Azzolini Riccardo 2020-04

Socket

1 Socket

Come già detto, i programmi applicativi utilizzano i protocolli mediante apposite interfacce (API) fornite dal sistema operativo e dal software di rete. In generale, queste API non sono standardizzate, ma, fortunatamente, per i protocolli TCP e UDP, esiste uno standard de facto (disponibile nella stragrande maggioranza dei linguaggi di programmazione, tra cui, ad esempio, C e Java): l'interfaccia **socket**.

I socket sono stati inizialmente definiti nel 1983, e da allora si sono affermati come lo standard de facto per l'implementazione di applicazioni distribuite.

1.1 Interoperabilità

I socket sono disponibili su moltissime piattaforme (sistemi operativi e linguaggi di programmazione). Grazie a essi, si ottiene quindi l'**interoperabilità** dei programmi di rete: due applicazioni che usano i socket (con lo stesso protocollo di trasporto, ad esempio TCP) possono interagire anche se vengono eseguite su macchine con sistemi operativi diversi, e/o se sono scritte in linguaggi di programmazione diversi.

Viceversa, applicazioni che utilizzano protocolli (di trasporto, applicativi, ecc.) diversi *non* possono interagire. Ad esempio, un'applicazione che utilizza UDP non può interagire con un'applicazione che utilizza TCP. Infatti, un elemento fondamentale della nozione di protocollo è che le applicazioni comunicanti usino appunto lo stesso protocollo.

2 Identificazione di una macchina

Per poter comunicare con una macchina, bisogna avere un modo di identificarla univocamente, distinguendola da tutte le altre macchine nel mondo. Come già detto, questa funzione è svolta dagli **indirizzi IP**, che (nel caso di IPv4) sono numeri di 32 bit, indicati solitamente in una notazione "decimale punteggiata", come ad esempio `193.206.183.131`.

Siccome, in generale, un indirizzo numerico di questo tipo non è molto facili da ricordare, vi si può associare un nome simbolico, detto *nome di dominio* (un esempio è

`www.dista.uninsubria.it`). La corrispondenza tra indirizzi simbolici e nomi di dominio è mantenuta dal *DNS* (*Domain Name System*).

Dal punto di vista programmativo, in Java, un indirizzo IP può essere ottenuto da un nome di dominio attraverso il metodo statico `InetAddress.getByName`, che restituisce un oggetto di tipo `InetAddress`. Ad esempio, il programma seguente visualizza l'indirizzo IP corrispondente al nome di dominio passato come argomento a riga di comando:

```
import java.net.InetAddress;
import java.net.UnknownHostException;

public class WhoAmI {
    public static void main(String[] args) throws UnknownHostException {
        if (args.length != 1) {
            System.err.println("Usage: WhoAmI DomainName");
            System.exit(1);
        }

        InetAddress addr = InetAddress.getByName(args[0]);
        System.out.println(addr);
    }
}
```

Alcuni esempi di esecuzione di questo programma sono:

```
$ java WhoAmI google.com
google.com/216.58.208.174
$ java WhoAmI www.dista.uninsubria.it
www.dista.uninsubria.it/193.206.183.131
```

3 Client e server in Java

Ad alto livello, per stabilire una connessione tra client e server in Java:

1. Il server deve rimanere in ascolto di richieste di connessione, usando un oggetto speciale `ServerSocket`.
2. Il client cerca di stabilire una connessione con un server, attraverso un oggetto di tipo `Socket`.
3. Una volta effettuata la connessione, vi si costruiscono sopra degli stream di I/O, che permettono di trattarla come se si stesse leggendo da e scrivendo su un file. Le regole secondo le quali leggere/scrivere (ad esempio, cosa contengono le richieste del client e le risposte del server) sono dettate dai protocolli di livello più alto.

4 Indirizzo localhost

Quando si sviluppano applicazioni concorrenti, per fare delle prove, è spesso utile eseguire client e server sulla stessa macchina, la quale magari non è neanche connessa a una rete. Ciò è reso possibile dall'indirizzo IP di loopback, `127.0.0.1`, che è associato al nome simbolico *localhost*, e riferisce, appunto, alla propria macchina.

In Java, ci sono vari modi equivalenti per ottenere un'istanza di `InetAddress` relativa a localhost, tra cui:

```
InetAddress localhost = InetAddress.getByName(null);
InetAddress localhost = InetAddress.getByName("localhost");
InetAddress localhost = InetAddress.getByName("127.0.0.1");
```

5 Identificazione di un'applicazione su una macchina

Un indirizzo IP non è sufficiente per identificare un server in modo univoco, dato che su una sola macchina possono essere attivi molti server. Perciò, è necessario specificare anche la **porta** sulla quale è in ascolto l'applicazione con la quale si vuole comunicare. sulla quale è in ascolto il server.

Attenzione: Le porte da 0 a 1023 sono solitamente riservate dal sistema, quindi è opportuno usare numeri di porta superiori.

Per la gestione dell'indirizzamento in Java, il pacchetto `java.net` fornisce due classi, tra loro correlate:

- `InetAddress` rappresenta un indirizzo IP (v4 o v6), e ha due sottoclassi:
  - `Inet4Address`: indirizzo IPv4 (formato da 32 bit);
  - `Inet6Address`: indirizzo IPv6 (che è composto da 128 bit).
- `SocketAddress` è un indirizzo astratto di un socket, non associato ad alcun protocollo; in pratica, si usa la sua sottoclasse `InetSocketAddress`, che corrisponde all'indirizzo di un socket basato su IP (IP Socket Address), costituito da una coppia di indirizzo IP e numero di porta.

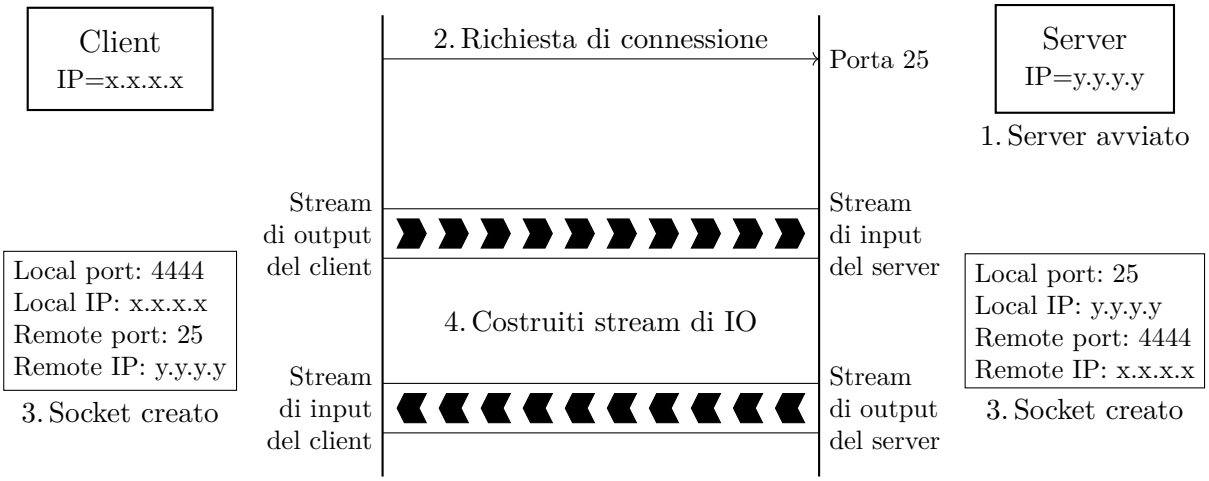
6 Concetto di socket

Letteralmente, il termine inglese *socket* significa "presa". Infatti, un socket rappresenta uno dei "terminali" di un collegamento tra due macchine.

Per una determinata connessione, c'è un socket su ogni macchina, e si può immaginare un ipotetico "cavo" avente le due estremità collegate a questi socket, attraverso il quale avviene la comunicazione, e che persiste finché non viene esplicitamente disconnesso.

6.1 Sequenza di operazioni

In seguito è riportata una rappresentazione schematica della sequenza di operazioni che tipicamente si svolgono quando due applicazioni comunicano tramite socket:



1. Su una macchina con indirizzo IP y.y.y.y viene avviato il server, che si mette in ascolto, ad esempio, sulla porta 25.
2. Il client, eseguito su una macchina con indirizzo x.x.x.x, invia una richiesta di connessione al server (cioè all'indirizzo y.y.y.y e alla porta 25).
3. Quando il server accetta la connessione, si creano (contemporaneamente) i socket a lato client e a lato server.
4. Client e server costruiscono degli stream di input e di output sopra ai rispettivi socket. L'input stream del server corrisponde all'output stream del client, e viceversa. Utilizzando questi stream, client e server possono comunicare "quanto vogliono", secondo le regole del protocollo applicativo.

Infine, al termine della comunicazione, la connessione verrà chiusa, e i socket verranno distrutti.

7 Uso dei socket in Java

Come già accennato, in Java i socket (stream-based, cioè basate su TCP) vengono usati mediante due principali classi:

- `ServerSocket`, che un server utilizza per ascoltare le connessioni in ingresso;
- `Socket`, utilizzata da un client al fine di avviare una connessione.

`ServerSocket` fornisce un metodo `accept()`, che sospende il chiamante in attesa di una richiesta di connessione, e restituisce un oggetto `Socket` nel momento in cui un client si connette. Da questo momento, si ha una connessione tra il socket del client e quello del server. Si possono allora usare i metodi `getInputStream()` e `getOutputStream()` di ciascun oggetto `Socket` per ottenere degli stream di I/O mediante i quali leggere/scrivere dati, senza più doversi preoccupare dei socket sottostanti.

Quando si crea un `ServerSocket`, è necessario specificare solo il numero di porta sul quale si accetteranno le connessioni (perché l'indirizzo IP è implicitamente quello della macchina su cui si esegue il server). Invece, nella creazione di un `Socket` (sul client), bisogna indicare sia l'indirizzo IP che il numero di porta del server al quale ci si vuole connettere.

7.1 Stream di I/O

Quello che viene trasmesso mediante i socket è una sequenza di byte. Perciò, i metodi `getInputStream()` e `getOutputStream()` restituiscono – come suggerito dai loro nomi – degli oggetti di tipo `InputStream` e `OutputStream` (che rappresentano, appunto, degli stream di byte).

Se si desidera trasmettere invece dati testuali, è possibile usare `InputStreamReader` e `OutputStreamWriter` per effettuare le conversioni tra byte e caratteri (ed eventualmente `BufferedReader`, `BufferedWriter`, `PrintWriter`, ecc. per una gestione più avanzata della lettura / scrittura).

8 Esempio: echo

Come primo semplice esempio di applicazione distribuita, si presenta un server che restituisce al client il testo ricevuto dal client stesso. Questo servizio è solitamente chiamato "echo" (per analogia con il fenomeno acustico dell'eco, che "ripete" ciò che viene detto).

8.1 Server

```
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;

public class EchoServer {
    private static final int PORT = 8080;

    public static void main(String[] args) throws IOException {
        ServerSocket server = new ServerSocket(PORT);
        try {
            System.out.println("Started " + server);
            Socket socket = server.accept();
            try {
                System.out.println("Connection accepted: " + socket);
                serveClient(socket);
            } finally {
                System.out.println("Closing...");
                socket.close();
            }
        } finally {
            server.close();
        }
    }

    private static void serveClient(Socket socket) throws IOException {
        try (
            BufferedReader in = new BufferedReader(
                new InputStreamReader(socket.getInputStream())
            );
            PrintWriter out = new PrintWriter(
                new BufferedWriter(
                    new OutputStreamWriter(socket.getOutputStream())
                ),
                true
            );
        ) {
            while (true) {
                String str = in.readLine();
                if (str.equals("END")) break;
                System.out.println("Echoing: " + str);
                out.println(str);
```

```
            }
        }
    }
}
```

1. Come numero di porta, si sceglie 8080.
2. Si crea un `ServerSocket` per ricevere connessioni su tale porta.
3. Con la chiamata `server.accept()`, il server si mette effettivamente in ascolto sulla porta, e attende una richiesta di connessione da parte di un client.
4. Quando arriva una richiesta di connessione, la `accept()` termina, restituendo un oggetto `Socket` connesso al client.
5. Si costruiscono due stream di I/O sul socket: rispettivamente, un `BufferedReader` per l'input e un `PrintWriter` per l'output.
6. Finché il client non invia un messaggio `"END"`, il server legge una riga di testo alla volta dallo stream di input e la riscrive sullo stream di output, rimandandola così al client. Grazie all'astrazione fornita dagli stream, queste operazioni possono essere scritte senza tenere conto del socket sottostante.
7. Una volta terminato il ciclo, è importante chiudere il `Socket` e (se, come in questo caso, non si vogliono accettare altre connessioni) il `ServerSocket`.

8.2 Client

```java
import java.io.*;
import java.net.InetAddress;
import java.net.Socket;

public class EchoClient {
    private static final int SERVER_PORT = 8080;

    public static void main(String[] args) throws IOException {
        InetAddress serverAddr = InetAddress.getByName(null);
        System.out.println("serverAddr = " + serverAddr);
        Socket socket = new Socket(serverAddr, SERVER_PORT);
        try {
            System.out.println("socket = " + socket);
            try (
                BufferedReader in = new BufferedReader(
                    new InputStreamReader(socket.getInputStream())
                );
                PrintWriter out = new PrintWriter(
```

```
                    new BufferedWriter(
                        new OutputStreamWriter(socket.getOutputStream())
                    ),
                    true
                )
            ) {
                for (int i = 0; i < 10; i++) {
                    out.println("hello " + i);
                    String str = in.readLine();
                    System.out.println(str);
                }
                out.println("END");
            }
        } finally {
            System.out.println("Closing...");
            socket.close();
        }
    }
}
```

1. Si ottiene un oggetto `InetAddress` che rappresenta l'indirizzo del server (in questo caso, è l'indirizzo localhost, perché client e server verranno eseguiti sulla stessa macchina).
2. Viene creato un socket, specificando l'indirizzo e il numero di porta del server.
3. Una volta connesso il socket, si costruiscono su di esso due stream di I/O, esattamente come sul server.

   Nota: L'argomento `true` passato al costruttore del `PrintWriter` attiva l'"auto-flush", cioè fa sì che venga eseguito un `flush()` dopo ogni chiamata `println`, in modo da assicurarsi che i dati scritti vengano immediatamente mandati al client (invece di rimanere nel buffer finché questo non si riempie).
4. Gli stream vengono utilizzati nel ciclo `for`, per inviare al server una serie di messaggi e ricevere le risposte.
5. Dopo il ciclo, si invia il messaggio `"END"` per indicare al server che la comunicazione è terminata.
6. Infine, viene chiuso il `Socket`.

Osservazione: Il `main` solleva una `IOException`: essa si verifica quando la connessione non va a buon fine, ad esempio perché il server non sta accettando connessioni.

8.3 Output

L'output su console del client è:

```
serverAddress = localhost/127.0.0.1
socket = Socket[addr=localhost/127.0.0.1,port=8080,localport=41992]
hello 0
hello 1
hello 2
hello 3
hello 4
hello 5
hello 6
hello 7
hello 8
hello 9
Closing...
```

Invece, il server visualizza in output:

```
Started ServerSocket[addr=0.0.0.0/0.0.0.0,localport=8080]
Connection accepted: Socket[addr=/127.0.0.1,port=41992,localport=8080]
Echoing: hello 0
Echoing: hello 1
Echoing: hello 2
Echoing: hello 3
Echoing: hello 4
Echoing: hello 5
Echoing: hello 6
Echoing: hello 7
Echoing: hello 8
Echoing: hello 9
Closing...
```

Osservazione: Mentre il server usa una porta predeterminata (scelta dallo sviluppatore), quella del client viene assegnata automaticamente al momento della connessione.

9 Esempio: data e ora

Come altro esempio, si realizza un server che, in un ciclo infinito, accetta connessioni dai client, una dopo l'altra, inviando a ciascuno la data e l'ora correnti.

Dal punto di vista di un client, esso chiama il server per ricevere data e ora, e poi la connessione viene terminata.

9.1 Server

```
import java.io.*;
import java.net.*;
import java.util.Date;

public class DateTimeServer {
    private static final int PORT = 1333;

    public static void main(String[] args) throws IOException {
        try (ServerSocket server = new ServerSocket(PORT)) {
            while (true) {
                try (
                    Socket connection = server.accept();
                    Writer out = new OutputStreamWriter(
                        connection.getOutputStream()
                    )
                ) {
                    Date now = new Date();
                    out.write(now + "\r\n");
                    out.flush();
                }
            }
        }
    }
}
```

Questo codice è in gran parte analogo a quello dell'esempio precedente. Le principali differenze, oltre al protocollo applicativo (cioè quali dati vengono scambiati usando il socket), sono:

- Si usa la porta 1333 invece della 8080 (a scopo illustrativo: la scelta è arbitraria).
- L'accettazione della connessione avviene in un ciclo infinito, quindi il server non termina dopo aver comunicato con un singolo client, ma rimane invece aperto ad altre richieste di connessione.
- Siccome il client non deve inviare dati al server (la richiesta di ricevere data e ora è rappresentata dalla connessione stessa), il server può costruire sul socket anche solo lo stream di output.

- Dopo la scrittura dei dati, viene eseguito manualmente un `flush()`, per assicurarsi che essi siano inviati al client prima della chiusura di una connessione (dato che qui, a differenza dell'esempio precedente, non si sta usando un `PrintWriter` con `flush()` automatico).

9.2 Client

```
import java.io.*;
import java.net.*;

public class DateTimeClient {
    private static final int SERVER_PORT = 1333;

    public static void main(String[] args) throws IOException {
        InetAddress serverAddr = InetAddress.getByName(null);
        System.out.println("serverAddr = " + serverAddr);
        try (
            Socket socket = new Socket(serverAddr, SERVER_PORT);
            BufferedReader in = new BufferedReader(
                new InputStreamReader(socket.getInputStream())
            )
        ) {
            System.out.println("socket = " + socket);
            String str = in.readLine();
            System.out.println(str);
            System.out.println("Closing...");
        }
    }
}
```

Anche il client rimane piuttosto simile all'esempio precedente. Così come il server ha bisogno solo di uno stream di output, per il client è sufficiente uno stream di input, sul quale riceve una singola riga di testo contenente la data e l'ora correnti.

Un esempio di output prodotto da un client è il seguente:

```
serverAddr = localhost/127.0.0.1
socket = Socket[addr=localhost/127.0.0.1,port=1333,localport=35382]
Mon Apr 27 13:01:55 CEST 2020
Closing...
```

10 Caratteristiche e svantaggi di TCP

Gli esempi presentati finora utilizzano socket *stream-based*, basati sul protocollo TCP. Come già visto, quest'ultimo è progettato per la massima affidabilità, e garantisce che i dati arrivino a destinazione.

Più nello specifico, TCP:

- ritrasmette i dati persi, sfruttando potenzialmente percorsi multipli, attraverso diversi router, nel caso in cui uno di questi diventi indisponibile;
- consegna i byte nell'ordine in cui sono stati inviati.

Il principale svantaggio è che, per garantire tale livello di controllo e affidabilità, TCP ha un notevole overhead. Allora, quando l'affidabilità non è necessaria, si possono usare invece socket basati su UDP, che non garantisce la consegna dei pacchetti né l'ordine in cui essi arriveranno, ma ha molto meno overhead.