LPR A 2005/2006
Lezione 4

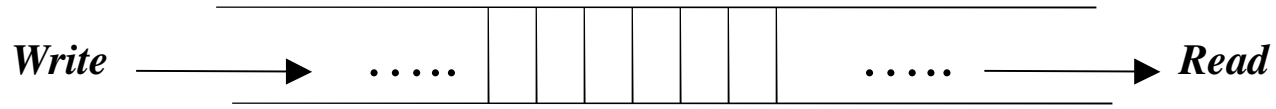
- JAVA streams
- invio di dati di tipo primitivo su socket UDP
- invio di oggetti
- il paradigma di programmazione client/server

JAVA: IL CONCETTO DI STREAM,

- *Studiare capitolo 4 Java Network , Elliotte Rusty Harold*

- 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:* flusso *continuo* di dati tra una sorgente ed una destinazione (dal programma ad un dispositivo o viceversa)



  - 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: applicazione scrive su un *FileOutputStream*. Il dispositivo legge i dati e li memorizza sul file

JAVA: IL CONCETTO DI STREAM

Principali caratteristiche 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…).
- non è richiesta una corrispondenza stretta tra letture/scritture

  esempio: una unica scrittura inietta 100 bytes sullo stream, he vengono letti con due write successive, la prima legge 20 bytes, la seconda 80 bytes)

JAVA: USO DEGLI STREAM PER LA PROGRAMMAZIONE DI RETE

Come utilizzeremo gli streams in questo corso:

- *trasmissione connectionless:*

  ByteArrayOutputStream, generano streams di bytes che possono essere convertiti in vettori di bytes da spedire con i pacchetti UDP

  ByteArrayInputStream, converte un vettore 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

JAVA: STREAMS DI BASE

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* offeset, *int* length) *throws* IOException;

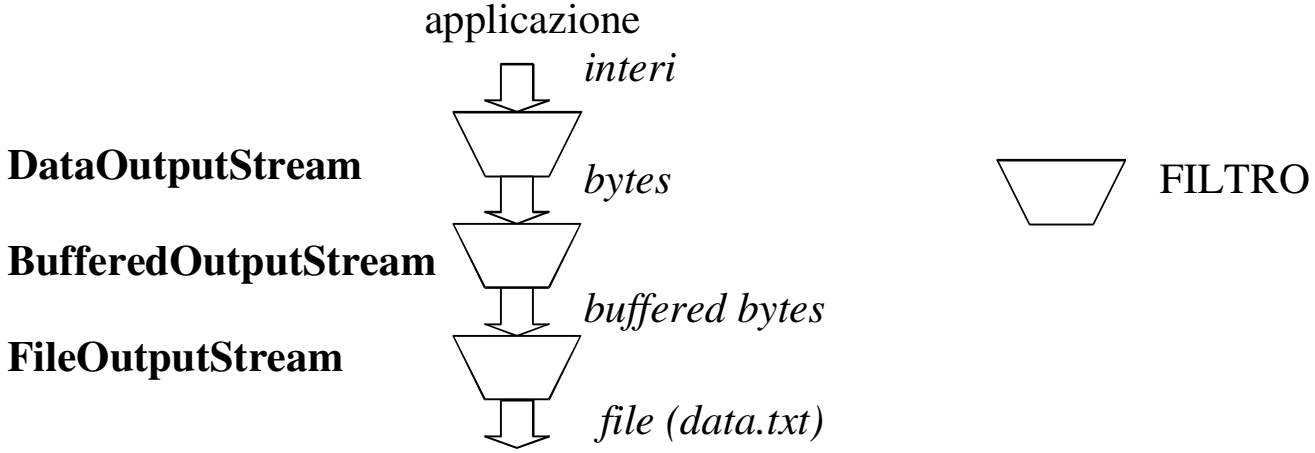
write (*int* b) scrive su un OuputStream il byte corrispondente all'intero passato

La classe *OutputStream* ed il metodo *write* sono dichiarati astratti. Le sottoclassi descrivono stream legati a particolari dispositivi di I/O (file, console,…)
L'implementazione del metodo *write* può richiedere codice nativo (es: scrittura su un file…).
Gli ultimi due metodi consentono la scrittura di gruppi di bytes.
Analogamente la classe *InputStream*

JAVA STREAMS: FILTRI

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 filtro*

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



JAVA STREAMS: FILTRI

DataOutputStream consente di trasformare dati di tipi primitivi 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 big-endian, complemento a due
- 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.

JAVA STREAMS: BUFFERIZZAZIONE

BufferedOutputStream

- memorizza una sequenza di bytes in un buffer B (un *byteArray*)

- *copia* i dati da B allo stream quando risulta verificata una delle seguenti condizioni:
  - B è *pieno*
  - viene effettuata una operazione di *flush sullo stream:*

    esempio: bos.*flush* ( ), se bos è un buffered outputstream

- la dimensione di B può essere stabilita al momento della costruzione del filtro, oppure stabilita per default (512 bytes), usando un costruttore opportuno

  public BufferedOutputStream(OutputStream out)

  public BufferedOutputStream(OutputStream out, *int* bufferSize)

JAVA STREAMS: BUFFERIZZAZIONE

- Se lo stream è associato ad una connessione TCP, è necessario valutare la relazione tra dimensione del buffer e quantità di dati da inviare sullo stream

- *Esempio:* un client utilizza una connessione TCP
  - invia una richiesta (300 bytes) ad un server HTTP. Utilizza un BufferOutputStream BS (dimensione del buffer = 1024 bytes)
  - attende risposta dal server prima di inviare una nuova richiesta

  $\Rightarrow$

  il client si blocca *indefinitamente,* perchè
  - i dati bufferizzati non riempiono completamente il buffer
  - il buffer non viene spedito
  - il client non riceve alcuna risposta dal server e non può inviare la nuova richiesta

- Soluzione: effettuare il *flush* dello stream dopo aver inviato la prima richiesta

JAVA STREAMS: INPUT STREAMS

- *public abstract class* InputStream

  metodi:
  - *public abstract int* read() *throws* IOException
  - *public int* read(byte[]input) *throws* IOException
  - *public long* skip(long n) *throws* IOException

$\Rightarrow$

- *Skip* consente di bypassare una certa sequenza di dati senza leggerli

- *BufferInputStream:* definisce un buffer B che viene riempito *on demand.* Quando l'utente richiede di leggere una sequenza di k bytes dallo stream, si controlla se i k bytes sono disponibili in B. In caso negativo si leggono dallo stream tutti gli n bytes disponibili sullo stream in quel momento.
  - se n>k i bytes rimanenti vengono memorizzati nel buffer per successive letture
  - se n<k la lettura si blocca

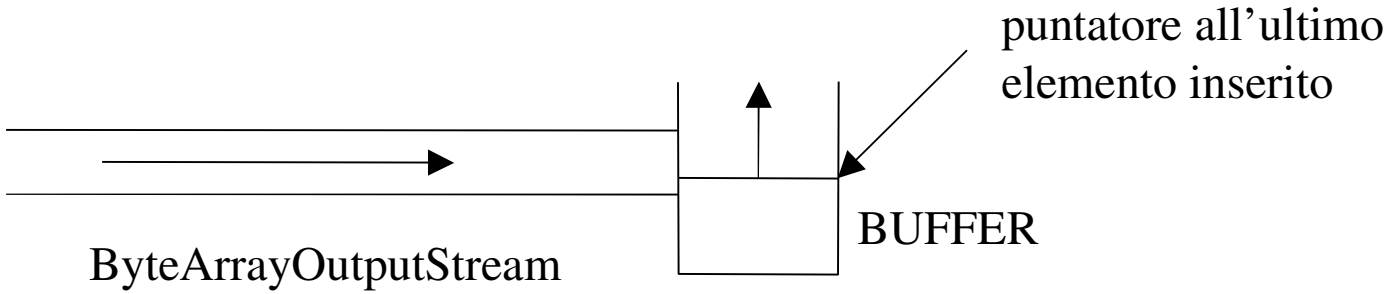
BYTE ARRAY INPUT/OUTPUT STREAMS

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* a *dimensione variabile* (dimensione di default = 32 bytes).

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



BYTE ARRAY INPUT/OUTPUT STREAMS NELLA COSTRUZIONE DI PACCHETTI UDP

È possibile

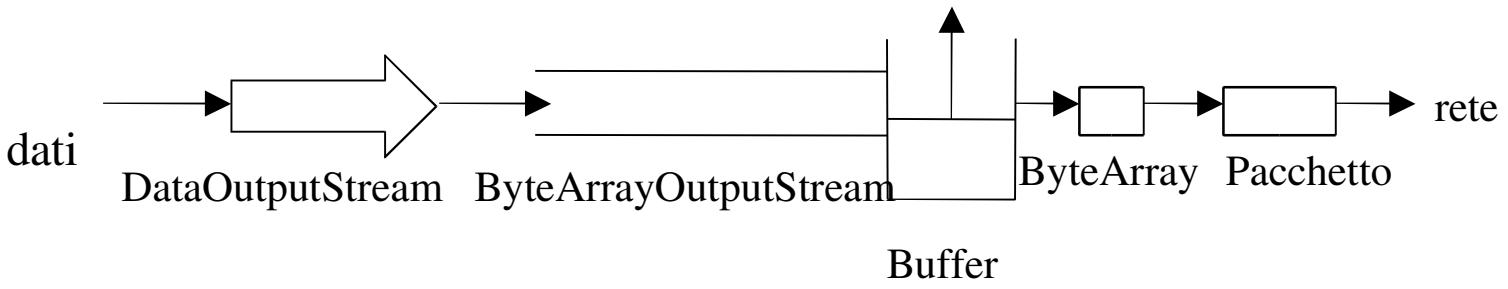
- collegare un filtro ad un *ByteArrayOutputStream*

  DataOutputStream do= *new DataOutputStream*(
  new ByteArrayOutputStream())

- copiare i dati presenti nel buffer B associato ad un *ByteArrayOutputStream* in un array di bytes, di dimensione uguale alla dimensione attuale di B

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

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



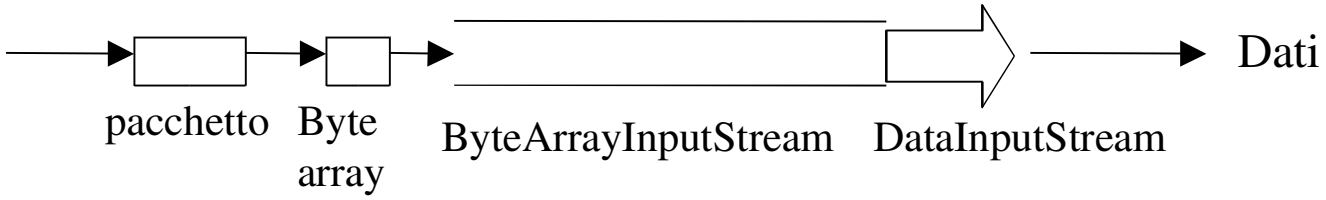
BYTE ARRAY INPUT/OUTPUT STREAMS

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

- Creano degli stream di byte a partire dai dati contenuti nel vettore di byte buf.
  Nel secondo caso copia length bytes iniziando alla posizione offset.

- E' possibile concatenare un *DataInputStream*

Ricezione di un pacchetto UDP dalla rete:



BYTE ARRAY INPUT/OUTPUT STREAMS

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

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

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

  baos.*reset*()

  il metodo toByteArray *non svuota il buffer!*

BYTE ARRAY INPUT/OUTPUT STREAMS

Ipotesi semplificativa: non consideriamo perdita/riordinamento dei pacchetti

```
import java.io.*;
import java.net.*;
public class multidatastreamsender
{public static void main(String args[]) throws Exception
    {
    // fase di inizializzazione

      InetAddress ia=….;
      int port=…;
      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=1; 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

```
import java.io.*;
import java.net.*;
public class MultiDataStreamReceiver
{public static void main(String args[]) throws Exception
    {
    // fase di inizializzazione


      int port =…;
      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=1; i<10; i++)
      {ds.receive(dp);
       ByteArrayInputStream bin= new
           ByteArrayInputStream(dp.getData,0,dp.getLength())
       DataInputStrea m 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();
       }}
```

BYTE ARRAY INPUT/OUTPUT STREAMS

- Problema: applicazioni *sensibili* alla perdita/riordino di pacchetti

- *Esempio:* 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 dai pacchetti ricevuti

  se un pacchetto viene perso $\Rightarrow$ il destinatario scritture/letture possono non corrispondere

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

INVIO DI OGGETTI SULLA RETE: LA SERIALIZZAZIONE

Serializzazione:

- consente di *convertire un qualsiasi oggetto* che implementa la *interfaccia serializable* in una *sequenza di bytes.*

- tale sequenza può successivamente essere utilizzata per *ricostruire* l'oggetto.

- l'oggetto deve essere definito mediante una classe che implementi l'interfaccia *serializable.*

- Utilizzare stream di tipo *ObjectOutputStream*(rs. *ObjectInputStream*)
  e metodi *writeObject* (rs. *readObjec*t).

INVIO DI OGGETTI SULLA RETE: LA SERIALIZZAZIONE

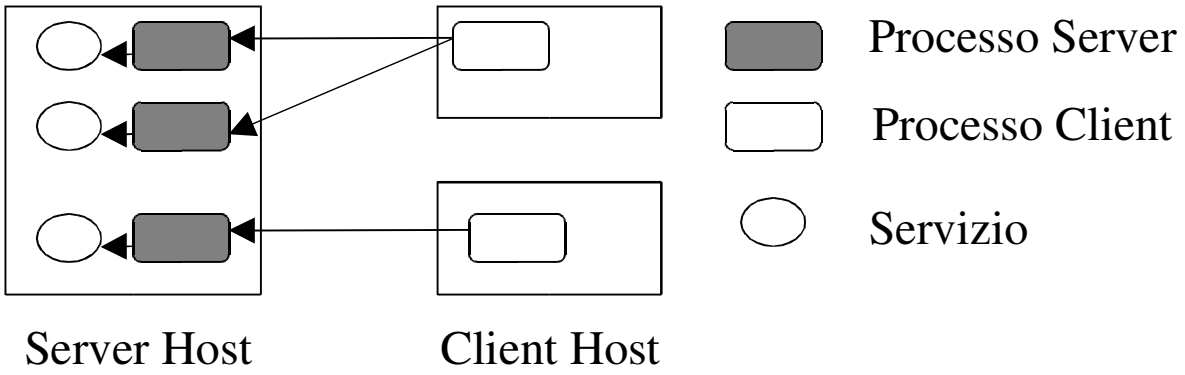
Esempio: un server ServerScuola gestisce un registro di classe. Un client può contattare il server inviandogli il nome di uno studente e riceve come risposta il numero di assenze giustificate ed il numero di assenze ingiustificate dello studente.

Il server può inviare al client una struttura con due campi interi (numero assenze giustificate, numero assenze ingiustificate).

```
public class assenze implements serializable
{     private int nassgiustificate;
      private int nassingiustificate
……………}
```

IL PARADIGMA DI PROGRAMMAZIONE CLIENT/ SERVER

- *Studiare capitolo 5 Liu*
- architettura client/server: un host (il server) fornisce accesso a risorse come stampanti, files…. Gli altri hosts (i clients) accedono a tali risorse mediante il server.
- *paradigma di programmazione client/server*: un processo(il server) fornisce qualche *servizio di rete* (esempio accesso ad un insieme di pagine web). I processi clients richiedono al server tale servizio.



IL PARADIGMA DI PROGRAMMAZIONE CLIENT/ SERVER

- per individuare un servizio di rete

  - indirizzo IP del server host + numero di porta che individua il server process
  - registrazione del servizio mediante *un nome logico* in una directory (registry) + mapping nome logico indirizzo del server process (meccanismo utilizzato da RMI)

- *sessione* = sequenza di interazioni tra il client ed il server

  durante la sessione il client ed il server devono osservare un insieme di *regole (protocollo)* che stabiliscono

  - la sequenza di interazioni
  - la rappresentazione dei dati scambiati

IL PARADIGMA DI PROGRAMMAZIONE CLIENT/ SERVER

- Struttura ad alto livello di un server sequenziale

  repeat

  accetta la richiesta di connessione da un client C

  esecuzione della sessione con C

  forever

- *Alcuni servizi elementari:*
  - *DayTime Server (porta 13)* : la sessione comprende una *singola interazione* tra client e server. Il client contatta il server ed il server risponde inviando la data e l'ora.
  - *Echo Server(porta 7)*: la sessione comprende una *sequenza di interazioni*; ogni interazione comprende l'invio di una stringa da parte del client;il server restituisce la medesima stringa al client (utile per testare l'affidabilità della rete)

IL PARADIGMA DI PROGRAMMAZIONE CLIENT/ SERVER

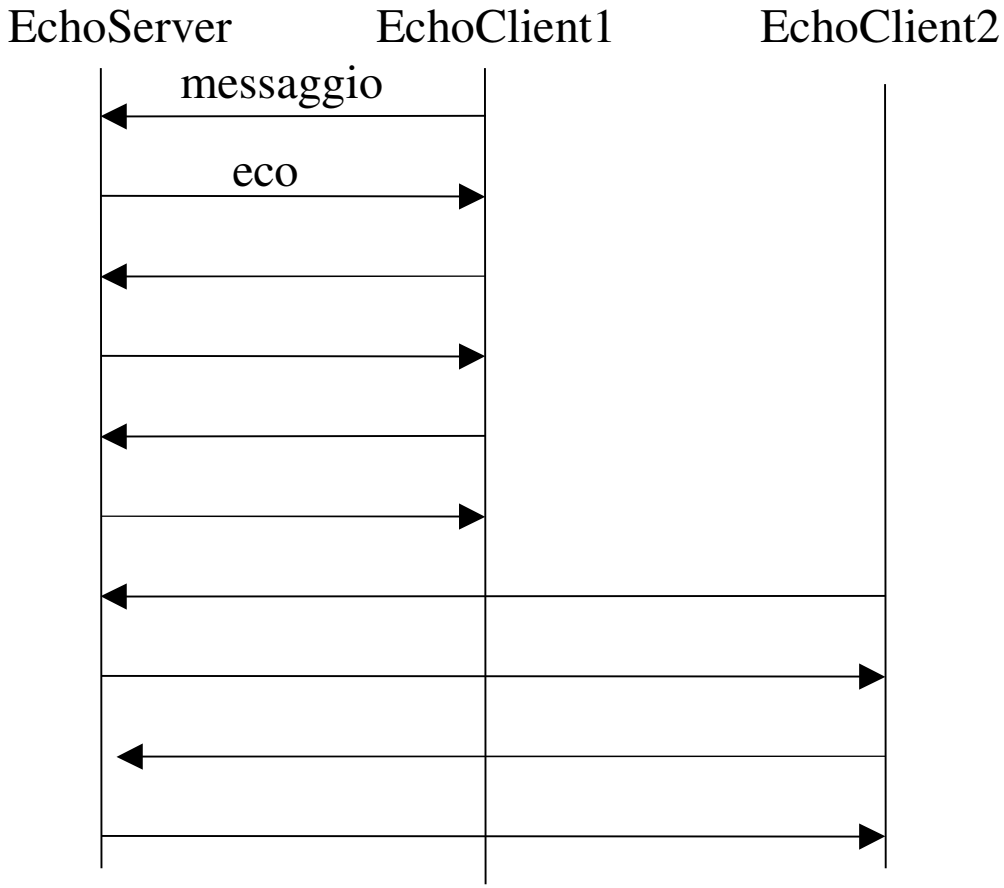
Struttura di un server: *connection oriented* / *connectionless*

Connection Oriented

- il client richiede una connessione
- il server accetta una richiesta, in modo *non deterministico* e crea una connessione con il client
- client e server eseguono una sequenza di interazioni sulla connessione creata
- Il client chiede di chiudere la connessione

Per servers sequenziali: nessun interleaving tra sessioni di utenti diversi
Un utente deve attendere che il server abbia terminato il servizio di una richiesta precedente, prima di iniziare ad essere servito

IL PARADIGMA DI PROGRAMMAZIONE CLIENT/ SERVER: SERVERS CONNECTION ORIENTED



IL PARADIGMA DI PROGRAMMAZIONE CLIENT/ SERVER

Struttura di un server: *connection oriented* / *connectionless*

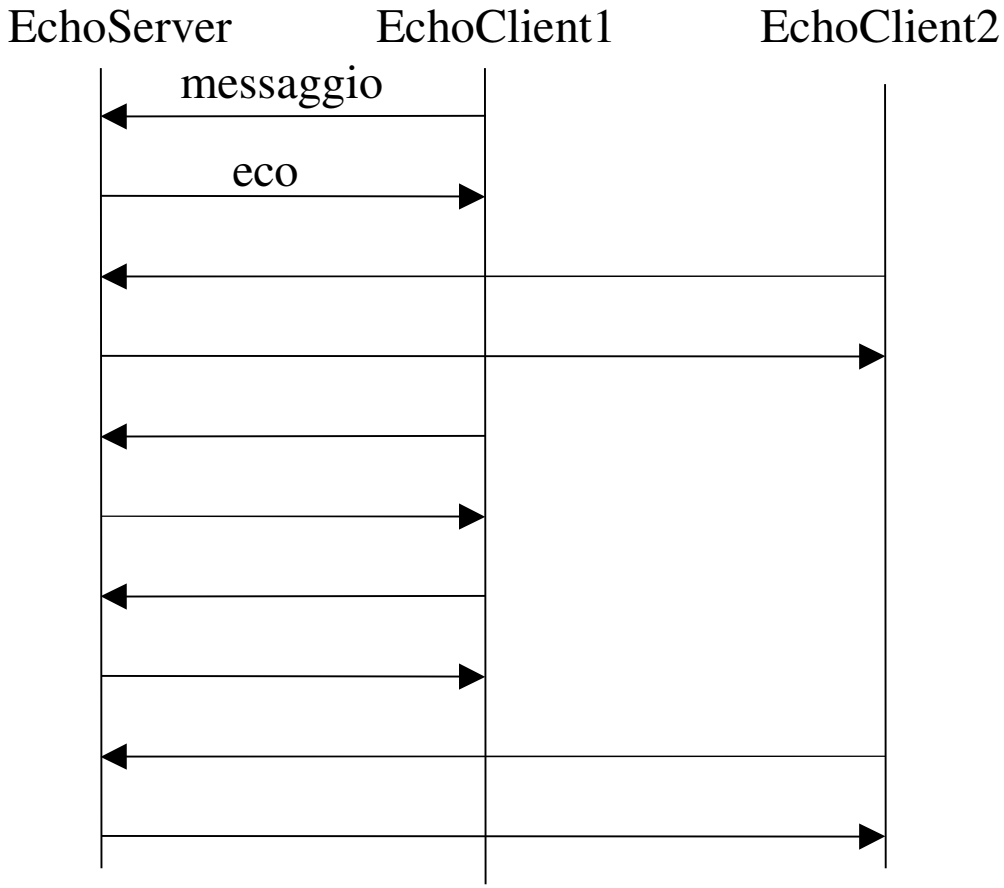
Connectionless

- Tutti i clients inviano una richiesta di servizio sullo stesso socket e sulla stessa porta

- il server fornisce il servizio

- non esiste il concetto di connessione: in casi semplici (come echo server) è possibile *un interleaving* tra sessioni di clients diversi

IL PARADIGMA DI PROGRAMMAZIONE CLIENT/SERVER SERVERS CONNECTIONLESS



IL PARADIGMA DI PROGRAMMAZIONE CLIENT/ SERVER

Struttura di un server: *iterativo* / *concorrente*

Server iterativo: non consente overlapping di sessioni di clienti diversi.
Esempio: EchoServer connection oriented, viene gestita una connessione per volta

Server concorrente: consente di gestire sessioni di utenti diversi in modo concorrente

- attivazione di un thread per ogni connessione
- utilizzo di sockets non bloccanti

IL PARADIGMA DI PROGRAMMAZIONE CLIENT/ SERVER

Struttura di un server: *con/senza stato*

Server senza stato: non viene mantenuta alcuna informazione tra interazioni della solita sessione o tra sessioni diverse (DayTime, EchoServer)

Server con stato:

- stato mantenuto all'interno della stessa sessione. *Esempio: ftp*, trasferimento di file a blocchi. Occorre ricordare il numero dell'ultimo blocco spedito
- stato mantenuto tra sessioni diverse: *Esempio: Counter Protocol,* quando un client si connette ad un server, quest'utlimo gli invia il numero di connessioni ricevute fino a quel momento.

ESERCIZIO: PARADIGMA CLIENT/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 rgenerazione/ricezione dei pacchetti.

ESERCIZIO:
CALCOLO DISTRIBUITO DEL MASSIMO

Si consideri una grande azienda produttrice di computers che possiede n filiali sparse sul territorio. Ogni filiale commercializza i 10 modelli di computers prodotti dalla azienda.
Si supponga che ogni filiale voglia conoscere, per *ogni modello di computer*, il nome della filiale che possiede nel suo magazzino il *maggior numero* di computers di quel modello.
Si attivi, per ogni filiale un programma, *peerfiliale*, che memorizzi in un file le giacenze dei computers presso quella filiale. Ogni peerfiliale è individuato da un indice univoco. I peerfiliale sono interconnessi mediante una *struttura logica di interconnessione ad anello* (ogni peerfiliale può solamente comunicare con il suo successore ed il suo predecessore sull'anello),
Non esiste un server centralizzato e non si possono definire altre comunicazioni oltre quelle sull'anello. La computazione del valore massimo per ogni modello è innescata dal PeerSurvive di indice 0. *Ipotesi semplificativa:* trascurare la eventuale perdita/riordinamento dei pacchetti.