Lezione n.5
LPR-Informatica Applicata
Costruzione Pacchetti UDP
Thread Miscellanea

30/3/2009

Laura Ricci

COSTRUZIONE DI PACCHETTI UDP

- JAVA consente la trasformazione automatica dei dati primitivi in sequenze di bytes da inserire all'interno del pacchetto mediante
  - Le classi DataOutputStream()
  - Le classi ByteArrayOutputStream/ByteArrayInputStream
  - Il metodo toByteArray( )
- I dati possono essere assemblati dal mittente mediante le corrispondenti classi
  - DataInputStream/ByteArrayInputStream
- E' possibile inserire più dati primitivi JAVA (interi, stringhe, booleani,....) all'interno dello stesso pacchetto UDP
  - dopo aver scritto una sequenza di dati (anche di tipo diverso) su DataOutputStream, si effettua una singola conversione mediante il metodo toByteArray( )

INSERIRE PIU' DATI IN UN PACCHETTO

```java
byte byteVal=101;
short shortVal=10001;
int intVal = 100000001;
long longVal= 100000000001L;
ByteArrayOutputStream buf = new ByteArrayOutputStream();
DataOutputStream out = new DataOutputStream(buf);
out.writeByte(byteVal);
out.writeShort(shortVal);
out.writeInt(intVal);
out.writeLong(longVal);
byte[  ] msg = buf.toByteArray(  );
```

ESTRARRE PIU' DATI DA UN PACCHETTO

```
byte byteValIn;
short shortValIn;
int intValIn;
long longValIn;
ByteArrayInputStream bufin = new ByteArrayInputStream(msg);
DataInputStream in = new DataInputStream(bufin);
byteValIn=in.readByte();
shortValIn=in.readShort();
intValIn=in.readInt();
longValIn=in.readLong();
```

CODIFICARE LE INFORMAZIONI

- Il protocollo UDP (e come vedremo anche per il TCP) consente di unicamente di gestire sequenze di bytes. Un byte viene interpretato come un intero nell'intervallo [0..255].
- La codifica dei tipi di dato primitivi di un linguaggio in sequenze di bytes può essere realizzata
  - dal supporto del linguaggio (come visto nei lucidi precedenti)
  - esplicitamente dal programmatore
- In ogni caso, il mittente ed il destinatario devono accordarsi sulla codifica stabilita. Ad esempio, per un dato di tipo intero si deve stabilire:
  - la dimensione in bytes per ogni tipo di dato
  - l'ordine dei bytes trasmessi,
  - l'interpretazione di ogni byte (con segno/senza segno)
- Il problema è semplificato se mittente e destinatario sono codificati mediante il seguente linguaggio (ad esempio entrambi in JAVA)

CODIFICARE VALORI DI TIPO PRIMITIVO

Per scambiare valori di tipo intero, occorre concordare:

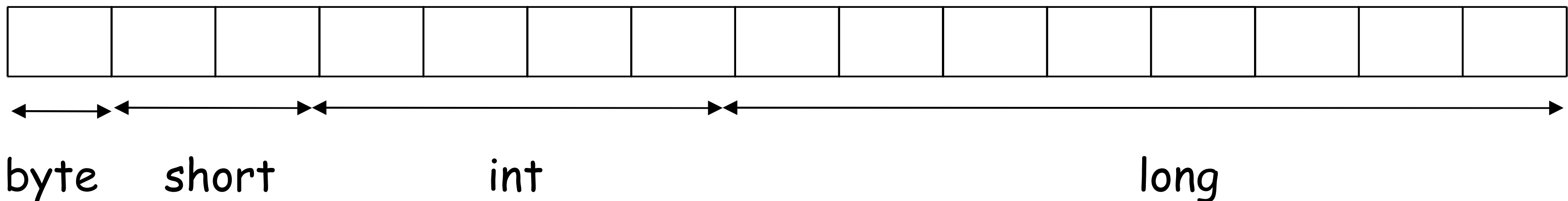
- dimensione dei tipi di dati scambiati. Long: 8 bytes, int: 4 bytes, short: 2 bytes
- ordine dei bytes trasmessi
  - Little-endian: il primo byte trasmesso è il meno significativo
  - Big-endian: il primo byte trasmesso è il più significativo
- Interpretazione dei valori: con/senza segno

Nel caso di valori di tipo stringa, occorre concordare

- codifica adottata per i caratteri contenuti nella stringa (UTF-8, UTF-16,...)

CODIFICARE VALORI DI TIPO INTERO

- Supponiamo di dover costruire un pacchetto UDP contenente quattro valori interi: un byte, uno short, un intero ed un long



- Non si vogliono utilizzare le classi filtro DataOutputStrem e ByteArrayOutputStream
- Soluzione alternativa: si utilizzano operazioni che operano direttamente sulla rappresentazione degli interi
  - si definisce message, un vettore di bytes
  - si selezionano i bytes della rappresentazione mediante shifts a livello di bits
  - si inserisce ogni byte selezionato in una posizione del vettore message

CODIFICARE VALORI DI TIPO INTERO

```
public static int encodePacket(byte[ ] dst, int offset,
   long val, int size)
     {for (int i=size; i >0; i--)
            { dst[offset++]  = (byte) (val >> ((i -1)*8));}
     return offset;};
```

- val valore intero
- size dimensione, in bytes, di val
- i bytes che rappresentano val devono essere inseriti nel vettore dst, a partire dalla posizione offset
- si utilizza lo shift destro per selezionare i bytes, a partire dal più significativo
- il cast a byte del valore shiftato V restituisce gli 8 bits meno significativi di V, eliminando gli altri bits

CODIFICARE VALORI DI TIPO INTERO

```
public static void main(String args[   ])
    {byte byteVal=101, short shortVal = 8, int intVal =53,
    long  longVal = 234567L;
    final int BSIZE =1;
    final int SSIZE= Short.SIZE / Byte.SIZE;
    final int ISIZE = Integer.SIZE/ Byte.SIZE;
    final int LSIZE = Long.SIZE / Byte.SIZE;
    byte [ ] message = new byte[BSIZE+SSIZE+ISIZE+LSIZE];
    int offset = encodePacket(message,0, byteVal, 1);
    offset = encodePacket(message,offset, shortVal, SSIZE);
    offset = encodePacket(message,offset,intVal, ISIZE);
    offset = encodePacket(message,offset, longVal, LSIZE);
```

DECODIFICARE VALORI DI TIPO INTERO

```
public static long decodePacket(byte[ ]val,int offset,int size)
    { long rtn =0;
      int BYTEMASK = 0xFF;
      for (int i= 0; i<size; i++)
         {rtn = (rtn << 8) | ((long) val[offset+i] & BYTEMASK);}
      return rtn; }
```

- decodePacket: decodifica il valore di un dato reppresentato dai bytes contenuti nel vettore val, a partire dalla posizione offset. La dimensione (in byte) del valore da decodificare è size
- con riferimento all'esempio dellucido precedente, se si vuole 'riassemblare' il valore di tipo short:

```
short value =(short) decodePacket(message,BSIZE,SSIZE);
```

CODIFICA DI STRINGHE

- I caratteri vengono codificati in JAVA mediante Unicode che mappa i caratteri ad interi nell'intervallo [0..65535]
- Unicode è back compatibile con la codifica ASCII
- Il metodo getBytes( ) applicato ad una stringa restituisce un vettore di bytes contenente la rappresentazione della stringa ottenuta secondo la codifica utilizzata di default dalla piattaforma su cui il programma viene eseguito
- E' possibile indicare esplicitamente la codifica desiderata, come argomento della getBytes()
- Esempio.
  - "Test!".getBytes( )
  - "Test!".getBytes("UTF-16BE")
- In generale mittente e destinatario devono accordarsi sulla codifica utilizzata per i valori di tipo stringa

SERIALIZZAZIONE DI OGGETTI

- Le classi ObjectInputStream e ObjectOutputStream definiscono streams (basati su streams di byte) su cui si possono leggere e scrivere oggetti.
- La scrittura e la lettura di oggetti va sotto il nome di serializzazione, poiché si basa sulla possibilità di scrivere lo stato di un oggetto in una forma sequenziale, sufficiente per ricostruire l'oggetto quando viene riletto.
  - la serializzazione di oggetti viene usata principalmente in diversi contesti:per inviare oggetti sulla rete, sia che si utilizzino i protocolli UDO o TCP, sia che si utilizzi RMI
  - per fornire un meccanismo di persistenza ai programmi, consentendo l'archiviazione di un oggetto. Si pensi ad esempio ad un programma che realizza una rubrica telefonica o un'agenda.

SERIALIZZAZIONE DI OGGETTI

- consente di convertire un qualsiasi oggetto che implementa la interfaccia serializable in una sequenza di bytes.

- tale sequenza può successivamente essere utilizzata per ricostruire l'oggetto.

- l'oggetto deve essere definito mediante una classe che implementi l'interfaccia serializable.

- tutte le classi che definiscono tipi di dati primitivi(es: String, Double,....) implementano l'interfaccia serializable. Quindi JAVA garantisce una serializzazione di default per tutti i dati primitivi

- utilizzare stream di tipo ObjectOutputStream (rs. ObjectInputStream) e metodi writeObject (rs. readObject).

SERIALIZZAZIONE DI OGGETTI

- Un oggetto è serializzabile solo se la sua classe implementa l'interfaccia Serializable.
- Quindi se si vuole che le istanze di una classe che state scrivendo siano serializzabili, è sufficiente dichiarare che la classe implementa Serializable. Poiché questa intefaccia non ha metodi, non occorre fare altro.
- La serializzazione delle istanze di una classe viene gestita dal metodo defaultWriteObject della classe ObjectOutputStream. Questo metodo scrive automaticamente tutto ciò che è richiesto per ricostruire le istanze di una classe
- E' possibile definire anche strategie di serializzazione personalizzate e diverse da quella di default, ma per il momento non ce ne occuperemo

LA CLASSE OBJECTOUTPUTSTREAM

```
public ObjectOutputStream (OutputStream out) throws Exception
```

Quando si costruisce un oggetto di tipo ObjectOutputStream, viene automaticamente registrato in testa allo stream un header

header = costituito da due short, 4 bytes

(costanti MAGIC NUMBER+NUMERODI VERSIONE)

- Magic Number = identifica univocamente un object stream
- I Magic Number vengono utilizzati in diversi contesti. Ad esempio, ogni struttura contenente la definizione di una classe JAVA deve iniziare con un numero particolare (magic number), codificato mediante una sequenza di 4 bytes, che identificano che quella struttura contiene effettivamente una classe JAVA (CAFEBABE)
- se l'header viene cancellato lo stream risulta corrotto e l'oggetto non può essere ricostruito. Infatti al momento della ricostruzione dell'oggetto si controlla innanzi tutto che l'header non sia corrotto

LA CLASSE OBJECTOUTPUTSTREAM

```
public ObjectInputStream (InputStream in) throws Exception
```

- L'header inserito dal costruttore ObjectOutputStream viene letto e decodificato dal costruttore ObjectInputStream
- Se il costruttore ObjectInputStream( ) rileva qualche problema nel leggere l'header (ad esempio l'header è stato modificato o cancellato) viene segnalato che lo stream risulta corrotto
- L'eccezione sollevata è `StreamCorruptedException`

LA CLASSE OBJECTOUTPUTSTREAM

```
import java.io.*;
public class test {
    public static void main(String Args[ ]) throws Exception
    { ByteArrayOutputStream bout = new ByteArrayOutputStream( );
    System.out.println (bout.size( ));
    // Stampa 0
      ObjectOutputStream out= new ObjectOutputStream(bout);
     System.out.println (bout.size( ));
    // Stampa 4, l'header è stato scritto sullo stream
```

LA CLASSE OBJECTOUTPUTSTREAM

```
 out.writeObject("prova");
 //la classe String implementa l'interfaccia Serializable
System.out.println (bout.size( ));
//Stampa 12
 bout.reset ( );
 out.writeObject("prato");
 System.out.println (bout.size( ));
 //Stampa 8= 12-4.
```

….(continua pagina successiva)

IMPORTANTE

- la reset ha distrutto l'header dello stream.
- Nel momento in cui si ricostruiscono gli oggetti memorizzati sullo stream, verrà segnalata un'eccezione di tipo `StreamCorruptedException`

LA CLASSE OBJECTOUTPUTSTREAM

```
bout.reset( );
out = new ObjectOutputStream (bout);
out.writeObject ("prova");
System.out.println (bout.size( ));
```

// Stampa 12. La creazione di un nuovo stream ha ricreato l'header dello Stream

.....(continua pagina successiva)

LA CLASSE OBJECTOUTPUTSTREAM

```
bout.reset( );
out = new ObjectOutputStream (bout);
out.writeObject ("prova"); System.out.println(bout.size( ));
//stampa 12
out.writeObject("pippo");
System.out.println(bout.size(    )); // stampa 20
out.writeObject("prova");
System.out.println(bout.size(    )); // stampa 25
```

ATTENZIONE: La implementazione della classe si ricorda se un oggetto è già stato inserito nello stream ed in quel caso, non lo riscrive, ma inserisce un"riferimento" al precedente.

Questo può provocare problemi se si scrive più volte lo stesso oggetto sullo stream, modificandone lo stato. Vedremo un caso concreto nell'esempio finale di questa lezione

INVIO DI OGGETTI SULLA RETE: SERIALIZZAZIONE

Esempio: Un server ServerScuola gestisce un registro di classe. Un client può contattare il server inviandogli il nome di uno studente e riceve come risposta il numero di assenze giustificate ed il numero di assenze ingiustificate dello studente.

Il server può inviare al client una struttura con due campi interi (numero assenze giustificate, numero assenze ingiustificate)

```
public class messaggio  implements serializable
                         {private int nassgiustificate;
                          private int nassingiustificate
                           ...............
                             }
```

Nota: La versione presentata è notevolmente semplificata per mettere in evidenza i concetti principali. Sviluppare la versione completa.

SERIALIZZAZIONE DI OGGETTI

- definizione di un oggetto M contenente due interi (assenze giustificate, assenze non giustificate) come implementazione della interfaccia Serializable

- utilizzo di ObjectInput/OutputStream per la serializzazione di M. In questo modo l'oggetto viene serializzato e trasformato in una sequenza di bytes

- NOTA: non posso scivere un oggetto istanza di una classe che non implementa l'interfaccia Serializable su un OutputStream!

SERIALIZZAZIONE DI OGGETTI

Se la classe messaggio non implementasse l'interfaccia serializable,

il seguente programma

```
import java.io.*;
public class prova {
public static void main (String args[])throws Exception
{ObjectOutputStream oo= new ObjectOutputStream(System.out);
messaggio m = new messaggio(2,3);
try{oo.writeObject(m); }catch (Exception e)
                              {System.out.println(e);}}}
```

solleva la seguente eccezione:

`java.io.NotSerializableException`

INVIO DI OGGETTI

```
import java.io.*;
public class messaggio implements Serializable
  { private int nassenzeg;
    private int nassenzeng;
    public messaggio(int na, int ng)
                      {this.nassenzeg = na;
                       this.nassenzeng= ng;  }
    public int getx ( ) {return nassenzeg;};
    public int gety ( ) {return nassenzeng;}
    public void setx(int na) {nassenzeg = na;};
    public void sety(int nng){nassenzeng = nng;}  }
```

IL SERVER ASSENZE

```
import java.net.*;
import java.io.*;
import java.util.*;
public class serverassenze
{ public static void main (String Args[ ]) throws Exception
                { InetAddress ia =InetAddress.getByName("LocalHost");
                  int port= 1300;
                  DatagramSocket ds = new DatagramSocket( );
                  ByteArrayOutputStream bout=new
                                              ByteArrayOutputStream();
                  byte [ ] data=new byte[256] ;
                  DatagramPacket dp= new DatagramPacket(data,
                                            data.length, ia, port);
```

IL SERVER ASSENZE

```
for (int i=1;i<10;i++)
   {int na=i; int nr=i;
  messaggio m=new messaggio(na,nr);
  ObjectOutputStream dout = new
                        ObjectOutputStream(bout);
  dout.writeObject(m);
  dout.flush ( );
  data =bout.toByteArray();
  dp.setData(data);
  dp.setLength(data.length);
  ds.send (dp);
  bout.reset ( );
  }  }
```

IL SERVER ASSENZE

- E' necessario costruire un nuovo ObjectOutputStream per ogni oggetto inviato. Questo permette di rigenerare l'header.

- E' necessario inserire bout.reset( ) all'interno del ciclo, in modo da eliminare dallo stream i bytes relativi ad oggetti già spediti

- posso eliminare la bout.reset( ) se sposto l'istruzione

  ```
  ByteArrayOutputStream bout=new ByteArrayOutputStream( );
  ```

  all'interno del ciclo for.

- Se si sposta fuori dal ciclo l'istruzione

  ```
  ObjectOutputStream dout = new ObjectOutputStream(bout)
  ```

  il destinatario non riesce a ricostruire l'oggetto serializzato (StreamCorruptedException).

IL CLIENT ASSENZE

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

public class clientassenze{
public static void main (String Args[]) throws Exception
       { InetAddress ia = InetAddress.getByName("LocalHost");
        int port=1300;
         DatagramSocket ds=new DatagramSocket(port);
         byte buffer[ ]=new byte[256];
        DatagramPacket dpin=
                 new DatagramPacket(buffer,  buffer.length);
```

IL CLIENT ASSENZE

```
for (int i=1;i<10;i++)
      {ds.receive(dpin);
       ByteArrayInputStream bais= new
                     ByteArrayInputStream(dpin.getData ( ));
       ObjectInputStream ois= new ObjectInputStream (bais);
       messaggio m = (messaggio) ois.readObject();
       System.out.println(m.getx());
       System.out.println(m.gety());
      }}
```

Provare a vedere cosa accade se si elimina dal server l' istruzione bout.reset ( ) !!

INVIO DI PIU' OGGETTI IN UN UNICO PACCHETTO: Il SERVER

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

public class serverassenze1
{ public static void main(String Args[]) throws Exception
    {messaggio m;
     InetAddress ia = InetAddress.getByName("LocalHost" );
     int port = 6500;
     DatagramSocket ds=new DatagramSocket( );
     ByteArrayOutputStream bout=new ByteArrayOutputStream( );
     ObjectOutputStream oo= new ObjectOutputStream(bout);
```

INVIO DI PIU' OGGETTI IN UN UNICO PACCHETTO: IL SERVER

```
m = new messaggio (1,1);
oo.writeObject(m);
m = new messaggio (2,2);
oo.writeObject(m);
byte [ ] data=bout.toByteArray();
DatagramPacket dp= new
  DatagramPacket(data,data.length, ia, port);
ds.send(dp);
    } }
```

INVIO DI PIU' OGGETTI IN UN UNICO PACCHETTO: IL CLIENT

```
import java.io.*;
import java.net.*;
public class clientassenze1
{    public static void main (String Args[]) throws Exception
         {messaggio m;
          byte [] data = new byte[516];
          DatagramSocket ds = new DatagramSocket(6500);
          DatagramPacket dp =new DatagramPacket(data, data.length);
          ds.receive(dp);
          ByteArrayInputStream bin= new
                                      ByteArrayInputStream(data);
          ObjectInputStream oin= new ObjectInputStream(bin);
```

INVIO DI PIU' OGGETTI IN UN UNICO PACCHETTO: IL CLIENT

```
    m=(messaggio) oin.readObject();
    System.out.println(m.getx());
    System.out.println(m.gety());
    m=(messaggio) oin.readObject();
    System.out.println(m.getx());
    System.out.println(m.gety());
      }
}
```

Il programma stampa correttamente i valori dei due oggetti ricevuti:

(1,1) (2,2)

INVIO DI PIU' OGGETTI IN UN UNICO PACCHETTO: IL SERVER

Modifichiamo il server come segue:

```
m = new messaggio (1,1);
oo.writeObject (m);
m.setx(2); m.sety(3);
oo.writeObject(m);
byte [ ] data=bout.toByteArray();
DatagramPacket dp= new DatagramPacket(data,data.length, ia,
                                       port);
ds.send(dp);
```

- Il client stampa (1,1), (1,1), perché l'oggetto riferito è lo stesso nelle due writeObject( )
- Il server può resettare l'ObjectOutputStream (oo.reset( )) per annullare lo stato dell'oggetto

BLOCCHI SINCRONIZZATI

- Sincronizzazione di un blocco di codice

```
synchronized (obj)
    { // blocco di codice
    }
```

- L'oggetto obj può essere quello su cui è stato invocato il metodo che contiene il codice (this) oppure un altro oggetto
- Il thread che esegue il blocco sincronizzato deve acquisire la lock su l'oggetto obj
- La lock viene rilasciata nel momento in cui il thread termina l'esecuzione del blocco (es: return, throw, esecuzione dell'ultima istruzione del blocco)
- La lock non viene rilasciata nel caso in cui si invochi un altro metodo all'interno del blocco di codice (lock rientrante)

BLOCCHI SINCRONIZZATI

- Utilizzo:
  - ridurre la lunghezza di una sezione critica
  - rendere indivisibile un frammento di codice che invoca due o più metodi synchronized
- L'esempio presentato nei lucidi successivi mostra in modo operativo il vantaggio di diminuire la lunghezza di una sezione critica

OTTIMIZZAZIONE DI SEZIONI CRITICHE

```
double A,B.....
public void setValues (int x, double r)
    double tempA= // questa elaborazione richiede molto
                        tempo...
    double tempB= // questa elaborazione richiede molto
                         tempo...
    synchronized ( this ) {
                  A = tempA;
                  B = tempB; }
```

- L'elaborazione che richiede molto tempo viene eseguita fuori dalla sezione critica ed i risultati vengono assegnati a variabili locali
- la sezione critica contiene solo l'aggiornamento delle variabili di istanza del metodo
- si riduce il tempo in cui un thread che ha acquisito la lock rimane all'interno della sezione critica

OTTIMIZZAZIONE DI SEZIONI CRITICHE

La classe Pair incapsula

- una coppia di valori interi (x,y)
- un valore intero accesscount.

La coppia di valori deve essere aggiornata in modo atomico e in mutua esclusione rispetto all'aggiornamento della variabile accesscount

```
public class Pair {
    private int x,y;
    private int accesscount=0;
    public Pair (int x, int y)
                {this.x=x;this.y=y;};
    public synchronized void accessincrement( )
                {accesscount++;};
    public synchronized int accessreturn( )
                {return accesscount;};
```

OTTIMIZZAZIONE DI SEZIONI CRITICHE

Per l'aggiornamento della coppia di valori, si definiscono due diversi metodi

- increment_syn_method( ): utilizza un metodo sincronizzato
- increment_syn_block( ): utilizza un blocco sincronizzato
- Nota Bene: la sleep simula un'operazione computazionalmente costosa

```
public synchronized void increment_syn_method(  )
    {x++;  y++;
    try {Thread.sleep(100);} catch(Exception e)  {  };
     }

public void increment_syn_block()
    {synchronized(this)
            {x++;  y++;}
    try {Thread.sleep(100);} catch(Exception e)  {  };
    }
```

OTTIMIZZAZIONE DI SEZIONI CRITICHE

```
public class Thread_Syn_Method  extends Thread
{Pair p;
public Thread_Syn_Method(Pair p) {this.p=p;};
  public void run(   )
             {while (true)  {p.increment_syn_method(  );
            }   }   }


public class Thread_Syn_Block extends Threadm{
 Pair p;
 public Thread_Syn_Block(Pair p) {this.p=p;};
 public void run(   )
              {while (true)
  {p.increment_syn_block();   }    }     }
```

OTTIMIZZAZIONE DI SEZIONI CRITICHE

```
public class CountAccess extends Thread {
        Pair p;
        public CountAccess(Pair p)
                {this.p=p;}
        public void run( )  {
            while (true)
            p.accessincrement( );
                                        }      }
```

OTTIMIZZAZIONE SEZIONI CRITICHE

```
public class MainPair {
public static void main(String args[])
    {Pair syn_method_pair= new Pair(1,2);
    Pair syn_block_pair= new Pair(3,4);
    new Thread_Syn_Method(syn_method_pair).start( );
    new CountAccess(syn_method_pair).start( );
    new Thread_Syn_Block(syn_block_pair).start( );
    new CountAccess(syn_block_pair).start( );
    try {Thread.sleep(200);} catch(Exception e)  {  }
    System.out.println( "Accessi con blocco sincronizzato
      "+syn_block_pair.accessreturn( )
       + " Accessi con metodo sincronizzato
      "+syn_method_pair.accessreturn( ));  }}
```

OTTIMIZZAZIONE SEZIONI CRITICHE

Risultati di alcune esecuzioni:

Accessi con blocco sincronizzato 2258106 Accessi con metodo sincronizzato 2843

Accessi con blocco sincronizzato 2242580 Accessi con metodosincronizzato 2

Accessi con blocco sincronizzato 3749759 Accessi con metodo sincronizzato 1252

Conclusioni:

- il thread che accede all'oggetto coppia con blocco sincronizzato (invece che con un metodo sincronizzato) consente un maggior numero di accessi al thread Countaccess, che viene eseguito concorrentemente
- La versione che utilizza il blocco sincronizzato aumenta il numero di accessi concorrenti effettuati sull'oggetto coppia
- NOTA BENE: La differenza nel numero di accessi dipende ovviamente dal fatto che il programma simula la computazione costosa con una sleep( ). Se si sincronizza tutto il metodo, quando il thread esegue la sleep rilascia il processore, ma non la lock().

COLLEZIONI SINCRONIZZATE

- JAVA Collections: strutture dati predefinite incluse nel package java.util

  a partire da JAVA 1.2
- Alcuni esempi: HashTables, Vectors
- Synchronized Collections: Definiscono strutture dati thread safe, cioè garantiscono che lo stato della struttura risulti corretto anche nel caso in cui la struttura venga acceduta in modo concorrente da più threads
- Thread Safety: la struttura dati viene incapsulata e ogni metodo pubblico viene sincronizzato
- Se l'operazione che il client (il thread che utilizza la collezione) è composta da una serie di operazioni elementari, il client deve spesso implementare ulteriori sincronizzazioni.
- Si possono definire blocchi di codice sincronizzati che incapsulano più operazioni elementari effettuate su una collezione sincronizzata

COLLEZIONI SINCRONIZZATE

```
import java.util.*;
public class threadremover extends Thread {
        Vector v;
        public threadremover(Vector v)
            {this.v=v;}
        public void run( )
            {int lastIndex = v.size( )- 1;
             Object o=v.remove(lastIndex); } }
```

- il thread threadremover elimina da un Vector l'elemento che si trova nella sua ultima posizione
- Le operazioni size( ) e remove( ) sono definite come operazioni sincronizzate

COLLEZIONI SINCRONIZZATE

```
import java.util.Vector;
public class threadget extends Thread{
    Vector v;
    public threadget(Vector v){this.v=v;}
        public void run(  ){
        int lastIndex = v.size( ) -1;
        try {Thread.sleep(5000);}
        catch(InterruptedException e){  };
        v.get(lastIndex);   } }
```

- Il thread threadget restituisce l'ultimo elemento del Vector
- La sleep( ) è stata introdotta per costringere threadget a rilasciare il processore a threadremover
- In questo modo si ottiene un interleaving scorretto tra i due threads ed il programma segnala la seguente eccezione

COLLEZIONI SINCRONIZZATE

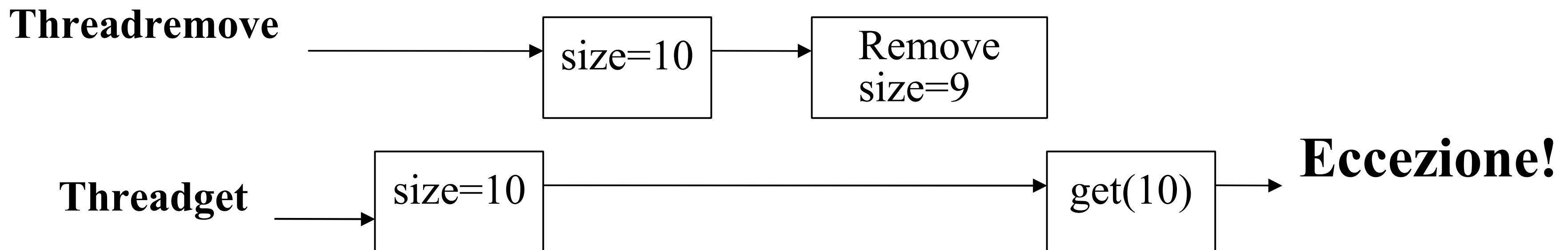
Eccezione sollevata dalla esecuzione del programma:

Exception in thread "Thread-0" java.lang.ArrayIndexOutOfBoundsException: Array index out of range: 9

at java.util.Vector.get(Unknown Source)

at threadget.run(threadget.java:10)

L'eccezione viene sollevata a causa del seguente interleaving (scorretto)



COLLEZIONI SINCRONIZZATE

```
import java.util.Vector;
public class threadget extends Thread {
    Vector v;
    public threadget(Vector v){this.v=v;}
    public void run ( ){
        synchronized(v) {
            int lastIndex = v.size( ) -1;
            Object x=v.get(lastIndex);
            System.out.println("oggetto prelevato"+x);
                        }}}
```

COLLEZIONI SINCRONIZZATE

Versione corretta del threadremover

```
import java.util.*;
public class threadremover extends Thread{
    Vector v;
    public threadremover(Vector v){this.v=v;}
    public void run( )
        {
        synchronized(v) {
            int lastIndex = v.size( )-1;
            Object o=v.remove(lastIndex);} }
        }
```

BLOCCHI DI CODICE SYNCHRONIZED

```
public class Coda { //....
    public synchronized boolean isSpaceAvailable( )
   { //....
    public synchronized void add(Item i){ //....
    public synchronized Item remove( ){ //....
    ......
    Coda c =//....
         syncronized (c)
              { if (c.isSpaceAvailable()) {
                        c.add(i); }}
```

- Se un thread entra nel blocco sincronizzato acquisisce la lock su c. Quando entra in un metodo sincronizzato, possiede già la lock e non la deve quindi richiedere nuovamente.

CODE BLOCCANTI (O SINCRONIZZATE)

- Code bloccanti: introdotte in JAVA 5 come supporto per il paradigmi computazionali di tipo produttore/consumatore
- Code sincronizzate:comportamento di base
  - inserimento: aggiunge un elemento infondo alla cosa, se la coda non è piena, altrimenti blocca il thread che ha invocato l'operazione.
  - rimozione:elimina il primo elemento della coda, se questo esiste, altrimenti blocca il thread che ha invocato l'operazione
- L'interfaccia java.util.concurrent.BlockingQueue definisce questo tipo di code
- Sono stati definiti
  - diverse varianti della coda
  - metodi caratterizzati da diversi comportamenti per l'inserzione/rimozione di elementi dalla coda

CODE BLOCCANTI: TIPI DEFINITI

Classi che implementano l'interfaccia BLockingQueue

- LinkedBlockingQueue: non si definisce un limite superiore alla capacità della coda
- ArrayBlockingQueue: definisce un numero fisso di posizioni della coda
- SynchronousQueue: non è una vera e propria coda, in quanto non ha capacità di memorizzazione. Mantiene solo le code per la gestione dei threads che aspettano in attesa di produrre/consumare elementi. Risparmia il tempo necessario per la bufferizzazione degli elementi prodotti
- PriorityBlockingQueue:implementa una coda a priorità

CODE BLOCCANTI: METODI

Metodi che generano un'eccezione quando si tenta di aggiungere un elemento ad una coda piena o di estrarre un elemento da una coda vuota:

add, remove, element (element restituisce l'elemento in testa, e non lo rimuove)

```
import java.util.concurrent.*;
public class provaqueue {
public static void main (String args[])
{BlockingQueue <String> queue=new ArrayBlockingQueue <String>(1);
queue.add("el1"); queue.add("el2");   }}
```

```
Exception in thread "main" java.lang.IllegalStateException: Queue full
at java.util.AbstractQueue.add(Unknown Source)
at java.util.concurrent.ArrayBlockingQueue.add(Unknown Source)
at procacrawler.main(procacrawler.java:7)
```

CODE BLOCCANTI: METODI

Metodi che restituiscono una segnalazione del fallimento dell'operazione di inserzione/estrazione: offer,poll, peek

```
import java.util.concurrent.*;
public class procacrawler {
public static void main (String args[])
{BlockingQueue <String> queue = new ArrayBlockingQueue<String>(1);
 boolean success=false;
 success=queue.offer("el1");  System.out.println(success);
 try{success=queue.offer("el2", 100, TimeUnit.MILLISECONDS);
    }catch (InterruptedException e){ }
 System.out.println(success);}}
```

Ouput del programma true, false

CODE BLOCCANTI:METODI

Metodi che bloccano il thread nel caso del fallimento della operazione di inserzione/rimozione di un elemento

put, take

```
import java.util.concurrent.*;
public class procacrawler {
public static void main (String args[ ])
{BlockingQueue <String> queue = new ArrayBlockingQueue <String>(1)
 try{queue.put("el1");} catch (InterruptedException e){ }
 try{queue.put("el2");} catch (InterruptedException e){ } }}
```

la seconda put blocca il thread. Il thread viene risvegliato quando l'inserzione/rimozione può essere effettuata

ESERCIZIO

Si vuole realizzare un programma Crawler che analizza tutti i files presenti in una directory specificata e nelle sue sottodirectory e visualzza tutte le righe presenti in tali file che contengono una determinata parola chiave P.

Il programma deve attivare due thread,

- un thread produttore che enumera tutti i file e inserisce un riferimento ad ogni file individuato in una coda bloccante
- un consumatore che estrae i riferimenti ai file dalla coda e, per ognuno di essi, visualizza tutte le righe che contengono la parola chiave P.

I due thread si scambiano i dati mediante una coda bloccante. Scegliere il tipo di coda bloccante ritenuto più opportuno.