Socket Java

In una rete di ampie dimensioni, ciascuna sottorete (es. LAN, WAN) è connessa ad altre sottoreti tramite router. Internet è un insieme di reti connesse tra loro.

Essenzialmente, in una rete alcune macchine possono fornire dei servizi ad altre macchine connesse. Spesso tali servizi si basano sul modello client/server.

Client: macchina o processo che richiede un servizio ad un'altra macchina o processo attraverso un ***protocollo***.

Server: fornisce un servizio al client.

Protocollo: un set di regole che descrive come trasmettere dati.

Il modello client/server consente a due processi di condividere risorse e di cooperare per il raggiungimento di un obiettivo.

La comunicazione dei sistemi che usano il modello client/server può avvenire attraverso le ***socket*** (presa o spinotto). Esse forniscono un'astrazione rispetto a meccanismi di più basso livello per la comunicazione in rete.

Una socket può essere rappresentata come il punto di accesso per il canale di comunicazione di due parti che risiedono su host diversi (o sullo stesso host).

Una socket in uso è solitamente legata ad un *indirizzo* (IP) ed ad una *porta*.

La libreria Java che fornisce il supporto per le socket è `java.net`

Il dominio Internet: indirizzi IP e porte

Un ***indirizzo IP*** (Internet Protocol) si può rappresentare, come stringa, con quattro interi a 8 bit (valori da 0 a 255) separati da punti (es. 151.97.253.200).

Esso consente di individuare univocamente la posizione di una sottorete e di un host al suo interno.

Per usare le socket, oltre a conoscere l'indirizzo IP dell'host a cui connettersi, bisogna disporre dell'informazione sufficiente per collegarsi al processo server corretto.

Per questo motivo esistono i ***numeri di porta*** (port number) che permettono di associare un servizio (un processo server che risponde alle richieste) ad un ben determinato numero.

Le connessioni avvengono sempre specificando un indirizzo IP ed un numero di porta.

I numeri di porta sono interi a 16 bit (da 0 a 65535).

Quelli da 0 a 1023 sono riservati per i servizi standard (es. FTP porta 21, HTTP porta 80), mentre i numeri da 1024 a 65535 sono disponibili per i processi utente.

Tipi di Socket

Esistono due modalità di comunicazione principali in rete:

- ***connectionless***
- ***connection oriented***

A tali modalità corrispondono rispettivamente i seguenti *tipi* di socket:

- ***Socket Datagram***: trasferiscono messaggi di dimensione variabile, preservando i confini, **ma** senza garantire ordine o arrivo dei pacchetti. Supportate nel dominio Internet dal protocollo UDP (*User Datagram Protocol*).

  Non instaurano una connessione tra client e server, e non verificano l'arrivo del dato o il ri-invio. Hanno il vantaggio di trasferire velocemente i dati.

  L'implementazione Java è la classe `DatagramSocket`.
  I dati sono incapsulati nella classe `DatagramPacket`.

- ***Socket stream***: forniscono stream di dati affidabili ed ordinati. Nel dominio Internet sono supportati dal protocollo TCP (*Transfer Control Protocol*).

  Permettono di creare una connessione bidirezionale tra client e server.

  L'implementazione Java è costituita dalle classi `Socket` e `ServerSocket`.

Datagram Socket

Usare le datagram socket consiste nell'implementazione di un programma che effettua i seguenti passi fondamentali:

Client:

(a) crea socket; (b) manda richiesta sulla socket con indirizzo, porta e messaggio; (c) riceve dati dalla socket; (d) chiude la socket.

Server:

(a) crea socket, per ascoltare richieste in arrivo; (b) riceve dati dalla socket; (c) invia dati sulla socket al client che ne ha fatto richiesta; (d) chiude la socket.

La classe Java `DatagramSocket` usa il protocollo UDP per trasmettere dati attraverso le socket. Essa permette ad un client di connettersi ad una determinata porta di un host per leggere e scrivere dati impacchettati attraverso la classe `DatagramPacket`.

Nota: l'indirizzo dell'host destinatario è nel `DatagramPacket`.

Metodi della classe `DatagramSocket`:

```
void DatagramSocket() throws SocketException
void DatagramSocket(int port) throws SocketException
void receive(DatagramPacket p) throws IOException
```
blocca il chiamante fino a quando arriva un pacchetto

```
void setSoTimeout(int timeout) throws SocketException
```
usando `setSoTimeout()`, il chiamante di `receive()` si blocca al più `timeout` millisec

```
void send(DatagramPacket p) throws IOException
void close()
```

Classe DatagramPacket

La classe Java con cui sono rappresentati i pacchetti UDP da inviare e ricevere sulle socket di tipo datagram è `DatagramPacket`.

Un oggetto istanza di `DatagramPacket` si costruisce inserendo nella chiamata al suo *costruttore* (**lato client**):

- il contenuto del messaggio (i primi `length` bytes dell'array `buf`)
- l'indirizzo IP del destinatario
- il numero di porta su cui il destinatario è in ascolto, quindi:

```
public DatagramPacket(byte buf[], int length,
                      InetAddress address, int port)
```

Se il pacchetto deve essere ricevuto (tipicamente lato server), basta definire una istanza di `DatagramPacket` per il contenuto (IP/porta saranno assegnati dal sistema):

```
public DatagramPacket(byte buf[], int length)
```

Metodi get/set della classe `DatagramPacket`:

`InetAddress getAddress()`
: restituisce l'indirizzo IP della macchina da cui il pacchetto sta per essere mandato o da cui è stato ricevuto

`void setAddress(InetAddress addr)`

`int getPort()`
: restituisce la porta della macchina remota a cui il pacchetto sta per essere mandato o da cui è stato ricevuto

`void setPort(int iport)`

`byte[] getData()` restituisce i dati del pacchetto

`void setData(byte[] buf)`

Classe InetAddress

La classe `InetAddress` serve a descrivere gli indirizzi Internet, astraendo dalla rappresentazione con byte, alfabetica o col nome.

Il metodo static `getByName()` restituisce un'istanza di `InetAddress` rappresentante l'host specificato dall'argomento, es. "*java.sun.com*":

```
static InetAddress getByName(String hostname)
```

Se viene passato il parametro `null`, viene restituita un'istanza di `InetAddress` rappresentante l'indirizzo di default dell'host locale.

Il metodo statico `getByAddress()` restituisce un'istanza `InetAddress` rappresentante l'host specificato come indirizzo IP.

```
public static InetAddress getByAddress(byte[] addr)
```

Il metodo static `getAllByName()` restituisce un array `InetAddress` utile in casi di più indirizzi IP registrati con lo stesso nome logico:

```
public static InetAddress[] getAllByName(String hostname)
```

Il metodo statico `getLocalHost()` restituisce un'istanza di `InetAddress` per la macchina locale; se questa non è registrata o è protetta da firewall, l'indirizzo è quello di loopback: 127.0.0.1

```
public static InetAddress getLocalHost()
```

Tutti i precedenti metodi possono sollevare l'eccezione `UnknownHostException` se l'indirizzo specificato non può essere risolto.

Codice

```
// esempio di uso di InetAddress con l'host locale
try {
    InetAddress host1 = InetAddress.getByName(null);
    // ovvero
    InetAddress host2 = InetAddress.getLocalHost();
    System.out.println("Host locale "+ host2.getHostName());
}
catch (UnknownHostException e) {
    System.out.println("Problemi con l'indirizzo locale");
    e.printStackTrace();
}
```

```
// esempio di creazione di socket datagram
try {
    // creazione socket datagram
    DatagramSocket dsocket = new DatagramSocket();

    byte[] buf = new byte[256];

    // preparazione pacchetto
    DatagramPacket packet = new DatagramPacket(buf, buf.length);

    // ricezione pacchetto tramite datagram
    dsocket.receive(packet);

    InetAddress mittAddr = packet.getAddress();
    System.out.println("Pacchetto ricevuto da " + mittAddr);
}
catch(Exception e) {
    System.out.println("Problemi: ");
    e.printStackTrace();
}
```

Esempio Datagram socket -1

Applicazione di esempio client-server.

Il *client* invia dei pacchetti con nome e riga del file richiesti al server. Attende la risposta e manda sullo schermo ciò che ha ricevuto.

Il *server*, a ogni richiesta, estrae dal file la riga richiesta e la invia al client. Se file o riga non esistono notifica l'errore al client.

Lato Client, frammenti significativi:

1. Creazione socket ed eventuale setting opzioni:

```
DatagramSocket socket = new DatagramSocket();
socket.setSoTimeout(30000);
```

2. Interazione con l'utente:

```
BufferedReader stdIn = new BufferedReader(
                       new InputStreamReader(System.in));
System.out.print("Numero linea: ");
String richiesta = stdIn.readLine();
```

3. Creazione del pacchetto di richiesta da mandare al server: usa `DatagramUtility.buildPacket` (4° arg. è `String`)

```
pack = DatagramUtility.buildPacket( // NB: classe ausiliaria
                  addr, LineServer.PORT, richiesta);
```

4. Invio del pacchetto al server:

```
socket.send(pack);
```

5. Attesa del pacchetto di risposta:

```
packetIN = new DatagramPacket(buf, buf.length);

socket.receive(packetIN);
```

6. Estrazione delle informazioni dal pacchetto ricevuto:

```
risposta = DatagramUtility.getContent(packetIN);
```

Esempio Datagram socket - 2

Lato Server: frammenti significativi

1. Creazione socket:

```
DatagramSocket socket = new DatagramSocket(PORT);
```

2. Attesa del pacchetto di richiesta:

```
DatagramPacket packet =
                new DatagramPacket(buf,buf.length);
socket.receive(packet);
```

3. Estrazione delle info *nome_file/n.riga* dal pacchetto ricevuto:

```
String richiesta = DatagramUtility.getContent(packet);
StringTokenizer st = new StringTokenizer(richiesta);
nomeFile = st.nextToken();
numLinea = Integer.parseInt(st.nextToken());
```

4. Creazione del pacchetto di risposta con la linea richiesta:

```
String l = LineUtility.getLine(nomeFile, numLinea);
pack = DatagramUtility.buildPacket(mittAddr,mittPort,l);
```

5. Invio del pacchetto al client:

```
socket.send(pack);
```

Socket Multicast - 1

***Multicast*:** mandare informazioni a un gruppo di destinatari simultaneamente.

Le socket datagram permettono di inviare un pacchetto ad un insieme di processi in multicast, con una sola invocazione a `send()`
`java.net` ha la classe `MulticastSocket` per il multicast.

Chi vuole ricevere pacchetti mandati in multicast deve conoscere:

- la porta della socket usata dal mittente
- l'indirizzo del gruppo a cui vengono inviati messaggi

Per ricevere messaggi multicast, i client devono ***unirsi al gruppo***.

Passi tipici per l'uso di una socket multicast:

1. Preparare un indirizzo, che fungerà da indirizzo di gruppo:
   ```
   InetAddress gruppo = InetAddress.getByName(ind)
   ```
2. Creazione di una socket per il multicast:
   ```
   MulticastSocket msocket = new MulticastSocket(porta)
   ```
3. Connessione della socket al gruppo
   ```
   msocket.joinGroup(gruppo)
   ```

I processi che eseguono `joinGroup()` sulla socket multicast potranno ricevere i datagram inviati a quel gruppo, invocando:
`msocket.receive(packet);`

Un client deve invocare `receive()` periodicamente fino a quando vuole ricevere pacchetti.

Un client può decidere ad un certo punto di uscire dal gruppo:

```
msocket.leaveGroup(gruppo)
```

Socket Multicast - 2

Il processo server crea la socket per una porta ed invia, quando vuole, pacchetti datagram al gruppo:

```
InetAddress group = InetAddress.getByName(ind);
MulticastSocket msocket = new MulticastSocket(porta);
msocket.joinGroup(group);
DatagramPacket packet =
      DatagramUtility.buildPacket(group, porta, linea);
msocket.send(packet);
```

Metodi della classe `MulticastSocket`:

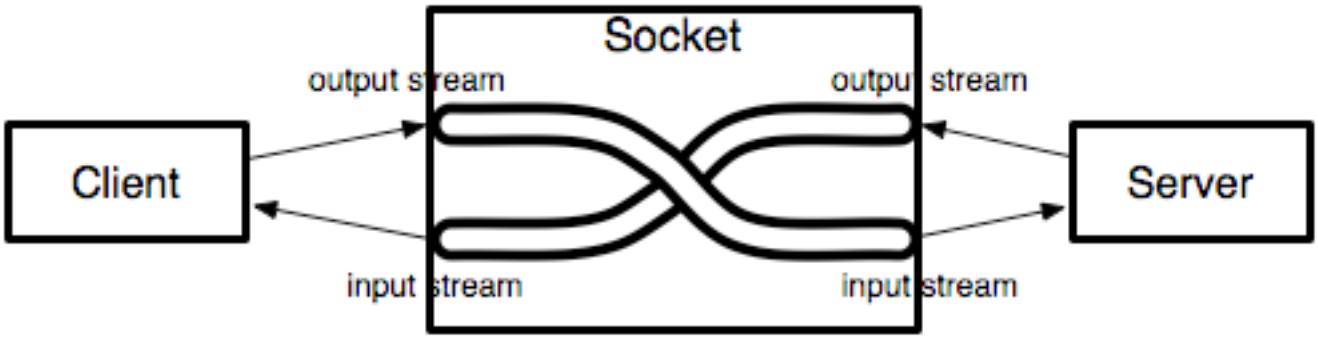
```
joinGroup(InetAddress addr) throws IOException
leaveGroup(InetAddress addr) throws IOException
send(DatagramPacket p)
setTimeToLive(int tlive) throws IOException
```

setta il tempo di vita per i pacchetti multicast mandati sulla socket (`tlive` deve essere compreso tra `0` e `255`).

```
int getTimeToLive() throws IOException
```

Stream socket

Client e server instaurano una connessione che consiste di due flussi di dati, uno per l'input ed uno per l'output.



Le API invocate da client e server, per instaurare una connessione, sono diverse e quindi esistono classi di libreria Java diverse.

Dal lato client vi è il processo che vuole instaurare una connessione per inviare e ricevere dati. La classe Java che usa il client è `Socket`.

Dal lato server vi è un processo che attende una richiesta di connessione, la classe Java che usa è `ServerSocket`.

La connessione è individuata da 4 elementi:

- indirizzo IP del client,
- porta del client,
- indirizzo IP del server,
- porta del server.

Client di uno stream socket - 1

Il client invoca il costruttore della socket, per creare la socket, specificando l'indirizzo e la porta dove risiede il processo server. L'istanziazione della socket e quindi la chiamata al costruttore **instaura la connessione** con il server, a patto che il server sia già in esecuzione e pronto per ricevere questa connessione.

```
Socket socket = new Socket(addr, PORT);
```

con `addr` (di tipo `InetAddress)` e `PORT` (di tipo `int`) che identificano il processo server

Se la creazione della stream socket ha successo, viene prodotta una connessione bidirezionale tra i due processi.

L'apertura della socket è implicita con il costruttore.

La chiusura deve essere chiamata esplicitamente ed è necessaria per non impegnare troppe risorse di sistema (ovvero connessioni). Il numero di connessioni che un processo può aprire è limitato e quindi conviene mantenere aperte solo le connessioni necessarie.

Il metodo della classe `Socket` per chiudere la connessione e disconnettere il client dal server è `close()`

Altri metodi della classe `Socket` sono:

`InetAddress getInetAddress()` restituisce l'indirizzo remoto a cui la socket è connessa

`InetAddress getLocalAddress()` restituisce l'indirizzo locale

`int getPort()` restituisce la porta remota

`int getLocalPort()` restituisce la porta locale

Client di uno Stream socket-2

Dopo la connessione, il client crea uno stream di input, tramite `getInputStream()`, con cui può ricevere i dati dal server.

```
InputStreamReader isr = new InputStreamReader(
                                        socket.getInputStream());
```

La classe `InputStreamReader` converte un flusso di byte in un flusso di caratteri. Per migliorare l'efficenza, creiamo un buffer che consente di leggere un gruppo di caratteri:

```
BufferedReader in = new BufferedReader(isr);
```

Ovvero, allo stream del socket applichiamo i filtri

`InputStreamReader` e `BufferedReader`

Per leggere una linea dallo stream di input della socket:

```
String s = in.readLine()
```

Questa ci restituisce la stringa inviata dal server.

Il client crea anche uno stream di output con cui invierà i dati al server in modo analogo allo stream di input.

```
OutputStreamWriter osw = new
                    OutputStreamWriter(socket.getOutputStream());
BufferedWriter bw = new BufferedWriter(osw);
PrintWriter out = new PrintWriter(bw, true);
```

Per inviare una stringa sullo stream di output: out.println(s)

Quando la connessione non è più necessaria è bene che il client liberi la risorsa “connessione”. Per fare ciò chiude la socket con:

```
out.close(); in.close(); socket.close();
```

Server di uno stream socket

Dal lato server, bisogna creare una istanza di `ServerSocket`:

```
ServerSocket serverSocket = new ServerSocket(PORT);
```

e dopo mettere il server in attesa di una connessione per mezzo di:

```
Socket clientSocket = serverSocket.accept()
```

La chiamata ad `accept()` blocca il server fino a quando un client non instaura una connessione, tuttavia tramite il metodo `setSoTimeout(t)` si può bloccare la `accept()` solo per `t` millisecondi.

La `accept()` restituisce un oggetto di tipo `Socket`. Questo permette al server di usare gli stream che il client ha stabilito.

```
clientSocket.getInputStream()
```

Il server crea gli stream di input e di output per poter poi ricevere e trasmettere dati. Allo stream di input del client corrisponderà lo stream di output del server e viceversa (vedi figura).

Una volta instaurata una connessione, la trasmissione dei dati avviene con gli stessi metodi sia nel client che nel server.

Creazione stream di input (nel server):

```
InputStreamReader isr = new
      InputStreamReader(clientSocket.getInputStream());
BufferedReader in = new BufferedReader(isr);
```

Ricezione di dati dalla socket:

```
in.readLine();
```

Server Socket Paralleli

Quando un server accetta una connessione esso può generare un nuovo thread per servire la richiesta, mentre il thread originale continua a stare in attesa (su `serverSocket.accept()`) di connessioni da altri client.

Questa soluzione permette di non perdere richieste di connessioni e di servire più client che fanno richieste in parallelo.