

La Programmazione Client/Server e le socket

Un Client e un Server su macchine diverse possono comunicare mediante socket.

Una **socket** rappresenta un **terminale** (end point) di un **canale di comunicazione bidirezionale**.

Ci sono due diversi tipi di modalità di comunicazione con le socket:

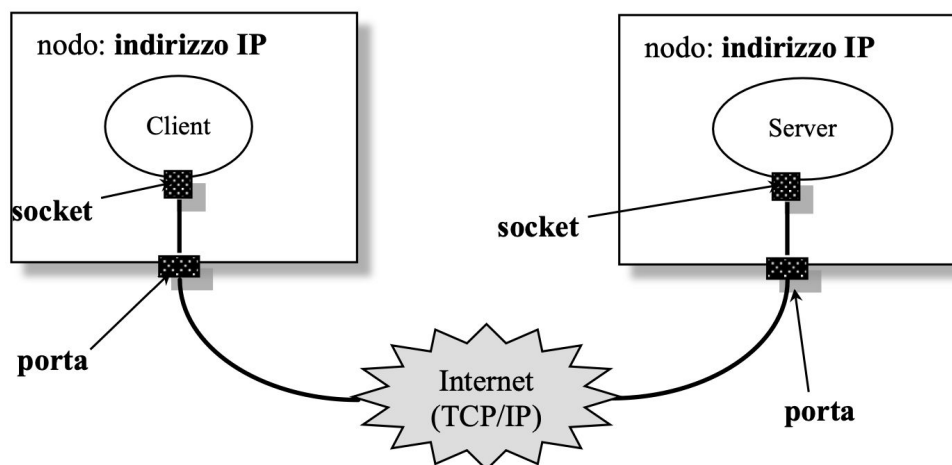
- **con connessione e affidabile**, in cui viene stabilita una connessione senza perdita di messaggi tra Client e Server (esempio, il servizio telefonico).
Uso di socket STREAM
- **senza connessione e non affidabile**, in cui non c'è connessione e i messaggi vengono recapitati uno indipendentemente dall'altro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

C/S in Java: le socket - 1

Identificazione dei processi comunicanti (sistema di nomi)

Identificazione dei processi (il Client e il Server) nella rete, uso di **nomi globali** univoci e sempre non ambigui:

Indirizzo IP della macchina + Porta come identificativo del Servizio



Un **indirizzo IP** e una **porta** rappresentano un endpoint di un canale di comunicazione.

C/S in Java: le socket - 2

Numeri di Porta

Funzione fondamentale delle porte è **identificare un servizio**. La richiesta di un servizio a un processo server non richiede quindi la conoscenza a priori del suo pid, ma solo della porta (a cui il server è legato) e dell'indirizzo IP della macchina su cui esegue il Server.

I numeri di porta tra 0 e 1023 sono riservati (**well-known ports**), nel senso che possono essere usati solo da processi di sistema (demoni che eseguono con privilegi di root).

Queste porte vengono usate per servizi **standardizzati** ben noti.

Per esempio, il servizio Web è identificato dalla porta numero 80, cioè il processo server di un sito Web deve legarsi alla porta 80, su cui riceve i messaggi con le richieste di pagine html. Altri esempi: porta 21 per ftp, porta 22 per ssh, porta 25 per mail,...

Porte 1024-49151 sono dette **registered**, e sono anche esse standardizzate ma disponibili per processi che eseguono senza privilegi di amministratore.

Porte 49152-65535 sono dette **dinamiche** (o private) a uso libero.

Attribuzione delle porte ai servizi su www.iana.org

Si provi il comando `netstat -l` e si veda il file `/etc/services` (su macchine Unix).

C/S in Java: le socket - 3

Le socket in Java: gerarchia classi

I servizi di comunicazione Java sono forniti dalle classi del package di networking `java.net`

Si consulti la documentazione:

<https://docs.oracle.com/en/java/javase/>

Classi per Socket:

Con connessione (TCP):

- classe `Socket` per Client
- classe `ServerSocket` per Server

Senza connessione (UDP):

- classe `DatagramSocket` per Client e Server

C/S in Java: le socket - 4

Comunicazione con connessione: Java socket 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 i dati che vengono **consegnati in sequenza** (in modalità **FIFO**, stessa semantica delle pipe di Unix).

Queste caratteristiche discendono dalle caratteristiche del protocollo di comunicazione sottostante, il TCP.

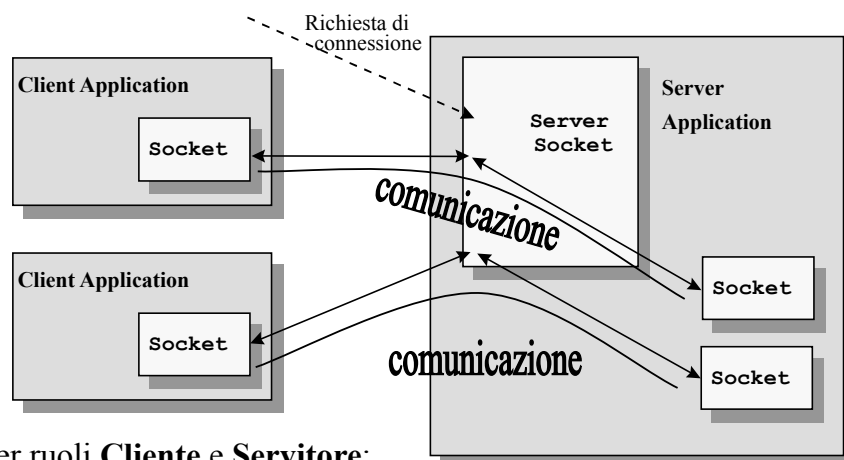
- TCP è un protocollo di trasporto (livello 4 OSI) e fornisce l'astrazione delle porte per identificare i servizi.
- IP è un protocollo di rete (livello 3 OSI) e fornisce l'indirizzo IP per identificare univocamente ogni macchina collegata alla rete Internet.

La **connessione** tra i processi Client e Server è definita da:

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

Comunicazione con connessione: Java socket per STREAM

La comunicazione tra Client e Server segue uno schema **asimmetrico n:1**. Questa considerazione ha portato al progetto di due tipi di socket diverse, una per il Client e una per il Server



Classi distinte per ruoli Cliente e Servitore:

`java.net.Socket` per la socket lato Client

`java.net.ServerSocket` per la socket lato Server

Lato Cliente: classe `java.net.Socket`

La classe `Socket` consente al **Client** di creare una socket STREAM (TCP) per la comunicazione con il Server.

Tale socket è detta anche socket “attiva”.

I **costruttori** della classe **creano** la socket, la **legano** a una porta locale e la **connettono** alla socket del Server (attraverso la porta della macchina remota).

Si noti che la creazione della socket produce in modo atomico anche la connessione al Server corrispondente.

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

Lato Cliente: classe `java.net.Socket`

Presenti diversi **costruttori** (ovviamente `public`) della classe **Socket**:

Socket(`InetAddress remoteAddr`, `int remotePort`) throws ...

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

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

Crea una **socket stream cliente** e la collega alla porta specificata della macchina remota di nome logico `remoteHost` (equivale in Unix a: `getaddrinfo`, `socket`, `bind`, `connect`)

Socket(`InetAddress remoteAddr`, `int remotePort`,
`InetAddress localAddr`, `int localPort`) throws ...

Crea una **socket stream cliente** e la collega alla porta specificata della macchina remota. Inoltre lega la socket a una porta della macchina locale (se `localPort` vale zero numero scelto automaticamente) e su uno specifico IP locale (utile nel caso di macchine multi-homed)

Lettura/scrittura su socket Java

Come si legge/scrive su una socket in Java? A questo proposito si ricordino gli stream: uno **stream** è una **sequenza ordinata di byte**, che viene acceduto in modo **sequenziale** (si ricordino le pipe Unix, per esempio).

Gli stream modellano bene i dispositivi e sono quindi il modo in cui i programmi Java accedono a tutto l'I/O.

Stream di ingresso, **InputStream** per **leggere** byte. Attenzione: lettura può sospendere un processo in attesa di ricevere dati.

Stream di uscita, **OutputStream** per **scrivere** byte.

Nel caso socket, i metodi (della classe socket):

```
public InputStream getInputStream()
```

```
public OutputStream getOutputStream()
```

Restituiscono `InputStream` e `OutputStream` per leggere/scrivere byte dalla/sulla socket. **Incapsulano il canale di comunicazione.**

Lettura/scrittura su socket Java

Esempio di creazione e uso stream nel caso socket:

Parte di ingresso del canale di comunicazione:

```
DataInputStream in = new DataInputStream(sock.getInputStream());  
float f = in.readFloat();
```

Parte di uscita del canale di comunicazione:

```
DataOutputStream out = new DataOutputStream(sock.getOutputStream());  
out.writeFloat(1.0);
```

Attenzione:

I metodi di tutte queste classi possono lanciare eccezioni di tipo **IOException**, che sono da gestire (si noti che in programmazione di rete le eccezioni sono possibili, anzi *purtroppo* probabili).

Lettura/scrittura su socket Java

Wrapping streams: uno stream può essere impacchettato in un altro stream per fornire altre funzioni, di più alto livello (si noti `DataInputStream` in esempio precedente che permette di leggere float dal canale di comunicazione).

Le varie classi di stream sono orientate ai byte. Per gestire dei caratteri si utilizzano invece le classi **Reader/Writer** (conversione automatica da flussi di byte a stringhe con supporto a diversi encoding e caratteri multibyte, es. UTF-8).

Esempio di creazione reader e wrapping in `BufferedReader` per line-oriented input:

```
InputStreamReader isr =  
    new InputStreamReader(sock.getInputStream(), "UTF-8");  
BufferedReader in = new BufferedReader(isr);  
String line = in.readLine();
```

Si deve sempre specificare
sempre l'encoding usato per
l'I/O sulle Socket!!!

Esempio di creazione writer e wrapping per buffering:

```
OutputStreamWriter osw =  
    new OutputStreamWriter(sock.getOutputStream(), "UTF-8");  
BufferedWriter bw = new BufferedWriter(osw);  
bw.write("qualcosa"); bw.newLine(); bw.flush();
```

C/S in Java: le socket - 11

Limiti alle aperture contemporanee di connessioni

Una connessione impegna risorse dei nodi e dei processi. Costa crearle e mantenerle. Per questo motivo, **il numero di connessioni che un processo (Client o Server) può aprire è limitato, per cui è necessario chiudere le connessioni non più utilizzate.**

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

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

C/S in Java: le socket - 12

Lato Cliente: classe `java.net.Socket`

Per ottenere delle **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();`

C/S in Java: le socket - 13

Client di echo (servizio standard sulla porta 7)

```
try {
    Socket theSocket = new Socket(hostname, 7);
    BufferedReader networkIn = new BufferedReader(
        new InputStreamReader(theSocket.getInputStream(), "UTF-8"));
    BufferedReader userIn = new BufferedReader(new
OutputStreamReader(System.in));
    BufferedWriter networkOut = new BufferedWriter(
        new OutputStreamWriter(theSocket.getOutputStream(), "UTF-8"));

    System.out.println("Connected to echo server");

    while (true) {
        String theLine = userIn.readLine();
        if (theLine.equals(".")) break;
        networkOut.write(theLine);
        networkOut.newLine();
        networkOut.flush();
        System.out.println(networkIn.readLine());
    }
} // end try
catch (IOException e) {
    System.err.println(e);
    System.exit(1);
}
```

Perché non specifichiamo esplicitamente l'encoding quando creiamo l'oggetto userIn di tipo BufferedReader?

BufferedWriter non è strettamente necessario (potremmo facilmente fare a meno del metodo newLine), ma è bene utilizzarlo comunque per motivi di performance.

SocketException

- BindException
- ConnectException
- NoRouteToHostException

BufferedWriter non è strettamente necessario (potremmo facilmente fare a meno del metodo `newLine`), ma è bene utilizzarlo comunque per motivi di performance.

```

} // end try
catch (IOException e) {
    System.err.println(e);
    System.exit(1);
}

```

SocketException

- BindException
- ConnectException
- NoRouteToHostException

Lato Server: classe `java.net.ServerSocket`

La classe `ServerSocket` definisce una socket di “**ascolto**”, capace di accettare delle richieste di connessione provenienti da qualunque Client. Tale socket è detta anche “**passiva**”, per ricordare che il Server ha un ruolo passivo nel rendez-vous con il Client.

Il costruttore **crea** la socket del Server e la **lega** alla porta della macchina server. Definisce anche la lunghezza della coda in cui vengono messe le richieste di connessione non ancora accettate dal Server (si ricordi che il modello C/S è asimmetrico molti:1).
(corrispondenti operazioni Unix: `socket`, `bind` e `listen`)

Costruttori:

```
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
```

C/S in Java: le socket - 15

Lato Server: classe `java.net.ServerSocket`

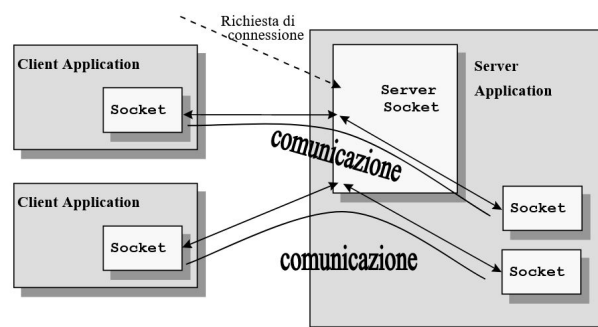
Dopo aver creato la `ServerSocket`, il Server si mette in attesa di nuove richieste di connessione chiamando il metodo `accept()`.

La chiamata di `accept` **blocca** il Server fino all'arrivo di una richiesta.

Quando arriva una richiesta, `accept` stabilisce una nuova connessione tra il Client e un nuovo oggetto `Socket` (una normale socket attiva già vista nei Client) restituito da `accept`.

La trasmissione dei dati avviene con i metodi visti per il lato Client.

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



C/S in Java: le socket - 16

Lato Server: classe `java.net.ServerSocket`

Per ottenere delle informazioni sulle socket passive si possono utilizzare i metodi:

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

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

C/S in Java: le socket - 17

Server daytime (servizio standard sulla porta 13)

```
. . . . .  
try {  
    ServerSocket server = new ServerSocket(13);  
    while (true) {  
        Socket sock = server.accept();  
        // Alla connessione, il Server invia la data al Client  
        BufferedWriter out = new BufferedWriter(  
            new OutputStreamWriter(sock.getOutputStream(), "UTF-8"));  
        out.write(new Date()); // Data e ora corrente  
        out.newLine();  
        out.flush();  
        sock.close();  
    }  
}  
catch (IOException e) {  
    System.err.println(e);  
    System.exit(1);  
}
```

Perché il metodo `accept` è progettato per ritornare una nuova socket, invece di usare la stessa socket passiva anche per lo scambio dei messaggi?

Come vengono gestite più richieste di servizio? (Server sequenziale)

C/S in Java: le socket - 18

Esercizio: copia remota di un file (rcp)

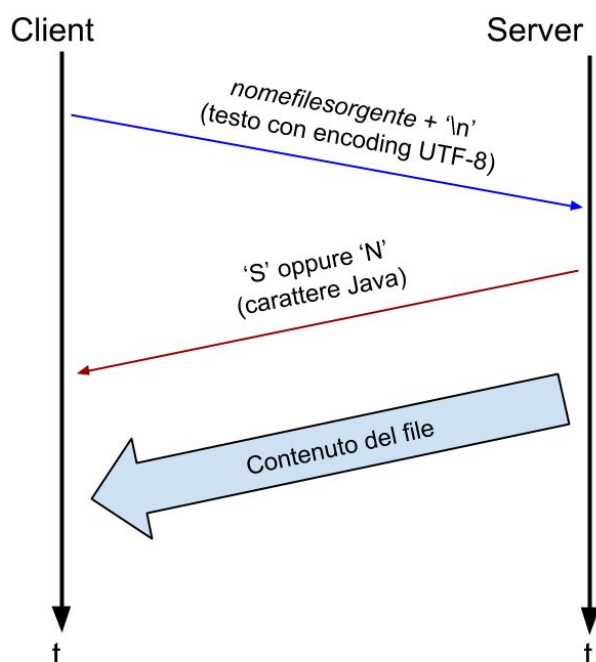
Si realizzi un'applicazione distribuita Client/Server per eseguire la copia remota (remote copy, rcp) di file dal Server al Client. Il Client deve presentare l'interfaccia:

rcp_client nodoserver portaserver nomefilesorgente nomefiledestinazione

dove *nodoserver* e *portaserver* indicano il processo Server e *nomefilesorgente* è il nome assoluto di un file presente nel file system della macchina Server. Il processo Server deve inviare il file *nomefilesorgente* al Client che lo scrive nel direttorio corrente con nome *nomefiledestinazione*.

Sul Client, la scrittura del file nel direttorio corrente deve essere eseguita **solo se** in tale direttorio non è presente un file con lo stesso nome, per evitare di sovrascriverlo.

Protocollo applicativo



rcp lato Client

```
File theFile = new File(args[3]);
// ... controllo esistenza file ...
try {
    char risposta; FileInputStream fileIn;
    Socket s = new Socket(args[0], Integer.parseInt(args[1]));
    DataInputStream in = new DataInputStream(s.getInputStream());
    BufferedWriter out = new BufferedWriter(
        new OutputStreamWriter(s.getOutputStream(), "UTF-8"));

    out.write(args[2]); out.newLine(); out.flush();
    if (in.readChar() == 'S') {
        byte buffer[] = new byte[BUFDIM];
        int bytesRead;
        FileOutputStream fileOut = new FileOutputStream(theFile);
        while ((bytesRead = in.read(buffer, 0, BUFDIM)) != -1) {
            fileOut.write(buffer, 0, bytesRead);
        }
        fileOut.close();
    }
    s.close();
} catch (IOException e) { ...
```

Perché per l'input da socket uso `DataInputStream`? E' una soluzione compatibile con applicazioni non sviluppate in Java?

C/S in Java: le socket - 21

rcp lato Server (caso iterativo)

```
...
try {
    ServerSocket ss = new ServerSocket(Integer.parseInt(args[0]));
    while (true) {
        Socket s = ss.accept();
        BufferedReader in = new BufferedReader(
            new InputStreamReader(s.getInputStream(), "UTF-8"));
        DataOutputStream out = new DataOutputStream(s.getOutputStream());
        File file = new File(in.readLine());
        if (file.exists()) {
            out.writeChar('S'); out.flush();
            byte buffer[] = new byte[BUFDIM];
            int bytesRead;
            FileInputStream fileIn = new FileInputStream(file);
            while ((bytesRead = fileIn.read(buffer, 0, BUFDIM)) != -1) {
                out.write(buffer, 0, bytesRead);
            }
            fileIn.close();
            out.flush();
        } else {
            out.writeChar('N'); out.flush();
        }
        s.close();
    }
} catch (IOException e) {
```

...

C/S in Java: le socket - 22

Opzioni delle Socket

Il comportamento di default delle socket Java può essere modificato agendo sulle *opzioni* delle socket, utilizzando i metodi:

```
setTcpNoDelay(boolean on) throws ...
```

il pacchetto è inviato immediatamente, senza bufferizzazioni

```
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 secondi)

```
setSoTimeout(int timeout) throws ...
```

la lettura da socket è normalmente bloccante. Questa opzione definisce un timeout, trascorso il quale il thread si sblocca (ma viene lanciata una eccezione da gestire)

Comunicazione senza connessione: Java Socket DATAGRAM

Le socket DATAGRAM permettono a due thread di scambiarsi **messaggi senza stabilire una connessione** tra loro.

La comunicazione con le socket DATAGRAM:

- **non è affidabile (!)**, cioè si può verificare la perdita di messaggi (per esempio, dovuti a problemi di rete);
- diversi messaggi tra una stessa coppia Client e Server possono essere consegnati **non in ordine**, in quanto può capitare che seguano strade diverse.

Queste caratteristiche derivano dal protocollo di comunicazione UDP su cui si appoggiano (in modo analogo, si ricordi che l'affidabilità e i messaggi in sequenza visti per le STREAM discendono dal protocollo TCP).

Vantaggi/svantaggi di STREAM vs DATAGRAM (e TCP vs UDP)?

Vi è un solo tipo di socket DATAGRAM sia per il Client sia per il Server:

- classe `DatagramSocket` (La classe **`java.net.DatagramSocket`**)

La classe `java.net.DatagramSocket`

Classe socket DATAGRAM senza connessione:

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

Costruttore:

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

Crea la socket DATAGRAM (UDP) **legata localmente** (a indirizzo IP e porta)

Per comunicare, uso di **send** e **receive**, esempio:

```
objDatagramSock.send(DatagramPacket);  
objDatagramSock.receive(DatagramPacket);
```

Metodi per recuperare varie informazioni:

```
objDatagramSock.getLocalPort();  
objDatagramSock.getLocalAddress();
```

La socket è legata SOLO localmente. Come/dove viene inviato il pacchetto?

C/S in Java: le socket - 25

La classe `java.net.DatagramPacket`

`DatagramPacket` è la classe per preparare i datagrammi da inviare sulla socket.

Una istanza **datagramma** specifica un array di byte su cui scrivere (o da cui leggere) e contiene le indicazioni per la comunicazione.

Costruttore nel caso di **INVIO** messaggio:

```
DatagramPacket(byte[] buf, int length, InetAddress address, int port);
```

Il buffer `buf` è un'area in cui mettere i dati, l'indirizzo è quello del destinatario.
`InetAddress` è la classe per gli indirizzi IP.

Costruttore nel caso di **RICEZIONE** messaggio:

```
DatagramPacket(byte[] buf, int length);
```

Il buffer `buf` è un'area preparata per ricevere dati.

La classe `DatagramPacket` offre metodi per estrarre o settare le informazioni:

```
getAddress(), getPort() (restituisce address e port di macchina remota)  
setAddress(InetAddress addr), setPort(int port)  
getData(), setData(byte[] buf), etc.
```

C/S in Java: le socket - 26

La classe `java.net.InetAddress`

La classe `InetAddress` permette di rappresentare gli indirizzi IP delle macchine, sia per socket unicast che multicast, sia per indirizzi IPv4 che IPv6.

La classe fornisce i metodi per **risolvere i nomi**, cioè passare dal nome simbolico all'indirizzo IP (e viceversa).

Esempio:

```
public static InetAddress getByName(String host)
    throws UnknownHostException
```

Restituisce l'indirizzo IP corrispondente al nome logico di host fornito. (Corrisponde alla funzione di libreria `getaddrinfo` in Unix/C.)

Client DATAGRAM: schema

Creazione socket:

```
dgramSocket = new DatagramSocket();
```

Interazione da console con l'utente:

```
BufferedReader stdIn = new BufferedReader(
    new InputStreamReader(System.in));
System.out.print("Domanda... ");
String richiesta = stdIn.readLine();
```

Creazione del pacchetto di richiesta con le informazioni inserite dall'utente:

```
sendBuff = richiesta.getBytes("UTF-8");
packetOUT = new DatagramPacket(sendBuff, sendBuff.length,
    serverAddr, serverPort);
```

Invio del pacchetto al server:

```
dgramSocket.send(packetOUT);
```

Attesa del pacchetto di risposta:

```
packetIN = new DatagramPacket(buf, buf.length);
dgramSocket.receive(packetIN);
```

Server DATAGRAM: schema

Creazione socket:

```
dgramSocket = new DatagramSocket(PORT);
```

Attesa del pacchetto di richiesta:

```
packet = new DatagramPacket(buf, buf.length);  
dgramSocket.receive(packet);
```

Estrazione delle informazioni dal pacchetto ricevuto:

```
byte[] requestBuf =  
    Arrays.copyOf(packet.getData(),  
        packet.getLength());  
String request = new String(requestBuf,  
    "UTF-8");
```

...

Creazione del pacchetto di risposta con la linea richiesta:

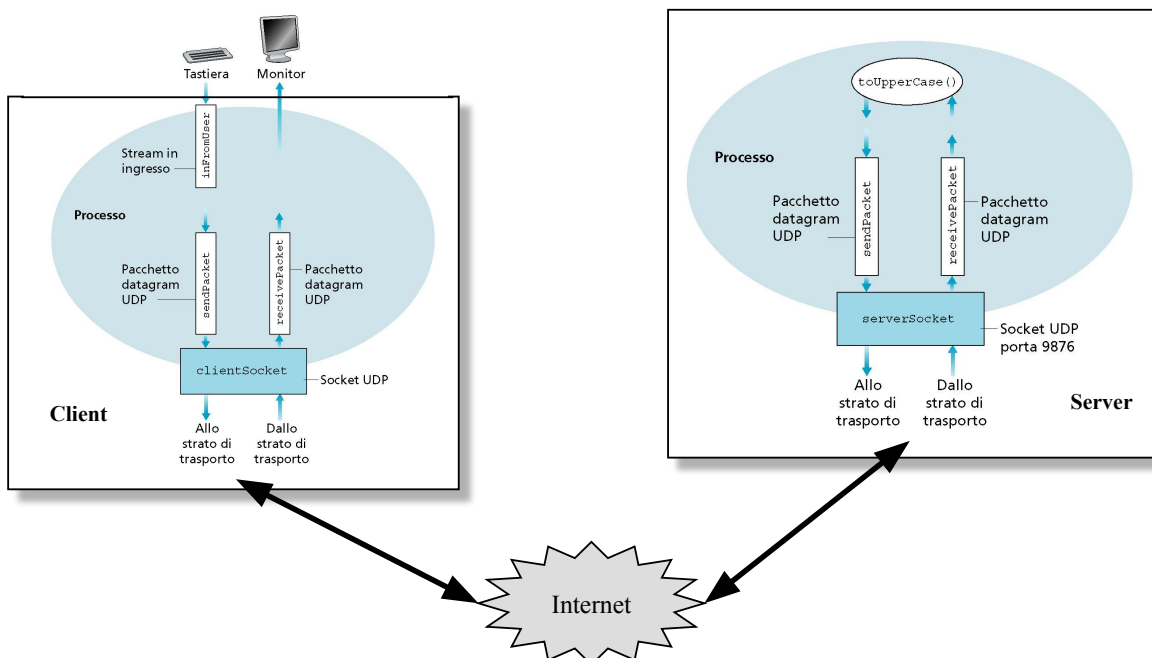
...

Invio del pacchetto al client:

```
dgramSocket.send(packet);
```

C/S in Java: le socket - 29

Esempio Client/Server DATAGRAM



C/S in Java: le socket - 30

```

...
BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));
DatagramSocket clientSocket = new DatagramSocket();

InetAddress serverAddr = InetAddress.getByName("server.unife.it");

byte[] receiveData = new byte[1024];

System.out.println("Inizio Client, inserire stringa:");
String sentence = inFromUser.readLine();
byte sendData = sentence.getBytes("UTF-8");

DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length,
                                              serverAddr, 9876);

clientSocket.send(sendPacket);

DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
clientSocket.receive(receivePacket);

String modifiedSentence = new String(receivePacket.getData(), "UTF-8");

System.out.println("FROM SERVER:" + modifiedSentence);

clientSocket.close();
...

```

Client

C/S in Java: le socket - 31

```

...
DatagramSocket serverSocket = new DatagramSocket(9876);
byte[] receiveData = new byte[1024];

while(true) {
    System.out.println("Server attende richiesta");

    DatagramPacket receivePacket =
        new DatagramPacket(receiveData, receiveData.length);
    serverSocket.receive(receivePacket);
    byte[] requestBuf = Arrays.copyOf(packet.getData(), packet.getLength());
    String sentence = new String(requestBuf, "UTF-8");
    System.out.println("Richiesta ricevuta: " + sentence);

    InetAddress IPAddress = receivePacket.getAddress();
    int port = receivePacket.getPort();

    String uppercasedSentence = sentence.toUpperCase();
    byte[] sendData = uppercasedSentence.getBytes("UTF-8");

    DatagramPacket sendPacket =
        new DatagramPacket(sendData, sendData.length, IPAddress, port);

    serverSocket.send(sendPacket);
} ...

```

Server

C/S in Java: le socket - 32

Note conclusive sulle socket DATAGRAM

Si notino alcuni aspetti importanti che possono verificarsi in applicazioni che fanno uso di socket DATAGRAM:

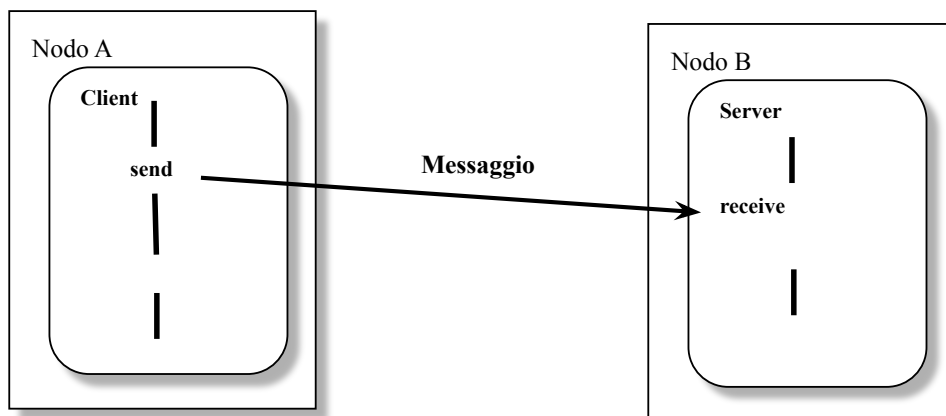
- **La comunicazione via Socket DATAGRAM non è affidabile**, per cui in caso di perdita del messaggio del Client o della risposta del Server, il Client può rimanere bloccato in attesa indefinita della risposta (utilizzo di timeout?)
- **Possibile blocco del Client** anche nel caso di invio di una richiesta a un Server non attivo (errori nel collegamento al server notificati solo sulle socket connesse)
- **Le DATAGRAM sono appoggiate sul protocollo UDP che non ha flow control**, se il Server riceve troppi datagrammi per le sue capacità di elaborazione, questi vengono scartati, senza nessuna notifica ai Client (c'è una coda per ogni socket, uso di `setReceiveBufferSize()` per modificarne la lunghezza)

C/S in Java: le socket - 33

Socket e malfunzionamenti

Le socket STREAM e DATAGRAM si comportano in maniera molto diversa a fronte di **malfunzionamenti**.

Questo deriva dalle differenze tra i protocolli sottostanti (rispettivamente TCP e UDP)



C/S in Java: le socket - 34

Semantica may-be

Le socket DATAGRAM presentano una semantica di comunicazione di tipo **may-be** ("forse"). Questo perché il protocollo di supporto UDP **non mette in pratica nessuna azione per fronteggiare il caso di guasto**. Un messaggio DATAGRAM viene semplicemente inviato, senza chiedere di essere avvisati della ricezione. Quindi il mittente non può desumere cosa sia successo del messaggio spedito.

Le socket DATAGRAM non sono quindi affidabili. Forniscono buone prestazioni, ma possono portare problemi a livello di applicazione.

Semantica at-most-once

Nel caso delle socket STREAM, la comunicazione è affidabile e la semantica ottenuta è detta **at-most-once** (al massimo una volta). Il supporto di comunicazione fa di tutto per garantire la consegna dei messaggi e ogni messaggio viene consegnato al massimo una volta.

Questo è reso possibile dal sottostante protocollo **TCP** che chiede conferma di ogni messaggio inviato e ritrasmette quelli persi. Inoltre il TCP numera i messaggi e quindi si accorge di messaggi arrivati più volte (e li scarta, senza consegnare più volte lo stesso messaggio all'applicazione).

Affidabilità a fronte di peggiori prestazioni.

Quale tipo di Socket utilizzare?

La **scelta del tipo di socket** viene fatta sulla base delle molte differenze tra le socket STREAM e quelle DATAGRAM:

Servizi che richiedono una connessione \longleftrightarrow servizi connectionless
Socket STREAM sono **affidabili**, DATAGRAM no.

Prestazioni STREAM (costo di mantenere una connessione logica) inferiori alle DATAGRAM.

STREAM han **semantica** at-most-once, DATAGRAM may-be.

Ordinamento messaggi (preservato in STREAM, non in DATAGRAM)

Per fare del **broadcast/multicast** più indicate le DATAGRAM (altrimenti richiesta apertura di molte connessioni contemporaneamente).

ATTENZIONE: queste differenze tra le socket derivano dalle differenze dei protocolli sottostanti (**STREAM su TCP, DATAGRAM su UDP**)

C/S in Java: le socket - 37

Le Socket Multicast

Invio di uno stesso messaggio a molti riceventi diversi (un gruppo) con socket **multicast**.

Le socket multicast *estendono* le datagram, aggiungendo funzionalità di gestione gruppi.

Un gruppo di multicast è specificato da indirizzi classe D (da 224.0.0.0 a 239.255.255.255)

Fase di **creazione della socket multicast**:

```
MulticastSocket ms = new MulticastSocket(7777); (numero di porta)
```

Definizione del gruppo (IP di classe D):

```
InetAddress gruppo = InetAddress.getByName("229.1.2.3");
```

Operazioni di ingresso/uscita dal gruppo:

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

Le Socket Multicast richiedono esplicito supporto dal sistema di routing IP (spesso assente) e sono sostanzialmente deprecate. Vengono qui presentate solo per motivi storici.

C/S in Java: le socket - 38

Le Socket Multicast

Spedizione e ricezione (stesse operazioni viste su datagram socket):

```
String msg = "ciao";
byte[] buf = msg.getBytes("UTF-8");
DatagramPacket packet=new DatagramPacket(buf,buf.length,gruppo,7777);
ms.send(packet);

byte[] buf = new byte[DIM];
DatagramPacket recvpacket = new DatagramPacket(buf,buf.length);
ms.receive(recvpacket);
```

Un messaggio inviato a una socket multicast viene ricevuto da tutto il gruppo dei sottoscrittori.

Un messaggio può essere inviato anche da una normale socket datagram.

Una stessa socket multicast può appartenere contemporaneamente a molti differenti gruppi.

C/S in Java: le socket - 39

Le Socket Unix

Nei sistemi Unix esiste un tipo di socket (dette “Unix”, appunto) che rappresenta una forma di IPC tra due processi nella stessa macchina simile alle FIFO (pipe bidirezionali con nome).

Rispetto alle connessioni TCP/IP su interfaccia loopback (ottimizzata per massimizzare le prestazioni), le socket di dominio Unix presentano alcuni vantaggi:

- ☐ Poiché possono essere utilizzati solo per la comunicazione sullo stesso host, l'apertura al posto di un socket TCP/IP non comporta il rischio di accettare connessioni remote.
- ☐ Il controllo dell'accesso viene applicato con meccanismi basati su file, dettagliati, ben compresi e applicati dal sistema operativo.
- ☐ Le socket Unix hanno **tempi di configurazione più rapidi e una maggiore velocità di trasmissione dei dati** rispetto alle connessioni di loopback TCP/IP.
- ☐ Le socket Unix **si possono usare per la comunicazione tra container sullo stesso sistema** (purché si creino su un volume condiviso).
- ☐ Supportano passaggio di file descriptor (si veda “Unix Network Programming”, par. 15.7).

Nonostante il nome non sono supportate solo su sistemi operativi basati su Unix (Linux, MacOS) ma anche su sistemi Microsoft (a partire dal 2017).

Sempre più utilizzate. **Introdotte in Java a partire dalla versione 16 (JEP 380).**

C/S in Java: le socket - 40

Le Socket Unix

Le socket Unix hanno tre tipologie:

- ❑ Stream: connection-oriented, stream-based, affidabili (semantica TCP)
- ❑ Datagram: connectionless, preservano i confini dei messaggi, e al contrario delle socket Datagram di dominio Internet **sono affidabili e consegnano i messaggi rispettando l'ordine di trasmissione**
- ❑ Sequential packet (SEQPACKET): connection-oriented, preservano i confini dei messaggi, sono affidabili e consegnano i messaggi rispettando l'ordine di trasmissione (semantica SCTP)

C/S in Java: le socket - 41

Dalla versione 10, Java
supporta Local
Variable Type
Inference (LVTI)

Esempio Server con Socket Unix di tipo Stream

Percorso della
socket nel file
system

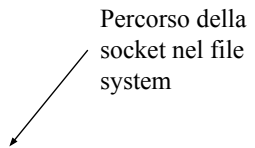
```
...  
var address = UnixDomainSocketAddress.of("/var/myapp/socket");  
try (var serverChannel = ServerSocketChannel.open(UNIX)) {  
    serverChannel.bind(address);  
    try (var clientChannel = serverChannel.accept()) {  
        ByteBuffer buf = ByteBuffer.allocate(1024);  
        clientChannel.read(buf);  
        buf.flip();  
        System.out.printf("Read %d bytes\n", buf.remaining());  
    }  
} finally {  
    Files.deleteIfExists(address.getPath());  
}  
...
```

Le Socket Unix in Java fanno uso di una nuova API, nota come NIO (per "New I/O"), introdotta in Java 4 per supportare I/O asincrono e I/O multiplexing. Concetti chiave: (Selector,) Buffer e Channel.

C/S in Java: le socket - 42

Esempio Client con Socket Unix di tipo Steam

```
...  
var address = UnixDomainSocketAddress.of("/var/myapp/socket");  
try (var clientChannel = SocketChannel.open(address)) {  
    var buf = ByteBuffer.wrap("Hello world".getBytes("UTF-8"));  
    clientChannel.write(buf);  
}  
...
```



Percorso della
socket nel file
system