



**Università degli Studi di Bologna
Facoltà di Ingegneria**

**Corso di
Reti di Calcolatori L-A**

Progetto C/S con Socket in Java

Antonio Corradi

Anno accademico 2009/2010

SOCKET per COMUNICAZIONE

**Necessità di Standard, in particolare tra macchine
distinte, diverse, fortemente eterogenee**

Useremo le socket per la comunicazione standard

Programmazione di rete in Java attraverso alcuni **meccanismi** di **visibilità della comunicazione** (*sul sistema operativo*)

Contenuti in classi specifiche del package di networking `java.net`

**le socket rappresentano il terminale locale (end point) di un
canale di comunicazione bidirezionale (da e per l'esterno)**

Un Client e un Server su macchine diverse possono comunicare sfruttando **due diversi tipi** di modalità di comunicazione che permettono una qualità e un costo diverso associato

Socket nascono in ambiente Unix BSD 4.2

TIPI di COMUNICAZIONE

- **con connessione**, in cui viene stabilita una connessione tra Client e Server (esempio, il sistema telefonico) socket **STREAM**
- **senza connessione**, in cui non c'è connessione e i messaggi vengono recapitati uno indipendentemente dall'altro (esempio, il sistema postale) socket **DATAGRAM**

classi per SOCKET INTERNET

con connessione usando il protocollo Internet TCP

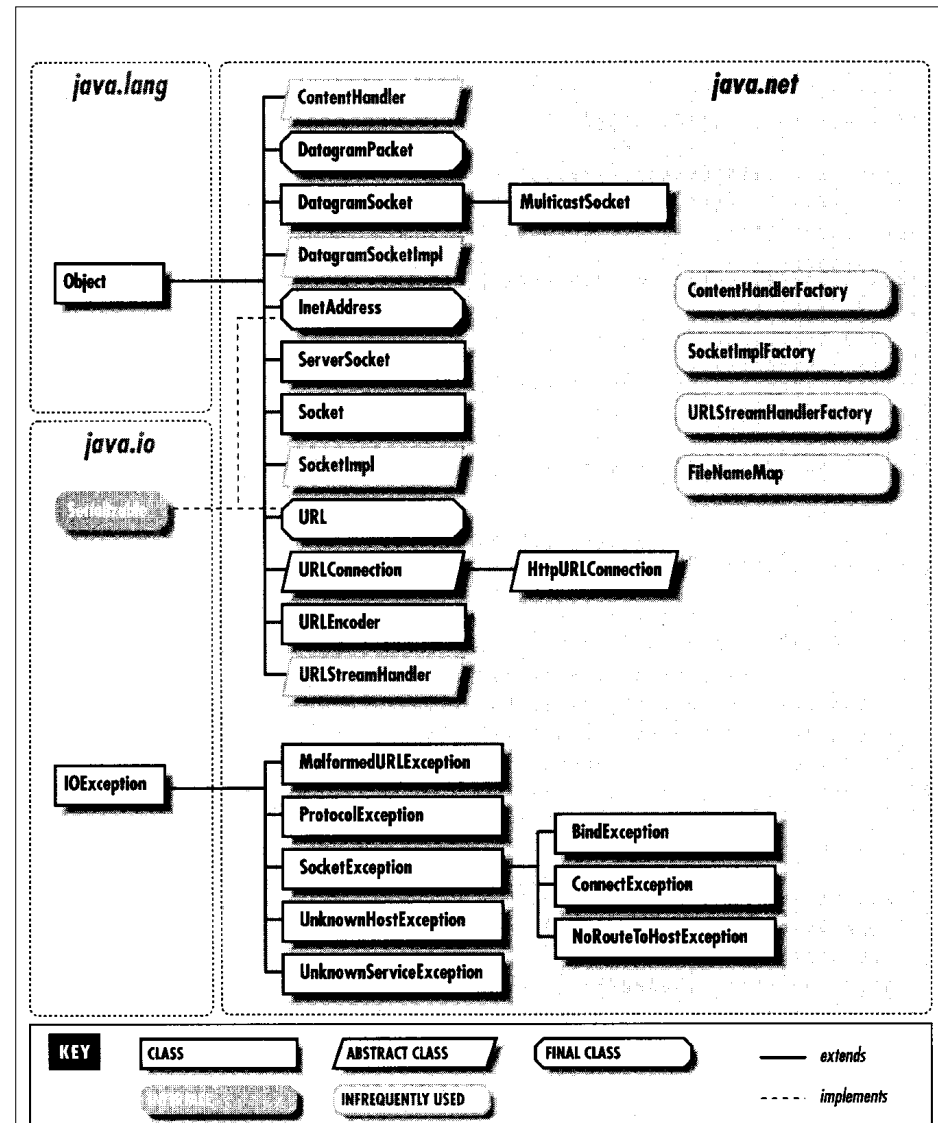
- classe **Socket**, per socket lato Client,
 - classe **ServerSocket**, per socket lato Server,
- senza connessione usando il protocollo Internet UDP
- classe **DatagramSocket**, per socket (C/S)

Gerarchia delle classi in Java

Package Java.net

Le classi e le interfacce sono state significativamente estese dalle diverse versioni della Java Virtual Machine JVM (fino alla versione 1.6 ...)

La filosofia e la struttura dei meccanismi rimane la stessa



SISTEMA di NOMI

Necessità di un **sistema di identificazione** degli enti 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 reciproca dei processi** (il Client o il Server) nella rete

Ogni processo deve essere associato ad un **NOME GLOBALE**
visibile in modo univoco, non ambiguo, e semplice

“nome” della macchina + “nome” del processo nel nodo

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

Il problema è risolto dai livelli bassi di protocollo (trasporto e rete)
per le socket in Internet, i nomi di trasporto (TCP, UDP) e rete (IP)

NOMI per le SOCKET

Un nodo è identificato univocamente da

- **indirizzo IP** (4 byte / 32 bit) ⇒ livello IP
- **porta** (numero intero di 16 bit) ⇒ astrazione in TCP e UDP

Uso di questo **NOME GLOBALE**

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

Per raggiungere una **risorsa con NOME LOCALE**

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

Con questo **doppio sistema di nomi**, è possibile identificare un processo senza conoscere il suo process identifier (pid) *locale*

(IP livello network - OSI 3, PORTA livello trasporto - OSI 4)

Un indirizzo IP e una porta rappresentano l'endpoint di un canale di comunicazione

NOMI GLOBALI per SOCKET

Numeri IP ⇒ indirizzo IP: ad es. `137.204.57.186`

Numeri di Porta ⇒ porte 4 cifre hex: `xxxxh` (dec. `1 - 65535`)
esprese spesso con unico decimale ad es. `153`, `2054`

Funzione fondamentale delle porte è identificare un servizio

I numeri di porta minori di 1024 sono riservati (well-known port)
i servizi offerti dai processi legati a tale porta standardizzati

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 per porte del protocollo connesso TCP, parte server

porta `21` per ftp,

porta `23` per telnet,

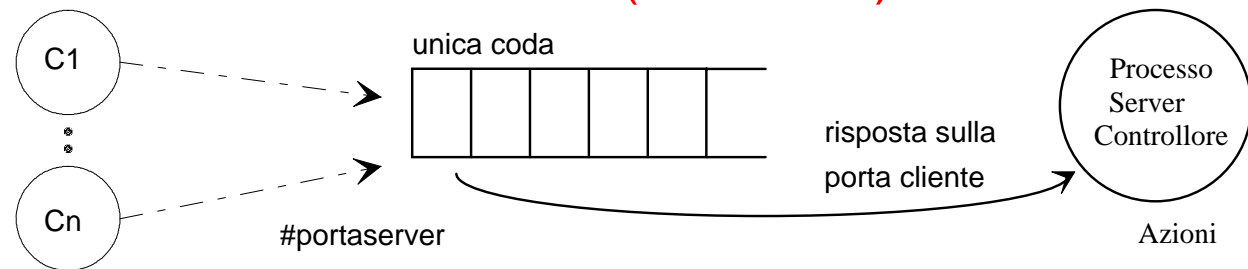
porta `25` per mail,...

SERVER SEQUENZIALI

Server per una richiesta alla volta (con connessione o meno)

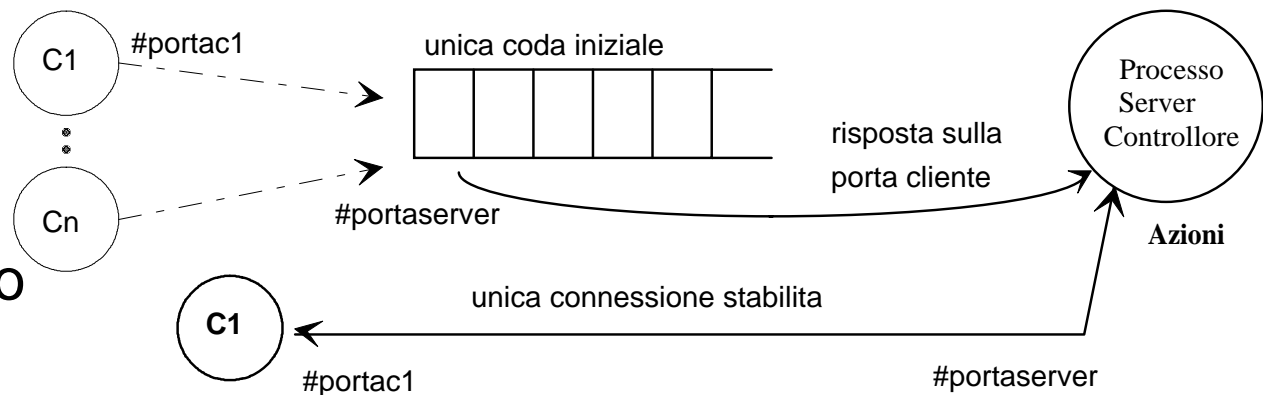
server sequenziale senza connessione (uso UDP)

servizi senza stato
e poco soggetti
a guasti



server sequenziale con connessione (uso TCP)

servizi
con reliability
limitando lo stato
overhead per controllo
della connessione



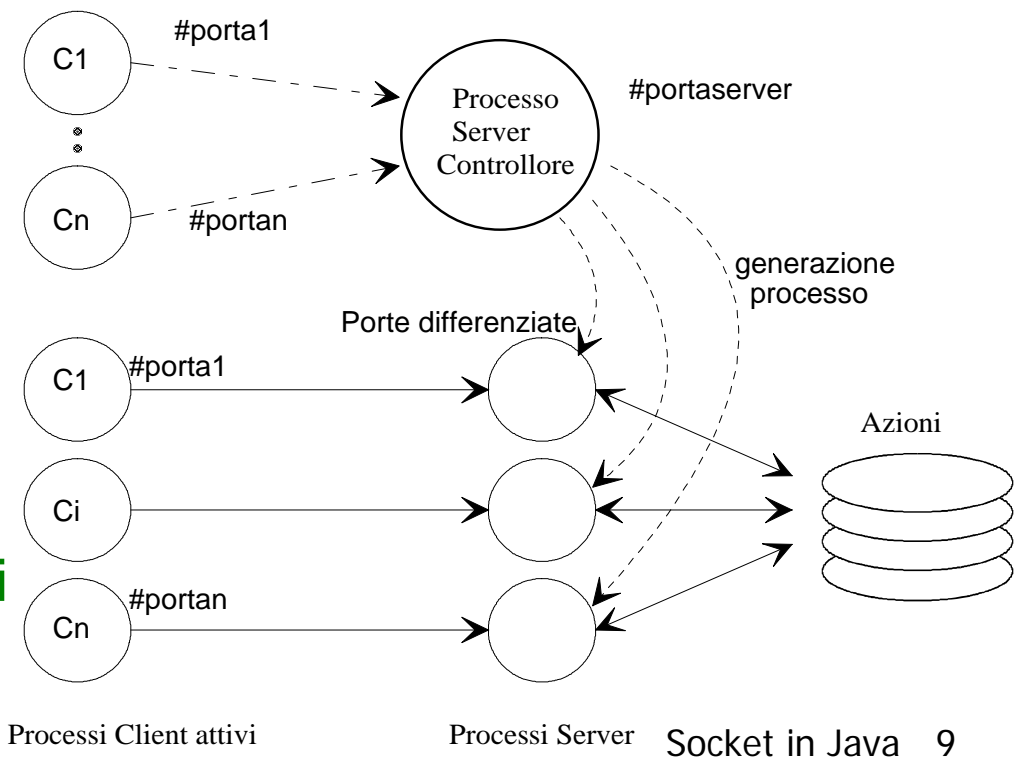
SERVER PARALLELI

Server concorrente con più richieste alla volta (multiprocesso)

Uso di processi multipli, con un **master server** che genera **processi interni** per ogni servizio

Si deve garantire che il costo di generazione del processo non ecceda il guadagno ottenuto

Soluzione con processi creati in precedenza per essere velocemente smistati al servizio necessario
numero fissato di processi iniziali e altri creati su necessità e mantenuti per un certo intervallo

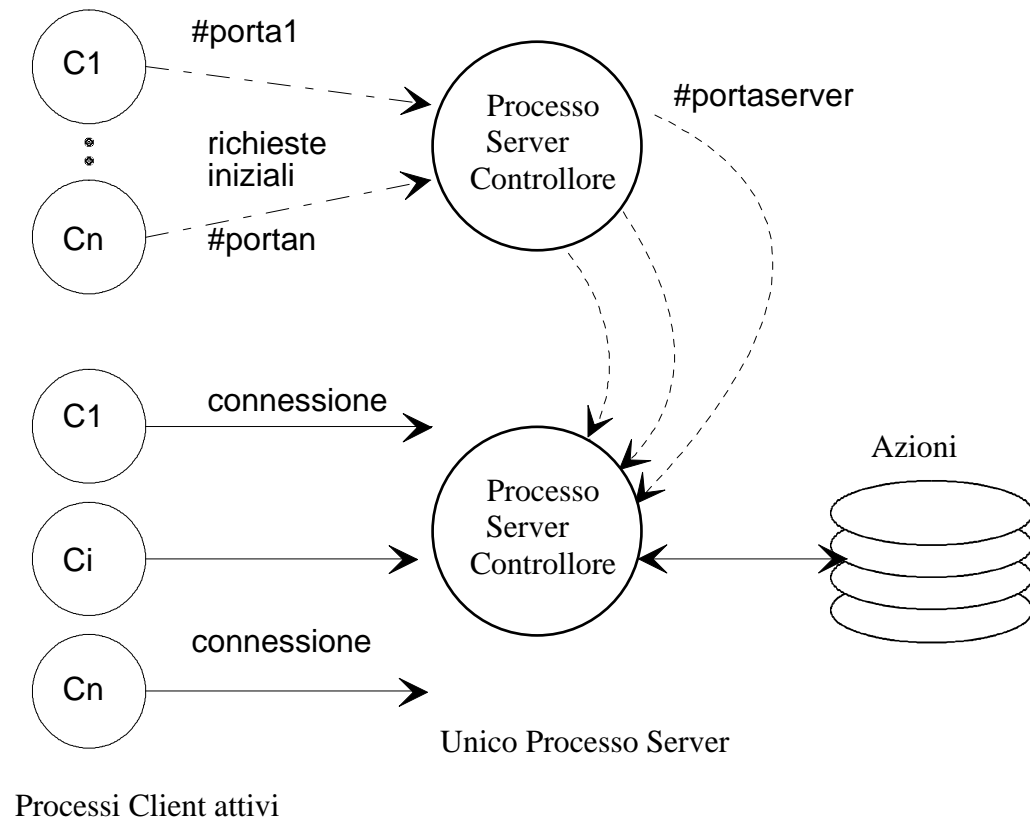


SERVER CONCORRENTE

Server concorrente con più richieste alla volta (monoprocesso)

In Java di difficile realizzazione, con un solo processo capace di portare avanti più servizi

Soluzione con connessione
ma in cui un **processo server unico** è capace di servire molte richieste **concorrentemente**



SOCKET DATAGRAM

Le socket DATAGRAM permettono a due thread di scambiarsi messaggi senza stabilire una connessione tra i thread coinvolti

meccanismo di comunicazione non è affidabile con possibili perdite di messaggi (per problemi di rete) e consegna non in ordine

a causa del protocollo UDP

un solo tipo di socket DATAGRAM sia Client sia Server

La classe `java.net.DatagramSocket`

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

Uno dei Costruttori prevede (*ce ne sono altri*)

```
DatagramSocket( InetAddress localAddress,  
    int localPort) throws SocketException; /* anche altri costruttori */
```

il costruttore `DatagramSocket` crea socket UDP e fa un binding locale a una specificata porta e nome IP: socket pronta

SOCKET DATAGRAM

SCAMBIO MESSAGGI con socket usando meccanismi primitivi di comunicazione, **send** e **receive** di pacchetti utente

Su una istanza di **sock** di **DatagramSocket** si fanno azioni di

```
void send(DatagramPacket p);
```

```
void receive(DatagramPacket p);
```

Le due primitive sono reali **operazioni di comunicazione**, *la prima invia un messaggio (datagramma), la seconda aspetta fino a ricevere il primo datagramma disponibile*

La **send**, come tutti gli invii, può sottendere solo la consegna ad un livello di kernel che si occupa dell'invio stesso (**asincrona con la ricezione**)

La **receive** invece, che pure assume che la vera ricezione sia delegata al kernel, richiede una attesa del ricevente fino all'arrivo della informazione (**semantica sincrona con il ricevente**)

basta ricevere un datagramma per sbloccare una receive

SEND e RECEIVE su SOCKET DATAGRAM

Le primitive send e receive, oltre a richiedere una socket correttamente inizializzata, usano sempre delle struttura di appoggio

```
sock.send(DatagramPacket p);
```

```
sock.receive(DatagramPacket p);
```

che servono in input per receive e in output per invio

Si usano molte **classi accessorie di supporto**

come i `DatagramPacket` e altre costanti

Ad esempio si devono usare gli `interi` per le porte, si devono usare `costanti` opportune per i nomi di IP, intesi come nomi interi IP e `string` come nomi di dominio

InetAddress

Oltre che eccezioni relative alla comunicazione nei costruttori

`SocketException`, `SecurityException`, ...

MODELLO di COMUNICAZIONE

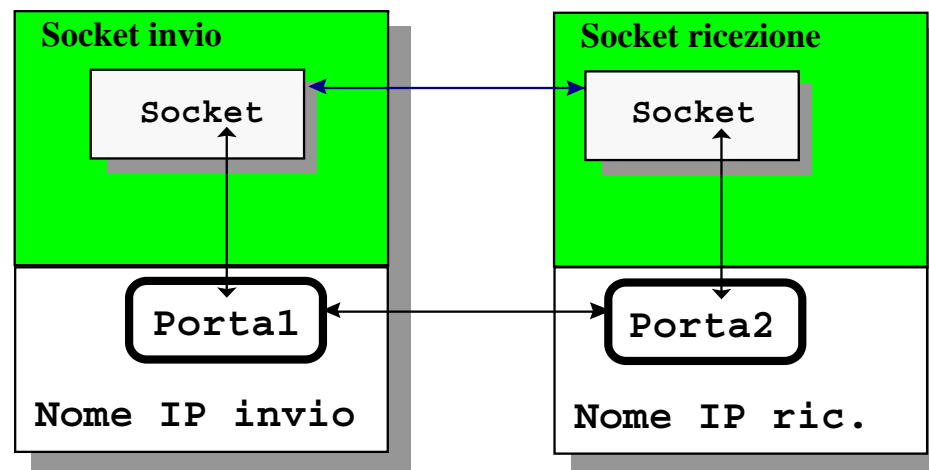
Le socket datagram per scambio di messaggi devono essere state inizializzate correttamente (create) e devono conoscersi

il mittente deve specificare nel messaggio un ricevente

Si devono specificare informazioni di tipo

Applicativo il messaggio (o lo spazio per il messaggio)

di Controllo il nodo e la porta associata alla socket del ricevente



Nessuna garanzia a causa del protocollo di supporto (UDP e IP)

CLASSI ACCESSORIE

Classe DatagramPacket

Classe per preparare e usare datagrammi che specificano cosa comunicare (**parte dati**) e con chi (**parte controllo**)

Parte dati ⇒ specifica un **array di byte** da/su cui scrivere e con indicazioni di comunicazione con diversi costruttori

Parte controllo ⇒ Interi per la porta e **InetAddress**

InetAddress classe per gli indirizzi IP solo metodi pubblici statici

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

fornisce un oggetto InetAddress per l'host specificato (null def. locale)

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

fornisce un array di oggetti InetAddress per più indirizzi IP sullo stesso nome logico

```
public static InetAddress getLocalHost();
```

fornisce InetAddress per macchina locale

PARTE DATI per DATAGRAMPACKET

Il **DatagramPacket** deve contenere la parte applicativa utente

```
DatagramPacket( byte [] buf, // array di byte dati
               int offset,    // indirizzo inizio
               int length,    // lunghezza dati
               InetAddress address, int port); // numero IP e porta
```

con molti altri costruttori e molte funzioni di utilità come

<code>InetAddress getAddress(),</code>	ottiene indirizzo associato
<code>void setAddress(InetAddress addr)</code>	cambia indirizzo
<code>int getPort(),</code>	ottiene porta associata
<code>void setPort(int port)</code>	cambia porta associata
<code>byte[] getData(),</code>	estrae i dati dal pacchetto
<code>void setData(byte[] buf), ...</code>	inserisce i dati nel pacchetto

DATAGRAMPACKET

Il **DatagramPacket** è un contenitore unico di aiuto all'utente
si deve considerare la operatività a secondo della funzione che
stiamo invocando:

sock.send (DatagramPacketp)

- in invio dobbiamo avere preparato una area su cui l'utente possa mettere i dati e l'area per accogliere le informazioni di controllo sul ricevente (fornite dal mittente del pacchetto)

Solo dopo averlo fatto facciamo l'invio...

sock.receive (DatagramPacketp)

- in ricezione dobbiamo avere preparato tutto per ricevere tutte le informazioni, sia per la parte dati, sia la parte di controllo

Solo dopo la ricezione, possiamo lavorare sul pacchetto ed estrarre le informazioni che ci servono

Un pacchetto potrà essere riutilizzato a piacere...

SCHEMA DI COMUNICAZIONE

Creazione socket

```
DatagramSocket socket = new DatagramSocket();
```

Parte mittente di invio...

Preparazione informazione da inviare e invio

```
byte[] buf = {'C','i','a','o'};
```

```
InetAddress addr = InetAddress.getByName("137.204.59.72");
```

```
int port = 1900;
```

```
DatagramPacket packet = new
```

```
    DatagramPacket(buf, buf.length, addr, port);
```

```
socket.send(packet); // invio immediato
```

Altre operazioni di invio oppure anche ricezioni...

COMUNICAZIONE: RICEZIONE

Creazione socket

```
DatagramSocket socket = new DatagramSocket();
```

Parte ricevente di comunicazione: Preparazione, attesa, e ricezione

```
int  report;  InetAddress recaddress;
```

```
byte[] res = new byte[200];
```

```
DatagramPacket packet = new
```

```
    DatagramPacket(res, res.length, recaddress, report);
```

```
packet.setData(res);    // riaggancio della struttura dati
```

```
socket.receive(packet); // ricezione con attesa sincrona
```

```
// estrazione delle informazione dal datagramma
```

```
report = packet.getPort();
```

```
recaddress = packet.getAddress();
```

```
res = packet.getData();
```

```
// uso delle informazioni ...
```

COMUNICAZIONE MULTICAST

Sono possibili anche **comunicazioni Multicast (non punto punto)**
SOCKET MULTICAST con una ulteriore classe per ricevere
messaggi multicast e gestire gruppi di multicast

```
MulticastSocket(int multicastport);
```

Socket legate a indirizzi di gruppo di **classe D** attraverso un **gruppo di multicast** su cui ricevere messaggi

Preparazione 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 (per ricevere)

```
// unisciti al gruppo ... e esci dal gruppo
```

```
s.joinGroup(gruppo);
```

```
s.leaveGroup(gruppo);
```

```
// tenere conto della porta per selezionare dipende dal sistema operativo
```

COMUNICAZIONE MULTICAST

Uso della socket multicast per inviare (quasi gratis ☺)

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

Uso della socket multicast per ricevere

```
// ottenere i messaggi inviati  
byte[] buf = new byte[1000];  
DatagramPacket recv =  
    new DatagramPacket(buf, buf.length);  
s.receive(recv);  
...
```

Si riceve nell'ambito della sessione di sottoscrizione

OPZIONI SOCKET

Tutti gli strumenti hanno un **comportamento chiaro e preciso** che potrebbe **non essere adatto in alcuni casi ...**

Opzioni delle Socket servono a cambiare il comportamento
la ricezione da socket (es., receive()) è sincrona bloccante

SetSoTimeout (int timeout) throws ...

Questa opzione definisce un **timeout** in msec, dopo il quale **la operazione termina** (e viene lanciata una eccezione da gestire)

Se timeout a zero, nessuna sospensione

SetSendBufferSize (int size) throws ...

SetReceiveBufferSize (int size) throws ...

il buffer di invio e ricezione della driver può essere variato

SetReuseAddress ()

si possono collegare più processi ad un certo indirizzo fisico

Sono previste le get corrispondenti

SOCKET A STREAM

Le socket STREAM sono i **terminali** di un **canale di comunicazione virtuale, creato prima della comunicazione**

semantica at-most once

La **comunicazione** punto-a-punto tra il Client e il Server avviene in **modo bidirezionale, affidabile, con dati (byte) consegnati in sequenza una sola volta** (modalità FIFO come sulle pipe di Unix)

se non consegnati??? Si può dire poco...

La connessione tra i processi Client e Server è definita non dai processi ma **da una quadrupla univoca e dal protocollo**

<indirizzo IP Client; porta Client; indirizzo IP Server; porta Server>

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

- TCP protocollo di trasporto, livello 4 OSI e fornisce l'astrazione porta
- IP è protocollo di rete, livello 3 OSI, per ogni identificazione di nodo

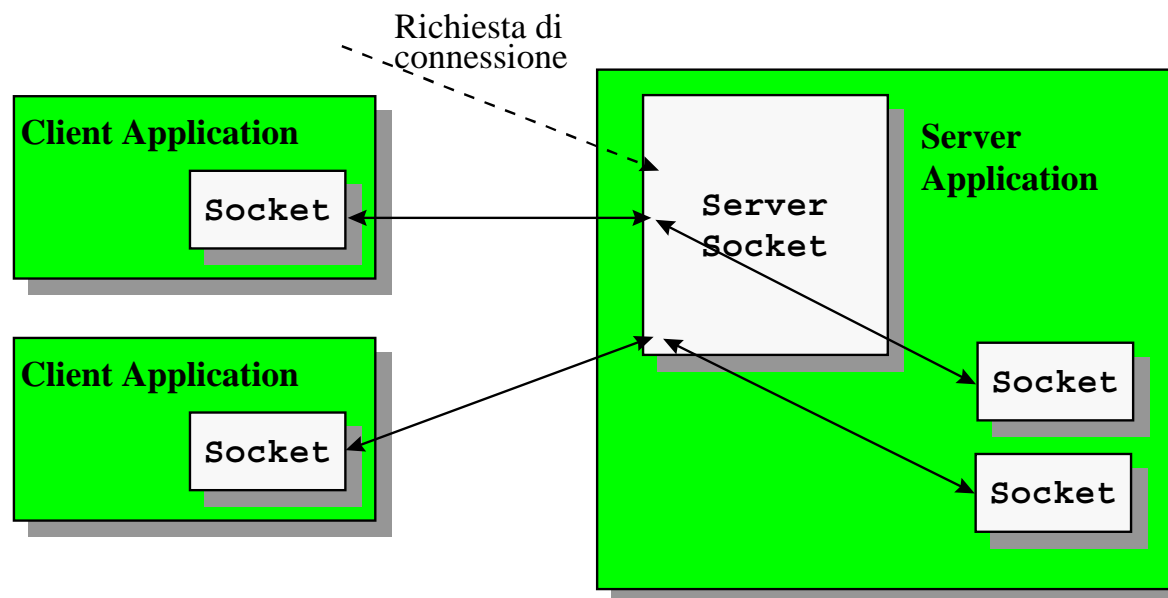
La comunicazione tra Client e Server su stream segue uno schema asimmetrico e il principio della connessione (relative API diverse)

SOCKET A STREAM

Java ha portato a due tipi di socket distinti per i ruoli distinti, una per il Client/Server e una per il solo Server, e quindi

Classi distinte per ruoli Cliente e Servitore

Le classi `java.net.Socket` e `java.net.ServerSocket`



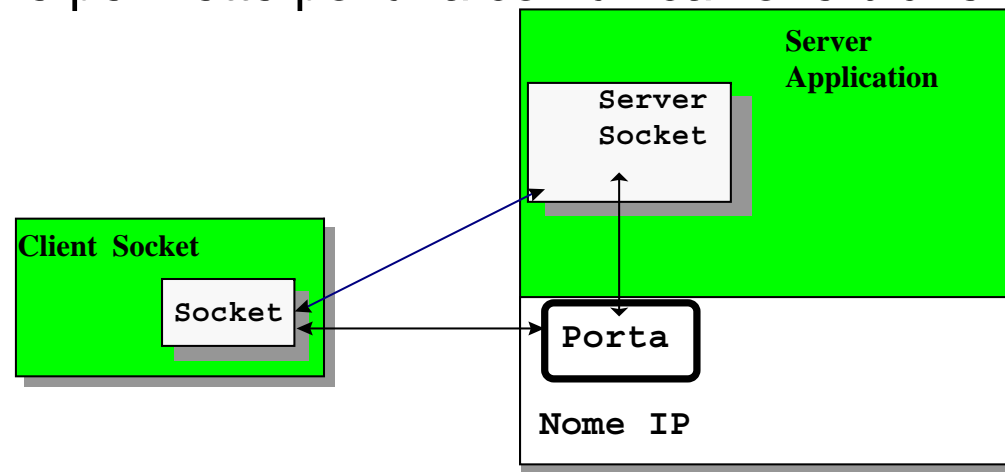
Altro principio, **dove possibile, si nascondono i dettagli realizzativi dei protocolli, ad esempio nei costruttori delle classi**

SOCKET A STREAM: CLIENTE

La classe **Socket** consente di creare una socket “attiva” connessa stream (TCP) per il collegamento di un Client a un Server

I costruttori della classe creano la socket, la legano a una porta locale e ... la connettono a una porta di una macchina remota su cui sta il server

La connessione permette poi una comunicazione bidirezionale (full duplex)



Si noti che la creazione della socket produce in modo atomico anche la connessione al server corrispondente (deve essere presente)

(Unix API più complesse e complete: vedi socket, bind, connect)

STREAM CLIENTE: COSTRUTTORI

```
public Socket(InetAddress remoteHost, int remotePort)  
    throws IOException;
```

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 remoteHost

```
public Socket(InetAddress remoteHost, int remotePort,  
InetAddress localhost, int localPort) throws IOException;
```

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

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

STREAM CLIENTE: GESTIONE

APERTURA ottenuta con il costruttore in modo implicito

la **creazione con successo** di una **socket** a stream produce una **connessione bidirezionale a byte** (stream) tra i due processi interagenti e impegna risorse sui nodi e tra i processi

CHIUSURA come operazione necessaria per non impegnare troppe risorse di sistema

Le risorse sono le **connessioni**: costa definirle e crearle, così si devono gestirle al meglio, mantenerle e distruggerle

Si tendono a mantenere le sole connessioni necessarie e a limitare alle aperture contemporanee di sessioni chiudendo quelle non più utilizzate

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

```
public synchronized void close() throws SocketException;
```

STREAM CLIENTE: SUPPORTO

Per informazioni sulle socket si possono utilizzare i metodi aggiuntivi

```
public InetAddress getInetAddress(); // remote
```

restituisce l'indirizzo del nodo remoto a cui socket è connessa

```
public InetAddress getLocalAddress(); // local
```

restituisce l'indirizzo della macchina locale

```
public int getPort(); // remote port
```

restituisce il numero di porta sul nodo remoto cui socket è connessa

```
public int getLocalPort(); // local
```

restituisce il numero di porta locale a cui la socket è legata

Esempio:

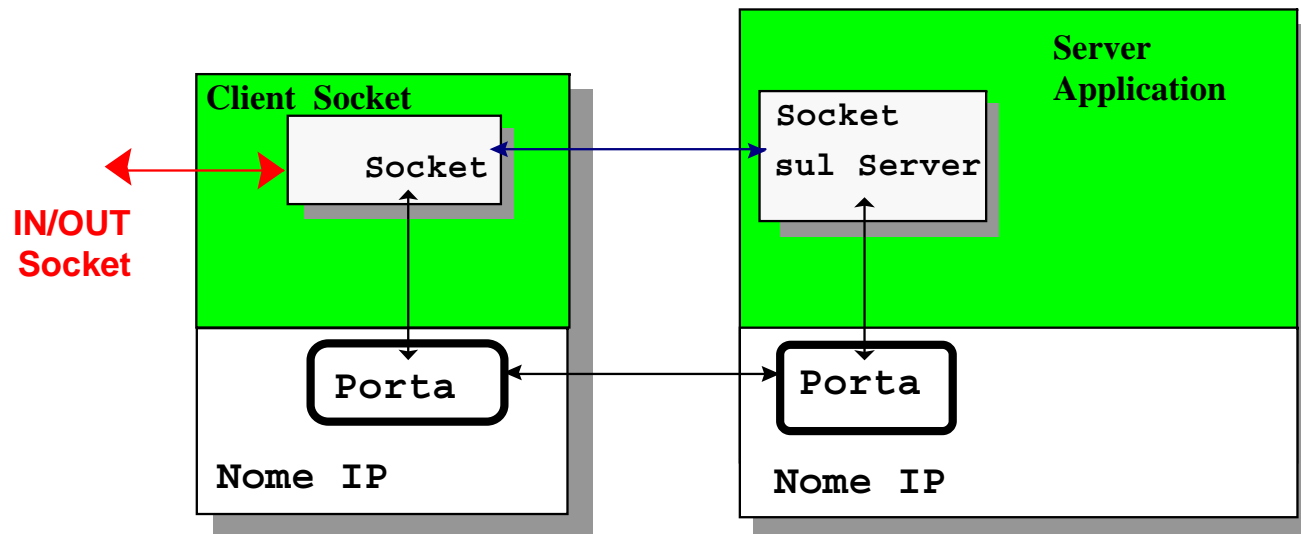
```
int porta = oggettoSocket.getPort();
```

Si possono ottenere dinamicamente (runtime) informazioni sulle connessioni correnti delle socket usate

STREAM: RISORSE di SUPPORTO

Non dimentichiamo che abbiamo sempre Cliente e Servitore, se si è creata la socket, e che ci sono risorse impegnate in gioco ...

A questo punto si deve comunicare ...
pensiamo di **leggere e scrivere** dalla socket cliente (**IN/OUT**)



STREAM di COMUNICAZIONE JAVA

Lettura o scrittura da/su una socket dopo avere qualificato le risorse stream della socket come Java stream

```
public InputStream getInputStream()
```

```
public OutputStream getOutputStream()
```

I due metodi restituiscono un **oggetto stream** che incapsula il canale di comunicazione (di classi `InputStream` e `OutputStream`)

Attraverso gli stream si possono spedire/ricevere solo **byte**,
senza nessuna formattazione in messaggi (vedi classi)

Naturalmente, i byte arrivano **ordinati e non duplicati** (non possono arrivare byte successivi, senza che arrivino i precedenti); i dati arrivano al più una volta (**at-most-once**)

e in caso di errore? nessuna conoscenza

Altri oggetti stream Java possono incapsulare gli stream socket, per fornire funzionalità di più alto livello (ad es., `DataInputStream`)

DataOutputStream e DataInputStream

DataOutputStream e DataInputStream offrono una serie di metodi per l'invio e la ricezione di **tipi primitivi Java**

Uso tipico: realizzazione di **protocolli** fra Client e Server **scritti in linguaggio Java** (con scambio di oggetti Java): **nel corso** vengono usati per la **realizzazione di applicazioni C/S in Java**

Ad esempio:

	DataOutputStream	DataInputStream
String	<code>void writeUTF(String str)</code>	<code>String readUTF()</code>
char	<code>void writeChar(int v)</code>	<code>char readChar()</code>
int	<code>void writeInt(int v)</code>	<code>int readInt()</code>
float	<code>void writeFloat(float v)</code>	<code>float readFloat()</code>
...	<code>...</code>	<code>...</code>

UTF Unified Transformation Format

ESEMPIO CLIENTE STREAM

Client di echo (il Server Unix è sulla *porta nota 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.getInputStream()));
userInput = new BufferedReader
    (new InputStreamReader(System.in)); /* ciclo lettura fino a fine file */
while((oggLine = userInput.readLine()) != null)
{out.println(oggLine); System.out.println(in.readLine());}
oggSocket.close();
} // fine try
catch (IOException e) { System.err.println(e);} ...
```

Per ogni ciclo si legge da standard input, si scrive sulla socket out e si attende da socket in la risposta di echo

STREAM SERVER: ARCHITETTURA

Il lato server prevede, dalla classe `java.net.ServerSocket`, una `ServerSocket` che definisce una socket capace solo di accettare richieste di connessione provenienti da diversi Client

- più richieste di connessione pendenti allo stesso tempo e
- più connessioni aperte contemporaneamente

Si deve definire anche la **lunghezza della coda** in cui vengono messe le richieste di connessione non ancora accettate dal server

Al momento della creazione si effettuano implicitamente le operazioni più elementari visibili in UNIX, come `socket`, `bind` e `listen`

La connessione richiede di essere **stabilita su iniziativa del server** (ottenuta tramite primitiva di comunicazione **accept**)

Obiettivo della `accept`, lato server, è restituire un normale oggetto **Socket** nel server (*restituito dalla accept*) per la specifica connessione e trasmissione dati

SERVERSOCKET: COSTRUTTORI

Sulle socket dalla parte server

```
public ServerSocket(int localPort)
```

```
    throws IOException, BindException;
```

crea una socket in ascolto sulla porta specificata

```
public ServerSocket(int localPort,int count)
```

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

Il server gioca un ruolo "passivo": deve attivare la coda delle possibili richieste ed aspettare i clienti

Il server comincia a decidere con la introduzione volontaria della primitiva di accettazione esplicita

Le richieste accodate non sono servite automaticamente ed è necessaria una API che esprima la volontà di servizio

SERVERSOCKET: ACCEPT

Il Server si deve **mettere in attesa di nuove richieste di connessione** chiamando la primitiva **accept()**

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

La invocazione di **accept** **blocca il Server** fino all'arrivo di almeno una richiesta di connessione

La **accept** restituisce un oggetto della classe **Socket** su cui avviene la comunicazione vera di byte tra Client e Server

Quando arriva una richiesta, la **accept crea una nuova socket per la connessione di trasporto già creata con il Client: la nuova Socket** restituito da **accept** rappresenta lo stream reale con il cliente

la chiamata di **accept** è **sospensiva**, in attesa di richieste di connessione

- Se non ci sono ulteriori richieste, il servitore si blocca in attesa
- Se c'è almeno una richiesta, si sblocca la primitiva e si crea la connessione per questa (la richiesta è consumata)

STREAM SERVER: SUPPORTO

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** (come nel protocollo TCP)

Informazioni sulle socket connesse come nel cliente:

```
public InetAddress getInetAddress(); // remote
```

restituisce l'indirizzo del nodo remoto a cui socket è connessa

```
public InetAddress getLocalAddress(); // local
```

restituisce l'indirizzo della macchina locale

```
public int getPort(); // remote port
```

restituisce il numero di porta sul nodo remoto cui socket è connessa

```
public int getLocalPort(); // local
```

restituisce il numero di porta locale a cui la socket è legata

ESEMPIO SERVER STREAM

Server daytime (il Server Unix su porta 13)

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

Ad ogni cliente il server sequenziale manda la data e chiude tutto

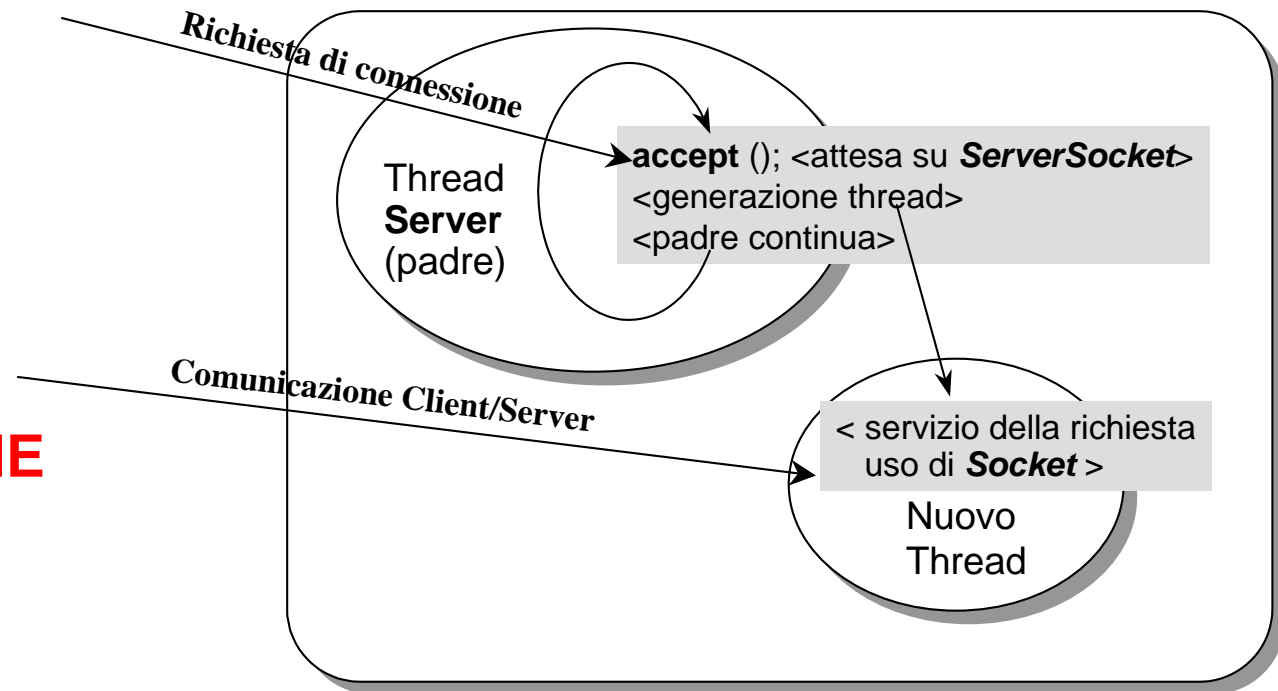
SERVER PARALLELO

In caso di **server parallelo**, ...

Alla accettazione il servitore **può generare una nuova attività responsabile del servizio** (che eredita la connessione nuova) e la chiude al termine dell'operazione

Il servitore principale può tornare immediatamente ad aspettare nuove richieste e servire nuove operazioni

**SERVER
PARALLELO
MULTIPROCESSO
con CONNESSIONE**



Esempio PROTOCOLLO C/S STREAM

Remote CoPy (RCP) ossia un'applicazione distribuita Client/Server per eseguire la copia remota (remote copy, rcp) di file da C a S

Progettare sia il programma client, sia il programma server.

Il programma Client deve consentire la invocazione:

rcp_client nodoserver portaserver nomefile nomefiledest

nodoserver e **portaserver** indicano il **Server** e **nomefile** è il nome 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 **nomefiledest**

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

Uso di connessione: il file richiede di trasferire anche grosse moli di dati e con i byte tutti in ordine e una volta sola

La connessione aperta dal cliente consente al server di coordinarsi per la richiesta del file che viene inviato (**trasmissione dati e coordinamento**)

! Versione non ottimizzata da migliorare !

Esempio: RCP parte CLIENT

RCP Client - Estratto del client ... UTF formato standard

```
rcpSocket = new Socket(host,porta);  
OutSocket = new DataOutputStream(rcpSocket.getOutputStream());  
InSocket = new DataInputStream(rcpSocket.getInputStream());  
OutSocket.writeUTF (NomeFileDest);  
Risposta = InSocket.readUTF(); System.out.println(Risposta);  
if (Risposta.equalsIgnoreCase("MSGSrv: attendofile") == true)  
{FDaSpedDescr = new File(NomeFile); // ottiene file locale in memoria  
  FDaSpedInStream = new FileInputStream(FDaSpedDescr);  
  byte ContenutoFile [] = new byte[FDaSpedInStream.available()];  
  FDaSpedInStream.read(ContenutoFile); // file come array di byte  
  OutSocket.write (ContenutoFile); // file scritto in un colpo?  
}  
}} catch(IOException e) ... {}  
rcpSocket.close(); ...
```

UTF Unicode Transformation Format per maggiore portabilità

RCP parte SERVER SEQUENZIALE

```
... try {  
    rcpSocketSrv = new ServerSocket(Porta);  
    System.out.println("Attesa su porta"+rcpSocketServer.getLocalPort());  
    while(true){ // ciclo: per ogni cliente si attua una connessione  
        SocketConn = rcpSocketSrv.accept( ); System.out.println("con"+Socketconn);  
        OutSocket = new DataOutputStream(SocketConn.getOutputStream( ));  
        InSocket = new DataInputStream(SocketConn.getInputStream( ));  
        NFile = InSocket.readUTF ( ); FileDaScrivere = new File(NFile);  
        if(FileDaScrivere.exists() == true)  
        { OutSocket.writeUTF("MSGSrv: file presente, bye"); } else {  
            OutSocket.writeUTF("MSGSrv: attendofile");  
            byte ContntFile[] = new byte [1000]; InSocket.read(ContntFile);  
            FileOutputStream = new FileOutputStream (FileDaScrivere);  
            FileOutputStream.write(ContntFile); // file letto tutto nel buffer ?  
        } SocketConn.close( ); } }  
catch (IOException e) {System.err.println(e);} ...
```

RCP: SERVER PARALLELO

```
... try {...  
rcpSocket = new ServerSocket(Porta);  
System.out.println("Attesa su porta"+rcpSocket.getLocalPort());  
while(true)  
{ rcpSocketConn = rcpSocket.accept();  
  threadServizio = new rcp_servizio (rcpSocketConn);  
  threadServizio.start();  
}  
catch (IOException e) {System.err.println(e);} }
```

Si genera un nuovo processo per ogni **connessione accettata** e sullo stream avviene la interazione

Una **socket usata** è visibile da tutti i thread (condivisione risorse in Java): la prima **close** chiude la socket definitivamente per tutti i thread, fino ad allora impegna risorse di supporto

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

RCP: parte SERVER THREAD / 1

```
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
        {
            OutSocket= new DataOutputStream
                (rcpSocketSrv.getOutputStream());

            InSocket = new DataInputStream(rcpSocketSrv.getInputStream());
            NomeFile = InSocket.readUTF();
            FileDaScrivere = new File(NomeFile);
            if(FileDaScrivere.exists () == true)
                /* in caso le cose siano terminate senza invio */
            { OutSocket.writeUTF( "MSGsrv: file presente, bye");
            }
        }
    }
}
```

RCP: parte SERVER THREAD / 2

```
// Nel caso di reale trasmissione del file
else /* solo in caso di scrittura effettiva del file */
{ OutSocket.writeUTF("MSGsrv: attendofile");
  byte ContentFile [] = new byte [1000];
  InSocket.read(ContentFile);
  DaScrivereStream = new FileOutputStream (FileDaScrivere);
  DaScrivereStream.write(ContentFile);
}
/* chiusura della connessione e terminazione del servitore specifico */
rcpSocketSrv.close();
System.out.println("Fine servizio thread numero " +
                  Thread.currentThread());
}
catch (IOException e)
{ System.err.println(e); exit(1);} ...
```

CHIUSURA SOCKET

Le socket in Java impegnano non solo il loro livello applicativo, ma anche una serie di risorse di sistema che sono collegate e necessarie per la operatività fino alla `socket.close()`

La chiusura è quindi necessaria sempre per dichiarare al sistema la non necessità di mantenere risorse non più in uso

In caso di una socket chiusa, le risorse sono mantenute per un certo tempo (in dipendenza dalla importanza delle operazioni e non memoria in)

In caso di socket connessa chiusa, la memoria di out viene mantenuta per continuare a spedire informazioni da inviare al pari

Il pari si accorge di questo tramite sia **eccezioni o predicati** sia **eventi** che gli vengono notificati in caso di operazioni (lettura o scrittura sulla socket chiusa dal pari produce eventi significativi)

La chiusura quindi è fatta su iniziativa di uno dei processi affacciati quando vuole ed ha impatto anche sull'altro

CHIUSURA SOCKET ... DOLCE

La chiusura rappresenta una chiusura di una connessione in entrambi i versi fatta da uno dei due connessi

In caso di una connessione, ognuno dei due partecipanti è più responsabile di solo uno dei versi: della sua uscita sulla connessione e dipende dall'altro per la lettura

Si hanno anche primitive differenziate per ragionare sulla chiusura per un verso solo, `shutdownInput()` e `shutdownOutput()`;

La primitiva più usata per chiusure responsabili è la `shutdownOutput()` per cui si chiude la direzione di responsabilità

In caso di socket **connessa in shutdown**, la memoria di out viene mantenuta per spedire **informazioni al pari**, la in è soggetta all'altro

Si vedano alcune funzioni come:

<code>isClosed()</code> ;	<code>isConnected()</code> ;
<code>isShutdownInput()</code> ;	<code>isShutdownOutput()</code> ;

OPZIONI SOCKET

Opzioni delle Socket per cambiare il comportamento

Si esplorino le opzioni delle socket in Java con funzioni definite per socket stream

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 dopo un **intervallo di linger in sec**

SetTcpNoDelay (boolean on) throws ...

il pacchetto è **inviato immediatamente, senza bufferizzare**

SetKeepAlive (boolean on) throws...

abilita, disabilita la opzione di **keepalive**

**le opzioni sono disponibili nella interfaccia SocketOptions
che prevede anche tutte le get corrispondenti**