LABORATORIO di Reti di Calcolatori

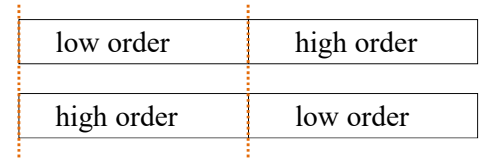
Java Socket: scambio dati

Bibliografia

- slide della docente
- *testo di supporto*: D. Maggiorini, "Introduzione alla programmazione client-server", Pearson Ed., 2009
  - cap.4 (tutto)
  - cap.5 (tutto)
  - cap.7 (tutto)
  - cap.8 (tutto)
- *Link utili:*
  - *http://docs.oracle.com/javase/tutorial/networking/index.html*
  - *http://docs.oracle.com/javase/6/docs/*

scambio dati: "marshalling"

- def.rete: "insieme interconnesso di calcolatori *autonomi ed eterogenei*"
  - es. non posso passare dati per puntatore!
  - ma può essere diversa anche rappresentazione interna
- byte order: non tutti gli host memorizzano i byte all'interno di una parola nello stesso ordine
  - rete: big endian (byte più significativo a indirizzo basso)



little endian (Intel x86)

big endian (Motorola, HP, IBM, Internet)

marshalling

- rete ha solo coscienza che sposta dati (byte/bit)
- programmatore di applicazione deve provvedere ad eventuali conversioni (numeri, struct)
  - formato host 1 → formato Internet → formato host 2
- nel caso byte-stream: programmatore di applicazione deve gestire formato PDU
  - lettura numero fisso byte se PDU di taglia costante
  - lettura byte header, e successivamente dati (per lunghezza indicata in header), se PDU di taglia variabile

→ *protocollo di applicazione*

4. scambio dati

- *tutti i dati devono essere convertiti a/da sequenze di byte*
- 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… → due 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 **trim**()
  - elimina spazi iniziali e finali in una stringa
  - es. per "pulire" input da spazi impropri prima dell'uso
- 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*
- 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

```
37      System.out.println("Inserisci frase:");
38      frase = br.readLine();
39      System.out.println("Inserisci float:");
40      numero = Double.parseDouble(br.readLine());
41      totale = frase + "---" + Double.toString(numero);
42      System.out.println("messaggio: " + totale);
43      // totale += "\r\n";
44      OutputStream toSrv = sClient.getOutputStream();
45      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 '**.**' dal client, chiude la connessione con lui
  - dopo che il client ha letto '**.**' da tastiera e inviato a server, chiude la socket con lui
- client può ricever IP e porta server da linea di comando