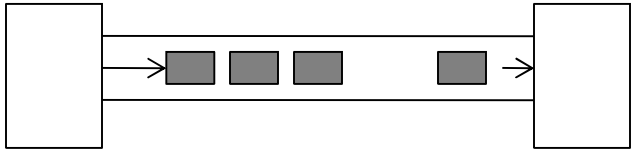
UDP e Datagram

CONNECTION ORIENTED VS. CONNECTIONLESS

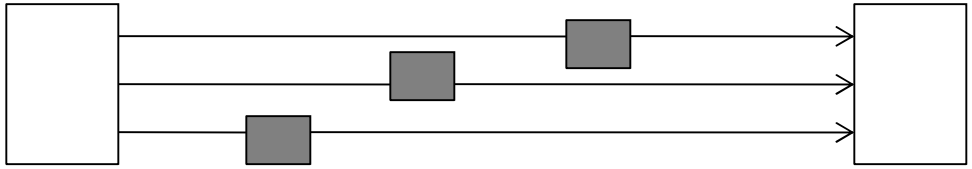
Comunicazione **Connection Oriented** (come una chiamata telefonica)

- creazione di una connessione (canale di comunicazione dedicato) tra mittente e destinatario
- invio dei dati sulla connessione
- chiusura della connessione



Comunicazione **Connectionless** (come l'invio di una lettera)

- non si stabilisce un canale di comunicazione dedicato
- mittente e destinatario comunicano mediante lo scambio di pacchetti
- orientato ai messaggi



TCP ED UDP: CONFRONTO

- in certi casi TCP offre “più di quanto sia necessario”, ad esempio se:
  - 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, quello delle porte all'indirizzamento offerto dal livello IP
  - scarto di 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

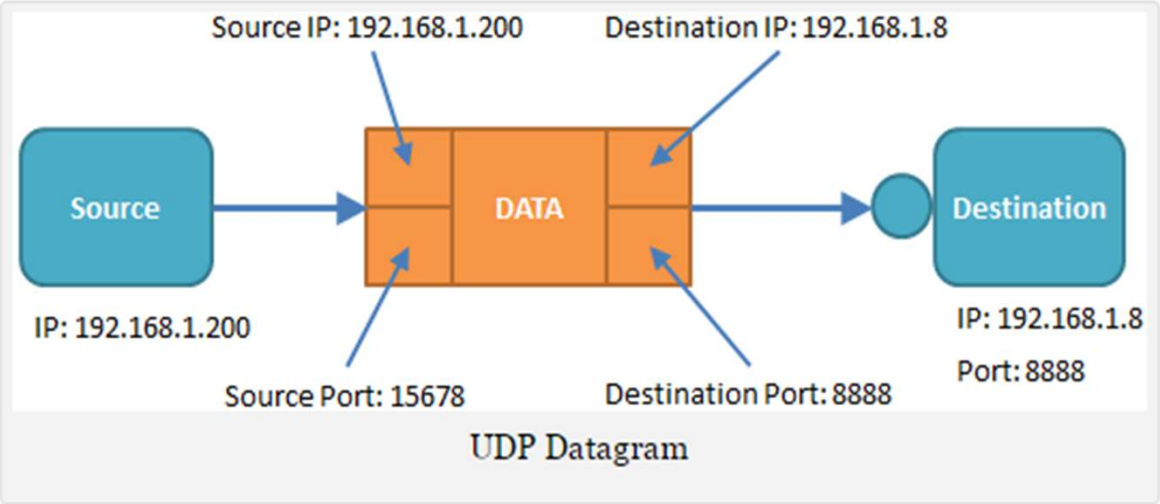
`JAVA socket API`: interfacce diverse per UDP e TCP

- **`TCP: Stream Sockets`**
- TCP: trasmissione vista come uno stream continuo di bytes provenienti dallo stesso mittente

- **`UDP: DatagramSocket`**
  - trasmissione orientata ai messaggi (One-way message)
  - in UDP, ogni messaggio, chiamato **Datagram**, è indipendente dagli altri e porta l'informazione per il suo instradamento
  - `send`, `receive` di DatagramPacket
  - ogni ricezione si riferisce ad un singolo messaggio inviato mediante una unica `send`. Corrispondenza tra `send` e `receive`: i dati inviati in un solo send() saranno ricevuti in un solo receive(). Questo non è valido per TCP

STRUTTURA DI UDP DATAGRAM

`UDP 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
- Tipicamente Source IP inserito automaticamente e Source Port scelta automaticamente in modo casuale

JAVA DATAGRAMSOCKET API

- Classi per la gestione di UDP:
  - **DatagramPacket** per costruire i datagram (sia per client che server)
    - usato per riempire e leggere pacchetti (datagram)
    - contiene la destinazione (ip e porta destinatario)
  - **DatagramSocket** per creare i sockets (sia per client che server).
    - invia e riceve DatagramPacket
    - conosce solo la *port* locale dalla quale inviare/ricevere
    - una `DatagramSocket` può inviare pacchetti a più destinazioni (non c'è una connessione tra due applicazioni)
    - non esiste uno stream tra applicazioni

JAVA DATAGRAMSOCKET API

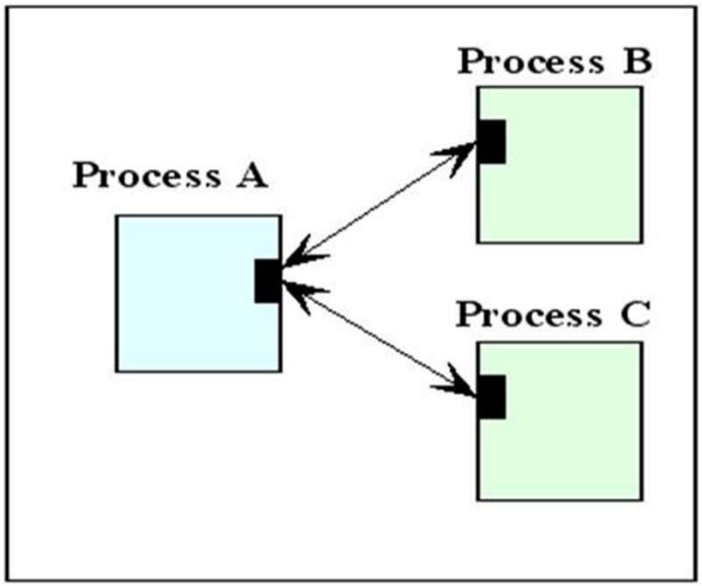
- Un processo mittente che desidera inviare dati su UDP, deve istanziare un oggetto di tipo **DatagramSocket,** collegato ad una porta locale
  - il processo mittente collega il suo socket ad una porta mittente (PM)
  - PM: può essere una porta effimera, non è necessario pubblicarla
    - Porte effimere: range di porte selezionabili dal SO, al di fuori del range delle well-known ports (tipicamente >49152)

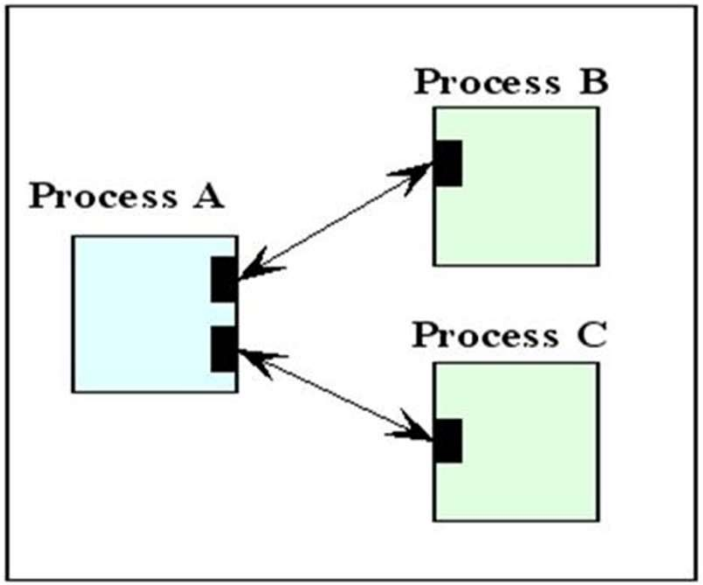
- il destinatario "pubblica" la porta a cui è collegato il socket di ricezione, affinché il mittente possa spedire pacchetti su quella porta

  il processo destinatario associa il suo socket ad una porta destinazione (PD)

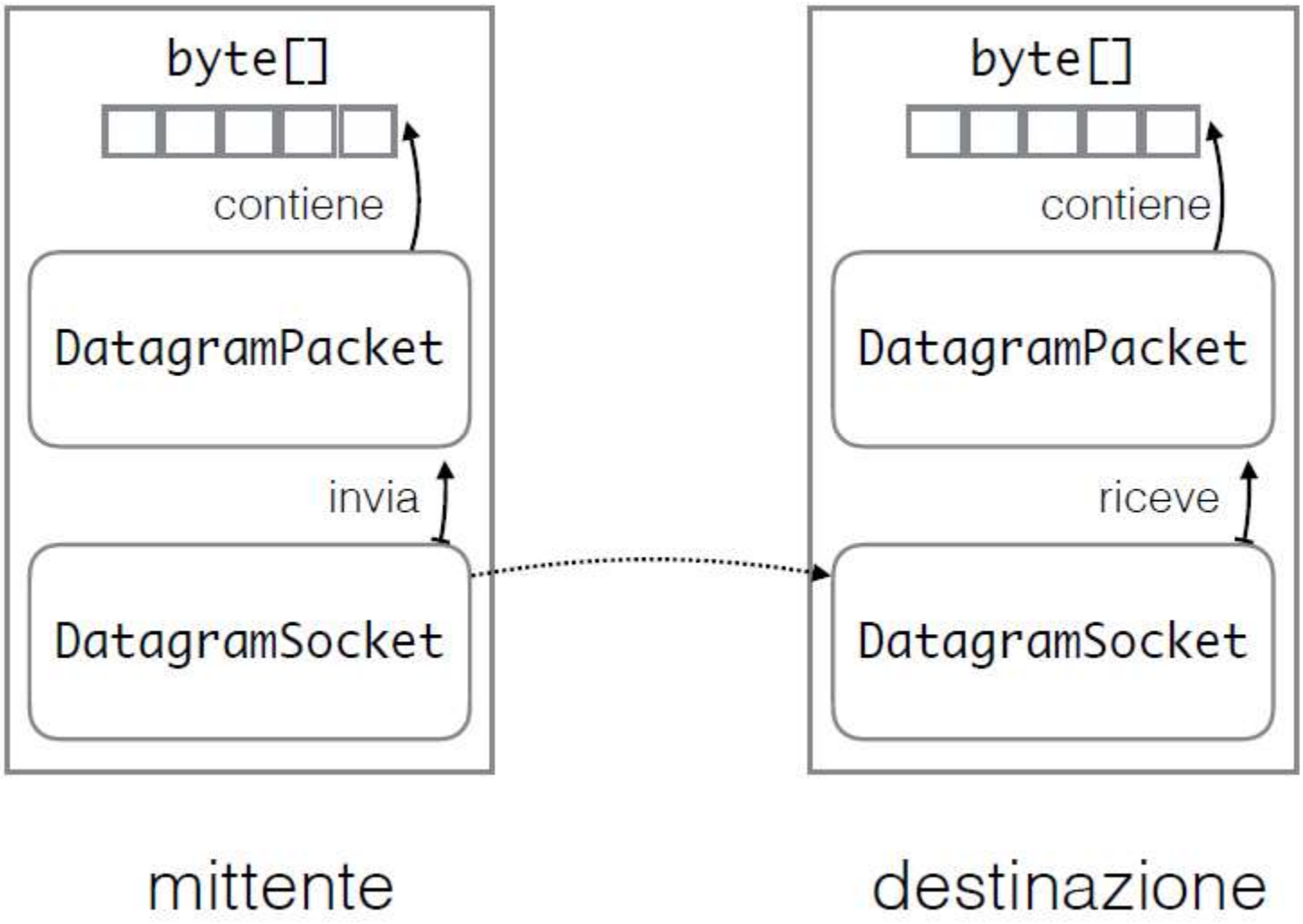
CARATTERISTICHE SOCKET UDP

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





UDP IN JAVA


byte[]
contiene
DatagramPacket
invia
DatagramSocket
byte[]
contiene
DatagramPacket
riceve
DatagramSocket
mittente
destinazione

UDP IN BREVE

Inviare un datagramma

- creare un **DatagramSocket** e collegarlo ad una porta locale
- 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
- invia 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 una **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**

DatagramSocket

```
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.
  - **costruttore 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 associato 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 può essere utilizzata per altre connessioni.

DatagramSocket

```
public class DatagramSocket extends Object

public DatagramSocket(int p) throws SocketException
```

- **costruttore 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 resa nota ai client.
  - 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<2000; 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);
   }
  }
}
```

SEND/RECEIVE BUFFERS

- Ad ogni socket sono associati due buffers: uno per la ricezione ed uno per la spedizione
- Questi buffers sono gestiti dal sistema operativo, non dalla JVM. La loro dimensione dipende dalla piattaforma su cui il programma è in esecuzione

```
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 65536 send buffer 65636**

SEND/RECEIVE BUFFERS

- Receive Buffer
  - La dimensione del receive buffer deve essere almeno uguale a quella del Datagram più grande che può essere ricevuto tramite quel buffer
  - Può consentire di bufferizzare un insieme di Datagram, nel caso in cui la frequenza con cui essi vengono ricevuti sia maggiore di quella con cui l'applicazione esegue la `receive()` e quindi preleva i dati dal buffer
- Send Buffer
  - La dimensione del send buffer viene utilizzata per stabilire la massima dimensione del Datagram
  - consente di bufferizzare un insieme di Datagram, nel caso in cui la frequenza con cui essi vengono generati sia molto alta (`send()`), rispetto alla frequenza con cui il supporto li preleva e spedisce sulla rete

- per modificare la dimensione del send/receive buffer:

```
void setSendBufferSize(int size)
void setReceiveBufferSize(int size)
```

sono da considerare 'suggerimenti' al supporto sottostante

DATAGRAMPACKET: COSTRUTTORI

- 2 costruttori per ricevere i dati

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

- 4 costruttori per inviare dati

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

Un oggetto **DatagramPacket** contiene:

- in ogni caso un riferimento ad un array di byte che contiene i dati da spedire (o quelli da ricevere)
- un insieme di informazioni utilizzate per individuare la posizione dei dati da estrarre dall' (o inserire nell') array di byte: `length`, `offset` (offset=0, di default)
- eventuali informazioni di addressing
  - informazioni sul destinatario (indirizzo IP e porta) se il Datagram deve essere spedito

DATAGRAMPACKET PER INVIO

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

- costruttore per un **DatagramPacket** da inviare
- **`length`** indica il numero di bytes che devono essere copiati dal buffer nel datagramma da inviare, a partire dal byte indicato da `offset`, se indicato.
- **destination + port** individuano il destinatario

DATAGRAMPACKET PER RICEZIONE

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

- **Costruisce un DatagramPacket per ricevere un messaggio di lunghezza `length`**.
- Il valore di `length` indica il numero di byte nel buffer che saranno usati per ricevere i dati. Deve essere minore o uguale di `buffer.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 suo payload
  - se settato offset, la copia avviene dalla posizione individuata da esso
  - 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
- `getLength()`: indica il numero di bytes effettivamente ricevuti

DATAGRAMPACKET: METODI SET

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

- costruzione del **DatagramPacket**
  - mediante il costruttore (che comunque richiama il metodo SetData)
  - mediante il metodo **setData()**

- i metodi set invocati su un oggetto **DatagramPacket,** inseriscono esplicitamente un riferimento al byte array in esso.
  - `void setData(byte[] buffer)`
    - Offset 0 e length uguale a buffer length
  - `void setData(byte[] buffer, int offset, int length)`
    - il secondo metodo aggiorna sia offset che length

METODI SET

```
void setPort(int iport)
```

- setta la porta nel datagram

```
void setLength(int length)
```

- Setta la lunghezza del payload del Datagram

```
void setAddress(InetAddress iaddr)
```

- setta l'InetAddress della macchina a cui il payload è diretto
- utile quando si deve mandare lo stesso Datagram a più destinatari

```
DatagramSocket socket= new DatagramSocket();
 String s = "Really important message";
byte [] data = s.getBytes("UTF-8");
DatagramPacket dp = new DatagramPacket(data, data.length);
 dp.setPort(2000);
 String network = "128.238.5.";
 for (int host =1; host <255; host++)
    {InetAddress remote = InetAddress.getByName(network+host);
     dp.setAddress(remote);
     socket.send(dp);
     System.out.println("sent");}
```

METODI SET

```
void setSocketAddress(SocketAddress addr)
```

- utile per inviare risposte

```
DatagramPacket input = new DatagramPacket(new byte[8192], 8192);
 socket.receive(input);
 DatagramPacket output = new DatagramPacket ("Hello
                                here".getBytes("UTF-8"),11);
 SocketAddress address = input.getSocketAddress();
 output.setSocketAddress(address);;
 socket.send(output);
```

DATAGRAMPACKET: METODI GET

`InetAddress getAddress()`

- restituisce l'indirizzo IP della macchina a cui il `Datagram` è stato inviato oppure della macchina a cui è stato spedito

`int getPort( )`

- restituisce il numero di porta sull'host remoto cui il `Datagram` è stato inviato, o dell'host a cui è stato spedito

`int [] getLength(), int [] getOffset()`

- restituisce la lunghezza/offset del `Datagram` da inviare o di quello ricevuto

`SocketAddress getSocketAddress()`

- restituisce (IP+numero di porta) del `Datagram` sull'host remoto cui il `Datagram` è stato inviato, o dell'host a cui è stato spedito

DATAGRAMPACKET: METODI GET

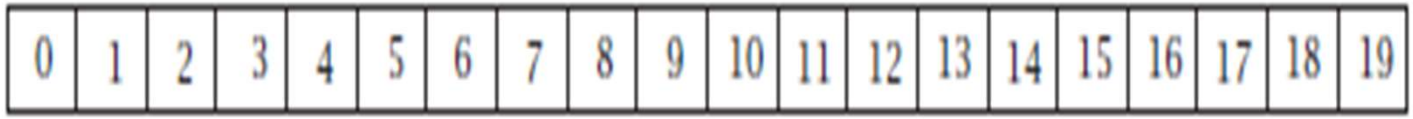
byte[] getData ()

- restituisce un riferimento al buffer associato più recentemente al **DatagramPacket** o mediante invocazione del costruttore o mediante il metodo **setData()**.
- Restituisce il buffer di dati. **The data received or the data to be sent starts from the `offset` in the buffer, and runs for `length` long**

- può provocare problemi nel caso in cui il buffer abbia dimensioni maggiori dell'effettivo dato ricevuto (vedi esempi nelle slide seguenti)
- `int getLength()`: numero di *byte* di dati
- `int getOffset()`: dove cominciano i dati

METODI GET E SET: UN ESEMPIO

- consideriamo il byte array **buf** di dimensione 20, inizializzato come segue



- sia dg un DatagramPacket

```
dg.setData(buf,5,12)
```

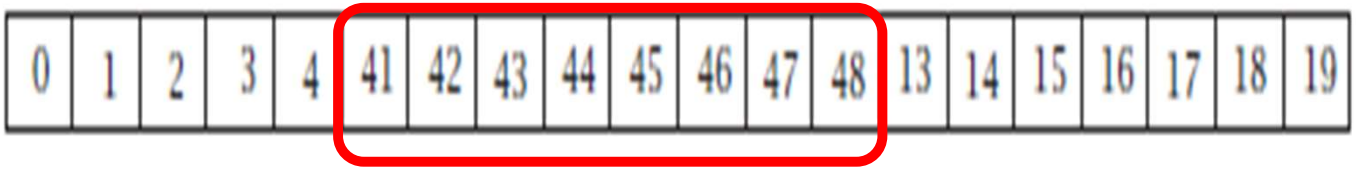
- imposta la porzione di **buf** dall'indice 5 per una lunghezza di 12 byte come buffer di ricezione/invio associato al DatagramPacket

```
dgsocket.receive(dg)
```

- supponiamo si riceva un messaggio di 8 bytes: viene memorizzato nelle posizioni del vettore impostate precedentemente

```
dg.getData()
```

- restituisce un riferimento all'intero buffer, che ora contiene:

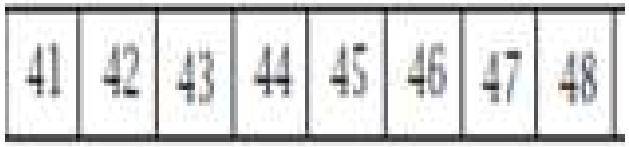


METODI GET E SET: UN ESEMPIO

- Un modo per leggere solo i dati effettivamente ricevuti:

```
byte[] destBuf = new byte[dg.getLength()];
System.arraycopy(dg.getData(),dg.getOffset(),destBuf,0,
  destBuf.length)
```

`java.lang.System.arraycopy()` method copia un array dall'array di origine, a partire dalla posizione specificata, nella posizione specificata dell'array di destinazione



JAVA: GENERAZIONE DEI PACCHETTI

- i dati inviati mediante UDP devono essere rappresentati come array 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` prelevando `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

JAVA : INVIARE E RICEVERE PACCHETTI

Datagram Socket

```
DatagramSocket clientsocket = new DatagramSocket()

DatagramSocket serverSock= new DatagramSocket(port);
```

Invio di pacchetti

```
sock.send(DatagramPacket dp)
```

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

Ricezione di pacchetti

```
sock.receive(DatagramPacket rp)
```

dove sock è il socket attraverso il quale ricevo il pacchetto rp

COMUNICAZIONE UDP: CARATTERISTICHE

`send()`

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

`receive()` è 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.

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 **java.net.SocketTimeoutException**

- 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.

INVIARE 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("127.0.0.1");
     for (int i = 1; i < buffer.length; i++) {
          //Constructs a datagram packet for sending packets of length I
          DatagramPacket mypacket = new DatagramPacket(buffer,i,address,
                                                       40000);
          clientSocket.send(mypacket);
          Thread.sleep(200);
     }
    }
    catch (Exception e) { e.printStackTrace(); }
  }
}
```

RICEVERE DATAGRAMPACKET

```
import java.io.IOException; import java.net.*;
public class Receiver {
 public static void main(String args[])  {
 try (DatagramSocket serverSock = new DatagramSocket(40000)){
     serverSock.setSoTimeout(20000);
     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);
     }
 }
 catch (SocketTimeoutException e) { System.out.println("Bye Bye");
 }
 catch (IOException e) { //e.printStackTrace();
 }}
```

“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 Sender2 {
   public static void main (String args[]) {
     try (DatagramSocket clientSocket = new DatagramSocket()) {;
      byte[] buffer="1234567890abcdefghijklmnopqrstuvwxyz".getBytes("US-ASCII");
      InetAddress address = InetAddress.getByName("127.0.0.1");
      for (int i = buffer.length; i >0; i--) {
          DatagramPacket mypacket = new DatagramPacket(buffer,i,address,40000);
          clientSocket.send(mypacket);
          Thread.sleep(200);
       }
      }
      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 Receiver2 {
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);
                        //new String(receivedPacket.getData(), 0, receivedPacket.getLength(), "US-ASCII");
        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
```

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 byte in un vettore di byte da spedire con i pacchetti UDP

- **ByteArrayInputStream**, converte un vettore di byte in uno stream di byte.

DATI STRUTTURATI IN PACCHETTI UDP

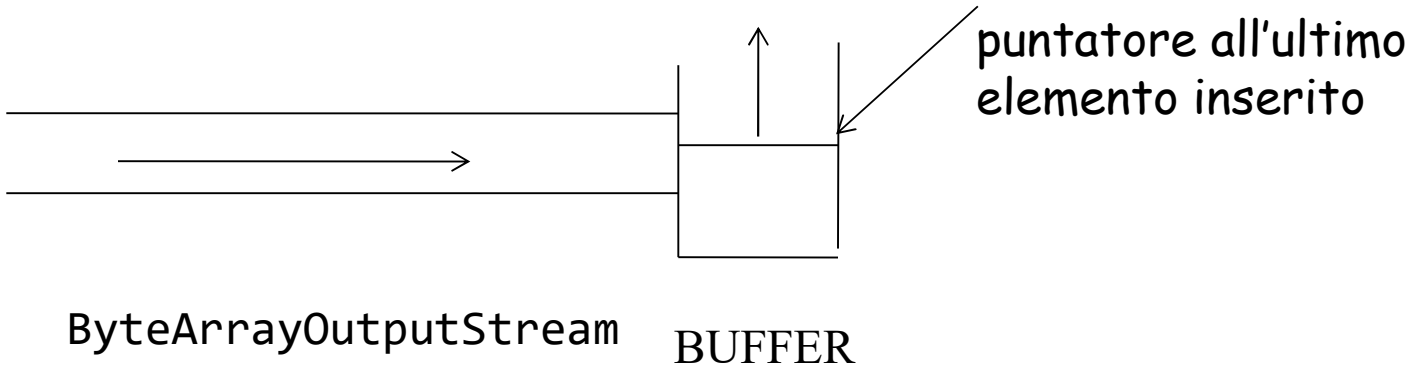
```
public ByteArrayOutputStream()
public ByteArrayOutputStream(int size)
```

- 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

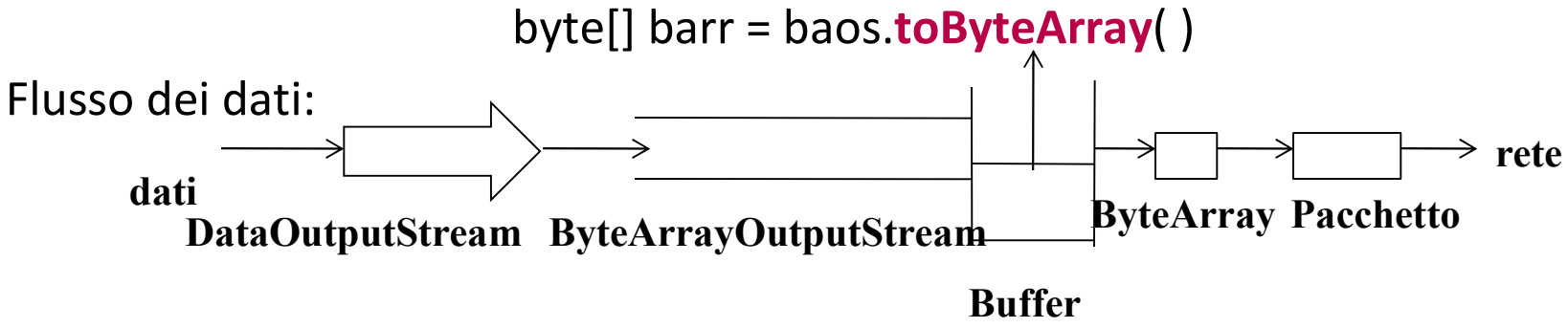


BYTE ARRAY OUTPUT STREAMS

- ad un **ByteArrayOutputStream** può essere collegato un altro filtro

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

- I dati possono essere recuperate con i metodi `toByteArray()` o `toString()`
- 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



LA CLASSE BYTEARRAYOUTPUTSTREAM

Metodi per la gestione dello stream:

- **public int size( )** restituisce `count`, cioè il numero di bytes memorizzati nello stream (numero di byte validi in buf)

- **public synchronized void reset()** svuota il buffer, assegnando 0 a `count`. Tutti i dati precedentemente scritti vengono eliminati.

  baos.**reset**()

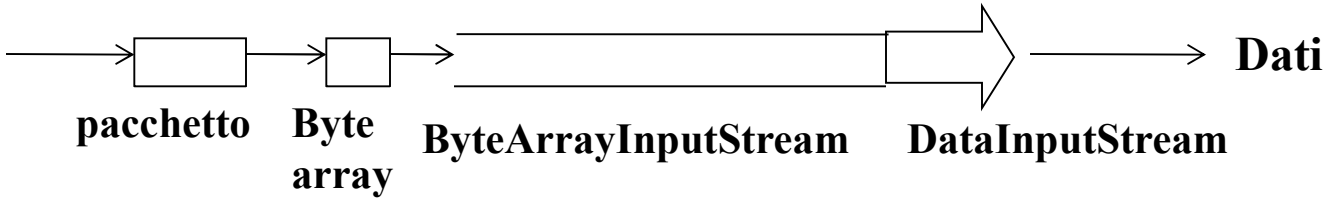
- **public synchronized byte toByteArray()** restituisce un vettore in cui sono stati copiati tutti i bytes presenti nello stream. Non modifica `count`
  - il metodo **toByteArray** non svuota il buffer.

BYTE ARRAY INPUT 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 dalla posizione `offset`.
- è possibile concatenare un **DataInputStream**

Flusso dei dati:



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();
  }
 }
}
```

Alterna scrittura e invio di interi e stringhe

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
    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();
    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);
    }
  }
}
```

Alterna lettura di interi e stringhe

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, presso il destinatario scritture/letture possono non corrispondere
- Realizzazione di UDP affidabile: utilizzo di ack per confermare la ricezione + identificatori unici

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();
```

Messaggio strutturato

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();
```

Conclusioni

- 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`.

Esercizio

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 un messaggio di "Ping".

Il server, se riceve il messaggio, risponde con un messaggio di "Pong".

Il client sta in attesa n secondi di ricevere il messaggio dal server (timeout) e poi termina.

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 si 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 entrambi 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. 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
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
```

Invocazione non corretta client/server:

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