UNIVERSITÀ DEGLI STUDI DI BRESCIA · UNIBS.it

AA 2026/27

PAJC · PROGRAMMAZIONE AVANZATA IN JAVA E C

L06

Comunicazione in rete

Socket e TCP, un protocollo di testo, un server per molti client con i thread virtuali, `HttpClient`



prof. Massimiliano Redolfi

Università degli Studi di Brescia · Ingegneria Informatica

Oggi

1. **La rete vista da un programma**: indirizzi, porte, i quattro livelli, TCP e UDP
2. **Il primo dialogo**: `ServerSocket` e `accept`, `Socket`, gli stream, `nc`, che cosa succede davvero
3. **Il protocollo**: a passo o libero, il client full-duplex, WhatsPAJC come **tabella** di comandi
4. **Un server per molti**: la `Sessione`, un thread virtuale per client, il `Registro` condiviso, fermarsi
5. **Oltre il filo**: la finestra, HTTP a mano, `HttpClient`

01

La rete vista da un programma



Una chat per l’aula

WhatsPAJC: ognuno sceglie un nome, vede chi c’è, scrive a tutti o a uno solo. Il server gira sul computer di uno, gli altri si collegano dalla rete dell’aula. Il primo tentativo, con gli strumenti che si hanno:

- il server risponde a un client; il **secondo** si collega e resta appeso, senza errori
- il client manda una riga e aspetta la risposta: il messaggio di Giulia arriva a Marco solo quando Marco scrive qualcosa
- un accento che arriva sbagliato da Windows a Mac
- la finestra congela su «Connetti»; spostata la lettura in un thread, si corrompe

> Tre domande: **come si parlano** due programmi, **come si mettono d'accordo** su che cosa dirsi, **come se ne servono molti** senza far aspettare nessuno.

Quattro cose da sapere

- **indirizzo IP**: identifica una macchina (`192.168.1.10`; `127.0.0.1` è sé stessa, il *loopback*). Un **nome** (`localhost`, `www.unibs.it`) lo traduce il DNS: `InetAddress.getByName`, `UnknownHostException` se non esiste
- **porta**: un numero da 0 a 65535 che identifica un programma su quella macchina. Indirizzo e porta insieme, come un palazzo e un interno; sotto 1024 riservate, noi useremo la `1234`
- **TCP**: una connessione, e dentro un flusso di byte che arrivano **tutti, in ordine, una volta sola**: un tubo. **UDP**: singoli messaggi senza garanzie: una lettera
- **client e server** sono ruoli solo all'inizio: uno aspetta su una porta, l'altro bussa. Poi la connessione è simmetrica

I livelli, e che cosa ne vede Java

Livello	Che cosa fa	In Java
applicazione	il nostro protocollo, HTTP, la posta: che cosa vogliono dire i byte	il nostro programma
trasporto	TCP e UDP: porte, consegna affidabile o singoli messaggi	`Socket`, `ServerSocket`, `DatagramSocket`
internet	IP: indirizzi, pacchetti che trovano la strada fra le reti	`InetAddress`
collegamento	Ethernet, Wi-Fi: i byte sul cavo o nell'aria, fra macchine vicine	niente

> Tutto il nostro codice sta nel livello più alto e parla con quello sotto attraverso i **socket**. Del resto si occupano il sistema operativo e le macchine in mezzo.

02

Il primo dialogo



Chi aspetta e chi bussa

`ServerSocket` è la porta su cui si bussa; `Socket` è un estremo della connessione.

```
ServerSocket server = new ServerSocket(1234);   // la porta: BindException se occupata
Socket socket = server.accept();                // aspetta, poi la connessione
Socket socket = new Socket("localhost", 1234);  // dall'altra parte: bussa
```

- `accept()` **blocca** finché nessuno si collega, poi restituisce il `Socket` di *quella* connessione: da uno stesso `ServerSocket` ne nascono tanti quanti i client
- i due `Socket` sono i due capi dello stesso filo: `getInputStream()` legge ciò che l'altro scrive, `getOutputStream()` scrive ciò che l'altro leggerà

> Quesito d'esame: che cosa restituisce `accept()`? Un `Socket` per un solo client. Non un `ServerSocket`, non un contatore, mai `null`.

Il server che fa l'eco

```
try (ServerSocket server = new ServerSocket(1234);
     Socket socket = server.accept();
     BufferedReader in = new BufferedReader(new InputStreamReader(
             socket.getInputStream(), StandardCharsets.UTF_8));
     PrintWriter out = new PrintWriter(
             socket.getOutputStream(), true, StandardCharsets.UTF_8)) {
    String riga;
    while ((riga = in.readLine()) != null) {          // null: il client ha chiuso
        out.println(riga.toUpperCase());
        if (riga.equals("fine")) break;
    }
}
```

- gli stream del socket sono **byte**: li incartiamo come un file, `InputStreamReader` traduce in caratteri, `BufferedReader` raggruppa in righe
- `true` e `UTF_8` non sono decorazioni: fra poco

Il client

```
try (Socket socket = new Socket(host, 1234);
     BufferedReader in = ...; PrintWriter out = ...;            // come nel server
     BufferedReader tastiera = new BufferedReader(new InputStreamReader(System.in))) {
    String riga;
    while ((riga = tastiera.readLine()) != null) {
        out.println(riga);
        String risposta = in.readLine();
        if (risposta == null) break;                            // il server ha chiuso
        System.out.println("> " + risposta);
        if (riga.equals("fine")) break;
    }
}
```

- `new Socket(host, porta)` traduce il nome, apre la connessione e aspetta che il server la accetti: `ConnectException` se nessuno ascolta

Il filo nudo: `nc`

Il client è solo un programma che scrive righe in un tubo e ne legge dall'altro. Ne esiste uno pronto su ogni sistema: `nc` (*netcat*; su Windows `ncat` o `telnet`).

```
$ nc localhost 1234
ciao
CIAO
fine
FINE
```

- per capire se un errore è nel server o nel client, si sostituisce il client con `nc`
- se `nc` riesce a parlare con il nostro server è perché il protocollo è fatto di **righe di testo**, che chiunque può scrivere

Un socket non è un file

- **i buffer**: `println` non manda, raccoglie. Il `true` del `PrintWriter` è l'**autoflush**; senza, la riga parte alla chiusura e i due programmi si aspettano per sempre, senza errori
- **il charset**: sul filo passano byte, e `è` sono due byte in UTF-8 e uno su Windows. Si dichiara `UTF_8` **da entrambi i lati**, sempre
- **la riga è la cornice**: TCP non ha confini fra messaggi; `readLine` li ritrova con l'a capo
- `null` **è la fine normale**: l'altro ha chiuso. *Connection reset*: è sparito senza salutare. Nessuno dei due è un bug nostro
- **chiudere** `in` **chiude tutto**; `shutdownOutput()` chiude il solo tubo di uscita

Quando va storto

Eccezione	Che cosa vuol dire, e dove guardare
`UnknownHostException`	il nome non si traduce: battitura, nessuna rete, DNS
`ConnectException` (*refused*)	la macchina c'è, nessuno ascolta su quella porta: server fermo o porta sbagliata
`SocketTimeoutException` in `connect`	la macchina non risponde affatto: indirizzo sbagliato, spenta, firewall
`BindException`	la porta è già usata: server avviato due volte, o sotto 1024 senza privilegi
`SocketException` (*reset*)	l'altro lato è sparito senza chiudere
`SocketTimeoutException` in lettura	`setSoTimeout` scaduto: nessuna riga entro il tempo fissato

> Va con `localhost` e non dall'altra macchina: serve l'indirizzo del server **sulla rete locale**, e il firewall chiede il permesso.

03

Il protocollo



Che cos’è un protocollo

Le regole che due programmi condividono per capirsi:

- la **cornice**: come si delimita un messaggio (una riga)
- la **sintassi**: quali messaggi esistono e come sono fatti
- la **semantica**: che cosa vuol dire ciascuno e che cosa si risponde
- il **turno**: chi può parlare in quale momento

A passo: richiesta e risposta, alternate (l’eco, la calcolatrice remota, HTTP). **Libero**: ognuno scrive quando vuole. Una chat è libera: il messaggio di Giulia deve arrivare a Marco *adesso*, non quando Marco avrà chiesto qualcosa.

> Un client a passo su un server libero: se il server manda una riga in più, il client resta indietro di una per sempre; una in meno, i due si aspettano a vicenda. È l’**interblocco di protocollo**: due programmi corretti ciascuno per sé.

Il client full-duplex

Due cicli bloccanti, ciascuno fermo nel suo `readLine`: due thread.

```
try (Socket socket = new Socket(host, 1234); /* in, out, tastiera come prima */
     ExecutorService exec = Executors.newFixedThreadPool(2)) {
    Future<?> lettore = exec.submit(() -> {                    // socket -> video
        String riga;
        while ((riga = in.readLine()) != null) System.out.println(riga);
        return null;
    });
    exec.submit(() -> {                                        // tastiera -> socket
        String riga;
        while (!lettore.isDone() && (riga = tastiera.readLine()) != null) out.println(riga);
        return null;
    });
    lettore.get();                                             // main aspetta la fine
}
System.exit(0);                                                // la tastiera e' ferma in readLine
```

Il client non conta più le righe: le stampa. Resta la domanda «che cosa vuol dire

Il protocollo di WhatsPAJC

Dal client: una barra per i comandi, senza barra è per tutti

Riga	Significato
`/nome Marco`	sceglie il nome; per primo
`/chi`	chi è collegato
`/a Giulia testo`	a Giulia soltanto
`/esci`, `/aiuto`	saluta; l'elenco dei comandi
`testo`	a tutti

Dal server: una parola maiuscola, il testo libero in fondo

Riga	Significato
`INFO testo`	un testo da mostrare
`BENVENUTO Marco`	nome accettato
`UTENTI Marco,Giulia`	l'elenco
`MSG Marco testo`	a tutti, mittente compreso
`PRIV Marco Giulia testo`	a entrambi
`ENTRA nome`, `ESCE nome`	chi arriva, chi se ne va
`ERR motivo`, `CIAO`	non è andata; chiudo

Una riga UTF-8 terminata da `\n`, nessun a capo dentro un messaggio. Il protocollo è **libero**; l'unico ordine: `/nome` prima di tutto.

Un dialogo, dal lato di Marco

```
INFO Benvenuto su WhatsPAJC. Scegli un nome con /nome
/nome Marco
BENVENUTO Marco
buongiorno a tutti
MSG Marco buongiorno a tutti
MSG Giulia ciao Marco
/a Giulia ci vediamo alle 10?
PRIV Marco Giulia ci vediamo alle 10?
ENTRA Sara
/a Luca ciao
ERR utente sconosciuto: Luca
/esci
CIAO
```

- il server rimanda `MSG` anche al mittente: tutti vedono lo stesso ordine, e il messaggio compare quando è davvero partito

Il protocollo come tabella

La chiave è il comando; il valore dice quanti argomenti, come si spiega, che cosa fa.

```
public record Azione(int argomenti, String aiuto, BiConsumer<Sessione, String[]> esegui) {}
static final Map<String, Azione> COMANDI = new LinkedHashMap<>();   // nell'ordine di /aiuto
static {
    azione("/nome",  1, "/nome N     il nome, per primo", (s, a) -> s.nome(a[0]));
    azione("/chi",   0, "/chi        chi e' collegato",   (s, a) -> s.chi());
    azione("/a",     2, "/a N testo  privato a N",        (s, a) -> s.privato(a[0], a[1]));
    azione("/esci",  0, "/esci       saluta",             (s, a) -> s.esci());
    azione("/aiuto", 0, "/aiuto      questo elenco",      (s, a) -> s.aiuto());
}
// azione(cmd, n, aiuto, esegui): COMANDI.put(cmd, new Azione(n, aiuto, esegui))
```

- aggiungere `/ora`: una riga qui, un metodo in `Sessione`, nessuno `switch` da toccare
- l'azione riceve la `Sessione`: la tabella non tiene stato, niente `static` oltre a lei

Da riga a comando

```
private void elabora(String riga) {                                    // in Sessione
    if (riga.isBlank()) return;
    if (!riga.startsWith("/")) { tutti(riga.strip()); return; }         // senza barra: a tutti
    String[] campi = riga.strip().split(" ", 2);                       // il comando e il resto
    Azione a = Protocollo.COMANDI.get(campi[0].toLowerCase());
    if (a == null) { invia("ERR comando sconosciuto: " + campi[0]); return; }
    String[] arg = Protocollo.argomenti(campi.length > 1 ? campi[1] : "", a.argomenti());
    if (arg == null) { invia("ERR uso: " + a.aiuto()); return; }
    a.esegui().accept(this, arg);
}
```

- `argomenti(resto, n)`: `split(" ", n)` con il **limite**, l'ultimo campo è il resto con i suoi spazi; meno di `n` o l'ultimo vuoto: `null`
- **non lancia mai**: ogni riga finisce in un'azione o in un `ERR`; parlare prima di `/nome` è `ERR`

L’aiuto esce dalla tabella

```
void aiuto() {
    for (Azione a : Protocollo.COMANDI.values()) invia("INFO " + a.aiuto());
    invia("INFO una riga senza barra e' un messaggio per tutti");
}
```

- l’elenco non può restare indietro rispetto ai comandi: è la tabella stessa, nell’ordine in cui è scritta
- lo stesso testo risponde `ERR uso:` quando gli argomenti non tornano: una sola fonte per l’uso e per l’aiuto
- un `INFO` per riga: niente a capo in un messaggio, e il server è il primo a rispettarlo

Provarlo senza rete

```
@Test void ilRestoTieneGliSpazi() {
    assertArrayEquals(new String[] {"Giulia", "ci vediamo alle 10?"},
            Protocollo.argomenti("Giulia ci vediamo alle 10?", 2));
}
@Test void privatoAUnoSconosciuto() {
    Reader righe = new StringReader("/nome Marco\n/a Luca ciao\n");
    StringWriter uscita = new StringWriter();
    Sessione s = new Sessione(righe, uscita, new Registro());      // il Registro e' vero
    s.run();
    assertTrue(uscita.toString().contains("ERR utente sconosciuto: Luca"));
}
```

- in rete un errore di parsing è un dialogo sfasato, lontano dalla causa: si prova **prima** del socket
- `Sessione` su `Reader` e `Writer`: il socket in produzione, due stringhe nel test

Mai fidarsi di ciò che arriva

Una riga dalla rete è input dell'utente, con un'aggravante: l'utente è dall'altra parte, con un client suo, e può sbagliare o provare apposta.

```
// SBAGLIATO: una riga malformata uccide la connessione
String[] campi = riga.split(" ");
String dest = campi[1];                  // ArrayIndexOutOfBounds con "/a"
int n = Integer.parseInt(campi[2]);      // NumberFormatException con "/x tre"
```

- un'eccezione non catturata esce dal ciclo di lettura, il try-with-resources chiude il socket e il client legge `null` senza spiegazioni
- ogni campo controllato prima di usarlo, ogni conversione protetta, e la risposta a un errore è un `ERR`. I controlli stanno in `elabora` e in `argomenti`: le azioni ricevono il numero giusto di argomenti, mai `null`

Due modi di scrivere un protocollo

UNA TABELLA

```
record Azione(int argomenti, String aiuto,
        BiConsumer<Sessione, String[]> esegui)

COMANDI.get(cmd).esegui().accept(this, arg);
```

Una riga per comando, l'aiuto esce dalla tabella; estendere è aggiungere una riga

UN TIPO PER MESSAGGIO

```
sealed interface Richiesta permits A, ... {}
record A(String a, String testo)
        implements Richiesta {}

switch (parse(riga)) { case A a -> ... }
```

Il messaggio è un valore con `equals`; un caso in più e il compilatore indica ogni `switch`

04

Un server per molti



Un client alla volta

Il thread del server è fermo in `readLine` sul socket di Marco e non chiama `accept`. Giulia si collega lo stesso.

- il sistema operativo completa la connessione e la mette in coda (il **backlog**, 50 per default): `new Socket` ritorna senza errori
- le righe di Giulia partono e si fermano nel buffer del server: silenzio, nessun errore
- quando Marco scrive `/esci`, il server ritira Giulia e legge di colpo tutto ciò che lei ha scritto nel frattempo

Due attese incompatibili con un thread solo: il **nuovo** client in `accept` e la **prossima riga** di ciascun client in `readLine`. Un thread fa solo `accept`; ogni connessione accettata va a un thread suo.

La `Sessione`

Tutto ciò che riguarda *un* client, scritto come `Runnable`; il server a un client ci entra quasi intatto.

```
public class Sessione implements Runnable {
    private final Socket socket;
    private final Registro registro;      // lo stato condiviso, ricevuto nel costruttore
    private final BufferedReader in;
    private final PrintWriter out;
    private String nome;                  // null finche' non arriva /nome

    public synchronized void invia(String riga) {
        out.println(riga);
    }
    // dialoga(): il ciclo di readLine che passa ogni riga a elabora, come prima
}
```

Il `synchronized` su `invia` e il `Registro` nel costruttore: le due conseguenze dei molti thread, fra poco.

Tutti i modi di finire

```
@Override public void run() {
    try {
        socket.setSoTimeout(30 * 60 * 1000);              // mezz'ora di silenzio, poi lancia
        dialoga();
    } catch (IOException e) {
        log("connessione persa: " + e.getMessage());      // informazione, non un bug
    } finally {
        if (nome != null) {
            registro.rimuovi(nome);
            registro.aTutti("ESCE " + nome);
        }
        chiudi();                                         // socket.close()
    }
}
```

Il `finally` passa in **tutti** i modi di finire: `/esci`, `null`, *reset*, timeout, un errore nostro. Senza, chi stacca il cavo resta in `/chi` per sempre: l'**utente fantasma**.

Un thread per sessione

```
private final ServerSocket server;                  // new ServerSocket(porta) nel costruttore
private final Registro registro = new Registro();
private final ExecutorService exec = Executors.newVirtualThreadPerTaskExecutor();

public void avvia() {
    try {
        while (true) {
            Socket socket = server.accept();
            exec.submit(new Sessione(socket, registro));
        }
    } catch (IOException e) {                       // il ServerSocket e' stato chiuso: fine
    }
    exec.close();                                   // aspetta le sessioni
}
```

Il ciclo di `accept` ritira una connessione e la **consegna** a un executor; `while (true)` è una scelta.

Quale executor

Una sessione dura **quanto la connessione**: minuti o ore, per il 99% del tempo ferma in `readLine`, con un lampo di lavoro a ogni riga.

- l'opposto dei compiti di calcolo: è il caso dei **thread virtuali** JAVA 21. `newVirtualThreadPerTaskExecutor()`, un thread per sessione; mille client sono mille thread virtuali (prima: `newCachedThreadPool()`, un thread del sistema per client)
- in Java 21 un thread virtuale bloccato *dentro* un `synchronized` resta inchiodato al carrier: sezioni corte, mai un'attesa dentro. In `invia` c'è una `println`, che non blocca

> `newFixedThreadPool(4)`: il quinto client viene accettato, la sua sessione resta in coda finché uno dei quattro non esce. Si collega, scrive, e tace.

Il `Registro`: stato condiviso

```
public class Registro {
    private final ConcurrentHashMap<String, Sessione> sessioni = new ConcurrentHashMap<>();
    public boolean aggiungi(String nome, Sessione s) {
        return sessioni.putIfAbsent(nome, s) == null;     // true se il nome era libero
    }
    // rimuovi(nome), trova(nome), nomi(): remove, get, le chiavi ordinate

    public void aTutti(String riga) {
        for (Sessione s : sessioni.values()) s.invia(riga);
    }
}
```

- `putIfAbsent` è un'operazione sola: con `if (!containsKey) put` fra le due un altro thread infila lo stesso nome, e sono due Marco (*check-then-act*)
- non è `static`: passato nel costruttore, si prova con un registro finto

Due thread, un tubo

`aTutti` gira sul thread di Marco e scrive sul `PrintWriter` di Giulia. Nello stesso istante il thread di Sara scrive sullo stesso `PrintWriter`.

- due `println` da due thread non sono atomiche l'una rispetto all'altra: Giulia riceve `MSG Marco buonMSG Sara ciao` e poi `giorno`
- `synchronized` su `invia`: il lock è quello della sessione di **Giulia**. Chi vuole scrivere sul suo socket lo prende, scrive una riga intera, lo lascia
- `out` è privato: se qualcuno scrivesse senza passare da `invia`, la garanzia sparirebbe

> Con tre persone che scrivono a mano non si vede mai; con cento client, o un programma che manda a raffica, succede subito. È l'errore che «non si riproduce»: si trova con il ragionamento, non con le prove.

Entrare, parlare a tutti, parlare a uno

```
void nome(String n) {                                          // le azioni della tabella
    if (!registro.aggiungi(n, this)) { invia("ERR nome gia' in uso"); return; }
    nome = n;
    invia("BENVENUTO " + nome);                                // prima a chi entra
    registro.aAltri("ENTRA " + nome, this);                    // poi agli altri
}
void tutti(String testo) { registro.aTutti("MSG " + nome + " " + testo); }
void privato(String a, String testo) {
    Sessione dest = registro.trova(a);
    if (dest == null) { invia("ERR utente sconosciuto: " + a); return; }
    String riga = "PRIV " + nome + " " + a + " " + testo;
    dest.invia(riga);
    if (dest != this) invia(riga);
}
```

Fra `trova` e `dest.invia` il destinatario può andarsene: `invia` scrive su un socket chiuso e nessuno se ne accorge. Una finestra inevitabile fra il controllo e l'azione.

Fermare un server

```
public void ferma() throws IOException {
    server.close();                          // accept() lancia SocketException
    registro.aTutti("CIAO");
    for (String nome : registro.nomi()) {
        Sessione s = registro.trova(nome);
        if (s != null) s.chiudi();           // readLine() lancia
    }
}
```

- il thread di `avvia` è fermo in `accept`, non guarda variabili e `interrupt()` non lo sveglia. Si chiude il `ServerSocket` **da un altro thread**: `accept` lancia e il `catch` esce dal ciclo. Non è un errore, è il segnale
- lo stesso per le sessioni: chiudere il socket fa lanciare il `readLine`, e si passa dal `finally`. Chi chiama `ferma`: un thread *daemon* che legge `stop` dalla console

05

Oltre il filo



Il filo che finisce in una finestra

`readLine` nel listener di «Connetti»: la finestra congela. `append` da un thread a parte: si corrompe. Il **thread lettore** fa una cosa sola: legge, traduce, **accoda** all'EDT.

```
lettore = Thread.ofVirtual().unstarted(() -> {
    try {
        in.lines().forEach(riga ->                                  // fino a null
                SwingUtilities.invokeLater(() -> ricevi.accept(riga)));  // la riga va all'EDT
    } catch (UncheckedIOException e) { /* chiusa da noi, o persa */ }
    SwingUtilities.invokeLater(() -> chiusa.accept(null));
});
```

- Model e View cambiano sull'EDT, un evento alla volta: il lettore non conosce nessun componente. `Connessione` incapsula socket, lettore e `invia`
- sull'EDT la stessa idea del server: la prima parola (`MSG`, `ENTRA`) è la chiave di una tabella, il resto l'argomento

HTTP è testo

```
$ printf 'GET / HTTP/1.1\r\nHost: example.com\r\nConnection: close\r\n\r\n' | nc example.com 80
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 1256

<!doctype html>
...
```

- un protocollo **a passo**: la riga di richiesta (metodo, percorso, versione), le intestazioni `Nome: valore`, una riga vuota; la risposta con la riga di stato, le intestazioni, la riga vuota e il corpo, lungo quanto dice `Content-Length`
- righe terminate da `\r\n`. Ciò che un browser fa cento volte per pagina sono righe di testo in un socket

`HttpClient`

```java
HttpClient http = HttpClient.newHttpClient();
HttpRequest richiesta = HttpRequest.newBuilder(URI.create("http://localhost:1236/stato"))
        .build();                                                    // GET, per default
HttpResponse<String> risposta = http.send(richiesta, BodyHandlers.ofString());
System.out.println(risposta.statusCode());                           // 200
System.out.println(risposta.body());                                 // <h1>WhatsPAJC</h1>...
```

`java.net.http` JAVA 11: il client, la richiesta, la risposta. Sa `https` e i reindirizzamenti.

- `send` **blocca** finché la risposta non è arrivata tutta: da una finestra va in un `SwingWorker`; `sendAsync` restituisce un `CompletableFuture`
- il `BodyHandler` dice che cosa fare del corpo: `ofString`, `ofFile`, `ofInputStream`; `POST` con `BodyPublishers.ofString`: quanto basta per un servizio REST

Quale strumento per quale messaggio

Il messaggio è…	Lo strumento	Perché
testo breve, un dialogo	una riga, `BufferedReader` e `PrintWriter`, TCP	leggibile, provabile con `nc`, parla con tutti
byte: un file, una foto	una cornice con la lunghezza davanti, `DataInputStream` e `DataOutputStream`	la riga non è una cornice per i byte
un oggetto Java	testo strutturato: il protocollo, o JSON; `ObjectOutputStream` solo fra parti dello stesso programma	formato indipendente dalla classe; mai deserializzare dati altrui
un segnale breve, a chi non si conosce	un datagramma UDP, anche in broadcast	nessuna connessione, nessuna attesa
una risorsa del web	`HttpClient`	HTTP è testo, ma ha le sue regole

Errori frequenti

- `PrintWriter` **senza autoflush**: niente parte, nessun errore
- **il client a passo** su un server che parla da solo: indietro di una riga per sempre
- `split(" ")` **senza limite**: il testo con gli spazi si frantuma
- **l'eccezione nel ciclo di lettura**: la sessione muore, il client legge `null`
- **il pool fisso** per le sessioni: il quinto client tace
- `HashMap` **condivisa**, `containsKey` poi `put`: due Marco
- **due thread sullo stesso** `PrintWriter`: righe mescolate
- **senza** `finally`: l'utente fantasma in `/chi`
- `readLine` **sull'EDT**: la finestra congela

Rete, in due righe

- un socket sono due tubi di byte fra due programmi: `ServerSocket` aspetta, `Socket` bussa, `accept` restituisce la connessione; charset e autoflush dichiarati, `null` è la fine, un socket **non è un file**
- un protocollo è l'accordo su che cosa passa nei tubi: righe, e una **tabella** che per ogni comando dice argomenti, aiuto e azione, provata senza rete; molti client sono una **sessione su un thread virtuale** ciascuno, con lo stato condiviso protetto e la scrittura in fila