



Università degli Studi di Bologna
Scuola di Ingegneria

Corso di Reti di Calcolatori T

Progetto C/S con Socket in Java

Antonio Corradi

Anno accademico 2020/2021

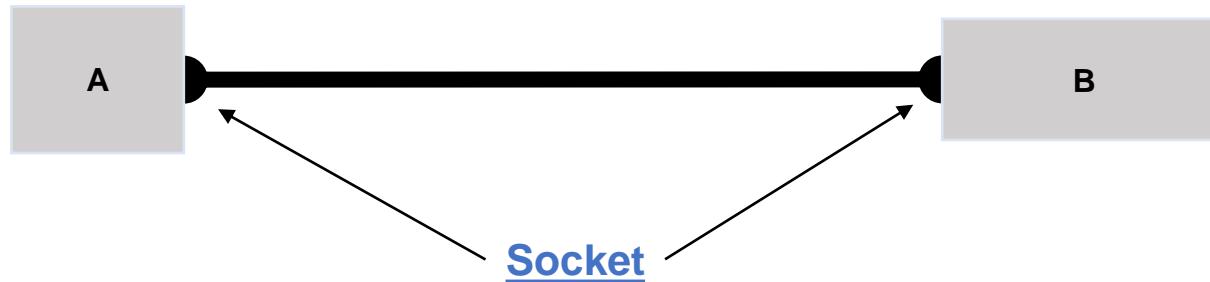
SOCKET PER COMUNICAZIONE

Problema: come comunicano tra loro macchine distinte, diverse, fortemente eterogenee?

Le **socket** consentono una **comunicazione standard usando API per lo scambio di messaggi**

Nascono in ambiente Unix BSD 4.2 (1983)

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

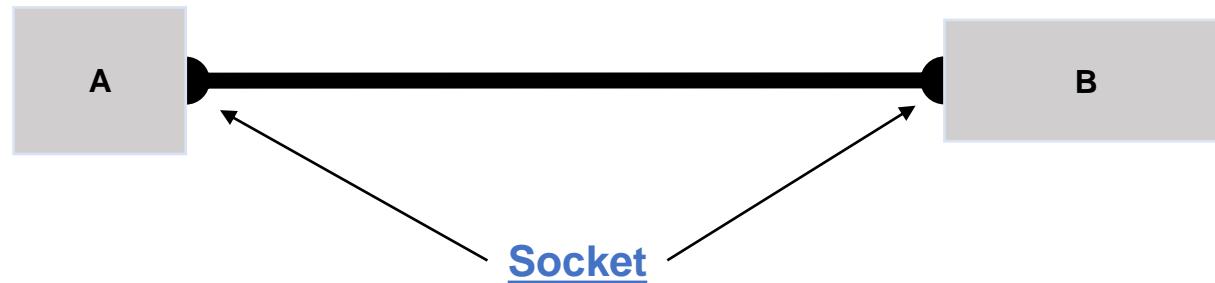


SOCKET PER COMUNICAZIONE

Problema: come comunicano tra loro macchine distinte, diverse, fortemente eterogenee?

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

in **Java** è possibile programmare la rete attraverso **meccanismi di visibilità della comunicazione** (*sul sistema operativo*) contenuti in classi specifiche del package di networking `java.net`



TIPI DI COMUNICAZIONE

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

classi per **SOCKET INTERNET** (in Java)

senza connessione usando il protocollo Internet **UDP**

- classe **DatagramSocket**, per socket (C/S)

con connessione usando il protocollo Internet **TCP**

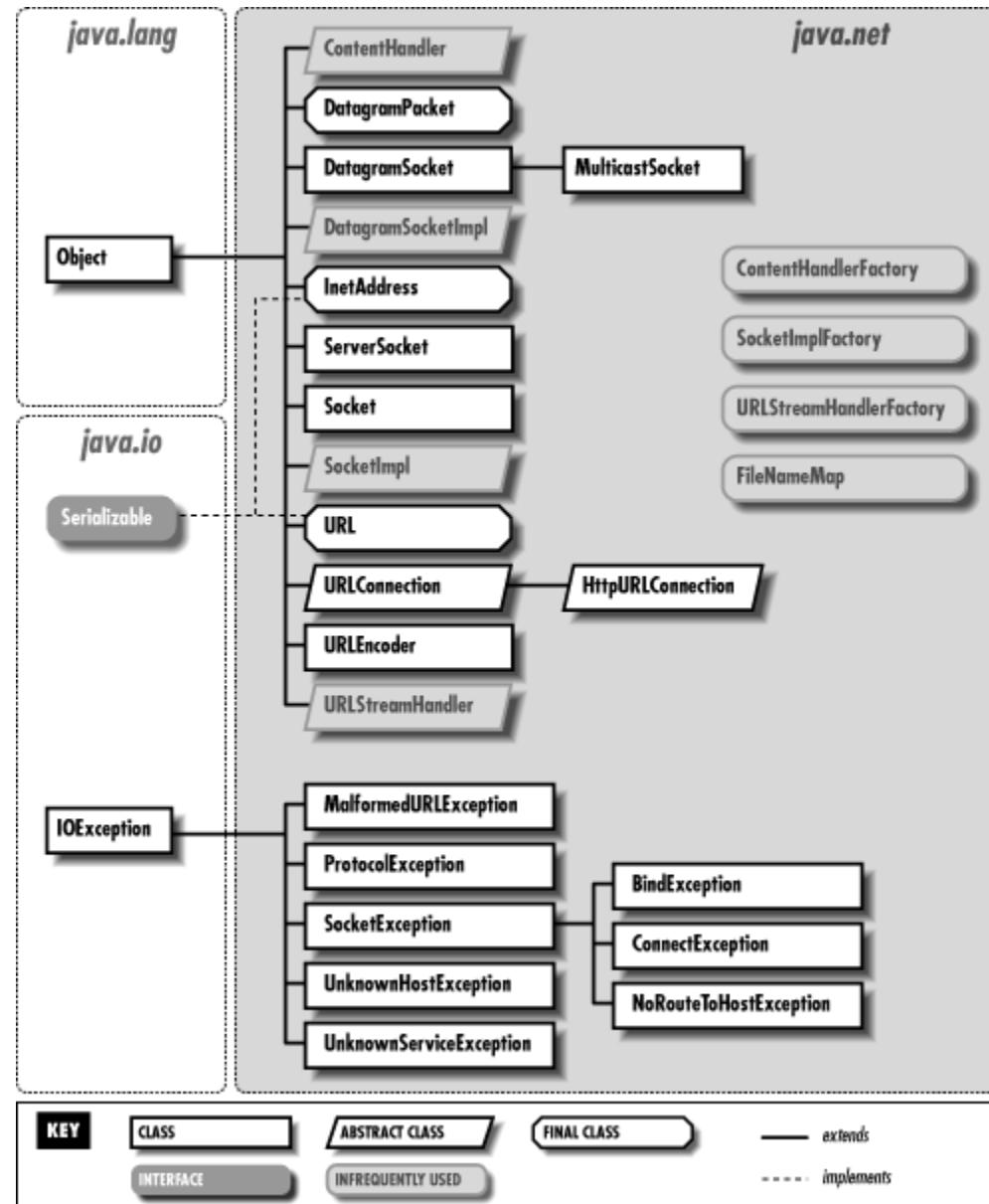
- classe **Socket**, per socket lato Client
- classe **ServerSocket**, per socket lato Server

GERARCHIA DELLE CLASSI IN JAVA

Package Java.net

Classi e 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:
creare meccanismi ed API standard per lo scambio di messaggi



SOCKET E APPLICAZIONI

Le socket sono uno **strumento di comunicazione**, ossia un supporto allo scambio di messaggi

Bisogna capire come potere comunicare

Obiettivo: progettare applicazioni Client/Server (**C/S**) con le socket

1) D. Come trovare **un pari di interesse?**

R. **Conoscendo il suo nome** o grazie ad un **sistema di servizi di nome (name service)**

2) D. Come potere creare **sistemi C/S** dalla comunicazione?

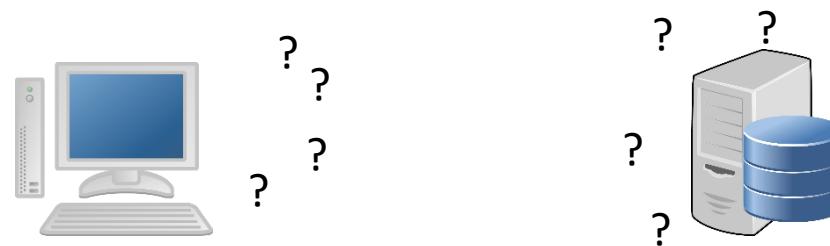
R. Progettando regole **Client / Server** in modo **armonico e consistente**

Le Socket **consentono questi due passi**

SISTEMA DI NOMI

Problema: Identificare gli 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 (Client o Server)** nella rete, i cui nomi hanno solo validità locale

SISTEMA DI NOMI

Problema: Identificare gli enti in gioco

Ogni processo locale deve essere associato ad un **NOME GLOBALE**, **visibile in modo univoco, non ambiguo, e semplice** che altri possano usare per raggiungerlo

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



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

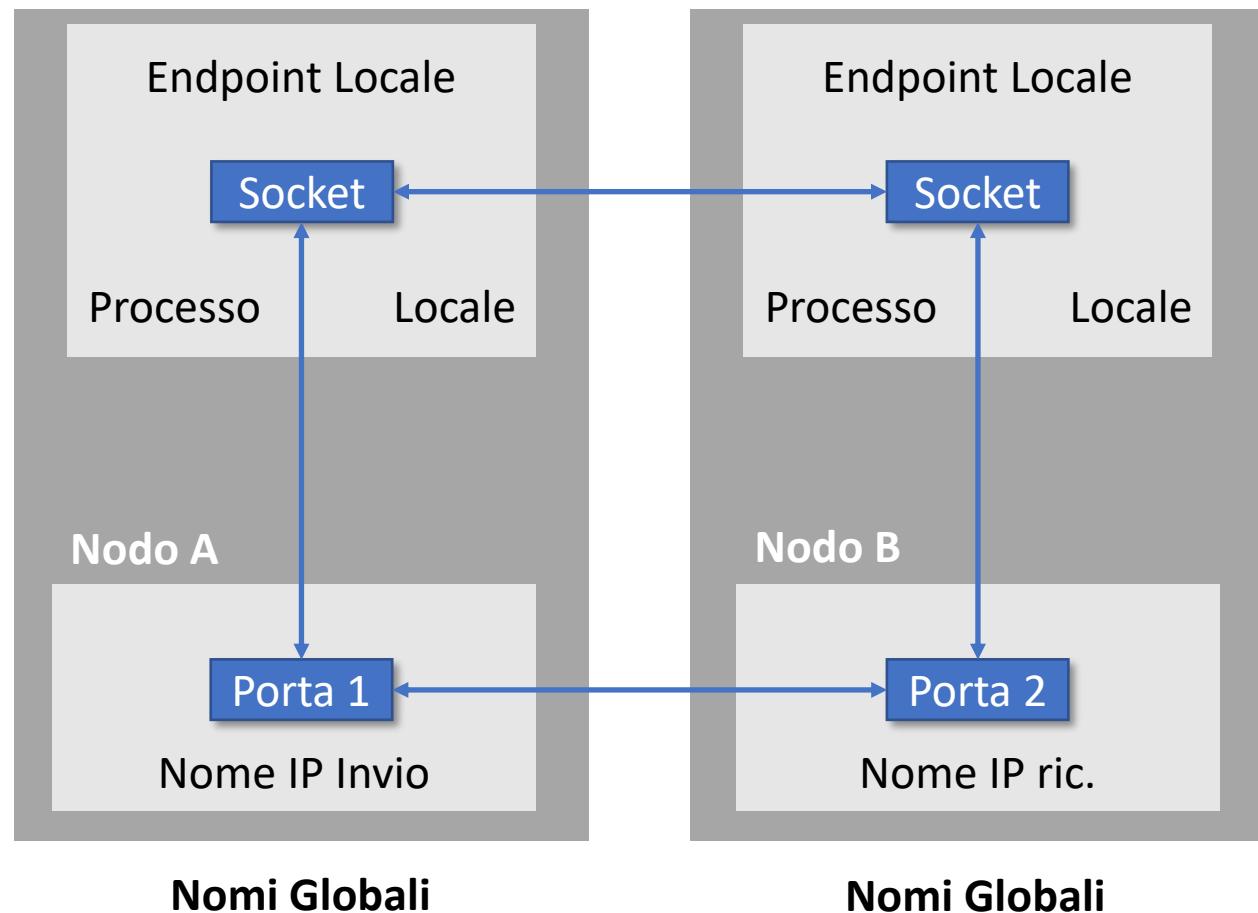
Il problema è risolto dai livelli bassi di protocollo (trasporto e rete) **per le socket in Internet, i nomi di trasporto (TCP, UDP) e rete (IP)**

SISTEMA DI NOMI

Nei sistemi **distribuiti** sono necessarie molte risorse di supporto per la comunicazione

Sistemi di
nomi visibili e
sistemi di
nomi non visibili

Sistemi di
nomi globali e
sistemi di
nomi locali

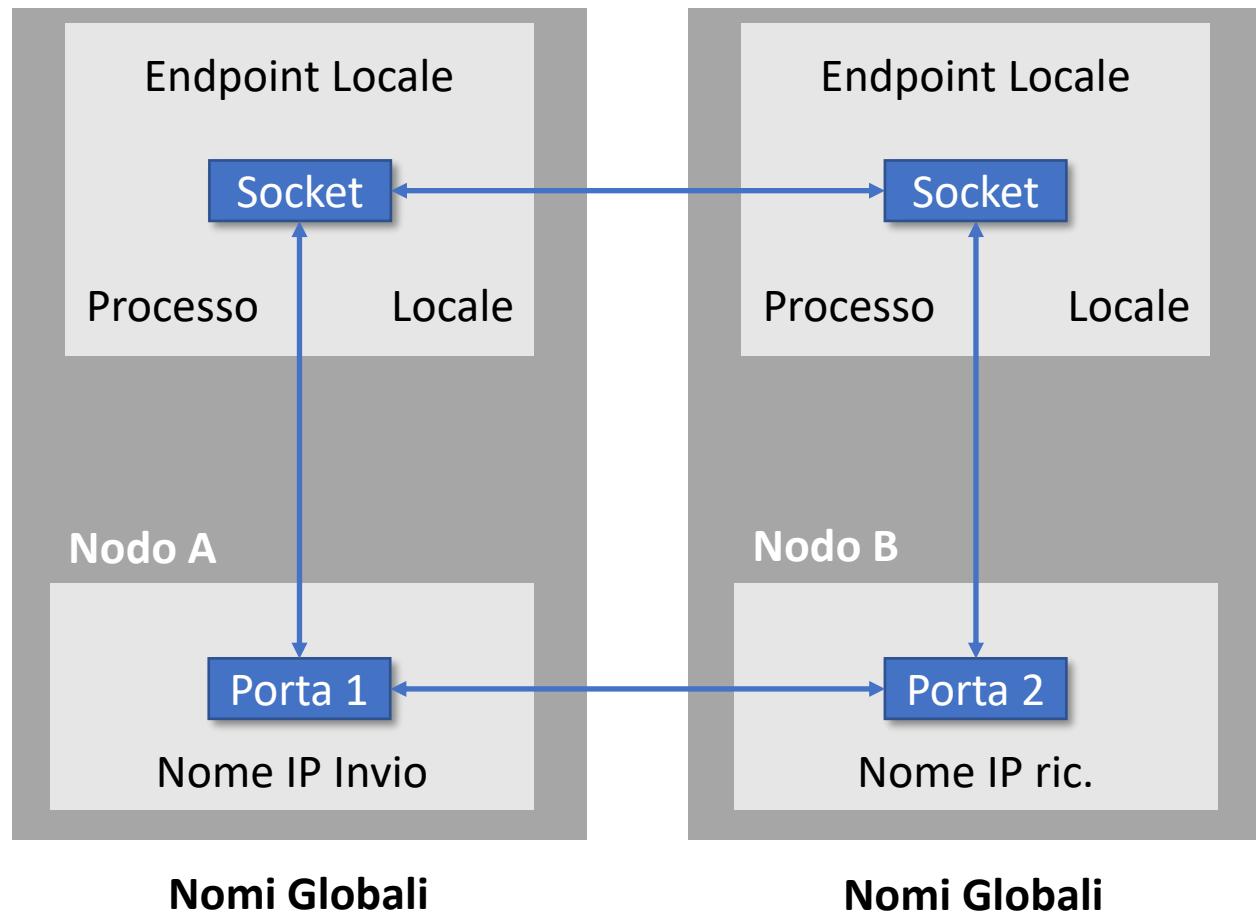


SISTEMA DI NOMI

Nei sistemi **distribuiti** sono necessarie molte risorse di supporto per la comunicazione

BINDING tra sistemi di nomi diversi

Sistemi di nomi globali insieme sistemi di nomi locali



NOMI PER LE SOCKET

Un servizio su un **nodo** è identificato da un **NOME GLOBALE** composto da:

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

Es: 137.204.59.45:25

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

La socket si lega il processo ad un nome globale per ricevere (o spedire) dei messaggi (*anche più processi si collegano*)

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

SERVER SEQUENZIALI IN JAVA

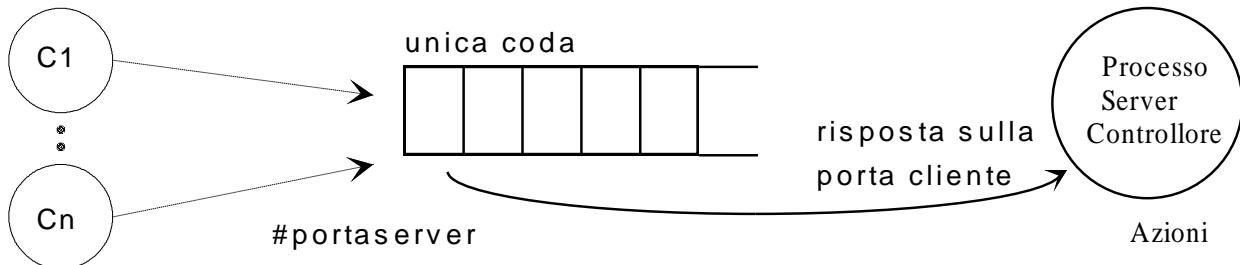
Socket datagram per un server

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

server sequenziale senza connessione (uso UDP)

servizi senza stato e

poco soggetti a guasti



Socket stream per un server

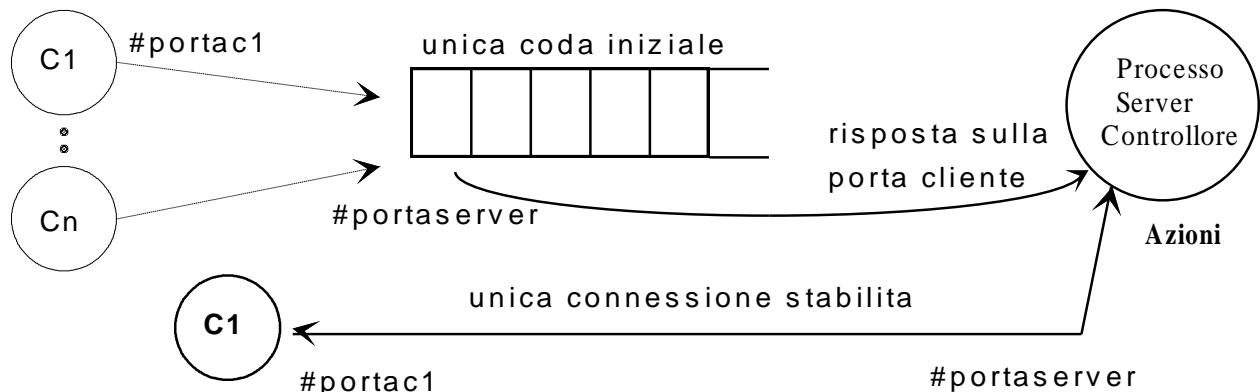
server sequenziale con connessione (uso TCP)

servizi

con reliability

limitando lo stato

overhead per controllo
della connessione



SERVER PARALLELI IN JAVA

Server concorrente con più richieste alla volta (multiprocesso)

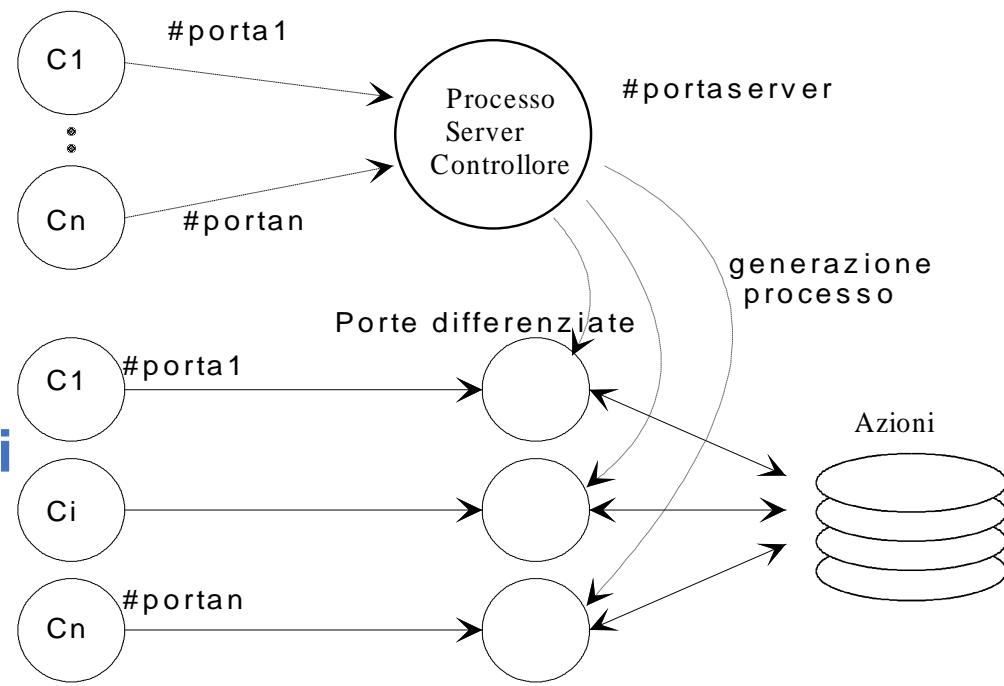
Uso di processi multipli, con un **master server** (main) che genera **processi interni**(thread) per ogni servizio

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

Soluzione con thread:

il costo di generazione è
molto limitato

Possiamo avere **multi processi**
(o thread) che lavorano sullo
stesso servizio



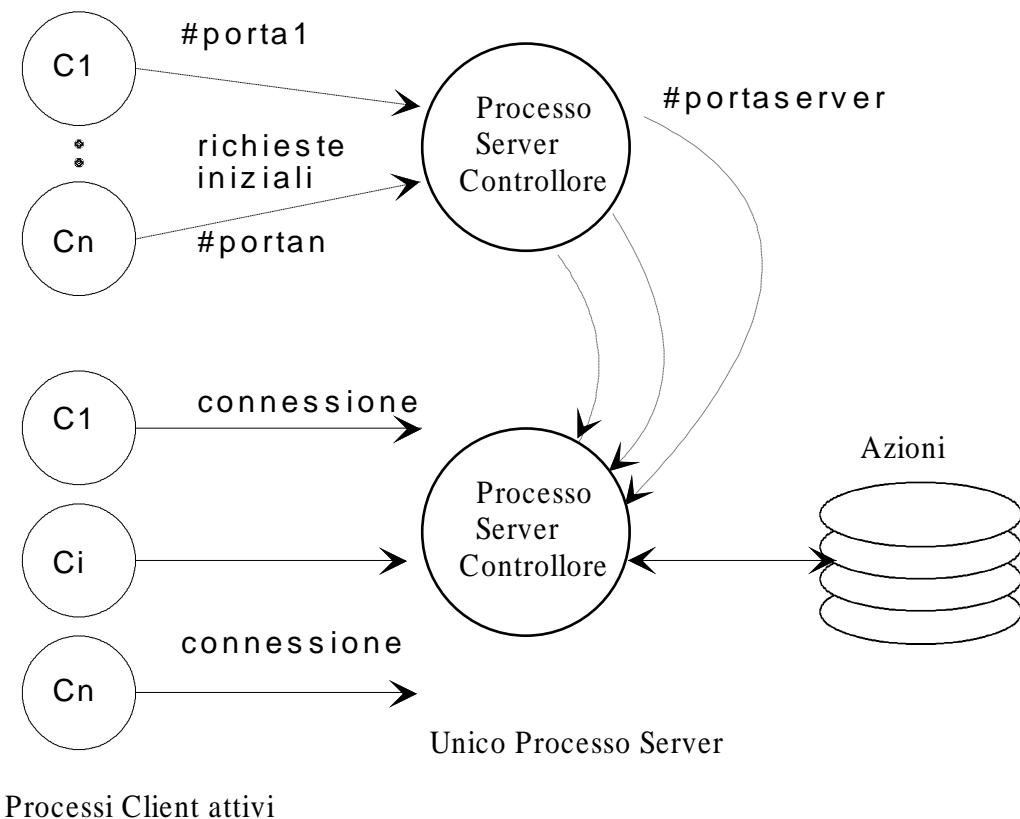
SERVER CONCORRENTI (IN JAVA)

Server concorrente con più richieste alla volta (monoprocesso)

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

Poco significativa in Java
e non usata in Java
avendo i thread a basso
costo da utilizzare

Soluzione in cui un processo server unico con connessione 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 (una o più socket datagram)

Attenzione! Meccanismo non 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: la socket è pronta

COMUNICAZIONE CON SOCKET DATAGRAM

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

Su una istanza di **socket** 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** implica solo la **consegna ad un livello di kernel locale** che si occupa dell'invio solo (**asincrona con la ricezione**).

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

Basta ricevere un datagramma per sbloccare una receive

SEND E RECEIVE SU SOCKED DATAGRAM

Send e **receive**, oltre a richiedere una socket correttamente inizializzata, usano sempre delle struttura di appoggio che servono in input per receive e in output per invio:

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

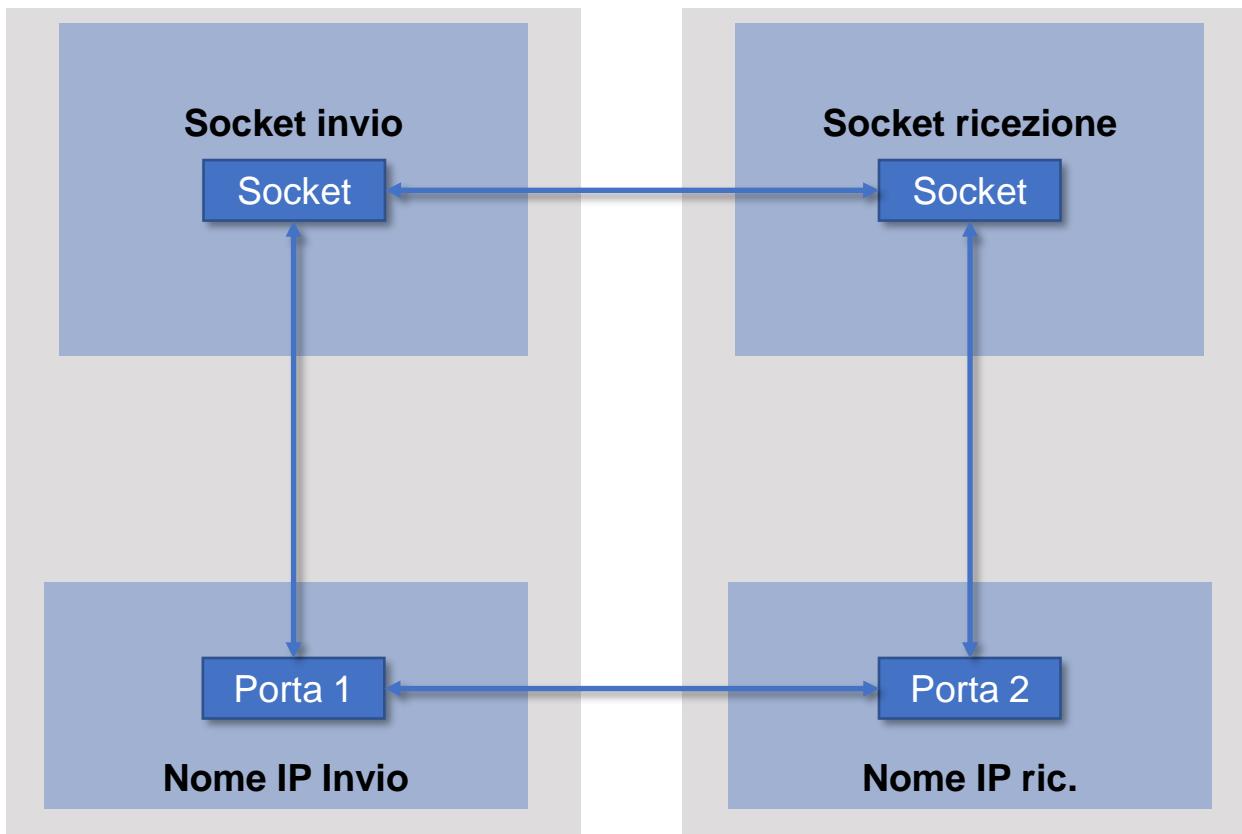
Si usano molte **classi accessorie di supporto** come i **DatagramPacket** e altre costanti

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

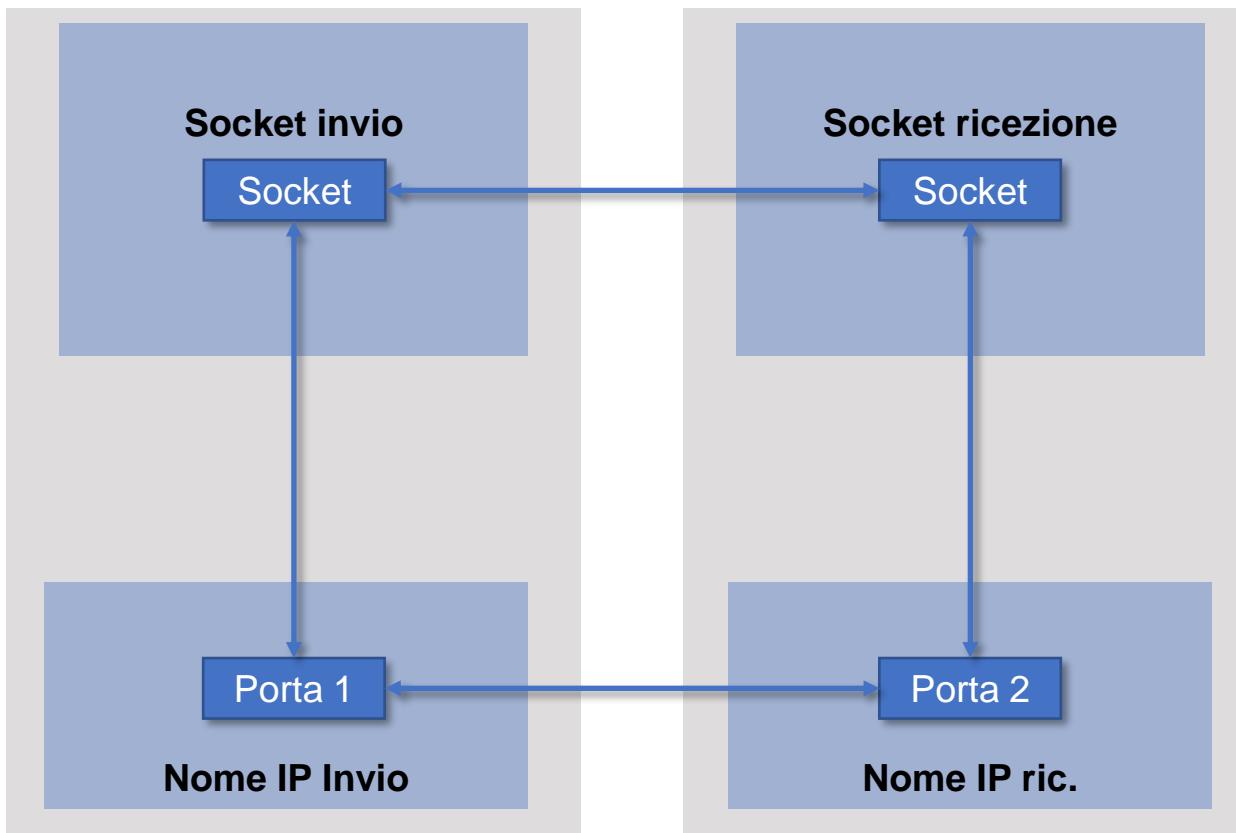
Le **socket datagram** per scambio di messaggi devono essere inizializzate correttamente (create) e devono conoscersi
il mittente deve specificare nel messaggio un ricevente



MODELLO DI COMUNICAZIONE

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.



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

`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

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

CLASSE DATAGRAMPACKET

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

sock.send (DatagramPacket)

- 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 (DatagramPacket)

- 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

PROTOCOLLO DI COMUNICAZIONE C/S

```
// Creazione socket Mittente
```

```
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
```

Sono possibili altre operazioni di invio o di ricezione

COMUNICAZIONE: RICEZIONE

```
// Creazione socket Ricevente Set di IP e porta
InetAddress add = InetAddress.getByName("137.204.59.72");
DatagramSocket socket = new DatagramSocket(add, 1900);

// Parte ricevente di comunicazione: Preparazione, attesa, e
// ricezione
int recport; InetAddress recadd; byte[] res = new byte[200];
DatagramPacket packet = new
DatagramPacket(res, res.length, recadd, recport);
packet.setData(res); // riaggancio della struttura dati
socket.receive(packet); // ricezione con attesa sincrona
// estrazione delle informazione dal datagramma
recport = packet.getPort();
recaddress = packet.getAddress();
res = packet.getData();
// uso delle informazioni ...
```

COMUNICAZIONE VIA DATAGRAM

Per potere comunicare, è necessario **inizializzare almeno una socket datagram** per invio e ricezione di datagrammi

- Le comunicazioni possono essere indirizzate da e verso ogni altro endpoint
- Per mandare o ricevere informazioni, dobbiamo avere preparato un datagramma che guida la parte di supporto
- Il datagramma deve contenere sia le informazioni di comunicazione sia le informazioni applicative di contenuto

Con una socket ed un datagramma possiamo **mandare e ricevere** informazioni da ogni altro endpoint.... Una alla volta.

I programmi quindi possono essere molto semplici

Ricordiamo che non c'è nessuna garanzia di qualità a livello di supporto

DATAOUTPUTSTREAM E DATAINPUTSTREAM

DataOutputStream e DataInputStream offrono una serie di metodi per l'invio e la ricezione di tipi primitivi Java tra macchie virtuali diverse

Uso tipico: realizzazione di **protocolli** fra Client e Server in Java con scambio di oggetti Java.

Nel corso vengono usati per la **realizzazione di applicazioni C/S in Java**

Ad esempio: (ricorda in input eof trasformato in eccezione)

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

UTF STRING: UTF-16

Il linguaggio Java deve ottenere un comportamento atteso per le stringhe e anche consentire le diverse internazionalizzazioni necessarie e a livello commerciale

Le stringhe in Java sono tipicamente accessibili attraverso **UTF**, ossia **UTF Unified o Unicode Transformation Format** che permette il massimo dello standard

Tipicamente il supporto JVM memorizza le stringhe attraverso un formato interno **UTF-16** (a 16 bit x carattere) e usa:

- un **contatore dei caratteri** della stringa
- un **array di coppie di byte** (due byte per ogni carattere della stringa)

COMUNICAZIONE: MODI DI GRUPPO

La comunicazione a datagrammi sfrutta in modo molto semplice i protocolli sottostanti (UDP e IP, nei due livelli)

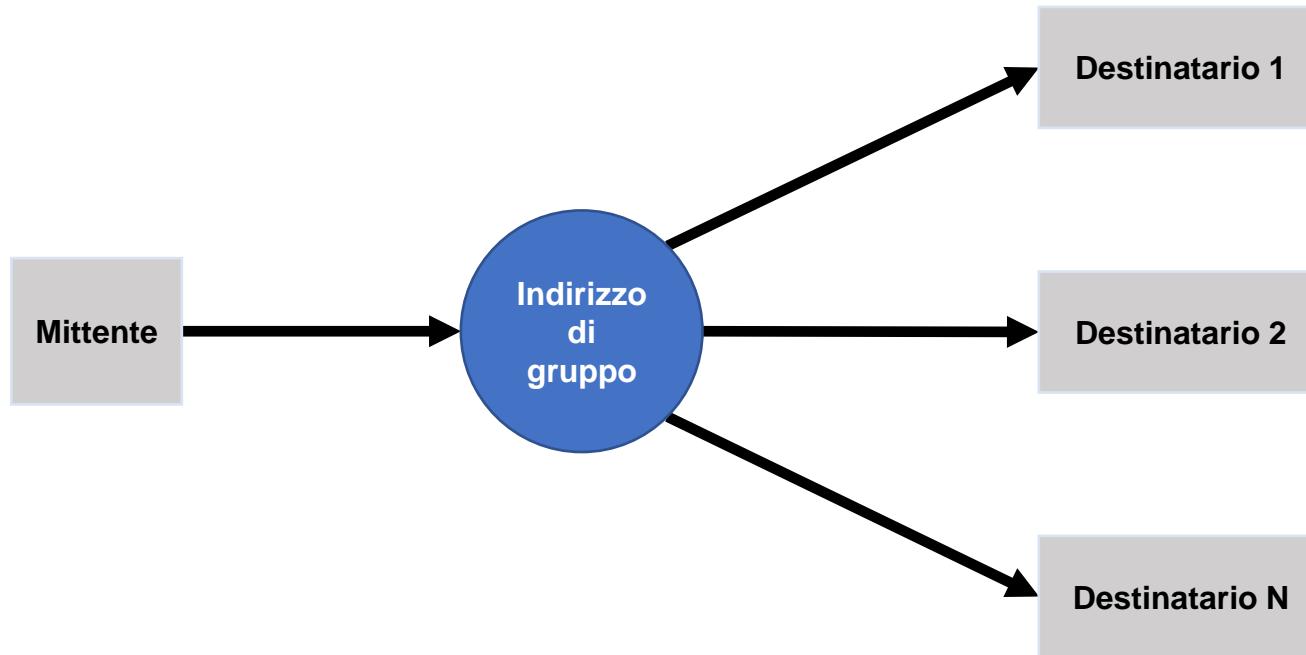
- Le comunicazioni più comuni sono **punto-punto**: un mittente comunica con un ricevente alla volta



COMUNICAZIONE: MODI DI GRUPPO

La comunicazione a datagrammi consente l'uso di protocolli di gruppo (non punto a punto), ossia protocolli di multicast

- Una comunicazione multicast permette di **inviare messaggi a una serie di destinatari** che siano registrati su un **indirizzo di gruppo** (di ricezione)



COMUNICAZIONE MULTICAST

SOCKET MULTICAST è registrata in una **ulteriore classe** per gestire gruppi di multicast e ricevere messaggi multicast (**Classe D: 224.0.0.0 - 239.255.255.255**)

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

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)

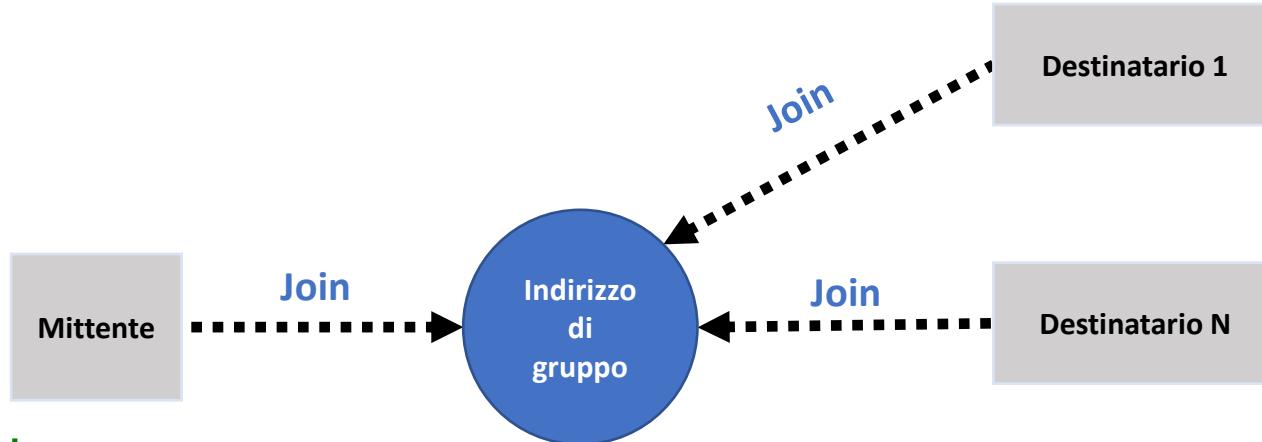
```
// unisciti al gruppo ... e esci dal gruppo
```

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

```
// il sistema operativo può tenere conto della porta per selezionare i messaggi
```

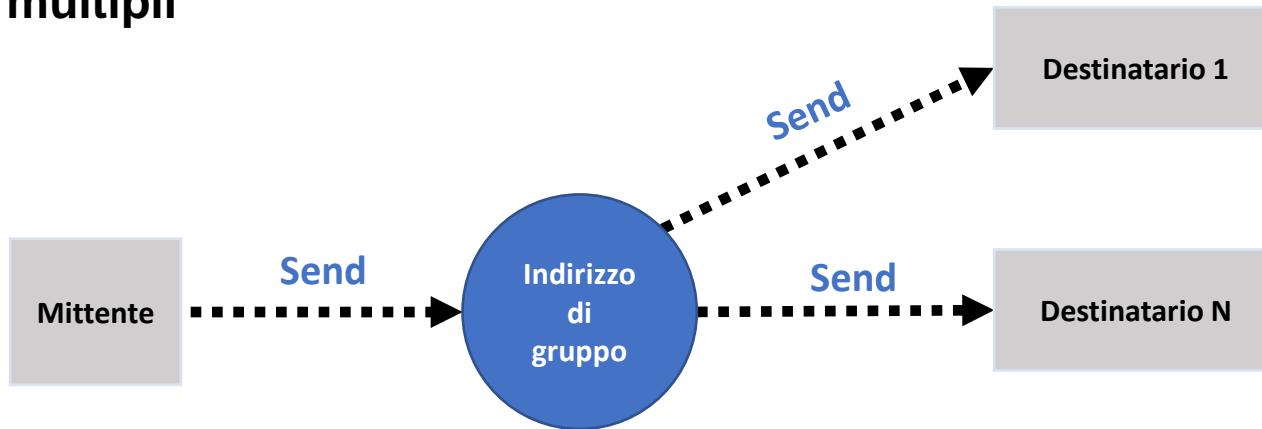
COMUNICAZIONE MULTICAST

Per comunicazioni Multicast join/leave



(unico invio

Poi semplice invio e ricezione multipla: unica azione di invio per raggiungere i riceventi multipli



JAVA MULTICAST

Per comunicazioni Multicast **join/leave**:

l'indirizzo IP di classe D è necessario e a questo si deve fare riferimento per **inviare (senza troppa preparazione)** e per **ricevere**

I nodi che vogliono ricevere **devono preparare la ricezione.**

Il gruppo viene creato ed alimentato **da ingressi, come registrazione o manifestazione di interesse, fino all'uscita (join e leave)**

In **Java** abbiamo le **porte** che consentono di essere **più selettivi**, ossia il gruppo è composto da **solo chi ha usato la stessa porta**

Si possono avere molti gruppi sullo stesso indirizzo IP di classe D, distinti dalla porta

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

Uso della socket multicast per ricevere (API aggiunte)

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

Si riceve nell'ambito della sessione di sottoscrizione

OPZIONI SOCKET

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

Le opzioni delle Socket servono ad adattare il comportamento degli strumenti a seconda delle necessità.

La ricezione da socket (es., receive()) è sincrona bloccante.

`SetSoTimeout (int timeout) throws ...` // Questa opzione definisce un **timeout** in msec, dopo il quale **l'operazione termina** (e viene lanciata una eccezione da gestire). Se il parametro timeout è zero, nessun timeout (timeout infinito).

`SetSendBufferSize (int size) throws ...`

`SetReceiveBufferSize (int size) throws ...` // Il buffer di invio e ricezione della driver può essere variato.

Sono previste le get corrispondenti della opzioni significative:

`SetReuseAddress ()`

Si possono collegare più processi ad un certo indirizzo fisico (porta).

SOCKET A STREAM

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

Semantica **at-most once** (ricezione una volta sola anche se reinvio)

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).

Si garantiscono ritrasmissioni al livello di trasporto

Se i dati non consegnati??? **Si può dire poco...**

SOCKET A STREAM

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

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

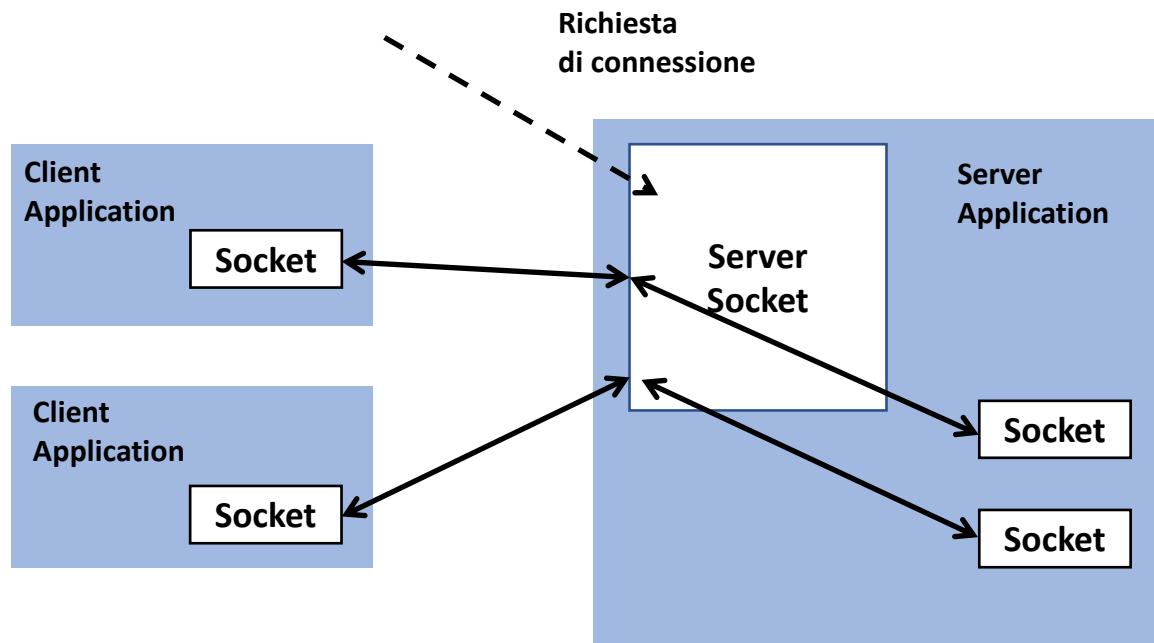
Nel caso delle socket STREAM, il protocollo è **TCP (+ IP)**

- **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 (**API specifiche**).

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 (`java.net.Socket`) e Servitore (`java.net.ServerSocket`)



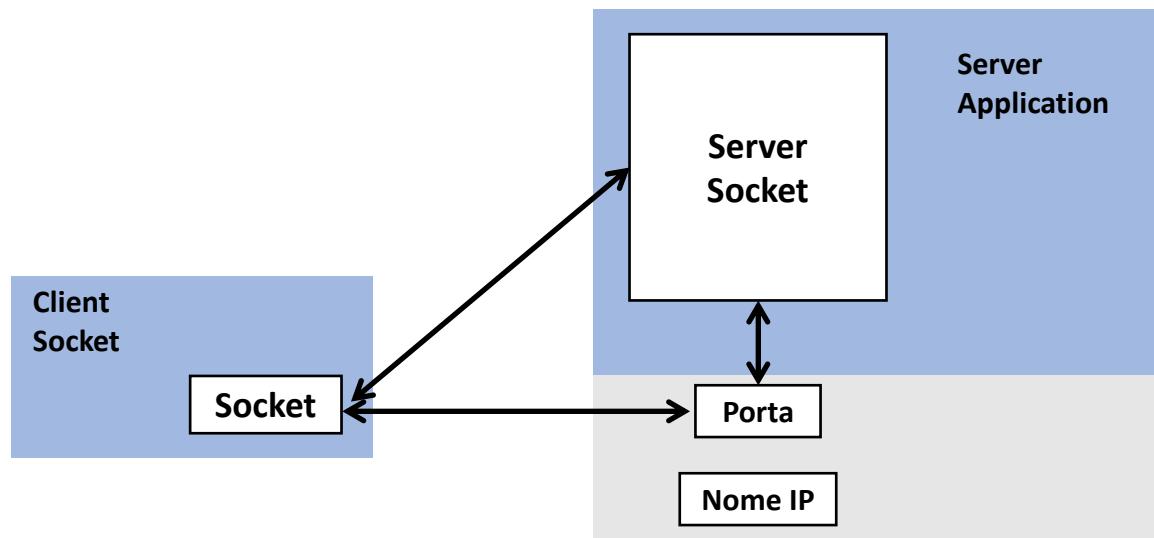
Altro principio: **dove possibile, si nascondono i dettagli realizzativi dei protocolli**, ad esempio nei **costruttori delle classi**

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 una comunicazione dati bidirezionale (**full duplex**)



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***

SOCKET A STREAM: 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 devono mantenere le sole connessioni necessarie e limitare le aperture contemporanee di sessioni chiudendo quelle non 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();`

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

`public InetAddress getLocalAddress();`

(restituisce l'indirizzo della macchina locale)

`public int getPort();`

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

`public int getLocalPort();`

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

Esempio:

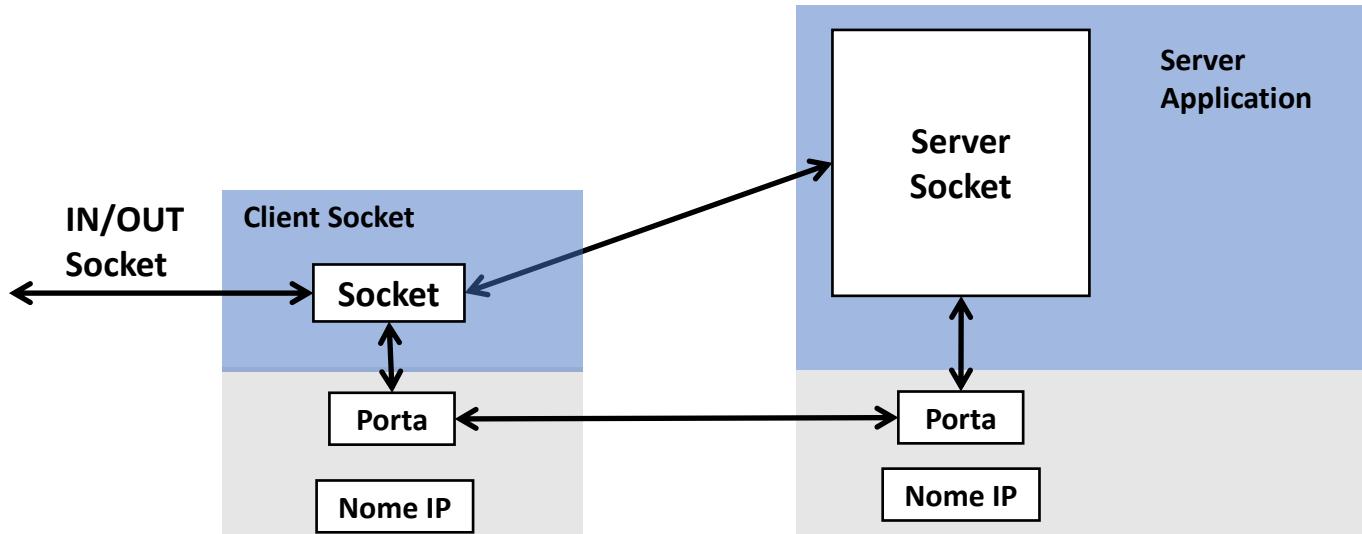
`int porta = oggettoSocket.getPort();`

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 (generici di byte)** si possono spedire/ricevere solo **byte, senza nessuna formattazione in messaggi** (vedi classi)

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 in Java con scambio di oggetti Java.

Nel corso vengono usati per la **realizzazione di applicazioni C/S in Java**

Ad esempio: (vedi eof trasformato in eccezione)

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

ESEMPIO CLIENTE STREAM

Client di echo (il Server Unix è sulla *porta nota 7*) progetto di filtro

```
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;
```

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

La **accept restituisce un oggetto della classe Socket** su cui avviene la comunicazione di byte vera 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();`

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

`public InetAddress getLocalAddress();`

(restituisce l'indirizzo della macchina locale)

`public int getPort();`

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

`public int getLocalPort();`

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

ESEMPIO C/S SERVER STREAM

Server daytime (il Server Unix su porta 13) progetto di demone

```
... 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)
{oggConnessione.close(); oggServer.close();
 System.err.println(e);}
```

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

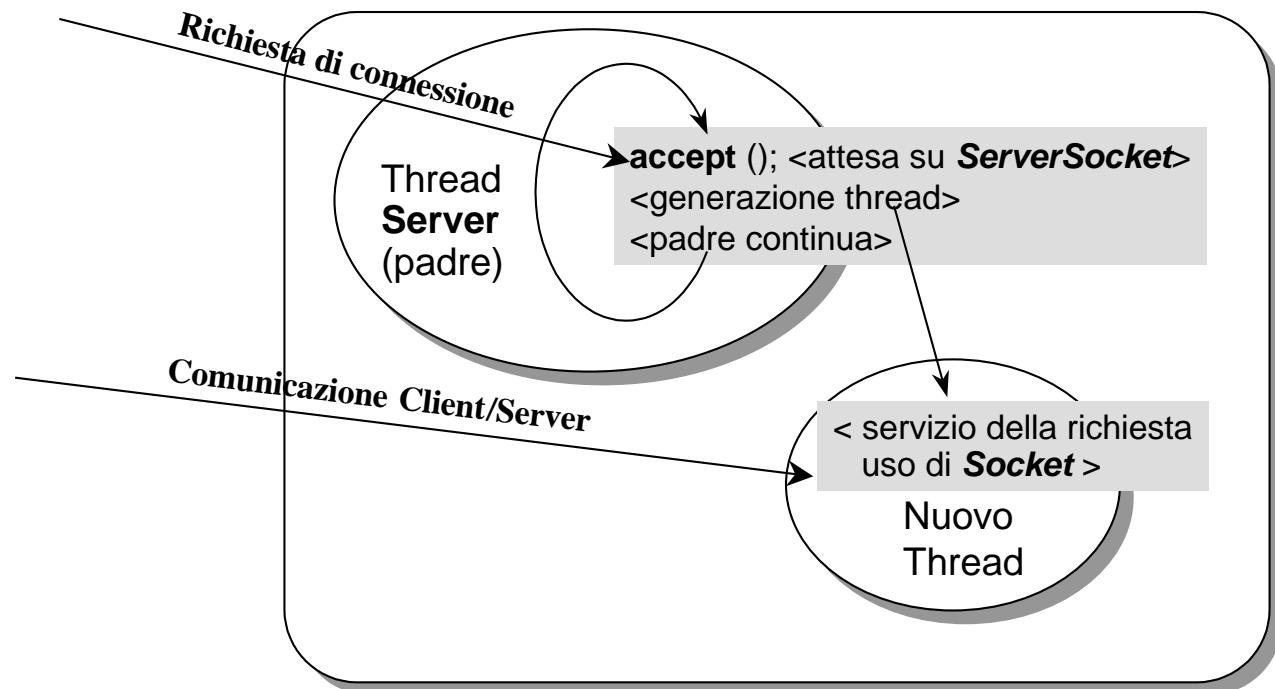
SERVER PARALLELO

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 DI SOCKET 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 nomefilesorg nomefiledest

nodoserver e portaserver indicano il **Server** e **nomefilesorg** è il nome di un **file** presente nel file system della macchina Client

Il processo Client deve inviare il file **nomefilesorg** al Server che lo scrive nel **direttorio corrente** con nome **nomefiledest**

La scrittura del file nel directory specificato deve essere eseguita solo se in tale directory 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**)

! Si vedano Versioni diverse dal punto di vista dei file: confrontare !

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)
/* scrittura file */
{FDaSpedDescr = new File(NomeFile); // ottiene file locale in memoria?!
FDaSpedInStream = new FileInputStream(FDaSpedDescr);
int singoloByte=0;
/* legge il file un byte alla volta: a eof valore negativo */
while ((singoloByte = FDaSpedInStream.read()) >= 0)
    OutSocket.write(singoloByte);
} catch(IOException e) ... {}
rcpSocket.close(); ...
```

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)
```

RCP PARTE SERVER SEQUENZIALE

```
// lettura file ...
{ OutSocket.writeUTF
    ("MSGsrv: file presente, bye");
else {
    OutSocket.writeUTF("MSGsrv: attendofile");
    FileOutputStream =
        new FileOutputStream (FileDaScrivere);
    int singoloByte=0;
    while ( (singoloByte = InSocket.read()) >= 0)
        FileOutputStream.write(singoloByte);
}
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 della **nuova socket** avviene la interazione

RCP SERVER PARALLELO

In **caso parallelo** si genera un nuovo processo per ogni connessione accettata

Il nuovo processo o thread ha la propria connessione e la propria ServerSocket di autorità

Ogni **socket usata** è idealmente visibile da tutti i thread (condivisione risorse in Java): ma qui ogni thread vede e usa solo la propria

Per la socket iniziale, la prima **close** chiude la socket definitivamente per tutti i thread, fino ad allora impegnava risorse di supporto

??? c'è un limite al numero di socket aperte per 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");

DaScrivereStream =
new FileOutputStream (FileDaScrivere);
int singoloByte=0;
while ( (singoloByte = InSocket.read()) >= 0)
    DaScrivereStream.write(singoloByte);
} /* 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 una serie di **risorse** di sistema che sono collegate e necessarie per la operatività fino alla `socket.close()`

La chiusura è sempre necessaria per consentire al sistema di liberare le risorse utilizzate dalla socket

- In caso di una socket chiusa, **le risorse sono mantenute per un certo tempo** (a seconda dell'importanza delle operazioni e non per la memoria **IN**, eliminata subito)
- In caso di socket **connessa chiusa**, la memoria **OUT** viene mantenuta fino a spedire tutte le informazioni da inviare al pari.

Il pari si accorge di questo tramite **eccezioni, predicati o 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 la fine di una connessione in entrambi i versi fatta da uno dei due endpoint connessi

In caso di una connessione, ogni partecipante è responsabile **esclusivamente della sua uscita sulla connessione**, mentre 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 `shutdownOutput()` che chiude solo 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:

`isClosed ()`;

`isConnected ()`;

`isShutdownInput ()`;

`isShutdownOutput ()`;

OPZIONI SOCKET STREAM

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 per un periodo. Questa opzione permette di scartare i pacchetti in attesa dopo l'**intervallo di linger** in sec)

SetTcpNoDelay (boolean **on**) throws ...

(il pacchetto è **invia 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