String> questions = new ArrayList<String>();
  private ArrayList<String> answers = new ArrayList<String>();
  private ServerSocket serverSocket;
  private int numQuestions;
  private int num = 0;
  private QAserverState state = QAserverState.WAITFORCLIENT;
  private File qaFile;
  private Random rand = new Random(System.currentTimeMillis());
```

Server che fa le domande al Client

```
public QAServer(String fileName) {
   try {
     serverSocket = new ServerSocket(PORTNUM);
     System.out.println("Server up and running...");
   } catch (IOException e) {
     System.err.println("Exception: couldn't create socket");
     System.exit(1);
   }
   qaFile = new File(fileName);
   if(!qaFile.exists()) {
     System.err.println("Error: "+ fileName+" doesn't exist!");
     System.exit(1);
   }
   if (!initQnA()) {
     System.err.println("Error: couldn't initialize Q&A");
     System.exit(1);
   }
 }
```

Server che fa le domande al Client

```
private boolean initQnA() {
   BufferedReader br=null;;
   try {
     br = new BufferedReader(new InputStreamReader(
                 new DataInputStream(new FileInputStream(qaFile))));
     String strLine;
     while ((strLine = br.readLine()) != null) {
       questions.add(strLine);
       if ((strLine = br.readLine()) != null)
         answers.add(strLine);
       numQuestions++;
     }
   } catch (IOException e) {
     System.err.println("I/O error trying to read questions");
     return false;
   } finally {
     try {  br.close(); } catch (Exception e) {  }
   }
   return true;
 }
```

Server che fa le domande al Client

```
public void run() {
   Socket clientSocket = null;
   while (true) {
     if (serverSocket == null)
       return;
     try {
       clientSocket = serverSocket.accept();
     } catch (IOException e) { System.exit(1); }
     try {
       BufferedReader is = new BufferedReader(
         new InputStreamReader(clientSocket.getInputStream()));
       PrintWriter os = new PrintWriter(new BufferedWriter(
         new OutputStreamWriter(clientSocket.getOutputStream())), true);
       String inLine;
       os.println(processInput(null));
       while ((inLine = is.readLine()) != null) {
         String outLine = processInput(inLine);
         os.println(outLine);
         if (outLine.equals("END"))
           break;
       }
       os.close(); is.close(); clientSocket.close();
     } catch (Exception e) { System.err.println("Exception "+e);}
   }
}
```

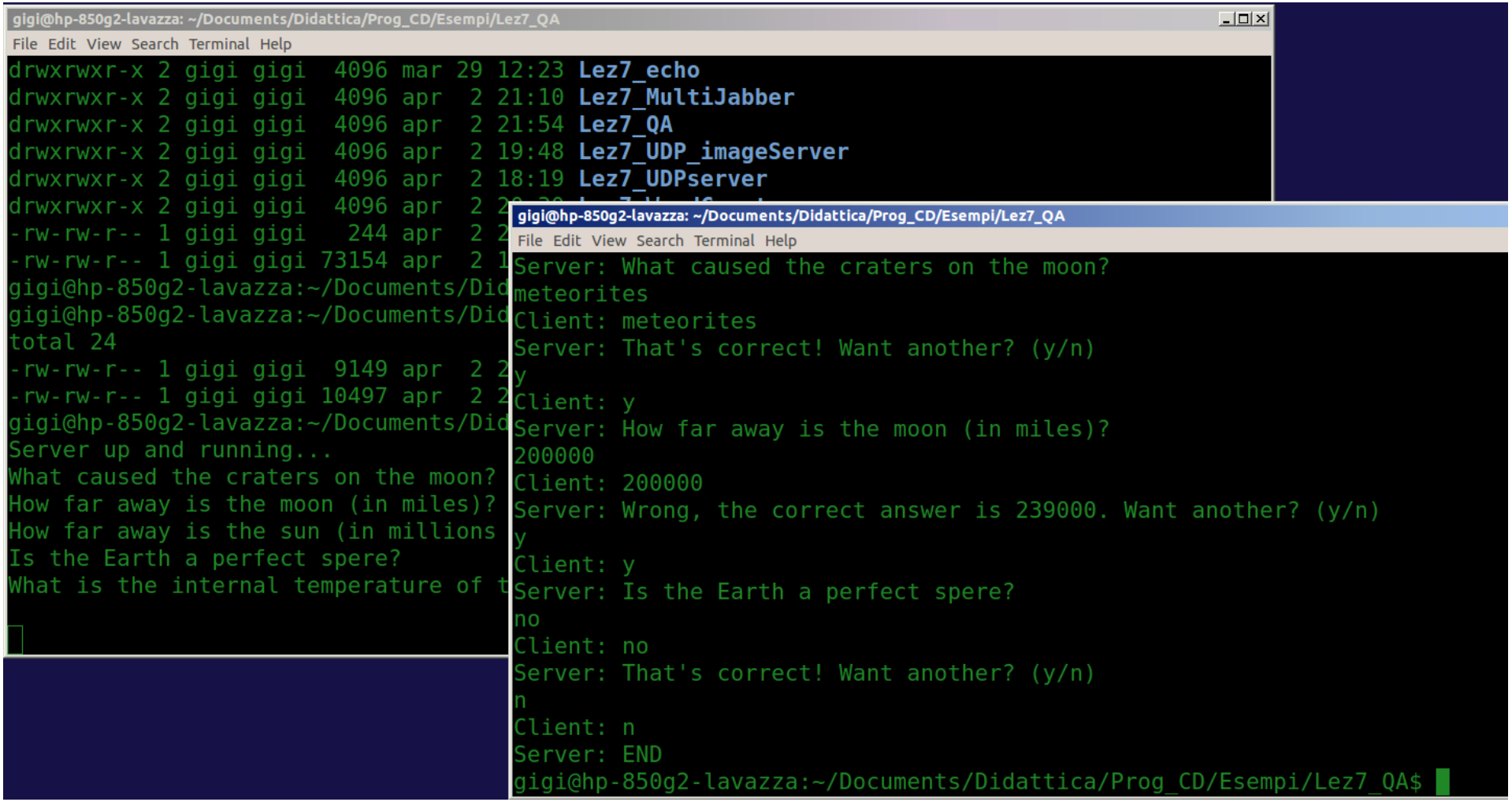
Server che fa le domande al Client

```
String processInput(String inStr) {
   String outStr = "";
   switch (state) {
   case WAITFORCLIENT:  // Ask a question
     outStr = questions.get(num);
     state = QAserverState.WAITFORANSWER;
     break;
   case WAITFORANSWER:  // Check the answer
     if (inStr.equalsIgnoreCase(answers.get(num)))
       outStr = "That's correct! Want another? (y/n)";
     else
       outStr = "Wrong, the correct answer is " + answers.get(num) +
                ". Want another? (y/n)";
     state = QAserverState.WAITFORCONFIRM;
     break;
```

Server che fa le domande al Client

```
    case WAITFORCONFIRM:  // See if they want another question
      if (inStr.equalsIgnoreCase("y")) {
        num = Math.abs(rand.nextInt()) % questions.size();
        outStr = questions.get(num);
        state = QAserverState.WAITFORANSWER;
      } else {
        outStr = "END";
        state = QAserverState.WAITFORCLIENT;
      }
      break;
    }
    return outStr;
  }
  public static void main(String[] args) {
    if (args.length != 1) {
      System.out.println("Usage: java QAServer <QA file name>");
      return;
    } else {
      new QAServer(args[0]).start();
    }
  }
```

Funzionamento



QA: Server multi client

- Modifichiamo il programma server in modo che accetti molti client contemporaneamente.
- Il programma client non ha bisogno di essere modificato.

Server

Non è un thread!

```
public class QAServer {
  public static final int PORTNUM = 1234;
  private ArrayList<String> questions = new ArrayList<String>();
  private ArrayList<String> answers = new ArrayList<String>();
  private ServerSocket serverSocket;
  private int numQuestions;
  private int num = 0;
  private File qaFile;
```

Server

```
public QAServer(String fileName) {
   try {
     serverSocket = new ServerSocket(PORTNUM);
     System.out.println("Server up and running...");
   } catch (IOException e) {
     System.err.println("Exception: couldn't create socket");
     System.exit(1);
   }
   qaFile = new File(fileName);
   if(!qaFile.exists()) {
     System.err.println("Error: "+ fileName+" doesn't exist!");
     System.exit(1);
   }
   if (!initQnA()) {
     System.err.println("Error: couldn't initialize Q&A");
     System.exit(1);
   }
 }
```

Server

```
private boolean initQnA() {
  BufferedReader br=null;;
  try {
    br = new BufferedReader(new InputStreamReader(
       new DataInputStream(new FileInputStream(qaFile))));
    String strLine;
    while ((strLine = br.readLine()) != null) {
      questions.add(strLine);
      if ((strLine = br.readLine()) != null)
        answers.add(strLine);
      numQuestions++;
    }
  } catch (IOException e) {
    System.err.println("I/O error trying to read questions");
    return false;
  } finally {
    try {  br.close(); } catch (Exception e) {  }
  }
  return true;
}
```

Server

```
public void exec() {
    Socket clientSocket = null;
    while (true) {
      if (serverSocket == null)
        return;
      try {
        clientSocket = serverSocket.accept();
      } catch (IOException e) { System.exit(1); }
      new QASlave(clientSocket, questions, answers).start();
    }
  }
public static void main(String[] args) {
    if (args.length != 1) {
      System.out.println("Usage: java QAServer <QA file name>");
      return;
    } else {
      new QAServer(args[0]).exec();
    }
  }
}
```

Slave

È un thread!

```
public class QASlave extends Thread{
  enum QAserverState {WAITFORCLIENT, WAITFORANSWER,
                      WAITFORCONFIRM};
  private ArrayList<String> questions = new ArrayList<String>();
  private ArrayList<String> answers = new ArrayList<String>();
  private int num = 0;
  Socket clientSocket;
  private QAserverState state = QAserverState.WAITFORCLIENT;
  private Random rand = new Random(System.currentTimeMillis());
  BufferedReader is;
  PrintWriter os;
  public QASlave(Socket s, ArrayList<String> q,
                           ArrayList<String> a) {
    questions=q;
    answers=a;
    clientSocket=s;
  }
```

Slave

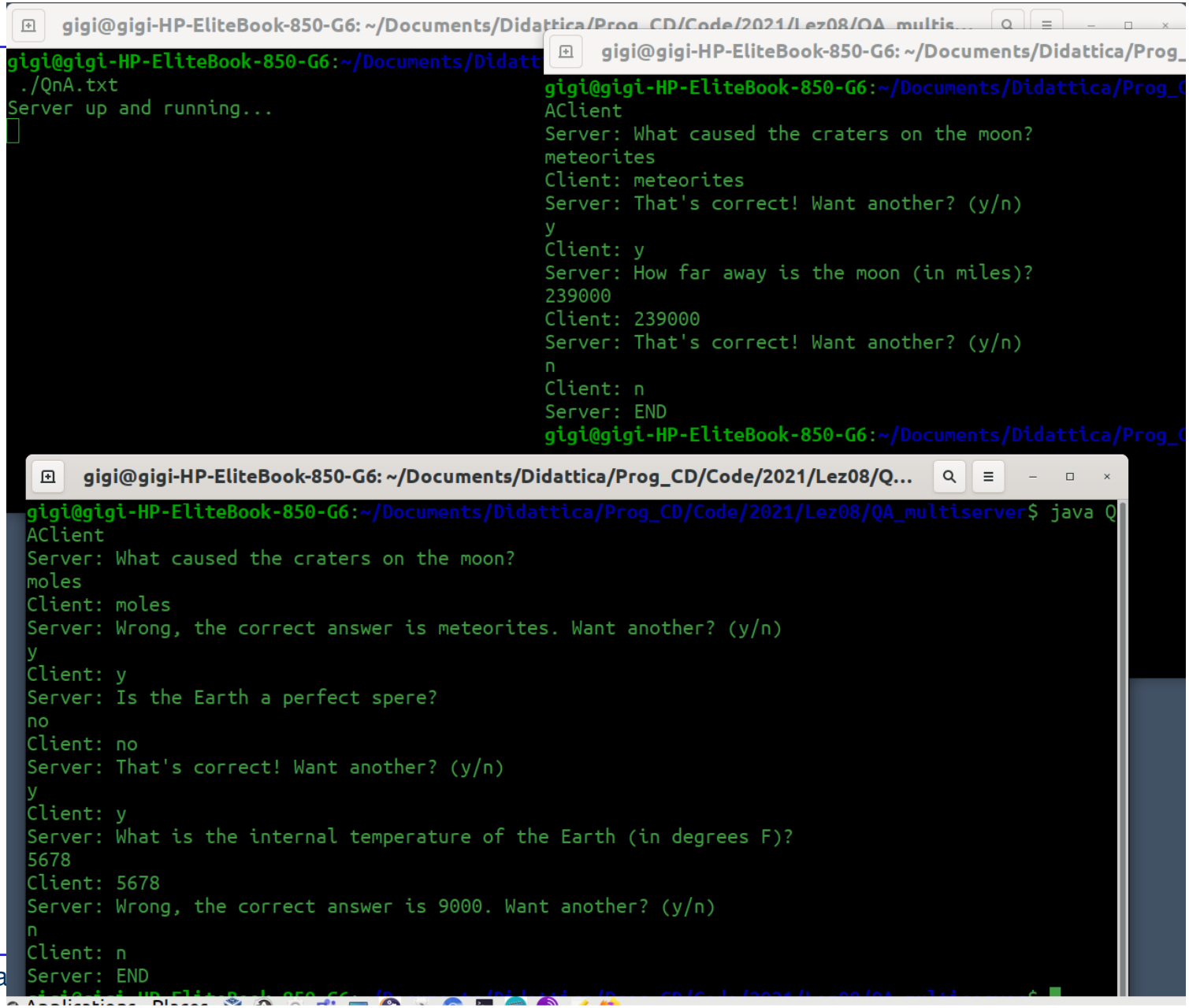
Come prima
(la logica non cambia)

```
String processInput(String inStr) {
   String outStr = "";
   switch (state) {
   case WAITFORCLIENT:  // Ask a question
     outStr = questions.get(num);
     state = QAserverState.WAITFORANSWER;
     break;
   case WAITFORANSWER:  // Check the answer
     if (inStr.equalsIgnoreCase(answers.get(num)))
       outStr = "That's correct! Want another? (y/n)";
     else
       outStr = "Wrong, the correct answer is " + answers.get(num) +
                ". Want another? (y/n)";
     state = QAserverState.WAITFORCONFIRM;
     break;
   case WAITFORCONFIRM:  // See if they want another question
     if (inStr.equalsIgnoreCase("y")) {
       num = Math.abs(rand.nextInt()) % questions.size();
       outStr = questions.get(num); state = QAserverState.WAITFORANSWER;
     } else {
       outStr = "END"; state = QAserverState.WAITFORCLIENT;
     }
     break;
   }
   return outStr;
}
```

QAserver_thread

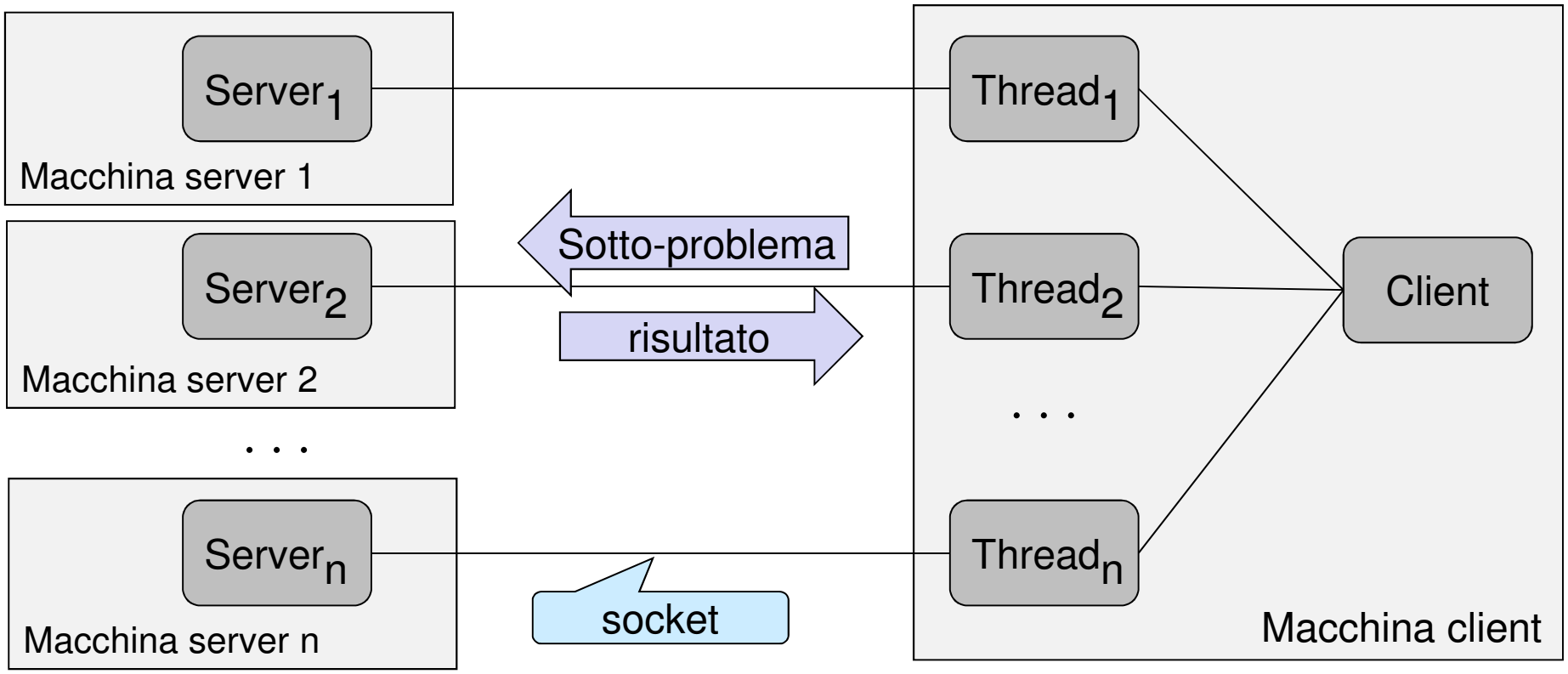
```
public void run() {
   String inLine;
   try {
     is = new BufferedReader(
       new InputStreamReader(clientSocket.getInputStream()));
     os = new PrintWriter(new BufferedWriter(new
     OutputStreamWriter(clientSocket.getOutputStream())), true);
     os.println(processInput(null));
     while ((inLine = is.readLine()) != null) {
       String outLine = processInput(inLine);
       os.println(outLine);
       if (outLine.equals("END"))
         break;
     }
     os.close(); is.close(); clientSocket.close();
   } catch (Exception e) { System.err.println("Exception "+e);}
 }
```

Test con due client



Molteplici server e parallelismo

- In molti casi è possibile suddividere un problema in tanti sotto-problemi indipendenti analizzabili singolarmente in parallelo.
- In particolare quando si hanno tanti processi server che forniscono tutti lo stesso servizio e girano ciascuno su un processore diverso



\esempio: Word Count

- Dato un insieme di documenti di testo, l'obiettivo è contare il numero di parole presenti nei documenti.
- Dividiamo il problema complessivo in un insieme di piccoli task, sullo stile di Map-Reduce.
- Ci sono N server disponibili.
- Il client genere N thread, ciascuno dei quali comunica con un server, chiedendogli di risolvere un sotto-problema.
  - Se girano su macchine diverse, i server lavorano effettivamente parallelo
- Il client raccoglie i risultati dai suoi vari thread e li assembla.

Word Count - Client

```
public class WordCountClient {
  int parties;
  String theArgs[];
  WordCountClient(String args[]){
    parties=args.length/2;
    theArgs=args;
  }

  public static void main(String args[]) {
    WordCountClient wcc=new WordCountClient(args);
    try {
      wcc.exec();
    } catch (Exception e) {
      System.err.println("qualcosa storto ando`");
    }
  }
}
```

Word Count - Client

```
void exec() throws UnknownHostException, IOException{
  int port;
  String addr;
  ArrayList<WordCounter> counters=new ArrayList<WordCounter>();
  long t0=System.currentTimeMillis();
  BarrierReachedAction reducer=new BarrierReachedAction(t0);
  CyclicBarrier cyclicBarrier=new CyclicBarrier(parties,
                                                reducer);
  int incr=400/parties;
  for(int i=0; i<parties; i++) {
    addr=theArgs[i*2];
    port=Integer.parseInt(theArgs[i*2+1]);
    counters.add(new WordCounter(addr, port, cyclicBarrier,
                                 (i)*incr, (i+1)*incr));
  }
  reducer.setCounters(counters);
}
```

Word Count – Client Thread

```
public class WordCounter extends Thread {
  CyclicBarrier cyclicBarrier;
  public int wordCount=0;
  int startFile, endFile;
  int thePORT;
  InetAddress addrIP;
  WordCounter(String addr, int port, CyclicBarrier c, int start,
              int end) throws UnknownHostException, IOException {
    cyclicBarrier=c;
    startFile=start;
    endFile=end;
    thePORT=port;
    addrIP = InetAddress.getByName(addr);
    System.out.println("word counter slave starting with addr="
                       +addrIP+" and port="+thePORT);
    this.start();
  }
```

Word Count – Client Thread

```
public void run() {
    String str;
    try {
      Socket socket = new Socket(addrIP, thePORT);
      BufferedReader in = new BufferedReader(
        new InputStreamReader(socket.getInputStream()));
      PrintWriter out = new PrintWriter(new BufferedWriter(
        new OutputStreamWriter(socket.getOutputStream())), true);
      out.println(startFile);
      out.println(endFile);
      str = in.readLine();
      wordCount=Integer.parseInt(str);
      System.out.println("word counter slave counted "+
                         wordCount+" words; going to wait");
      cyclicBarrier.await();
    } catch (Exception e1) {
      System.out.println("word counter slave aborting "+e1);
    }
  }
}
```

Word Count – Barrier implementa reduction

```
public class BarrierReachedAction implements Runnable {
  private ArrayList<WordCounter> wordCounters;
  long t0=0;
  public BarrierReachedAction(long start) {
    t0=start;
  }
  public void setCounters(ArrayList<WordCounter> wordCounters) {
    this.wordCounters=wordCounters;
  }
  public void run() {
    System.out.println("BarrierReached!");
    int totalWords=0;
    for(WordCounter wc:wordCounters) {
      totalWords+=wc.wordCount;
    }
    long totalTimeElapsed=System.currentTimeMillis()-t0;
    System.out.println("Elapsed time = "+totalTimeElapsed);
    System.out.println("Word count is = " + totalWords);
  }
}
```

Word Count – Server

```
public class CounterServer {
  int wordCount=0;
  String filePath;
  int startFile, endFile;
  int myPORT;
  BufferedReader in;
  PrintWriter out;
  CounterServer(String path, String port){
    myPORT=Integer.parseInt(port);
    filePath=path;
  }
  public static void main(String args[]) throws IOException {
    if(args.length!=2) {
      System.out.println("usage: java CounterServer <file path>
<PORT>");
    } else {
      CounterServer CS=new CounterServer(args[0], args[1]);
      CS.serve();
    }
  }
```

Word Count – Server

```
  void serve() throws IOException {
    String str;
    ServerSocket server = new ServerSocket(myPORT);
    Socket connection = null;
    while (true) {
      try {
        connection = server.accept();
        in=new BufferedReader(new
InputStreamReader(connection.getInputStream()));
        out=new PrintWriter(new BufferedWriter(new
OutputStreamWriter(connection.getOutputStream())),true);
        str=in.readLine();
        this.startFile=Integer.parseInt(str);
        str=in.readLine();
        this.endFile=Integer.parseInt(str);
        count();  // qui c'e` il servizio
        out.println(wordCount);
        System.out.println("server sent:"+wordCount);
        connection.close();
      }
```

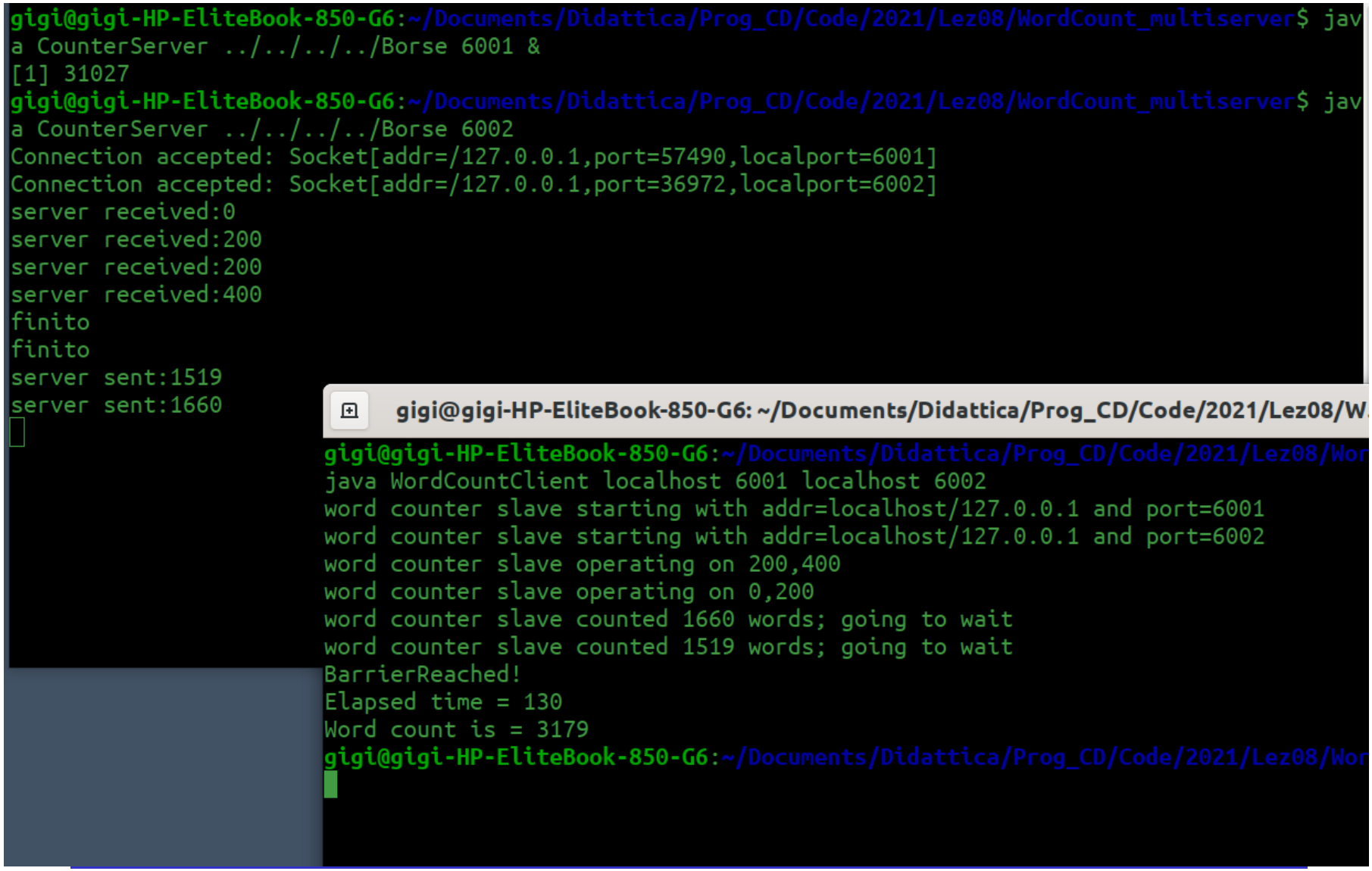
Word Count – Server

```
// conclusione serve()
      catch (IOException ex) {
        System.err.println("server IO error");
      }
      finally {
        try {
          if (connection != null) connection.close();
        } catch (IOException ex) {}
      }
    }
  }
```

Word Count – Server

```
public void count() {
  BufferedReader br = null;
  this.startFile=startFile;
  this.endFile=endFile;
  try {
    for(int i=startFile; i<endFile; i++) {
      br=null;
      String fileName=filePath+"/file_" + i + ".txt";
      br=new BufferedReader(new FileReader(fileName));
      String line=br.readLine();
      while(line!=null) {
        String[] wordArray=line.split("\\ s+");
        wordCount+=wordArray.length;
        line=br.readLine();
      }
    }
    br.close();
  } catch(Exception exc) { System.out.println(exc); }
}
```

Test con due server



Bibliografia

- Molte delle informazioni presentate sono state estratte dal capitolo 1 del libro:
  - Thinking in Enterprise Java
  - by Bruce Eckel et. Al.