

Appunti di Progettazione del Software

Canale 1 · Prof. Paolo Liberatore

Capitolo 10: Connessioni di rete

Autore: Tommaso C.

Appunti tratti dalle lezioni in aula e dal materiale del corso

Argomenti: Socket e porte · Flussi del socket · Protocolli SMTP e HTTP · Server e accept · Un server web

Indice

1	Socket e porte	1
2	I flussi del socket	1
3	Un protocollo: SMTP	3
4	Un altro protocollo: HTTP	4
5	Il server	6
5.1	Un client e un server	7
6	Esercizi	9
7	Soluzioni	10

1 Socket e porte

Definizione – Indirizzo e porta

Un computer in rete si raggiunge con il suo **nome**, come `smtp.gmail.com`, o con il suo indirizzo. La **porta** è un numero che sceglie il servizio dentro quel computer: lo stesso computer può offrire pagine web, posta e altri servizi, ognuno su una porta diversa.

Porta	Servizio	Protocollo
25	invio della posta	SMTP
80	pagine web	HTTP
443	pagine web cifrate	HTTPS

Definizione – Socket

Un **socket** (classe `Socket`, pacchetto `java.net`) è un canale di comunicazione fra il programma e un altro programma in rete. `new Socket(nome, porta)` apre la connessione:

```
Socket s = new Socket("smtp.gmail.com", 25);
```

Aprire la connessione può fallire in due modi, e in entrambi `new Socket` genera un'eccezione:

- il nome non corrisponde a nessun computer: `UnknownHostException`;
- nessun programma ascolta su quella porta: `ConnectException`, con il messaggio *Connection refused*.

Sono entrambe `IOException`, come gli errori sui file del Capitolo 9, quindi basta `throws IOException` nell'intestazione del `main`. Alla fine si chiama `s.close()`.

2 I flussi del socket

Proposizione – Due flussi per ogni socket

Un socket ha due flussi di byte, uno per direzione:

- `s.getInputStream()` restituisce un `InputStream` con i dati che arrivano dall'altro programma;
- `s.getOutputStream()` restituisce un `OutputStream` su cui il programma scrive i dati da mandare all'altro.

I flussi del socket danno byte, come il `FileInputStream` del Capitolo 9. Per leggere e scrivere righe di testo si costruisce sopra ciascuno una catena di flussi.

Definizione – `InputStreamReader`

`new InputStreamReader(is)` trasforma il flusso di byte `is` in un flusso di caratteri: è un `Reader`, come `FileReader`, e si può passare a un `BufferedReader`.

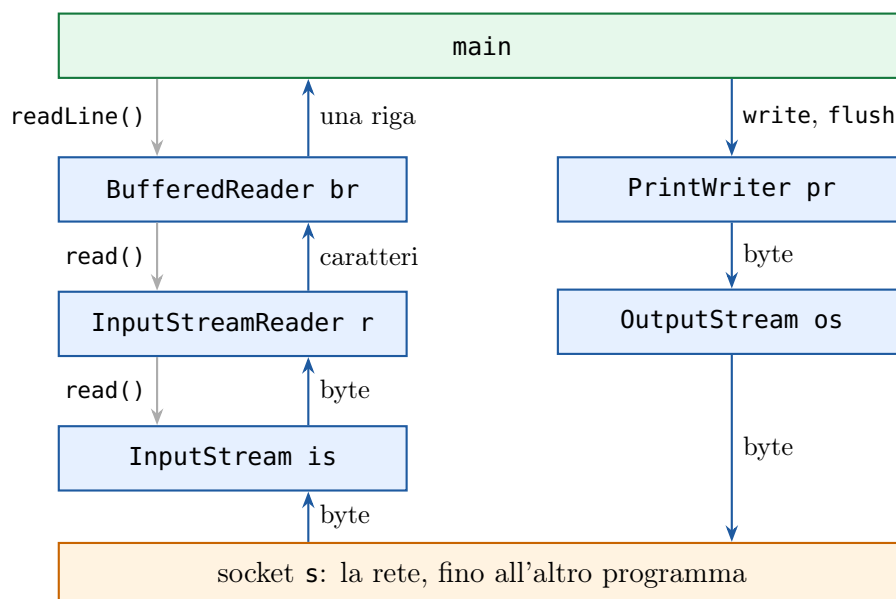
Un carattere non è sempre un byte: i byte si trasformano in caratteri secondo una **codifica**. `InputStreamReader` usa quella predefinita, UTF-8, in cui un carattere occupa da uno a quattro byte.

Proposizione – Aprire i flussi di un socket

1. In ingresso, il flusso di byte `is` si passa a un `InputStreamReader`, che dà caratteri, e questo a un `BufferedReader` `br`, che dà righe.
2. In uscita: `new PrintWriter(s.getOutputStream())`, che si chiama `pr`.
3. Si legge una riga con `br.readLine()`.
4. Si scrive una riga con `pr.write(linea + "\r\n")`, seguita da `pr.flush()`.

```
InputStream is = s.getInputStream();
InputStreamReader r = new InputStreamReader(is);
BufferedReader br = new BufferedReader(r);

OutputStream os = s.getOutputStream();
PrintWriter pr = new PrintWriter(os);
```



La catena di ingresso è quella del Capitolo 9 con un'altra sorgente: `br` chiede caratteri a `r` finché non ha una riga, e `r` li ottiene decodificando i byte che arrivano dal socket. Al

posto del `BufferedReader` si può usare uno `Scanner` sul flusso del socket, come sul file ZIP del Capitolo 9.

Osservazione – Senza `flush` non parte niente

`PrintWriter` tiene i caratteri nel suo buffer, come `FileWriter` nel Capitolo 9. Senza `flush()` la riga non viene spedita: l'altro programma non risponde, e i due restano bloccati in `readLine()`, ognuno in attesa dell'altro.

Chiudere il socket chiude anche i suoi flussi. Il canale è **bidirezionale**: ogni lato legge quello che l'altro scrive.

3 Un protocollo: SMTP

Definizione – Protocollo

Un **protocollo** stabilisce che cosa i due programmi si mandano e in che ordine: chi scrive per primo, il formato delle righe, i comandi e le risposte. Ogni servizio ha il suo, e il programma deve seguirlo alla lettera.

SMTP (*Simple Mail Transfer Protocol*) è il protocollo con cui si invia la posta. Le sue regole che servono qui:

1. appena il client si connette, il server manda una riga di benvenuto;
2. ogni riga finisce con i due caratteri `\r\n`, ritorno a capo e nuova riga;
3. ogni comando comincia con una parola di quattro lettere: **HELO** per presentarsi, **QUIT** per chiudere;
4. a ogni comando il server risponde con una riga che comincia con un codice numerico.

Il programma `Lettura` si connette al server di posta di Gmail, legge il benvenuto, si presenta e chiude. Prima di ogni invio stampa la riga mandata preceduta da `>>`, per distinguerla da quelle ricevute:

```
public class Lettura {  
    public static void main(String[] args) throws IOException {  
        Socket s = new Socket("smtp.gmail.com", 25);  
        InputStream is = s.getInputStream();  
        InputStreamReader isr = new InputStreamReader(is);  
        BufferedReader br = new BufferedReader(isr);  
        PrintWriter pr = new PrintWriter(s.getOutputStream());  
  
        String linea = br.readLine();  
        System.out.println(linea);
```

```

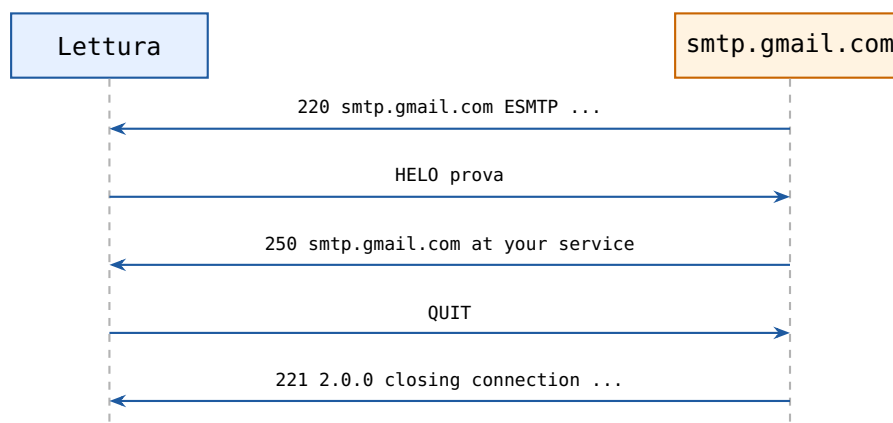
        linea = "HELO prova";
        System.out.println(">>> " + linea);
        pr.write(linea + "\r\n");
        pr.flush();
        linea = br.readLine();
        System.out.println(linea);

        linea = "QUIT";
        System.out.println(">>> " + linea);
        pr.write(linea + "\r\n");
        pr.flush();
        linea = br.readLine();
        System.out.println(linea);

        s.close();
    }
}

```

Il dialogo stampato ha la forma seguente; il codice numerico all'inizio di ogni risposta dice l'esito del comando.



Dopo **QUIT** il server chiude la connessione dal suo lato, e il programma dal suo con `s.close()`. Per spedire davvero un messaggio il protocollo prevede altri comandi dello stesso tipo, per mittente, destinatario e testo.

4 Un altro protocollo: HTTP

HTTP è il protocollo delle pagine web. Il programma resta lo stesso fino ai flussi: cambiano solo il nome del server, la porta e le righe scambiate.

Proposizione – Lo schema di un client

1. Si apre il socket con `new Socket(nome, porta)`.
2. Si creano il flusso di ingresso e quello di uscita.
3. Si segue il protocollo: `write` e `flush` per scrivere, `readLine` per leggere.
4. Si chiude il socket.

In HTTP scrive per primo il client: manda una richiesta di più righe, che finisce con una riga vuota, e il server risponde con più righe. Le righe ricevute si leggono con il ciclo del Capitolo 9, che si ferma quando `readLine()` restituisce `null`:

```
Socket s = new Socket("google.com", 80);
InputStream is = s.getInputStream();
InputStreamReader isr = new InputStreamReader(is);
BufferedReader br = new BufferedReader(isr);
PrintWriter pr = new PrintWriter(s.getOutputStream());

pr.write("GET / HTTP/1.1\r\n");
pr.write("Host: google.com\r\n");
pr.write("\r\n");
pr.flush();

while (true) {
    String linea = br.readLine();
    if (linea == null) {
        break;
    }
    System.out.println(linea);
}
s.close();
```

La risposta comincia con la riga **HTTP/1.1 301 Moved Permanently**: la pagina si è spostata, e la riga **Location** dice dove. Seguono altre righe di intestazione, una riga vuota e il documento.

Osservazione – Quando finisce la risposta

Dopo il documento il server non chiude la connessione, quindi `readLine()` non restituisce `null` e il ciclo resta in attesa. La fine si riconosce in due modi:

- la riga **Content-Length** dà la lunghezza del documento in byte, da leggere e non oltre;
- con la riga **Connection: close** nella richiesta, il server chiude dopo la risposta e il ciclo termina.

	SMTP	HTTP
Chi scrive per primo	il server	il client
Scambio	una riga per volta, avanti e indietro	una richiesta, una risposta
Richiesta	un comando per riga	più righe, poi una riga vuota
Risposta	una riga per comando	più righe, lette con un ciclo

5 Il server

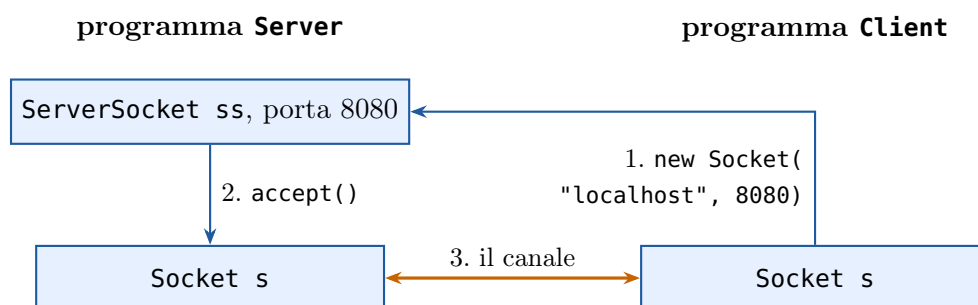
Definizione – Client e server

Il **client** è il programma che apre la connessione verso un altro computer. Il **server** è il programma che si mette in ascolto su una porta e aspetta che un client si connetta.

Proposizione – ServerSocket e accept

1. `new ServerSocket(8080)` dichiara che il programma ascolta sulla porta 8080. Da sola non riceve nessuna connessione.
2. `ss.accept()` blocca il programma finché un client non si connette, poi restituisce un `Socket` collegato a quel client.

Senza `accept()` il programma termina subito: si è messo in ascolto, ma non ha chiesto nessuna connessione. Con `accept()` resta fermo finché un client non si connette, e se nessuno lo fa va terminato a mano.



Dopo `accept()` i due lati sono uguali: ognuno ha il suo `Socket`, ne apre i flussi come nella Sezione 2 e segue il protocollo. Quello che uno scrive, l'altro lo legge.

5.1 Un client e un server

Nel protocollo dell'esempio il client manda una riga, e il server la stampa e la rimanda indietro due volte sulla stessa riga. Il server:

```
public class Server {
    public static void main(String[] args) throws IOException {
        ServerSocket ss = new ServerSocket(8080);
        System.out.println("attesa");
        Socket s = ss.accept();

        InputStream is = s.getInputStream();
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);
        PrintWriter pr = new PrintWriter(s.getOutputStream());

        String linea = br.readLine();
        System.out.println("server: " + linea);
        pr.write(linea + " " + linea + "\r\n");
        pr.flush();

        s.close();
        ss.close();
    }
}
```

Il client si connette a `localhost`, cioè al computer stesso, sulla porta del server:

```
public class Client {
    public static void main(String[] args) throws IOException {
        Socket s = new Socket("localhost", 8080);
        InputStream is = s.getInputStream();
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);
        PrintWriter pr = new PrintWriter(s.getOutputStream());

        String linea = "ciao";
        System.out.println("client: " + linea);
        pr.write(linea + "\r\n");
        pr.flush();
        linea = br.readLine();
        System.out.println("client: " + linea);

        s.close();
    }
}
```

I due programmi girano insieme, ognuno con la sua console. Va lanciato prima il server: se il client parte quando nessuno ascolta sulla porta, `new Socket` genera `ConnectException`. Il server stampa `attesa` e si ferma in `accept()`; quando parte il client, entrambi proseguono e terminano.

Osservazione – *Address already in use*

Su una porta ascolta un solo programma per volta. Se un server è ancora in esecuzione, per esempio fermo in `accept()`, un secondo `new ServerSocket(8080)` genera `BindException: Address already in use`: si termina il server vecchio e si rilancia.

Questo server gestisce una sola connessione e poi termina. Per servire un altro client dovrebbe tornare ad `accept()`, ma anche così servirebbe un client per volta: finché è occupato con uno, gli altri aspettano.

Riepilogo

- **Porta:** sceglie il servizio dentro un computer; 25 posta, 80 web.
- **Socket:** si apre con `new Socket(nome, porta)`. Se fallisce dà un'`IOException`: nome sconosciuto o porta senza server.
- **Ingresso:** `getInputStream()` dà i byte, un `InputStreamReader` li fa caratteri e un `BufferedReader` righe.
- **Uscita:** `PrintWriter` su `getOutputStream()`; si scrive con `write`, terminando la riga con `\r\n`, e poi `flush()`.
- **Protocollo:** chi scrive per primo, comandi e risposte. SMTP comincia dal server, HTTP dal client.
- **Fine della risposta:** `readLine()` dà null solo quando l'altro chiude la connessione.
- **Server:** `new ServerSocket(porta)`, poi `accept()` blocca e restituisce il `Socket` del client.
- **Simmetria:** dopo `accept()` i due lati usano il socket allo stesso modo; si lancia prima il server.

6 Esercizi

Esercizio 1 – Salvare la risposta

Scrivere un client HTTP che chieda la pagina / a **google.com**, sulla porta 80, e scriva le righe della risposta nel file **risposta.txt** invece di stamparle. Il programma deve terminare da solo, alla fine della risposta.

Soluzione a pagina 10 →

Esercizio 2 – Un server web

Scrivere un server sulla porta 8080 che, quando un browser si connette:

1. legga e stampi le righe della richiesta, fino alla riga vuota compresa;
2. mandi la risposta formata dalle righe **HTTP/1.0 200 OK, Content-Type: text/html**, una riga vuota e **<h1>Ciao dal server</h1>**;
3. chiuda la connessione e termini.

Provarlo aprendo nel browser l'indirizzo **http://localhost:8080**.

Soluzione a pagina 10 →

7 Soluzioni

Soluzione dell'Esercizio 1 – Salvare la risposta

Perché il ciclo termini, la richiesta contiene la riga **Connection: close** (Sezione 4). Il resto è il client della Sezione 4, con un **FileWriter** (Capitolo 9) al posto delle stampe:

```
Socket s = new Socket("google.com", 80);
InputStream is = s.getInputStream();
InputStreamReader isr = new InputStreamReader(is);
BufferedReader br = new BufferedReader(isr);
PrintWriter pr = new PrintWriter(s.getOutputStream());

pr.write("GET / HTTP/1.1\r\n");
pr.write("Host: google.com\r\n");
pr.write("Connection: close\r\n");
pr.write("\r\n");
pr.flush();

FileWriter fw = new FileWriter("risposta.txt");
while (true) {
    String linea = br.readLine();
    if (linea == null) {
        break;
    }
    fw.write(linea + "\n");
}
fw.close();
s.close();
```

`readLine()` toglie il terminatore di riga, quindi `"\n"` va rimesso a ogni `write`. Il `main` dichiara `throws IOException`, che copre sia il socket sia il file.

[← Torna all'Esercizio 1, pagina 9](#)

Soluzione dell'Esercizio 2 – Un server web

Il server è quello della Sezione 5, con un altro protocollo. La riga si stampa prima del controllo, quindi il ciclo è un **do-while**, che esegue il corpo almeno una volta:

```
ServerSocket ss = new ServerSocket(8080);
Socket s = ss.accept();
InputStream is = s.getInputStream();
InputStreamReader isr = new InputStreamReader(is);
```

```
BufferedReader br = new BufferedReader(isr);
PrintWriter pr = new PrintWriter(s.getOutputStream());

String linea;
do {
    linea = br.readLine();
    System.out.println(linea);
} while (!linea.equals(""));

pr.write("HTTP/1.0 200 OK\r\n");
pr.write("Content-Type: text/html\r\n");
pr.write("\r\n");
pr.write("<h1>Ciao dal server</h1>\r\n");
pr.flush();

s.close();
ss.close();
```

La riga vuota si riconosce con **equals**, non con **==**, perché le stringhe sono oggetti (Capitolo 6). La prima riga stampata è quella della richiesta, per esempio **GET / HTTP/1.1**. Chiudendo la connessione il server segnala al browser che il documento è finito. Dopo una richiesta il server termina: ricaricando la pagina, il browser non trova più nessuno sulla porta 8080.

[← Torna all'Esercizio 2, pagina 9](#)