Lezione n.5
LPR-A
UDP: COSTRUZIONE DI PACCHETTI

27/10/2008

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

```
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:

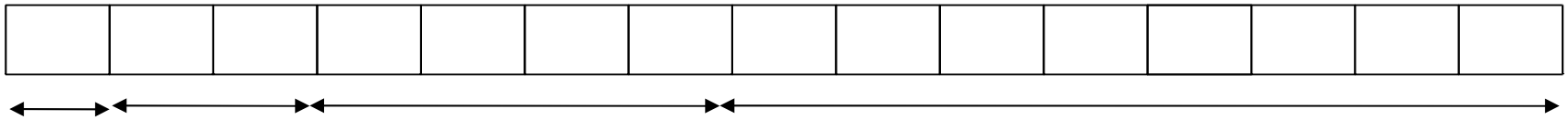
- dimensione dei tipi di dati scambiati. Long: 8 bytes, int: 4 bytes, short: 2 bytes
- ordine dei bytes trasmessi
  - Little-endian: il primo byte trasmesso è il meno significativo
  - Big-endian: il primo byte trasmesso è il più significativo
- Interpretazione dei valori: con/senza segno

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