Lezione n.6
LPR -INFORMATICA APPLICATA
SOCKETS UDP

7/4/2008

Laura Ricci

RIASSUNTO DELLA PRESENTAZIONE

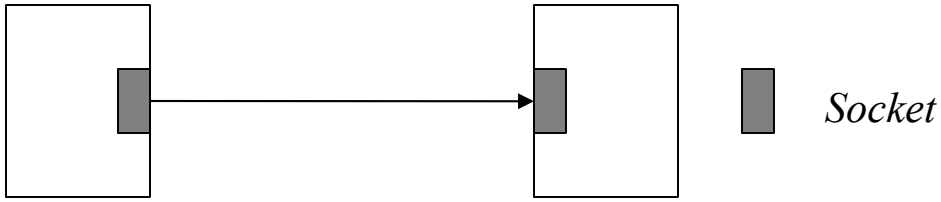
- Meccanismi di comunicazione interprocess (IPC)
- Sockets UDP
- JAVA:Le classi DatagramSocket e DatagramPacket
- Invio di oggetti su sockets UDP

JAVA IPC: I SOCKETS

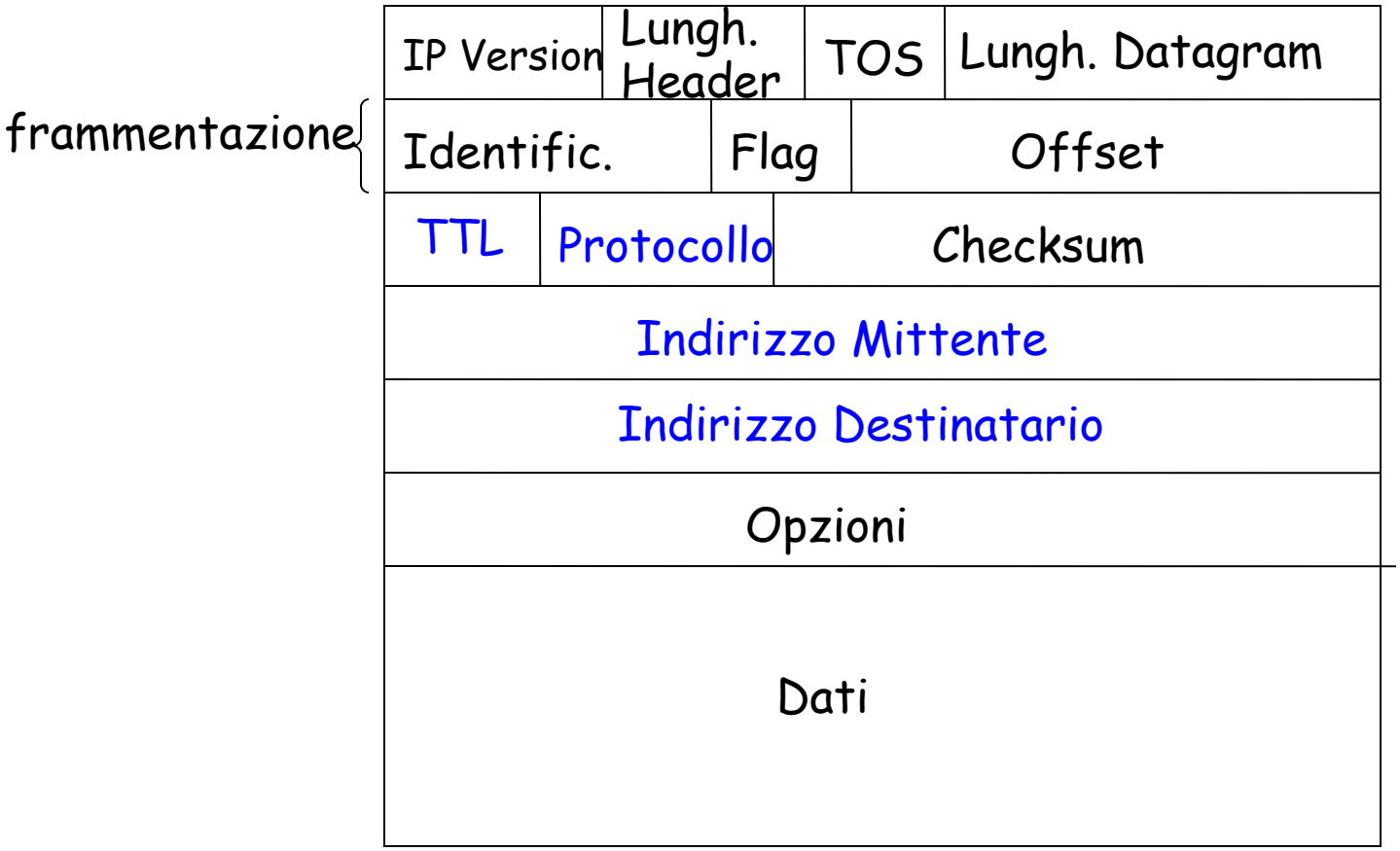
Socket = presa di corrente

Termine utilizzato in tempi remoti in telefonia. La connessione tra due utenti veniva stabilita tramite un operatore che inseriva fisicamente i due estremi di un cavo in due ricettacoli (*sockets*), ognuno dei quali era assegnato ai due utenti.

Socket è una astrazione che indica una "presa " a cui un processo si può collegare per spedire dati sulla rete. Al momento della creazione un socket viene collegato ad una porta.



FORMATO DEL DATAGRAM IP



LIVELLO IP: FORMATO DEL PACCHETTO

IP Version: *IPV4 / IPV6*

TOS (Type of Service) Consente un trattamento differenziato dei pacchetti.
Esempio: un particolare valore di TOS indica che il pacchetto ha una priorità maggiore rispetto agli altri, Utile per distinguere per distinguere tipi diversi di traffico (traffico real time, messaggi per la gestione della rete,..)

TTL – Time to Live
Consente di limitare la diffusione del pacchetto sulla rete

- valore iniziale impostato dal mittente
- quando il pacchetto attraversa un router, il valore viene decrementato
- quando il valore diventa 0, il pacchetto viene scartato

Introdotto per evitare *percorsi circolari* infiniti del pacchetto. Utilizzato anche per limitare la diffusione del pacchetto nel multicast

LIVELLO IP: FORMATO DEL PACCHETTO

- Protocol: Il valore di questo campo indica il protocollo a livello trasporto utilizzato (es: TCP o UDP). Consente di interpretare correttamente l'informazione contenuta nel datagram e costituisce l'interfaccia tra livello IP e livello di trasporto
- Frammentazione: Campi utilizzati per gestire la frammentazione e la successiva ricostruzione dei pacchetti
- Checksum: per controllare la correttezza del pacchetto
- Indirizzo mittente/destinatario

HEADER UDP

Porta sorgente (0-65535)	Porta Destinazione(0-65535)
Lunghezza	Checksum
Dati	

HEADER UDP

- L'header UDP viene inserito in testa al pacchetto IP
- Contiene 4 campi, ognuno di 2 bytes
- I numeri di porta (0-65536) mittente/destinazione consentono un servizio di multiplexing/demultiplexing
- Demultiplexing: l'host che riceve il pacchetto UDP riesce a consegnare i dati al corretto processo applicativo
- Checksum: si riferisce alla verifica di correttezza delle 4 parole di 16 bits dell'header
- Lunghezza massima teorica del datagram UDP = 65507. In pratica, sono ammessi pacchetti di lunghezza inferiore, a seconda dei sistemi operativi
- Lunghezza consigliata del segmento dati < 512 bytes

JAVA : TRASMISSIONE PACCHETTI UDP

Trasmissione di pacchetti UDP:

- mittente e destinatario devono creare due diversi sockets attraverso i quali avviene la comunicazione.
- il mittente collega il suo socket ad una porta PM, il destinatario collega il suo socket ad una porta PD

Per spedire un pacchetti UDP, il mittente

- crea un datagram socket SM collegato a PM
- crea un pacchetto DP (datagram).
- invia il pacchetto DP sul socket SM

Ogni pacchetto UDP spedito dal mittente deve contenere:

- indirizzo IP dell'host su cui è in esecuzione il destinatario + porta PD
- riferimento ad un vettore di bytes che contiene il valore del messaggio.

JAVA : COMUNICAZIONE UDP

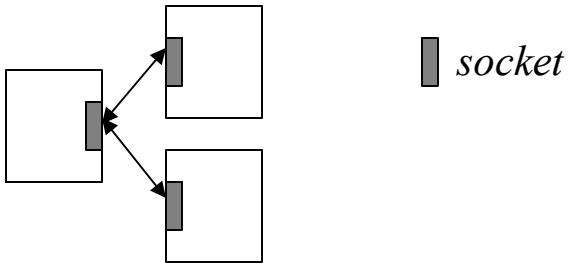
Il destinatario, per ricevere un pacchetto UDP

- crea un datagram socket SD collegato a PD
- crea una struttura adatta a memorizzare il pacchetto ricevuto
- riceve un pacchetto dal socket SD e lo memorizza in una struttura locale
  - i dati inviati mediante UDP devono essere rappresentati come vettori di bytes
  - JAVA offre diversi tipi di filtri per generare streams di bytes a partire da dati strutturati/ad alto livello

JAVA : COMUNICAZIONE UDP

Caratteristiche dei sockets UDP

- il destinatario deve "pubblicare" la porta a cui è collegato il socket di ricezione, affinchè il mittente possa spedire pacchetti su quella porta
- non è in genere necessario pubblicare la porta a cui è collegato il socket del mittente
- un processo può utilizzare lo stesso socket per spedire pacchetti verso destinatari diversi
- processi diversi possono spedire pacchetti sullo stesso socket allocato da un processo destinatario



JAVA : LA CLASSE DATAGRAM SOCKET

public class DatagramSocket **extends Object**

Costruttori:

public DatagramSocket ( ) **throws SocketException**

- crea un socket e lo collega ad una porta anonima (o effimera), il sistema sceglie una porta non utilizzata e la assegna al socket. Per reperire la porta allocata utilizzare il metodo *getLocalPort().*

- utilizzato generalmente da chi inizia la trasmissione.

- Esempio: un client si connette ad un server mediante un socket collegato ad una porta anonima. Il server invia la risposta sullo stesso socket, ⇒ preleva l'indirizzo del mittente (IP+porta) dal pacchetto ricevuto. Quando il client termina la porta viene utilizzata per altre connessioni.

JAVA : LA CLASSE DATAGRAM SOCKET

public class DatagramSocket **extends Object**
Costruttori:

public DatagramSocket (**int** p) **throws SocketException**

- crea un socket sulla porta specificata (p).
- viene sollevata un'eccezione quando la porta è già utilizzata, oppure se si tenta di connettere il socket ad una porta su cui non si hanno diritti.
- utilizzato da chi attende una comunicazione.
- Esempio: il server crea un socket collegato ad una porta resa nota ai clients. Di solito la porta viene allocata permanentemente a quel servizio (porta non effimera)

INDIVIDUAZIONE DELLE PORTE LIBERE

Un programma per individuare le porte libere su un host:

```
import java.net.*;
public class scannerporte {
public static void main(String args[ ])
    { for (int i=1; i<1024; i++)
                {try  {
                 DatagramSocket s =new DatagramSocket(i);
                System.out.println ("Porta libera"+i);
                      }
      catch (BindException e) {System.out.println ("porta già in uso")  ;}
      catch (Exception e) {System.out.println (e);}
     } }
```

JAVA : LA CLASSE DATAGRAMPACKET

public final class DatagramPacket **extends Object**
public DatagramPacket (**byte**[ ] data, **int** length, **InetAddress** destination,
int port)

- costruttore utilizzato dal mittente

- il messaggio deve essere trasformato in una sequenza di bytes e memorizzato nel vettore data (strumenti necessari per la traduzione, es: metodo getBytes ( ), la classe java.io.ByteArrayOutputStream)

- length indica il numero di bytes da prelevare dal vettore data per costruire il pacchetto

- Il byte array data viene passato al costruttore per riferimento, per cui se si modifica il contenuto dell'array, viene modificato il contenuto del Datagram Packet
- destination+port individuano il destinatario

JAVA: LA CLASSE DATAGRAMPACKET

public final class DatagramPacket **extends Object**

public DatagramPacket (**byte**[ ] buffer, **int** length)

- definisce la struttura utilizzata per memorizzare il pacchetto ricevuto. Quindi è utilizzato dal destinatario.
- il buffer viene passato vuoto alla receive che lo riempie al momento della ricezione di un pacchetto.
- il payload del pacchetto (la parte che contiene i dati) viene copiato nel buffer al momento della ricezione.
- la copia del payload termina quando l'intero pacchetto è stato copiato oppure, se la lunghezza del pacchetto è maggiore di length, quando length bytes sono stati copiati

JAVA: GENERAZIONE DEI PACCHETTI

Metodi per la conversione stringhe/vettori di bytes

- Byte [ ] getBytes( ) applicato ad un oggetto String, restituisce una sequenza di bytes che codificano i caratteri della stringa usando la codifica di default dell'host e memorizza il risultato in un vettore di Bytes

- String (byte[ ] bytes, int offset, int length) costruisce un nuovo oggetto di tipo String prelavando length bytes dal vettore bytes, a partire dalla posizione offset

JAVA : INVIARE E RICEVERE PACCHETTI

Invio di pacchetti

- sock.***send*** (dp)

  dove: *sock* è il socket attraverso il quale voglio spedire il pacchetto *dp*

Ricezione di pacchetti

- sock.***receive***(buffer)

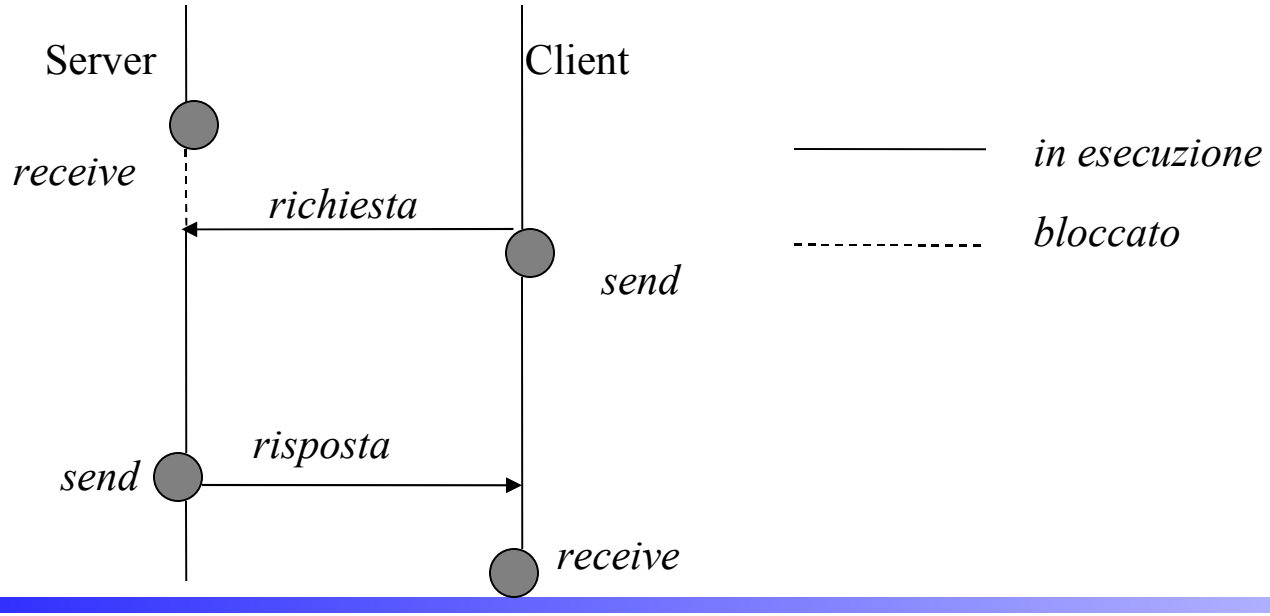
dove sock è il socket attraverso il quale ricevo il pacchetto e buffer è la struttura in cui memorizzo il pacchetto ricevuto

COMUNICAZIONE TRAMITE SOCKETS: CARATTERISTICHE

send non bloccante = il processo che esegue la send prosegue la sua esecuzione, senza attendere che il destinatario abbia ricevuto il pacchetto

receive bloccante = il processo che esegue la receive si blocca fino al momento in cui viene ricevuto un pacchetto.

per evitare attese indefinite è possibile associare al socket un timeout. Quando il timeout scade, viene sollevata una InterruptedIOException



RECEIVE CON TIMEOUT

- SO_TIMEOUT: proprietà associata al socket, indica l'intervallo di tempo, in millisecondi, di attesa di ogni receive eseguita su quel socket
- Nel caso in cui l'intervallo di tempo scada, prima che venga ricevuto un pacchetto dal socket, viene sollevata una eccezione di tipo InterruptedException
- Metodi per la gestione di time out

  public synchronized void setSoTimeout(int timeout) **throws SocketException**

  Esempio: se ds è un datagram socket,

  ds.setSoTimeout(30000)}

  associa un timeout di 30 secondi al socket ds.

PER ESEGUIRE IL PROGRAMMA SU UN UNICO HOST

- Attivare il client ed il server in due diverse shell
- Se l'host è connesso in rete: utilizzare come indirizzo IP del mittente/destinatario l'indirizzo dell'host su cui sono in esecuzione i due processi (reperibile con getLocalHost())
- Se l'host non è connesso in rete utilizzare l'indirizzo di loopback
- Tenere presente che mittente e destinatario sono in esecuzione sulla stessa macchina ⇒ devono utilizzare porte diverse
- Mandare in esecuzione per primo il server, poi il client

UTILIZZO DI SOCKETS UDP

Esercizio:

Scrivere un'applicazione composta da un processo Sender ed un processo Receiver. Il Sender riceve da linea di comando una stringa, l'indirizzo del receiver (indirizzo IP+porta) ed invia al Receiver la stringa. Il Receiver riceve il messaggio e lo visualizza.

Considerare poi i seguenti punti:

- cosa cambia se mando in esecuzione prima il Sender, poi il Receiver rispetto al caso in cui mando in esecuzione prima il Receiver?
- nel processo Receiver, aggiungere un time-out sulla receive, in modo che la receive non si bocchi per più di 5 secondi. Cosa accade se attivo il receiver, ma non il sender?

UTILIZZO DI SOCKETS UDP

- modificare il codice del Sender in modo che usi lo stesso socket per inviare lo stesso messaggio a due diversi receivers. Mandare in esecuzione prima i due Receivers, poi il Sender. Cosa accade?
- modificare il codice del Sender in modo che esso usi due sockets diversi per inviare lo stesso messaggio a due diversi receivers. Mandare in esecuzione prima i due Receivers, poi il Sender. Cosa accade?
- modificare il codice ottenuto al passo precedente in modo che il Sender invii una sequenza di messaggi ai Receivers. Ogni messaggio contiene il valore della sua posizione nella sequenza. Il Sender si sospende per 3 secondi tra un invio ed il successivo. Ogni receiver deve essere modificato in modo che esso esegua la receive in un ciclo infinito. Cosa accade?
- modificare il codice ottenuto al passo precedente in modo che il Sender non si sospenda tra un invio e l'altro. Cosa accade?
- modificare il codice iniziale in modo che il Receiver invii al Sender un ack quando riceve il messaggio. Il Sender visualizza l'ack ricevuto.

INVIO DI OGGETTI SU CONNESIIONI UDP

- Per inviare oggetti su sockets UDP è necessario costruire un pacchetto di bytes a partire dall'oggetto che si vuole inviare
- L'oggetto deve essere serializzabile
- Si possono utilizzare le classi

  ByteArrayInputStream

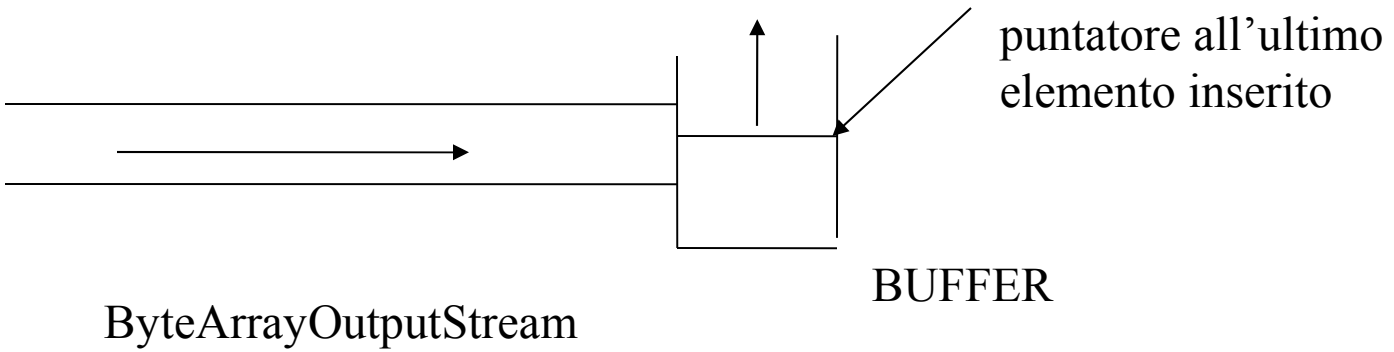
  ByteArrayOutputStream

BYTE ARRAY INPUT/OUTPUT STREAMS NELLA COSTRUZIONE DI PACCHETTI UDP

public ByteArrayOutputStream ( )

public ByteArrayOutputStream (**int** size)

- gli oggetti di questa classe rappresentano stream di bytes tali che ogni dato scritto sullo stream viene riportato in un buffer di memoria a dimensione variabile (dimensione di default = 32 bytes).
- quando il buffer si riempie la sua dimensione viene raddoppiata automaticamente



BYTE ARRAY INPUT/OUTPUT STREAMS NELLA COSTRUZIONE DI PACCHETTI UDP

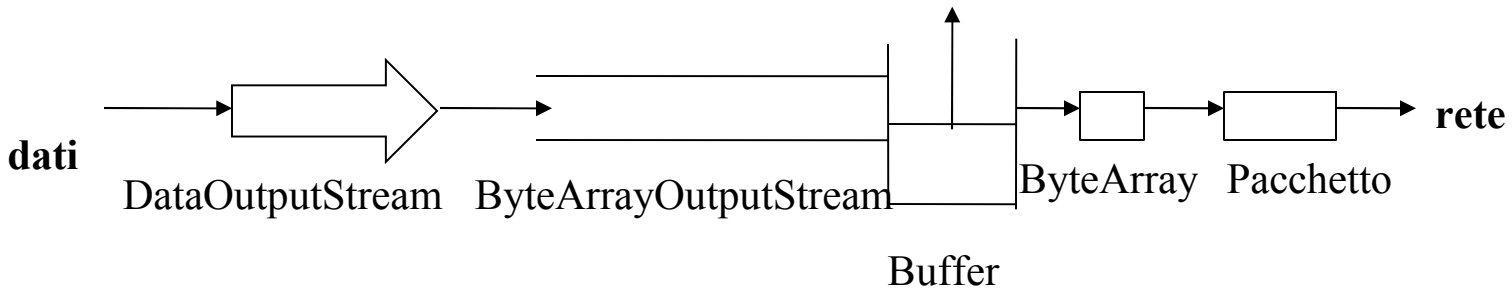
- ad un ByteArrayOutputStream può essere collegato un altro filtro

```
DataOutputStream do= new DataOutputStream(
                         new ByteArrayOutputStream( ))
```

- i dati presenti nel buffer B associato ad un ByteArrayOuputStream baos possono essere copiati in un array di bytes, di dimensione uguale alla dimensione attuale di B

```
byte [ ] barr = baos. toByteArray( )
```

Creazione di un pacchetto UDP a partire da dati di qualsiasi tipo

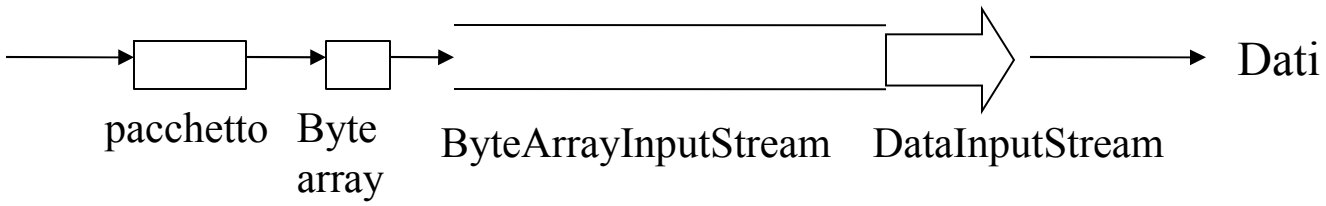


BYTE ARRAY INPUT/OUTPUT STREAMS

public ByteArrayInputStream (**byte** [ ] buf)
public ByteArrayInputStream (**byte** [ ] buf, **int** offset, **int** length)

- creano stream di byte a partire dai dati contenuti nel vettore di byte buf.
- il secondo costruttore copia length bytes iniziando alla posizione offset.
- E' possibile concatenare un DataInputStream

Ricezione di un pacchetto UDP dalla rete:



BYTE ARRAY INPUT/OUTPUT STREAMS

- Le classi ByteArrayInput/OutputStream facilitano l'invio dei dati di qualsiasi tipo (anche oggetti) sulla rete. La trasformazione in sequenza di bytes è automatica.

- uno stesso ByteArrayOuput/InputStream può essere usato per produrre streams di bytes a partire da dati di tipo diverso

- il buffer interno associato ad un ByteArrayOutputStream baos viene svuotato (puntatore all'ultimo elemento inserito = 0) con

  - baos.reset ( )

  - il metodo toByteArray non svuota il buffer!

BYTE ARRAY INPUT/OUTPUT STREAMS

Ipotesi semplificativa: non consideriamo perdita/riordinamento di pacchetti

```
import java.io.*;
import java.net.*;
public class multidatastreamsender{
public static void main(String args[ ]) throws Exception
        {// fase di inizializzazione
         InetAddress ia=InetAddress.getByName("localhost");
         int port=13350;
         DatagramSocket ds= new DatagramSocket (  );
         ByteArrayOutputStream bout= new ByteArrayOutputStream( );
         DataOutputStream dout = new DataOutputStream (bout);
         byte [ ] data = new byte [20];
         DatagramPacket dp= new DatagramPacket(data, data.length, ia , port);
```

BYTE ARRAY INPUT/OUTPUT STREAMS

```
for (int i=0; i< 10; i++)
   {dout.writeInt(i);
   data = bout.toByteArray();
   dp.setData(data,0,data.length);
   dp.setLength(data.length);
   ds.send(dp);
   bout.reset( );
   dout.writeUTF("***");
   data = bout.toByteArray( );
   dp.setData (data,0,data.length);
   dp.setLength (data.length);
   ds.send (dp);
   bout.reset( ); } } }
```

BYTE ARRAY INPUT/OUTPUT STREAMS

Ipotesi semplificativa: non consideriamo perdita/riordinamento di pacchetti

```
import java.io.*;
import java.net.*;
public class multidatastreamreceiver
        {public static void main(String args[ ]) throws Exception
              {// fase di inizializzazione
                FileOutputStream fw = new FileOutputStream("text.txt");
                DataOutputStream dr = new DataOutputStream(fw);
                int port =13350;
                DatagramSocket ds = new DatagramSocket (port);
                byte [ ] buffer = new byte [200];
                DatagramPacket dp= new DatagramPacket
                                                (buffer, buffer.length);
```

BYTE ARRAY INPUT/OUTPUT STREAMS

```
for (int i=0; i<10; i++)
    {ds.receive(dp);
     ByteArrayInputStream bin= new   ByteArrayInputStream
                                          (dp.getData(),0,dp.getLength());
    DataInputStream ddis= new DataInputStream(bin);
    int x = ddis.readInt();
    dr.writeInt(x);
    System.out.println(x);
    ds.receive(dp);
    bin= new ByteArrayInputStream(dp.getData(), 0 ,dp.getLength());
    ddis= new DataInputStream(bin);
    String y=ddis.readUTF( );
    System.out.println(y);
    } } }
```

BYTE ARRAY INPUT/OUTPUT STREAMS

- Nel programma precedente, la corrispondenza tra la scrittura nel mittente e la lettura nel destinatario potrebbe non essere più corretta

- Esempio:

  - il mittente alterna la spedizione di pacchetti contenenti valori interi con pacchetti contenenti stringhe

  - il destinatario alterna la lettura di interi e di stringhe

  - se un pacchetto viene perso ⇒ il destinatario scritture/letture possono non corrispondere

- Realizzazione di UDP affidabile: utilizzo di ack per confermare la ricezione + identificatori unici

LA CLASSE BYTEARRAYOUTPUTSTREAM

Implementazione della classe Byte ArrayOutputStream

- Definisce una struttura dati

```
protected byte buf [ ]
protected int count
```

buf memorizza i bytes che vengono scaricati sullo stream

count indica quanti sono i bytes memorizzati in buf

- Costruttori

ByteArrayOutputStream ( ): crea un buf di 32 bytes (ampiezza di default)

ByteArrayOutputStream (int size): crea un buf di dimensione size

LA CLASSE BYTEARRAYOUTPUTSTREAM

- Ogni volta che un byte viene scritto sull'oggetto ByteArrayOutputStream( ), il byte viene automaticamente memorizzato nel buffer
- Se il buffer risulta pieno, la sua lunghezza viene automaticamente raddoppiata
- Il risultato è che si ha l'impressione di scrivere su uno stream di lunghezza non limitata (stream)
- Metodi per la scrittura sullo stream
  - **public synchronized void** write (**int** b)
  - **public synchronized void** write (**byte** b [ ], **int** off, **int** len)

LA CLASSE BYTEARRAYOUTPUTSTREAM

Metodi per la gestione dello stream

- **public int** size( ) restituisce count, cioè il numero di bytes memorizzati nello stream (NON la lunghezza del vettore buf!)
- **public synchronized void** reset( ) assegna 0 a count. In questo modo lo stream risulta vuoto e tutti i dati precedentemente scritti vengono eliminati.
- **public synchronized byte** toByteArray ( ) restituisce un vettore in cui sono stati copiati tutti i bytes presenti nello stream. Non modifica count, per cui lo stream NON viene resettato

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 una serializzazione di default è garantita 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

```
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
```

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

```
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 implementi 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);}}}
```

solleverebbe 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 ( ) !!