Laboratorio di Reti
Lezione 9
UDP: Datagram Sockets e Channels

19/11/2020

Laura Ricci

TCP ED UDP: CONFRONTO

- in certi casi TCP offre "più di quanto sia necessario"
  - non interessa garantire che tutti i messaggi vengano recapitati
  - si vuole evitare l'overhead dovuto alla ritrasmissione dei messaggi
  - non è necessario leggere i dati nell'ordine con cui sono stati spediti
- UDP supporta una comunicazione connectionless e fornisce un insieme molto limitato di servizi, rispetto a TCP:
  - aggiunge un ulteriore livello di indirizzamento a quello offerto dal livello IP, quello delle porte.
  - offre un servizio di scarto dei pacchetti corrotti.
- uno slogan per indicare quando utilizzare UDP:

  " *Timely, rather than orderly and reliable delivery*"

QUANDO USARE UDP

- stream video/audio: meglio perdere un frame che introdurre overhead nella trasmissione di ogni frame

- tutti gli host di un ufficio inviano, ad intervalli di tempo brevi e regolari, un keep-alive ad un server centrale
  - la perdita di un keep alive non è importante
  - non è importante che il messaggio spedito alle 10:05 arrivi prima di quello spedito alle 10:07

- compravendita di azioni:
  - le variazioni di prezzo tracciate in uno "stock ticker"
  - la perdita di una variazione di prezzo può essere tollerata per titoli azionari di minore importanza
  - il prezzo deve essere controllato al momento della compra/vendita

CONNECTION ORIENTED VS. CONNECTIONLESS

JAVA socket API: interfacce diverse per UDP e TCP

- TCP: Stream Sockets
  - occorre connettere il socket al server per aprire una connessione
- UDP: DatagramSocket
  - un socket non deve essere connesso ad un altro socket prima di essere utilizzato
    - come una lettera
      - non è richiesto che mi colleghi a qualcuno prima di inviare una lettera,
      - piuttosto devo specificare l'indirizzo del destinatario per ogni lettera spedita
  - in UDP, ogni messaggio, chiamato Datagram, è indipendente dagli altri e porta l'informazione per il suo instradamento

CONNECTION ORIENTED VS. CONNECTIONLESS

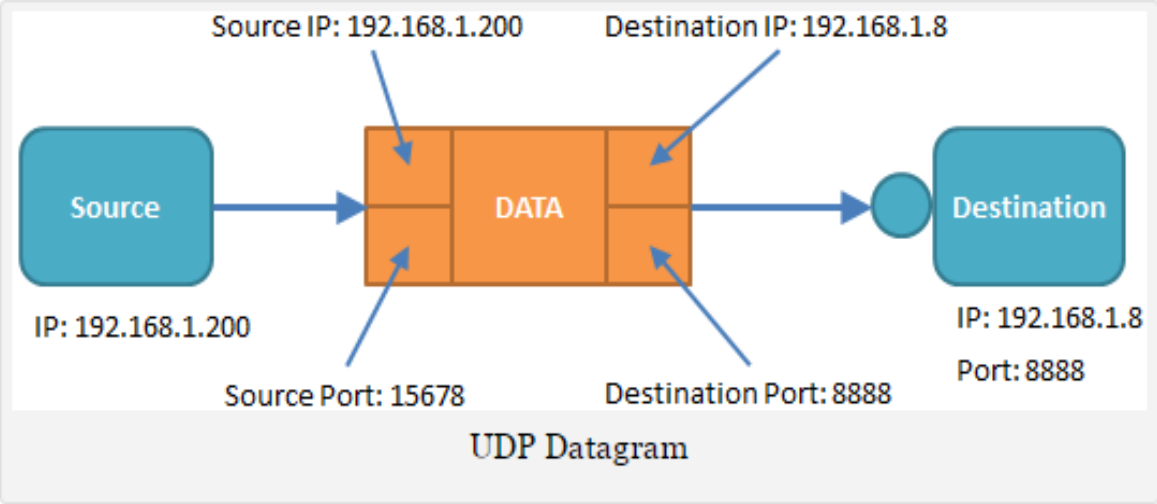
JAVA socket API: interfacce diverse per UDP e TCP

- TCP: trasmissione vista come uno stream continuo di bytes provenienti dallo stesso mittente

- UDP: trasmissione orientata ai messaggi: "preserves message boundaries"
  - send, riceve `DatagramPacket`
  - socket come una mailbox: in essa possono essere inseriti messaggi in arrivo da diverse sorgenti (mittenti) o i messaggi inviati a diverse destinazioni
  - ogni ricezione si riferisce ad un singolo messaggio inviato mediante una unica send.
    - dati inviati dalla stessa send non possono essere ricevuti in receive diverse

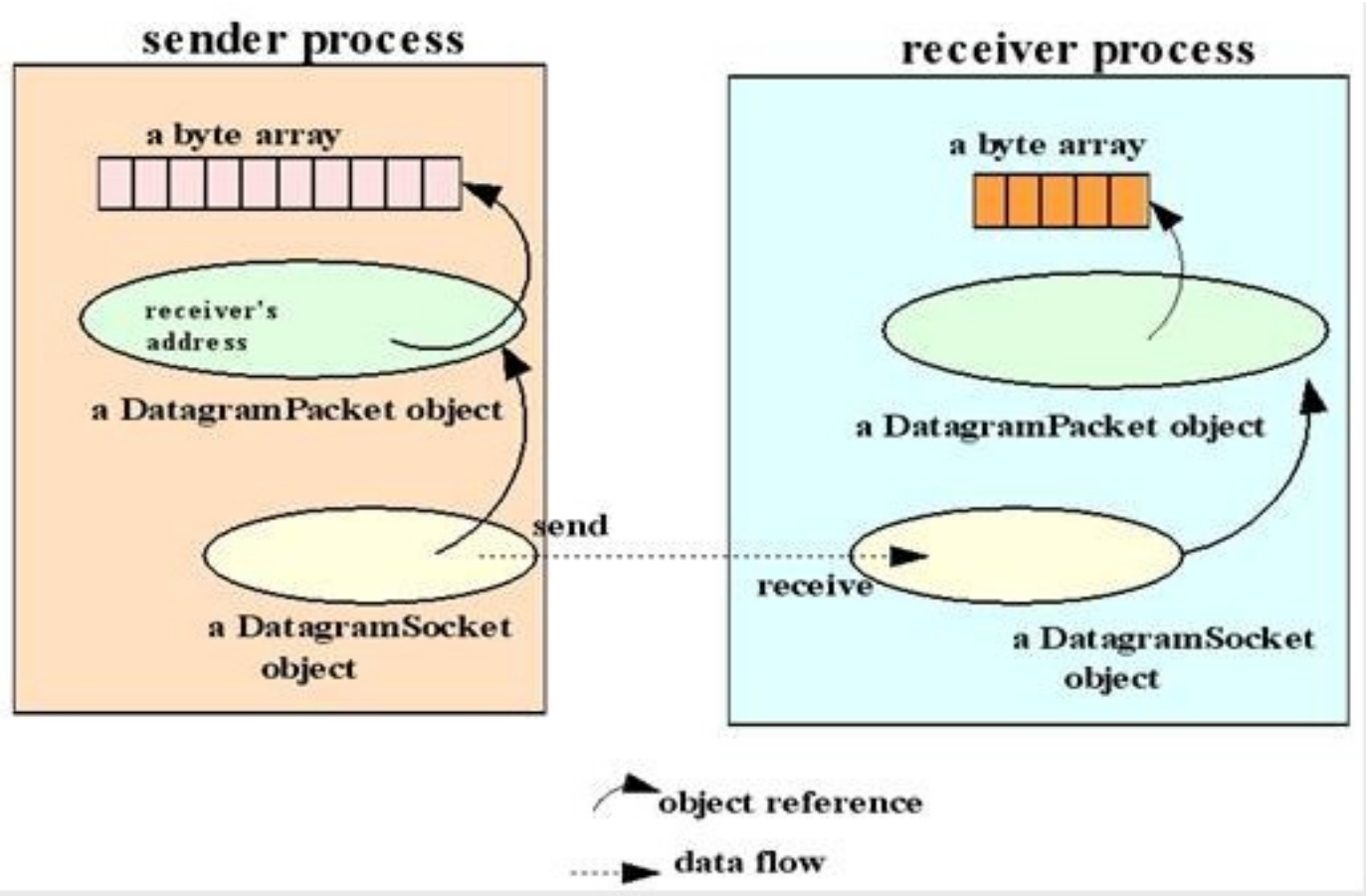
STRUTTURA DEL DATAGRAM

`Datagram`: un messaggio indipendente, self-contained in cui l'arrivo ed il tempo di ricezione non sono garantiti, modellato in JAVA come un `DatagramPacket`



- il mittente deve inizializzare
  - il campo `DATA`
  - destination `IP` e destination `port`
- source IP inserito automaticamente, source port scelta automaticamente in modo casuale

UDP IN JAVA


sender process
a byte array
receiver's address
a DatagramPacket object
send
a DatagramSocket object
receiver process
a byte array
a DatagramPacket object
receive
a DatagramSocket object
object reference
data flow

JAVA DATAGRAMSOCKET API

- Classi per la gestione di UDP:
  - `DatagramSocket` per creare i sockets.
  - `DatagramPacket` per costruire i datagram

- un processo mittente che desidera ricevere/inviare dati su UDP, deve istanziare un oggetto di tipo `DatagramSocket`, collegato ad una porta locale
  - il processo mittente collega il suo socket ad una porta PM
    - può essere una porta effimera, non è necessario pubblicarla

- il destinatario "pubblica" la porta a cui è collegato il socket di ricezione, affinchè il mittente possa spedire pacchetti su quella porta

  - il processo destinatario collega il suo socket ad una porta PD

UDP IN BREVE

Inviare un datagramma:

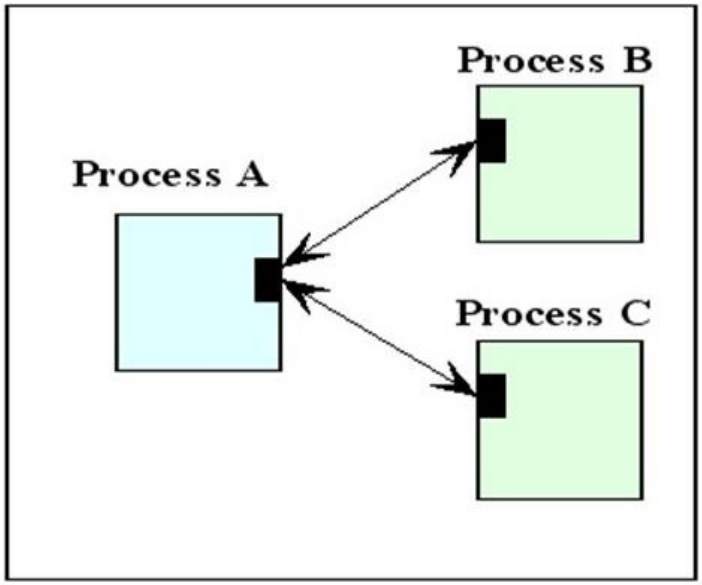
- creare un `DatagramSocket` e collegarlo ad una porta, anche effimera
- creare un oggetto di tipo `DatagramPacket`, in cui inserire
  - un riferimento ad un byte array contenente i dati da inviare nel payload del datagramma
  - indirizzo IP e porta del destinatario nell'oggetto creato
- inviare il `DatagramPacket` tramite una send invocata sull'oggetto `DatagramSocket`

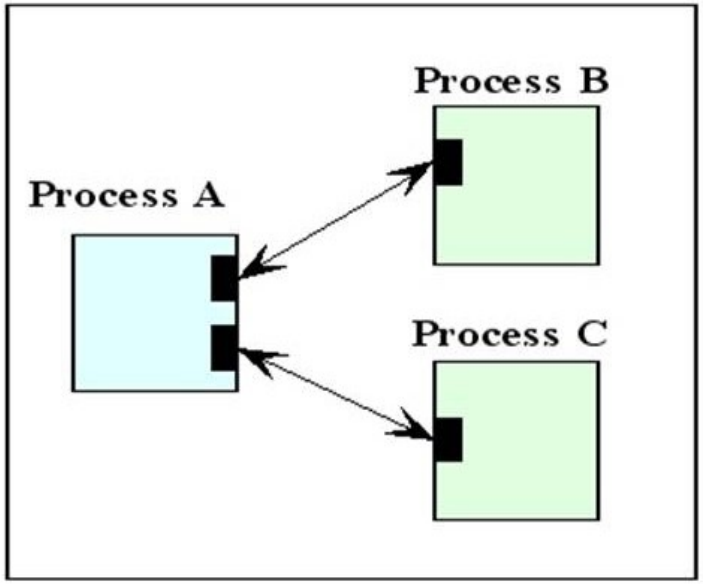
Ricevere un datagramma:

- creare un `DatagramSocket` e collegarlo ad una porta pubblicata (che corrisponde a quella specificata dal mittente nel pacchetto)
- creare un `DatagramPacket` per memorizzare il pacchetto ricevuto. Il `DatagramPacket` contiene un riferimento ad un byte array che conterrà il messaggio ricevuto.
- invocare una receive sul `DatagramSocket` passando il `DatagramPacket`

CARATTERISTICHE SOCKET UDP

- un processo può utilizzare lo stesso socket per spedire pacchetti verso destinatari diversi
- processi (applicazioni) diverse possono spedire pacchetti sullo stesso socket allocato dal destinatario: in questo caso l'ordine di arrivo dei massaggi è non deterministico, in accordo con il protocollo UDP
- ...ma anche utilizzare socket diversi per comunicazioni diverse





JAVA DATAGRAM SOCKETS

```
public class DatagramSocket extends Object
    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.
- utilizzato generalmente lato client, per spedire datagrammi
- per reperire la porta allocata utilizzare il metodo `getLocalPort( )`
- esempio:
  - un client si connette ad un server mediante un socket collegato ad una porta anonima.
  - il server preleva l'indirizzo del mittente (IP+porta) dal pacchetto ricevuto e può così inviare una risposta.
  - quando il socket viene chiuso, la porta viene utilizzata per altre connessioni.

JAVA SOCKETS UDP

```
public class DatagramSocket extends Object
        public DatagramSocket(int p) throws SocketException
```

- utilizzato in genere lato server.
- crea un socket sulla porta specificata (`int p`).
- solleva 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.
- esempio,
  - il server crea un socket collegato ad una porta che rende nota ai clients.
  - di solito la porta viene allocata permanentemente a quel servizio (porta non effimera)

INDIVIDUAZIONE 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=1024; i<.....; 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);}
         }  }
```

DATAGRAMPACKET: COSTRUTTORI

```
DatagramPacket(byte[ ] buffer, int length)
DatagramPacket(byte[ ] buffer, int offset, int length)
DatagramPacket(byte[ ] buffer, int length, InetAddress remoteAddr,
               int remotePort)
DatagramPacket(byte[ ] buffer, int offset, int length,
               InetAddress remoteAddr, int remotePort)
```

- costruttori diversi per invio/ricezione di pacchetti
- in generale, un oggetto `DatagramPacket` contiene:
  - in ogni caso un riferimento ad un vettore di byte buffer che contiene i dati da spedire oppure quelli ricevuti
  - eventuali informazioni di addressing, se il DatagramPacket deve essere spedito
    - informazioni sul destinatario
    - indirizzo IP e porta.

COSTRUTTORI PER INVIARE DATI

```
public DatagramPacket(byte[] buffer,int length,InetAddress destination,int port)
```

- costruttore per un `DatagramPacket` da inviare
- length indica il numero di bytes che devono essere copiati dal byte buffer nel pacchetto IP, a partire dal byte 0 o da offset, se indicato
  - solleva un aeccezione se `length` è maggiore di `buffer.length`
  - il byte buffer può contenere più di length bytes, ma questi non verranno spediti sulla rete
- destination + port individuano il destinatario
- molti altri costruttori sono disponibili
- notare che, per essere memorizzato nel buffer, il messaggio deve essere trasformato in una sequenza di bytes. Per generare vettori di bytes:
  - il metodo `getBytes( )`
  - la classe `java.io.ByteArrayOutputStream`

COSTRUTTORI PER RICEVERE DATI

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

- definisce la struttura utilizzata per memorizzare il pacchetto ricevuto.
- il buffer viene passato vuoto alla receive che lo riempie al momento della ricezione di un pacchetto, con il payload del pacchetto
- se settato offset, la copia avviene nella posizioni individuata da esso
- il parametro length
  - indica il numero massimo di bytes che possono essere copiati nel buffer
  - deve essere minore di `buffer.length`, altrimenti viene sollevata eccezione
- la copia del payload termina quando
  - l'intero pacchetto è stato copiato
  - se la lunghezza del pacchetto è maggiore di length, quando length bytes sono stati copiati
  - `getLength` restituisce il numero di bytes effettivamente copiati

DECIDERE LA DIMENSIONE DEL DATAGRAMPACKET

- ad ogni socket sono associati due buffers: uno per la ricezione ed uno per la spedizione, gestiti dal sistema operativo, non dalla JVM

```
import java.net.*;
public class udpproof {
public static void main (String args[])throws Exception
        {DatagramSocket dgs = new DatagramSocket( );
         int r = dgs.getReceiveBufferSize();
         int s = dgs.getSendBufferSize();
         System.out.println("receive buffer"+r);
         System.out.println("send buffer"+s); } }
```

- stampa prodotta : **receive buffer 8192 send buffer 8192**
- in generale la dimensione massima di un pacchetto UDP è 64k bytes, ma in molte piattaforme è 8k
- pacchetti più grandi vengono in generale troncati
- safety: DatgramPacket minori di 512 bytes

DATAGRAMPACKET: METODI GET

```
byte[ ] getData ()
```

- restituisce un riferimento all'intero buffer associato più recentemente al DatagramPacket
  - mediante invocazione del costruttore o
  - mediante il metodo `setData()`.
- ignora offset e lunghezza.
- può provocare problemi nel caso in cui il buffer abbia dimensioni maggiori dell'effettivo dato ricevuto (vedi esempi nelle slide seguenti)

```
public int getLength()
```

- restituisce la lunghezza dei dati ricevuti

DATAGRAMPACKET: METODI SET

```
void setData(byte[ ] buffer)
void setData(byte[ ] buffer, int offset, int length)
```

- i dati nel pacchetto da spedire possono essere inseriti mediante il costruttore o mediante il metodo SetData
  - utile quando si deve mandare una grossa quantità di dati

```
int offset = 0;
DatagramPacket dp = new DatagramPacket(bigarray, offset, 512);
int bytesSent = 0;
while (bytesSent < bigarray.length) {
 socket.send(dp);
 bytesSent += dp.getLength( );
 int bytesToSend = bigarray.length - bytesSent;
 int size = (bytesToSend > 512) ? 512 : bytesToSend;
 dp.setData(bigarray, bytesSent, 512);
```

JAVA: GENERAZIONE DEI PACCHETTI

- i dati inviati mediante UDP devono essere rappresentati come vettori di bytes
- alcuni 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 li memorizza nel vettore
  - `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
- altri meccanismi per generare pacchetti a partire da dati strutturati:
  - utilizzare i filtri per generare streams di bytes a partire da dati strutturati/ad alto livello

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)`

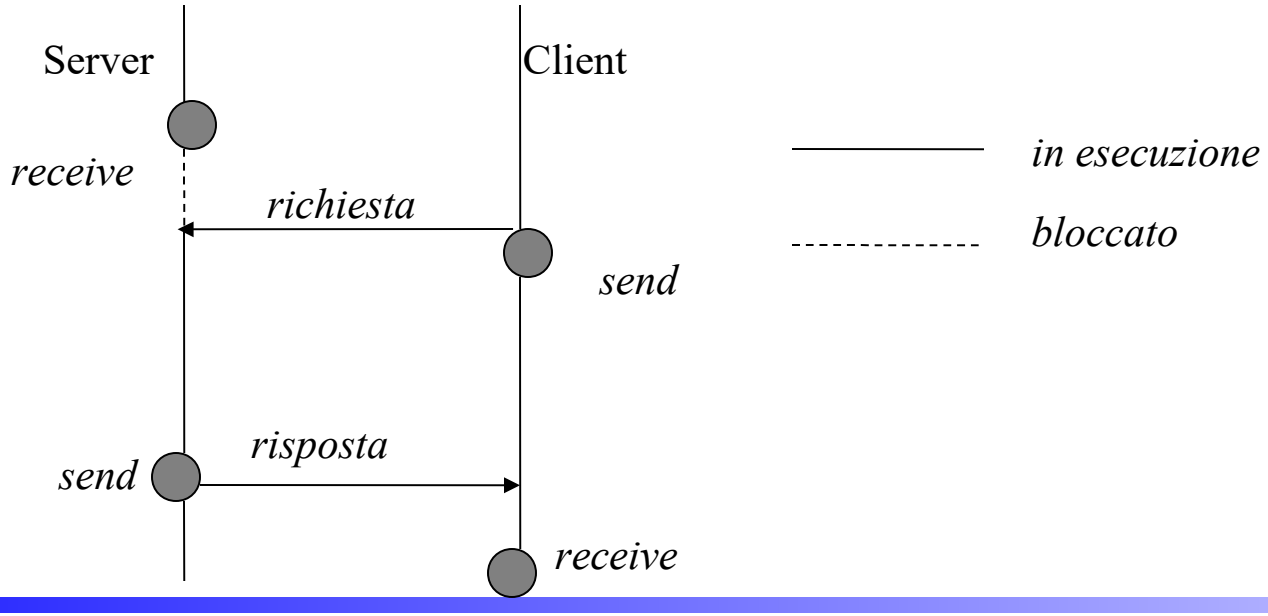
dove sock è il socket attraverso il quale ricevo il pacchetto e buffer è la struttura in cui memorizzo il pacchetto ricevuto

COMUNICAZIONE UDP: CARATTERISTICHE

send non bloccante nel senso che 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.

"GIOCARE" CON DATAGRAMPACKET

```
import java.net.*;
public class Sender
 { public static void main (String args[]) {
    try
     {DatagramSocket clientsocket = new DatagramSocket();
      byte[] buffer="1234567890abcdefghijklmnopqrstuvwxyz".getBytes("US-
                                                                ASCII");
      InetAddress address = InetAddress.getByName("Localhost");
      for (int i = 1; i < buffer.length; i++) {
          DatagramPacket mypacket = new DatagramPacket(buffer,i,address,
                                                       40000);
          clientSocket.send(mypacket);
          Thread.sleep(200); }
          System.exit(0);}
    catch (Exception e) {e.printStackTrace();}}}
```

"GIOCARE" CON DATAGRAMPACKET

```
import java.net.*;
public class Receiver
  {public static void main(String args[]) throws Exception {
    DatagramSocket serverSock= new DatagramSocket(40000);
    byte[] buffer = new byte[100];
    DatagramPacket receivedPacket = new DatagramPacket(buffer, buffer.length);

    while (true) {
        serverSock.receive(receivedPacket);
        String byteToString = new String(receivedPacket.getData(),
                              0, receivedPacket.getLength(), "US-ASCII");
        System.out.println("Length " + receivedPacket.getLength() +
             " data " + byteToString);
    }
} }
```

"GIOCARE" CON DATAGRAMPACKET

Dati inviati dal mittente:

Length 1 data 1

Length 2 data 12

Length 3 data 123

Length 4 data 1234

…….

Length 34 data 1234567890abcdefghijklmnopqrstuvwx

Length 35 data 1234567890abcdefghijklmnopqrstuvwxy

Modificando la dimensione del buffer, nel receiver, i dati ricevuti sono:

```
byte[] buffer= new byte[5]; otteniamo:
```

Length 1 data 1

Length 2 data 12

Length 3 data 123

Length 4 data 1234

Length 5 data 12345

Length 5 data 12345

Length 5 data 12345 ….

….

"GIOCARE" CON DATAGRAMPACKET

```
import java.net.*;
public class Sender
   { public static void main (String args[]) {
     try
     {DatagramSocket clientsocket = new DatagramSocket();
     byte[] buffer="1234567890abcdefghijklmnopqrstuvwxyz".getBytes("US-
                                                                ASCII");
      InetAddress address = InetAddress.getByName("Localhost");
      for (int i = buffer.length; i >0; i--) {
          DatagramPacket mypacket = new DatagramPacket(buffer,i,address,
                                                          40000);
          clientSocket.send(mypacket);
          Thread.sleep(200); }
          System.exit(0);}
     catch (Exception e) {e.printStackTrace();}}}
```

"GIOCARE" CON DATAGRAMPACKET

Dati inviati dal mittente:

```
Length 36 data 1234567890abcdefghijklmnopqrstuvwxyz
Length 35 data 1234567890abcdefghijklmnopqrstuvwxy
Length 34 data 1234567890abcdefghijklmnopqrstuvwx
Length 33 data 1234567890abcdefghijklmnopqrstuvw
Length 32 data 1234567890abcdefghijklmnopqrstuv
Length 31 data 1234567890abcdefghijklmnopqrstu

……..

Length 5 data 12345
Length 4 data 1234
Length 3 data 123
Length 2 data 12
Length 1 data 1
```

"GIOCARE" CON DATAGRAMPACKET

Cosa accade se il destinatario non utilizza offset e length?

```
import java.net.*;
public class Receiver
{public static void main(String args[]) throws Exception {
    DatagramSocket serverSock= new DatagramSocket(40000);
    byte[] buffer = new byte[100];
    DatagramPacket receivedPacket = new DatagramPacket(buffer,
                                            buffer.length);
    while (true) {
        serverSock.receive(receivedPacket);
        String byteToString = new String(receivedPacket.getData(),"US-
                                                              ASCII");
        int l=byteToString.length();
        System.out.println(l);
        System.out.println("Length " + receivedPacket.getLength() +
                    " data " + byteToString);}}}
```

"GIOCARE" CON DATAGRAMPACKET

```
100
Length 36 data 1234567890abcdefghijklmnopqrstuvwxyz
100
Length 35 data 1234567890abcdefghijklmnopqrstuvwxyz
100
Length 34 data 1234567890abcdefghijklmnopqrstuvwxyz
100
Length 33 data 1234567890abcdefghijklmnopqrstuvwxyz
…..
100
Length 2 data 1234567890abcdefghijklmnopqrstuvwxyz
100
Length 1 data 1234567890abcdefghijklmnopqrstuvwxyz
```

SOCKETS: "UNDER THE HOOD"

```
class TestSocket {
    public static void main(String[] args) throws IOException {
        DatagramSocket ds = new DatagramSocket(50000,
                                    InetAddress.getByName("localhost"));
        final DatagramPacket dp = new DatagramPacket(new byte[1024], 1024);
         //thread used to try to access the packet's data asynchronously
        new Thread() {
            public void run() {
                while (true)
                   { try { sleep(1000); } catch (InterruptedException e)
                                    { e.printStackTrace(); }
                     System.out.println("Will try to call getData on dp");
                     dp.getData(); //should block
                     System.out.println("getData ran");
                   }
                } }.start();
              ds.receive(dp); }}
```

SOCKETS: "UNDER THE HOOD"

- l'idea alla base del programma precedente è che il programma principale si blocca su una receive, mentre un thread controlla, in modo asincrono, il contenuto del buffer di ricezione

- ma....l'istruzione System.*out*.println("getData ran"); non viene mai eseguita: quale è il motivo di questo comportamento?
  - la receive acquisisce una lock sul `DatagramPacket` e non la rilascia fino a che la receive non è completata
  - poichè non c'è alcun nodo che invia messaggi sul socket, la lock non viene rilaciata e la `getData( )` rimane bloccata un attesa di acquisire la lock.

DATI STRUTTUATI IN PACCHETTI UDP

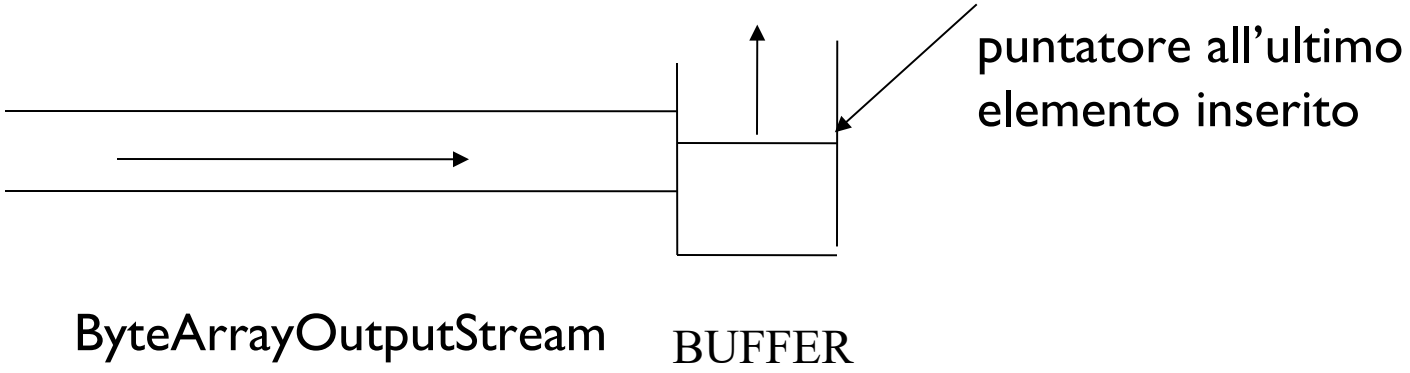
```
public ByteArrayOutputStream ( )
public ByteArrayOutputStream (int size)
```

- gli oggetti istanze di questa classe rappresentano stream di bytes
- ogni dato scritto sullo stream viene riportato in un buffer di memoria a dimensione variabile (dimensione di default = 32 bytes).

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

count indica quanti sono i bytes memorizzati in buf

- quando il buffer si riempie la sua dimensione viene raddoppiata automaticamente



BYTE ARRAY OUTPUT STREAMS

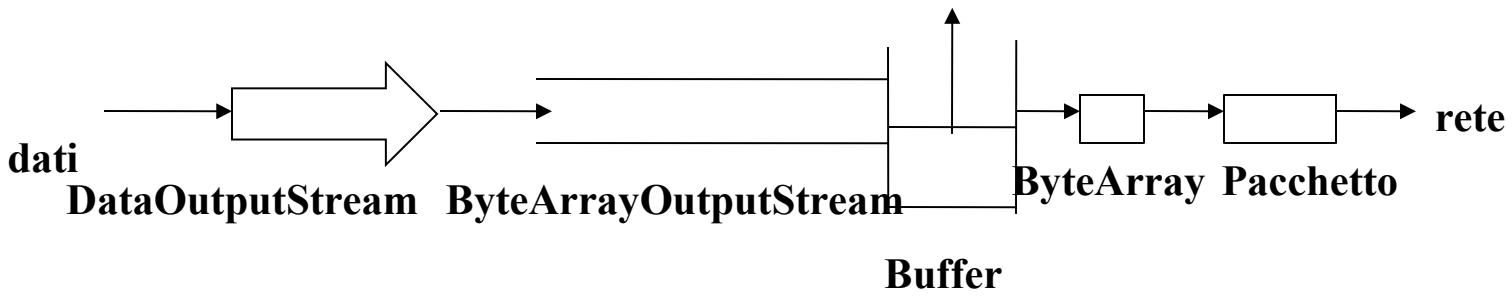
- ad un ByteArrayOutputStream può essere collegato un altro filtro

```
ByteArrayOutputStream baos= new ByteArrayOutputStream ( );
DataOutputStream      dos = new DataOutputStream (baos)
```

- i dati presenti nel buffer B associato ad un ByteArrayOuputStream baos possono essere copiati in un array di bytes

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

Flusso dei dati:

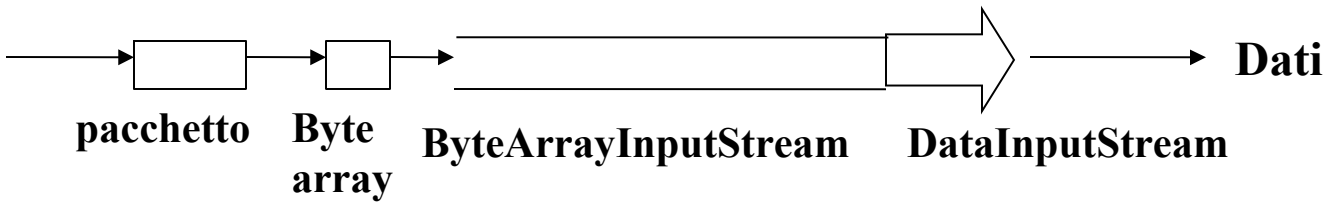


BYTE ARRAY INPUT STREAMS

```
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.
- è possibile concatenare un `DataInputStream`

Flusso dei dati:



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( )` svuota il buffer, assegnando 0 a count. Tutti i dati precedentemente scritti vengono eliminati.

```
baos.reset ( )
```

- `public synchronized byte toByteArray ( )` restituisce un vettore in cui sono stati copiati tutti i bytes presenti nello stream.

  - non modifica count

  - 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
  - ma 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

SERIALIZZAZIONE E DATAGRAM:"UNDER THE HOOD"

```
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( ));
       // Anche se non ho scritto niente sulla stream, stampa 4 ,
          l'header è stato scritto sullo stream !!
       out.writeObject("ciao mondo");
       byte[] b=bout.toByteArray();
       ByteArrayInputStream bin = new ByteArrayInputStream(b);
       ObjectInputStream oin = new ObjectInputStream(bin);
       String s = (String) oin.readObject();
       System.out.println(s);
```

SERIALIZZAZIONE E DATAGRAM:"UNDER THE HOOD"

```
 // Stampa "ciao mondo"
bout.reset();
//riflettere sul fatto di inserire o meno l'istruzione
 precedente!
out.writeObject("ciao ciao");
b = bout.toByteArray();
bin = new ByteArrayInputStream(b);
oin = new ObjectInputStream(bin);
s = (String) oin.readObject();
System.out.println(s);
s = (String) oin.readObject();
System.out.println(s);
Exception in thread "main" java.io.StreamCorruptedException:
 invalid stream header: 74000963
at java.io.ObjectInputStream.readStreamHeader(Unknown Source)
at java.io.ObjectInputStream.<init>(Unknown Source)
at Test.main(Test.java:20) }}
```

SERIALIZZAZIONE E DATAGRAM:"UNDER THE HOOD"

- se elimino l'istruzione `bout.reset()`, ottengo la seguente stampa:

```
0
4
ciao mondo
ciao mondo
ciaociao
```

- se inserisco l'istruzione `bout.reset()`, ottengo la seguente stampa:

```
0
4
ciao mondo
Exception in thread "main" java.io.StreamCorruptedException:
 invalid stream header: 74000963
at java.io.ObjectInputStream.readStreamHeader(Unknown Source)
at java.io.ObjectInputStream.<init>(Unknown Source)
at Test.main(Test.java:20)
```

Perchè??

SERIALIZZAZIONE E DATAGRAM:"UNDER THE HOOD"

- se non inserisco l'istruzione `bout.reset()`
  - non resetto il buffer, quando effettuo la seconda deserializzazione trovo nel buffer sia i byte che rappresentano il primo oggetto che quelli del secondo
- se inserisco l'istruzione `bout.reset()`
  - quando vado a serializzare il secondo oggetto, resetto il buffer e distruggo lo stream header. Quando lo deserializzo, la JVM segnala l'eccezione perchè non trova lo streamheader
  - ricreando l'oggetto si ricrea lo streamheader

INSERIRE PIU' DATI IN UN PACCHETTO

Utilizzare lo stesso **ByteArrayOuput/InputStream** per produrre streams di bytes a partire da dati di tipo diverso

```
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();
```

UDP E STREAMS: RIFLESSIONI

- Trasmissione connection oriented: una connessione viene modellata con uno stream.

  invio di dati scrittura sullo stream

  ricezione di dati lettura dallo stream

- Trasmissione connectionless: stream utilizzati per la generazione dei pacchetti:

  `ByteArrayOutputStream`, consentono la conversione di uno stream di bytes in un vettori di bytes da spedire con i pacchetti UDP

  `ByteArrayInputStream`, converte un vettore di bytes in uno stream di byte.

UDP CHANNELS

```
DatagramChannel channel = DatagramChannel.open();
channel.socket().bind(new InetSocketAddress(9999));
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
// Prepare buffer for reading
channel.receive(buf);
...
```

- modalità bloccante: la receive si blocca fino a che non è stato ricevuto un `DatagramPacket`
- come la receive di `DatagramSocket`, ma trasferimento più efficiente tramite buffer
- `buf.clear( )`
  - re-inizializza `Position` e `Limit`

UDP CHANNELS

```
DatagramChannel channel = DatagramChannel.open();
channel.socket().bind(new InetSocketAddress(9999));
String newData = "New String to write to file..."
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());
buf.flip();
int bytesSent = channel.send(buf, new InetSocketAddress("www.google.it", 80));
```

- la operazione di write si blocca fino non c'è spazio disponibile nel send buffer
- `buf.flip()` prepara il buffer alla lettura. Ciò che ha scritto l'applicazione viene letto dal supporto.

DATAGRAM CHANNEL: READ/PUT NON BLOCCANTI

```
ByteBuffer buffer = ByteBuffer.allocate(8192);
DatagramChannel channel= DatagramChannel.open();
channel.configureBlocking(false);
channel.socket().bind(new InetSocketAddress(9999));
SocketAddress address = new InetSocketAddress("localhost",7);
buffer.put(…);
buffer.flip();
while (channel.send(buffer, address) == 0);
        //do something useful …
buffer.clear();
while ((address = channel.receive(buffer))==null);
     //do something useful …
```

- se la send restituisce 0, ciò indica che il send-buffer è pieno
- se la receive restituisce null, non è stato ricevuto alcun Datagram
- il programma può eseguire altre operazioni, nell'attesa che il canale sia disponible per l'operazione

DATAGRAMCHANNEL: MULTIPLEXING

- possibilità di registrare il canale con un selettore
- invocazione del metodo `Selector.select` per testare la "`readibility`" o "`writability`" del canale.

Operation	Meaning
OP_READ	Data is present in the socket receive-buffer or an exception is pending.
OP_WRITE	Space exists in the socket send-buffer or an exception is pending. In UDP, OP_WRITE is almost always ready except for the moments during which space is unavailable in the socket send-buffer. It is best only to register for OP_WRITE once this buffer-full condition has been detected, i.e. when a channel write returns less than the requested write length, and to deregister for OP_WRITE once it has cleared, *i.e.* a channel write has fully succeeded.

NON BLOCKING MULTIPLEXED UDP ECHO SERVER

```
import java.io.*;
import java.util.*;
import java.nio.charset.*;
public class ASyncUDPSv {
    static int BUF_SZ = 1024;
    class Con {
        ByteBuffer req;
        ByteBuffer resp;
        SocketAddress sa;
        public Con() {
            req = ByteBuffer.allocate(BUF_SZ);
                    }
         }
  static int port = 8340;
```

NON BLOCKING MULTIPLEXED UDP ECHO SERVER

```
private void process() {
        try {
            Selector selector = Selector.open();
            DatagramChannel channel = DatagramChannel.open();
            InetSocketAddress isa = new InetSocketAddress(port);
            channel.socket().bind(isa);
            channel.configureBlocking(false);
            SelectionKey clientKey = channel.register(selector,
                                                      SelectionKey.OP_READ);
            clientKey.attach(new Con());
            System.out.println("Server Pronto");
```

NON BLOCKING MULTIPLEXED UDP ECHO SERVER

```
while (true)
   { try {selector.select();
          Iterator selectedKeys = selector.selectedKeys().iterator();
           while (selectedKeys.hasNext()) {
               try { SelectionKey key = (SelectionKey) selectedKeys.next();
                     selectedKeys.remove();
                     if (!key.isValid()) { continue; }
                     if (key.isReadable()) {
                               read(key);
                               key.interestOps(SelectionKey.OP_WRITE);
                     } else if (key.isWritable()) {
                               write(key);
                               key.interestOps(SelectionKey.OP_READ); }
                       } catch (IOException e) {System.out.println(e);}
                } catch (IOException e) { System.out.println(e)); }
           }  } catch (IOException e) {System.out.println(e));} }
```

NON BLOCKING MULTIPLEXED UDP ECHO SERVER

```
private void read(SelectionKey key) throws IOException {
        DatagramChannel chan = (DatagramChannel) key.channel();
        Con con = (Con) key.attachment();
        con.sa = chan.receive(con.req);
        con.req.flip();
        System.out.println(new String(con.req.array(), "UTF-8"));
        con.resp = con.req;
    }
```

NON BLOCKING MULTIPLEXED UDP ECHO SERVER

```
private void write(SelectionKey key) throws IOException {
        DatagramChannel chan = (DatagramChannel)key.channel();
        Con con = (Con) key.attachment();
        chan.send(con.resp, con.sa);
        con.req.clear();

    }
```

NON BLOCKING MULTIPLEXED UDP ECHO SERVER

```
static public void main(String[] args) {
        ASyncUDPSv svr = new ASyncUDPSv();
        svr.process();
    }
}
```

UDP ECHO CLIENT

```
import java.io.*;
import java.net.*;
public class EchoClient
 {public static void main(String args[]) throws Exception
   { BufferedReader inFromUser =
                    new BufferedReader(new InputStreamReader(System.in));
    DatagramSocket clientSocket = new DatagramSocket();
    InetAddress IPAddress = InetAddress.getByName("localhost");
    byte[] sendData = new byte[1024];
    byte[] receiveData = new byte[1024];
     while (true)
       { String sentence = inFromUser.readLine();
          System.out.println("sentence"+sentence);
          sendData = sentence.getBytes();
```

UDP ECHO CLIENT

```
DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length,
                                                IPAddress, 8340);

clientSocket.send(sendPacket);

DatagramPacket receivePacket = new DatagramPacket(receiveData,
                                receiveData.length);

clientSocket.receive(receivePacket);

String modifiedSentence = new String(receivePacket.getData());

System.out.println("FROM SERVER:" + modifiedSentence);

   }}}
```

ESERCIZIO DI PREPARAZIONE ALL'ASSIGNMENT

- l'esercizio consiste nella scrittura di un server che offre il servizio di "Ping Pong" e del relativo programma client.
- un client si connette al server ed invia il messaggio di "Ping".
- il server, se riceve il messaggio, risponde con il messaggio di "Pong".
- Client e Server usano il protocollo UDP per lo scambio di messaggi.

ASSIGNMENT: JAVA PINGER

- PING è una utility per la valutazione delle performance della rete utilizzata per verificare la raggiungibilità di un host su una rete IP e per misurare il round trip time (RTT) per i messaggi spediti da un host mittente verso un host destinazione.

- lo scopo di questo assignment è quello di implementare un server PING ed un corrispondente client PING che consenta al client di misurare il suo RTT verso il server.

- la funzionalità fornita da questi programmi deve essere simile a quella della utility PING disponibile in tutti i moderni sistemi operativi. La differenza fondamentale è che si utilizza UDP per la comunicazione tra client e server, invece del protocollo ICMP (Internet Control Message Protocol).

- inoltre, poichè l'esecuzione dei programmi avverrà su un solo host o sulla rete locale ed in entrambe i casi sia la latenza che la perdita di pacchetti risultano trascurabili, il server deve introdurre un ritardo artificiale ed ignorare alcune richieste per simulare la perdita di pacchetti

PING CLIENT

- accetta due argomenti da linea di comando: nome e porta del server. Se uno o più argomenti risultano scorretti, il client termina, dopo aver stampato un messaggio di errore del tipo `ERR -arg x`, dove x è il numero dell'argomento.

- utilizza una comunicazione UDP per comunicare con il server ed invia 10 messaggi al server, con il seguente formato:

  `PING seqno timestamp`

  in cui seqno è il numero di sequenza del PING (tra 0-9) ed il timestamp (in millisecondi) indica quando il messaggio è stato inviato

- non invia un nuovo PING fino che non ha ricevuto l'eco del PING precedente, oppure è scaduto un timeout.

PING CLIENT

- stampa ogni messaggio spedito al server ed il RTT del ping oppure un * se la risposta non è stata ricevuta entro 2 secondi
- dopo che ha ricevuto la decima risposta (o dopo il suo timeout), il client stampa un riassunto simile a quello stampato dal `PING UNIX`

```
---- PING Statistics ----
10 packets transmitted, 7 packets received, 30% packet loss
round-trip (ms) min/avg/max = 63/190.29/290
```

- il `RTT` medio è stampato con 2 cifre dopo la virgola

PING SERVER

- è essenzialmente un echo server: rimanda al mittente qualsiasi dato riceve
- accetta un argomento da linea di comando: la porta, che è quella su cui è attivo il server + un argomento opzionale, il seed, un valore long utilizzato per la generazione di latenze e perdita di pacchetti. Se uno qualunque degli argomenti è scorretto, stampa un messaggio di errore del tipo ERR -arg x, dove x è il numero dell'argomento.
- dopo aver ricevuto un PING, il server determina se ignorare il pacchetto (simulandone la perdita) o effettuarne l'eco. La probabilità di perdita di pacchetti di default è del 25%.
- se decide di effettuare l'eco del PING, il server attende un intervallo di tempo casuale per simulare la latenza di rete
- stampa l'indirizzo IP e la porta del client, il messaggio di PING e l'azione intrapresa dal server in seguito alla sua ricezione (PING non inviato,oppure PING ritardato di x ms).

PING SERVER

```
java PingServer 10002 123
128.82.4.244:44229> PING 0 1360792326564 ACTION: delayed 297 ms
128.82.4.244:44229> PING 1 1360792326863 ACTION: delayed 182 ms
128.82.4.244:44229> PING 2 1360792327046 ACTION: delayed 262 ms
128.82.4.244:44229> PING 3 1360792327309 ACTION: delayed 21 ms
128.82.4.244:44229> PING 4 1360792327331 ACTION: delayed 173 ms
128.82.4.244:44229> PING 5 1360792327505 ACTION: delayed 44 ms
128.82.4.244:44229> PING 6 1360792327550 ACTION: delayed 19 ms
128.82.4.244:44229> PING 7 1360792327570 ACTION: not sent
128.82.4.244:44229> PING 8 1360792328571 ACTION: not sent
128.82.4.244:44229> PING 9 1360792329573 ACTION: delayed 262 ms
```

PING CLIENT

```
java PingClient localhost 10002
PING 0 1360792326564 RTT: 299 ms
PING 1 1360792326863 RTT: 183 ms
PING 2 1360792327046 RTT: 263 ms
PING 3 1360792327309 RTT: 22 ms
PING 4 1360792327331 RTT: 174 ms
PING 5 1360792327505 RTT: 45 ms
PING 6 1360792327550 RTT: 20 ms
PING 7 1360792327570 RTT: *
PING 8 1360792328571 RTT: *
PING 9 1360792329573 RTT: 263 ms
---- PING Statistics ----
10 packets transmitted, 8 packets received, 20% packet loss
round-trip (ms) min/avg/max = 20/158.62/299
```

JAVA PINGER

Invocazione corretta client/server:

```
java PingClient
Usage: java PingClient hostname port
java PingServer
Usage: java PingServer port [seed]
```

Invocazione non corretta client/server:

```
java PingClient atria three
ERR - arg 2
java PingServer abc
ERR - arg 1
```