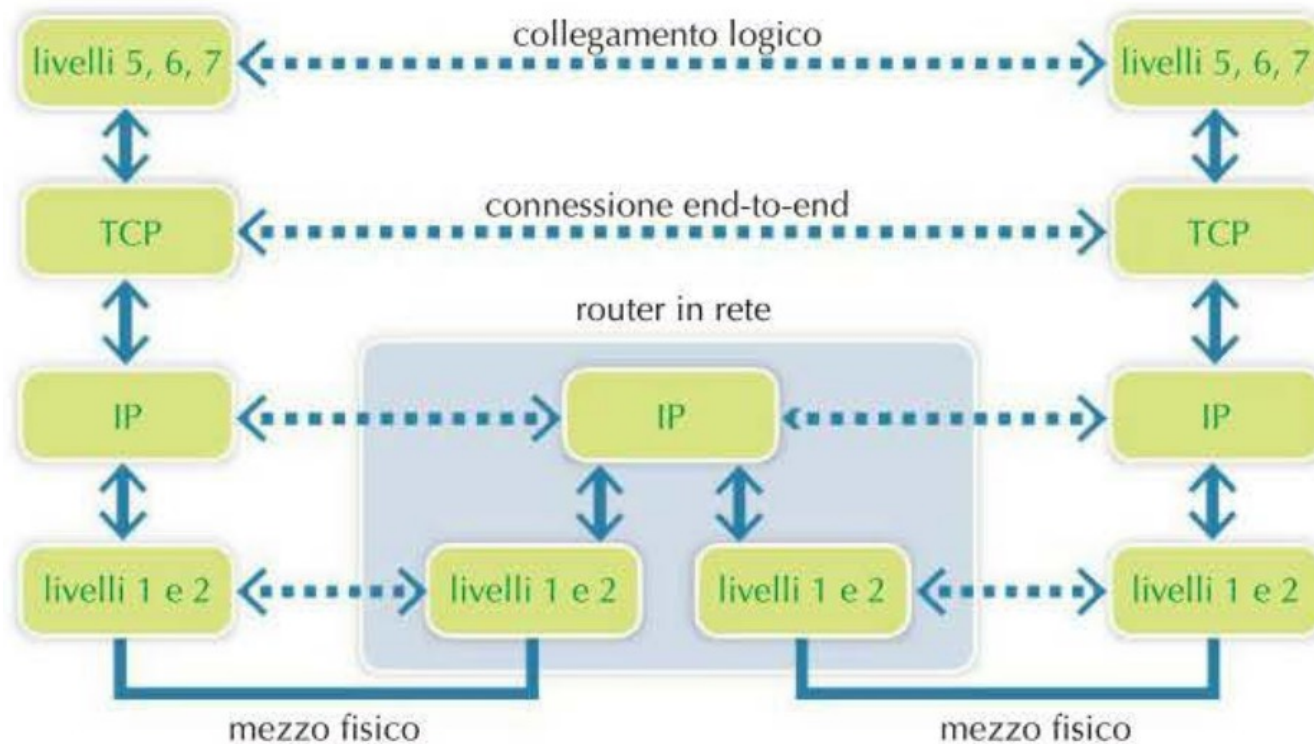


Socket, API e Java

Unità 2 – lezione 1 e lezione 2



ISO-OSI	TCP-IP	
Applicazione	Applicazione	HTTP, FTP SMTP, POP3 Telnet, SSL
Presentazione		
Sessione		
Trasporto	Trasporto	TCP, UDP
Rete	Rete	IP
Collegamento	Fisico (Host)	
Fisico		

Modello OSI		Modello TCP/IP
7 Applicazione	DHCP, DNS, FTP, HTTP, HTTPS, POP, SMTP, SSH, DNS, POP3, SNMP, etc...	Applicazione
6 Presentazione		Trasporto
5 Sessione		
4 Trasporto	TCP UDP	Internet
3 Rete	IP Address: IPv4, IPv6	Rete fisica
2 Collegamento	MAC Address	
1 Fisico	Ethernet cable, fibre, wireless, coax, etc...	

Connessione Logica a Livello di Trasporto

Una connessione logica end-to-end tra due processi su una rete è identificata da una quintupla composta da cinque elementi fondamentali: <Protocollo, IP_sorgente, Porta_sorgente, IP_destinazione, Porta_destinazione>

La quintupla definita viene in realtà costituita a partire da due elementi simmetrici, un'associazione locale ed una remota:

- Mittente: identificato da <IP_sorgente, Porta_sorgente>
- Destinatario: identificato da <IP_destinazione, Porta_destinazione>

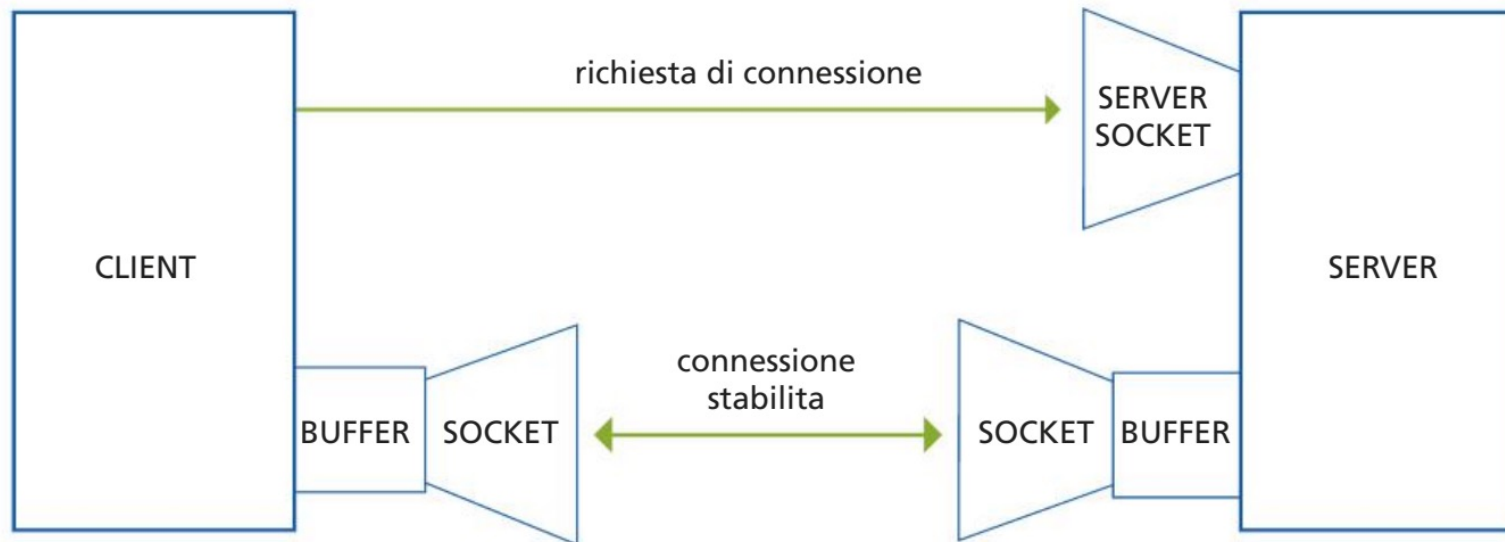
Il socket è l'astrazione di un canale di comunicazione tra processi distribuiti in rete. Data una connessione logica end-to-end, ciascuna di queste parti è detta socket

- protocollo, Ind. locale, porta locale (TCP,123.23.4.221,1500) → socket client
- protocollo, Ind. remoto, porta remota (TCP ,234.151.124.2,4000) → socket server

Nota:

- protocolli differente possono usare la stessa porta
- Es. 1040 per TCP diverso da 1040 per UDP

Socket e API



Api e Socket

Si presentano sotto la forma di un'API (Application Programming Interface), cioè un insieme di funzioni che le applicazioni possono invocare per ricevere il servizio desiderato.

Rappresentano una estensione delle API di UNIX per la gestione dell'I/O su periferica standard (files su disco, stampanti, etc.).

Questa API è poi divenuta uno standard de facto, ed oggi è diffusa nell'ambito di tutti i maggiori sistemi operativi (Linux, FreeBSD, Solaris, Windows... etc.)

Vedere Berkeley Sockets: https://en.wikipedia.org/wiki/Berkeley_sockets

Socket API

- Same API used for Streams and Datagrams

		Primitive	Meaning
Only needed for Streams	{	SOCKET	Create a new communication endpoint
		BIND	Associate a local address (port) with a socket
		LISTEN	Announce willingness to accept connections
		ACCEPT	Passively establish an incoming connection
		CONNECT	Actively attempt to establish a connection
To/From forms for Datagrams	{	SEND(TO)	Send some data over the socket
		RECEIVE(FROM)	Receive some data over the socket
		CLOSE	Release the socket

Well-known ports - server

- Server spesso usano le well-known ports da 0 a 1024
- Client spesso usa porte scelte da OS e oltre 1024

Port	Protocol	Use
20, 21	FTP	File transfer
22	SSH	Remote login, replacement for Telnet
25	SMTP	Email
80	HTTP	World Wide Web
110	POP-3	Remote email access
143	IMAP	Remote email access
443	HTTPS	Secure Web (HTTP over SSL/TLS)
543	RTSP	Media player control
631	IPP	Printer sharing

Interazione tra Applicazione e SO

L'applicazione chiede al sistema operativo di utilizzare i servizi di rete


Il sistema operativo crea una socket e la restituisce all'applicazione

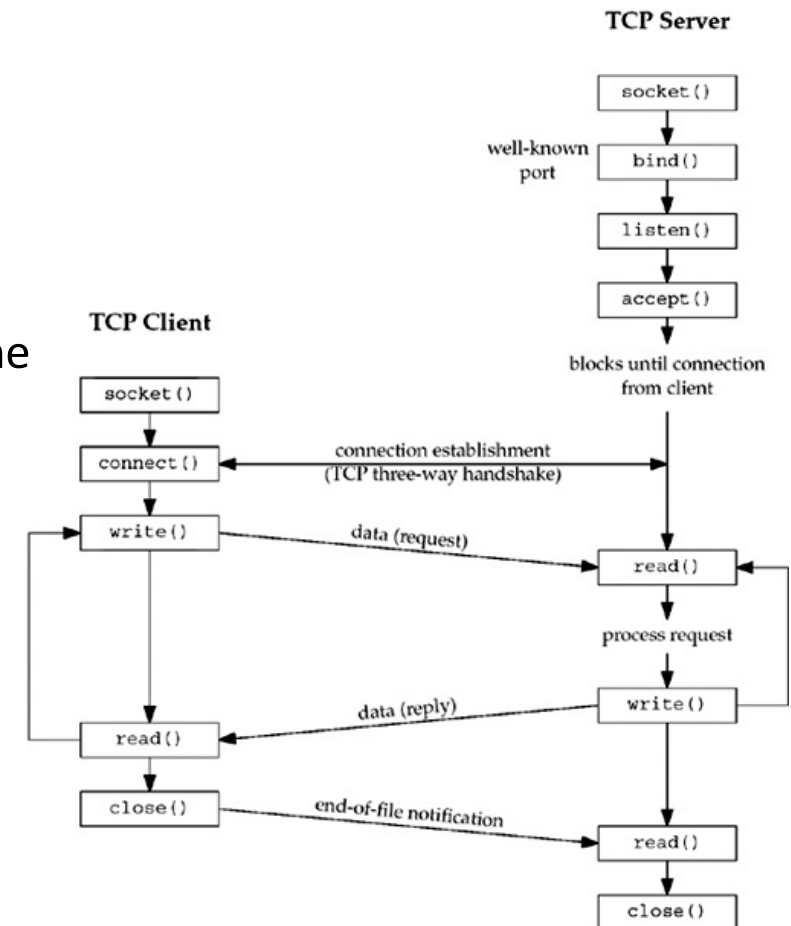
L'applicazione utilizza la socket

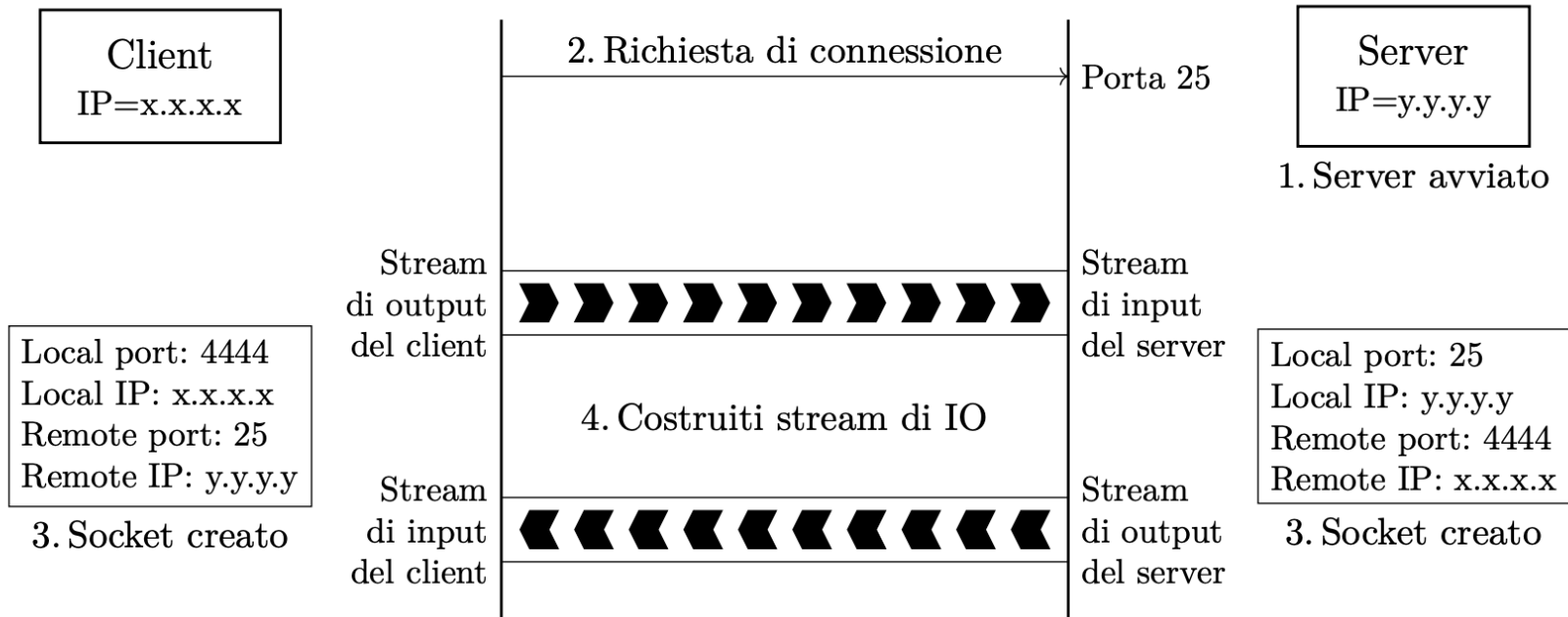
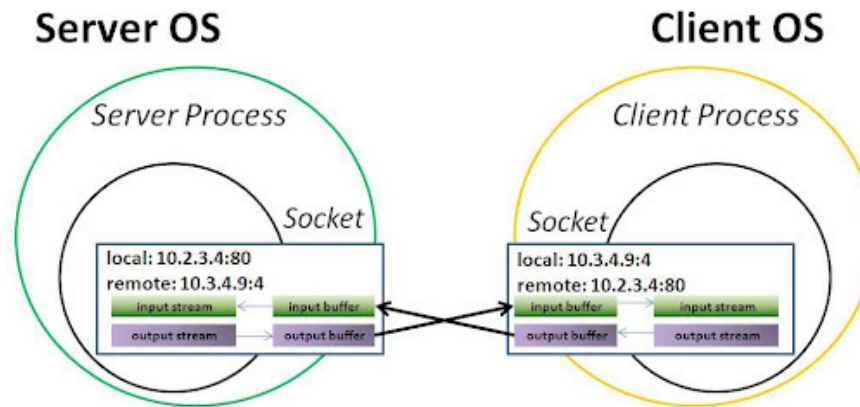
Open, Read, Write, Close.

L'applicazione chiude la socket e la restituisce al sistema operativo

Creazione comunicazione end to end con i socket

- Lato server
 - 1. Creazione del socket
 - 2. Bind ad una porta
 - 3. Listen, predisposizione a ricevere sulla porta
 - 4. Accept, blocca il server in attesa di una connessione
 - 5. Lettura - scrittura dei dati
 - 6. Chiusura
-  Lato client
 - 1. Creazione del socket
 - 2. Richiesta di connessione
 - 3. Lettura - scrittura dei dati
 - 4. Chiusura





Socket in Java

- L'interfaccia deriva da quella standard di Berkeley, introdotta nel 1981
- E' organizzata in maniera da soddisfare i principi dell'Object Orienting Programming
- Contiene classi per la manipolazione degli indirizzi e la creazione di socket lato client o server
- Implementazione nel package `java.net`

Documentazione ufficiale

- Studiare I metodi che usiamo in laboratorio

- Vedere documentazione ufficiale

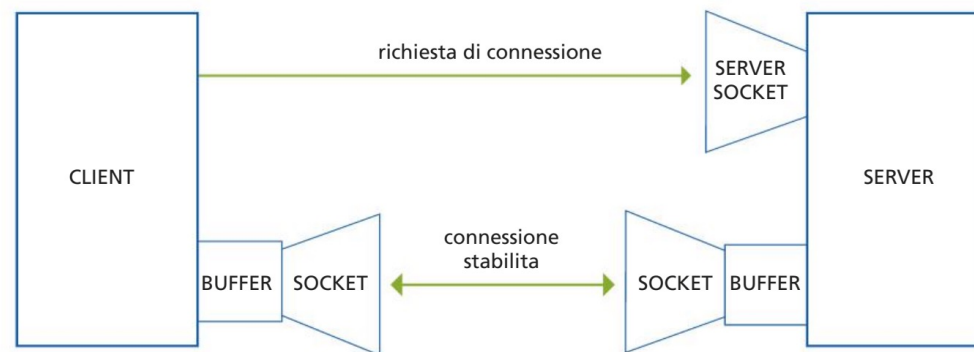
<https://docs.oracle.com/en/java/javase/22/docs/api/index.html>

Pacchetto java.net

<https://docs.oracle.com/javase/8/docs/api/java/net/package-summary.html>

- Socket: <https://docs.oracle.com/javase/8/docs/api/java/net/Socket.html>
- ServerSocket:
<https://docs.oracle.com/javase/8/docs/api/java/net/ServerSocket.html>

- L'API Java Socket fornisce le classi e i metodi necessari per stabilire connessioni di rete tra client e server. Le due principali classi coinvolte sono:
 - `java.net.Socket`: Utilizzata dal client per connettersi a un server.
 - `java.net.ServerSocket`: Utilizzata dal server per ascoltare le richieste di connessione dai client.
- **Principali Metodi**
- **Socket:**
 - `Socket(String host, int port)`: Crea un socket e si connette al server specificato.
 - `getInputStream()` e `getOutputStream()`: Ottengono i flussi di input e output per la comunicazione.
- **ServerSocket:**
 - `ServerSocket(int port)`: Crea un server socket che ascolta sulla porta specificata.
 - `accept()`: Blocca e attende una connessione da un client. Restituisce un Socket per comunicare con il client.



Comportamento Dopo che il Client Contatta il Server

Server in Attesa (accept() Bloccato):

Quando un server avvia il metodo `accept()` su un oggetto `ServerSocket`, il thread che esegue `accept()` è bloccato e in attesa di una connessione in ingresso. Il server non può eseguire altre operazioni finché non viene accettata una connessione.

Client Stabilisce Connessione:

Il client crea un `Socket` e chiama il costruttore `Socket(String host, int port)`, specificando l'indirizzo IP del server e la porta a cui connettersi.

Il client invia una richiesta di connessione al server.

Connessione Stabilita:

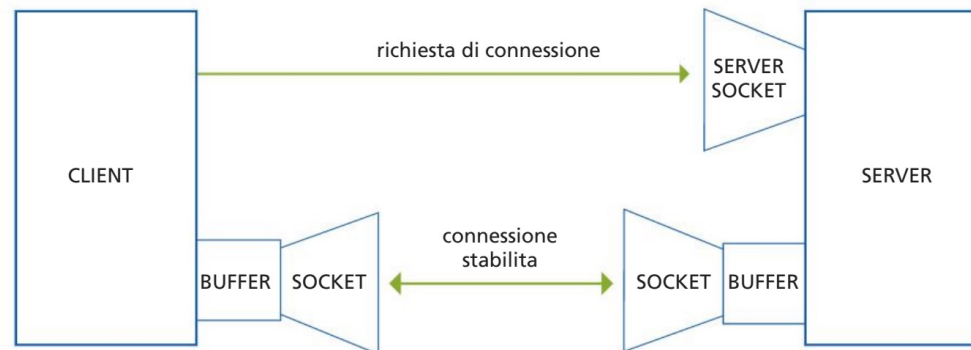
Quando il server accetta la connessione, il metodo `accept()` restituisce un nuovo `Socket` che rappresenta la connessione con il client. Questo nuovo `Socket` viene utilizzato per la comunicazione con il client.

Il server può ora utilizzare il flusso di input e output del nuovo `Socket` per scambiare dati con il client.

Continua l'Esecuzione:

Dopo aver accettato la connessione, il server può continuare a chiamare `accept()` per accettare ulteriori connessioni, se necessario. Ogni connessione verrà gestita da un nuovo `Socket`.

```
Socket socket = new  
Socket("server_address", 1234);  
// Usa socket per comunicare con il  
server
```



```
ServerSocket serverSocket = new  
ServerSocket(1234);  
Socket clientSocket = serverSocket.accept();  
// Blocca finché non arriva una connessione  
// Usa clientSocket per comunicare con il client
```

InputStream e OutputStream

- InputStream e OutputStream sono classi base per la lettura e scrittura di dati binari. Operano direttamente sui dati come flusso di byte.
 - <https://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html>
 - <https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html>
- **InputStream:**
 - **Metodo** read(): Legge un byte alla volta (o un array di byte) dal flusso di input.
 - **Operazione a byte:** Non considera l'encoding dei caratteri e lavora solo con dati grezzi.
 - **Esempio:** Lettura di file binari, dati di rete, o byte da altre fonti.
- **OutputStream:**
 - **Metodo** write(int b): Scrive un byte alla volta (o un array di byte) nel flusso di output.
 - **Operazione a byte:** Gestisce i dati grezzi senza considerare l'encoding dei caratteri.
 - **Esempio:** Scrittura di file binari, invio di dati su una rete, o scrittura in dispositivi hardware.

- `BufferedReader` e `PrintWriter` forniscono un'interfaccia di livello più alto, operando a livello di carattere e stringa.
 - Gestiscono la lettura e la scrittura di dati a livello di caratteri e stringhe, con buffering per migliorare le prestazioni.
- **BufferedReader:**
 - **Metodo** `readLine()`: Legge una riga di testo come una stringa.
 - **Gestione caratteri**: Converte i byte in caratteri secondo l'encoding specificato (tipicamente UTF-8 o un altro charset).
 - **Esempio**: Lettura di file di testo, input da console.
- **PrintWriter:**
 - **Metodo** `println(String s)`: Scrive una stringa seguita da una nuova riga.
 - **Gestione caratteri**: Gestisce il formato del testo e il buffering per migliorare le performance.
 - **Esempio**: Scrittura di file di testo, output formattato su console o file.

CLIENT

```
Socket client = new Socket("localhost", 5005);

// Stampa e scrittura della stringa
PrintWriter outServer = new PrintWriter(client.getOutputStream(),
true);

// Creazione del BufferedReader per leggere la risposta dal server
BufferedReader inServer = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

// Invia il messaggio al server
String message = "Ciao, server!";
out.println(message);
System.out.println("Messaggio inviato al server: " + message);

// Leggi la risposta del server
String response = in.readLine();
System.out.println("Risposta dal server: " + response);

// Chiudi le risorse
out.close();
in.close();
socket.close();
```

SERVER

```
Socket client = serverSocket.accept();

// Creazione del BufferedReader per leggere il messaggio del client
BufferedReader inClient = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

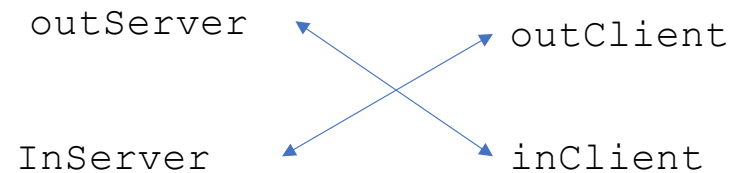
// Creazione del PrintWriter per inviare una risposta al client
PrintWriter outClient = new PrintWriter(clientSocket.getOutputStream(),
true);

// Invia una risposta al client
String response = "Ciao, client!";
out.println(response);
System.out.println("Risposta inviata al client: " + response);

// Leggi il messaggio dal client
String message = in.readLine();
System.out.println("Messaggio ricevuto dal client: " + message);

// Invia una risposta al client
String response = "Ciao, client!";
out.println(response);
System.out.println("Risposta inviata al client: " + response);

// Chiudi le risorse
in.close();
out.close();
clientSocket.close();
serverSocket.close();
```



CLIENT

```
Socket client = new Socket("localhost", 5005);

// Stampa e scrittura della stringa
PrintWriter outServer = new PrintWriter(client.getOutputStream(),
true);

// Creazione del BufferedReader per leggere la risposta dal server
BufferedReader inServer = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

// Invia il messaggio al server
String message = "Ciao, server!";
out.println(message);
System.out.println("Messaggio inviato al server: " + message);

// Leggi la risposta del server
String response = in.readLine();
System.out.println("Risposta dal server: " + response);

// Chiudi le risorse
out.close();
in.close();
socket.close();
```

SERVER

```
Socket client = serverSocket.accept();

// Creazione del BufferedReader per leggere il messaggio del client
BufferedReader inClient = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

// Creazione del PrintWriter per inviare una risposta al client
PrintWriter outClient = new PrintWriter(clientSocket.getOutputStream(),
true);

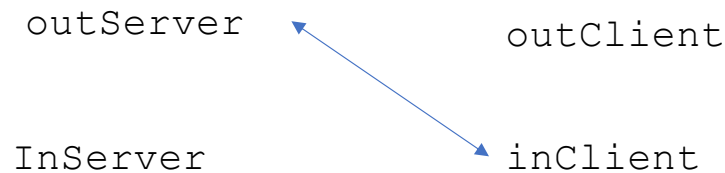
// Invia una risposta al client
String response = "Ciao, client!";
out.println(response);
System.out.println("Risposta inviata al client: " + response);

// Leggi il messaggio dal client
String message = in.readLine();
System.out.println("Messaggio ricevuto dal client: " + message);

// Invia una risposta al client
String response = "Ciao, client!";
out.println(response);
System.out.println("Risposta inviata al client: " + response);

// Chiudi le risorse
in.close();
out.close();
clientSocket.close();
serverSocket.close();
```

*Il client scrive su **outServer** e il server riceve su **inClient***



CLIENT

```
Socket client = new Socket("localhost", 5005);

// Stampa e scrittura della stringa
PrintWriter outServer = new PrintWriter(client.getOutputStream(),
true);

// Creazione del BufferedReader per leggere la risposta dal server
BufferedReader inServer = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

// Invia il messaggio al server
String message = "Ciao, server!";
out.println(message);
System.out.println("Messaggio inviato al server: " + message);

// Leggi la risposta del server
String response = in.readLine();
System.out.println("Risposta dal server: " + response);

// Chiudi le risorse
out.close();
in.close();
socket.close();
```

SERVER

```
Socket client = serverSocket.accept();

// Creazione del BufferedReader per leggere il messaggio del client
BufferedReader inClient = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

// Creazione del PrintWriter per inviare una risposta al client
PrintWriter outClient = new PrintWriter(clientSocket.getOutputStream(),
true);

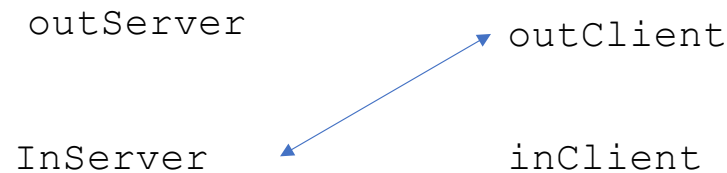
// Invia una risposta al client
String response = "Ciao, client!";
out.println(response);
System.out.println("Risposta inviata al client: " + response);

// Leggi il messaggio dal client
String message = in.readLine();
System.out.println("Messaggio ricevuto dal client: " + message);

// Invia una risposta al client
String response = "Ciao, client!";
out.println(response);
System.out.println("Risposta inviata al client: " + response);

// Chiudi le risorse
in.close();
out.close();
clientSocket.close();
serverSocket.close();
```

*Il server scrive su **outClient** e il client riceve su **inServer***



- FARE SLIDES JAVA SOCKET DA QUI
http://www.cs.unibo.it/~sangio/SO_currentAA/Luc_SO/Schirinzi/Socket.pdf
- Vedere libro zanichelli Mondadori
- Vedere documentazione ufficiale