LABORATORIO di Reti di Calcolatori

Socket in linguaggio Java: modelli di servizio

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/*

...e i modelli di servizio?!

	connection-oriented	connectionless
s. iterativo	√	non ha senso
s. concorrente	non supportato dal linguaggio	non supportato dal linguaggio *(ma viene gratis)*
s. multi-thread	√	non ha senso

- in realtà, per servizio connection-less non ha senso parlare di alcun modello di servizio…
- server iterativo: si ottiene facilmente da esempi mostrati

```
while(true) {
    accept;
    comunicazione con client corrente su nuova socket;
    chiusura socket dedicata;
}
```

→ homework!

pseudo-codice client/server iterativo

CLIENT

```
do  {
    letto ← read from tastiera;
    toServer.write(letto);
    if (letto != carattere finale)
          toServer.read(buffer);
} while(letto != carattere finale);
toServer.close();
```

SERVER

```
while(true)
{
   fromClient ← ServSock.accept();
   do  {
     fromClient.read(letto);
     if (letto != carattere finale)
          fromClient.write(letto);
   } while((letto != carattere finale);
   fromClient.close();
}
ServSock.close();
```

server connection-oriented concorrente

- si può fare con **time-out** e politica di polling (analisi circolare su socket passiva, e tutte le socket attive già aperte)
- **void setSoTimeout(int msec)**
  - su ServerSocket interrompe accept
  - su Socket interrompe read
  - se scatta timeout è sollevata eccezione
    - ma socket restano valide
  - se msec=0 → attesa infinita
- è un busy waiting (!)
- problema gestione socket client chiuse

```
Socket[] fromcl
serverSocket passive
Int index=0
While(True)
{
    try{
            passive.setSoTimeout(3)
            fromcl[index] ← passive.accept()
    }catch( java.net.SocketTimeoutException){…}
Index++
    for(i=0; i++; i<index)
    {
            fromcl[i].setSoTimeout(3)
            try{
            fromcl[i].read()
            }catch(java.net.SocketTimeoutException)
                    {…}
            consuma quanto letto
    }
}
```


esempio codice (1)

il server alterna il suo tempo tra guardare la socket passiva per nuove richieste di connessione, e guardare le socket client

```
44              sServ = new ServerSocket(0);
45              System.out.println("sAddr:" + sServ.getInetAddress()
46                                 + "; sPort: " + sServ.getLocalPort());
47              while(true){
48                  // Creazione ServerSocket
49                  // Accept Client fino a Timeout o max_conn
50                  try{
51                      sServ.setSoTimeout(sServ_timeout);
52
53                      while(index<max_conn){
54                          sClient.add(sServ.accept());
55                          index++;
56                      }
57                  }catch(SocketTimeoutException ste){
58                      System.out.println("\nServerSocket: Timeout expired!!!\n");
59                  }catch(IOException ioe){
60                      System.out.println("SocketServer Exception:");
61                      ioe.printStackTrace();
62                  }
```

parte (2)

```
65                  // Gestione RR dei Client
66                  while(index > 0){
67                      System.out.println("cAddr:" + sClient.get(i).getInetAddress()
68                                         + "; cPort: " + sClient.get(i).getPort());
69                      try{
70                          sClient.get(i).setSoTimeout(sClient_timeout);
71                          InputStream isC = sClient.get(i).getInputStream();
72                          while(true){
73                              int letti = isC.read(buff);
74                              String str_cli = (new String(buff, 0, letti)).trim();
75
76                              if(str_cli.equals("0")){
77                                  //throw new Exception("End of Client");
78                                  sClient.get(i).close();
79                                  sClient.remove(i);
80                                  index--;
81                                  break;
82                              }
83                              System.out.println(str_cli);
84                          }
85                      }catch(SocketTimeoutException ste){
86                          System.out.println("Client: Timeout expired!!!");
87                      }catch(Exception e){
88                          e.printStackTrace();
89                          try{
90                              sClient.get(i).close();
91                              sClient.remove(i);
92                              index--;
93                          }catch(IOException ioe){
94                              ioe.printStackTrace();
95                          }
96                      }
97                      i = index!=0 ? (i+1)%index : 0;
98                  }
```


server multi-thread

- isoliamo la parte di comunicazione con il cliente in una classe che estende la classe Thread
- il metodo `run` di tale nuova classe deve eseguire la parte di codice che gestisce la comunicazione con il client
- in alternativa:
  - sul thread viene chiamato il metodo `start` dopo la creazione
  - oppure, il metodo `start` è inglobato nel creatore della nuova classe
  - … teniamo il client come nel primo esempio di servizio connection-oriented
    - N.B.: bisogna ricompilare con porta server corretta

server multi-thread

```
9  public class es1SrvIter
10 {
11     public static void main(String[] args)
12     {
13         ServerSocket sSrv;
14         Socket toClient;
15         try {
16             sSrv = new ServerSocket(0);
17             System.out.println("Indirizzo: " + sSrv.getInetAddress()
18                                + "; porta: " + sSrv.getLocalPort());
19
20             while (true)
21             {
22                 toClient = sSrv.accept();
23                 System.out.println("Ind Client: " + toClient.getInetAddress()
24                                    + "; porta: " + toClient.getPort());
25                 Thread t = new erogaServizio(toClient);
26                 t.start();
27                 // toClient.close();
28             }
29
30         } catch(Exception e) {
31             e.printStackTrace();  }
32     }
33 }
```

problema: la socket connessa al client va chiusa solo quando termina la comunicazione con esso, ovvero quando termina il thread (fig.8.23)

classe per server dedicato

```
6  public class erogaServizio extends Thread
7  {
8      private Socket sock2Cl;
9
10     public erogaServizio(Socket socket)
11     {
12         this.sock2Cl = socket;
13     }
14
15     public void run()
16     {
17         int dim_buffer = 100;
18         byte buffer[] = new byte[dim_buffer];
19
20         while (true)
21         {
22             try  {
23                 InputStream fromCl = sock2Cl.getInputStream();
24                 int letti = fromCl.read(buffer);
25                 if (letti > 0)  {
26                     String stampa = new String(buffer, 0, letti);
27                     System.out.println("Ricevuta stringa: " + stampa + " di " + letti + " byte da " + sock2Cl.getInetAddress()
28                                        + "; " + sock2Cl.getPort() );
29                 }
30                 else  {
31                     sock2Cl.close();
32                     return;
33                 }
34             }  catch(Exception e) {
35                 e.printStackTrace();  }
36         }
37     }
38 }
```

eseguito quando si fa partire (start) il thread

stampiamo identità client da cui si è ricevuto lo specifico messaggio

qui si!
e si chiude anche per server primario perché condividono memoria

- provare a lanciare più client concorrenti da terminali differenti
- il server (*giustamente*) non termina mai…

homework

- modificare client/server connessi in modo che:
  1. il client possa mandare più stringhe. Il client termina quando riceve in input da tastiera il carattere '.' → lo invia al server che chiude connessione con questo client
  2. il server invii in risposta al client la stringa da esso ricevuta (servizio standard *Echo*)
  3. punti 1+2 con server sia iterativo sia multi-thread che gestisce conversazioni con più client contemporaneamente
- modificare client/server connectionless in modo che:
  1. il server invii in risposta al client la stringa da esso ricevuta (servizio standard *Echo*)
     - guardando il file /etc/services si scopre che *Echo* è un servizio (standard) multiprotocollo: può usare sia UDP sia TCP
- testare i codici con più client contemporanei

Thread e mutua esclusione

- può essere utile far condividere dati tra server principale e i vari thread che gestiscono i client
  - coordinamento, raccolta dati…
- due meccanismi possibili:
  1. gli oggetti condivisi vengono gestiti da metodi synchronized che pongono un lock all'accesso
     - no garanzie sull'ordine di esecuzione in caso di concorrenza
     - *synchronized* anche su semplice blocco:

```
synchronized (oggetto su cui si vuole lock) { statement }
```

  2. metodi wait() e notify() per far comunicare thread

Thread: mutua esclusione (1)

- lasciamo il client invariato
  - scambio stringhe fino a ricezione di "." da tastiera
- diciamo che i thread condividono un oggetto di class *contaStringhe*, che mantiene il numero totale di stringhe scambiate dai vari thread con i loro client

```
1  public class contaStringhe
2  {
3      private int totale;
4
5      public contaStringhe()
6      {
7          this.totale = 0;
8          System.out.println("inizializzato totale: " + totale);
9      }
10
11     public synchronized int incrementa(int valore)
12     {
13         totale += valore;
14         return totale;
15     }
16 }
```

mutua esclusione su accesso a totale

Thread: mutua esclusione (2)

- il (main) server dichiara oggetto da condividere tra i thread, a cui lo passa al momento della creazione

```
22              lavoro = new contaStringhe();
23
24              while (true)
25              {
26                  toClient = sSrv.accept();
27                  System.out.println("Ind Client: " + toClient.getInetAddress()
28                                     + "; porta: " + toClient.getPort());
29                  Thread t = new erogaServizio(toClient, numThread, lavoro);
```

struttura condivisa

- ogni thread accede alla variabile

```
10      private contaStringhe lavoro;
11
12      public erogaServizio(Socket socket, int ID, contaStringhe lavoro)
13      {
14          sock2Cl = socket;
15          mioID= ID;
16          this.lavoro = lavoro;
17      }
```

identificatore progressivo incrementato e passato dal main server

  - e mantiene conteggio #stringhe scambiate durante la sua vita

Thread: mutua esclusione (3)

- a termine, ogni thread accumula nel contatore condiviso il proprio numero stringhe

```
48            finally  {
49                try {
50                    sock2Cl.close();
51                    System.out.println("Thread " + mioID + " termina; scambiate " + valore + " stringhe");
52                    System.out.println("nuovo totale: " + lavoro.incrementa(valore));
53                } catch (Exception e) {
54                    System.err.println("server thread error");
55                    e.printStackTrace();
56                }
57            }
```

  - qui thread modificato per scambiare più stringhe
  - esce a ricezione di "." ed esegue blocco *finally*
    - la ricezione di "." *non* è contata come stringa scambiata ☺
- *esercizio:* modificare codice e provare con più client che dinamicamente si (s)collegano

wait( ) e notify( )

- un thread si può bloccare in attesa di un evento
  - la wait() non ha condizioni → quando il thread viene risvegliato **DEVE** controllare se è perché si è verificato l'evento atteso

```
public final void wait()
public final void wait(long timeout)
public final void wait(long timeout, int nanos)
```

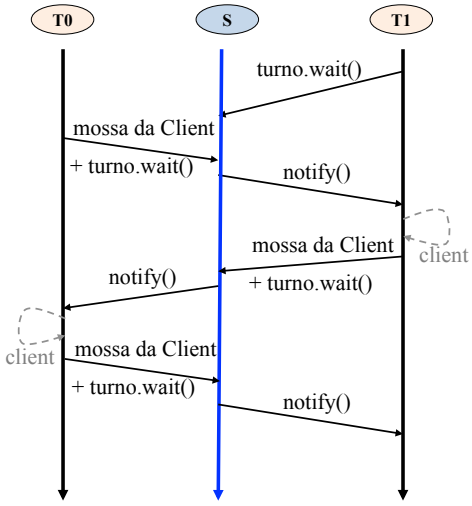
  - si può specificare timeout dopo il quale il thread comunque si risveglia
  - se il tempo è zero, l'attesa è indefinita (equivalente a wait())
- notify() risveglia **un** thread (quello in attesa da più tempo) tra tutti quelli che avevano chiamato la wait()
  - notifyAll() per risvegliarli tutti

comunicazione tra thread (1)

- wait() e notify() devono essere chiamati in metodi synchronized
  - azioni di *wait()* e rilascio *lock* sono atomiche
  - thread risvegliato da *notify()* ri-acquisisce il lock
- proviamo a fare un gioco degli scacchi
  - il primo client che si connette può giocare subito
  - il secondo client (e relativo thread) aspettano il loro turno
  - i thread si sincronizzano sull'accesso alla scacchiera
- N.B.: *soluzione abbozzata!* Bisognerebbe
  - controllare che i giocatori siano esattamente due
  - controllare che le stringhe siano mosse valide, ecc…

comunicazione tra thread (2)

- client ←→ thread:
  - client manda mossa
  - attende notifica di turno
- thread ←→ scacchiera:
  - thread manda mossa proprio client
  - se mossa era "." termina
  - altrimenti va in wait
  - quando riceve notify() avvisa il client che è il suo turno e legge mossa successiva



comunicazione tra thread (3)

- client: analogo a esempi precedenti
  - deve alternare invio a server e lettura da server

```
            do {
                InputStreamReader tastiera = new InputStreamReader(System.in);
                BufferedReader br = new BufferedReader(tastiera);
                mossa = br.readLine();
                // mossa += "\r\n";
                toSrv.write(mossa.getBytes(), 0, mossa.length());
                if (!mossa.equals("."))  {
                    letti = fromSrv.read(buffer);
                    String stampa = new String(buffer, 0, letti);
                    System.out.println("ricevuto da server " + stampa);
                }
            } while (!mossa.equals("."));  // butto giù il re
        } catch(Exception e) {
            e.printStackTrace();  }
```

- N.B.: in fase di test verificare che la turnazione tra i due giocatori sia corretta (non ammesse due mosse di seguito)

comunicazione tra thread (4)

```
import java.lang.InterruptedException;

public class scacchiera
{

    public scacchiera()
    {
        System.out.println("inizializzata scacchiera");
    }

    public synchronized void muovi(String player, int playerID)
    {
        System.out.println("mossa " + player + " da giocatore " + playerID);
        notify();   // lascio giocare l'altro
        return;
    }

    public synchronized void turno()
    {
        try  {
            wait();
        }  catch(Exception e) {
            e.printStackTrace();  }
    }
}
```

eccezioni sollevate da wait() e notify()

- main server simile a prima: crea un oggetto *scacchiera* che condivide con i due thread via costruttore

comunicazione tra thread (5)

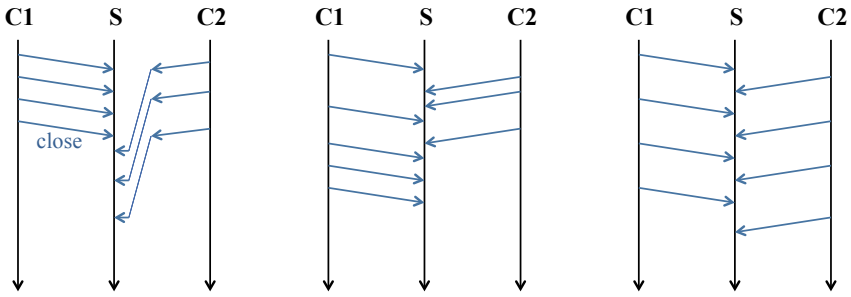
```
31              if (mioID > 0)  {
32                  System.out.println("Thread " + mioID + " va a dormire");
33                  myBoard.turno();
34              }
35              do {
36              letti = fromCl.read(buffer);
37              if (letti > 0)  {
38                  String stampa = new String(buffer, 0, letti);
39                  System.out.println(mioID + ": Ricevuta mossa da " + sock2Cl.getInetAddress()
40                                     + "; " + sock2Cl.getPort() );
41                  myBoard.muovi(stampa, mioID);
42                  if (stampa.equals("."))  {
43                      fine = 1;
44                      return;
45                  }
46                  else  {
47                      myBoard.turno();     // aspetto mossa altro giocatore
48                      stampa = "tocca a te";
49                      toCli.write(stampa.getBytes(), 0, stampa.length());
50                  }
51              }
52              } while (fine==0);
53          }  catch(Exception e) {
54              e.printStackTrace();  }
```

un thread (deterministicamente) aspetta che agisca l'altro (spezziamo simmetria)

risveglio l'altro thread

- … e blocco *finally* come in precedenza per chiudere socket client

Confronto pattern di coumunicazione



ITERATIVO (p.12)**:** tutte le interazioni del client 2 vengono DOPO tutte le interazioni del client 1

MULTI-TREAD (stringhe, p.14-20)**:** le interazioni si possono interfogliare *in qualsiasi modo*

MULTI-TREAD (scacchi, p.23-26)**:** le interazioni si interfogliano secondo una *stretta alternanza*

alcune considerazioni finali

- il S.O. memorizza le richieste di connessione dei client in una coda first-in first-out
  - la massima lunghezza coda dipende dal S.O. (di solito 50)
    - costruttore `ServerSocket(int port, int backlog, InetAddress bindAddr)`
  - Se il numero di richieste in coda eccede la capacità massima, le successive richieste vengono scartate direttamente dal S.O.
  - Il client deve gestire le situazioni in cui la richiesta di connessione non va a buon fine
    - quindi: bisogna fare il *catch* dell'eccezione e gestirla

alcune considerazioni finali

- `ServerSocket.close()` rilascia la porta passiva *e tutte le porte create da* `accept()`
  - lo fa anche il garbage collector quando il programma termina
  - in ogni caso, le porte non sono immediatamente riutilizzabili
    - → *Teoria* per definizione *Maximum Segment Lifetime*
- Java permette anche limitata configurazione del modo di operazione delle socket
  - `Socket.getReuseAddress()` / `Socket.setReuseAddress()`
  - `Socket.setKeepAlive()` , `Socket.SoTimeout()`
  - …ma le vediamo meglio in C dopo lezioni di Teoria

gestione eccezioni

- negli esempi fatta un po' brutalmente
- bisognerebbe distinguere i vari casi di errore ed intraprendere operazioni opportune in dipendenza della semantica del servizio
  - quando il server è in situazione di errore e va chiuso?
  - quando la connessione è in situazione di errore e va chiusa, ma il server può continuare ad operare con altri client?
- distinguere tra errori su indirizzi, errori su canali, errori su I/O da tastiera…