LABORATORIO di Reti di Calcolatori

Socket in linguaggio Java: servizio connection-oriented

Bibliografia

- slide della docente
- *testo di supporto*: D. Maggiorini, "Introduzione alla programmazione client-server", Pearson Ed., 2009
  - cap.7 (tutto)
  - cap.8 (tutto)
- *Link utili:*
  - *http://docs.oracle.com/javase/tutorial/networking/index.html*
  - *http://docs.oracle.com/javase/6/docs/*

limitazioni Java socket

cosa *non* possiamo fare?

- non supportato il dominio AF_UNIX
- non è possibile interagire direttamente con IP in Java
- connection-oriented, connectionless, domini di indirizzi…
  - Java considera che gli indirizzi possano essere solo IP
  - e che i servizi/protocolli siano solo TCP e UDP
- meccanismi particolari per costruire server concorrenti…

- cosa possiamo fare?
  - possiamo lavorare anche con IPv6

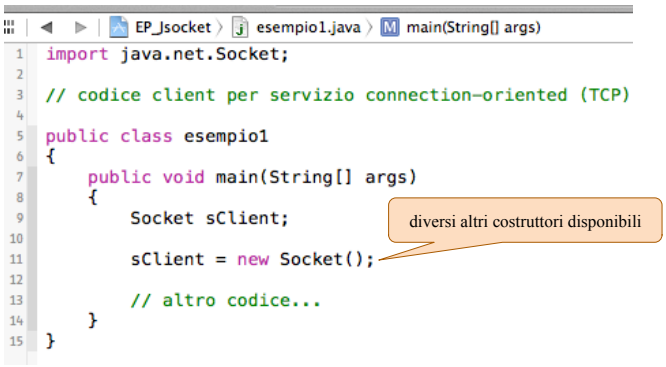
servizio connection-oriented

- ripasso fasi…
  1. creazione socket
  2. binding → gestione indirizzi host + #porta
  3. connessione client e server
  4. scambio dati *(byte stream)*
  5. chiusura

client: 1, 2
server: 2, 3

- classi utilizzate: *package* ***java.net***
  - class Socket (client); ServerSocket (…server)
  - class InetAddress (indirizzi host); InetSocketAddress

1. creazione socket lato client



- abbiamo creato una struttura *del processo* per gestire il *punto terminale del canale* di comunicazione
- adesso dobbiamo indicare a quale indirizzo di rete corrisponde

2. indirizzo host: class InetAddress

- classe senza costruttore; ha metodi per convertire tra i diversi formati degli indirizzi IP (IPv4 o IPv6)
  - `InetAddress InetAddress.getByName(hostName)`
    - da nome simbolico a indirizzo numerico
  - `InetAddress[] InetAddress.getAllByName(hostName)`
    - tutti gli indirizzi associati a quel nome simbolico
  - `InetAddress InetAddress.getLocalHost()`
    - indirizzo dello host locale
- gestiscono struttura utile per lavorare con le socket
  - `byte[] InetAddress.getAddress`
    - estrae indirizzo IP dalla struttura (ma serve conversione!)
  - `String InetAddress.getHostAddress`
    - estrae indirizzo IP decimale puntato dalla struttura
  - `String InetAddress.getHostName`
    - estrae nome simbolico host dalla struttura
- struttura a partire da indirizzo numerico fornita da
  - `InetAddress InetAddress.getByAddress(addr)`

proviamoci un po'...

EP_Jsocket › indirizzi.java › class indirizzi

```
1  import java.net.InetAddress;
2  import java.net.UnknownHostException;
3
4  // codice gestione indirizzi IP / nomi simbolici
5
6  public class indirizzi
7  {
8      public static void main(String[] args)
9      {
10         String nome = "www.unimi.it";   // da convertire in indirizzo IP
11
12         try {
13             InetAddress ia = InetAddress.getByName(nome);
14             byte[] ndp = ia.getAddress();
15             System.out.println("Indirizzo: " + (ndp[0] & 0xff) + "." +
16                                (ndp[1] & 0xff) + "." + (ndp[2] & 0xff) +
17                                "." + (ndp[3] & 0xff));
18         }
19         catch (UnknownHostException uhe) {
20             uhe.printStackTrace();
21         }
22     }
23 }
```

convertiamo lo unsigned long (32 bit) in 4 ottetti di interi senza segno

homework

1. modificare codice in modo da estrarre l'indirizzo IP dello host locale su cui si lavora
2. proviamo a usare *getAllByName* per estrarre tutti gli indirizzi IP per il dominio www.google.com
   - serve ciclo lungo `<nome_struttura>.length`

```
10         String nome = "www.google.com";   // da convertire in indirizzo IP
11
12         try {
13             InetAddress[] iaa = InetAddress.getAllByName(nome);
14
15             for (int i=0; i < iaa.length; i += 1)
16             {
17                 System.out.println("Indirizzo " + iaa[i].getHostName() +
18                                    " --> " + iaa[i].getHostAddress());
19             }
20         }
21         catch (UnknownHostException uhe) {
22             uhe.printStackTrace();
```

- come funziona? interroga il *resolver* (→ *Teoria*!)

2. costruzione indirizzo per socket

- abbiamo visto che serve indirizzo host + #port
- per server: numero porta ben noto
  - … o comunicato ai potenziali client
- per client: numero qualsiasi, anche scelto da S.O.
  - è iniziatore: al 1° messaggio dà #port a server per risposta
- class InetSocketAddress con costruttori:
  - `InetSocketAddress(InetAddress addr, int port)`
  - `InetSocketAddress(String hostname, int port)`
  - `InetSocketAddress(int port)`
- e con metodi utili:
  - `InetAddress getAddress()`
  - `String getHostName()`
  - `int getPort()`

2. proviamo un po'

EP_Jsocket › esempio1.java › class esempio1

```
1  import java.net.Socket;
2  import java.net.InetAddress;
3  import java.net.InetSocketAddress;
4  import java.net.UnknownHostException;
5
6  // codice client per servizio connection-oriented (TCP)
7
8  public class esempio1
9  {
10     public static void main(String[] args)
11     {
12         Socket sClient;
13         InetAddress ia;     // IP address client
14         InetSocketAddress isa;  // socket address client
15
16
17         sClient = new Socket();
18         try {
19             ia = InetAddress.getLocalHost();
20             isa = new InetSocketAddress(ia, 50000);
21         }
22         catch (UnknownHostException uhe) {
23             uhe.printStackTrace();  }
24     }
25 }
```

scelta porta in codice
…ma potrebbe essere già in uso!

2. binding esplicito

- metodo `void Socket.bind(SocketAddress bindpoint)`
  - colleghiamo struttura processo a informazioni per S.O.
  - SocketAddress è superclasse di InetSocketAddress

```
19          sClient = new Socket();
20          try {
21              ia = InetAddress.getLocalHost();
22              isa = new InetSocketAddress(ia, 0); // S.O. sceglie #port libero
23              sClient.bind(isa);
24              System.out.println("Porta allocata: " + sClient.getLocalPort());
25              Thread.sleep(120 * 1000);
26          } catch(Exception e) {
27              e.printStackTrace();  }
```

dopo associazione

- #port 0 lascia scelta porta libera al S.O.
  - non va tanto bene per il server…
- comandi *netstat* oppure *lsof* mostrano stato socket
  - `CLOSED` : non è connessa ad alcun server

implementazione server e connessione

- creazione socket con due costruttori di ServerSocket:
  - `ServerSocket()` oppure `ServerSocket(int port)`
  - il primo crea socket non connessa → *serve bind successiva*
    - manipolazione indirizzi come per caso client
  - nel secondo caso, #port può essere 0
    - si crea già **coda** per ospitare richieste connessione pendenti
    - stato socket risulta `LISTEN`
- connessione: il server si mette in attesa di richieste
  - `Socket ServerSocket.accept()`
    - bloccante in attesa di clienti
    - crea nuova Socket per comunicare con specifico client
    - ricordate discorso su *associazione*?

3. creazione connessione (server)

EP_Jsocket › es1SrvIter.java › main(String[] args)

```
1  import java.net.ServerSocket;
2  import java.net.Socket;
3  import java.io.IOException;
4
5  // codice server per servizio connection-oriented (TCP)
6
7  public class es1SrvIter
8  {
9      public static void main(String[] args)
10     {
11         ServerSocket sSrv;
12         Socket toClient;
13         try {
14             sSrv = new ServerSocket(0);
15             System.out.println("Indirizzo: " + sSrv.getInetAddress()
16                                + "; porta: " + sSrv.getLocalPort());
17             toClient = sSrv.accept();
18             System.out.println("Ind Client: " + toClient.getInetAddress()
19                                + "; porta: " + toClient.getPort());
20             Thread.sleep(240 * 1000);
21         } catch(Exception e) {
22             e.printStackTrace();  }
23     }
24 }
```

visualizza indirizzo (di trasporto) locale che è stato associato alla socket

visualizza indirizzo (di trasporto) del client

3. creazione connessione (client)

EP_Jsocket › esempio1.java › class esempio1

```
1  import java.net.Socket;
2  import java.net.InetAddress;
3  import java.net.InetSocketAddress;
4
5  import java.net.UnknownHostException;
6  import java.io.IOException;
7
8  // codice client per servizio connection-oriented (TCP)
9
10 public class esempio1
11 {
12     public static void main(String[] args)
13     {
14         Socket sClient;
15         InetAddress ia;     // IP address SERVER
16         InetSocketAddress isa;  // socket address SERVER
17
18
19         sClient = new Socket();
20         try {
21             ia = InetAddress.getLocalHost();
22             isa = new InetSocketAddress(ia, 57195); // porta server da inserire...
23             sClient.connect(isa);
24             System.out.println("Porta locale: " + sClient.getLocalPort());
25             System.out.println("Indirizzo: " + sClient.getInetAddress()
26                                + "; porta: " + sClient.getPort());
27             Thread.sleep(120 * 1000);
28         } catch(Exception e) {
29             e.printStackTrace();  }
30     }
31 }
```

CLIENT CHE DIALOGA CON SERVER SU STESSO HOST

deve essere la porta stampata come locale dal server

stampa porta locale e indirizzo server

3. creazione connessione

- metodo `void Socket.connect(SocketAddress peer)`
  - esecuzione *three-way handshake* (→ *Teoria)*
  - esegue contestualmente anche bind implicito
- indirizzo locale server è 0.0.0.0 che indica *any*
  - *attenzione che getInetAddress su ServerSocket mostra indirizzo locale; su Socket mostra indirizzo remoto*
- lo output di lsof mostra 3 socket sul sistema:

```
java   pid_srv   user   IP_srv   TCP   *:57220   (LISTEN)
java   pid_srv   user   IP_srv   TCP   nome_srv:57220->
nome_cli:57223 (ESTABLISHED)
java   pid_cli   user   IP_cli   TCP   nome_cli:57223->
nome_srv:57220 (ESTABLISHED)
```


4. scambio dati

- tutti i dati devono essere convertiti in sequenze di caratteri
- caratteri: cast a tipo byte (unicode → ASCII)
- stringhe: attenzione a carriage return `\r` e line feed `\n`
  - se danno fastidio: `String.replace()` per sostituire con “”
- numeri: formato dipende da architettura… → tre strade
  - `String stringa = "" + numero`
  - metodo `toString` di classe base. Es: `Double.toString(num)`
  - per l’inverso sui dati ricevuti: metodo `parse<type>`
    - es. `double numero = Double.parseDouble(stringa)`
- dati strutturati: conversione dei singoli campi
  - o struttura definita come implementazione di `Serializable`

Serializable

- Object serialization: *is the process of saving an object's state to a sequence of bytes, as well as the process of rebuilding those bytes into a live object at some future time*
- Viene anche detto *(un)marshalling*
- Attenzione: non salvo la classe ma l'oggetto!
  - Questo significa che il lato ricevente deve avere accesso alla classe (ovvero deve disporre del file *.class*)
- possibile se (super)classe implementa interfaccia `Serializable`
- in generale introduce parecchie complicazioni

Ripasso Java···

per l'esame è sufficiente ricordarsi di:

- metodo String **split**(String regex, int limit)
  - rompe la stringa eliminando il separatore campi indicato da *regex* ottenendo il numero di sottostringhe indicato da *limit*
- metodo String **trim**()
  - elimina spazi iniziali e finali in una stringa
  - es. per "pulire" input da spazi impropri prima dell'uso
- classe **StringTokenizer**:
  - costruttore per sottostringhe delimitate da separatore
  - metodo nextToken() per ottenere successiva sottostringa

4. scambio dati

- terminali canali di comunicazione (unidirezionali) da
  - `InputStream Socket.getInputStream()`
  - `OutputStream Socket.getOutputStream()`
- da essi si può scrivere / leggere con `write` / `read`
  - `write` passa dati a livello Transport (non a canale!)
  - `read` è bloccante finchè non legge dei byte dal canale
    - in tal caso rende #byte effettivamente letti
    - con byte stream, questi non sono necessariamente tutti i byte del messaggio /* → *Teoria* per struttura segmenti TCP */
    - serve protocollo di applicazione per sapere *quanto* o *fino a quando* leggere
    - se canale chiuso da peer, `read` si sblocca tornando <0

4. scambio dati client-server

- con l'import di tutti i package del caso...
- e gestendo opportunamente tutte le eccezioni sollevabili

CLIENT

```
31      InputStreamReader tastiera = new InputStreamReader(System.in);
32      BufferedReader br = new BufferedReader(tastiera);
33      String frase = br.readLine();
34      OutputStream toSrv = sClient.getOutputStream();
35      toSrv.write(frase.getBytes(), 0, frase.length());
36    } catch(Exception e) {
37      e.printStackTrace();  }
```

conversione…

SERVER

```
22      int dim_buffer = 100;
23      byte buffer[] = new byte[dim_buffer];
24      InputStream fromCl = toClient.getInputStream();
25      int letti = fromCl.read(buffer);
26      String stampa = new String(buffer, 0, letti);
27      System.out.println("Ricevuta stringa: " + stampa + " di " +
28                         letti + " byte");
29    } catch(Exception e) {
30      e.printStackTrace();  }
```

conversione…

4. uso di *split*

CLIENT

```
System.out.println("Inserisci frase:");
frase = br.readLine();
System.out.println("Inserisci float:");
numero = Double.parseDouble(br.readLine());
totale = frase + "---" + Double.toString(numero);
System.out.println("messaggio: " + totale);
// totale += "\r\n";
OutputStream toSrv = sClient.getOutputStream();
toSrv.write(totale.getBytes(), 0, totale.length());
```

conversione da String a double

String

conversione da double a String

SERVER

```
letti = fromCl.read(buffer);
if (letti > 0)  {
    String stampa = new String(buffer, 0, letti);
    System.out.println("Server: Ricevuta stringa: " + stampa + " di " + letti +
                       " byte da " + toClient.getInetAddress() + "; " + toClient.getPort() );
    String[] splittata = stampa.split("---",0);
    for(int i=0; i<splittata.length; i++)  {
        System.out.println(splittata[i] + "  ");
    }
```

divisione in numero illimitato di sottostringhe

- N.B.: numero e tipo campi è parte del (vostro) protocollo
- N.B.: i campi numerici vanno ri-convertiti da String al tipo opportuno

4. uso di *StringTokenizer*

CLIENT

```
System.out.println("Inserisci frase:");
frase = br.readLine();
System.out.println("Inserisci float:");
numero = Double.parseDouble(br.readLine());
totale = frase + "@" + Double.toString(numero);
System.out.println("messaggio: " + totale);
// totale += "\r\n";
OutputStream toSrv = sClient.getOutputStream();
toSrv.write(totale.getBytes(), 0, totale.length());
```

- client identico a prima (solo cambiato delimitatore)

SERVER

```
letti = fromCl.read(buffer);
if (letti > 0)  {
    String stampa = new String(buffer, 0, letti);
    System.out.println("Server: Ricevuta stringa: " + stampa + " di " + letti +
                       " byte da " + toClient.getInetAddress() + "; " + toClient.getPort() );
    StringTokenizer splittata = new StringTokenizer(stampa, "@");
    while (splittata.hasMoreTokens())  {
        System.out.println(splittata.nextToken());
    }
```

- N.B.: in entrambi i casi il delimitatore deve essere tale da non poter *mai* essere incluso in un valore valido di un campo

5. chiusura

- metodo `close()` non permette ulteriore utilizzo del canale
  - attenzione nel server: quale socket si vuole chiudere?
  - attiva con client servito correntemente?
  - passiva → non accetto altri client
- non vuol dire che rilascio tutte le strutture
  - problema delayed data; dati ancora bufferizzati in kernel S.O. …
  - → *Teoria* per procedura di chiusura a livello trasporto
- per garantire che tutte le socket siano chiuse si può usare *close* in blocco

```
40          finally  {
41              try {
42                  sClient.close();
43              } catch (Exception e) {
44                  System.err.println("Client error");
45                  e.printStackTrace();
46              }
47          }
```


homework

- guardare documentazione metodi per alternative
  - es. i vari costruttori `Socket` disponibili
- implementato servizio Echo → complichiamolo
  - client può mandare più stringhe che il server riproduce
  - dopo che il server ha stampato una frase, notifica al client che può mandargli la successiva
  - se il server riceve carattere '0' dal client, chiude la connessione con lui
  - dopo che il client ha letto '0' da tastiera e inviato a server, chiude la socket con lui
- client può ricever IP e porta server da linea di comando