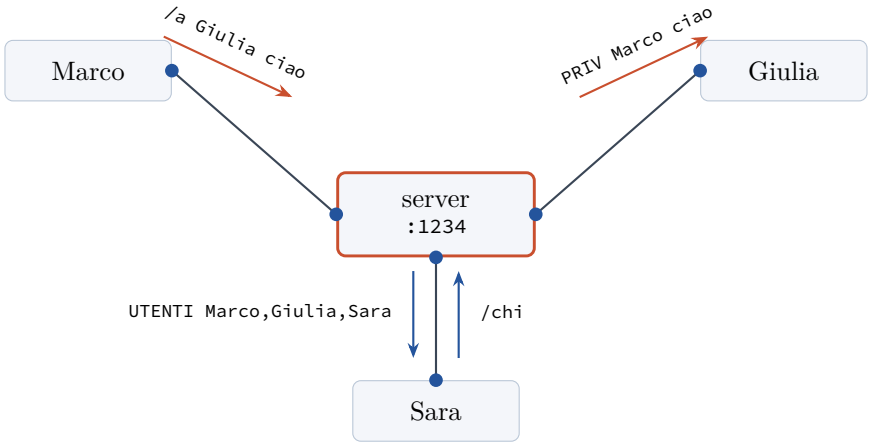
PAJC · Programmazione Avanzata in Java e C · 2026/27

Dispensa 6

Comunicazione in rete

Socket e TCP in Java, protocolli di testo, un server per molti client, il client con la finestra, byte, oggetti e datagrammi



Un filo, due porte e un accordo su che cosa dirsi.

prof. Massimiliano Redolfi

Università degli Studi di Brescia · Ingegneria Informatica

massimiliano.redolfi@unibs.it

Versione 0.2

Quest'opera è distribuita con licenza Creative Commons "Attribuzione – Non commerciale – Condividi allo stesso modo 4.0 Internazionale".

Indice

1 Introduzione: una chat per l'aula **1**
- 1.1 Il problema . . . 1
- 1.2 L'immagine guida: il filo e l'accordo . . . 3
- 1.3 Che cosa serve . . . 4
- 1.4 La strada che percorreremo . . . 4
- Esercizi . . . 5
- Risposte . . . 5

2 Due programmi che si parlano: socket e TCP **7**
- 2.1 Che cosa serve sapere della rete . . . 7
- 2.2 Il primo dialogo . . . 9
- 2.3 Che cosa succede davvero . . . 12
- 2.4 Quando va storto . . . 14
- 2.5 L'esempio guida: Saluto . . . 15
- Esercizi . . . 17
- Risposte . . . 19

3 Mettersi d'accordo: il protocollo **20**
- 3.1 Un accordo su che cosa dirsi . . . 20
- 3.2 Il protocollo di WhatsPAJC . . . 22
- 3.3 Il protocollo come codice . . . 24
- 3.4 Mai fidarsi di ciò che arriva . . . 29
- 3.5 Un protocollo che si usa tutti i giorni: HTTP . . . 29
- 3.6 L'esempio guida: WhatsPAJC v1 . . . 30
- Esercizi . . . 33
- Risposte . . . 35

4 Un server per molti **37**
- 4.1 Un client alla volta . . . 37
- 4.2 La sessione e il ciclo di `accept` . . . 38
- 4.3 Il registro: stato condiviso . . . 41
- 4.4 Entrare e uscire . . . 42
- 4.5 Fermare un server . . . 44
- 4.6 L'esempio guida: WhatsPAJC v2 . . . 45
- Esercizi . . . 46
- Risposte . . . 48

5 Il client con la finestra **50**
- 5.1 Il problema . . . 50
- 5.2 La connessione dietro la finestra . . . 51
- 5.3 Il Model non sa di socket . . . 53
- 5.4 La View e il Controller . . . 55
- 5.5 Le cose che si vedono solo in due . . . 60

5.6 L'esempio guida: WhatsPAJC v3 . . . 61
Esercizi . . . 62
Risposte . . . 64

6 Oltre le righe di testo: byte, oggetti e datagrammi 66
6.1 Il problema . . . 66
6.2 Byte con una cornice . . . 67
6.3 Oggetti sul filo . . . 69
6.4 Datagrammi: trovare il server . . . 70
6.5 Parlare con il web . . . 72
6.6 L'esempio guida: WhatsPAJC v4 . . . 73
Esercizi . . . 74
Risposte . . . 75

7 Palestra 77

8 Dove andare adesso 82
8.1 Canali e `Selector`: NIO . . . 82
8.2 Connessioni cifrate: TLS . . . 82
8.3 Server HTTP e WebSocket . . . 83
8.4 JSON e Protocol Buffers . . . 83
8.5 Chiamare un metodo a distanza: RMI e gRPC . . . 83
8.6 Giochi in rete: la Dama e il torneo . . . 83
8.7 Nelle altre dispense . . . 84
8.8 Riferimenti . . . 84

Capitolo 1

Introduzione: una chat per l'aula

1.1 Il problema

Il programma che accompagna tutta la dispensa è una chat, *WhatsPAJC*. L'idea è semplice: un server gira sul computer di uno studente, gli altri si collegano dalla rete dell'aula con un programma client, ognuno sceglie un nome, vede chi c'è, scrive a tutti oppure a una persona sola. Il client ha una finestra con l'elenco degli utenti a sinistra, la conversazione a destra e un campo di testo in basso: si scrive, si preme Invio, e il messaggio compare sulle finestre di tutti gli altri. Non c'è nulla che non abbiamo già visto, sembrerebbe: una finestra Swing, qualche lista, delle righe di testo che vanno e vengono.

Fin qui, però, tutti i programmi che abbiamo scritto vivevano in *un solo processo*. Lo stato stava in memoria, l'input arrivava dalla tastiera o da un file, e se due parti del programma dovevano parlarsi condividevano un oggetto. Qui no: il server e ogni client sono processi diversi, su macchine diverse, e non condividono niente. Quello che condividono è un *filo*: una connessione di rete su cui passano byte, in ordine, in entrambi i versi. Java la presenta con una classe, `Socket`, che restituisce due stream, uno in ingresso e uno in uscita, molto simili a quelli con cui leggiamo e scriviamo i file. Chi conosce gli stream può scrivere il primo client in dieci righe, ed è quello che facciamo. Il client si collega al server, legge una riga dalla tastiera, la manda, aspetta la risposta e la stampa:

```
try (Socket socket = new Socket("192.168.1.10", 1234);
     BufferedReader in = new BufferedReader(
             new InputStreamReader(socket.getInputStream()));
     PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
     BufferedReader tastiera = new BufferedReader(
             new InputStreamReader(System.in))) {
    String riga;
    while ((riga = tastiera.readLine()) != null) {
        out.println(riga);                    // mando ...
        System.out.println(in.readLine());    // ... e aspetto la risposta
    }
}
```

Il server è simmetrico: apre una porta, aspetta che un client si colleghi, e poi risponde a ogni riga che riceve inoltrandola, con il nome del mittente davanti. Il riquadro rosso segnala che questo client, pur compilando e funzionando al primo tentativo, è sbagliato in modi che si vedono solo usandolo, e la dispensa serve a capire perché. Proviamolo in aula.

Si collega il secondo. Marco avvia il client e tutto funziona: scrive, il server gli risponde. Giulia avvia il suo: `new Socket` ritorna senza errori, la connessione è aperta, Giulia scrive una

riga e non succede niente. Nessun messaggio d'errore, nessuna eccezione, da nessuna delle due parti: Giulia aspetta per sempre. Il server, scritto con gli strumenti che conosciamo, serve un client alla volta, e il sistema operativo nel frattempo *accetta* le connessioni al posto suo e le mette in fila. Il telefono squilla, qualcuno risponde, ma nessuno parla.

I messaggi arrivano in ritardo. Supponiamo di aver risolto il primo problema. Marco scrive "ciao" e lo vede inoltrato. Giulia scrive "ciao Marco". Sul video di Marco non compare niente: il suo client sta fermo su `tastiera.readLine()`, in attesa che Marco scriva, e la riga di Giulia resta nel buffer del socket. Comparirà solo quando Marco avrà scritto qualcosa, perché solo allora il client arriva a `in.readLine()`. Peggio: se il server, per una sola riga ricevuta, ne manda due (il messaggio inoltrato e un avviso), il client ne legge una e la seconda resta in attesa della riga seguente, e da lì in poi tutto è sfasato di uno. Il client parla e ascolta *a turno*, e una chat non funziona a turno.

L'accento. Sara scrive "perché" da Windows e Marco, su Mac, legge "perch" seguito da un simbolo che non è una lettera. Nessuno dei due programmi ha dichiarato *come* le lettere diventano byte, e i due sistemi hanno risposto in modo diverso alla domanda non fatta.

La finestra. Passiamo dal client in console a quello con la finestra. Il listener del pulsante *Connetti* apre il socket e si mette a leggere, riga dopo riga, aggiungendo ogni riga all'area della conversazione. Premiamo *Connetti* e la finestra congela: niente si ridisegna, il campo di testo non accetta più un carattere, e il sistema dopo qualche secondo propone di terminare il programma. La lettura *blocca* il thread che la esegue finché una riga non arriva, e il thread che esegue il listener è lo stesso che disegna la finestra. Lo sappiamo da *Thread e concorrenza*: spostiamo la lettura in un thread a parte. La finestra torna viva. Dopo un po' però la conversazione mostra righe mezze, righe doppie, e una volta ogni tanto un'eccezione nella console: l'area di testo viene toccata da due thread, e non è fatta per questo.

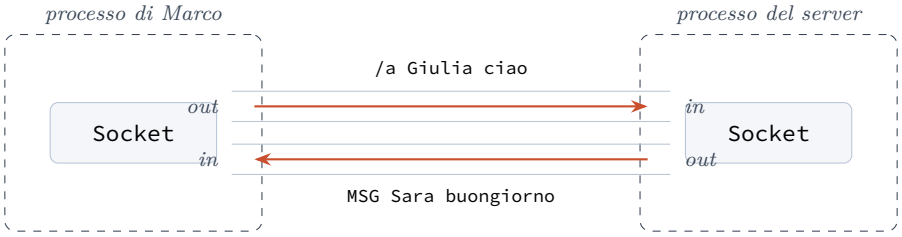
Ognuno di questi difetti ha una causa precisa, e nessuna è un errore di battitura. Sono le tre domande della dispensa.

- **Come si parlano due programmi?** Che cosa passa davvero sul filo, con quali oggetti Java si apre una connessione e la si usa, che cosa succede quando l'altro chiude o sparisce, perché una riga può restare in un buffer e non partire mai.
- **Come si mettono d'accordo?** Le due parti devono condividere un *protocollo*: che cosa è un messaggio, come si distingue un comando da un testo, chi parla quando, che cosa si risponde a un errore. Il protocollo è codice, e si può progettare e provare prima di collegare qualunque filo.
- **Come si servono molti senza far aspettare nessuno?** Il server deve parlare con tutti i client insieme, il client deve ascoltare mentre l'utente scrive, e la finestra deve restare viva mentre il filo tace. Sono problemi di thread, e la rete è il posto in cui i thread diventano indispensabili.

Il capitolo 7 riprende WhatsPAJC e la porta fino in fondo, con il server per molti, il client con la finestra e tutto ciò che è stato descritto all'inizio. In mezzo c'è quello che serve per arrivarci.

1.2 L'immagine guida: il filo e l'accordo

Un'immagine tornerà in ogni capitolo. Una connessione fra due programmi è un *filo* con due *tubi* dentro, uno per verso: ciò che un programma scrive nel suo tubo di uscita esce, nello stesso ordine e senza perdere nulla, dal tubo di ingresso dell'altro. In Java ogni estremo del filo è un `Socket`, e i due tubi sono `getOutputStream()` e `getInputStream()`. Sul filo passano *byte*: non righe, non messaggi, non oggetti. Tutto ciò che ha un significato ("questa è una riga", "questo è un comando", "questo testo è per Giulia") è un *accordo* fra i due programmi su come leggere quei byte: il *protocollo*.



Due tubi, uno per verso: byte in ordine. Che cosa vogliono dire è l'accordo.

I due lati dell'immagine corrispondono alle due metà della dispensa. Il *filo* è TCP e i socket: come si apre, che garanzie dà (i byte arrivano tutti, in ordine, una volta sola) e quali no (non dice dove finisce un messaggio, non avverte subito se l'altro è sparito), che cosa costa e come si chiude. L'*accordo* è il protocollo: le righe come unità, i comandi e le risposte, e il codice che traduce una riga in un oggetto Java e un oggetto Java in una riga. Il filo lo dà la libreria; l'accordo lo scriviamo noi, ed è la parte in cui si decide se un programma di rete è chiaro o un groviglio di `split` e `if`.

Attorno al filo ci sono i *thread*. Un server con molti client ha molti fili, e per ciascuno qualcuno deve stare in ascolto: un thread per filo, che passa quasi tutto il tempo fermo ad aspettare una riga. Un client ha un filo solo, ma deve ascoltarlo mentre l'utente scrive, e se ha una finestra deve consegnare ciò che arriva al thread che la disegna. L'immagine si arricchisce di capitolo in capitolo: molti fili che entrano in un server (capitolo 4), un filo che finisce in una finestra con un thread lettore in mezzo (capitolo 5), un filo che porta byte invece di righe, e una lettera al posto del filo (capitolo 6).

Osservazione 1.1

Il salto è nel modo di pensare l'*altro*. Nei programmi scritti finora, chiamare un metodo voleva dire ottenere una risposta, subito e di sicuro. In rete l'altro è un programma che non controlliamo: può rispondere fra un secondo o mai, può sparire a metà di una frase, può mandare una riga che non capiamo o che non ci aspettavamo. Un programma di rete corretto non è quello in cui tutto va bene, ma quello che sa che cosa fare quando qualcosa non va: è per questo che la dispensa insiste tanto sugli errori, i timeout e le chiusure, e li tratta come casi normali e non come eccezioni.

1.3 Che cosa serve

Gli stream di byte e di caratteri, `BufferedReader` e `PrintWriter`, il try-with-resources, i record e le collezioni (*Java Fast Forward*): un socket si legge e si scrive con gli stessi oggetti con cui si leggono e scrivono i file, e il protocollo è una mappa. Le lambda e i method reference (*Dal nome alla funzione*): i compiti dei thread e i listener sono lambda, e ogni comando del protocollo è una funzione in una tabella; le interfacce `sealed` e lo `switch` sui tipi (*Le interfacce in Java moderno*) compaiono come l'altra forma possibile del protocollo. I thread, gli executor, lo stato condiviso con `synchronized` e le collezioni concorrenti, i thread virtuali, e il thread degli eventi di Swing con `invokeLater` e `SwingWorker` (*Thread e concorrenza*): dal capitolo 4 in poi servono tutti, e le pagine che servono vengono riassunte dove servono, con il rimando. Una finestra con i suoi componenti, una `JList` con il suo modello e le finestre di dialogo (*Interfacce grafiche con Swing*) per il client del capitolo 5; dalla dispensa *Pattern in gioco* il Model che avvisa i suoi ascoltatori, usato in una pagina.

Non serve nessun corso di reti. Ciò che un programma deve sapere della rete (che cosa è un indirizzo, che cosa è una porta, che cosa promette TCP e che cosa no) sta nel capitolo 2, in poche pagine. Resta fuori, di proposito, la teoria delle reti oltre quello che un programma vede: come i pacchetti trovano la strada, che cosa c'è nei cavi, i sette livelli del modello ISO/OSI. Restano fuori anche i canali non bloccanti di NIO, le connessioni cifrate con TLS, i WebSocket e le librerie per JSON: ne parla, con un assaggio ciascuno, il capitolo 8.

Il codice è Java 21 e usa solo la libreria standard. Per provare un server si usa, oltre ai client scritti in Java, un programma che ogni sistema ha o può avere: `nc` (*netcat*), che apre una connessione a un indirizzo e una porta e collega la tastiera e il video ai due tubi. È lo strumento più utile di tutta la dispensa, perché mostra il filo nudo, senza il nostro codice in mezzo.

1.4 La strada che percorreremo

Il capitolo 2 risponde alla prima domanda. Che cosa un programma deve sapere della rete: indirizzi, nomi, porte, TCP e UDP, i ruoli di client e server. Poi il primo dialogo: `ServerSocket` e `accept`, `Socket` e i suoi stream, le righe con il charset dichiarato e il flush, la chiusura, i tempi e gli errori. Alla fine un server risponde a un client, e sappiamo che cosa succede davvero a ogni riga.

Il capitolo 3 risponde alla seconda. Che cosa è un protocollo, perché la chat non può andare a turno, come si progetta il protocollo di WhatsPAJC (i comandi del client, le risposte del server, gli errori) e come lo si scrive in Java: una tabella da comando ad azione, una sessione che legge righe e chiama le azioni, provata con JUnit senza nessuna rete. Alla fine un server capisce tutti i comandi, per un client solo, e il client di console ascolta mentre si scrive.

Il capitolo 4 risponde alla terza, dal lato del server: una sessione per client su un thread, i thread virtuali, il registro degli utenti come stato condiviso, i messaggi a tutti e a uno, l'ingresso e l'uscita, il client che sparisce, lo spegnimento ordinato. Alla fine il server di WhatsPAJC è completo.

Il capitolo 5 risponde dal lato del client con la finestra: una connessione dietro una finestra, il thread lettore e il thread degli eventi, il Model che non sa di socket e la vista che lo ascolta, il dialogo di connessione, gli errori a video, la riconnessione. Alla fine il client di WhatsPAJC è completo.

Il capitolo 6 va oltre le righe di testo: una cornice binaria per mandare una foto, gli oggetti serializzati e perché preferiamo il testo, i datagrammi UDP per trovare il server sulla rete locale senza conoscerne l'indirizzo, e HTTP, il protocollo di testo che tutti usano ogni giorno, letto con `HttpClient` e scritto a mano.

Il capitolo 7 è la palestra: otto programmi completi, in difficoltà crescente, dal server d'esame che manda l'ora fino a WhatsPAJC finita. Il capitolo 8 indica dove continuare.

Esercizi

Esercizio 1.1: Il secondo client (⋆)

Nel primo tentativo il server serve un client alla volta. Giulia avvia il suo client mentre Marco è collegato: `new Socket(...)` ritorna senza errori e Giulia può scrivere. Dire che cosa fa ciascuna delle sue righe (parte? resta da qualche parte? si perde?) e che cosa vedrebbe Giulia se Marco, a un certo punto, chiudesse il suo client.

Esercizio 1.2: A turno (⋆)

Con il client del primo tentativo, Marco e Giulia sono collegati a un server che inoltra a tutti ogni riga ricevuta. Marco scrive `A`, poi Giulia scrive `B`, poi Giulia scrive `C`, poi Marco scrive `D`. Dire che cosa compare sul video di Marco e in quale momento, supponendo che il server inoltri anche al mittente.

Esercizio 1.3: La finestra che si corrompe (⋆)

Nel client con la finestra, la lettura è stata spostata in un thread a parte che esegue `areaTesto.append(riga + "\n")` per ogni riga ricevuta. La finestra non congela più, ma ogni tanto il testo è doppio o spezzato. Il problema è nel socket, nel thread o nella `JTextArea`? Che cosa dice *Thread e concorrenza* dei componenti di Swing toccati da un thread che non è quello degli eventi, e qual è lo strumento che risolve?

Risposte

Risposta 1.1. La connessione di Giulia è aperta davvero: il sistema operativo del server l'ha accettata e messa in fila, in attesa che il programma la prenda con `accept`. Le righe che Giulia scrive partono, attraversano la rete e si fermano nel buffer di ricezione del sistema operativo del server, dove nessun programma le legge: non si perdono (TCP le conserva, finché il buffer non si riempie), ma non arrivano a nessuno. Quando Marco chiude, il server torna ad `accept`, riceve la connessione di Giulia e legge tutte le righe accumulate una dopo l'altra, come se Giulia le avesse scritte in quel momento: Giulia vede arrivare di colpo tutte le risposte. Il capitolo 4 risolve il problema con un thread per client.

Risposta 1.2. Il client di Marco legge una riga dal server soltanto dopo averne mandata una. Dopo `A` legge una riga: `A` inoltrata. Le righe `B` e `C` arrivano mentre Marco non legge e restano nel buffer. Quando Marco scrive `D`, il client legge *una* riga: `B`. Sul video di Marco compare quindi `A` subito, poi `B` dopo che ha scritto `D`; `C` e `D` aspettano le due righe successive. Da quel momento ogni messaggio è in ritardo di due, e non c'è modo di rimettersi in pari. Serve un lettore sempre in ascolto, indipendente dalla tastiera: il capitolo 3.

Risposta 1.3. Nella `JTextArea`, o meglio nel thread che la tocca. Il socket consegna le righe correttamente e il thread lettore fa il suo lavoro; ma i componenti di Swing non sono thread-safe, e possono essere letti e modificati solo dal thread degli eventi (`AWT-EventQueue-0`). `append` chiamato da un altro thread lavora sullo stesso documento che il thread degli eventi sta disegnando o modificando, e il risultato è imprevedibile: righe spezzate, doppie, ogni tanto un'eccezione. Lo strumento è `SwingUtilities.invokeLater`: il thread lettore riceve la riga e accoda al thread degli eventi il compito di aggiungerla. È il centro del capitolo 5.

Capitolo 2

Due programmi che si parlano: socket e TCP

Nel capitolo 1 abbiamo visto un client di dieci righe che compila, funziona al primo tentativo e poi tradisce in cinque modi diversi. Prima di correggerlo dobbiamo sapere che cosa c'è sotto quelle dieci righe: che cosa è un indirizzo, che cosa è una porta, che cosa promette la connessione che `new Socket` apre, e che cosa succede a una riga fra `println` e `readLine`. Il problema di questo capitolo è la prima delle tre domande: come si parlano due programmi. Alla fine sapremo scrivere un server e un client che si scambiano righe di testo, sapremo che cosa succede quando uno dei due chiude, e sapremo leggere le eccezioni della rete, che non sono errori nel nostro codice ma fatti del mondo. WhatsPAJC comincia qui, nella sua forma più piccola: un server che risponde a un client e lo chiama per nome.

2.1 Che cosa serve sapere della rete

Un programma che usa la rete non ha bisogno di sapere come i byte attraversano un cavo o l'aria, né come trovano la strada fra una macchina e l'altra: ci pensano il sistema operativo e le macchine in mezzo. Ha bisogno di sapere quattro cose: come si identifica una macchina, come si identifica un programma su quella macchina, che cosa garantisce il servizio che usa, e chi dei due comincia.

Indirizzi e nomi. Ogni macchina collegata a una rete ha un *indirizzo IP*, un numero che la identifica. Nella forma più comune (IPv4) è un numero di 32 bit scritto come quattro numeri da 0 a 255 separati da punti, `192.168.1.10`; nella forma più recente (IPv6) è un numero di 128 bit scritto in esadecimale, come `fe80::1c2a:9f3b:7c1d:5e4f`. Una macchina può averne più d'uno, perché può avere più schede di rete (il cavo e il Wi-Fi), e ne ha sempre uno speciale, `127.0.0.1`, che indica *sé stessa*: il *loopback*, con cui un client e un server sulla stessa macchina si parlano senza che nessun byte esca dal computer. È l'indirizzo con cui proveremo quasi tutto. Gli indirizzi si ricordano male, e per questo esistono i *nomi*: `localhost` è il nome di `127.0.0.1`, e `www.unibs.it` è il nome di una macchina da qualche parte; a tradurre un nome in un indirizzo pensa il DNS, un servizio che il sistema operativo interroga per noi. In Java sia il nome sia l'indirizzo si passano come stringa, e la classe `InetAddress` rappresenta il risultato della traduzione:

```
InetAddress a = InetAddress.getByName("www.unibs.it");
System.out.println(a.getHostAddress());        // 193.204.253.11, o quello che e'
System.out.println(InetAddress.getByName("localhost").isLoopbackAddress());  //
    true
```

Se il nome non esiste, o il DNS non risponde, `getByName` lancia `UnknownHostException`. È la prima eccezione della dispensa, e la prima regola: *nome sconosciuto* e *macchina che non risponde* sono due cose diverse, con due eccezioni diverse.

Porte. Un indirizzo identifica una macchina, ma su una macchina girano molti programmi, e molti possono usare la rete insieme. A distinguerli è la *porta*: un numero da 0 a 65535 che un programma sceglie quando si mette in ascolto, e che il client deve conoscere per raggiungerlo. Il servizio è identificato da indirizzo e porta insieme, come un palazzo e un interno. Alcune porte sono riservate per convenzione ai servizi noti (80 per il web, 443 per il web cifrato, 22 per `ssh`), e quelle sotto 1024 sui sistemi Unix richiedono i privilegi di amministratore: i nostri server useranno porte alte, come `1234`. Su una macchina una porta può avere un solo programma in ascolto: avviare due volte lo stesso server dà un errore preciso, `BindException`, che vedremo.

I livelli. Fra il nostro programma e il cavo ci sono diversi strati di software, ciascuno con un compito, e la tabella dice che cosa ne vede un programma Java. I livelli bassi non si vedono mai; il nostro codice sta tutto nel livello più alto, e parla con quello sotto attraverso i socket.

Livello	**Che cosa fa**	**Che cosa ne vede Java**
applicazione	il nostro protocollo, HTTP, la posta: che cosa vogliono dire i byte	il nostro programma
trasporto	TCP e UDP: porte, consegna affidabile (TCP) o singoli messaggi (UDP)	`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

TCP e UDP. Il livello di trasporto offre due servizi, e la scelta decide tutto il resto. *TCP* apre una *connessione* fra due programmi e dentro la connessione trasporta un flusso di byte, con tre garanzie: i byte arrivano tutti, nello stesso ordine in cui sono partiti, una volta sola. Se un pacchetto si perde per strada, TCP lo rimanda; se arriva fuori ordine, lo rimette a posto; il programma non vede nulla di tutto questo, vede un tubo. *UDP* non apre niente: manda singoli messaggi (*datagrammi*) a un indirizzo e una porta, e ognuno può arrivare, non arrivare, arrivare due volte o dopo quello partito dopo. In cambio costa meno e non ha bisogno di nessuna connessione. Quasi tutto ciò che somiglia a un dialogo (il web, la posta, la nostra chat) usa TCP; UDP serve quando un messaggio in ritardo vale meno di uno perso (il video in diretta, un gioco in tempo reale) o quando non si sa a chi parlare (cercare un server sulla rete locale, capitolo 6). Fino a quel capitolo la dispensa usa solo TCP, e l'immagine guida del filo con i due tubi è l'immagine di TCP.

Client e server. In una connessione TCP i due programmi hanno ruoli diversi solo all'inizio. Uno dei due, il *server*, si mette in ascolto su una porta e aspetta; l'altro, il *client*, conosce indirizzo e porta e *bussa*. Una volta aperta, la connessione è simmetrica: entrambi possono scrivere ed entrambi possono leggere, in qualunque ordine. Chi aspetta e chi bussa è la sola differenza, ed è la differenza fra le due classi che vediamo adesso.

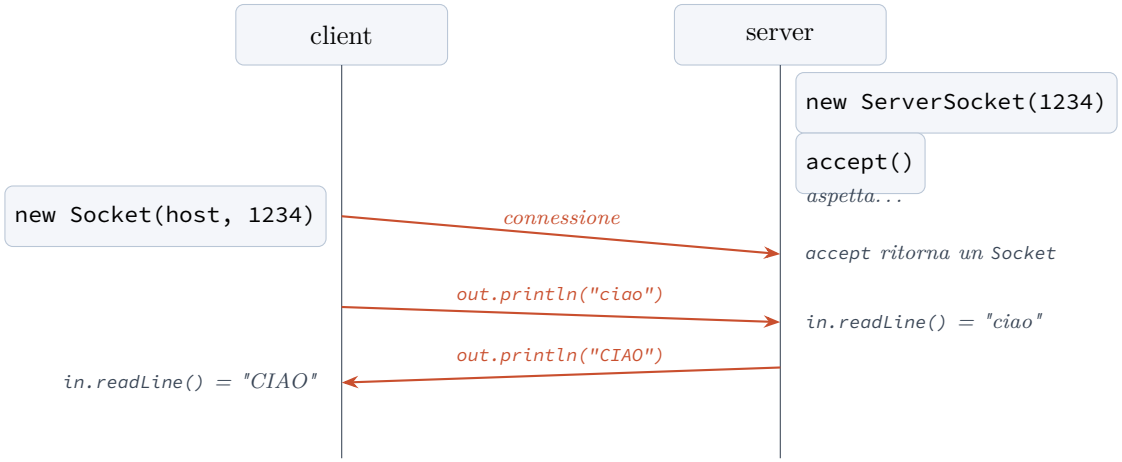
2.2 Il primo dialogo

Java ha una classe per ciascun ruolo. `ServerSocket` è la porta su cui si bussa: si crea con il numero della porta, e da quel momento il sistema operativo accetta le connessioni dirette a quella porta per conto del programma. Il suo metodo `accept()` *aspetta* che un client si colleghi e restituisce la connessione aperta, sotto forma di un `Socket`. `Socket` è l'estremo di una connessione: il client lo crea direttamente con indirizzo e porta del server (e la creazione è il bussare), il server lo riceve da `accept`. Da quel momento i due `Socket` sono i due capi dello stesso filo, e ciascuno ha i due tubi: `getInputStream()` per leggere ciò che l'altro scrive, `getOutputStream()` per scrivere ciò che l'altro leggerà.

Definizione 2.1: Socket e ServerSocket

Un *socket* è un estremo di una connessione di rete, con uno stream di ingresso e uno di uscita. In Java `Socket` rappresenta una connessione TCP aperta; `ServerSocket` rappresenta una porta in ascolto, e il suo metodo `accept()` blocca il thread che lo chiama finché un client non si collega, poi restituisce il `Socket` di *quella* connessione. Un `ServerSocket` non è una connessione: da uno stesso `ServerSocket` nascono tanti `Socket` quanti sono i client che si collegano.

La sequenza completa, dal lato del tempo, è questa:



Il server aspetta, il client bussa; da lì in poi i due tubi sono simmetrici.

2.2.1 Il server che fa l'eco

Il primo server risponde a ogni riga ricevuta con la stessa riga in maiuscolo, e chiude quando il client scrive `fine`. È il programma più semplice che mostri tutta la catena: una porta, una connessione, due stream incartati come facciamo con i file, un ciclo di lettura.

```java
import java.io.*;
import java.net.*;
import java.nio.charset.StandardCharsets;

public class ServerEco {
    public static void main(String[] args) throws IOException {
        int porta = 1234;
        System.out.println("In ascolto sulla porta " + porta);
        try (ServerSocket server = new ServerSocket(porta);
             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)) {
            System.out.println("Collegato: " + socket.getRemoteSocketAddress());
            String riga;
            while ((riga = in.readLine()) != null) {
                System.out.println("ricevuto: " + riga);
                out.println(riga.toUpperCase());
                if (riga.equals("fine")) break;
            }
        }
        System.out.println("Server chiuso");
    }
}
```

Leggiamolo riga per riga, perché ogni riga ha un motivo. `new ServerSocket(porta)` chiede al sistema operativo la porta: se è libera, da questo istante le connessioni a `1234` vengono accettate e messe in fila; se è occupata, `BindException`. `server.accept()` ferma il thread finché un client non si collega, e restituisce il `Socket` della connessione. `getRemoteSocketAddress()` dice da dove viene il client, indirizzo e porta. Poi i due stream: `getInputStream()` restituisce uno stream di *byte*, e noi vogliamo righe di testo, quindi lo incartiamo come un file, in un `InputStreamReader` che traduce byte in caratteri e in un `BufferedReader` che li raggruppa in righe; lo stesso in uscita con `PrintWriter`. I due argomenti in più, `true` e `UTF_8`, non sono decorazioni: sono due delle trappole del capitolo 1, e li spieghiamo nella sezione 2.3. Infine il ciclo, identico a quello con cui si legge un file: `readLine` restituisce la riga seguente, oppure `null` quando non ce ne saranno più, cioè quando il client ha chiuso. Il try-with-resources chiude tutto, in ordine inverso, qualunque cosa succeda.

2.2.2 Il client

Il client è il programma del capitolo 1, con le stesse due correzioni sugli stream. Legge una riga dalla tastiera, la manda, legge la risposta, la stampa; con `fine` esce dal ciclo e chiude.

```java
public class ClientEco {
    public static void main(String[] args) throws IOException {
        String host = args.length > 0 ? args[0] : "localhost";
        try (Socket socket = new Socket(host, 1234);
             BufferedReader in = new BufferedReader(new InputStreamReader(
                     socket.getInputStream(), StandardCharsets.UTF_8));
             PrintWriter out = new PrintWriter(
                     socket.getOutputStream(), true, StandardCharsets.UTF_8);
             BufferedReader tastiera = new BufferedReader(
                     new InputStreamReader(System.in))) {
            System.out.println("Collegato a " + socket.getRemoteSocketAddress());
            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, 1234)` fa tre cose in una: traduce il nome in indirizzo, apre la connessione, e aspetta che il server l'accetti (o che il sistema operativo del server la metta in fila: ci torniamo). Se nessuno ascolta su quella porta, `ConnectException`; se il nome non esiste, `UnknownHostException`. Il controllo su `risposta == null` è la cortesia che il client del capitolo 1 non aveva: se il server chiude per primo, `readLine` restituisce `null` e il client esce invece di stampare `null` e continuare.

Proviamo. In un terminale avviamo il server, in un altro il client, e scriviamo:

```
$ java ClientEco
Collegato a localhost/127.0.0.1:1234
ciao
> CIAO
come va?
> COME VA?
fine
> FINE
```

Il server, nel suo terminale, mostra le righe ricevute e poi "Server chiuso": dopo `fine` esce dal ciclo, il try-with-resources chiude il socket e il `ServerSocket`, e il programma termina. Un secondo `java ClientEco` ora riceve `ConnectException`: il server non c'è più. Riavviarlo per ogni client è scomodo, e nel capitolo 4 il server imparerà a restare in ascolto; per ora il punto è un altro: il dialogo funziona, e vale la pena guardarlo senza il nostro client in mezzo.

2.2.3 Il filo nudo: `nc`

Il client è solo un programma che scrive righe in un tubo e legge righe dall'altro. Qualunque programma che sappia fare questo può parlare con il nostro server, e ne esiste uno già pronto su ogni sistema: `nc` (*netcat*; su Windows si installa `ncat`, oppure si usa `telnet`). Gli si danno indirizzo e porta, e collega la tastiera al tubo di uscita e il tubo di ingresso al video:

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

È lo strumento più utile della dispensa, per due motivi. Il primo è pratico: per provare un server non serve scrivere un client, e per capire se un errore è nel server o nel client basta sostituire il client con `nc`. Il secondo è concettuale: `nc` non sa niente del nostro programma, e se riesce a parlarci è perché il nostro protocollo è fatto di righe di testo, che chiunque può scrivere. È una proprietà che la dispensa difenderà fino all'ultimo capitolo.

2.3 Che cosa succede davvero

Il dialogo funziona, e sembra identico alla lettura di un file. Non lo è, e le differenze sono esattamente i difetti del capitolo 1. Le passiamo in rassegna, perché ciascuna è una regola.

I buffer e il flush. Quando chiamiamo `out.println("ciao")`, la riga non parte. Il `PrintWriter` ha un buffer, lo stream del socket ne ha un altro nel sistema operativo, e TCP stesso ne ha uno: i byte vengono raccolti e spediti a gruppi, perché un pacchetto per ogni carattere costerebbe cento volte tanto. Il programma deve dire quando il gruppo è completo, con `flush()`. Il `true` nel costruttore di `PrintWriter` è l'*autoflush*: dopo ogni `println`, `printf` e `format` (non dopo `print`) il buffer viene svuotato da solo. Senza, la riga resta nel buffer finché il buffer non si riempie o lo stream non viene chiuso: il server "risponde" solo alla chiusura, e il client, che aspetta la risposta per mandare la riga seguente, aspetta per sempre.

Attenzione 2.1: Il `PrintWriter` che non manda niente

```
PrintWriter out = new PrintWriter(socket.getOutputStream());   // senza
    autoflush
out.println("ciao");          // resta nel buffer: l'altro non riceve nulla
```

Il programma non dà nessun errore: la riga è scritta, dal punto di vista del `PrintWriter`, e verrà consegnata alla chiusura, quando ormai non serve più. I due programmi si aspettano a vicenda, ciascuno convinto che tocchi all'altro. Vale per ogni `Writer` e `OutputStream` con un buffer (`BufferedWriter`, `BufferedOutputStream`, `DataOutputStream` sopra uno di questi): o si attiva l'autoflush, o si chiama `flush()` alla fine di ogni messaggio. È l'errore più frequente nei primi programmi di rete, e il più difficile da riconoscere perché non succede niente.

C'è un secondo tradimento del `PrintWriter`, più sottile. I suoi metodi non lanciano `IOException`: se il socket è chiuso o l'altro è sparito, `println` non si lamenta, e il programma lo scopre solo chiamando `checkError()`. Per il nostro uso va bene, perché un server non deve

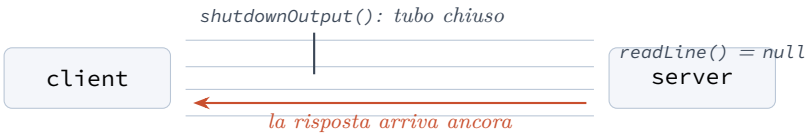
morire se un client se ne va a metà di una risposta; ma vuol dire che *scrivere* non è mai il modo per accorgersi che l'altro non c'è più. Il modo è *leggere*.

Righe e charset. Sul filo passano byte, e una lettera diventa byte secondo una *codifica* (*charset*). Per le lettere senza accento tutte le codifiche in uso sono d'accordo; per `è`, `à` e gli altri caratteri oltre i primi 128 no: in UTF-8 una `è` sono due byte, in ISO-8859-1 e nelle codifiche di Windows è uno, e un altro. Se i due lati non usano la stessa codifica, "perché" arriva rotto, come nel capitolo 1. Il costruttore `InputStreamReader(stream)` senza charset usa quello *predefinito* della JVM: dalla versione 18 è UTF-8 ovunque, ma su un Java più vecchio, o in un programma scritto in un altro linguaggio, è quello del sistema. La regola è di non lasciare la domanda al caso: si dichiara `StandardCharsets.UTF_8` su entrambi i lati, sempre, e il protocollo dice che il testo è UTF-8. (Resta la tastiera: su Windows la console può consegnare a `System.in` byte in una codifica sua; non è un problema di rete, e si risolve con `System.console()` o con l'opzione `-Dstdin.encoding`.)

La *riga* è la seconda parte dell'accordo. TCP consegna un flusso di byte senza confini: due `println` da una parte possono arrivare in un pacchetto solo dall'altra, e un `println` lungo può arrivare in due. Il programma deve decidere da solo dove finisce un messaggio, e `readLine` lo fa per noi usando il carattere di fine riga (`\n`, o `\r\n`, che accetta entrambi). È la *cornice* più semplice che esista, e per i protocolli di testo è quella giusta; il capitolo 6 ne vedrà un'altra per i byte.

`null`, la fine normale. `readLine` restituisce `null` quando lo stream è finito, e lo stream di un socket finisce quando l'altro lato ha chiuso la sua uscita. Non è un errore: è il modo in cui TCP dice "non arriverà più niente". Il ciclo `while ((riga = in.readLine()) != null)` è la forma canonica di ogni lettore di rete, e il codice dopo il ciclo è quello che si esegue quando l'altro se ne è andato con educazione.

La chiusura e la mezza chiusura. Chiudere un socket (o uno dei suoi stream: `in.close()` e `out.close()` chiudono il socket intero) manda all'altro lato la fine dello stream in ingresso e rende inutilizzabile il nostro. C'è però un caso in cui vogliamo dire "ho finito di scrivere" e continuare a leggere: un client che manda una richiesta e vuole la risposta completa, per esempio un file da trasferire. Per questo esiste `shutdownOutput()`: chiude il solo tubo di uscita, l'altro lato riceve `null` e sa che può rispondere, e il nostro tubo di ingresso resta aperto.



La mezza chiusura: ho finito di scrivere, ma ascolto ancora.

Chi sparisce senza salutare. Se l'altro programma termina in modo pulito, il sistema operativo chiude il socket per lui e noi leggiamo `null`. Se invece sparisce davvero (il processo ucciso, il cavo staccato, la macchina spenta) le cose cambiano: la nostra lettura riceve una `SocketException` con il messaggio *Connection reset* quando il sistema remoto ce lo comunica, oppure resta bloccata a tempo indefinito se nessuno ce lo comunica, perché TCP non manda nulla quando non c'è nulla da mandare. Per il primo caso basta un `catch`: non è un bug, è un cliente che se ne è andato male. Per il secondo esiste il *timeout*: `socket.setSoTimeout(ms)` fa lanciare `SocketTimeoutException` a una lettura che aspetta più di `ms` millisecondi, e il programma decide che cosa fare (chiudere, riprovare, mandare un "ci sei?"). Anche l'apertura può aspettare troppo, quando l'indirizzo esiste ma la macchina non risponde: `new Socket()`

vuoto e poi `connect(new InetSocketAddress(host, porta), 3000)` limitano l'attesa a tre secondi.

Osservazione 2.1

Le regole di questa sezione si riassumono in una frase: *un socket non è un file*. Un file è tutto lì, e leggerlo o scriverlo non dipende da nessun altro. Un socket ha un altro programma all'altro capo, che può essere lento, assente, o avere un'idea diversa della codifica; i byte possono fermarsi in un buffer da una parte o dall'altra; la fine può arrivare con grazia (`null`) o con violenza (*reset*) o non arrivare mai (timeout). Il codice che li tratta tutti è lungo il doppio di quello che "funziona" al primo tentativo, ed è l'unico corretto.

2.4 Quando va storto

Le eccezioni della rete sono poche e dicono cose precise. Vale la pena conoscerle per nome, perché il messaggio d'errore è spesso l'unica informazione che si ha quando un programma di rete non funziona, e ciascuna indica un posto diverso dove guardare.

Eccezione	**Che cosa vuol dire, e dove guardare**
`UnknownHostException`	il nome non si traduce in un indirizzo: errore di battitura, oppure nessuna rete, oppure DNS non raggiungibile
`ConnectException` (*refused*)	la macchina c'è, ma nessuno ascolta su quella porta: il server non è avviato, o ascolta su un'altra porta, o su un altro indirizzo
`SocketTimeoutException` (in `connect`)	la macchina non risponde affatto: indirizzo sbagliato, macchina spenta, o un firewall che butta via i pacchetti senza rispondere
`BindException`	la porta è già usata: il server è avviato due volte, o un altro programma occupa la porta; oppure è sotto 1024 e mancano i privilegi
`SocketException` (*reset*)	l'altro lato è sparito senza chiudere; oppure abbiamo scritto su un socket che l'altro ha già chiuso
`SocketTimeoutException` (in lettura)	`setSoTimeout` è scaduto: nessuna riga entro il tempo fissato

Fuori dalle eccezioni c'è un errore che non ne lancia nessuna, e che in aula capita a tutti: il server funziona con `localhost` e non dall'altra macchina. Le cause sono due. La prima è l'indirizzo: `localhost` è la macchina stessa, e il client sull'altro computer deve usare l'indirizzo che il server ha *sulla rete locale* (`192.168.x.y`, `10.x.y.z`); lo si legge dalle impostazioni di rete del sistema, oppure da Java, perché `InetAddress.getLocalHost()` risponde spesso con `127.0.0.1` e non serve:

```
for (NetworkInterface ni :
    Collections.list(NetworkInterface.getNetworkInterfaces()))
    for (InetAddress a : Collections.list(ni.getInetAddresses()))
        if (a instanceof Inet4Address && !a.isLoopbackAddress())
            System.out.println(ni.getName() + ": " + a.getHostAddress());
```

La seconda causa è il *firewall*: macOS e Windows chiedono il permesso la prima volta che un programma si mette in ascolto, e se si risponde no (o si chiude la finestra) i pacchetti degli altri vengono scartati in silenzio: il client riceve un timeout, non un rifiuto. Alcune reti Wi-Fi pubbliche, inoltre, impediscono ai client di parlarsi fra loro (*isolamento*), e allora non c'è programma che tenga: serve la rete dell'aula o un telefono in condivisione.

Ricetta: Provare un server da un'altra macchina

1. avviare il server e, sulla stessa macchina, provarlo con `nc localhost porta`: se non risponde, l'errore è nel server;
2. leggere l'indirizzo della macchina del server sulla rete locale (dalle impostazioni di rete, o con il frammento qui sopra), e accettare la richiesta del firewall se compare;
3. dall'altra macchina, `nc indirizzo porta`: *refused* vuol dire porta sbagliata o server fermo, un'attesa senza risposta vuol dire indirizzo sbagliato, firewall o rete isolata;
4. solo quando `nc` funziona, provare il client scritto da noi: se non va, il problema è nel client.

2.5 L'esempio guida: Saluto

L'eco è un dialogo in cui il server non deve ricordare nulla. Il primo passo verso WhatsPAJC è un server che si ricorda *chi* gli parla: appena il client si collega, il server lo saluta e gli chiede il nome; da quel momento ogni riga ricevuta viene rimandata con il nome davanti, nella forma che useremo per tutta la dispensa, `[Marco] ciao`. È poco, ma contiene già tre cose nuove: il server parla per primo, le due parti condividono uno stato (il nome), e c'è un ordine da rispettare.

```
public class ServerSaluto {
    public static void main(String[] args) throws IOException {
        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)) {
            out.println("Benvenuto su WhatsPAJC. Come ti chiami?");
            String nome = in.readLine();
            if (nome == null || nome.isBlank()) return;    // se ne e' andato
            out.println("Ciao " + nome + ", scrivi pure (fine per uscire)");
            String riga;
            while ((riga = in.readLine()) != null && !riga.equals("fine"))
                out.println("[" + nome + "] " + riga);
            out.println("Ciao " + nome + "!");
        }
    }
}
```

Il client dell'eco non va più bene, e il motivo è istruttivo: aspetta una riga di risposta per ogni riga mandata, mentre questo server manda una riga *prima* di ricevere qualunque cosa, e due righe (`Ciao Marco...` e poi i messaggi) dopo il nome. Con `ClientEco` il dialogo si sfasa subito: il client manda il nome e legge il benvenuto, che era già lì; poi manda la prima riga e

legge il "Ciao Marco"; da lì in poi ogni risposta è quella della riga precedente. Con `nc`, che non ha nessuna idea di turni, funziona tutto:

```
$ nc localhost 1234
Benvenuto su WhatsPAJC. Come ti chiami?
Marco
Ciao Marco, scrivi pure (fine per uscire)
buongiorno a tutti
[Marco] buongiorno a tutti
fine
Ciao Marco!
```

Il client in Java che si comporta come `nc` ha bisogno di leggere e scrivere senza un ordine prestabilito, cioè di due thread, e lo scriviamo nel capitolo 3, dove il problema dei turni è il punto di partenza. Qui fermiamoci su quello che Saluto ha già: un *accordo*. Il client deve sapere che la prima riga è una domanda, che la seconda riga che manda è il nome, che `fine` chiude. Non sta scritto da nessuna parte se non nei due programmi, e finché sono due e li scriviamo noi va bene; nel capitolo 3 l'accordo prende un nome, una tabella e una classe.

2.5.1 Il nome delle cose in rete: `URI`

Un'ultima cosa che un programma deve saper fare con la rete è leggere un indirizzo scritto come lo scrivono le persone:

`https://www.unibs.it:443/didattica/corsi?anno=2026#pajc`

È un *URI* (*Uniform Resource Identifier*), e tiene insieme, con una sintassi fissa, tutto ciò che abbiamo visto: lo *schema* (il protocollo di applicazione, `https`), l'*host* e la *porta* (se manca, quella convenzionale dello schema), il *percorso*, la *query* e il *frammento*. La classe `java.net.URI` lo scompone:

```java
URI u = URI.create("https://www.unibs.it:443/didattica/corsi?anno=2026#pajc");
System.out.println(u.getScheme());     // https
System.out.println(u.getHost());       // www.unibs.it
System.out.println(u.getPort());       // 443  (-1 se non indicata)
System.out.println(u.getPath());       // /didattica/corsi
System.out.println(u.getQuery());      // anno=2026
System.out.println(u.getFragment());   // pajc
```

La classe più vecchia, `URL`, fa la stessa cosa e in più sa aprire una connessione (`openStream()`); il suo costruttore è però deprecato da Java 20, perché accettava indirizzi malformati, e oggi si costruisce con `URI.create(...).toURL()`. Per scaricare una pagina il capitolo 6 userà `HttpClient`, che vuole una `URI`; qui basta sapere che dietro un indirizzo web ci sono un host, una porta e un protocollo di testo, e che con `nc www.unibs.it 80` ci si può parlare a mano. Lo faremo.

Esercizi

Esercizio 2.1: Che cosa restituisce `accept` (⋆)

`server` è un `ServerSocket` creato correttamente. Dire quale affermazione è vera e perché le altre sono false: (a) `server.accept()` restituisce un `ServerSocket` pronto a ricevere connessioni; (b) restituisce il numero di client in attesa; (c) restituisce `null` se nessun client è in attesa; (d) restituisce un `Socket` che rappresenta la connessione con un singolo client.

Esercizio 2.2: Senza flush (⋆)

In `ServerEco` si toglie il `true` dal costruttore del `PrintWriter`. Descrivere, nell'ordine, che cosa vede il client che scrive `ciao`, `come va?` e `fine`: che cosa compare sul suo video, quando, e come termina il programma.

Esercizio 2.3: Chi chiude (⋆)

Nel dialogo fra `ServerEco` e `ClientEco`, dopo `fine`, dire in che ordine avvengono le chiusure e che cosa vedrebbe ciascuno dei due se l'altro chiudesse per primo. Che cosa cambia se il client, invece di `fine`, viene terminato con Ctrl+C?

Esercizio 2.4: Contare i socket (⋆)

Un server tiene un `ServerSocket` sulla porta 1234 e accetta, uno dopo l'altro, tre client. Quanti oggetti `ServerSocket` e quanti `Socket` esistono lato server? Quanti lato client, in tutto? E quante porte sono in uso sulla macchina del server?

Esercizio 2.5: Tre eccezioni (⋆⋆)

Per ciascuna situazione dire quale eccezione riceve il client e dove va cercato il problema: (a) `new Socket("localhots", 1234)`; (b) `new Socket("localhost", 1234)` con il server avviato sulla porta 1235; (c) `new Socket("10.0.0.99", 1234)` con nessuna macchina a quell'indirizzo; (d) il server avviato una seconda volta sulla stessa porta.

I problemi che seguono chiedono un programma completo, in ordine crescente di difficoltà. Le soluzioni sono nel repository delle soluzioni del corso, nel progetto e nel package indicati.

Esercizio 2.6: Il server dell'ora (⋆)

Scrivere un server che resta in ascolto sulla porta 1234 e, ogni volta che un client si collega, gli manda una riga con l'ora corrente (`LocalTime.now()`), chiude la connessione e torna ad aspettare il client successivo, senza mai terminare. Provarlo con `nc` tre volte di seguito, e poi scrivere il client Java che si collega, stampa la riga e termina.

Soluzione: progetto `sol-d06-c02`, package `it.unibs.pajc.redolfi.e06`.

Attenzione 2.2: Il `ServerSocket` nel try-with-resources

Se il `ServerSocket` sta nello stesso try-with-resources del `Socket` del client, viene chiuso insieme a lui dopo il primo: il secondo `nc` riceve *refused*. Il `ServerSocket` si apre una volta, fuori dal ciclo; dentro il ciclo si apre e si chiude il solo `Socket`.

Esercizio 2.7: Eco con contatore (⋆)

Modificare `ServerEco` perché risponda a ogni riga con il numero d'ordine e la riga in maiuscolo (`1: CIAO`, `2: COME VA?`), e perché con `fine` risponda con il totale delle righe ricevute prima di chiudere. Il client stampa tutto ciò che riceve, compresa l'ultima riga.

Soluzione: progetto `sol-d06-c02`, package `it.unibs.pajc.redolfi.e07`.

Esercizio 2.8: La calcolatrice remota (⋆⋆)

Scrivere un server che riceve righe della forma `ADD 2 3`, `SUB 10 4`, `MUL 6 7`, `DIV 1 0` e risponde con il risultato su una riga, oppure con `ERR` seguito dal motivo (operazione sconosciuta, numero non valido, divisione per zero). Il server non deve mai terminare per una riga sbagliata. Scrivere il client, che legge le righe dalla tastiera e stampa le risposte.

Soluzione: progetto `sol-d06-c02`, package `it.unibs.pajc.redolfi.e08`.

Attenzione 2.3: L'eccezione che chiude il server

`Integer.parseInt("tre")` lancia `NumberFormatException`; se nessuno la cattura il thread del server esce dal ciclo di lettura e chiude la connessione, e il client riceve `null` invece di un `ERR`. Ogni riga che arriva dalla rete è un input dell'utente, e va trattata con lo stesso sospetto.

Esercizio 2.9: Il benvenuto a più righe (⋆⋆)

Scrivere un server che, appena il client si collega, gli manda un messaggio di benvenuto di tre o quattro righe (il nome del server, l'ora, le istruzioni) seguito da una riga *vuota*, e poi si comporta come l'eco. Scrivere il client che legge il benvenuto riga per riga finché non trova la riga vuota, lo stampa, e solo allora comincia il dialogo a turno. Spiegare nel commento perché la riga vuota è necessaria e che cosa succederebbe senza.

Soluzione: progetto `sol-d06-c02`, package `it.unibs.pajc.redolfi.e09`.

Risposte

Risposta 2.1. La (d). Un `ServerSocket` è la porta in ascolto, non una connessione, e `accept` non ne crea altri (a). Non conta nessuno: il numero di client in fila lo conosce solo il sistema operativo (b). Non restituisce mai `null`: se nessuno è in attesa, *blocca* il thread finché qualcuno non arriva, e l'unico modo per non aspettare per sempre è `setSoTimeout` sul `ServerSocket`, che fa lanciare `SocketTimeoutException` (c). La connessione con un singolo client è un `Socket`, uguale a quello che il client ha creato dall'altra parte.

Risposta 2.2. Il client manda `ciao` e chiama `in.readLine()`: il server ha scritto `CIAO` nel buffer del `PrintWriter`, ma niente è partito, e il client aspetta. Sul video del client non compare nulla, e il client non legge più la tastiera, perché è fermo nella lettura dal socket: le righe `come va?` e `fine` non possono nemmeno essere scritte (il terminale le accetta, ma il programma non le legge). I due programmi restano così per sempre, ciascuno in attesa dell'altro, senza nessun errore. Se si termina il client con Ctrl+C, il server legge `null`, esce dal ciclo, chiude il `PrintWriter` e *allora* svuota il buffer, su un socket che nessuno legge più.

Risposta 2.3. Il client manda `fine` e legge `FINE`; il server, dopo averla mandata, esce dal ciclo con il `break` e chiude (socket, poi `ServerSocket`). Il client, dopo aver stampato `FINE`, esce a sua volta per il suo `break` e chiude. Se il server chiudesse per primo e il client provasse a leggere ancora, leggerebbe `null`; se il client chiudesse per primo e il server provasse a leggere, leggerebbe `null`, e se provasse a scrivere il `PrintWriter` non direbbe nulla. Con Ctrl+C sul client il sistema operativo chiude il socket per conto del processo terminato: il server legge `null` esattamente come dopo una chiusura pulita; il *reset* arriva solo se il client muore *dopo* che il server ha già scritto qualcosa che il client non ha letto, oppure se il cavo viene tolto, nel qual caso non arriva niente e il server resta in `readLine` finché un timeout non lo libera.

Risposta 2.4. Lato server un solo `ServerSocket` e tre `Socket`, uno per client, anche se uno alla volta (ciascuno chiuso prima di accettare il successivo). Lato client tre `Socket` in tutto, uno per programma. Sulla macchina del server è in uso la porta 1234, in ascolto, e per ogni connessione aperta la stessa porta 1234 compare nell'estremo locale della connessione: una connessione TCP è identificata dalla quaterna indirizzo e porta locali, indirizzo e porta remoti, e il server distingue i client dall'indirizzo e dalla porta *del client*, che il sistema operativo del client sceglie a caso fra quelle libere (la porta *effimera*, che `getPort()` mostra dal lato del server).

Risposta 2.5. (a) `UnknownHostException`: il nome non esiste; errore di battitura nel client. (b) `ConnectException` *Connection refused*: la macchina risponde, ma nessuno ascolta sulla 1234; porta sbagliata in uno dei due. (c) Nessuna risposta: dopo il timeout del sistema (decine di secondi) o quello di `connect(addr, ms)`, `SocketTimeoutException` (o `ConnectException` con *timed out*); indirizzo sbagliato o macchina assente. (d) `BindException` *Address already in use*, sul *server*, al secondo avvio: il client non se ne accorge e parla con il primo.

Capitolo 3

Mettersi d'accordo: il protocollo

Il capitolo 2 ci ha lasciato con un server, *Saluto*, che parla per primo e manda due righe dove il client ne aspettava una, e con un client che per questo si sfasa alla seconda riga. Il filo funziona; quello che manca è l'accordo. Il problema di questo capitolo è la seconda domanda della dispensa: come si mettono d'accordo due programmi su che cosa dirsi. Alla fine avremo progettato il protocollo di WhatsPAJC, lo avremo scritto come codice Java che si prova senza nessuna rete, e avremo un server che capisce tutti i comandi della chat (per un client solo: i molti arrivano nel capitolo 4) e un client di console che ascolta mentre l'utente scrive.

3.1 Un accordo su che cosa dirsi

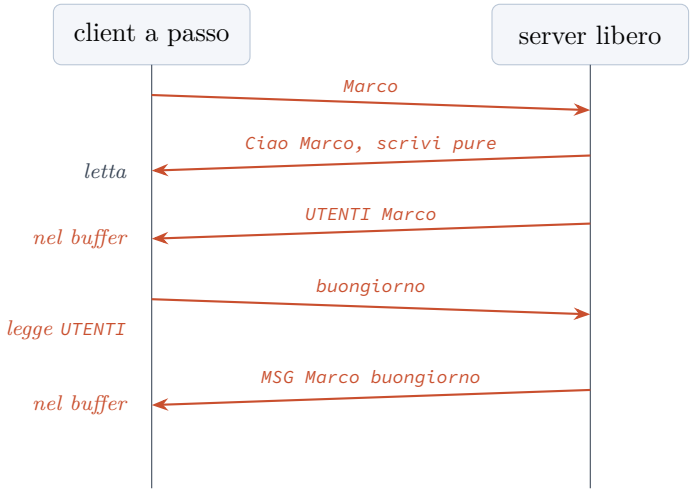
Nel dialogo fra `ServerEco` e `ClientEco` c'era già un accordo, anche se non lo abbiamo chiamato così: ogni messaggio è una riga; il client manda una riga e il server ne risponde una; `fine` chiude. Chi scrive un client per quel server deve conoscere queste tre regole, e nient'altro. Sono il suo *protocollo*.

Definizione 3.1: Protocollo

Un *protocollo* è l'insieme delle regole che due programmi condividono per capirsi: come si delimita un messaggio (la *cornice*), quali messaggi esistono e come sono fatti (la *sintassi*), che cosa vuol dire ciascuno e che cosa si risponde (la *semantica*), e chi può parlare in quale momento (il *turno*). Il protocollo non è nel filo: è nei due programmi, e vale solo se entrambi lo rispettano.

La prima scelta, quella che decide la forma di tutto il resto, riguarda il turno. Un protocollo *a passo* alterna una richiesta e una risposta: il client parla, il server risponde, il client parla di nuovo. È il caso dell'eco e della calcolatrice remota, ed è il caso di HTTP, che vedremo nella sezione 3.5: semplice da scrivere, perché ogni lato sa sempre che cosa aspettarsi, e sufficiente ogni volta che il client *chiede* qualcosa. Una chat non è così. Quando Giulia scrive, il suo messaggio deve arrivare a Marco *adesso*, non quando Marco avrà chiesto qualcosa; il server deve poter parlare al client in qualunque momento, e il client deve ascoltare sempre. È un protocollo *libero*: i due lati scrivono quando vogliono, e nessuna riga è la risposta di un'altra, salvo dove il protocollo lo dice.

Un client a passo collegato a un server libero si blocca, e vale la pena vedere come, sulla linea del tempo. Il client di `ClientEco` manda il nome e aspetta *una* riga; il server di Saluto ne manda due, il benvenuto e il "Ciao Marco". Il client ne legge una e torna alla tastiera; la seconda resta nel buffer. Alla riga seguente il client manda il messaggio e legge… il "Ciao Marco" avanzato. Da lì ogni risposta è quella della riga precedente. E se il server mandasse una riga in *meno* di quelle attese (per esempio nessuna risposta a un comando ignoto), il client resterebbe fermo in `readLine` per sempre, e il server fermo ad aspettare la riga seguente: è l'*interblocco di protocollo*, due programmi corretti ciascuno per sé che si aspettano a vicenda.



Il client legge una riga per ogni riga mandata: da qui in poi è sempre indietro di una.

La cura è togliere al client l'idea di turno: un *lettore* che riceve le righe dal server e le stampa appena arrivano, e uno *scrittore* che prende le righe dalla tastiera e le manda appena scritte, senza che nessuno dei due aspetti l'altro. Sono due cicli bloccanti, ciascuno fermo nel suo `readLine`, e due cicli bloccanti vogliono due thread. È il primo posto della dispensa in cui i thread sono indispensabili e non un'ottimizzazione, e il client di console che ne esce è quello che useremo fino al capitolo 5:

```
public class ClientConsole {
    public static void main(String[] args) throws Exception {
        String host = args.length > 0 ? args[0] : "localhost";
        try (Socket socket = new Socket(host, 1234);
             BufferedReader in = new BufferedReader(new InputStreamReader(
                     socket.getInputStream(), StandardCharsets.UTF_8));
             PrintWriter out = new PrintWriter(
                     socket.getOutputStream(), true, StandardCharsets.UTF_8);
             ExecutorService exec = Executors.newFixedThreadPool(2)) {
            Future<?> lettore = exec.submit(() -> {
                String riga;
                while ((riga = in.readLine()) != null)
                    System.out.println(riga);
                System.out.println("[connessione chiusa dal server]");
                return null;
            });
```

Il lettore è un `Callable` che restituisce `null` (così può lanciare `IOException` senza un `try` intorno), e il suo `Future` serve a chi resta per sapere quando la connessione è finita. Lo scrittore e la chiusura:

```
exec.submit(() -> {
                BufferedReader tastiera = new BufferedReader(
                        new InputStreamReader(System.in));
                String riga;
                while (!lettore.isDone() && (riga = tastiera.readLine()) != null)
                    out.println(riga);
                return null;
            });
            lettore.get();                  // main aspetta la fine della lettura
        }
        System.exit(0);                     // la tastiera e' ferma in readLine
    }
}
```

Due compiti su un executor da due thread, come in *Thread e concorrenza*. Il lettore stampa tutto ciò che arriva, finché il server non chiude; lo scrittore manda tutto ciò che l'utente scrive. `main` non ha niente da fare e aspetta il lettore con `get()`: quando il server chiude, il try-with-resources chiude socket ed executor. Resta un dettaglio sgradevole, e vale la pena non nasconderlo: lo scrittore è fermo in `tastiera.readLine()`, e non c'è modo di interromperlo da fuori, perché la lettura da `System.in` non risponde a `interrupt()`; il `System.exit(0)` finale chiude il programma al posto suo. Il client con la finestra del capitolo 5 non avrà questo problema, perché l'utente scrive in un campo di testo e non in uno stream.

Con questo client, Saluto funziona come con `nc`: il benvenuto compare subito, il nome parte subito, i messaggi tornano con il nome davanti. E un server che manda tre righe, o nessuna, non sfasa più niente, perché il client non conta le righe: le stampa. La domanda "quante righe mi aspetto" è sparita; resta la domanda "che cosa vuol dire ciascuna", ed è il protocollo vero.

3.2 Il protocollo di WhatsPAJC

Progettiamo ora l'accordo della chat, per intero, prima di scrivere una riga di codice che lo usi. Le scelte sono poche e ciascuna ha un motivo.

La cornice. Un messaggio è una riga di testo, codificata in UTF-8 e terminata da `\n`. Lo abbiamo già visto: è la cornice più semplice, `readLine` la riconosce da sola, e `nc` la sa scrivere. La conseguenza è che il testo di un messaggio *non può contenere* un a capo: il client che lo lascia passare spedisce due righe, e il server le legge come due messaggi. È una regola del protocollo e il client la farà rispettare.

I comandi del client. Le righe che il client manda al server cominciano con una barra quando sono un comando, come nelle chat che gli studenti usano; una riga senza barra è un messaggio per tutti, così scrivere è la cosa più naturale. Il primo campo decide il tipo; il testo è tutto il resto della riga, spazi compresi.

Riga dal client	Significato
`/nome Marco`	sceglie il nome; deve essere il primo comando; lettere, cifre e `_`, da 2 a 16 caratteri
`/chi`	chiede l'elenco di chi è collegato
`/a Giulia testo`	manda `testo` a Giulia soltanto
`/esci`	saluta e chiude
`/aiuto`	chiede l'elenco dei comandi
`testo` (senza barra)	manda `testo` a tutti

Le righe del server. Le righe che il server manda al client cominciano con una parola maiuscola che dice il tipo, seguita dai campi; il testo libero è sempre l'ultimo campo, così gli spazi dentro non creano ambiguità. Le parole maiuscole sono per il programma, non per la persona: il client con la finestra le tradurrà in ciò che l'utente vede.

Riga dal server	Significato
`INFO testo`	un testo da mostrare: il benvenuto, l'aiuto
`BENVENUTO Marco`	il nome è accettato; da qui in poi si può scrivere
`UTENTI Marco,Giulia,Sara`	l'elenco di chi è collegato, separato da virgole
`MSG Marco testo`	un messaggio di Marco a tutti (anche a Marco stesso)
`PRIV Marco Giulia testo`	un messaggio di Marco a Giulia; lo ricevono entrambi
`ENTRA Marco`	Marco si è collegato
`ESCE Marco`	Marco se n'è andato
`ERR motivo`	l'ultimo comando non è andato: il motivo in chiaro
`CIAO`	il server chiude la connessione subito dopo

Il turno. Il protocollo è libero: il server può mandare `MSG`, `ENTRA` ed `ESCE` in qualunque momento, e il client non deve mai supporre che la riga che legge sia la risposta a quella che ha mandato. Due sole righe hanno un ordine: alla connessione il server manda un `INFO` di benvenuto, e il client *deve* mandare `/nome` prima di qualunque altra cosa (altrimenti `ERR`). Un dialogo completo, dal lato di Marco:

```
INFO Benvenuto su WhatsPAJC. Scegli un nome con /nome
/nome Marco
BENVENUTO Marco
/chi
UTENTI Marco,Giulia
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
```

Due scelte meritano una parola. Il server rimanda `MSG` anche al mittente, invece di lasciare che il client mostri da solo ciò che ha scritto: così tutti i client vedono i messaggi nello stesso ordine, quello in cui il server li ha ricevuti, e un messaggio compare sul video di chi lo ha

scritto solo quando è davvero partito. Per lo stesso motivo `PRIV` porta mittente e destinatario e arriva a entrambi. La seconda scelta è che il server risponde a *ogni* comando che non va con un `ERR` che dice il motivo, e non chiude mai la connessione per un errore del client: un comando sbagliato è un fatto normale, non una violazione.

Osservazione 3.1

Un protocollo si legge meglio di quanto si scriva, e il modo per capire se è ben fatto è provarlo con `nc` prima di scrivere il client: se il dialogo è comprensibile a una persona che lo scrive a mano, lo sarà anche per il programma. I protocolli di testo che hanno retto decenni (SMTP per la posta, HTTP, IRC per le chat) sono tutti così: righe, una parola che dice il tipo, campi separati da spazi, testo libero in fondo. Il nostro ne è una versione in piccolo, e chi lo capisce sa leggere anche quelli.

3.3 Il protocollo come codice

Un protocollo scritto solo in una tabella finisce sparso nel programma: uno `split` qui, un `startsWith` là, una catena di `if` che confronta il primo campo con ogni comando, e il testo dell'aiuto scritto a mano in un altro posto ancora. Quando cambia una riga del protocollo bisogna trovarli tutti, e quando si aggiunge un comando bisogna ricordarsi di ogni posto. La forma giusta è quella che il protocollo ha già sulla carta: *una tabella*, con il comando come chiave e, per ogni comando, ciò che serve a eseguirlo. Scritta in Java, la tabella è una mappa, e il valore è un record.

3.3.1 La tabella dei comandi

Che cosa deve sapere il server di un comando? Quanti argomenti vuole (nessuno per `/chi`, uno per `/nome`, due per `/a`, di cui l'ultimo è il testo con tutti i suoi spazi), come si spiega all'utente che chiede `/aiuto`, e che cosa fa. Tre cose, un record:

```java
public final class Protocollo {

    /** Una riga della tabella: argomenti, aiuto, che cosa fare. */
    public record Azione(int argomenti, String aiuto,
                         BiConsumer<Sessione, String[]> esegui) {}

    /** I comandi, nell'ordine in cui /aiuto li elenca. */
    public static final Map<String, Azione> COMANDI = new LinkedHashMap<>();

    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());
    }

    private static void azione(String comando, int argomenti, String aiuto,
                               BiConsumer<Sessione, String[]> esegui) {
        COMANDI.put(comando, new Azione(argomenti, aiuto, esegui));
    }
```

La chiave della mappa è il comando così come arriva sul filo; il valore dice quanti argomenti, come si spiega, che cosa fa. Il "che cosa fa" è una funzione, una `BiConsumer` che riceve la *sessione* del client che ha scritto il comando (la classe che tiene i suoi stream e il suo nome, fra poco) e gli argomenti già separati: `(s, a) -> s.privato(a[0], a[1])` dice che `/a` si esegue chiamando il metodo `privato` della sessione con il destinatario e il testo. È il pattern *Command* con le lambda, e i suoi pezzi sono tutti noti da *Dal nome alla funzione*: una mappa, un'interfaccia funzionale, cinque lambda. La mappa è una `LinkedHashMap` perché conserva l'ordine di inserimento, e `/aiuto` elencherà i comandi nell'ordine in cui sono scritti qui; la tabella è `static` e `final` perché è il protocollo, uguale per tutte le sessioni, e non tiene nessuno stato.

Le righe del server non hanno bisogno di una tabella: è il server a comporle, e sa già che sono ben fatte. `"MSG " + nome + " " + testo` è tutto il codice che serve, e sta nel metodo che manda il messaggio; la parola maiuscola in testa è la chiave che il client, dall'altra parte, userà per la sua tabella (capitolo 5).

3.3.2 Da riga a comando

Chi usa la tabella è la sessione: per ogni riga che arriva dal client, la spezza in comando e resto, cerca il comando nella tabella, controlla gli argomenti ed esegue l'azione. Il metodo è corto, e ogni riga ha un motivo:

```
    /** Da riga a comando: la tabella decide. Non lancia mai: al massimo risponde
        ERR. */
    void elabora(String riga) {
        if (riga.isBlank()) return;
        if (!riga.startsWith("/")) { tutti(riga.strip()); return; }   // a tutti
        String[] campi = riga.strip().split(" ", 2);                   // comando
    e resto
        Azione a = Protocollo.COMANDI.get(campi[0].toLowerCase());
        if (a == null) { invia("ERR comando sconosciuto: " + campi[0]); return; }
        String resto = campi.length > 1 ? campi[1] : "";
        String[] arg = Protocollo.argomenti(resto, a.argomenti());
        if (arg == null) { invia("ERR uso: " + a.aiuto()); return; }
        a.esegui().accept(this, arg);
    }
```

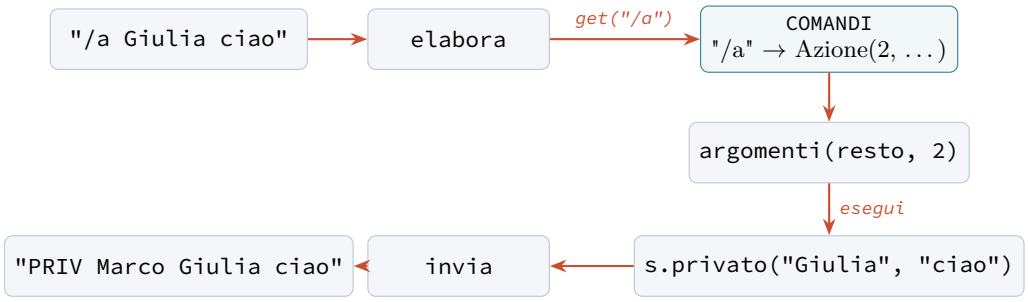
Una riga senza barra è un messaggio per tutti, e non passa dalla tabella. Per le altre, `split(" ", 2)` con il *limite* separa il primo campo da tutto il resto, spazi compresi; il comando è confrontato in minuscolo, così `/A Giulia ciao` vale come `/a`. Se il comando non è nella tabella la risposta è un `ERR`, e il metodo ritorna. Se c'è, il resto della riga va spezzato nel numero di argomenti che l'azione vuole, ed è il lavoro di `argomenti`, l'altro metodo di `Protocollo`:

```
    /** Spezza il resto della riga in n argomenti, l'ultimo con tutti i suoi spazi;
        null se sono meno di n, se uno e' vuoto, o se ce ne sono quando n e'
    zero. */
    public static String[] argomenti(String resto, int n) {
        resto = resto.strip();
        if (n == 0) return resto.isEmpty() ? new String[0] : null;
        String[] a = resto.split(" ", n);
        if (a.length < n) return null;
        for (int i = 0; i < n; i++) {
            a[i] = a[i].strip();
            if (a[i].isEmpty()) return null;
        }
        return a;
    }

    private static final Pattern NOME = Pattern.compile("[\\p{L}\\p{N}_]{2,16}");
    public static boolean nomeValido(String nome) { return
    NOME.matcher(nome).matches(); }
}
```

Anche qui lo `split` ha il limite, ed è il dettaglio che fa la differenza fra un parser che funziona e uno che funziona "quasi sempre": con `/a Giulia ci vediamo alle 10` il resto è `Giulia ci vediamo alle 10`, e `split(" ", 2)` dà `Giulia` e `ci vediamo alle 10`; senza limite darebbe cinque pezzi e il testo andrebbe perso. Se gli argomenti non tornano (manca il testo, manca il nome, c'è qualcosa dopo `/chi`) il metodo restituisce `null`, e `elabora` risponde `ERR uso:` seguito dall'aiuto del comando: lo stesso testo che l'utente vedrebbe con `/aiuto`, scritto una volta sola nella tabella. Il risultato è che `elabora` *non lancia mai*: ogni riga, anche la più strana, finisce in un'azione o in un `ERR`, e le azioni ricevono sempre il numero giusto di argomenti, mai vuoti. Il controllo del nome (lettere, cifre e `_`, da 2 a 16 caratteri) sta nell'azione `nome`, perché riguarda il valore e non la forma della riga: il nome finirà nelle righe `MSG Marco`

`testo` come secondo campo, e un nome con uno spazio dentro le renderebbe ambigue.



La riga entra, la tabella decide, l'azione risponde con una riga: il protocollo sta in un posto solo.

3.3.3 L'aiuto esce dalla tabella

`/aiuto` risponde con l'elenco dei comandi, un `INFO` per riga, e l'elenco non va scritto: è la tabella stessa, letta nell'ordine in cui è stata riempita.

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

Il vantaggio non è il risparmio di cinque righe: è che l'aiuto non può restare indietro rispetto ai comandi. Chi aggiunge `/ora` alla tabella lo vede comparire in `/aiuto` e negli `ERR uso:` senza fare altro, e chi lo toglie non lascia una riga d'aiuto orfana. Un `INFO` per riga, infine, perché il protocollo non ammette a capo dentro un messaggio, e il server è il primo a rispettarlo.

3.3.4 Un'alternativa: un tipo per messaggio

C'è un secondo modo di organizzare lo stesso lavoro, e vale la pena conoscerlo perché è quello che *Le interfacce in Java moderno* suggerirebbe: un *tipo* per ogni messaggio. Le richieste sono un insieme chiuso, quindi un'interfaccia `sealed` con un record per comando, e una classe che traduce la riga nel record giusto; il server smista i record con uno `switch` sui tipi JAVA 21:

```
public sealed interface Richiesta {
    record Nome(String nome) implements Richiesta {}
    record A(String destinatario, String testo) implements Richiesta {}
    record Tutti(String testo) implements Richiesta {}
    record Errata(String motivo) implements Richiesta {}   // una riga non valida
    // ... Chi, Esci, Aiuto
}

switch (Protocollo.parse(riga)) {
    case Richiesta.A a -> privato(a.destinatario(), a.testo());
    case Richiesta.Errata e -> invia("ERR " + e.motivo());
    // ... un case per record: il compilatore controlla che ci siano tutti
}
```

I due modi si equivalgono in ciò che il client vede, e differiscono in ciò che garantiscono. Con i record il messaggio è un valore: ha `equals`, si stampa, si mette in una coda, si passa

fra thread, e lo `switch` è *esaustivo*: un record in più, e il compilatore indica ogni `switch` da completare. Con la tabella un comando è una riga: aggiungerlo non tocca nessun punto centrale, l'aiuto esce da solo, e i comandi si possono persino caricare da un file o da un plugin; il prezzo è che il compilatore non sa quali comandi esistono, e che un argomento controllato male lo si scopre a runtime. Per un server che vogliamo estendere, con comandi semplici e indipendenti, la tabella è la forma più naturale, ed è quella che la dispensa usa da qui in avanti; il tipo per messaggio torna utile quando i messaggi viaggiano dentro il programma (una coda fra thread, un Model che li conserva) e non solo sul filo.

3.3.5 Provare il protocollo senza rete

Né `Protocollo` né `elabora` importano `java.net`: ricevono stringhe e rispondono con stringhe. Si possono quindi provare con JUnit come qualunque classe, e vale la pena farlo prima di collegare un solo socket, perché gli errori di parsing in rete si manifestano come dialoghi sfasati, difficili da ricondurre alla causa. Per `argomenti` i test sono immediati:

```
class ProtocolloTest {
    @Test void ilRestoTieneGliSpazi() {
        assertArrayEquals(new String[] {"Giulia", "ci vediamo alle 10?"},
                Protocollo.argomenti("Giulia ci vediamo alle 10?", 2));
    }
    @Test void argomentiMancanti() {
        assertNull(Protocollo.argomenti("Giulia", 2));          // manca il testo
        assertNull(Protocollo.argomenti("", 1));
        assertNull(Protocollo.argomenti("di troppo", 0));
    }
    @Test void ogniAiutoIniziaConIlSuoComando() {
        Protocollo.COMANDI.forEach((comando, a) ->
                assertTrue(a.aiuto().startsWith(comando), comando));
    }
}
```

Per provare anche le azioni serve una sessione, e qui paga una scelta fatta nella sezione 3.6: la sessione non conosce il socket, legge da un `Reader` e scrive su un `Writer`. In produzione sono quelli del socket; in un test sono due stringhe, e il dialogo intero si prova in quattro righe:

```
class SessioneTest {
    @Test void privatoAUnoSconosciuto() {
        Reader righe = new StringReader("/nome Marco\n/a Luca ciao\n");
        StringWriter uscita = new StringWriter();
        Sessione s = new Sessione(righe, uscita);
        s.dialoga();
        assertTrue(uscita.toString().contains("ERR utente sconosciuto: Luca"));
    }
    @Test void usoSbagliato() {
        StringWriter uscita = new StringWriter();
        new Sessione(new StringReader("/nome Marco\n/a Giulia\n"),
    uscita).dialoga();
        assertTrue(uscita.toString().contains("ERR uso: /a N testo"));
    }
}
```

Chi scrive un test per ogni riga della tabella, e uno per ogni modo di sbagliarla, scopre subito i casi ambigui (un nome con uno spazio, un testo vuoto, due spazi fra il comando e l'argomento) e li decide a tavolino, prima che un utente li trovi in aula.

3.4 Mai fidarsi di ciò che arriva

Una riga che arriva dalla rete è come una riga scritta dall'utente, con un'aggravante: l'utente è dall'altra parte, può usare un client scritto da lui, e può sbagliare o provare apposta. Il server deve trattare ogni riga con sospetto e rispondere sempre, a tutto, senza mai morire. `elabora` e `argomenti` fanno metà del lavoro: nessuna riga li fa lanciare, e un'azione parte solo con i suoi argomenti al completo. L'altra metà sta nelle azioni: un `/a` verso un utente che non c'è, un messaggio prima del `/nome`, un `/nome` ripetuto o con uno spazio dentro, sono tutti casi che hanno una risposta (`ERR` con il motivo) e nessuna conseguenza.

Attenzione 3.1: L'eccezione che chiude la sessione

```
// 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 nel ciclo di lettura esce dal ciclo, il try-with-resources chiude il socket e il client legge `null` senza nessuna spiegazione; nel server con molti client del capitolo 4 è l'intera sessione di quell'utente a sparire. Ogni accesso a un campo va preceduto dal controllo che il campo esista; ogni conversione va protetta; e la risposta a un errore è un `ERR`, non un'eccezione. Il modo più sicuro per non dimenticarlo è tenere i controlli di *forma* in un posto solo, `elabora` e `argomenti`, così le azioni ricevono sempre argomenti presenti e non vuoti, e lasciare alle azioni i soli controlli di *valore*: il nome valido, il destinatario che esiste, il numero che è un numero.

C'è anche un sospetto nel verso opposto. Il client riceve righe dal server, e un server diverso dal nostro (o il nostro, in una versione futura) può mandarne di inattese. Il client con la finestra avrà la sua tabella, con la prima parola della riga come chiave, e per la parola che non conosce un'azione di riserva che mostra la riga com'era: non è il comportamento migliore possibile, ma è quello che non si blocca.

3.5 Un protocollo che si usa tutti i giorni: HTTP

Prima di tornare a WhatsPAJC, un'esperienza che vale più di molte pagine. HTTP, il protocollo del web, è un protocollo di testo a passo esattamente come quelli di questo capitolo: il client manda una *richiesta* fatta di righe, il server risponde con delle righe e poi con il contenuto. Con `nc` possiamo scriverne una a mano. La prima riga dice il metodo, il percorso e la versione; seguono le *intestazioni*, una per riga, nella forma `Nome: valore`; una riga vuota chiude la richiesta (la cornice del benvenuto dell'esercizio 2.9). HTTP vuole le righe terminate da `\r\n`, e per questo le scriviamo con `printf`:

```
$ 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
Connection: close
Date: Sun, 01 Nov 2026 15:23:53 GMT
Last-Modified: Thu, 29 Oct 2026 13:15:03 GMT

<!doctype html>
<html>
<head>
    <title>Example Domain</title>
...
```

La risposta ha la stessa forma: una riga di stato (versione, codice, descrizione), le intestazioni, una riga vuota, e il contenuto, lungo quanto dice `Content-Length`. È tutto qui: ciò che un browser fa centinaia di volte per pagina sono righe di testo in un socket, con un accordo su quali righe. Lo stesso dialogo si scrive in Java con dieci righe sopra un `Socket` (e lo faremo in un esercizio); nel capitolo 6 useremo la classe della libreria che lo sa fare bene, `HttpClient`, e scriveremo un piccolo server che risponde al browser. Qui il punto è un altro: avendo capito il protocollo della chat, si è capito anche questo.

3.6 L'esempio guida: WhatsPAJC v1

Mettiamo insieme i pezzi. La prima versione del server di WhatsPAJC serve un client alla volta, come i server del capitolo 2, ma capisce *tutti* i comandi del protocollo e risponde a tutti secondo le regole: con un solo client l'elenco degli utenti ha un nome solo e i messaggi a tutti tornano al mittente, ma il dialogo è già quello definitivo, e nel capitolo 4 questo codice diventerà, quasi intatto, la sessione di ogni client. Il cuore è la classe `Sessione`: tutto ciò che riguarda *un* client (i suoi stream, il suo nome, il suo dialogo) e niente altro. Non sa nemmeno di socket: riceve un `Reader` e un `Writer`, e chi la costruisce decide da dove vengono.

```
public class Sessione {
    private final BufferedReader in;
    private final PrintWriter out;
    private String nome;                        // null finche' non arriva /nome
    private boolean attiva = true;              // false dopo /esci

    public Sessione(Reader in, Writer out) {
        this.in = new BufferedReader(in);
        this.out = new PrintWriter(out, true);
    }

    void invia(String riga) { out.println(riga); }

    void dialoga() throws IOException {
        invia("INFO Benvenuto su WhatsPAJC. Scegli un nome con /nome");
        String riga;
        while (attiva && (riga = in.readLine()) != null)
            elabora(riga);                      // la tabella decide
    }
```

Il ciclo legge una riga e la passa a `elabora`, che conosciamo; `attiva` diventa `false` con `/esci`, e il metodo ritorna. Le azioni sono i metodi che la tabella chiama, e con un client solo sono brevi: chi c'è è un nome solo, un messaggio a tutti torna al mittente, un privato ha senso solo verso se stessi. La regola di turno del protocollo ("`/nome` prima di tutto") sta in `senzaNome`, che le azioni che parlano chiamano per prime:

```
void nome(String n) {
        if (nome != null) { invia("ERR hai gia' un nome: " + nome); return; }
        if (!Protocollo.nomeValido(n)) { invia("ERR nome non valido: " + n);
    return; }
        nome = n;
        invia("BENVENUTO " + nome);
    }
    void chi() {
        if (senzaNome()) return;
        invia("UTENTI " + nome);
    }
    void tutti(String testo) {
        if (senzaNome()) return;
        invia("MSG " + nome + " " + testo);
    }
    void privato(String a, String testo) {
        if (senzaNome()) return;
        if (!a.equals(nome)) { invia("ERR utente sconosciuto: " + a); return; }
        invia("PRIV " + nome + " " + a + " " + testo);
    }
    void esci()  { invia("CIAO"); attiva = false; }
    // aiuto(): la tabella, come nella sezione precedente

    private boolean senzaNome() {
        if (nome != null) return false;
        invia("ERR prima scegli un nome con /nome");
        return true;
    }
}
```

`/aiuto` ed `/esci` passano anche senza nome: chiedere aiuto prima di presentarsi è ragionevole, e andarsene pure. Il `main` è quello dei server a un client alla volta, e costruisce la sessione sugli stream del socket: è l'unico punto in cui socket e sessione si incontrano, ed è il motivo per cui la sessione si prova con due stringhe.

```
public class ServerChatV1 {
    public static void main(String[] args) throws IOException {
        try (ServerSocket server = new ServerSocket(1234)) {
            System.out.println("WhatsPAJC v1 in ascolto sulla porta 1234");
            while (true) {
                try (Socket socket = server.accept()) {
                    new Sessione(new InputStreamReader(socket.getInputStream(),
                                                       StandardCharsets.UTF_8),
                                 new OutputStreamWriter(socket.getOutputStream(),
                                                        StandardCharsets.UTF_8))
                            .dialoga();
                } catch (IOException e) {
                    System.err.println("client perso: " + e.getMessage());
                }
            }
        }
    }
}
```

Proviamolo con `nc` e con `ClientConsole`, scrivendo il dialogo della sezione 3.2: le risposte sono quelle della tabella, un comando sbagliato produce un `ERR` e il dialogo continua, `/esci` produce `CIAO` e la connessione si chiude. Il client di console mostra le righe grezze del protocollo, `MSG Marco buongiorno` e non "Marco: buongiorno": è voluto, perché in questo capitolo vogliamo vedere il protocollo; tradurlo per l'utente è il lavoro del client con la finestra, con la stessa idea rovesciata: una tabella con la prima parola della riga come chiave.

Ricetta: Progettare un protocollo di testo

1. scegliere la cornice: una riga per messaggio, UTF-8, niente a capo nel testo;
2. scrivere le due tabelle, richieste e risposte, con il primo campo che decide il tipo e il testo libero sempre in fondo;
3. decidere il turno: a passo se il client chiede e il server risponde, libero se il server deve poter parlare da solo (e allora il client ha un thread lettore);
4. scrivere un dialogo completo a mano, come lo si farebbe con `nc`, errori compresi: ogni riga deve avere una risposta, anche se è `ERR`;
5. una tabella da comando ad azione (argomenti, aiuto, che cosa fa) in una classe `Protocollo`, con un metodo che spezza il resto della riga negli argomenti e non lancia mai; la sessione che legge da un `Reader` e scrive su un `Writer`; i test della tabella e del dialogo prima di aprire un socket.

Esercizi

Esercizio 3.1: Prevedere il dialogo (⋆)

Marco è collegato a WhatsPAJC v1 e manda, nell'ordine: `ciao`, `/nome Marco`, `/nome Marco2`, `/a Marco mi parlo da solo`, `/a Giulia ciao`, `/chi`. Scrivere le righe che il server manda in risposta a ciascuna.

Esercizio 3.2: Il limite dello `split` (⋆)

Dire che cosa restituisce `"/a Giulia ci vediamo alle 10".split(" ")` e che cosa restituisce con il limite 3, e che cosa riceverebbe Giulia se il server usasse la prima forma prendendo `campi[2]` come testo. Che cosa succede, con il limite 3, se fra `/a` e `Giulia` ci sono due spazi?

Esercizio 3.3: Quale azione (⋆)

Per ciascuna riga dire che cosa fa `elabora`: quale azione chiama e con quali argomenti, oppure quale `ERR` risponde: (a) `/A Giulia Ciao`; (b) `/chi ` ; (c) `/nome Marco Rossi`; (d) `/nome`; (e) `ciao /a Giulia`; (f) una riga vuota.

Esercizio 3.4: Il client a passo (⋆⋆)

Si collega a WhatsPAJC v1 il `ClientEco` del capitolo 2, che legge una riga di risposta per ogni riga mandata. L'utente scrive `/nome Marco`, poi `/aiuto`, poi `ciao`. Dire che cosa compare sul suo video dopo ciascuna riga e in quale stato restano i due programmi.

Esercizio 3.5: Che cosa fa `nc` (⋆⋆)

Un utente si collega a WhatsPAJC v1 con `nc` e, invece di `/nome`, incolla un testo di tre righe copiato da un documento. Dire che cosa manda `nc`, quante richieste vede il server, e quante risposte riceve l'utente; poi dire quale regola del protocollo è in gioco e dove va fatta rispettare, nel server o nel client.

I problemi che seguono chiedono un programma completo, in ordine crescente di difficoltà. Le soluzioni sono nel repository delle soluzioni del corso, nel progetto e nel package indicati.

Esercizio 3.6: La calcolatrice con il protocollo (⋆)

Riscrivere la calcolatrice remota dell'esercizio 2.8 con il metodo di questo capitolo: una tabella delle operazioni (un record `Azione` con il numero di operandi, l'aiuto e che cosa fa), un metodo che spezza la riga negli operandi e li converte in numeri senza lanciare mai, una sessione su `Reader` e `Writer`, `/aiuto` che esce dalla tabella, e una classe di test JUnit con almeno un test per ogni riga della tabella e uno per ogni modo di sbagliarla (operando mancante, non numerico, divisione per zero).

Soluzione: progetto `sol-d06-c03`, package `it.unibs.pajc.redolfi.e06`.

Esercizio 3.7: Il quiz (⋆⋆)

Scrivere un server che, dopo il `/nome`, fa al client una domanda alla volta (`DOMANDA n testo`), legge la risposta (`/r testo`), dice se è giusta (`OK`, oppure `NO risposta giusta`) e tiene il punteggio; `/classifica` mostra il punteggio corrente, `/esci` chiude con il totale. Le domande si leggono da un file di testo, una per riga, nella forma `domanda|risposta`. Progettare il protocollo con le due tabelle e scrivere prima `Protocollo` con i suoi test.

Soluzione: progetto `sol-d06-c03`, package `it.unibs.pajc.redolfi.e07`.

Attenzione 3.2: Il server che fa la domanda seguente senza aspettare

Il quiz è un protocollo a passo guidato dal *server*: dopo una domanda il server deve aspettare `/r`, e un `/classifica` nel frattempo è lecito e non è la risposta. Chi scrive il ciclo come "manda domanda, leggi riga, confronta" tratta `/classifica` come una risposta sbagliata: lo stato "in attesa della risposta alla domanda `n`" va tenuto in un campo e consultato nello `switch`.

Esercizio 3.8: Il registro dei messaggi (⋆⋆)

Estendere WhatsPAJC v1 perché il server scriva ogni messaggio a tutti in un file di testo, in append, nella forma `ora nome testo` (una riga ciascuno), e perché il comando nuovo `/storia n` rimandi al client le ultime `n` righe del file come `INFO`. Il protocollo va esteso in entrambe le direzioni (un record in più, i test aggiornati), e `n` non valido deve produrre un `ERR`.

Soluzione: progetto `sol-d06-c03`, package `it.unibs.pajc.redolfi.e08`.

Esercizio 3.9: Un mini-HTTP (⋆⋆⋆)

Scrivere un server che risponde a un browser. Accetta una connessione, legge la riga di richiesta e le intestazioni fino alla riga vuota, e se il percorso è `/ora` risponde con una pagina HTML che mostra l'ora; per qualunque altro percorso risponde `404 Not Found` con una pagina di errore. Provarlo aprendo `http://localhost:1234/ora` nel browser. Scrivere poi il client Java che fa la richiesta a mano su un `Socket` e stampa la riga di stato e il corpo.

Suggerimento: le righe della risposta terminano con `\r\n`; dopo le intestazioni serve una riga vuota; `Content-Length` è il numero di byte *del corpo in UTF-8, non di caratteri; `Connection: close` e poi chiudere è il modo più semplice per dire al browser che il corpo è finito.*

Soluzione: progetto `sol-d06-c03`, package `it.unibs.pajc.redolfi.e09`.

Risposte

Risposta 3.1. `ciao` prima del nome: `ERR prima scegli un nome con /nome`. `/nome Marco`: `BENVENUTO Marco`. `/nome Marco2`: `ERR hai gia' un nome`. `/a Marco mi parlo da solo`: `PRIV Marco Marco mi parlo da solo` (con un client solo, il destinatario esiste se è il mittente). `/a Giulia ciao`: `ERR utente sconosciuto: Giulia`. `/chi`: `UTENTI Marco`. Prima di tutto, alla connessione, il server aveva mandato l'`INFO` di benvenuto.

Risposta 3.2. Senza limite: sei campi, `/a`, `Giulia`, `ci`, `vediamo`, `alle`, `10`; con il limite 3: tre campi, e il terzo è `ci vediamo alle 10`. Con la prima forma Giulia riceverebbe `PRIV Marco Giulia ci`: il resto del messaggio sarebbe perso senza nessun errore. Con due spazi fra `/a` e `Giulia`, `split(" ", 3)` produce `/a`, una stringa vuota e `Giulia ci vediamo alle 10`: il destinatario sarebbe vuoto. È il motivo per cui `elabora` spezza prima in due (comando e resto) e `argomenti` fa `strip` del resto e di ogni campo: i due spazi spariscono, e un campo che resta vuoto dà `null`, cioè `ERR uso`. L'alternativa è `split("\\s+", n)`, che tratta più spazi come uno.

Risposta 3.3. (a) `privato("Giulia", "Ciao")`: il comando è confrontato in minuscolo, il testo no. (b) `chi()`: lo `strip` toglie gli spazi ai bordi. (c) `nome("Marco Rossi")`: con un argomento solo, `argomenti` restituisce tutto il resto, spazio compreso, e l'azione risponde `ERR nome non valido: Marco Rossi`, perché lo spazio non passa l'espressione regolare. (d) `argomenti("", 1)` è `null`: `ERR uso: /nome N il nome, per primo`. (e) `tutti("ciao /a Giulia")`: la barra conta solo all'inizio. (f) niente: la riga vuota viene ignorata, senza risposta, perché un Invio a vuoto non è un comando.

Risposta 3.4. Alla connessione il server manda l'`INFO` di benvenuto, che il client non legge subito. Dopo `/nome Marco` il client legge una riga: l'`INFO`; il `BENVENUTO` resta nel buffer. Dopo `/aiuto` legge il `BENVENUTO`; le sei righe `INFO` dell'aiuto restano nel buffer. Dopo `ciao` legge la prima riga dell'aiuto; nel buffer restano cinque `INFO` e il `MSG Marco ciao`. Nessuno dei due programmi è bloccato: il server aspetta la riga seguente, il client aspetta la tastiera; ma il client è indietro di sette righe, e lo sarà sempre di più.

Risposta 3.5. `nc` manda tre righe, perché il testo incollato contiene due a capo, e il server legge tre richieste: tre messaggi per tutti (una riga vuota verrebbe ignorata), e a ciascuna risponde `ERR prima scegli un nome con /nome`: tre risposte. La regola è che un messaggio è una riga: il server non può fare altro che leggere tre righe, e infatti non si rompe. È il client che deve impedire che un testo con degli a capo parta come più righe, sostituendoli con spazi o rifiutandolo: con `nc` non c'è nessun client, e l'utente ottiene esattamente ciò che ha mandato.

Capitolo 4

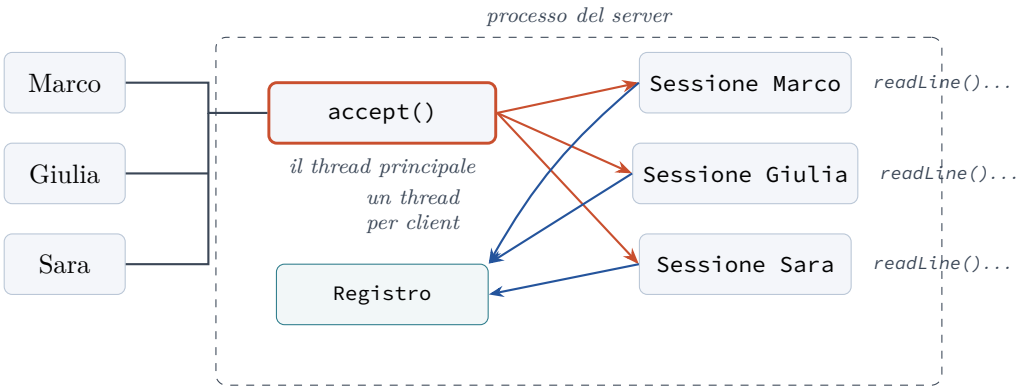
Un server per molti

Il server del capitolo 3 capisce tutto il protocollo, ma serve un client alla volta: finché Marco è collegato, Giulia bussa e nessuno risponde. È la terza domanda della dispensa vista dal lato del server: come si servono molti client insieme, senza che nessuno aspetti gli altri. La risposta è un thread per client, e con i thread arrivano le loro conseguenze, che conosciamo da *Thread e concorrenza*: uno stato condiviso da proteggere, scritture che si mescolano, thread che restano appesi, un programma da fermare in modo pulito. Alla fine del capitolo il server di WhatsPAJC è completo: molti client, messaggi a tutti e a uno, nomi univoci, ingressi e uscite annunciati, un arresto ordinato, e il codice del capitolo 3 ci sta dentro quasi intatto.

4.1 Un client alla volta

Vediamo che cosa succede davvero quando Giulia si collega mentre Marco sta parlando con il server a un client. Il thread del server è fermo in `in.readLine()` sul socket di Marco, e non chiama `accept`. Il sistema operativo però accetta lo stesso la connessione di Giulia: la completa, la mette in una coda (il *backlog* del `ServerSocket`, lunga 50 per default) e aspetta che il programma la ritiri. Dal lato di Giulia `new Socket` ritorna senza errori, le righe che scrive partono e si fermano nel buffer di ricezione del server, e nessuno le legge: silenzio. Quando Marco scrive `/esci`, il server torna ad `accept`, ritira la connessione di Giulia e legge di colpo tutto ciò che lei ha scritto nel frattempo, come se fosse appena arrivato. Non è un errore di nessuno, è il comportamento normale di TCP, e per questo è così facile non accorgersene: il server "funziona", solo con una persona alla volta.

Il ciclo del server fa due cose incompatibili con un thread solo: aspettare un *nuovo* client in `accept`, e aspettare la *prossima riga* di ciascun client già collegato in `readLine`. Sono tutte attese, e una sola può essere in corso. La soluzione è separare i ruoli: un thread fa solo `accept`, e per ogni client accettato consegna la connessione a un thread nuovo, che da quel momento si occupa solo di lui. È l'immagine guida con molti fili: ogni filo entra nel server e finisce in un thread, che passa quasi tutto il suo tempo fermo ad aspettare una riga.



Un thread ritira le connessioni; ogni connessione vive in una sessione su un thread suo; tutte condividono il registro.

Nella figura c'è già il secondo problema del capitolo. Le sessioni non sono indipendenti: per mandare un messaggio a tutti, la sessione di Marco deve scrivere sui socket di Giulia e di Sara, e per sapere chi c'è deve consultare un elenco che le altre sessioni aggiornano. Quell'elenco, il *registro*, è uno stato condiviso fra thread, con tutto quello che comporta. Prima però la sessione.

4.2 La sessione e il ciclo di `accept`

La `Sessione` del capitolo 3 è già tutto ciò che riguarda *un* client: i suoi stream, il suo nome, il suo dialogo, le azioni della tabella. Per servirne molti cambia in tre punti: diventa un `Runnable`, così che un thread possa eseguirla; riceve il socket, e lo apre in `run` (così il costruttore non lancia e il ciclo di `accept` resta pulito); riceve il `Registro`, lo stato condiviso con le altre sessioni. Il costruttore su `Reader` e `Writer` resta, per i test.

```
public class Sessione implements Runnable {
    private final Socket socket;               // null senza rete
    private final Registro registro;           // lo stato condiviso
    private BufferedReader in;
    private PrintWriter out;
    private String nome;                       // null finche' non arriva /nome
    private boolean attiva = true;

    public Sessione(Socket socket, Registro registro) {
        this.socket = socket;
        this.registro = registro;
    }

    Sessione(Reader in, Writer out, Registro registro) {   // senza rete: per i
    test
        this(null, registro);
        this.in = new BufferedReader(in);
        this.out = new PrintWriter(out, true);
    }

    public synchronized void invia(String riga) {
        out.println(riga);
    }

    void chiudi() {
        try {
            if (socket != null) socket.close();
        } catch (IOException e) { /* gia' chiuso */ }
    }
```

Il `synchronized` su `invia` è la prima conseguenza dei molti thread, e la spieghiamo nella sezione 4.3; il `Registro` ricevuto nel costruttore è il secondo. Il metodo `run` apre gli stream e incarta il dialogo in tre strati di protezione:

```
@Override
    public void run() {
        log("collegato da " + socket.getRemoteSocketAddress());
        try {
            socket.setSoTimeout(30 * 60 * 1000);          // mezz'ora di silenzio
            in = new BufferedReader(new InputStreamReader(
                    socket.getInputStream(), StandardCharsets.UTF_8));
            out = new PrintWriter(socket.getOutputStream(), true,
    StandardCharsets.UTF_8);
            dialoga();
        } catch (SocketTimeoutException e) {
            invia("ERR inattivo da troppo tempo");
        } catch (IOException e) {
            log("connessione persa: " + e.getMessage());
        } finally {
            if (nome != null) {
                registro.rimuovi(nome);
                registro.aTutti("ESCE " + nome);
            }
            chiudi();
            log("chiuso");
        }
    }
```

`dialoga()` ed `elabora` sono quelli del capitolo 3, senza una modifica: a cambiare sono tre azioni, nella sezione 4.4. Il `finally` è il punto più importante del metodo, perché viene eseguito in *tutti* i modi in cui una sessione può finire: `/esci`, il client che chiude, il client che sparisce (la `IOException`), il silenzio troppo lungo (il timeout), e perfino un errore nostro nel dialogo. In ogni caso il nome viene tolto dal registro, gli altri vengono avvisati e il socket viene chiuso. Senza il `finally`, un client che stacca il cavo resterebbe nell'elenco di `/chi` per sempre: l'*utente fantasma*, che ogni server di chat scritto in fretta ha conosciuto.

4.2.1 Un thread per sessione

Il ciclo di `accept` non cambia quasi niente rispetto al capitolo precedente: ritira una connessione, costruisce la sessione e, invece di eseguirla, la consegna a un executor.

```
public class ServerChat {
    private final ServerSocket server;
    private final Registro registro = new Registro();
    private final ExecutorService exec =
    Executors.newVirtualThreadPerTaskExecutor();

    public ServerChat(int porta) throws IOException {
        server = new ServerSocket(porta);
    }
```

Il `ServerSocket` e il registro vivono quanto il server; l'executor è quello che decide su quale thread gira ogni sessione, e ne parliamo fra un momento. Il ciclo:

```
public void avvia() {
        System.out.println("WhatsPAJC in ascolto sulla porta " +
    server.getLocalPort());
        try {
            while (true) {
                Socket socket = server.accept();
                exec.submit(new Sessione(socket, registro));
            }
        } catch (IOException e) {
            System.out.println("accept interrotto: " + e.getMessage());
        }
        exec.close();                                    // aspetta le sessioni
        System.out.println("server fermo");
    }
```

La scelta dell'executor merita una pagina, perché è il posto in cui la rete e i thread si incontrano meglio. Una sessione è un compito che dura *quanto la connessione*: minuti o ore, passati per il 99% del tempo fermi in `readLine`, con un lampo di lavoro a ogni riga. È l'opposto dei compiti di calcolo per cui si usa un pool fisso, ed è esattamente il caso per cui Java 21 ha introdotto i *thread virtuali* (*Thread e concorrenza*, capitolo sul tempo e l'attesa): thread che la JVM gestisce da sé, così leggeri che se ne possono avere migliaia, e che quando si bloccano in una lettura non occupano nessun thread del sistema operativo. `newVirtualThreadPerTaskExecutor()` crea un thread virtuale per compito e lo butta alla fine: mille client sono mille thread virtuali, e il server se ne accorge appena.

Attenzione 4.1: Il pool fisso che fa tacere il quinto client

```
ExecutorService exec = Executors.newFixedThreadPool(4);   // SBAGLIATO per
    le sessioni
```

Con un pool di quattro thread, il quinto client viene accettato, la sua sessione viene accodata, e resta in coda finché uno dei primi quattro non se ne va: il client riceve la connessione, scrive, e tace tutto, esattamente come con il server a un client alla volta. Un pool fisso limita la *concorrenza*, e qui ogni compito tiene un thread per tutta la vita del client. Prima dei thread virtuali la scelta era `newCachedThreadPool()`, che crea thread quando servono e li riusa: funziona ancora, con il costo di un thread del sistema operativo per client, e un tetto di qualche migliaio.

C'è un dettaglio dei thread virtuali, in Java 21, che riguarda proprio il nostro codice. Un thread virtuale bloccato *dentro* un blocco `synchronized` resta inchiodato al thread di piattaforma che lo esegue (*pinning*): finché non esce, quel thread non serve nessun altro. Il `synchronized` di `invia` è innocuo, perché dentro c'è una `println` che non blocca (scrive in un buffer); sarebbe un problema un `readLine` dentro un `synchronized`, che infatti non c'è. La regola: sezioni sincronizzate corte, e mai un'attesa dentro.

4.3 Il registro: stato condiviso

Le sessioni devono sapere l'una dell'altra: per `/chi`, per i messaggi a tutti, per trovare il destinatario di un `/a`, per rifiutare un nome già preso. Serve un oggetto che tenga la mappa da nome a sessione, e che venga letto e modificato da tutti i thread delle sessioni insieme. È stato condiviso, e una `HashMap` toccata da molti thread si rompe in silenzio (*Thread e concorrenza*, capitolo sullo stato condiviso): due `put` nello stesso istante possono perdere una voce, un'iterazione durante un `remove` lancia `ConcurrentModificationException`. Lo strumento giusto è `ConcurrentHashMap`, e il registro è piccolo:

```
public class Registro {
    private final ConcurrentHashMap<String, Sessione> sessioni = new
    ConcurrentHashMap<>();

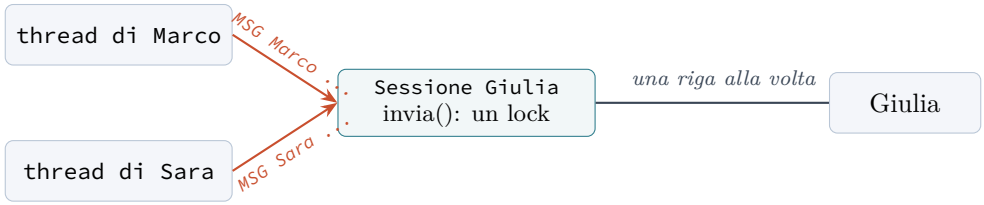
    public boolean aggiungi(String nome, Sessione s) {
        return sessioni.putIfAbsent(nome, s) == null;     // true se il nome era
    libero
    }
    public void rimuovi(String nome) { sessioni.remove(nome); }
    public Sessione trova(String nome) { return sessioni.get(nome); }
    public List<String> nomi() { return
    sessioni.keySet().stream().sorted().toList(); }

    public void aTutti(String riga) {
        for (Sessione s : sessioni.values()) s.invia(riga);
    }
    public void aAltri(String riga, Sessione tranne) {
        for (Sessione s : sessioni.values())
            if (s != tranne) s.invia(riga);
    }
}
```

Tre osservazioni. `putIfAbsent` è un'operazione sola: "se il nome non c'è, mettilo, e dimmi com'è andata". Scritta come `if (!containsKey(nome)) put(nome, s)` sarebbero due operazioni, e fra le due un altro thread potrebbe infilare lo stesso nome: Marco e Marco, entrambi convinti di essere i soli. È il *check-then-act* della dispensa sui thread, e la mappa concorrente ha i metodi atomici apposta. Seconda: il ciclo di `aTutti` scorre la mappa mentre altri thread possono aggiungere o togliere voci, e con `ConcurrentHashMap` è lecito, perché i suoi iteratori sono *debolmente consistenti*: non lanciano, e mostrano la mappa com'era più o meno in quel momento. Con una mappa normale protetta da un lock, la forma giusta sarebbe copiare i valori sotto il lock e scorrere la copia fuori. Terza: `nomi()` restituisce una lista nuova, non la vista delle chiavi, così chi la riceve può usarla con calma.

4.3.1 Due thread, un tubo

Resta la scrittura. `aTutti` viene eseguito dal thread della sessione di Marco, e scrive sul socket di Giulia: il `PrintWriter` di Giulia viene usato dal thread di Marco. Nello stesso istante il thread di Sara, che sta mandando un altro messaggio a tutti, scrive sullo stesso `PrintWriter`. Due `println` dello stesso `PrintWriter` da due thread non sono atomiche rispetto all'altra: i caratteri delle due righe possono mescolarsi, e Giulia riceve `MSG Marco buonMSG Sara ciao...`.



Sul socket di un client scrivono i thread di tutti gli altri: `invia` li mette in fila.

Il `synchronized` su `invia` serve a questo: il lock è quello della sessione di *Giulia*, e chiunque voglia scrivere sul suo socket lo prende, scrive una riga intera, lo lascia. Le righe arrivano integre, in un ordine qualunque ma una alla volta. È il confinamento della scrittura in un solo metodo, e non per caso `out` è privato: se un altro pezzo di codice scrivesse sul `PrintWriter` senza passare da `invia`, la garanzia sparirebbe.

Osservazione 4.1

In una versione precedente di questo server il registro era un campo `static`: una mappa di classe, raggiungibile da tutte le sessioni senza passarla. Funzionava, ma è una scelta da evitare, per due motivi che la dispensa *Pattern in gioco* discute a proposito dei singleton. Un oggetto statico è un globale nascosto: ogni sessione dipende da lui senza dichiararlo, e non si può provare una `Sessione` con un registro finto. E non si possono avviare due server nella stessa JVM (per esempio nei test, o per due chat su due porte): condividerebbero gli utenti. Il `Registro` passato nel costruttore costa una riga e non ha nessuno di questi difetti.

4.4 Entrare e uscire

La tabella ed `elabora` non cambiano: cambiano tre azioni, quelle in cui le sessioni si parlano. Il nome ora va registrato, e può essere rifiutato perché un altro lo ha già; chi entra viene annunciato agli altri:

```
void nome(String n) {
    if (nome != null) { invia("ERR hai gia' un nome: " + nome); return; }
    if (!Protocollo.nomeValido(n)) { invia("ERR nome non valido: " + n); return; }
    if (!registro.aggiungi(n, this)) { invia("ERR nome gia' in uso: " + n);
    return; }
    nome = n;
    invia("BENVENUTO " + nome);
    registro.aAltri("ENTRA " + nome, this);
}
void chi() {
    if (senzaNome()) return;
    invia("UTENTI " + String.join(",", registro.nomi()));
}
void tutti(String testo) {
    if (senzaNome()) return;
    registro.aTutti("MSG " + nome + " " + testo);
}
```

L'ordine delle due righe dopo `aggiungi` conta: prima il `BENVENUTO` a chi entra, poi l'`ENTRA` agli altri; e l'`ENTRA` non va a chi è appena entrato, che ha già avuto il suo benvenuto. Il

messaggio a tutti è una riga: è il registro a scorrere le sessioni, mittente compreso, come il protocollo prescrive. Il privato cerca il destinatario nel registro e, se c'è, manda la stessa riga a lui e al mittente:

```
void privato(String a, String testo) {
    if (senzaNome()) return;
    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 potrebbe andarsene: `dest` resta un oggetto valido, `invia` scrive su un socket chiuso, e il `PrintWriter`, come sappiamo, non si lamenta. Il messaggio si perde, e il mittente non lo sa. È una *finestra* inevitabile in un sistema distribuito: fra il momento in cui si controlla e quello in cui si agisce, il mondo può cambiare. Un protocollo più ambizioso risponderebbe con una conferma di consegna; il nostro accetta la finestra, e vale la pena sapere che c'è.

4.4.1 Chi se ne va

Una sessione finisce in quattro modi, e li vediamo sulla linea del tempo. Con `/esci` il server risponde `CIAO` e `dialoga` ritorna. Se il client chiude il socket (o termina normalmente), `readLine` restituisce `null` e il ciclo esce. Se il client sparisce senza chiudere, `readLine` lancia `SocketException` (*Connection reset*) appena il sistema remoto lo comunica, oppure, se il cavo è staccato e nessuno comunica niente, resta in attesa finché `setSoTimeout` non scade: è per questo che il timeout c'è, anche se è lungo. In tutti i casi si passa dal `finally`: il nome esce dal registro, gli altri ricevono `ESCE`, il socket viene chiuso.



L'uscita con `/esci`; per `null`, *reset* e timeout il `finally` è lo stesso.

Il `log` di `Sessione` stampa sulla console del server chi si collega, come si chiama e quando se ne va, con il motivo. In un server è l'unico modo per sapere che cosa è successo, e la riga `connessione persa: Connection reset` è informazione, non un errore da correggere.

4.5 Fermare un server

Il ciclo `while (true)` di `avvia` non è una dimenticanza: un server vive finché qualcuno non lo ferma, e va deciso *come*. Terminare il processo (Ctrl+C) funziona, e il sistema operativo chiude tutti i socket; ma i client ricevono un *reset* invece di un saluto, e un server che scrive su file o su un database potrebbe lasciare qualcosa a metà. L'arresto ordinato ha tre passi: smettere di accettare, salutare chi c'è, chiudere.

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

Il primo passo è il meno intuitivo. Il thread di `avvia` è fermo in `accept`, e non c'è modo di dirgli "basta" con una variabile, perché non la guarda; `interrupt()` non lo sveglia, perché l'attesa è nel sistema operativo. Il modo giusto è chiudere il `ServerSocket` da un altro thread: `accept` lancia `SocketException` (*Socket closed*), il `catch` di `avvia` la raccoglie ed esce dal ciclo. Non è un errore, è il segnale di arresto, e il `catch` lo tratta così. Lo stesso vale per le sessioni: chiudere il socket di un client fa lanciare il suo `readLine`, e la sessione passa dal `finally`. Infine `exec.close()` aspetta che tutte le sessioni abbiano finito, e il programma termina.

Chi chiama `ferma`? Nel nostro server, un thread che legge la console: la parola `stop` ferma tutto. È un thread *daemon*, così non tiene in vita la JVM quando il server ha finito.

```
public static void main(String[] args) throws IOException {
        ServerChat chat = new ServerChat(1234);
        Thread console = new Thread(() -> {
            try (BufferedReader tastiera = new BufferedReader(
                    new InputStreamReader(System.in))) {
                String riga;
                while ((riga = tastiera.readLine()) != null)
                    if (riga.equals("stop")) { chat.ferma(); return; }
            } catch (IOException e) { /* fine */ }
        });
        console.setDaemon(true);
        console.start();
        chat.avvia();
    }
}
```

4.5.1 Quanti client regge

Resta la domanda che ogni server si sente fare. La risposta onesta è una misura, e la misura è facile: un programma che apre n connessioni verso il server, sceglie un nome per ciascuna e manda un messaggio a tutti, poi conta sulla prima connessione quanti `MSG` arrivano e in quanto tempo. Sul portatile di chi scrive, con i thread virtuali, 300 client ricevono i loro 300 messaggi in circa un decimo di secondo, e 1000 client in meno di mezzo secondo; i numeri cambiano da macchina a macchina, ma l'ordine di grandezza è questo, e la chat di un'aula è tre ordini di grandezza sotto. Con un pool fisso di quattro thread il programma si ferma al quinto client, come promesso; con un `newCachedThreadPool` i tempi sono simili fino a qualche centinaio di client e poi crescono, perché ogni client è un thread del sistema operativo. Il limite vero, per un server come il nostro, non sono i thread: è il giro di `aTutti`, che costa n scritture per ogni messaggio e quindi n^2 per n messaggi; oltre qualche migliaio di utenti si passa alle stanze (un esercizio di questo capitolo) o a un'architettura diversa (capitolo 8).

4.6 L'esempio guida: WhatsPAJC v2

Il server è completo: `ServerChat` con il ciclo di `accept` e l'arresto, `Sessione` con il dialogo e le azioni, `Registro`, e la tabella di `Protocollo` del capitolo 3 senza una modifica. Proviamolo con tre `nc` in tre terminali. Marco entra per primo, poi Giulia; Sara prova a chiamarsi Marco; Marco scrive a tutti e a Giulia, chiede chi c'è ed esce; Giulia scrive a Luca, che non esiste. Dal terminale di Giulia:

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

Dal terminale di Sara, nel frattempo, il nome rifiutato e poi accettato:

```
INFO Benvenuto su WhatsPAJC. Scegli un nome con /nome
/nome Marco
ERR nome gia' in uso: Marco
/nome Sara
BENVENUTO Sara
MSG Marco buongiorno a tutti
MSG Giulia ciao Marco
ESCE Marco
```

Sara non vede il `PRIV` fra Marco e Giulia, Giulia vede `ENTRA Sara` ma non il proprio ingresso, tutti vedono i messaggi a tutti nello stesso ordine: è il protocollo del capitolo 3, fatto

rispettare da un server che ora ha davvero qualcuno a cui parlare. Sulla console del server scorre il log, e `stop` saluta tutti e chiude.

Ricetta: Da un server a un client a un server per molti

1. spostare tutto ciò che riguarda un client (socket, stream, stato del dialogo, ciclo di lettura) in una classe `Runnable`, la sessione, con il `finally` che pulisce;
2. il ciclo di `accept` consegna ogni sessione a un executor: thread virtuali (`newVirtualThreadPerTaskExecutor`), mai un pool fisso;
3. ciò che le sessioni condividono va in un oggetto a parte, passato nel costruttore, con collezioni concorrenti e operazioni atomiche (`putIfAbsent`);
4. la scrittura sul socket di una sessione passa da un solo metodo sincronizzato;
5. l'arresto chiude il `ServerSocket` e i socket dei client da un altro thread, e `close` dell'executor aspetta la fine.

Esercizi

Esercizio 4.1: L'instradamento (⋆)

Un server di chat di qualche anno fa aveva questo ciclo (semplificato):

```
while ((riga = in.readLine()) != null) {
    String dest = riga.startsWith("@") ? riga.substring(1,
    riga.indexOf(':')) : null;
    String testo = riga.substring(riga.indexOf(':') + 1);
    if (dest != null && !dest.equals("all"))
        registro.trova(dest).invia(testo);
    registro.aTutti(testo);
}
```

Dire che cosa riceve Giulia e che cosa ricevono gli altri quando Marco scrive `@Giulia:ciao`, e che cosa succede se Marco scrive `@Luca:ciao` e Luca non esiste. Riscrivere il ciclo corretto in tre righe.

Esercizio 4.2: Il quinto client (⋆)

Il server usa `Executors.newFixedThreadPool(4)`. Quattro client sono collegati e un quinto si collega e scrive `/nome Luca`. Dire che cosa vede Luca, che cosa vede il server nel log, e che cosa succede quando uno dei primi quattro scrive `/esci`. Che cosa cambia con `newCachedThreadPool()` e con `newVirtualThreadPerTaskExecutor()`?

Esercizio 4.3: Righe mescolate (⋆)

Si toglie `synchronized` da `invia`. Descrivere una sequenza di eventi in cui Giulia riceve una riga che non è né di Marco né di Sara, e dire perché il problema non si manifesta quasi mai con tre utenti che scrivono a mano.

Esercizio 4.4: Il cavo staccato (⋆⋆)

Sara è collegata dal portatile e chiude il coperchio: la macchina dorme e non manda nulla. Dire che cosa vedono Marco e Giulia in `/chi` un minuto dopo, che cosa succede quando Marco scrive a tutti, e dopo quanto tempo, e per quale via, Sara esce dal registro. Che cosa cambierebbe senza `setSoTimeout`?

Esercizio 4.5: `accept` dopo `close` (⋆⋆)

In `ferma` si scambiano le prime due righe: prima `aTutti(Ciao)`, poi `server.close()`. Dire che cosa cambia. Poi si toglie `server.close()` del tutto, e si sostituisce con una variabile `volatile boolean fermo` controllata nel `while` di `avvia`: dire perché il server non si ferma, e quando si fermerebbe.

I problemi che seguono chiedono un programma completo, in ordine crescente di difficoltà. Le soluzioni sono nel repository delle soluzioni del corso, nel progetto e nel package indicati.

Esercizio 4.6: Il server dell'ora, per molti (⋆)

Scrivere il server dell'ora dell'esercizio 2.6 in modo che serva più client insieme: ogni client, dopo essersi collegato, riceve una riga con l'ora ogni secondo finché non scrive `fine` (o chiude). Un thread virtuale per client; il server stampa nel log quanti client sono collegati a ogni ingresso e uscita. Provarlo con tre `nc` aperti insieme.

Soluzione: progetto `sol-d06-c04`, package `it.unibs.pajc.redolfi.e06`.

Attenzione 4.2: Il thread che scrive e quello che legge

Ogni client qui ha *due* attese: il secondo che passa e la riga `fine` che può arrivare. Un solo ciclo con `sleep` e `readLine` non può fare entrambe: servono due thread per client (uno scrive l'ora, uno legge), oppure un `setSoTimeout` di un secondo usato come orologio. In entrambi i casi la `println` dell'ora e l'eventuale risposta a `fine` scrivono sullo stesso socket.

Esercizio 4.7: Le stanze (⋆⋆)

Estendere WhatsPAJC con le stanze: `/entra nome-stanza` sposta l'utente nella stanza (creandola se non esiste; all'ingresso tutti sono nella stanza `generale`), `/stanze` elenca le stanze con il numero di utenti, `/chi` elenca solo chi è nella propria stanza, e i messaggi a tutti arrivano solo alla propria stanza; i privati funzionano fra stanze diverse. Estendere protocollo, test e registro; una stanza vuota scompare.

Soluzione: progetto `sol-d06-c04`, package `it.unibs.pajc.redolfi.e07`.

Attenzione 4.3: La stanza come stato condiviso

La mappa da stanza a insieme di sessioni è toccata da tutti i thread: spostare un utente da una stanza all'altra sono due operazioni (togliere, aggiungere), e un `aTutti` che scorre la stanza nel frattempo può vederlo in nessuna delle due o in entrambe. Decidere che cosa è accettabile (perdere un messaggio durante lo spostamento lo è) e proteggere lo spostamento con un lock, o accettare la finestra e documentarla.

Esercizio 4.8: Inattività e `/ping` (⋆⋆)

Modificare il server perché un utente che non scrive nulla per due minuti riceva un avviso (`INFO sei ancora lì?`) e, dopo un altro minuto di silenzio, venga salutato con `CIAO` e scollegato; un nuovo comando `/ping`, a cui il server risponde `PONG`, azzera il conteggio senza mandare messaggi agli altri. Usare `setSoTimeout` e `SocketTimeoutException` per misurare il silenzio, e tenere in un campo quante volte di seguito è scaduto.

Soluzione: progetto `sol-d06-c04`, package `it.unibs.pajc.redolfi.e08`.

Esercizio 4.9: Il tris in rete (⋆⋆⋆)

Scrivere un server arbitro per il tris: i client si collegano e scelgono un nome; il server accoppia i primi due liberi in una partita, dice a ciascuno il simbolo (`PARTITA X Giulia` o `PARTITA O Marco`) e a chi tocca (`TOCCA`); una mossa è `/m riga colonna`; il server la rifiuta se non è il turno o la casella è occupata (`ERR`), altrimenti la comunica a entrambi (`MOSSA X 1 2`), controlla la fine (`VINCE X`, `PATTA`) e rimette i due nella lista dei liberi. Progettare il protocollo e scrivere il client di console.
Suggerimento: la partita è un oggetto condiviso fra due sessioni, con la griglia e il turno; le sue modifiche vanno sincronizzate su di lui, e ciascuna sessione tiene un riferimento alla propria partita (o `null` se è libera).

Soluzione: progetto `sol-d06-c04`, package `it.unibs.pajc.redolfi.e09`.

Risposte

Risposta 4.1. Giulia riceve `ciao` due volte: una da `trova(dest).invia`, una da `aTutti`, che viene eseguito in ogni caso perché non c'è un `else` né un `continue`; tutti gli altri ricevono il messaggio privato. Con `@Luca`, `trova` restituisce `null` e `invia` su `null` lancia `NullPointerException`: la sessione di Marco esce dal ciclo e si chiude, e per giunta prima dell'`aTutti`. Corretto: se `dest` è `null` o vale `all`, `aTutti`; altrimenti, se `trova(dest)` non è `null`, `invia` su quella sessione; altrimenti un errore al mittente. Tre rami che si escludono, e nessun `invia` su un riferimento non controllato (nel nostro server tutto questo è l'azione `privato`, e `elabora` ha già garantito che il destinatario non sia vuoto).

Risposta 4.2. Luca si collega senza errori (il backlog), scrive e non riceve niente, nemmeno l'`INFO` di benvenuto, perché la sua sessione non è ancora partita: è nella coda dell'executor. Nel log non compare nulla di lui. Quando uno dei quattro scrive `/esci`, il suo thread si libera, prende dalla coda la sessione di Luca, e Luca riceve di colpo l'`INFO` e poi la risposta al `/nome` che aveva scritto un minuto prima. Con il pool cached la sessione parte subito su un thread nuovo; con i thread virtuali anche, e il thread costa quasi nulla.

Risposta 4.3. Il thread di Marco esegue la `println` di `MSG Marco buongiorno` sul `PrintWriter` di Giulia: scrive i caratteri nel buffer, poi l'a capo, poi `flush`. Se viene sospeso dopo aver scritto `MSG Marco buon`, e il thread di Sara scrive `MSG Sara ciao` più a capo e `flush` sullo stesso buffer, Giulia riceve una riga sola fatta di `MSG Marco buon` seguito da `MSG Sara ciao`, e poi `giorno` come riga seguente: la prima ha un testo strano, la seconda non è nel protocollo, e il client la mostra com'è. Con tre persone che scrivono a mano, due `println` nello stesso microsecondo sono rarissime; con cento client o con un programma che manda messaggi a raffica succede subito. È il tipo di errore che "non si riproduce" e che *Thread e concorrenza* insegna a non cercare con le prove ma con il ragionamento.

Risposta 4.4. Un minuto dopo, Sara è ancora in `/chi`: nessuno ha chiuso il suo socket, e TCP non manda nulla da solo. Quando Marco scrive a tutti, `invia` sulla sessione di Sara scrive nel buffer, il sistema operativo prova a spedire e, non ricevendo conferma, ritrasmette per minuti; il `PrintWriter` non segnala niente. Sara esce dal registro quando il suo `readLine` lancia: o per il timeout di mezz'ora (`SocketTimeoutException`, via `finally`), oppure prima, se il sistema operativo del server rinuncia alle ritrasmissioni e segnala un errore sul socket (una `SocketException`, dopo un tempo che dipende dal sistema, in genere qualche minuto). Senza `setSoTimeout`, e senza nessuno che le scriva, Sara resterebbe nel registro per sempre.

Risposta 4.5. Scambiando le righe, fra il `CIAO` e la chiusura del `ServerSocket` un nuovo client potrebbe essere accettato e ricevere una sessione su un server che sta chiudendo; la differenza è piccola, ma l'ordine giusto è "prima la porta, poi gli ospiti". Con la sola variabile, `avvia` è fermo in `accept`, che non legge nessuna variabile: la controllerebbe solo *dopo* il prossimo client accettato, cioè il server si fermerebbe alla connessione successiva a `stop`, e quel client verrebbe accettato e poi abbandonato. L'alternativa con la variabile funziona solo insieme a `server.setSoTimeout(ms)`, che fa uscire `accept` a intervalli con `SocketTimeoutException`, lasciando al ciclo l'occasione di controllarla.

Capitolo 5

Il client con la finestra

Il server del capitolo 4 serve molti client, ma i client sono ancora `nc` e la console. WhatsPAJC vuole una finestra: l'elenco di chi c'è, la conversazione, un campo in cui scrivere. È la terza domanda della dispensa vista dal lato del client, e si incontra con una regola che conosciamo da *Thread e concorrenza*: i componenti di Swing vivono su un solo thread, quello degli eventi, che non va mai bloccato. Una lettura dalla rete blocca, per sua natura; una finestra non può aspettare. In mezzo serve un thread, e serve il modo giusto di passargli le cose. Alla fine del capitolo il client di WhatsPAJC è completo: una connessione dietro una finestra, un Model che non sa di socket e che si prova senza rete, una View che lo ascolta, un Controller che traduce nei due versi, e tutto ciò che si vede solo quando si è in due.

5.1 Il problema

Riprendiamo la finestra del capitolo 1, scritta come la scriverebbe chi ha appena letto il capitolo 2. Il listener del pulsante *Connetti* apre il socket e legge:

```
btnConnetti.addActionListener(e -> {
    try {
        Socket socket = new Socket(host, 1234);             // SBAGLIATO:
    sull'EDT
        BufferedReader in = new BufferedReader(new InputStreamReader(
                socket.getInputStream(), StandardCharsets.UTF_8));
        String riga;
        while ((riga = in.readLine()) != null)              // blocca l'EDT
            txtConversazione.append(riga + "\n");
    } catch (IOException ex) {
        ex.printStackTrace();
    }
});
```

Il listener gira sul thread degli eventi (`AWT-EventQueue-0`), e non ritorna finché il server non chiude: per tutto quel tempo nessun altro evento viene servito, la finestra non si ridisegna, i pulsanti non rispondono, e il sistema operativo la segnala come bloccata. Non è nemmeno necessario che arrivi qualcosa: già `new Socket` può durare secondi, se il server non risponde. La prima correzione viene da sé, dopo *Thread e concorrenza*: la lettura in un thread a parte. La finestra torna viva, e dopo un po' la conversazione mostra righe doppie e spezzate, e ogni tanto un'eccezione dentro Swing. La `JTextArea` viene modificata dal thread lettore mentre il thread degli eventi la disegna, e i componenti non sono fatti per questo. Il quadro è il seguente, ed è l'immagine guida con il filo che finisce in una finestra:



Il thread lettore non tocca niente: accoda un compito, e l'EDT lo esegue fra un evento e l'altro.

Il thread lettore ha un solo compito: leggere una riga e *accodarla*, così com'è, al thread degli eventi con `invokeLater`; sarà il Controller, sull'EDT, a leggerne la prima parola. Tutto il resto, dall'aggiornamento dello stato al disegno, avviene sull'EDT, un evento alla volta, come ogni altra cosa in una finestra Swing. Il thread lettore non conosce nessun componente, e per questo non può romperne nessuno.

5.2 La connessione dietro la finestra

Incapsuliamo questo meccanismo in una classe, `Connessione`, così che il resto del client non veda mai socket, stream o thread. Riceve host e porta, e due funzioni: che cosa fare con ogni risposta e che cosa fare quando la connessione finisce. Entrambe vengono chiamate sull'EDT.

```
public class Connessione implements AutoCloseable {
    private final Socket socket;
    private final PrintWriter out;
    private final Thread lettore;

    public Connessione(String host, int porta, Consumer<String> ricevi,
                       Consumer<String> chiusa) throws IOException {
        socket = new Socket();
        socket.connect(new InetSocketAddress(host, porta), 3000);
        out = new PrintWriter(socket.getOutputStream(), true,
    StandardCharsets.UTF_8);
        BufferedReader in = new BufferedReader(new InputStreamReader(
                socket.getInputStream(), StandardCharsets.UTF_8));
        lettore = Thread.ofVirtual().unstarted(() -> {
            String motivo = null;
            try {
                in.lines().forEach(riga ->                              // fino a
    null
                        SwingUtilities.invokeLater(() -> ricevi.accept(riga)));
            } catch (UncheckedIOException e) {
                motivo = socket.isClosed() ? null : e.getCause().getMessage();
            }
            String m = motivo;
            SwingUtilities.invokeLater(() -> chiusa.accept(m));
        });
        lettore.start();
    }
```

Il ciclo del lettore è quello di sempre, scritto con `lines()` (*Stream, per intero*): per ogni riga, un compito accodato all'EDT che la consegna, e il lettore passa subito alla riga seguente. `riga` è il parametro della lambda, quindi effettivamente final, e ogni compito accodato porta la sua riga. `lines()` incarta la `IOException` del `readLine` in una `UncheckedIOException`, che il lettore raccoglie alla fine; qualunque essa sia, avvisa: `null` se la chiusura è normale (il server ha chiuso, oppure siamo stati noi: in quel caso il `readLine` lancia, ma il socket risulta già chiuso), altrimenti il messaggio della causa. Il thread è virtuale perché passa la vita in `readLine`, ed è il caso ideale che il capitolo 4 ha discusso; un thread di piattaforma andrebbe bene lo stesso. Il resto della classe:

```
public void invia(String riga) {
        out.println(riga);
    }

    @Override
    public void close() {
        try {
            socket.close();                 // il lettore esce dal readLine
        } catch (IOException e) {
            // gia' chiuso
        }
    }
}
```

`invia` viene chiamato dall'EDT, nel listener del campo di testo, e è lecito chiedersi se non sia un'altra operazione bloccante sul thread sbagliato. In linea di principio sì: `println` scrive nel buffer del sistema operativo, e se il buffer è pieno (l'altro lato non legge da tempo, la rete è ferma) aspetta. In pratica una riga di chat sono poche decine di byte, il buffer ne tiene decine di migliaia, e il server legge sempre: l'attesa non si verifica, e il codice resta semplice. Quando non è così (un file da mandare, un server che può fermarsi a leggere) la soluzione è un esecutore a un solo thread per le scritture, a cui l'EDT consegna le righe e che le scrive con calma; lo useremo nel capitolo 6 per la foto. La chiusura, infine, è il modo visto per il server: chiudere il socket fa uscire il lettore dal `readLine` con un'eccezione, che il lettore riconosce come chiusura nostra.

Osservazione 5.1

C'è una seconda via, che *Thread e concorrenza* ha mostrato: un `SwingWorker` il cui `doInBackground` legge le righe e le passa a `publish`, e il cui `process` le riceve sull'EDT. Fa la stessa cosa di `invokeLater`, con la coda gestita dal worker, e in più raggruppa le righe arrivate vicine in una sola chiamata di `process`. È la scelta giusta quando le righe sono molte e veloci (un log che scorre); per una chat le due forme si equivalgono, e `Connessione` usa la più esplicita. Il `SwingWorker` lo usiamo invece per *aprire* la connessione, nella sezione 5.4: un lavoro che dura e finisce.

5.3 Il Model non sa di socket

Che cosa *sa* il client? Se è connesso e con quale nome, chi c'è, le righe della conversazione. È lo stato, e lo mettiamo in una classe senza un solo `import` di `java.net` o di `javax.swing`: `ChatModel`. La finestra lo legge e lo ascolta; il Controller lo modifica quando arriva una risposta. Per avvisare chi ascolta usiamo `PropertyChangeSupport`, nel modo che la dispensa *Pattern in gioco* spiega per esteso: una proprietà ha un nome, e chi la cambia avvisa con il valore vecchio e quello nuovo.

```
public class ChatModel {
    public enum Stato { DISCONNESSO, IN_CONNESSIONE, CONNESSO }

    public record Riga(Tipo tipo, String da, String a, String testo) {
        public enum Tipo { PUBBLICO, PRIVATO, SISTEMA }
    }

    private final PropertyChangeSupport ascoltatori = new
    PropertyChangeSupport(this);
    private Stato stato = Stato.DISCONNESSO;
    private String nome;
    private List<String> utenti = List.of();
    private final List<Riga> conversazione = new ArrayList<>();

    public void addPropertyChangeListener(PropertyChangeListener l) {
        ascoltatori.addPropertyChangeListener(l);
    }

    public void setStato(Stato nuovo) {
        Stato vecchio = stato;
        stato = nuovo;
        ascoltatori.firePropertyChange("stato", vecchio, nuovo);
    }

    public void setUtenti(List<String> nuovi) {
        List<String> vecchi = utenti;
        utenti = List.copyOf(nuovi);
        ascoltatori.firePropertyChange("utenti", vecchi, utenti);
    }

    public void aggiungi(Riga riga) {
        conversazione.add(riga);
        ascoltatori.firePropertyChange("riga", null, riga);
    }

    public void errore(String testo) {
        ascoltatori.firePropertyChange("errore", null, testo);
    }
```

Una `Riga` della conversazione è un record con il tipo (pubblico, privato, di sistema), chi la manda, a chi, e il testo: è ciò che la finestra mostrerà, non ciò che il server ha mandato, e la distinzione è il punto. I `PRIV` e i `MSG` del protocollo diventano righe; gli `ENTRA` ed `ESCE` diventano modifiche dell'elenco *e* righe di sistema; un `ERR` diventa, a seconda del momento, una riga di sistema o un errore da mostrare. L'ultimo metodo, `errore`, non cambia nessuno stato: è un evento ("c'è qualcosa da dire all'utente"), e `PropertyChangeSupport` lo trasporta lo stesso, con `null` come valore vecchio così che non venga mai scartato come "nessun cambiamento". Nel Model c'è anche `setNome`, `entra` ed `esce` per l'elenco, `sistema` per le righe grigie, e `svuota`.

Il premio di questa separazione lo abbiamo già riscosso: le schermate di questo capitolo sono state prodotte da un programma che costruisce la finestra, riempie il Model come farebbe il server (tre utenti, sei righe) e la dipinge, senza nessun server e nessun socket. Lo stesso Model si può provare con JUnit ("dopo `entra("Sara")` l'elenco ha tre nomi ordinati"), e lo

stesso Model potrebbe stare dietro un'altra finestra, o una console, senza cambiare una riga.

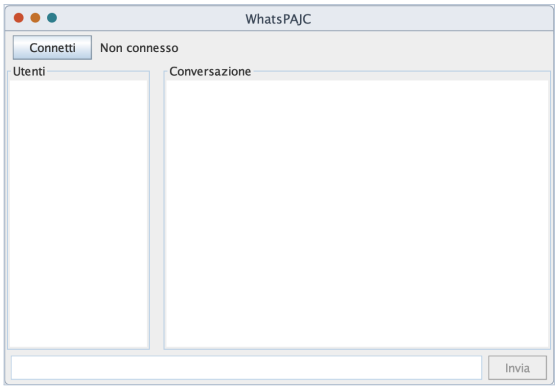
5.4 La View e il Controller

La finestra, `ChatView`, è composta con gli strumenti di *Interfacce grafiche con Swing*: una `JList` con un `DefaultListModel` per gli utenti, un `JTextPane` per la conversazione (un'area di testo che accetta stili: i privati in corsivo e colore, le righe di sistema in grigio), un `JTextField` con un pulsante in basso, una barra in alto con *Connetti* e la riga di stato. Quello che conta qui è come si collega al Model: si registra come ascoltatore e reagisce al nome della proprietà.

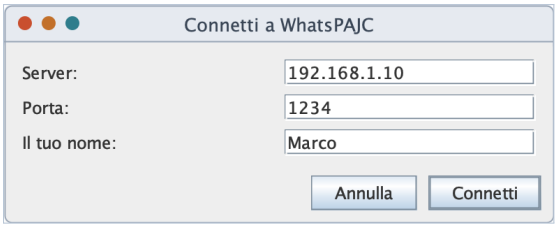
```
public ChatView(ChatModel model) {
    super("WhatsPAJC");
    this.model = model;
    // ... composizione dei componenti ...
    model.addPropertyChangeListener(this::modelCambiato);
    aggiornaStato();
}

private void modelCambiato(PropertyChangeEvent e) {
    switch (e.getPropertyName()) {
        case "stato", "nome" -> aggiornaStato();
        case "utenti" -> {
            utenti.clear();                          // il DefaultListModel
            utenti.addAll(model.utenti());
        }
        case "riga" -> aggiungiRiga((ChatModel.Riga) e.getNewValue());
        case "errore" -> JOptionPane.showMessageDialog(this, e.getNewValue(),
                "WhatsPAJC", JOptionPane.ERROR_MESSAGE);
        default -> { }
    }
}
```

`aggiornaStato` legge lo stato dal Model e sistema tutto ciò che ne dipende: il testo della riga di stato, il pulsante che dice *Connetti* o *Disconnetti*, il campo di testo abilitato solo da connessi. È lo schema "dato lo stato, ecco come appare la finestra" di *Interfacce grafiche con Swing*: nessun listener cambia un componente direttamente, tutti cambiano il Model e la finestra si rifà. Dal lato opposto, la View espone tre punti in cui il Controller si aggancia: che cosa fare con il testo scritto, con il pulsante *Connetti*, con il doppio click su un utente.



La finestra appena aperta: lo stato è `DISCONNESSO`, il campo è spento.



Il dialogo di connessione: server, porta e nome, e un `Optional` come risultato.

5.4.1 Il Controller

Il Controller tiene la `Connessione` e traduce nei due versi. Dall'utente al server la traduzione è minima: il testo del campo parte com'è, una riga. È il server ad avere la tabella, e a rispondere `ERR` a ciò che non va; il client non ha bisogno di conoscere i comandi per mandarli, e un comando nuovo nel server funziona da un client vecchio. Solo `/esci` viene intercettato, perché chiude anche da questo lato.

```
public class ChatController {
    private final ChatModel model;
    private final ChatView view;
    private Connessione connessione;

    public ChatController(ChatModel model, ChatView view) {
        this.model = model;
        this.view = view;
        view.onConnetti(this::connettiODisconnetti);
        view.onInvia(this::scritto);
        view.onUtenteScelto(view::preparaPrivato);     // doppio click: un privato
    }

    public void scritto(String testo) {
        if (connessione == null) return;
        String riga = testo.replace('\n', ' ').strip();
        if (riga.isEmpty()) return;
        if (riga.equalsIgnoreCase("/esci")) { disconnetti(); return; }
        connessione.invia(riga);
    }
```

Il `replace` degli a capo è la regola del protocollo (un messaggio è una riga) fatta rispettare dal client, come promesso nel capitolo 3: da un campo di testo a una riga non possono entrare, ma il Controller non lo dà per scontato. Dal server all'utente la traduzione è la stessa idea del server, rovesciata: una tabella con la prima parola della riga come chiave e, per ogni parola, che cosa fare del resto. Il Controller la riempie nel costruttore:

```
private final Map<String, Consumer<String>> risposte = new HashMap<>();

    private void tabellaRisposte() {
        risposte.put("INFO", model::sistema);
        risposte.put("BENVENUTO", nome -> {
            model.setNome(nome);
            connessione.invia("/chi");                          // e chiedo chi
    c'e'
        });
        risposte.put("UTENTI", elenco -> model.setUtenti(
                elenco.isEmpty() ? List.of() : List.of(elenco.split(","))));
        risposte.put("MSG", resto -> riga(ChatModel.Riga.Tipo.PUBBLICO, resto,
    2));
        risposte.put("PRIV", resto -> riga(ChatModel.Riga.Tipo.PRIVATO, resto,
    3));
        risposte.put("ENTRA", nome -> {
            model.entra(nome);
            model.sistema(nome + " e' entrato");
        });
        risposte.put("ESCE", nome -> {
            model.esce(nome);
            model.sistema(nome + " e' uscito");
        });
        risposte.put("ERR", this::errore);
        risposte.put("CIAO", resto -> model.sistema("il server ha chiuso"));
    }
```

Il metodo che `Connessione` chiama sull'EDT per ogni riga spezza la riga in parola e resto, e cerca la parola nella tabella; per la parola che non c'è, `getOrDefault` fornisce un'azione di riserva che mostra la riga com'è:

```
void ricevuta(String riga) {
        String[] campi = riga.split(" ", 2);
        String resto = campi.length > 1 ? campi[1] : "";
        risposte.getOrDefault(campi[0], r -> model.sistema("? " + riga))
                .accept(resto);
    }

    private void riga(ChatModel.Riga.Tipo tipo, String resto, int campi) {
        String[] c = Protocollo.argomenti(resto, campi);   // "da testo" / "da a
    testo"
        if (c == null) { model.sistema("? " + resto); return; }
        model.aggiungi(campi == 2 ? new ChatModel.Riga(tipo, c[0], null, c[1])
                                  : new ChatModel.Riga(tipo, c[0], c[1], c[2]));
    }

    private void errore(String motivo) {
        if (model.nome() == null) {                        // il nome e' stato
    rifiutato
            model.errore(motivo);
            disconnetti();
        } else {
            model.sistema("errore: " + motivo);
        }
    }
```

`MSG` e `PRIV` hanno il resto composto da più campi, e per spezzarlo si usa lo stesso `argomenti` del server, con il testo libero che tiene i suoi spazi: una riga malformata non fa lanciare niente, viene mostrata con un punto interrogativo. Tre scelte di questa tabella meritano attenzione. Dopo il `BENVENUTO` il client chiede da solo `/chi`, perché l'elenco iniziale non arriva da sé: il protocollo non lo prevede, e il client se lo procura. Un `ERR` ricevuto *prima* di avere un nome vuol dire che il nome è stato rifiutato (già in uso, non valido): in quel caso non ha senso restare connessi, e il Controller mostra l'errore e chiude; un `ERR` dopo è un comando sbagliato, e finisce nella conversazione come riga di sistema. E la parola sconosciuta non ferma niente: un client di questa versione collegato a un server più nuovo mostra le righe che non capisce e continua a funzionare per tutto il resto.

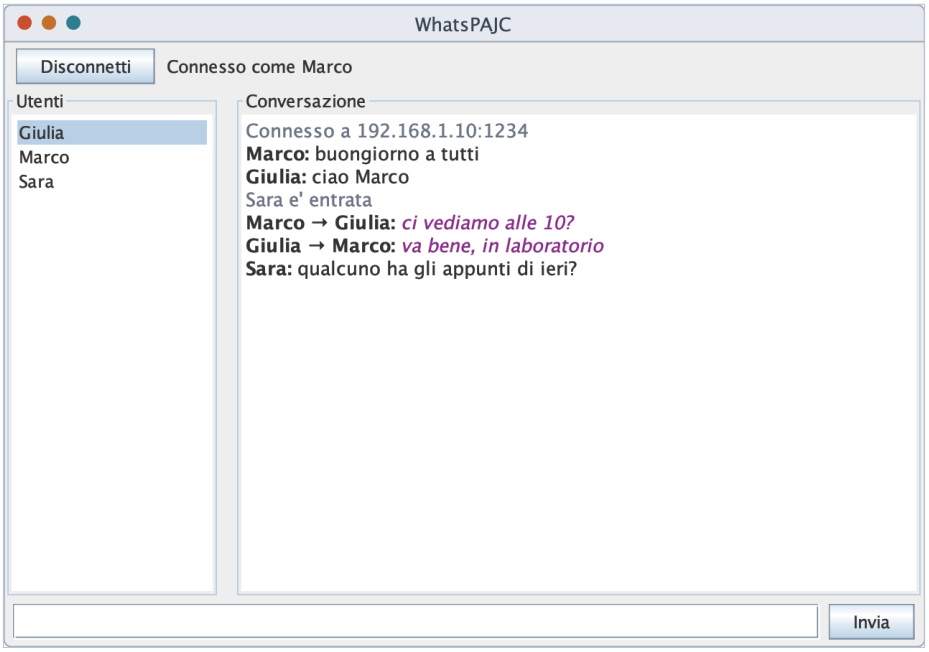
5.4.2 Aprire la connessione

Resta l'apertura. `new Connessione(...)` fa un `connect`, che può durare fino a tre secondi se il server non risponde, e un'operazione che può durare secondi non si fa sull'EDT. È il caso tipico del `SwingWorker`: un lavoro in background con un risultato, e un `done` sull'EDT che lo raccoglie.

```
public void connetti(DialogConnessione.Parametri p) {
        model.svuota();                          // prima che arrivi la prima riga
        model.setStato(ChatModel.Stato.IN_CONNESSIONE);
        new SwingWorker<Connessione, Void>() {
            @Override
            protected Connessione doInBackground() throws IOException {
                return new Connessione(p.host(), p.porta(),
                        ChatController.this::ricevuta,
    ChatController.this::chiusa);
            }

            @Override
            protected void done() {
                try {
                    connessione = get();
                    model.setStato(ChatModel.Stato.CONNESSO);
                    model.sistema("Connesso a " + p.host() + ":" + p.porta());
                    connessione.invia("/nome " + p.nome());
                } catch (Exception e) {
                    model.setStato(ChatModel.Stato.DISCONNESSO);
                    model.errore("Impossibile connettersi a " + p.host() + ":" +
    p.porta()
                            + "\n" + e.getCause().getMessage());
                }
            }
        }.execute();
    }
```

Durante l'attesa il Model è in `IN_CONNESSIONE`, e la View, che lo ascolta, mostra "Connessione in corso…" e spegne il pulsante. Lo `svuota` sta *prima* del worker e non in `done`: il lettore parte dentro il costruttore di `Connessione`, sul thread del worker, e la prima riga del server (l'`INFO` di benvenuto) può essere già in coda sull'EDT quando `done` viene eseguito; uno svuotamento lì la cancellerebbe. Se `connect` fallisce, `get()` rilancia l'eccezione incartata in una `ExecutionException`, e il messaggio vero (*Connection refused*, *timed out*, *nome sconosciuto*: le eccezioni del capitolo 2) sta nella causa. Se riesce, il Controller manda subito il `/nome` scelto nel dialogo. I parametri vengono da `DialogConnessione`, una finestra modale che restituisce un `Optional<Parametri>` (il record con host, porta e nome), vuoto se l'utente annulla, nello stile di *Interfacce grafiche con Swing*: il dialogo controlla i campi (il nome con la stessa regola del protocollo, la porta numerica) e non tocca la finestra che lo ha aperto.

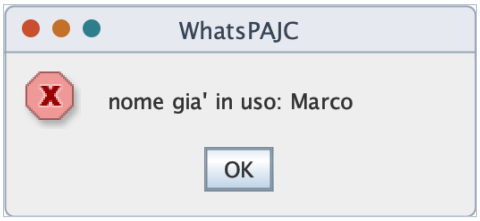


WhatsPAJC connesso come Marco: i messaggi a tutti, i privati in corsivo, le righe di sistema in grigio, l'elenco a sinistra.

5.5 Le cose che si vedono solo in due

Un client di rete si prova in due, e alcune situazioni compaiono solo allora. Le passiamo in rassegna, perché il codice che le gestisce è già nelle pagine precedenti e vale la pena vederlo all'opera.

Il nome rifiutato. Giulia è già collegata e Marco, sull'altro computer, sceglie "Giulia". Il server risponde `ERR nome gia' in uso: Giulia`; il Controller, che non ha ancora un nome nel Model, lo mostra in una finestra e chiude la connessione; la riga di stato torna a "Non connesso" e il pulsante a *Connetti*. Un secondo tentativo con un altro nome riparte da capo, con un socket nuovo e un lettore nuovo: il vecchio è finito da solo quando il socket è stato chiuso.



Il nome è già in uso: l'`ERR` arrivato prima del `BENVENUTO` diventa un errore a video, e la connessione si chiude.

Il server che cade. Il server viene fermato (o il computer su cui gira si spegne) mentre tutti sono collegati. Se è stato fermato con `stop`, ogni client riceve `CIAO` e poi `null`: `chiusa(null)`, e la finestra dice che il server ha chiuso. Se è sparito, il lettore riceve *Connection reset*, o non riceve nulla finché il sistema operativo non se ne accorge: `chiusa("Connection reset")`, e la finestra lo dice. In entrambi i casi la finestra *non muore*: lo stato torna `DISCONNESSO`, l'elenco si svuota, e l'utente può riconnettersi quando il server torna. È il motivo per cui `chiusa` esiste, e per cui il Controller distingue la chiusura chiesta da noi (`connessione == null`: già gestita) da quella subita.

I messaggi mentre si scrive. Marco sta scrivendo una frase lunga e intanto arrivano tre messaggi di Giulia. Il thread lettore li accoda all'EDT, che li esegue fra un carattere e l'altro: la conversazione cresce, il campo di testo conserva ciò che Marco ha scritto, niente si mescola. Nel client a passo del capitolo 1 quei messaggi sarebbero arrivati solo dopo l'Invio; qui arrivano subito, perché nessuno aspetta nessuno.

La disconnessione. *Disconnetti* manda `/esci`, chiude il socket e azzera la connessione; il server risponde `CIAO` e chiude a sua volta, e il lettore, che trova il socket già chiuso, chiama `chiusa(null)`: il Controller vede che la connessione è già `null` e non mostra niente. L'ordine delle due chiusure non è garantito, e il codice funziona in entrambi.

5.6 L'esempio guida: WhatsPAJC v3

Il client è completo e si monta nel `main` come ogni applicazione MVC: il Model, la View che lo ascolta, il Controller che li collega, tutto sull'EDT.

```
public class ClientApp {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            ChatModel model = new ChatModel();
            ChatView view = new ChatView(model);
            new ChatController(model, view);
            view.setVisible(true);
        });
    }
}
```

Con il server del capitolo 4 in aula, ognuno avvia il client, preme *Connetti*, scrive l'indirizzo del server e il proprio nome, e chiacchiera. Un doppio click su un nome prepara un privato; `/chi`, `/aiuto` e gli altri comandi funzionano dal campo di testo perché arrivano al server così come sono scritti, e la tabella del server li esegue; chi scrive `/esci` si disconnette. Le classi sono sei (`Protocollo`, condivisa con il server per `argomenti` e il controllo del nome, `Connessione`, `ChatModel`, `ChatView`, `ChatController`, `DialogConnessione`) e nessuna fa due mestieri.

Ricetta: Mettere una connessione dietro una finestra

1. una classe `Connessione` che apre il socket, avvia un thread lettore e consegna ogni messaggio sull'EDT con `invokeLater`; `invia` dall'EDT per righe corte, `close` che chiude il socket;
2. un Model senza `java.net` né `javax.swing`, che avvisa con `PropertyChangeSupport` e che si riempie a mano per le prove;
3. una View che ascolta il Model e si rifà dallo stato, e che espone le azioni dell'utente come callback;
4. un Controller che manda le righe dell'utente e traduce quelle del server in modifiche del Model con una tabella da parola ad azione, come nel server, con un'azione di riserva per la parola sconosciuta; l'apertura in un `SwingWorker`;
5. ogni modo in cui la connessione può finire (il server chiude, cade, il nome è rifiutato, l'utente si disconnette) riporta il Model a `DISCONNESSO` senza chiudere la finestra.

Esercizi

Esercizio 5.1: Su quale thread (⋆)

Per ciascuna riga di `Connessione` e del Controller dire su quale thread viene eseguita: il `connect` nel costruttore; la lettura delle righe in `in.lines()`; `ricevi.accept(riga)`; lo `split` in `ricevuta`; `out.println` in `invia`; `socket.close()` in `close`. Quale di queste potrebbe bloccare l'EDT, e in quali condizioni?

Esercizio 5.2: Quasi sempre (⋆)

Un compagno toglie l'`invokeLater` dal thread lettore e chiama direttamente `ricevi.accept(riga)`, e dice che "funziona lo stesso". Spiegare perché ha quasi sempre ragione e sempre torto: che cosa fa il thread lettore sul `JTextPane`, che cosa può succedere, e perché il problema si manifesta più spesso con un server che manda molte righe di seguito.

Esercizio 5.3: Il Model senza `invokeLater` (⋆⋆)

Supponiamo che il lettore chiami `ricevuta(riga)` direttamente, senza passare per l'EDT, ma che la View, nel suo ascoltatore, incarti ogni modifica dei componenti in un `invokeLater`. I componenti sono al sicuro? E il Model? Descrivere un caso in cui `model.entra("Sara")` dal thread lettore e `model.svuota()` dall'EDT si intrecciano, e dire che cosa può vedere l'elenco.

Esercizio 5.4: `ENTRA` e `UTENTI` (⋆⋆)

Marco si collega e manda `/nome Marco`. Nello stesso istante Sara si collega e manda `/nome Sara`. Dire in quali ordini possono arrivare a Marco le righe `BENVENUTO Marco`, `ENTRA Sara` e `UTENTI ...` (quest'ultima in risposta al `/chi` che il Controller manda da solo), e che cosa mostra l'elenco alla fine in ciascun caso. Perché il Controller usa `entra`, che non duplica, invece di aggiungere e basta?

Esercizio 5.5: Il server cade (⋆⋆)

Il processo del server viene ucciso. Descrivere, nell'ordine, che cosa fa il thread lettore, quale dei due `Consumer` viene chiamato e con quale argomento, che cosa fa il Controller, e che cosa vede l'utente. Poi lo stesso con il server fermato con `stop`.

I problemi che seguono chiedono un programma completo, in ordine crescente di difficoltà. Le soluzioni sono nel repository delle soluzioni del corso, nel progetto e nel package indicati.

Esercizio 5.6: L'ora in finestra (⋆)

Scrivere il client con la finestra del server dell'ora (esercizio 2.6): un'etichetta grande e un pulsante *Aggiorna*, che si collega, legge la riga, chiude e mostra l'ora. La connessione e la lettura vanno in un `SwingWorker`; durante l'attesa il pulsante è spento; un errore (server fermo, host sbagliato) compare in un `JOptionPane` e non chiude il programma.

Soluzione: progetto `sol-d06-c05`, package `it.unibs.pajc.redolfi.e06`.

Esercizio 5.7: La calcolatrice remota in finestra (⋆⋆)

Scrivere il client con la finestra della calcolatrice remota (esercizio 3.6): un campo per l'espressione (`ADD 2 3`), un pulsante, e uno storico delle richieste con le risposte in una `JList`; gli `ERR` del server compaiono nello storico in rosso. La connessione resta aperta fra una richiesta e l'altra, con una classe `Connessione` come quella del capitolo; il protocollo è a passo, quindi il Controller tiene in un campo l'ultima richiesta in attesa e la abbina alla risposta.

Soluzione: progetto `sol-d06-c05`, package `it.unibs.pajc.redolfi.e07`.

Attenzione 5.1: Il protocollo a passo nel client a eventi

Se l'utente preme il pulsante due volte prima che arrivi la prima risposta, le due risposte arrivano nell'ordine delle richieste, e il Controller deve abbinarle: una coda delle richieste in attesa, non un solo campo, oppure il pulsante spento finché la risposta non arriva. Scegliere e dirlo nel commento.

Esercizio 5.8: Le conversazioni separate (⋆⋆)

Modificare il client di WhatsPAJC perché la conversazione con ogni utente stia in una scheda a parte (`JTabbedPane`): la scheda *Tutti* per i messaggi pubblici, una scheda per ogni utente con cui si sono scambiati privati, aperta al primo privato in arrivo o in uscita; il titolo della scheda mostra fra parentesi i messaggi non letti (quelli arrivati mentre era selezionata un'altra scheda), e scrivere in una scheda privata manda un `/a` a quell'utente senza doverlo scrivere. Il Model va esteso (conversazioni per interlocutore e contatori) e la View si rifà dal Model.

Soluzione: progetto `sol-d06-c05`, package `it.unibs.pajc.redolfi.e08`.

Esercizio 5.9: Il tris in finestra (⋆⋆⋆)

Scrivere il client con la finestra per il server arbitro del tris (esercizio 4.9): un pannello che disegna la griglia con i simboli, un click su una casella manda la mossa, la riga di stato dice se tocca a noi o all'avversario, e la fine della partita (vittoria, patta, abbandono) compare in un dialogo con la proposta di una nuova partita. Il Model tiene la griglia, il turno e lo stato della partita; il pannello si disegna dal Model.
Suggerimento: "non è il tuo turno" è uno stato del Model, non un `ERR` da aspettare: il pannello ignora i click quando il turno non è nostro, e l'`ERR` del server resta per i casi in cui il Model e il server non sono d'accordo.

Soluzione: progetto `sol-d06-c05`, package `it.unibs.pajc.redolfi.e09`.

Risposte

Risposta 5.1. Il `connect` gira sul thread che costruisce la `Connessione`: nel Controller è il thread del `SwingWorker`, apposta. La lettura delle righe gira sul thread lettore. `ricevi.accept(riga)`, e con esso lo `split` di `ricevuta` e tutta la tabella, girano sull'EDT, perché sono dentro l'`invokeLater`. `out.println` gira sul thread di chi chiama `invia`, cioè l'EDT, e `socket.close()` pure. Potrebbero bloccare l'EDT `println`, se il buffer di uscita fosse pieno (l'altro lato non legge e la rete non scarica: con righe corte e un server che legge non succede), e il `connect`, se fosse chiamato dall'EDT: per questo sta nel worker.

Risposta 5.2. Chiamando `ricevi.accept(riga)` dal thread lettore, il Controller aggiorna il Model e la View modifica il `JTextPane` dal thread lettore, mentre l'EDT può essere nel mezzo di un `paint` dello stesso componente o di un'altra modifica. Il documento del `JTextPane` non è thread-safe: il risultato va da niente (quasi sempre: le due operazioni non si sovrappongono nel tempo) a testo spezzato, righe in ordine sbagliato, eccezioni `BadLocationException` o `ArrayIndexOutOfBounds` dentro Swing. Con molte righe di seguito il thread lettore passa più tempo dentro i componenti e la probabilità di incrociare l'EDT cresce: un server che manda l'aiuto in sei righe lo fa vedere prima di una chat tranquilla.

Risposta 5.3. I componenti sono al sicuro, perché li tocca solo l'EDT. Il Model no: le sue liste vengono modificate dal thread lettore e lette dall'EDT senza nessuna protezione, ed è lo stato condiviso di *Thread e concorrenza*. `entra("Sara")` legge `utenti`, costruisce una lista nuova con Sara e la assegna; se nel frattempo l'EDT esegue `svuota()`, che assegna `List.of()`, una delle due assegnazioni vince e l'altra si perde: l'elenco può contenere Sara dopo uno svuotamento, o essere vuoto dopo un ingresso, a seconda dell'ordine. In più `firePropertyChange` dal thread lettore chiama gli ascoltatori su quel thread, e la View dovrebbe ricordarsi di incartare ogni cosa. Spostare *tutto* sull'EDT, come fa `Connessione`, rende il Model un oggetto confinato a un solo thread, e il problema non esiste.

Risposta 5.4. Dal lato del server, la sessione di Marco manda `BENVENUTO` e poi, quando legge `/chi`, `UTENTI`; la sessione di Sara manda `ENTRA Sara` agli altri quando Sara si registra. Sul socket di Marco il `BENVENUTO` viene sempre prima dell'`UTENTI` (stessa sessione, stesso ordine), ma `ENTRA Sara` può cadere prima del `BENVENUTO` (se Sara si registra prima di Marco: ma allora Marco non lo riceve, perché non è ancora nel registro), fra `BENVENUTO` e `UTENTI`, o dopo `UTENTI`. Nel secondo caso l'`UTENTI` che segue contiene già Sara: `setUtenti` sostituisce l'elenco e Sara c'è una volta sola. Nel terzo, `UTENTI` non ha Sara ed `ENTRA` la aggiunge. Con un `add` cieco il secondo caso mostrerebbe Sara due volte: `entra` controlla prima di aggiungere, e l'elenco è giusto in ogni ordine.

Risposta 5.5. Server ucciso: il sistema operativo del server chiude i socket, il lettore riceve `null` da `readLine` (se la chiusura arriva pulita) oppure `SocketException: Connection reset`; nel primo caso il motivo è `null`, nel secondo è il messaggio; il lettore accoda `chiusa(motivo)` e termina. Sull'EDT il Controller trova `connessione` diversa da `null` (non siamo stati noi), la azzera, porta il Model a `DISCONNESSO` e chiama `errore`: la View svuota l'elenco, spegne il campo, riporta il pulsante a *Connetti* e mostra "Il server ha chiuso la connessione" o "Connessione persa: Connection reset". Con `stop` arriva prima `CIAO` (una riga di sistema nella conversazione) e poi `null`: stessa sequenza, con la riga in più. In nessun caso la finestra si chiude.

Capitolo 6

Oltre le righe di testo: byte, oggetti e datagrammi

Tutto ciò che WhatsPAJC ha mandato finora è una riga di testo, e con le righe di testo si va lontano. Tre cose però non ci stanno. Una foto non è una riga: sono byte, e la fine riga può comparire in mezzo. Un oggetto Java con dentro altri oggetti si può trasformare in righe, ma qualcuno deve scrivere la traduzione nei due versi. E prima di scrivere qualunque riga bisogna sapere *dove* è il server: in aula lo si chiede al vicino, ma un programma potrebbe scoprirlo da solo. Il capitolo risponde a queste tre domande con tre strumenti che la libreria offre accanto ai socket di testo, e chiude con il protocollo di testo più usato al mondo, HTTP, letto e scritto con gli oggetti della libreria. Alla fine WhatsPAJC manda foto, trova il server sulla rete locale e ha una pagina di stato che si apre nel browser: è il punto d'arrivo dell'esempio guida.

6.1 Il problema

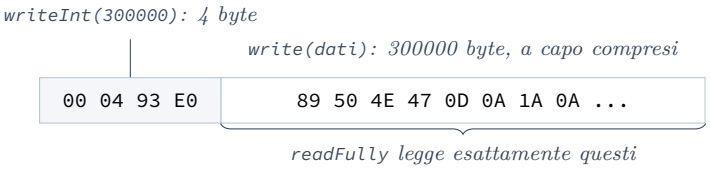
Proviamo a mandare una foto con gli strumenti del capitolo 3. Il file è un array di byte; un `PrintWriter` vuole caratteri, e i byte oltre i primi 128 non sono caratteri validi in UTF-8: trasformarli in `String` li rovina. Potremmo codificarli in testo (Base64, che esiste apposta: `java.util.Base64`) e mandarli come una riga lunghissima: funziona, costa un terzo in più, e un `readLine` che accumula tre megabyte in una stringa non è una bella idea. Il problema vero è un altro: il filo trasporta byte, la *riga* è una cornice che abbiamo scelto perché comoda per il testo, e per i byte serve una cornice diversa. Ciò che cambia è l'accordo sul confine fra un messaggio e l'altro, non il filo.

Il secondo problema è più sottile. Un messaggio dentro il programma è un oggetto con dei campi (nel client, una `Riga` della conversazione); sul filo diventa `MSG Marco ciao`, e qualcuno fa la traduzione, con uno `split` da un lato e una concatenazione dall'altro. Quando gli oggetti sono dieci e hanno liste dentro, la traduzione cresce, e viene da chiedersi se la libreria non possa farla da sé: prendere un oggetto, trasformarlo in byte, ricostruirlo dall'altra parte. Può, si chiama *serializzazione*, e vedremo perché la dispensa la mostra e non la usa.

Il terzo è pratico. Il dialogo di connessione chiede l'indirizzo del server, e in aula qualcuno lo detta; fuori dall'aula, su una rete domestica, il server è "quel computer là" e il suo indirizzo lo sanno in pochi. Un client potrebbe chiedere alla rete: "c'è un server di WhatsPAJC?". Non ha nessuno a cui collegarsi, quindi non può aprire un socket TCP: deve *gridare*, e per gridare serve l'altro servizio del livello di trasporto, UDP.

6.2 Byte con una cornice

Gli stream di un socket sono stream di byte, e `java.io` ha già gli strumenti per scriverci numeri e byte in modo portabile: `DataOutputStream` e `DataInputStream` (*Java Fast Forward* li usa per i file binari). La cornice più semplice per un blocco di byte è *la lunghezza davanti*: quattro byte con il numero dei byte che seguono, poi i byte. Chi legge sa quanto aspettare.



La cornice con la lunghezza davanti: chi legge sa quanto aspettare.

Due metodi bastano, e vale la pena scriverli una volta sola:

```
public final class Cornice {
    public static final int MAX = 20 * 1024 * 1024;       // oltre, e' un errore

    public static void scrivi(DataOutputStream out, byte[] dati)
            throws IOException {
        out.writeInt(dati.length);
        out.write(dati);
        out.flush();
    }

    public static byte[] leggi(DataInputStream in) throws IOException {
        int n = in.readInt();
        if (n < 0 || n > MAX)
            throw new IOException("cornice non valida: " + n + " byte");
        byte[] dati = new byte[n];
        in.readFully(dati);               // tutti gli n byte
        return dati;
    }
}
```

Il `readFully` è il punto. Un `read(buf)` normale restituisce i byte *disponibili adesso*, che possono essere meno di quelli chiesti: su un file sono quasi sempre tutti, su un socket quasi mai, perché i byte arrivano a pacchetti. `readFully` cicla finché il buffer non è pieno, o lancia `EOFException` se lo stream finisce prima. Il controllo su `MAX` è la regola "mai fidarsi di ciò che arriva" applicata ai byte: un `readInt` che legge spazzatura restituisce un numero qualunque, e senza il controllo il programma proverebbe ad allocare due gigabyte. Per le stringhe corte (un nome, un comando) i `Data` stream hanno `writeUTF` e `readUTF`, che mettono davanti la lunghezza in due byte: comodi, e limitati a 65535 byte, quindi non per i testi lunghi.

Attenzione 6.1: Il `BufferedReader` che mangia i byte

```
BufferedReader in = new BufferedReader(new
    InputStreamReader(socket.getInputStream()));
String comando = in.readLine();                          // "FOTO
    gatto.png"
DataInputStream dati = new DataInputStream(socket.getInputStream());
byte[] foto = Cornice.leggi(dati);                       // SBAGLIATO:
    legge spazzatura
```

Il `BufferedReader` legge dal socket *più* di quanto restituisce: riempie un buffer di qualche migliaio di byte e poi distribuisce le righe. I primi byte della cornice sono già nel suo buffer, e il `DataInputStream`, che legge dal socket nudo, trova il seguito. Su uno stesso socket si usa *un solo* lettore: o tutto testo, o tutto byte (e allora anche i comandi passano da `readUTF`), oppure si aprono due socket, uno per il testo e uno per i byte. È quello che fanno molti protocolli veri (FTP ha una connessione di comando e una di dati), ed è la scelta di WhatsPAJC: la chat resta di testo, e le foto viaggiano su una seconda porta.

6.2.1 Le foto di WhatsPAJC

Il server apre un secondo `ServerSocket`, sulla porta successiva, servito da una classe `Allegati` tutta a byte: il client si collega, dice che cosa vuole con `writeUTF` e poi parla a cornici. Due comandi: `METTI`, con il nome dell'utente, il nome del file e la cornice con i byte, a cui il server risponde `OK` e avvisa la chat con una riga nuova del protocollo, `FOTO Marco gatto.png`; e `PRENDI`, con il nome del file, a cui il server risponde con la cornice (vuota se non c'è). Il lato server di `METTI`:

```
case "METTI" -> {
    String utente = in.readUTF();
    String nome = Path.of(in.readUTF()).getFileName().toString();   // niente
    cartelle
    byte[] dati = Cornice.leggi(in);
    foto.put(nome, dati);                       // una ConcurrentHashMap<String,
    byte[]>
    out.writeUTF("OK");
    out.flush();
    registro.aTutti("FOTO " + utente + " " + nome);
}
```

Il `getFileName()` è un'altra forma di sospetto: un nome di file che arriva dalla rete potrebbe contenere `../../`, e un server che lo usasse per scrivere su disco scriverebbe dove il client vuole. Qui le foto stanno in memoria, ma l'abitudine vale sempre. Il lato client è simmetrico, e si noti che è un socket *nuovo*, aperto e chiuso per ogni foto:

```
public static void metti(String host, int porta, String utente, Path file)
        throws IOException {
    try (Socket s = new Socket(host, porta);
         DataOutputStream out = new DataOutputStream(
                 new BufferedOutputStream(s.getOutputStream()));
         DataInputStream in = new DataInputStream(s.getInputStream())) {
        out.writeUTF("METTI");
        out.writeUTF(utente);
        out.writeUTF(file.getFileName().toString());
        Cornice.scrivi(out, Files.readAllBytes(file));
        String esito = in.readUTF();
        if (!esito.equals("OK")) throw new IOException(esito);
    }
}
```

Nel client con la finestra, `metti` gira in un `SwingWorker`: una foto da qualche megabyte su una rete lenta dura secondi, e il capitolo 5 ha spiegato perché l'EDT non può aspettarli; con `publish` si può anche mostrare l'avanzamento, scrivendo la cornice a blocchi (un esercizio). Quando arriva `FOTO Marco gatto.png`, il Controller aggiunge una riga di sistema con un pulsante, e il click prende la foto con `prendi` (in un altro worker) e la mostra in un dialogo con un `JLabel` e un `ImageIcon`.

6.3 Oggetti sul filo

Java sa trasformare un oggetto in byte e ricostruirlo: è la *serializzazione*. Le classi sono `ObjectOutputStream` e `ObjectInputStream`, e si incartano attorno a qualunque stream, di un file come di un socket. La classe deve dichiararsi serializzabile implementando l'interfaccia vuota `Serializable`; i record lo possono fare come ogni classe, e per loro la ricostruzione passa dal costruttore canonico, quindi gli invarianti restano validi.

```
record Evento(String da, String testo, long quando) implements Serializable {}

ByteArrayOutputStream bo = new ByteArrayOutputStream();
try (ObjectOutputStream oo = new ObjectOutputStream(bo)) {
    oo.writeObject(new Evento("Marco", "ciao", 42));
}
try (ObjectInputStream oi = new ObjectInputStream(
        new ByteArrayInputStream(bo.toByteArray()))) {
    Evento e = (Evento) oi.readObject();            // Evento[da=Marco,
    testo=ciao, quando=42]
}
```

Centoventinove byte per tre campi: l'intestazione dello stream, il nome della classe, i nomi e i tipi dei campi, e solo in fondo i valori. Lo stesso `Evento` nel protocollo di testo sarebbe `EVENTO Marco 42 ciao`, venti byte, e si leggerebbe con `nc`. Il confronto dice già molto, ma i motivi per cui la dispensa preferisce il testo sono altri tre.

- **Il formato è la classe.** I byte contengono il nome della classe e la sua forma; se la classe cambia (un campo in più), i byte vecchi non si leggono più, a meno di gestire a mano il `serialVersionUID` e la compatibilità. Un server aggiornato e un client vecchio smettono di

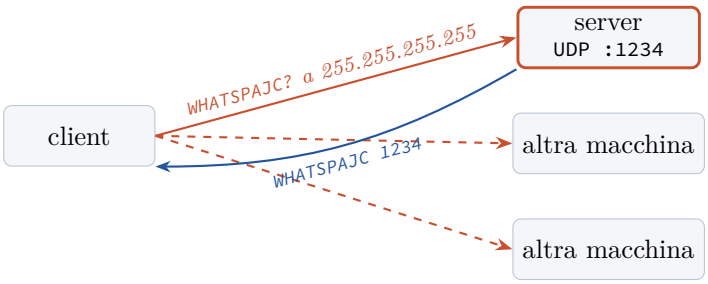
parlarsi. Con il testo si aggiunge una riga al protocollo e i client vecchi la mostrano com'è e vanno avanti.

- **Parla solo Java.** Un client in Python, in JavaScript nel browser, o `nc`, non può leggere un `ObjectOutputStream`. Il testo lo leggono tutti.
- **È pericoloso.** `readObject` costruisce *qualunque* classe serializzabile presente nel classpath, secondo ciò che i byte dicono; una sequenza di byte costruita apposta può far eseguire codice durante la ricostruzione, ed è stata la via di molti attacchi reali. La regola è assoluta: **mai deserializzare dati di cui non ci si fida**, e dati che arrivano dalla rete sono per definizione dati di cui non ci si fida. Se proprio serve, `ObjectInputFilter` (dal Java 9) permette di elencare le sole classi ammesse.

La serializzazione ha il suo posto: salvare lo stato di un programma su un file che solo lui rileggeraà, passare oggetti fra due parti della stessa applicazione. Fra due programmi diversi, oggi, gli oggetti viaggiano come testo strutturato: JSON, che è esattamente un record scritto in una sintassi che tutti i linguaggi leggono, `{"da": "Marco", "testo": "ciao", "quando": 42}`. La libreria standard di Java non ha un traduttore JSON; per un record piccolo si scrive a mano (è un esercizio) e per il resto si usa una libreria (capitolo 8). Il protocollo di WhatsPAJC, con la prima parola che dice il tipo e i campi dietro, è un JSON in miniatura.

6.4 Datagrammi: trovare il server

Torniamo al client che non sa dove è il server. TCP non può aiutarlo: una connessione ha bisogno di un indirizzo. UDP sì: un datagramma è un pacchetto spedito a un indirizzo e una porta senza nessuna connessione, e fra gli indirizzi ce n'è uno speciale, `255.255.255.255`, il *broadcast*: il pacchetto va a tutte le macchine della rete locale. Il client grida "c'è un server di WhatsPAJC?", e il server, se c'è, risponde con un datagramma al mittente. Le classi sono `DatagramSocket`, che manda e riceve, e `DatagramPacket`, che è il pacchetto con i byte, l'indirizzo e la porta.



Un datagramma per tutti; risponde solo chi ascolta sulla porta e riconosce la domanda.

Il lato server è un ciclo che riceve e risponde, su un thread suo:

```
try (DatagramSocket socket = new DatagramSocket(1234)) {       // la porta UDP
    1234
    byte[] buf = new byte[256];
    while (true) {
        DatagramPacket p = new DatagramPacket(buf, buf.length);
        socket.receive(p);                                      // aspetta un
    datagramma
        String testo = new String(p.getData(), 0, p.getLength(),
    StandardCharsets.UTF_8);
        if (!testo.equals("WHATSPAJC?")) continue;              // non e' per noi
        byte[] risposta = ("WHATSPAJC " +
    portaChat).getBytes(StandardCharsets.UTF_8);
        socket.send(new DatagramPacket(risposta, risposta.length, p.getAddress(),
    p.getPort()));
    }
}
```

Tre cose nuove. La porta UDP 1234 e la porta TCP 1234 sono porte *diverse*: i due protocolli hanno spazi di numeri separati, e il server può usare lo stesso numero per entrambi. `receive` riempie il pacchetto con i byte arrivati e con l'*indirizzo e la porta del mittente*, ed è così che il server sa a chi rispondere, senza nessuna connessione. E `getLength()`: il buffer è di 256 byte, il datagramma ne occupa dieci, e il resto è spazzatura del giro precedente. Il lato client grida e aspetta, con un timeout, perché se nessuno risponde `receive` aspetterebbe per sempre:

```
public static Optional<String> cerca(int attesaMs) throws IOException {
    try (DatagramSocket socket = new DatagramSocket()) {         // una porta a
    caso
        socket.setBroadcast(true);
        socket.setSoTimeout(attesaMs);
        byte[] domanda = "WHATSPAJC?".getBytes(StandardCharsets.UTF_8);
        socket.send(new DatagramPacket(domanda, domanda.length,
                InetAddress.getByName("255.255.255.255"), 1234));
        byte[] buf = new byte[256];
        DatagramPacket p = new DatagramPacket(buf, buf.length);
        socket.receive(p);                                        // la prima
    risposta
        String testo = new String(p.getData(), 0, p.getLength(),
    StandardCharsets.UTF_8);
        return Optional.of(p.getAddress().getHostAddress() + ":" +
    testo.substring(10));
    } catch (SocketTimeoutException e) {
        return Optional.empty();                                  // nessuno ha
    risposto
    }
}
```

Il risultato è l'indirizzo da cui è arrivata la risposta, cioè quello del server sulla rete locale, e la porta TCP che il server ha dichiarato: esattamente ciò che il dialogo di connessione chiedeva all'utente. Se ci sono due server, arrivano due risposte e il client prende la prima; se non ce ne sono, dopo il timeout l'`Optional` è vuoto e il dialogo torna a chiedere.

Osservazione 6.1

UDP dà quello che TCP non dà (il broadcast, nessuna connessione, nessuna attesa) e non dà quello che TCP dà: un datagramma può perdersi, arrivare due volte, o dopo quello spedito dopo, e nessuno lo rimanda. Per la scoperta va benissimo, perché se la domanda si perde si rifaà. Per la chat no: un messaggio perso è un messaggio perso. La regola è UDP per i segnali brevi che si possono ripetere o ignorare (la scoperta, un "sono vivo" periodico, la posizione in un gioco in tempo reale, dove conta l'ultima e non tutte), TCP per tutto ciò che deve arrivare intero e in ordine. Un datagramma deve inoltre stare in un pacchetto: in pratica sotto i 1400 byte per essere sicuri che passi ovunque.

6.5 Parlare con il web

Nel capitolo 3 abbiamo scritto una richiesta HTTP a mano, e abbiamo visto che è testo: righe, intestazioni, riga vuota, corpo. Da Java 11 la libreria ha un client HTTP fatto bene, `java.net.http.HttpClient` JAVA 11, che sa tutto ciò che il protocollo richiede (le versioni, la cifratura di `https`, i reindirizzamenti, la compressione) e che espone tre oggetti: il client, la richiesta, la risposta.

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

`send` *blocca* finché la risposta non è arrivata tutta (c'è anche `sendAsync`, che restituisce un `CompletableFuture`); da una finestra Swing va quindi in un `SwingWorker`, come ogni altra operazione di rete. Il `BodyHandler` dice che cosa fare del corpo: una stringa, un file (`ofFile`), uno stream di byte (`ofInputStream`) per i file grandi. Con `POST` e un corpo (`BodyPublishers.ofString`) si manda un dato a un servizio web, e con le intestazioni (`header("Accept", "application/json")`) si chiede un formato: è tutto ciò che serve per interrogare un servizio REST, e la risposta, di solito, è JSON.

6.5.1 La pagina di stato

Il verso opposto è un server HTTP, e per una pagina sola non serve nessuna libreria: un `ServerSocket`, la lettura della riga di richiesta e delle intestazioni, e una risposta composta come nel capitolo 3. WhatsPAJC apre una terza porta e risponde a `GET /stato` con chi c'è:

```
if (percorso.equals("/stato")) {
    StringBuilder html = new StringBuilder("<h1>WhatsPAJC</h1><p>Collegati: ")
            .append(registro.quanti()).append("</p><ul>");
    for (String n : registro.nomi())
    html.append("<li>").append(n).append("</li>");
    rispondi(out, "200 OK", "text/html; charset=utf-8",
    html.append("</ul>").toString());
} else {
    rispondi(out, "404 Not Found", "text/plain; charset=utf-8",
            "niente in " + percorso);
}
```

`rispondi` scrive la riga di stato, `Content-Type`, `Content-Length` (in *byte*, dopo la codifica in UTF-8), `Connection: close`, la riga vuota e il corpo, e chiude. Un browser che apre `http://indirizzo:1236/stato` vede la pagina; il client Java qui sopra la legge come stringa; `curl` la stampa. È un server HTTP in trenta righe, sufficiente per una pagina di diagnostica, e la prova che dietro ogni sito c'è un `accept`, un `readLine` e una `println`: per un sito vero si usa un server fatto apposta (capitolo 8), per le stesse ragioni per cui si usa `HttpClient` e non `nc`.

6.6 L'esempio guida: WhatsPAJC v4

Il server di WhatsPAJC ora ascolta su quattro porte: la chat (TCP 1234), gli allegati (TCP 1235), la pagina di stato (TCP 1236) e la scoperta (UDP 1234), ciascuna servita dalla sua classe su un thread suo, tutte avviate dal costruttore di `ServerChat` e fermate da `ferma`. Il protocollo di testo ha una riga in più, `FOTO da nome`, e tutto il resto è invariato: il capitolo 4 non ha dovuto cambiare. Il client ha un pulsante per la foto (un `JFileChooser` e poi `Allegati.metti` in un worker), mostra le foto ricevute, e il dialogo di connessione ha un pulsante *Cerca* che chiama `Scoperta.cerca` e riempie host e porta. È il punto d'arrivo dell'esempio guida; la palestra lo chiede per intero, con tutte le funzioni dei capitoli precedenti, ed è il momento di fissare che cosa si usa per che cosa.

Il messaggio è…	**Lo strumento**	**Perché**
testo breve, un dialogo	una riga, `BufferedReader`/`PrintWriter`, TCP	leggibile, provabile con `nc`, parla con tutti
byte (un file, un'immagine)	una cornice con la lunghezza, `Data` stream, TCP (meglio su un socket a parte)	la riga non è una cornice per i byte
un oggetto Java	testo strutturato (il protocollo, o JSON); `ObjectStream` solo fra parti dello stesso programma	formato indipendente dalla classe, sicuro
un segnale breve, ripetibile, a chi non si conosce	un datagramma UDP, anche in broadcast	nessuna connessione, nessuna attesa
una risorsa del web	`HttpClient`; un `ServerSocket` per una pagina di diagnostica	HTTP è testo, ma ha le sue regole

Esercizi

Esercizio 6.1: Contare i byte (⋆)

Un client esegue `out.writeUTF("METTI")`, `out.writeUTF("Marco")`, `out.writeUTF("gatto.png")` e poi `Cornice.scrivi(out, dati)` con un array di 300000 byte. Quanti byte passano sul filo in tutto? E se il nome del file fosse "`perché.png`"?

Esercizio 6.2: Dopo un `readLine` (⋆)

Sullo stesso socket un server fa `in.readLine()` con un `BufferedReader` e poi `dati.readInt()` con un `DataInputStream` creato sullo stesso stream. Il client ha mandato la riga `FOTO gatto.png`, poi una cornice da 300000 byte. Dire che cosa legge `readInt`, perché, e che cosa succede dopo.

Esercizio 6.3: Due risposte (⋆)

Sulla rete ci sono due server di WhatsPAJC. Il client chiama `cerca(1500)`. Dire che cosa riceve, che cosa fa con la seconda risposta, e che cosa succede se la prima risposta si perde e la seconda no. Che cosa cambierebbe se `cerca` usasse TCP?

Esercizio 6.4: `GET /altro` (⋆)

Un browser apre `http://localhost:1236/altro`. Scrivere le righe esatte che il server manda, con i valori delle intestazioni, e dire che cosa mostra il browser. Poi: `curl -i http://localhost:1236/stato` con nessun utente collegato.

Esercizio 6.5: `send` nel listener (⋆⋆)

Un pulsante *Aggiorna* chiama `http.send(...)` nel suo listener e mette il corpo in un'etichetta. Dire che cosa succede alla finestra se il server risponde in tre secondi, e riscrivere il listener nelle due forme corrette: con un `SwingWorker` e con `sendAsync` (suggerimento: `thenAccept` gira su un thread del client HTTP, non sull'EDT).

I problemi che seguono chiedono un programma completo, in ordine crescente di difficoltà. Le soluzioni sono nel repository delle soluzioni del corso, nel progetto e nel package indicati.

Esercizio 6.6: Il contatore di byte (⋆)

Scrivere un server che riceve cornici (lunghezza davanti) e risponde a ciascuna con una riga `writeUTF` che dice quanti byte ha ricevuto e quanti in totale dalla connessione; una cornice di lunghezza zero chiude. Scrivere il client che manda tre cornici di dimensioni diverse (anche vuote, anche con dentro dei byte `10`, cioè degli a capo) e stampa le risposte; provare che una cornice con un a capo in mezzo arriva intera.

Soluzione: progetto `sol-d06-c06`, package `it.unibs.pajc.redolfi.e06`.

Esercizio 6.7: Il trasferimento con la barra (⋆⋆)

Scrivere un client con la finestra che manda un file al server degli allegati di WhatsPAJC mostrando l'avanzamento in una `JProgressBar`: la cornice va scritta a blocchi di 64 KB e dopo ogni blocco il `SwingWorker` chiama `publish` con i byte mandati; un pulsante *Annulla* interrompe il worker e chiude il socket. Il server non cambia.

Soluzione: progetto `sol-d06-c06`, package `it.unibs.pajc.redolfi.e07`.

Attenzione 6.2: La lunghezza davanti e i blocchi

La cornice dichiara la lunghezza totale *prima* dei byte: se il client annulla a metà, il server resta in `readFully` ad aspettare il resto finché la chiusura del socket non gli fa lanciare `EOFException`, ed è giusto così (la foto incompleta non va tenuta). Ciò che non va fatto è scrivere la lunghezza dopo aver contato i blocchi: il server deve saperla subito.

Esercizio 6.8: La connessione automatica (⋆⋆)

Aggiungere al dialogo di connessione del client un pulsante *Cerca* che chiama `Scoperta.cerca` in un `SwingWorker` e, se trova il server, riempie host e porta; se non lo trova, lo dice in un'etichetta. Poi far sì che il client, all'avvio, cerchi da solo per un secondo e, se trova il server, proponga direttamente il nome.

Soluzione: progetto `sol-d06-c06`, package `it.unibs.pajc.redolfi.e08`.

Esercizio 6.9: Un server di file (⋆⋆⋆)

Scrivere un server HTTP minimo che serve i file di una cartella: `GET /foto.png` risponde con il file `foto.png` della cartella, con il `Content-Type` dedotto dall'estensione (`html`, `png`, `jpg`, `txt`, `css`; altrimenti `application/octet-stream`) e il `Content-Length` esatto; `GET /` risponde con l'elenco dei file come pagina HTML con i link; un file che non esiste dà `404`; un percorso che tenta di uscire dalla cartella (`..`) dà `403`. Un thread virtuale per richiesta. Provarlo con il browser.
Suggerimento: `Files.readAllBytes` per il corpo; il percorso richiesto va normalizzato e confrontato con la cartella base prima di aprire qualunque cosa; i caratteri `%20` nel percorso vanno decodificati con `URLDecoder`.

Soluzione: progetto `sol-d06-c06`, package `it.unibs.pajc.redolfi.e09`.

Risposte

Risposta 6.1. `writeUTF` scrive due byte di lunghezza e poi i byte della stringa in UTF-8: `METTI` 2+5, `Marco` 2+5, `gatto.png` 2+9; la cornice 4+300000. In tutto 300029 byte. Con "`perché.png`" la `é` vale due byte in UTF-8, e il nome ne occupa 2+11: 300031. Il numero scritto nei due byte di `writeUTF` è quello dei byte, non dei caratteri, ed è il motivo del limite a 65535.

Risposta 6.2. Il `BufferedReader`, al primo `readLine`, ha letto dal socket un blocco di 8192 caratteri (il suo buffer predefinito), cioè la riga *e* i primi byte della cornice, già trasformati in caratteri. `readInt` legge quattro byte dal socket a partire da dove il buffer si era fermato: quattro byte presi a caso dentro la foto, interpretati come un intero qualunque, probabilmente enorme o negativo. `Cornice.leggi` lo rifiuta (`cornice non valida`); senza il controllo, `readFully` aspetterebbe miliardi di byte che non arrivano, o fallirebbe l'allocazione. I byte finiti nel buffer del `BufferedReader` sono persi per il `DataInputStream`: non c'è modo di restituirli al socket.

Risposta 6.3. Entrambi i server ricevono il broadcast e rispondono; il client riceve il primo datagramma arrivato, con l'indirizzo di quel server, e ritorna. Il secondo datagramma resta nel buffer del socket UDP, che viene chiuso dal try-with-resources: scartato. Se la prima risposta si perde, arriva solo la seconda e il client si collega al secondo server: per lui è indifferente. Con TCP il client dovrebbe sapere a chi collegarsi, cioè l'indirizzo che sta cercando: la scoperta è esattamente il caso in cui una connessione non può esistere.

Risposta 6.4. La riga di stato `HTTP/1.1 404 Not Found`; le intestazioni `Content-Type` con `text/plain; charset=utf-8`, `Content-Length` con `16` ("`niente in /altro`" sono sedici byte) e `Connection` con `close`; la riga vuota; il corpo `niente in /altro`. Il browser mostra il testo semplice, e nella barra degli strumenti di sviluppo lo stato 404. Con `curl -i` sulla pagina di stato senza utenti: `HTTP/1.1 200 OK`, il tipo `text/html`, la lunghezza del corpo, e il corpo `<h1>WhatsPAJC</h1>` seguito da `<p>Collegati: 0</p>` e da una lista vuota `<ul></ul>`.

Risposta 6.5. La finestra congela per tre secondi: il listener gira sull'EDT e `send` non ritorna finché il corpo non è arrivato. Con il worker: `doInBackground` fa il `send` e restituisce il corpo, `done` lo mette nell'etichetta. Con `sendAsync`: il `CompletableFuture` restituito si completa con la risposta, e con `thenAccept` gli si dà una lambda che la riceve; dentro la lambda la `setText` va incartata in un `invokeLater`, perché `thenAccept` viene eseguito sul thread che ha completato la richiesta, uno del client HTTP, e un componente toccato da lì è l'errore del capitolo 5.

Capitolo 7

Palestra

Questo capitolo non introduce niente di nuovo: mette alla prova ciò che i capitoli precedenti hanno costruito, con otto programmi completi in ordine crescente di difficoltà. Ogni problema dice che cosa allena, avverte dell'errore in cui si cade di solito, e rimanda alla soluzione nel repository del corso. L'ultimo è WhatsPAJC per intero, il programma del capitolo 1, con tutto ciò che nel frattempo ha imparato a fare. Le soluzioni sono progetti che si avviano: per ciascuno vale la pena provare in due, su due computer, perché è l'unico modo di vedere la rete al lavoro.

Esercizio 7.1: Il server dell'ora, due versioni (⋆)

Scrivere il server dell'ora dei temi d'esame nelle due versioni richieste: (a) resta in ascolto sulla porta 1234 e, a ogni client che si collega, manda l'ora corrente, chiude e torna ad aspettare, un client alla volta; (b) la stessa cosa con un thread per client, così che dieci client collegati insieme siano serviti insieme. Scrivere poi un client che si collega dieci volte di seguito, stampa le dieci ore ricevute e il tempo totale impiegato, e usarlo per confrontare le due versioni con un server che, prima di rispondere, aspetti mezzo secondo (`Thread.sleep`) per simulare un lavoro.
Che cosa allena: `ServerSocket` e `accept`, il try-with-resources al posto giusto, un thread (virtuale) per client, la misura della differenza.

Soluzione: progetto `sol-d06-c07`, package `it.unibs.pajc.redolfi.e01`.

Attenzione 7.1: Il `ServerSocket` chiuso con il primo client

Nella versione (a) il `ServerSocket` è nel ciclo e il `Socket` nel try-with-resources interno: metterli insieme chiude la porta dopo il primo client. Nella versione (b) il tempo totale del client deve scendere da cinque secondi a mezzo: se non scende, le sessioni non girano insieme (un pool troppo piccolo, o un `synchronized` di troppo).

Esercizio 7.2: Eco a rovescio con scadenza (⋆)

Scrivere un server per molti client che risponde a ogni riga con la riga rovesciata (`ciao` diventa `oaic`) e che saluta e chiude un client che non scrive niente per dieci secondi, con la riga `troppo lento, ciao`; il server stampa nel log ogni ingresso e ogni uscita con il motivo (`fine`, chiusura del client, scadenza, connessione persa). Scrivere il client di console con il lettore su un thread, che mostra il saluto di chiusura e termina da solo quando il server chiude.
Che cosa allena: il timeout di lettura (`setSoTimeout`) e la sua eccezione, i quattro modi in cui una sessione finisce e il `finally` che li raccoglie, il client full-duplex che si accorge della chiusura.

Soluzione: progetto `sol-d06-c07`, package `it.unibs.pajc.redolfi.e02`.

Attenzione 7.2: Il timeout che chiude senza salutare

`SocketTimeoutException` è una `IOException`: un `catch (IOException)` generico la tratta come una connessione persa e chiude senza la riga di saluto. Il `catch` del timeout va prima, e dentro si manda la riga; e il client, dopo averla stampata, deve vedere `null` e uscire.

Esercizio 7.3: La calcolatrice remota completa (⋆⋆)

Mettere insieme le tre calcolatrici dei capitoli precedenti in un progetto solo: il protocollo come tabella di azioni con la classe `Protocollo` e i suoi test JUnit (esercizio 3.6), il server per molti client con una sessione per connessione e un contatore delle operazioni eseguite in tutto, stampato nel log a ogni richiesta (uno stato condiviso: va protetto), e il client con la finestra con lo storico (esercizio 5.7). Aggiungere al protocollo `STAT`, a cui il server risponde con il numero di operazioni eseguite e di client collegati.
Che cosa allena: un protocollo in un posto solo, usato da server e client; lo stato condiviso fra sessioni, con `LongAdder` o `AtomicLong`; un client Swing a passo con la coda delle richieste.

Soluzione: progetto `sol-d06-c07`, package `it.unibs.pajc.redolfi.e03`.

Attenzione 7.3: Il contatore che perde

`operazioni++` in un campo `static int` toccato dalle sessioni perde incrementi, e il numero stampato è sbagliato di poco e di rado: esattamente il tipo di errore che non si vede provando. `AtomicLong` o `LongAdder`, e il valore letto è giusto.

Esercizio 7.4: Il quiz a più giocatori (⋆⋆)

Scrivere il quiz del capitolo 3 per molti giocatori: il server fa la stessa domanda a tutti i collegati (`DOMANDA n testo`), il primo che risponde giusto prende il punto e tutti ricevono `PUNTO nome`; dopo la risposta giusta, o dopo venti secondi, il server passa alla domanda seguente; `/classifica` mostra i punti di tutti; chi entra a quiz iniziato riceve la domanda corrente. Il quiz parte quando l'amministratore scrive `via` sulla console del server e finisce con l'ultima domanda, con la classifica finale a tutti. Client di console. *Che cosa allena:* un server con uno stato globale (la domanda corrente, i punti) modificato da molte sessioni e da un timer, le sezioni critiche ("il primo che risponde" è un check-then-act), un messaggio a tutti da un thread che non è una sessione.

Soluzione: progetto `sol-d06-c07`, package `it.unibs.pajc.redolfi.e04`.

Attenzione 7.4: Due primi

Due giocatori rispondono giusto nello stesso istante: senza una sezione critica attorno a "la domanda è ancora aperta? allora il punto è tuo e la chiudo" entrambi prendono il punto. Il controllo e la chiusura della domanda vanno fatti insieme, sotto lo stesso lock, oppure con un `compareAndSet` su un `AtomicInteger` che tiene il numero della domanda aperta.

Esercizio 7.5: Il trasferimento di file (⋆⋆)

Scrivere il trasferimento di un file fra due client passando dal server: un client manda `/invia nomeutente percorso`, il server avvisa il destinatario con `FILE mittente nome dimensione`, e il destinatario, se accetta con `/accetta`, lo riceve sul suo disco in una cartella `ricevuti/`. I byte passano su un secondo socket a cornici, come gli allegati del capitolo 6, e il server li tiene in memoria solo finché il destinatario non li ha presi (o per cinque minuti, poi li scarta). Client con la finestra: una `JProgressBar` per l'invio e una per la ricezione, in due `SwingWorker`.
Che cosa allena: testo e byte su due socket, le cornici a blocchi con l'avanzamento, un server che tiene dati con una scadenza (un executor a tempo, `ScheduledExecutorService`, li scarta), il worker con `publish`.

Soluzione: progetto `sol-d06-c07`, package `it.unibs.pajc.redolfi.e05`.

Attenzione 7.5: Il nome del file che arriva dalla rete

Il nome che il destinatario salva viene dal mittente: `../../.bashrc` è un nome valido per `Path.of`. Si tiene solo `getFileName()` e si scrive dentro la cartella scelta; e se il file esiste già non lo si sovrascrive in silenzio.

Esercizio 7.6: Il tris in rete completo (⋆⋆⋆)

Mettere insieme il server arbitro (esercizio 4.9) e il client con la finestra (esercizio 5.9), e aggiungere gli *spettatori*: chi si collega con `/guarda` invece di giocare vede l'elenco delle partite in corso (`PARTITE Marco-Giulia,Sara-Luca`), ne sceglie una con `/guarda Marco-Giulia` e riceve tutte le mosse di quella partita, con la finestra in sola lettura; a partita finita riceve l'esito e torna all'elenco. Il server tiene le partite in un registro e manda ogni mossa ai due giocatori e a tutti gli spettatori di quella partita.
Che cosa allena: un protocollo con due ruoli, lo stato condiviso fra più di due sessioni (la partita con i suoi spettatori), l'elenco che cambia mentre lo si scorre, un Model Swing con più fasi.

Soluzione: progetto `sol-d06-c07`, package `it.unibs.pajc.redolfi.e06`.

Attenzione 7.6: Lo spettatore che arriva a metà

Chi comincia a guardare una partita già iniziata non ha visto le mosse precedenti: il server deve mandargli lo stato corrente (le mosse fatte, o la griglia intera), sotto il lock della partita, prima di aggiungerlo agli spettatori; altrimenti una mossa fatta nel frattempo gli arriva prima dello stato, o mai.

Esercizio 7.7: Il sondaggio in aula (⋆⋆⋆)

Scrivere un sistema di sondaggi per l'aula: il docente avvia il server e, dalla sua finestra, apre una domanda con due o più risposte; i client (una finestra con i pulsanti delle risposte) trovano il server da soli con la scoperta UDP del capitolo 6, ricevono la domanda appena aperta, votano una volta sola (il server rifiuta il secondo voto dello stesso nome) e vedono l'istogramma dei voti aggiornarsi dal vivo, insieme a quello del docente, a ogni voto che arriva; il docente chiude la domanda e ne apre un'altra. Il protocollo è da progettare; l'istogramma è un pannello che si disegna dal Model (*Interfacce grafiche con Swing*).
Che cosa allena: UDP per scoprire e TCP per votare, un server con due ruoli (docente e studenti), lo stato condiviso con i conteggi, il messaggio a tutti a ogni voto, un Model Swing che si aggiorna da righe in arrivo.

Soluzione: progetto `sol-d06-c07`, package `it.unibs.pajc.redolfi.e07`.

Attenzione 7.7: Il voto doppio

"Ha già votato?" seguito da "allora conto il voto" è il solito check-then-act, e due click veloci dello stesso client arrivano su due righe: un `ConcurrentHashMap` da nome a risposta con `putIfAbsent` risolve il conteggio e il rifiuto in una mossa sola.

Esercizio 7.8: WhatsPAJC completa (⋆⋆⋆)

Chiudere il problema d'apertura: WhatsPAJC con tutto. Il server: il protocollo del capitolo 3 con le stanze (esercizio 4.7) e il `/ping` (esercizio 4.8), le sessioni su thread virtuali, il registro, l'arresto ordinato da console, gli allegati, la pagina di stato e la scoperta UDP del capitolo 6. Il client con la finestra: le conversazioni separate con i non letti (esercizio 5.8), le stanze in una tendina, il pulsante per mandare una foto e le foto ricevute che si aprono in un dialogo, il dialogo di connessione con *Cerca* (esercizio 6.8), la riconnessione dopo una caduta del server senza chiudere la finestra. Il tutto si prova in aula, con il server su un computer e i client sugli altri.
Che cosa allena: tutto. In particolare la tenuta dell'architettura quando le funzioni crescono: il protocollo in un posto solo, le sessioni che non si conoscono, il Model che non sa di socket, e la domanda, a ogni funzione nuova, "su quale thread gira questa riga?".

Soluzione: progetto `sol-d06-c07`, package `it.unibs.pajc.redolfi.e08`.

Attenzione 7.8: La funzione che bypassa l'architettura

L'errore tipico dell'ultimo passo non è tecnico: è la foto mandata direttamente dal listener del pulsante, o la stanza tenuta nella View, perché "è solo una cosa in più". Ogni funzione nuova passa dallo stesso percorso (una riga del protocollo, una riga nella tabella, un metodo della sessione, una proprietà del Model), e se non ci passa, prima o poi la finestra congela o due thread si incrociano.

Capitolo 8

Dove andare adesso

La dispensa si ferma a un server che serve qualche migliaio di client con un thread virtuale ciascuno, a un protocollo di testo scritto a mano e a un client Swing. È abbastanza per capire ogni programma di rete che si incontrerà, e per scriverne di propri; ciò che segue sono le strade che partono da qui, con un assaggio ciascuna.

8.1 Canali e `Selector`: NIO

Prima dei thread virtuali, un server con diecimila connessioni non poteva avere diecimila thread, e Java ha offerto un altro modello: `java.nio.channels`, con i canali *non bloccanti* e un `Selector` che dice quali canali hanno qualcosa da leggere. Un solo thread scorre i canali pronti, legge ciò che c'è, e torna ad aspettare: nessun `readLine` bloccante, ma il programma deve tenere da sé lo stato di ogni connessione (la riga a metà, il comando in corso), ed è per questo che il codice NIO è più difficile da scrivere e da leggere. I thread virtuali riportano il modello bloccante, che è quello della dispensa, alla scala di NIO: conviene conoscerlo, perché le librerie che seguono ci sono costruite sopra, e usarlo direttamente solo con un motivo preciso.

```
Selector selector = Selector.open();
ServerSocketChannel server = ServerSocketChannel.open().bind(new
    InetSocketAddress(1234));
server.configureBlocking(false);
server.register(selector, SelectionKey.OP_ACCEPT);
while (true) {
    selector.select();                              // aspetta che qualcosa sia
    pronto
    for (SelectionKey k : selector.selectedKeys()) { /* accept o read, senza
    bloccare */ }
    selector.selectedKeys().clear();
}
```

8.2 Connessioni cifrate: TLS

Tutto ciò che WhatsPAJC manda passa in chiaro: chiunque sulla stessa rete può leggerlo. *TLS* (quello della `s` di `https`) cifra il filo e garantisce l'identità del server con un certificato. In Java è un socket come gli altri, creato da una fabbrica:

```
Socket s = SSLSocketFactory.getDefault().createSocket("www.unibs.it", 443);
```

restituisce un `Socket` i cui stream cifrano e decifrano da soli, e il resto del programma non cambia. La parte difficile non è il codice, è il certificato: il server ne ha bisogno di uno firmato

da un'autorità che il client riconosca (`keytool` per crearne uno di prova, Let's Encrypt per uno vero), e lo carica con un `SSLContext`. `HttpClient` fa tutto da sé per `https`.

8.3 Server HTTP e WebSocket

La pagina di stato del capitolo 6 è un server HTTP, ma un sito vero ha bisogno di molto altro: percorsi, metodi, intestazioni, sessioni, compressione, cifratura. Il JDK ha un server HTTP piccolo e completo, `com.sun.net.httpserver.HttpServer`, con cui la pagina di stato si scrive in dieci righe senza `\r\n`; per un'applicazione web si usano Jetty o Tomcat, o i framework che li incorporano (Spring Boot, Javalin). E quando il browser deve ricevere messaggi *spinti* dal server, come la nostra chat, il protocollo è *WebSocket*: una connessione che nasce come richiesta HTTP e poi diventa un filo bidirezionale, con messaggi di testo o binari già incorniciati. `HttpClient` sa aprirne uno (`newWebSocketBuilder()`), e un client di WhatsPAJC per il browser sarebbe un server WebSocket che traduce le righe del nostro protocollo.

8.4 JSON e Protocol Buffers

Il protocollo di WhatsPAJC è un formato di testo scritto a mano, e per un progetto piccolo è la scelta giusta. Fra programmi di organizzazioni diverse il formato di testo è JSON, e le librerie che traducono record e classi da e verso JSON (Jackson, Gson, Moshi) tolgono di mezzo ogni `split`: un metodo legge un oggetto da una stringa e uno lo scrive, con un campo `"tipo"` al posto della prima parola della riga. Quando contano la dimensione e la velocità (telefoni, sensori, servizi che si scambiano milioni di messaggi) si passa a un formato binario con uno schema, come Protocol Buffers: si scrive lo schema in un file, un compilatore genera le classi, e la cornice con la lunghezza davanti è di nuovo quella del capitolo 6.

8.5 Chiamare un metodo a distanza: RMI e gRPC

Un'altra idea, vecchia quanto le reti: invece di righe, *chiamate di metodo* su un oggetto che sta su un'altra macchina. Java la offre da sempre con RMI (`java.rmi`): un'interfaccia, un oggetto esportato, e il client chiama `server.saluta("Marco")` come se fosse locale, con la serializzazione sotto. Il modello moderno è gRPC: lo schema dei messaggi e dei metodi in un file, le classi generate per molti linguaggi, il trasporto su HTTP/2. In entrambi i casi la rete è nascosta, ed è la ragione per cui conviene averla vista prima: una chiamata che può durare un secondo, fallire o non tornare non è una chiamata come le altre, e chi lo sa scrive codice che lo prevede.

8.6 Giochi in rete: la Dama e il torneo

Il tris del capitolo 7 è lo schema di ogni gioco a turni in rete: un server arbitro che tiene lo stato, due client che mandano mosse, un protocollo con il turno. La Dama è lo stesso schema con un modello più ricco (le prese multiple, la promozione), e un client che può anche essere un *programma*: un bot che riceve `TOCCA` e risponde con la mossa scelta da una ricerca minimax a poche mosse di profondità. Da lì a un torneo di bot il passo è breve: il server accoppia i bot, conta le vittorie, pubblica la classifica sulla pagina di stato. È un progetto che usa ogni capitolo di questa dispensa e quasi ogni capitolo delle altre.

8.7 Nelle altre dispense

Thread e concorrenza è la dispensa gemella di questa: i thread virtuali, gli executor, lo stato condiviso, l'EDT e `SwingWorker` sono spiegati lì per esteso, e qui usati. *Interfacce grafiche con Swing* per la finestra del client e i pannelli che si disegnano; *Pattern in gioco* per il Model che avvisa, MVC e gli eventi; *Le interfacce in Java moderno* per le gerarchie `sealed` e lo `switch` esaustivo, l'altra forma possibile del protocollo; *Java Fast Forward* per gli stream di byte e di caratteri, i record e i file; *Dal nome alla funzione* per le lambda che sono i compiti dei thread e i listener.

8.8 Riferimenti

- Il Javadoc di Java 21: il package `java.net` (`Socket`, `ServerSocket`, `DatagramSocket`, `InetAddress`, `URI`), il package `java.net.http` con `HttpClient`, e in `java.io` le classi `DataInputStream`, `ObjectInputStream` e `ObjectInputFilter`.
- JEP 444, *Virtual Threads*: il perché e il come dei thread virtuali, con la sezione sul pinning.
- J. Kurose, K. Ross, *Computer Networking: A Top-Down Approach*: il livello di applicazione e quello di trasporto, con TCP e UDP spiegati dal punto di vista di chi scrive programmi.
- W. R. Stevens, *UNIX Network Programming, vol. 1*: il modello dei socket da cui discendono quelli di Java e di ogni altro linguaggio.
- RFC 9293 (TCP), RFC 768 (UDP), RFC 9110 (semantica di HTTP): i protocolli nelle parole di chi li ha definiti, più leggibili di quanto si pensi.
- J. Bloch, *Effective Java*, item 85 e 86: perché preferire alternative alla serializzazione di Java, e come limitare i danni quando serve.
- B. Goetz et al., *Java Concurrency in Practice*: lo stato condiviso fra le sessioni e il confinamento su un thread, che qui sono il registro e il Model.