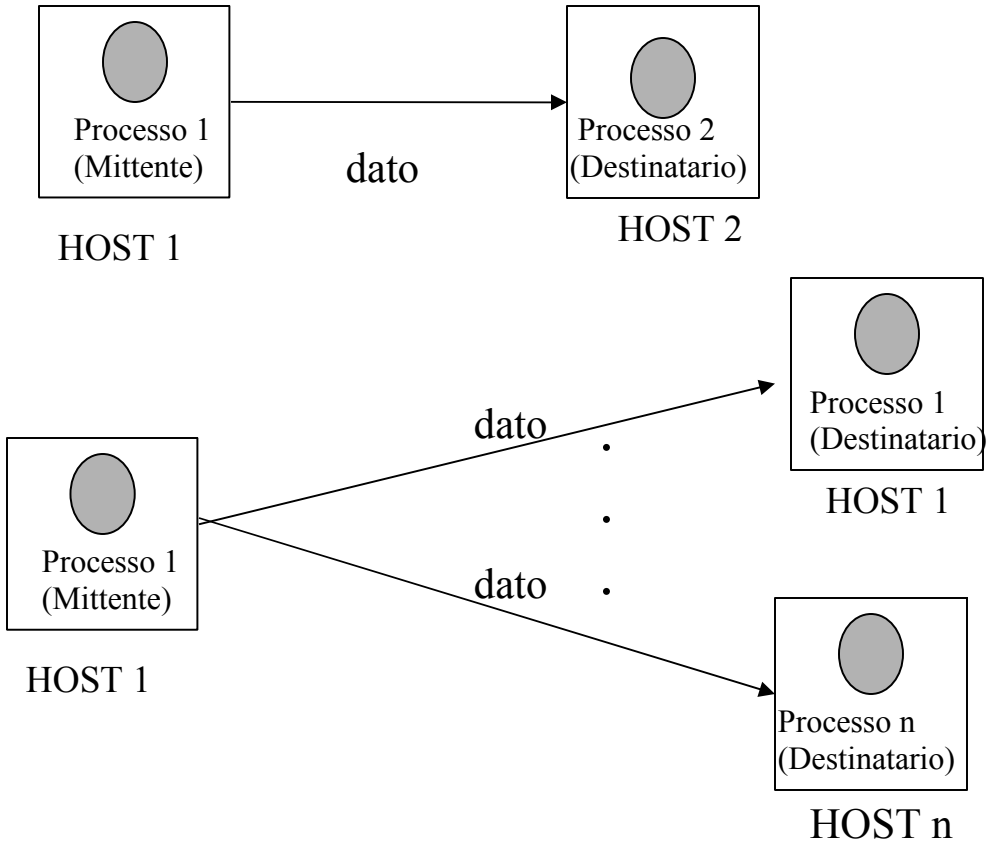
Lezione n.5
LPR-A-09
Il protocollo UDP; Socket e Datagram

20/10/2008

Vincenzo Gervasi

MECCANISMI DI COMUNICAZIONE TRA PROCESSI

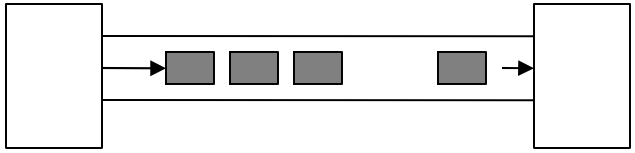
Meccanismi di comunicazione tra processi (IPC)



COMUNICAZIONE
CONNECTION ORIENTED VS. CONNECTIONLESS

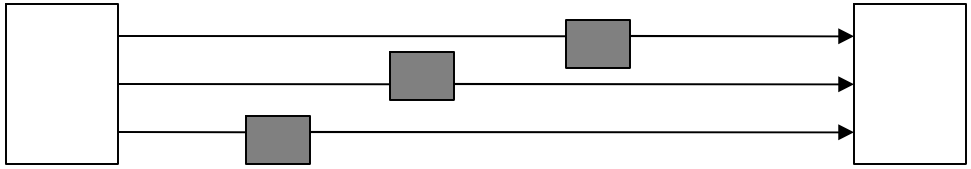
Comunicazione Connection Oriented (come una chiamata telefonica)

- 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



COMUNICAZIONE
CONNECTION ORIENTED VS. CONNECTIONLESS

- Indirizzamento:
  - *Connection Oriented:* l'indirizzo del destinatario è specificato al momento della connessione
  - *Connectionless:* l'indirizzo del destinatario viene specificato in ogni pacchetto (per ogni send)
- Ordinamento dei dati scambiati:
  - *Connection Oriented:* ordinamento dei messaggi garantito
  - *Connectionless:* nessuna garanzia sull'ordinamento dei messaggi
- Utilizzo:
  - *Connection Oriented:* grossi streams di dati
  - *Connectionless* : invio di un numero limitato di dati

COMUNICAZIONE: CONNECTION ORIENTED VS. CONNECTIONLESS

Protocollo UDP (User Datagram Protocol) =

connectionless, trasmette pacchetti di dati (Datagrams)

- ogni datagram deve contenere l'indirizzo del destinatario
- datagrams spediti dallo stesso processo possono seguire percorsi diversi ed arrivare al destinatario in ordine diverso rispetto all'ordine di spedizione

Protocollo TCP (Transmission Control Protocol) =

trasmissione connection-oriented o stream oriented

- viene stabilita una connessione tra mittente e destinatario
- su questa connesione si spedisce una sequenza di dati = stream di dati
- per modellare questo tipo di comunicazione in JAVA si possono sfruttare i diversi tipi di stream definiti dal linguaggio.

IPC: MECCANISMI BASE

Una API per la comunicazione tra processi (IPC= Inter Process Communication) deve garantire almeno le seguenti funzionalità

- Send per trasmettere un dato al processo destinatario
- Receive per ricevere un dato dal processo mittente
- Connect (solo per comunicazione connection oriented) per stabilire una connessione logica tra mittente e destinatario
- Disconnect per eliminare una connessione logica

Possono esistere diversi tipi di send/receive (sincrona/asincrona, simmetrica/asimmetrica)

IPC: MECCANISMI BASE

Un esempio: HTTP (1.0)

- il processo che esegue il Web browser esegue una connect per stabilire una connessione con il processo che esegue il Web Server
- il Web browser esegue una send, per trasmettere una richiesta al Web Server (operazione GET)
- il Web server esegue una receive per ricevere la richiesta dal Web Browser, quindi a sua volta esegue una send per inviare la risposta
- i due processi eseguono una disconnect per terminare la connessione

HTTP 1.1: Più richieste su una connessione (più send e receive).

IPC: MECCANISMI BASE

Comunicazione sincrona (o bloccante): il processo che esegue la send o la receive si sospende fino al momento in cui la comunicazione è completata.

send sincrona = completata quando i dati spediti sono stati ricevuti dal destinatario (è stato ricevuto un ack da parte del destinatario)

receive sincrona = completata quando i dati richiesti sono stati ricevuti

send asincrona (non bloccante) = il destinatario invia i dati e prosegue la sua esecuzione senza attendere un ack dal destinatario

receive asincrona = il destinatario non si blocca se i dati non sono arrivati. Possibile diverse implementazioni

IPC: MECCANISMI BASE

Receive Non Bloccante.

- se il dato richiesto è arrivato, viene reso disponibile al processo che ha eseguito la receive

- se il dato richiesto non è arrivato:

  - il destinatario esegue nuovamente la receive, dopo un certo intervallo di tempo (polling)

  - il supporto a tempo di esecuzione notifica al destinatario l'arrivo del dato (richiesta l'attivazione di un event listener)

IPC: MECCANISMI BASE

Comunicazione non bloccante: per non bloccarsi indefinitamente

- Timeout – meccanismo che consente di bloccarsi per un intervallo di tempo prestabilito, poi di proseguire comunque l'esecuzione

- Threads – l'operazione sincrona può essere effettuate in un thread. Se il thread si blocca su una send/receive sincrona, l'applicazione può eseguire altri thread.

- Nel caso di receive sincrona, gli altri threads ovviamente non devono richiedere per l'esecuzione il valore restituito dalla receive

INVIARE OGGETTI

Invio di strutture dati ed, in generale, di oggetti richiede :

- il mittente deve effettuare la serializzazione delle strutture dati (eliminazione dei puntatori)
- il destinatario deve ricostruire la struttura dati nella sua memoria

Da Wikipedia: La serializzazione è un processo per salvare un oggetto in un supporto di memorizzazione lineare (ad esempio, un file o un'area di memoria), o per trasmetterlo su una connessione di rete. La serializzazione può essere in forma binaria o può utilizzare codifiche testuali (ad esempio il formato XML)... Lo scopo della serializzazione è di trasmettere l'intero stato dell'oggetto in modo che esso possa essere successivamente ricreato nello stesso identico stato dal processo inverso, chiamato deserializzazione.

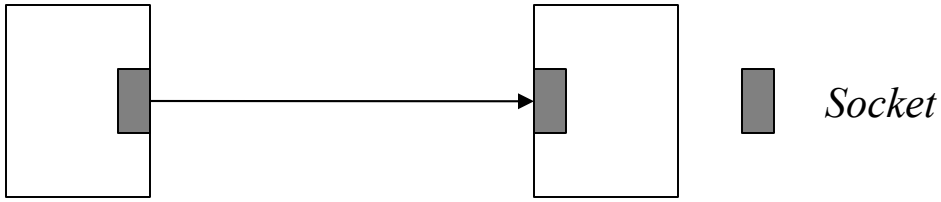
Il processo di serializzare un oggetto viene anche indicato come marshalling

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.



JAVA IPC: I SOCKETS

Socket Application Program Interface = Definisce un insieme di meccanismi che supportano la comunicazione di processi in ambiente distribuito.

- JAVA socket API: definisce classi diverse per UDP e TCP

  Protocollo UDP = **DatagramSocket**

  Protocollo TCP = **ServerSocket** e **Socket**

FORMATO DEL PACCHETTO 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 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

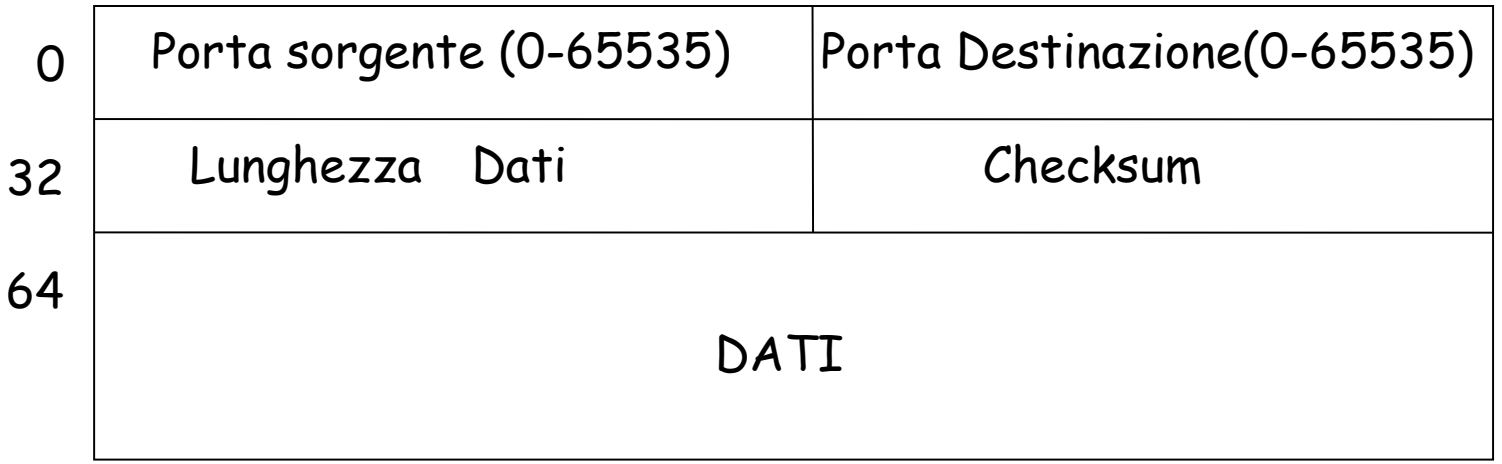
Introdotto per evitare *percorsi circolari* infiniti del pacchetto. Utilizzato anche per limitare la diffusione del pacchetto nel multicast

LIVELLO IP: FORMATO DEL PACCHETTO

- Protocol: Il valore di questo campo indica il protocollo a livello trasporto utilizzato (es: TCP 6, UDP 17, IPv6 41). 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

L'HEADER UDP

- Datagram UDP = unità di trasmissione definita dal protocollo UDP
- Ogni datagram UDP
  - viene incapsulato in un singolo pacchetto IP
  - definisce un header che viene aggiunto all'header IP



L'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 decide in base al numero di porta il servizio (processo) a cui devono essere consegnare i dati
- Checksum: si riferisce alla verifica di correttezza delle 4 parole di 16 bits dell'header
- Lunghezza: lunghezza del datagram

DATAGRAM UDP: LUNGHEZZE AMMISSIBILI

- IPV4 limita la lunghezza del datagram a 64K (65507 bytes + header)
- In pratica, la lunghezza del pacchetto UDP è limitata alla dimensione dei buffer associati al socket in ingresso/uscita
  - dimensione del buffer = 8K nella maggior parte dei sistemi operativi.
  - in certi sistemi si può incrementare la dimensione di questo buffer
- I routers IP possono frammentare i pacchetti IP che superano una certa dimensione
  - se un pacchetto IP che contiene un datagram UDP viene frammentato, il pacchetto non viene ricostruito e viene di fatto scartato
  - per evitare problemi legati alla frammentazione, è meglio restringere la lunghezza del pacchetto a 512 bytes
- E' possibile utilizzare dimension maggiori per pacchetti spediti su LAN
- IPV6 datagrams = 2^{32} -1 bytes (jumbograms!)

TRASMISSIONE PACCHETTI UDP

Per scambiare un pacchetto UDP:

- mittente e destinatario devono creare due 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 pacchetto UDP, il mittente

- crea un 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.

TRASMISSIONE PACCHETTI UDP

Il destinatario, per ricevere un pacchetto UDP

- crea un 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

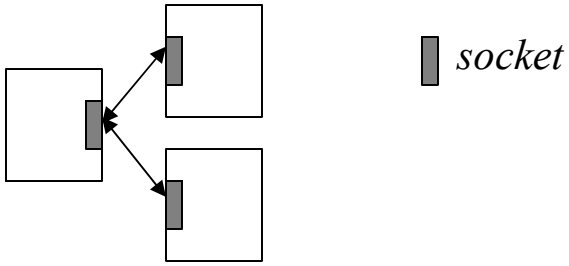
  - i dati inviati mediante UDP devono essere rappresentati come vettori di bytes

  - JAVA offre diversi tipi di filtri per generare streams di bytes a partire da dati strutturati/ad alto livello

TRASMISSIONE PACCHETTI 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 : SOCKETS UDP

public class DatagramSocket

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 un **client UDP**.

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

JAVA : SOCKETS UDP

Altro costruttore:

public DatagramSocket (int p) throws SocketException

- crea un socket sulla porta specificata (p).
- viene sollevata un'eccezione (**BindException / SocketException**) se la porta è già utilizzata, oppure se si tenta di connettere il socket ad una porta su cui non si hanno diritti (**SecurityException**)
- utilizzato da un **server UDP**.

- 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  {
                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);}
       } }
```

I DATAGRAMPACKET

- Un oggetto di tipo **DatagramPacket** può essere utilizzato per
  - memorizzare un datagram che deve essere spedito sulla rete
  - contenere i dati copiati da un datagram ricevuto dalla rete
- Struttura di un **DatagramPacket**
  - **Buffer**: riferimento ad **un array di byte** per la memorizzazione dei dati spediti/ricevuti
  - **Metadati**:
    - Lunghezza: quantità di dati presenti nel buffer
    - Offset: localizza il primo byte significativo nel buffer
  - **InetAddress** e **porta** del mittente o del destinatario
- I campi assumono diverso significato a seconda che il **DatagramPacket** sia utilizzato per spedire o per ricevere dati

LA CLASSE DATAGRAMPACKET

```
public final class DatagramPacket

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

- per la costruzione di un **DatagramPacket** da inviare sul socket
- **length** indica il numero di bytes che devono essere copiati dal vettore **data** *[a partire dalla posizione offset]* nel pacchetto UDP/IP.
- **destination + port** individuano il destinatario
- il messaggio deve essere trasformato in una sequenza di bytes e memorizzato nell'array data (strumenti necessari per la traduzione, es: metodo **getBytes ( )**, la classe **java.io.ByteArrayOutputStream**)

LA CLASSE DATAGRAMPACKET

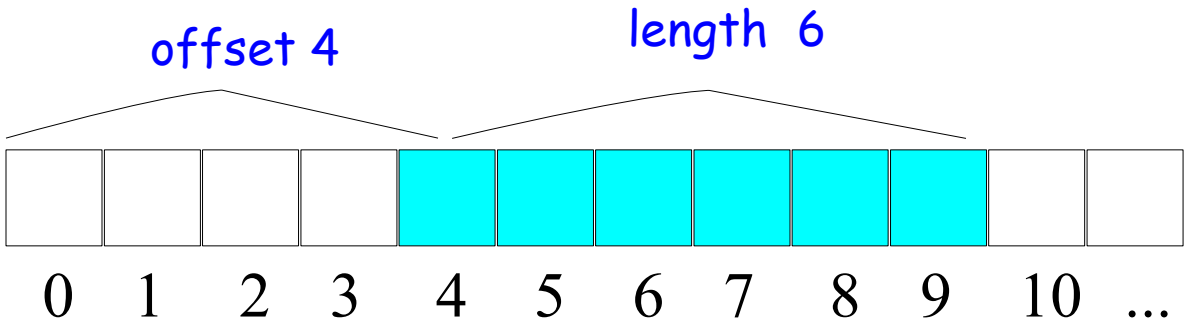
public DatagramPacket (**byte**[ ] data, *[**int** offset,]* **int** length)

- 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.
- il **payload** del pacchetto (la parte che contiene i dati) viene copiato nel buffer *[a partire dalla posizione offset]* 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
- il parametro **length**
  - prima della copia, indica il numero massimo di bytes che possono essere copiati nel buffer
  - dopo la copia, indica il numero di bytes effettivamente copiati.

GENERAZIONE DEI PACCHETTI

Metodi per la conversione stringhe/vettori di bytes

- byte[ ] getBytes( ) applicato ad un oggetto **String**, restituisce una sequenza di bytes che codifica i caratteri della stringa usando la codifica di default dell'host
- String (byte[ ] bytes, int offset, int length) costruisce un nuovo oggetto di tipo **String** decodificando la sottosequenza di length bytes dall'array bytes, a partire dalla posizione offset



INVIARE E RICEVERE PACCHETTI

Invio di pacchetti

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

dove: ***sock*** è il **DatagramSocket** attraverso il quale voglio spedire il pacchetto (**DatagramPacket**) *dp*

Ricezione di pacchetti

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

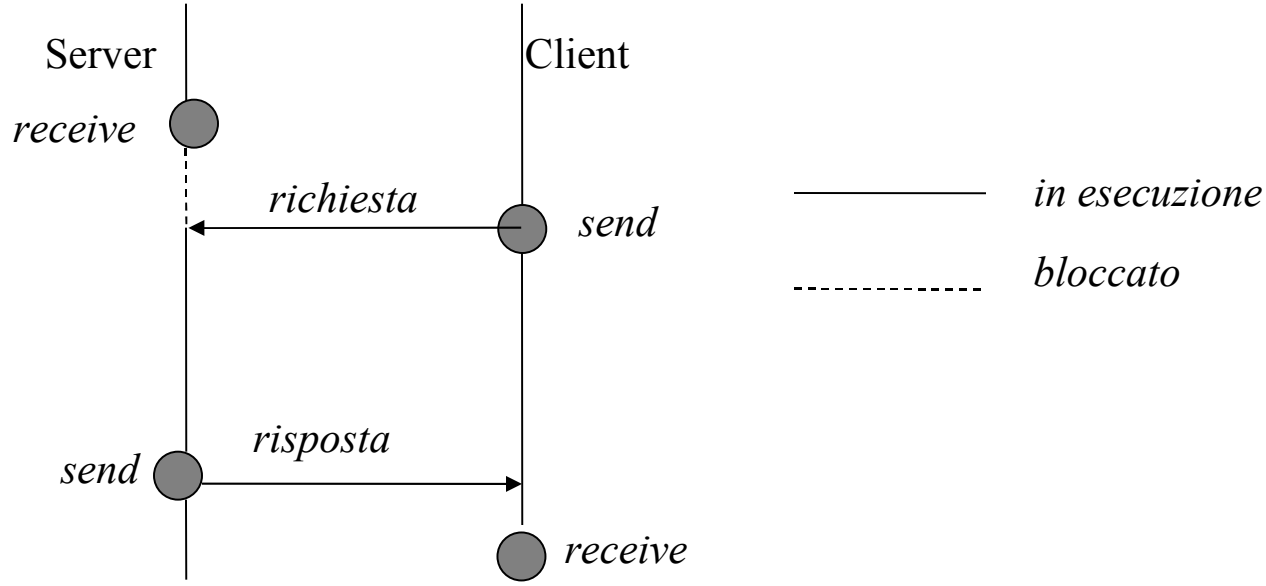
dove **sock** è il **DatagramSocket** attraverso il quale ricevo il pacchetto e buffer è il **DatagramPacket** in cui memorizzo il pacchetto ricevuto

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 InterruptedIOException
- 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.

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 UDPBufferSize {
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 **(8 Kbyte)**

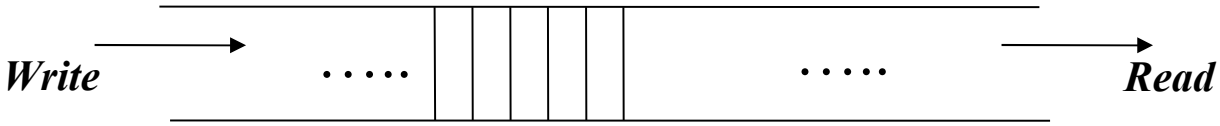
SEND/RECEIVE BUFFERS

- La dimensione del **receive buffer** deve essere almeno uguale a quella del Datagram più grande che può essere ricevuto tramite quel buffer
- Receive Buffer: consente la bufferizzazione di 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
- La dimensione del **send buffer** viene utilizzata per stabilire la massima dimensione del Datagram
- Send Buffer: consente la bufferizzare di un insieme di Datagram, nel caso in cui la frequenza con cui essi vengono generati sia molto alta, rispetto alla frequenza con cui il supporto li preleva e spedisce sulla rete
- Per modificicare la dimensione del send/receive buffer
  - void setSendBufferSize(int size)
  - **void** setReceiveBufferSize(int size)

  sono da considerare 'suggerimenti' al supporto sottostanta

JAVA: IL CONCETTO DI STREAM

- Streams: introdotti per modellare l'interazione del programma con i dispositivi di I/O (console, files, connessioni di rete,…)
- JAVA Stream I/O: basato sul concetto di stream: si può immaginare uno stream come una condotta tra una sorgente ed una destinazione (dal programma ad un dispositivo o viceversa), da un estremo entrano dati, dall'altro escono



- L'applicazione può inserire dati ad un capo dello stream

- I dati fluiscono verso la destinazione e possono essere estratti dall'altro capo dello stream

Esempio: l'applicazione scrive su un FileOutputStream. Il dispositivo legge i dati e li memorizza sul file

JAVA: IL CONCETTO DI STREAM

Caratteristiche principali degli streams:

- mantengono l'ordinamento **FIFO**
- read only o write only
- accesso sequenziale
- bloccanti: quando un'applicazione legge un dato dallo stream (o lo scrive) si blocca finchè l'operazione non è completata (ma le ultime versioni di JAVA introducono l' I/O non bloccante: **java.nio**)
- non è richiesta una corrispondenza stretta tra letture/scritture

  Esempio: una unica scrittura inietta 100 bytes sullo stream, che vengono letti con due read successive, la prima legge 20 bytes, la seconda 80 bytes

STREAMS DI BASE: OUTPUTSTREAM

Streams di bytes: **public abstract class OutputStream**

Metodi di base:

```
public abstract void write (int b) throws IOException;

public void write(byte [ ] data) throws IOException;

public void write(byte [ ] data, int offset, int length) throws
                                                    IOException;
```

La classe **OutputStream** ed il metodo **write** sono dichiarati astratti.

- **write (*int* b)** scrive su un **OutputStream** il byte meno significativo dell'intero passato (gli ultimi 8 bit dei 32)
- L'implementazione del metodo **write** può richiedere codice nativo (es: scrittura su un file…).
- Le sottoclassi descrivono stream legati a particolari dispositivi di I/O (file, console,…) .

STREAMS DI BASE: INPUTSTREAM

Stream di bytes: **public abstract class InputStream**

Metodi di base:

```
public abstract int read () throws IOException;

public int read(byte [ ] b) throws IOException;

public int read(byte [ ] b, int offset, int length) throws
                                                IOException;
```

La classe **InputStream** ed il metodo **read** sono dichiarati astratti.

- **read ()** restituisce un byte (un int nel rage 0-255) oppure -1 se ha raggiunto la fine dello stream.
- L'implementazione del metodo **read** può richiedere codice nativo (es: lettura da file…).
- Le sottoclassi descrivono input stream legati a particolari dispositivi di I/O (file, console,…) .

STREAMS DI BASE E WRAPPERS

- Stream di base: classi utilizzate
  - *Stream utilizzate per la trasmissione di bytes
  - *Reader o *Writer: utilizzate per la trasmissione di caratteri
- Per non lavorare direttamente a livello di bytes o di carattere
  - Si definisce una serie di wrappers che consentono di avvolgere / incapsulare uno stream intorno all'altro (come un tubo composto da più guaine...)
  - l'oggetto più interno è uno stream di base che 'avvolge' la sorgente dei dati (ad esempio il file, la connessione di rete,....).
  - i wrappers sono utilizzati per il trasferimento di oggetti complessi sullo stream, per la compressione di dati, per la definizione di strategie di buffering (per rendere più veloce la trasmissione)

JAVA STREAMS: FILTRI

DataOutputStream consente di trasformare dati di un tipo primitivo JAVA in una sequenza di bytes da iniettare su uno stream.
Alcuni metodi utili:

```
public final void writeBoolean(boolean b) throws IOException;
public final void writeInt (int i) throws IOException;
public final void writeDouble (double d) throws IOException;
……
```

Il filtro produce una sequenza di bytes che rappresentano il valore del dato.
Rappresentazioni utilizzate:

- interi 32 bit complemento a due, big-endian
- float 32 bit IEEE754 floating points

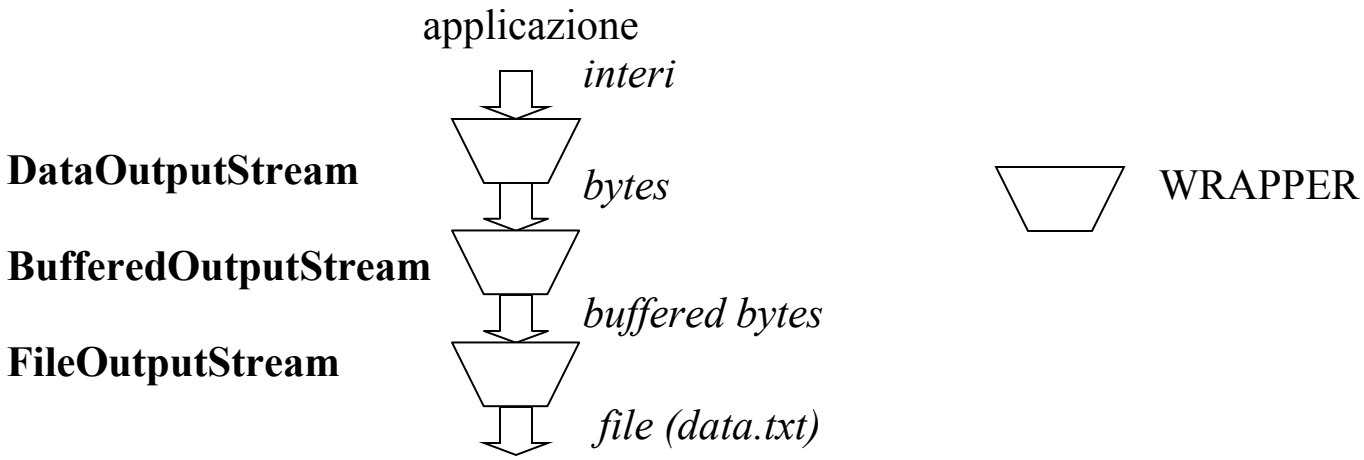
Formati utilizzati dalla maggior parte dei protocolli di rete
Nessun problema se i dati vengono scambiati tra programmi JAVA.

STREAMS DI BASE E WRAPPERS

InputStream, OuputStream consentono di manipolare dati a livello molto basso, per cui lavorare direttamente su questi streams risulta parecchio complesso.

Per estendere le funzionalità degli streams di base: classi wrapper

```
DataOutputStream = new DataOutputStream(
                       new BufferedOutputStream(
                       new FileOutputStream("data.txt")))
```

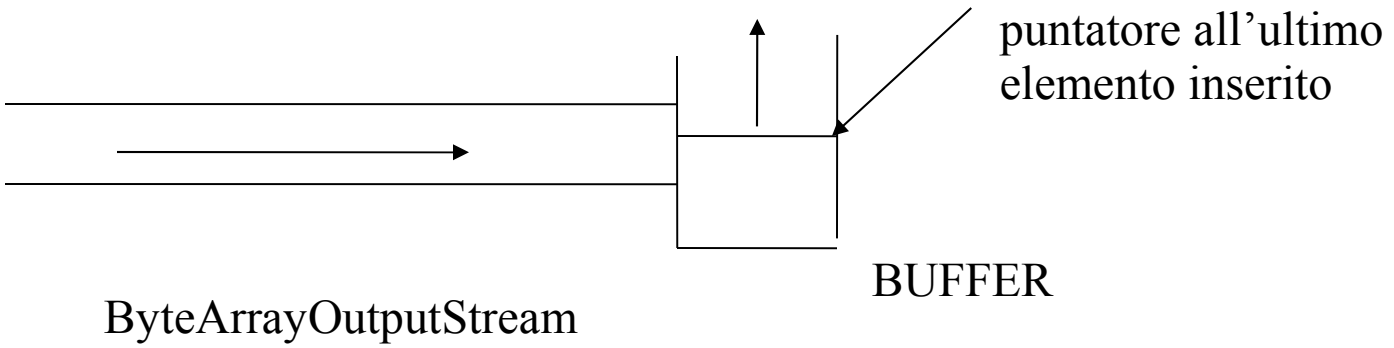


BYTEARRAY OUTPUT STREAMS

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 (array di byte) a dimensione variabile (dimensione di default = 32 bytes).
- quando il buffer si riempie la sua dimensione viene raddoppiata
- quindi consente di accedere ad un array di byte come se fosse uno stream; l'array può essere estratto con il metodo toByteArray()



JAVA: USO DEGLI STREAM PER LA PROGRAMMAZIONE DI RETE

Come utilizzeremo gli streams in questo corso:

- Trasmissione connectionless:
  - **ByteArrayOutputStream**, consentono la conversione di uno stream di bytes in un array di bytes da spedire con i pacchetti UDP
  - **ByteArrayInputStream**, converte un array di bytes in uno stream di byte. Consente di manipolare più agevolmente i bytes

- Trasmissione connection oriented:

  Una connessione viene modellata con uno stream.

  invio di dati = scrittura sullo stream

  ricezione di dati = lettura dallo stream

BYTEARRAY OUTPUT STREAMS NELLA COSTRUZIONE DI PACCHETTI UDP

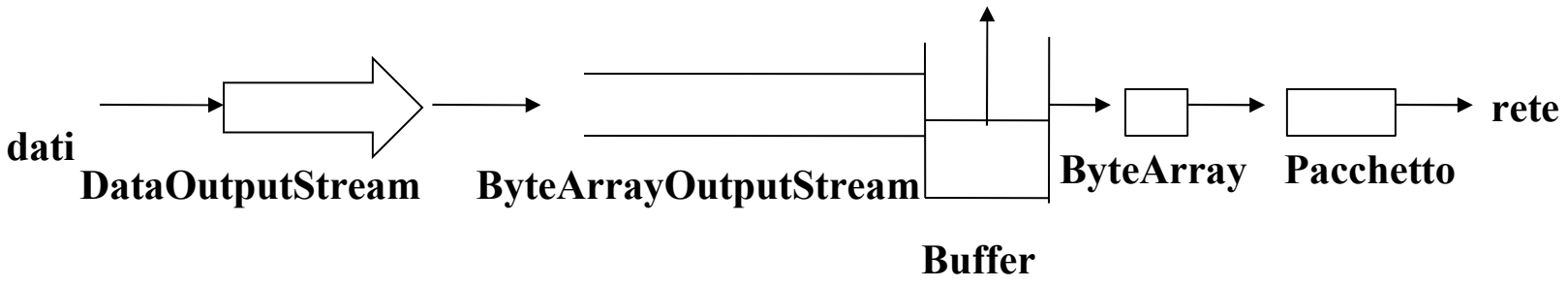
Ad un ByteArrayOutputStream può essere collegato un altro wrapper

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

Posso scrivere un dato di qualsiasi tipo sul **DataOutputStream( )**

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( )
```

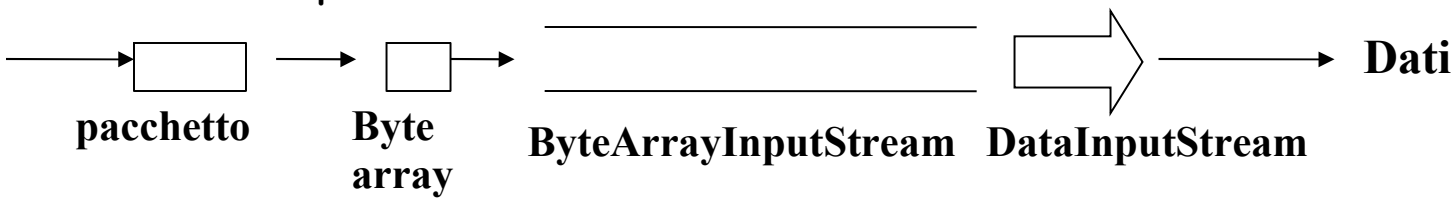


BYTEARRAY INPUT STREAMS

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

- creano stream di byte a partire dai dati contenuti nell'array di byte **buf**.
- il secondo costruttore copia **length** bytes iniziando alla posizione **offset**.
- E' possibile incapsularlo in 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!

INVIO DI UDP DATAGRAMS

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{  // inizializzazione
      InetAddress ia = InetAddress.getByName("localhost");
      int port = 13350;
      DatagramSocket ds = new DatagramSocket (  );
      byte [ ] data = new byte [20];
      DatagramPacket dp = new DatagramPacket(data, data.length, ia , port);
      ByteArrayOutputStream bout= new ByteArrayOutputStream( );
      DataOutputStream dout = new DataOutputStream (bout);
```

INVIO DI UDP DATAGRAMS

```
for (int i=0; i< 10; i++){
   dout.writeInt(i);                    // scrivo 4 bytes nello stream
   data = bout.toByteArray();           // estraggo l'array di byte
   dp.setData(data,0,data.length);      // lo inserisco nel DatagramPacket
   dp.setLength(data.length);           // definisco la lunghezza del buffer
   ds.send(dp);                         // invio il DatagramPacket sul socket
   bout.reset( );                       // svuoto lo stream
   dout.writeUTF("***");                // scrivo una stringa nello stream
   data = bout.toByteArray( );          // ...
   dp.setData (data,0,data.length);
   dp.setLength (data.length);
   ds.send (dp);
   bout.reset( );  }  }  }
```

RICEZIONE DI UDP DATAGRAMS

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);
```

RICEZIONE DI UDP DATAGRAMS

```
for (int i = 0; i < 10; i++){
     ds.receive(dp);                          // ricevo il DatagramPacket
     ByteArrayInputStream bin=                // getLength() è il numero di byte letti
               new ByteArrayInputStream(dp.getData(), 0, dp.getLength());
    DataInputStream ddis= new DataInputStream(bin);
    int x = ddis.readInt();                   // leggo un intero attraverso lo stream
    dr.writeInt(x);                           // lo scrivosul file
    System.out.println(x);                    // lo stampo
    ds.receive(dp);                           // ricevo altro DatagramPacket
    bin = new ByteArrayInputStream(dp.getData(), 0 ,dp.getLength());
    ddis = new DataInputStream(bin);
    String y = ddis.readUTF( );               // leggo una stringa
    System.out.println(y); }   }   }          // la stampo
```

Protocollo UDP: problemi

- 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 $\Rightarrow$ scritture/letture possono non corrispondere

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

PER ESEGUIRE GLI ESERCIZI 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 InetAddress.getLocalHost( ))
- Se l'host non è connesso in rete utilizzare l'indirizzo di loopback ("localhost" o 127.0.0.1)
- 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

ESERCIZIO 1: INVIO DI DATAGRAM UDP

Esercizio:

Scrivere un'applicazione composta da un processo Sender ed un processo Receiver. Il Receiver riceve da linea di comando la porta su cui deve porsi in attesa. Il Sender riceve da linea di comando una stringa e l'indirizzo del Receiver (indirizzo IP + porta), e invia al Receiver la stringa. Il Receiver riceve la stringa e stampa, nell'ordine, la stringa ricevuta, l'indirizzo IP e la porta del mittente.

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?

ESERCIZIO 1: INVIO DI DATAGRAM 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. Controllare l'output dei Receiver.
- 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.
- Modificare il codice ottenuto al passo precedente in modo che il Sender invii una sequenza di messaggi ai due 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.
- 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.

ESERCIZIO 2: COUNT DOWN SERVER

Si richiede di programmare un server CountDownServer che fornisce un semplice servizio: ricevuto da un client un valore intero n, il server spedisce al client i valori n-1,n-2,n-3,….,1, in sequenza.

La interazione tra i clients e CountDownServer è di tipo connectionless.

Si richiede di implementare due versioni di CountDownServer

- realizzare CountDownServer come un server iterativo. L'applicazione riceve la richiesta di un client, gli fornisce il servizio e solo quando ha terminato va a servire altre richieste
- realizzare CountDownServer come un server concorrente. Si deve definire un thread che ascolta le richieste dei clients dalla porta UDP a cui è associato il servizio ed attiva un thread diverso per ogni richiesta ricevuta. Ogni thread si occupa di servire un client.

Opzionale: Il client calcola il numero di pacchetti persi e quello di quelli ricevuti fuori ordine e lo visualizza alla fine della sessione.

Utilizzare le classi ByteArrayOutput/InputStream per la generazione/ricezione dei pacchetti.