SOCKET

Le **socket** rappresentano il **terminale** (end point) di un **canale di comunicazione bidirezionale**

Un Client e un Server su macchine diverse possono comunicare sfruttando due diversi tipi di modalità di comunicazione

- **con connessione**, in cui viene stabilita una connessione tra Client e Server (esempio, il sistema telefonico)

  Uso di **socket STREAM**

- **senza connessione**, in cui non c'è connessione e i messaggi vengono recapitati uno indipendentemente dall'altro (esempio, il sistema postale)

  Uso di **socket DATAGRAM**

Sistema di NOMI

Necessità di definire un sistema di identificazione per le diverse entità messe in gioco

*Un'**applicazione distribuita** è costituita da processi distinti per località che comunicano e cooperano attraverso lo scambio di messaggi, per ottenere risultati coordinati*

Il primo problema da affrontare riguarda la **identificazione dei processi** (il Client o il Server) nella rete

Per ogni processo bisogna definire un
nome globale visibile in modo univoco e sempre non ambiguo

"nome" della macchina
+
"nome" del processo all'interno della macchina

Gli **endpoint** di processo (**socket**) sono tipicamente locali al processo stesso
(**livello applicativo o sottostante fino a sessione**)

Il problema è risolto dai livelli sottostanti di protocollo

per le socket nel dominio Internet
i nomi di trasporto (**TCP, UDP**) e rete (**IP**)

NOMI per SOCKET

Una macchina è identificata univocamente da un **indirizzo IP** (**4 byte** / **32 bit**) (livello IP)

La **porta** è un numero intero di 16 bit
(astrazione fornita dal TCP e da UDP)

NOME GLOBALE

I messaggi sono consegnati su una specifica porta di una macchina, non direttamente a un processo

NOME LOCALE

*Un processo si **lega** a una porta per ricevere (o spedire) dei messaggi*
Anche più processi

In questo modo è possibile identificare un processo senza dover conoscere il suo process identifier (pid)

Un indirizzo IP e una porta possono rappresentare un endpoint di un canale di comunicazione

NOMI GLOBALI

Numeri IP => vedi protocollo IP

indirizzo IP: **ad es. 137.204.57.186**

Numeri di Porta

porte **4 cifre hex: XXXXh** espresse in decimale

ad es. 153, 2054

Funzione fondamentale delle porte è **identificare un servizio**

I numeri di porta minori di 1024 sono riservati
(well-known ports, o reserved)

sono **standardizzati** i servizi offerti dai processi che a tale porta sono legati

Per esempio, il servizio **Web** è identificato dalla **porta** numero **80**, cioè il processo server di un sito Web deve essere legato alla porta 80, su cui riceve i messaggi con le richieste di pagine html
Altri esempi:
porta 21 per ftp,
porta 23 per telnet,
porta 25 per mail,…

La richiesta di un servizio a un processo server non richiede quindi la conoscenza del suo pid, ma solo della porta e IP (nomi globali)

La Programmazione di rete in Java

Java fornisce per la gestione della comunicazione le classi del package di networking **`java.net`**

Classi per implementare le socket:

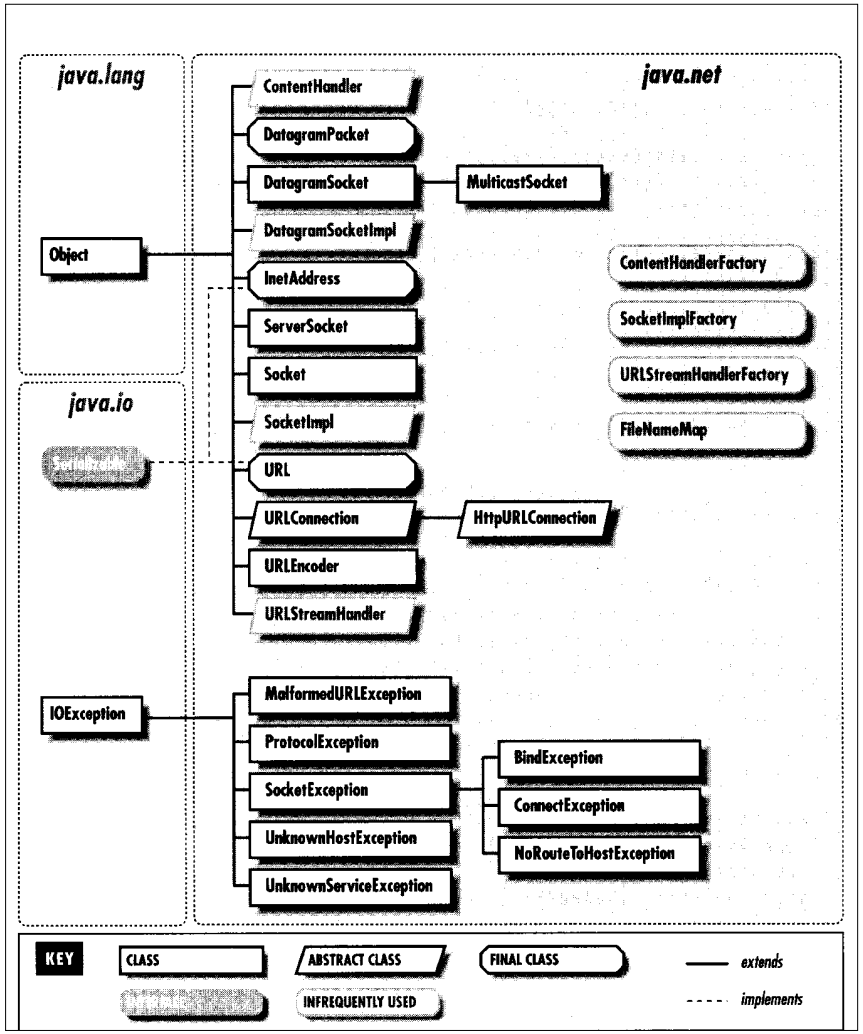
- con **connessione (TCP)**
  - classe `Socket`, per socket lato **Client**
  - classe `ServerSocket`, per socket lato **Server**

- senza **connessione (UDP)**
  - classe `DatagramSocket`
  - classe `MulticastSocket`

Classi di supporto:

- classe `URL`
- classe `URLConnection`
- classe `InetAddress`
- classe `HttpURLConnection`
- classe `JarURLConnection`
- classe `DatagramPacket`

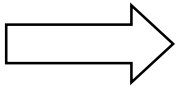
Gerarchia del package `java.net`



Le classi sono state significativamente estese dalla diverse versioni delle JVM
per tenere conto delle diverse necessità applicative

La Classe InetAddress

Classe con cui vengono rappresentati gli indirizzi Internet, astraendo dal modo con cui vengono specificati (a numeri o a lettere)



Portabilità e trasparenza

No costruttori, utilizzo di tre metodi statici:

public static InetAddress **getByName**(String hostname);

restituisce un oggetto InetAddress rappresentante l'host specificato (come nome o indirizzo numerico); con il parametro **`null`** ci si riferisce all'indirizzo di default della macchina locale

public static InetAddress[] **getAllByName**(String hostname);

restituisce un array di oggetti InetAddress; utile in casi di più indirizzi IP registrati con lo stesso nome logico

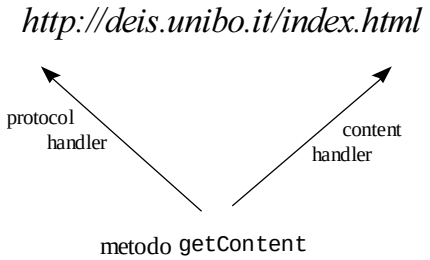
public static InetAddress **getLocalHost**();

restituisce un oggetto InetAddress corrispondente alla macchina locale; se tale macchina non è registrata oppure è protetta da un firewall, l'indirizzo è quello di loopback: 127.0.0.1

Tutti possono sollevare l'eccezione UnknownHostException se l'indirizzo specificato non può essere risolto (tramite il DNS)

La Classe URL

Inoltre visto l'ampio uso di **Indirizzi Web** Java definisce una classe per descrivere le risorse Internet



La classe URL è il meccanismo di base per la costruzione di browser

Esempio: Scaricamento di una pagina

```
...
URL deis = new URL
   ("http://www.deis.unibo.it/");
BufferedReader in = new BufferedReader(
   new InputStreamReader(
        deis.openStream()));

String inLine;
 while ((inLine = in.readLine()) != null)
   System.out.println(inLine);
in.close();
...
```

non solo la possibilità di agganciarsi dinamicamente ad una URL ma di gestire il protocollo HTTP

SOCKET per STREAM

Le socket STREAM sono i terminali di un canale di comunicazione virtuale **con connessione** creato tra il Client e il Server. La comunicazione avviene in modo

bidirezionale,
affidabile,
con dati (byte) consegnati in sequenza
(modalità **FIFO** come sulle pipe di Unix)

La **connessione** tra i processi Client e Server è definita non dai processi che devono cumincare ma da una quadrupla univoca
<protocollo; indirizzo IP Client; porta Client; indirizzo IP Server; porta Server>

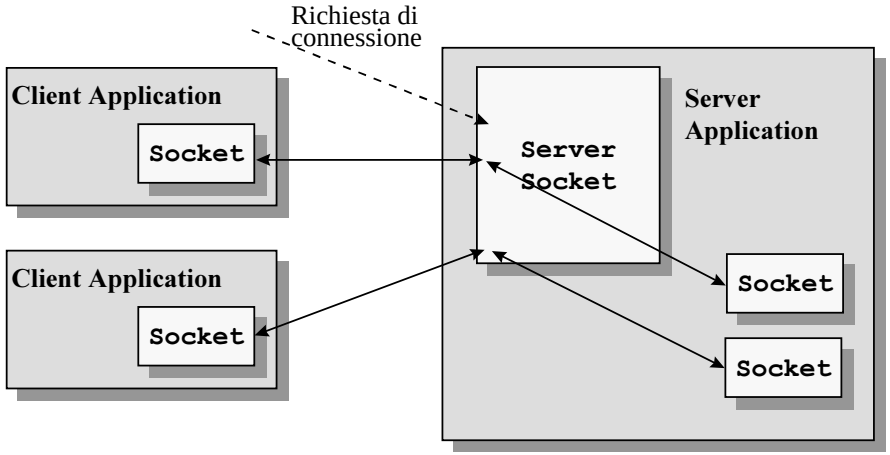
Nel caso delle socket STREAM, il protocollo di comunicazione sottostante è il protocollo TCP (+ IP)

- TCP è un protocollo di trasporto, livello 4 OSI e fornisce l'astrazione porta
- IP è un protocollo di rete, livello 3 OSI e identifica univocamente ogni macchina collegata alla rete Internet

La comunicazione tra Client e Server su stream segue uno schema **asimmetrico** e il principio della **connessione (relative API diverse)**
Questa considerazione ha portato al progetto di due tipi di socket diverse una per il Client e una per il Server

Socket Stream in Java

Protocollo di Comunicazione **con Connessione** in Java



Classi distinte per ruoli **Cliente** e **Servitore**

La classe **`java.net.Socket`**
per il client

La classe **`java.net.ServerSocket`**
per il server

Lato CLIENT

La classe `Socket` consente di creare una socket con **connessione**, stream (**TCP**) per il collegamento di un **Client** a un Server.
Socket "attiva"

Costruttori

I costruttori della classe creano la socket, la legano a una porta locale e la connettono a una porta di una macchina remota (*Unix: socket, bind, connect*)

⇒ tendono a nascondere i dettagli realizzativi

```
public Socket(InetAddress remoteHost,
              int remotePort) throws ...
```

Crea una **socket stream cliente** e la collega alla porta specificata della macchina all'indirizzo IP dato (equivale in Unix a: socket, bind, connect)

```
public Socket(String remoteHost,
              int remotePort) throws ...
```

Crea una **socket stream cliente** e la collega alla porta specificata della macchina di nome logico `String`


```
public Socket(InetAddress remoteHost,
              int remotePort,
              InetAddress localInterface,
              int localPort) throws ...
```

Crea una **socket stream cliente** e la collega

- sia a una porta della macchina locale
  (se `localPort` vale zero, il numero di porta è scelto automaticamente dal sistema)
- sia a una porta della macchina remota
  (macchine multihomed)

*La **creazione** della socket produce in modo atomico anche la **connessione** al server corrispondente o lancia la eccezione opportuna*

Connessioni

La creazione di una socket a stream se va a buon fine produce una **connessione bidirezionale a byte** (stream) tra i due processi interagenti e impegna risorse tra i nodi e tra i processi

La connessione permette poi una
comunicazione bidirezionale (full duplex)

APERTURA
ottenuta con il costruttore in modo implicito

CHIUSURA
Operazione da fare in modo esplicito, necessaria per non impegnare troppe risorse di sistema

Le connessioni sono risorse:
costa crearle e distruggerle, ma anche mantenerle.
=> Si tendono a mantenere le sole connessioni necessarie
=> Limiti al numero di connessioni che un processo (Client o Server) può aprire contemporaneamente
=> si chiudono le connessioni non utilizzate

Il metodo **close**() chiude l'oggetto socket e disconnette il Client dal Server

```
public synchronized void close()
```


Operazioni di supporto

Per ottenere informazioni sulle socket si possono utilizzare i metodi:

```
public InetAddress getInetAddress()
```
restituisce l'indirizzo della macchina remota a cui la socket è connessa

```
public InetAddress getLocalAddress()
```
restituisce l'indirizzo della macchina locale

```
public int getPort()
```
restituisce il numero di porta sulla macchina remota a cui la socket è connessa

```
public int getLocalPort()
```
restituisce il numero di porta sulla macchina locale a cui la socket è legata

Esempio:

```
int porta = oggettoSocket.getPort();
```

Operazioni di comunicazione

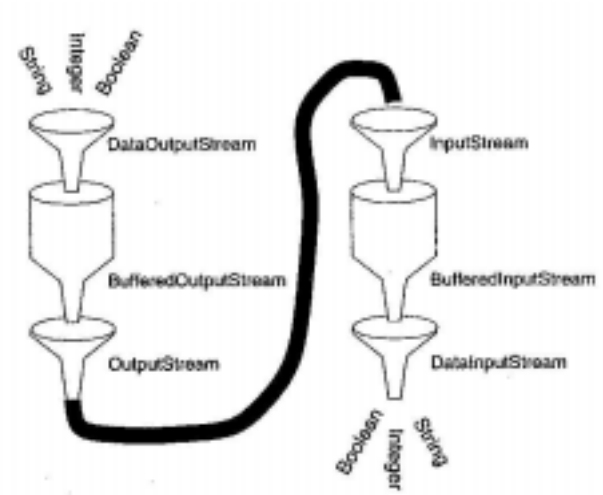
Lettura/scrittura vengono effettuate da/su stream, ottenuto con i metodi:

`public InputStream` **`getInputStream`**`()`
ritorna l'input stream per leggere byte dalla socket

`public OutputStream` **`getOutputStream`**`()`
ritorna l'output stream per scrivere byte sulla socket

I due metodi restituiscono un oggetto stream (`InputStream` e `OutputStream`) attraverso cui si possono spedire/ricevere solo byte, senza nessuna formattazione; altri oggetti possono incapsulare questi stream, per fornire funzionalità di più alto livello (es. DataInputStream)

Uso di stream tipici di Java



Creazione stream di input da socket:

```
InputStreamReader isr = new
InputStreamReader(socket.getInputStream());
BufferedReader in = new BufferedReader(isr);
```

Creazione stream di output su socket:

```
OutputStreamWriter osw = new
OutputStreamWriter(socket.getOutputStream());
BufferedWriter bw = new BufferedWriter(osw);
PrintWriter out = new PrintWriter( bw, true);
```

Esempio:

Client di echo (il Server Unix e' sulla porta 7)

```
. . .
try {
  oggSocket = new Socket(hostname, 7);
/* input ed output sugli endpoint
della connessione via socket */
  out = new PrintWriter
    (oggSocket.getOutputStream(),true);
  in = new BufferedReader(
   new InputStreamReader
        (oggSocket.getIntputStream());
/* lettura da input */
  userInput = new newBufferedReader(
   new InputStreamReader(System.in));

while (true)
{   oggLine = userInput.readLine();
    if (oggLine.equals(".")) break;
    out.println(oggLine);
    System.out.println(in.readLine());
}
oggSocket.close();
   }  // fine try
  catch (IOException e)
     { System.err.println(e);}
...
```


Lato SERVER

La classe `ServerSocket` definisce una socket capace di accettare richieste di connessione provenienti dai Client
Socket "passiva"

Consente di gestire:

- più richieste di connessione pendenti allo stesso tempo (definisce anche la lunghezza della coda in cui vengono messe le richieste di connessione non ancora accettate dal server)
- più connessioni aperte contemporaneamente

Una volta **stabilita la connessione**
(ottenuta dal server tramite il metodo `accept`)
la trasmissione dei dati avviene attraverso un normale oggetto `Socket` del server (restituito dalla accept)

Costruttori

I costruttori della classe creano la socket, la legano a una porta locale e la connettono a una porta di una macchina remota (*Unix: socket, bind, connect*)

⇒ tendono a nascondere i dettagli realizzativi

```
public ServerSocket(int localPort)
   throws IOException, BindException;
```

crea una socket in ascolto sulla porta specificata

```
public ServerSocket(int localPort, int count)
   throws IOException, BindException;
```

crea una socket in ascolto sulla porta specificata con una coda di lunghezza count

Connessione

Il **Server** si mette in attesa di nuove richieste di connessione chiamando il metodo **`accept()`**

La invocazione di **accept** blocca il Server fino all'arrivo di una richiesta
Quando arriva una richiesta, **accept** stabilisce una **la reale connessione** tra il Client e un **nuovo oggetto `Socket`** restituito da accept e che rappresenta lo stream con il cliente

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

La `accept` restituisce un oggetto della classe `Socket` su cui avviene la fase di comunicazione tra il Client e il Server

la chiamata di accept mette il servitore in attesa di nuove richieste di connessione
Tipicamente se non ci sono ulteriori richieste, il servitore si blocca in attesa

La trasmissione dei dati avviene con i metodi visti per il lato Client in modo del tutto indifferente in uno o l'altro verso della connessione

⇒ **i due endpoint sono del tutto omogenei**
(uso di protocollo TCP)

Operazioni di supporto

`public InetAddress` **`getInetAddress`**`()`
restituisce l'indirizzo della macchina **locale** a cui la socket server è connessa

`public int` **`getLocalPort`**`()`
restituisce il numero di porta sulla macchina **locale**

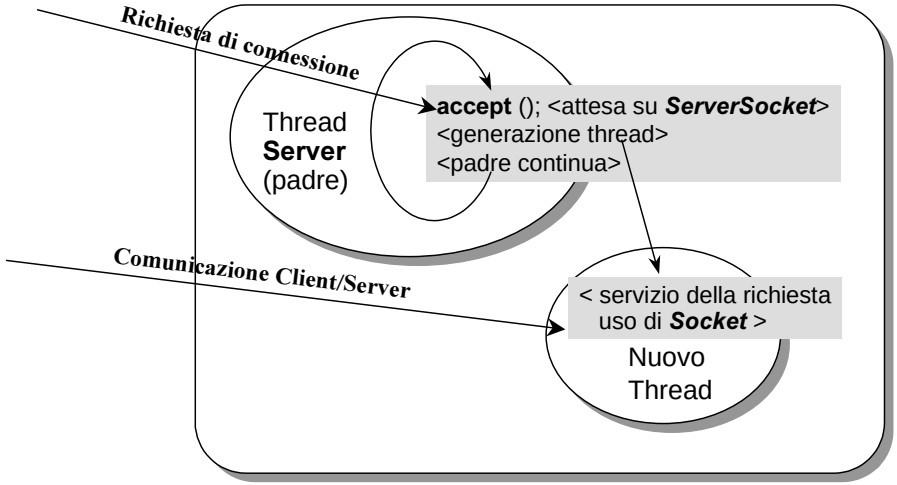
Esempio:

Server daytime (il Server Unix su porta 13)

```
. . .
oggServer =
   new ServerSocket(portaDaytime);
try {

/* il server alla connessione invia la data
al cliente */
while (true)
{ oggConnessione = oggServer.accept();
  out = new PrintWriter
   ( oggConnessione.getOutputStream(),
     true);
  Date now =new Date();
  out.write(now.toString()+ "\r\n");
  oggConnessione.close();
 }
    }
 catch (IOException e)
{
   oggServer.close();
   System.err.println(e);
 }
 . . .
```


Server Parallelo con connessione



Alla **accettazione** il servitore può generare
una nuova attività responsabile del servizio (che eredita la connessione nuova)

Il servitore principale può
aspettare nuove richieste e servire nuove operazioni

SOCKET per DATAGRAM

Le socket DATAGRAM permettono a due processi di comunicare scambiandosi messaggi senza stabilire una connessione.

La comunicazione è:

bidirezionale,
non affidabile,
con ordine di consegna dei messaggi non garantito

Nel caso delle socket DATAGRAM, il protocollo di comunicazione sottostante è il protocollo UDP (+ IP)

- UDP è un protocollo di trasporto, livello 4 OSI e fornisce l'astrazione porta
- IP è un protocollo di rete, livello 3 OSI e identifica univocamente ogni macchina collegata alla rete Internet

La comunicazione tra Client e Server tramite datagram segue uno schema **simmetrico**
Questa considerazione ha portato al progetto di un unico tipo di socket per il Client e per il Server

Socket Datagram in Java

```
public final class DatagramSocket
                                extends Object
```

Costruttore

```
DatagramSocket(InetAddress localAddress,
                int localPort) throws...
```

crea socket UDP legata localmente

Uso di **send** e **receive**

```
Sock.sendMessage("stringa");
Sock.send(DatagramPacket);
Sock.receive(DatagramPacket);
```

Operazioni di supporto

Esistono anche le classi per usare e preparare datagrammi che consentono di specificare come comunicare

Un pacchetto datagram specifica un array di byte da/su cui scrivere e con indicazioni di comunicazione con diversi costruttori

La Classe DatagramPacket

Classe con cui vengono rappresentati i pacchetti UDP da inviare e ricevere sulle socket di tipo Datagram.

Si costruisce un datagram packet specificando:

- il contenuto del messaggio (i primi ilength bytes dell'array ibuf)
- l'indirizzo IP del destinatario
- il numero di porta su cui il destinatario è in ascolto

```
public DatagramPacket(byte ibuf[], int ilength,
                      InetAddress iaddr, int iport)
```

Se il pacchetto deve essere ricevuto basta definire il contenuto:

```
public DatagramPacket(byte ibuf[], int ilength)
```

La classe mette a disposizione una serie di metodi per estrarre o settare le informazioni:

```
getAddress(), setAddress(InetAddress addr)
getPort(),    setPort(int port)
getData(),    setData(byte[] buf), etc.
```


Opzioni delle Socket

Si possono vedere alcune opzioni delle socket in Java con funzioni definite

`SetTcpNoDelay` `(boolean on) throws ...`
il pacchetto è inviato immediatamente, senza bufferizzare

`SetSoLinger` `(boolean on, int linger)`
dopo la close, il sistema tenta di consegnare i pacchetti ancora in attesa di spedizione. Questa opzione permette di scartare i pacchetti in attesa (linger in sec)

`SetSoTimeout` `(int timeout) throws...`
la lettura da socket (es., read()) è bloccante.
Questa opzione definisce un timeout (in msec), trascorso il quale si sblocca la read (ma viene lanciata una eccezione da gestire)

`SetSendBufferSize` `(int size) throws...`
il buffer di invio dello stream può essere variato
`SetReceiveBufferSize` `(int size)`
il buffer di invio dello stream può essere variato

`SetKeepAlive` `(boolean on) throws...`
abilita, disabilita la opzione di keepalive

Sono previste le get corrispondenti
alcune opzioni sono nella classe SocketOptions

SOCKET MULTICAST

Sono anche possibili ulteriori classi per
- inviare **messaggi multicast**
- creare **gruppi di multicast**

Preparazione del gruppo:
IP classe D e porta libera

```
InetAddress gruppo =
   InetAddress.getByName("229.5.6.7");
MulticastSocket s =
           new MulticastSocket(6666);
```

Operazioni di ingresso/uscita dal gruppo

```
// unisciti al gruppo ...
s.joinGroup(gruppo);

byte[] msg = {'H', 'e', 'l', 'l', 'o'};
DatagramPacket packet =
  new DatagramPacket
    (msg, msg.length, gruppo, 6666);
s.send(packet);

// ricevi la risposta ...
byte[] buf = new byte[1000];
DatagramPacket recv = new
     DatagramPacket (buf, buf.length);
s.receive(recv);
 ...
// esci dal gruppo ...
s.leaveGroup(group);
```


Esempio:

Remote CoPy (RCP)

Si realizzi un'applicazione distribuita Client/Server per eseguire la copia remota (remote copy, rcp) di file
Progettare
sia programma client
sia il programma server

Il programma Client deve consentire la invocazione:
rcp_client nodoserver portaserver nomefilesorgente nomefiledest
nodoserver e ***portaserver*** indicano il nome Server e ***nomefile*** è il nome assoluto di un file presente nel file system della macchina Client
Il processo Client deve inviare il file ***nomefile*** al Server che lo scrive nel direttorio corrente con nome ***nomefiledes***

La scrittura del file nel direttorio specificato deve essere eseguita **solo se** in tale direttorio non è presente un file con lo stesso nome, evitando di sovrascriverlo

La connessione aperta dal cliente consente al server di coordinarsi per la richiesta del file che viene inviato

RCP Client

Estratto dal client

```
...
rcpSocket = new Socket(host,porta);

StrDatiOutSocket =
   new DataOutputStream
      (rcpSocket.getOutputStream());
StrDatiInSocket =
   new DataInputStream
      (rcpSocket.getInputStream());
StrDatiOutSocket.writeUTF(FileDest);
Risposta = StrDatiInSocket.readUTF();
System.out.println(Risposta);

if (Risposta.equalsIgnoreCase
     ("MSGSrv: attendofile") == true)
{FDaSpedireDescrittore =
     new File(NFileOrigine);
 FDaSpedireInputStream =
     new FileInputStream
        (FDaSpedireDescrittore);
 byte ContenutoFile [] = new byte
  [FDaSpedireInputStream.available()];
 FDaSpedireInputStream.read
        (ContenutoFile);
 StreamDatiOutSocket.write
        (ContenutoFile);
}
} catch (IOException e)
{System.err.println(e);}
rcpSocket.close();
...
```


RCP Server iterativo

```
... try {
rcpSocketSrv =
   new ServerSocket(Porta);
System.out.println("Attesa su porta" +
   rcpSocketServer.getLocalPort());
while(true)
{SocketConn = rcpSocketServer.accept();
 System.out.println("Connesso con" +
      rcpSocketConn);
 StrDatiOutSocket = new DataOutputStream
  (rcpSocketConn.getOutputStream());
 StrDatiInSocket = new DataInputStream
  (rcpSocketConn.getInputStream());
 NFile = StreamDatiInSocket.readUTF ();
 FileDaScrivere = new File(NomeFile);
 if(FileDaScrivere.exists() == true)
 { StrDatiOutSocket.writeUTF(
      "MSGSrv: file presente, bye");
 } else
 { StrDatiOutSocket.writeUTF(
      "MSGSrv: attendofile");
   byte ContntFile [] = new byte [1000];
   StrDatiInSocket.read
      (ContntFile);
   FileDaScrivereOutputStream = new
      FileOutputStream (FileDaScrivere);
   FileDaScrivereOutputStream.write
      (ContntFile);
 }
 rcpSocketConn.close();
}
}catch (IOException e)
   {System.err.println(e);} ...
```

RCP Server concorrente parallelo

```
...
try
{
...rcpSocket =
   new ServerSocket(Porta);
System.out.println("Attesa su porta" +
   rcpSocketServer.getLocalPort());

 while(true)
 {
  rcpSocketConn = rcpSocket.accept();
  threadServizio = new rcp_servizio
   (rcpSocketConnessa);
  threadServizio.start();
 }

}
catch (IOException e)
{System.err.println(e);}
```

Si genera un nuovo processo per ogni connesione generata e su questo avviene la interazione

Si noti che le porte usate dai thread seaparatamente sono tutte contemporaneamente impegnate per il sistema di supporto

???e se c'è un limite al numero di socket aperte per processo???

RCP Server Processi Figli

```
public class rcp_servizio
   extends Thread {
...
rcp_servizio(Socket socketDaAccept)
 {rcpSocketSrv = socketDaAccept;}

public void run() {
 System.out.println("thread numero " +
      Thread.currentThread());
 System.out.println(""Connesso con" +
      + rcpSocketSrv);

try
{
 StrDatiOutSocket=new DataOutputStream
   (rcpSocketSrv.getOutputStream());
 StrDatiInSocket = new DataInputStream
   (rcpSocketSrv.getInputStream());
 NomeFile = StrDatiInSocket.readUTF();

 FileDaScrivere = new File(NomeFile);

 if(FileDaScrivere.exists () == true)
 { StrDatiOutSocket.writeUTF(
     "MSGSrv: file presente, bye");
 }

/* in caso le cose siano chiuse /*
```

```
else
/* in caso si consenta di trasmettere
il file /*
/* scrittura effettiva del file */
{
  StrDatiOutSocket.writeUTF(
      "MSGSrv: attendofile");
  byte ContentFile [] = new byte [1000];
   StrDatiInSocket.read(ContenutoFile);
  FileDaScrivereOutputStream = new
   FileOutputStream (FileDaScrivere);
  FileDaScrivereOutputStream.write
   (ContenutoFile);
}

/* chiusura della connessione e
terminazione del servitore specifico
*/
  rcpSocketServizio.close();

  System.out.println(
   "Fine servizio thread numero " +
   Thread.currentThread());

}
catch (IOException e)
 { System.err.println(e);}
...
```