



Lezione n.7

LPR-B-08

**UDP: Costruzione di
pacchetti, e esercizi avanzati**

10/11/2009

Andrea Corradini

SOMMARIO

- Invio di dati tramite UDP
 - codifica di dati primitivi come array di byte
 - invio di oggetti tramite serializzazione
- Discussione di esercizi avanzati
 - MiniTalk (Instant Messenger)
 - handshaking per stabilire una "connessione"
 - invio bidirezionale di stringhe
 - TFTP (Trivial File Transfer Protocol)
 - protocollo per trasferimento di file in lettura/scrittura tra client e server
 - simulazione di trasmissione basata su stream

ABBIAMO VISTO CHE IN JAVA...

- Il protocollo UDP prevede la trasmissione di una **sequenza (array) di bytes** incapsulato in un **DatagramPacket**.
- Possiamo trasformare dati primitivi in **sequenze di bytes** da inserire all'interno del pacchetto usando
 - la classe **DataOutputStream** e i metodi **writeInt()**, **writeDouble()**, ...
 - la classe **ByteArrayOutputStream** e il metodo **toByteArray()**
- I dati possono essere ricostruiti dal mittente mediante
 - la classe **ByteArrayInputStream**
 - la classe **DataInputStream** e i metodi **readInt()**, **readDouble()**, ...
- Possiamo inviare più dati primitivi nello stesso pacchetto UDP, rileggendoli nello stesso ordine

SENDER: INVIA PIU' DATI IN UN PACCHETTO

```
byte byteVal = 101;
```

```
short shortVal = 10001;
```

```
int intVal = 100000001;
```

```
long longVal = 1000000000001L;
```

```
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( ); // Va inserito nel DatagramPacket  
// e spedito sul DatagramSocket
```

RECEIVER: ESTRAE PIU' DATI DAL PACCHETTO

...

```
ds.receive(dp);           // si riceve il DatagramPacket
byte byteValIn;
short shortValIn;
int intValIn;
long longValIn;
ByteArrayInputStream bufIn =
    new ByteArrayInputStream(dp.getData(), 0, dp.getLength());
DataInputStream in = new DataInputStream(bufIn);
byteValIn = in.readByte();
shortValIn = in.readShort();
intValIn = in.readInt();
longValIn = in.readLong();
```

CODIFICARE DATI PRIMITIVI COME BYTE[]

- Il protocollo UDP (e come vedremo anche TCP) consente di inviare unicamente **sequenze di bytes**. Un byte viene interpretato come un intero nell'intervallo [0..255].
- La **codifica** dei tipi di dato primitivi di un linguaggio in **sequenze di bytes** può essere realizzata
 - dal supporto del linguaggio (come visto nei lucidi precedenti)
 - più a basso livello, esplicitamente dal programmatore
- In ogni caso, il mittente ed il destinatario **devono accordarsi sulla codifica stabilita**. Ad esempio, per un dato di tipo intero si deve stabilire:
 - la **dimensione** in bytes per ogni tipo di dato
 - l'**ordine** dei bytes trasmessi,
 - l'**interpretazione** di ogni byte (con segno/senza segno)
- Il problema è semplificato se mittente e destinatario sono codificati mediante lo stesso linguaggio (ad esempio entrambi in JAVA)

CODIFICARE VALORI DI TIPO PRIMITIVO

Per scambiare valori di **tipo intero**, occorre concordare:

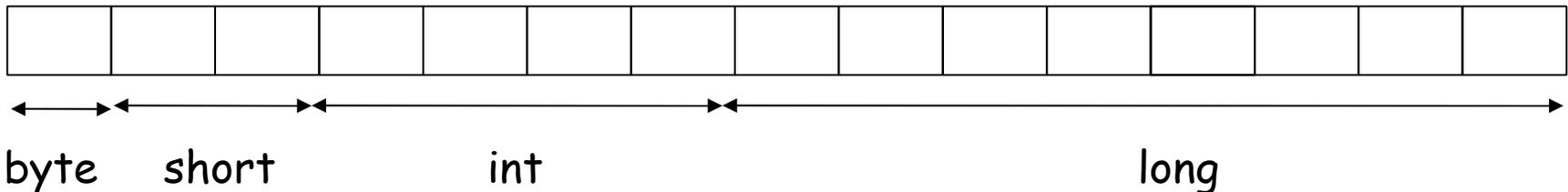
- dimensione dei tipi di dati scambiati: **long**: 8 bytes, **int**: 4 bytes, **short**: 2 bytes
- ordine dei bytes trasmessi
 - **Little-endian**: il primo byte trasmesso è il meno significativo
 - **Big-endian**: il primo byte trasmesso è il più significativo
- Interpretazione dei valori: con/senza segno

Nel caso di valori di **tipo stringa**, occorre concordare

- codifica adottata per i caratteri contenuti nella stringa
 - UTF-8,
 - UTF-16

CODIFICARE VALORI DI TIPO INTERO

- Supponiamo di dover costruire un pacchetto UDP contenente quattro valori interi: un **byte**, uno **short**, un **intero** ed un **long**



- **Non si vogliono** utilizzare le classi filtro `DataOutputStream` e `ByteArrayOutputStream`
- **Soluzione alternativa:** si utilizzano operazioni che operano direttamente sulla rappresentazione degli interi
 - si definisce **message**, un vettore di bytes
 - si **selezionano** i bytes della rappresentazione mediante **shifts a livello di bits**
 - si inserisce ogni byte selezionato in una posizione del vettore **message**

CODIFICARE VALORI DI TIPO INTERO

```
public static int encodePacket(byte[ ] dst, int offset, long val, int size){  
    for (int i = size; i > 0; i--){  
        dst[offset++] = (byte) (val >> ((i - 1) * 8));  
    }  
    return offset;  
}
```

- `val` valore intero
- `size` dimensione, in bytes, del valore `val`
- `i` bytes che rappresentano `val` devono essere inseriti nel vettore `dst`, a partire dalla posizione `offset`
- si utilizza lo `shift destro` per selezionare i bytes, a partire dal più significativo
- il `cast a byte` del valore shiftato `V` restituisce gli 8 bits meno significativi di `V`, eliminando gli altri bits

CODIFICARE VALORI DI TIPO INTERO

```
public static void main(String args[ ])
{ byte byteVal=101, short shortVal = 8, int intVal = 53,
  long longVal = 234567L;
  final int BSIZE = 1;
  final int SSIZE= Short.SIZE / Byte.SIZE;
  final int ISIZE = Integer.SIZE/ Byte.SIZE;
  final int LSIZE = Long.SIZE / Byte.SIZE;
  byte [ ] message = new byte[BSIZE+SSIZE+ISIZE+LSIZE];
  int offset = encodePacket(message,0, byteVal, 1);
  offset = encodePacket(message,offset, shortVal, SSIZE);
  offset = encodePacket(message,offset,intVal, ISIZE);
  offset = encodePacket(message,offset, longVal, LSIZE);
  .....
```

DECODIFICARE VALORI DI TIPO INTERO

```
public static long decodePacket(byte[] val, int offset, int size){  
    long rtn =0; int BYTEMASK = 0xFF;  
    for (int i= 0; i < size; i++){  
        rtn = (rtn << 8) | (((long) val[offset +i] & BYTEMASK); }  
    return rtn; }
```

- `decodePacket`: decodifica il valore di un dato rappresentato dai bytes contenuti nel `vettore val`, a partire dalla `posizione offset`. La dimensione (in byte) del valore da decodificare è `size`.
- Se si vuole 'riassemblare' il valore di tipo `short` dell'esempio precedente:

```
short value = (short) decodePacket(message, BSIZE, SSIZE);
```
- `"& BYTEMASK"` è necessario per mettere a zero i bit più significativi (posizioni 0-55), che potrebbero essere ad 1 per il cast. Si provi il comando:

```
System.out.println((short) (byte) 128); // Stampa -128 ! Perché ?
```

CODIFICA DI STRINGHE

- I caratteri vengono codificati in JAVA mediante **Unicode** che mappa i caratteri ad interi nell'intervallo [0..65535]
- **Unicode** è compatibile all'indietro con la codifica **ASCII**
- Il metodo **getBytes()** applicato ad una stringa restituisce un array di bytes contenente la rappresentazione della stringa, ottenuta secondo la codifica utilizzata di default dalla piattaforma su cui il programma viene eseguito
- E' possibile indicare esplicitamente la codifica desiderata, come argomento della **getBytes()**
- Esempio.
 - "Test!".getBytes()
 - "Test!".getBytes("UTF-16BE")
- In generale mittente e destinatario devono accordarsi sulla codifica utilizzata per i valori di tipo stringa

SERIALIZZAZIONE DI OGGETTI

- Le classi `ObjectInputStream` e `ObjectOutputStream` definiscono streams (basati su streams di byte) su cui si possono leggere e scrivere oggetti.
- La scrittura e la lettura di oggetti va sotto il nome di **serializzazione**, poiché si basa sulla possibilità di scrivere lo stato di un oggetto in una forma **sequenziale**, sufficiente per ricostruire l'oggetto quando viene riletto. La serializzazione di oggetti viene usata in diversi contesti:
 - Per inviare oggetti sulla rete, sia che si utilizzino i protocolli UDP o TCP, sia che si utilizzi RMI.
 - Per fornire un meccanismo di **persistenza** ai programmi, consentendo l'archiviazione di un oggetto per esempio in un file. Si pensi ad esempio ad un programma che realizza una rubrica telefonica o un'agenda.

SERIALIZZAZIONE DI OGGETTI

- Consente di convertire un oggetto E TUTTI QUELLO DA ESSO RAGGIUNGIBILI in una **sequenza di bytes**.
- Tale sequenza può essere utilizzata per **ricostruire** l'oggetto e il contesto.
- L'oggetto e tutti quelli raggiungibili devono essere definiti in classi **che implementi l'interfaccia `Serializable`**, che **non ha metodi!** Altrimenti viene lanciata una **`NotSerializableException`**.
- Tutte le classi involucro per i tipi di dati primitivi (es: **`Integer`**, **`Double`**, ...) e anche **`String`** implementano **`Serializable`**: quindi JAVA garantisce una serializzazione di default per tutti i dati primitivi.
- Per serializzare uno o più oggetti si utilizzano stream di tipo **`ObjectOutputStream`** e **`ObjectInputStream`**, e i rispettivi metodi **`writeObject()`** e **`readObject()`**.
- Il meccanismo di serializzazione può essere personalizzato, ma non lo vediamo...

LA CLASSE `ObjectOutputStream`

`public ObjectOutputStream (OutputStream out) throws Exception`

Quando si costruisce un oggetto di tipo `ObjectOutputStream`, viene automaticamente registrato in testa allo stream un header, costituito da due short, 4 bytes

(costanti `MAGIC NUMBER + NUMERO DI VERSIONE`)

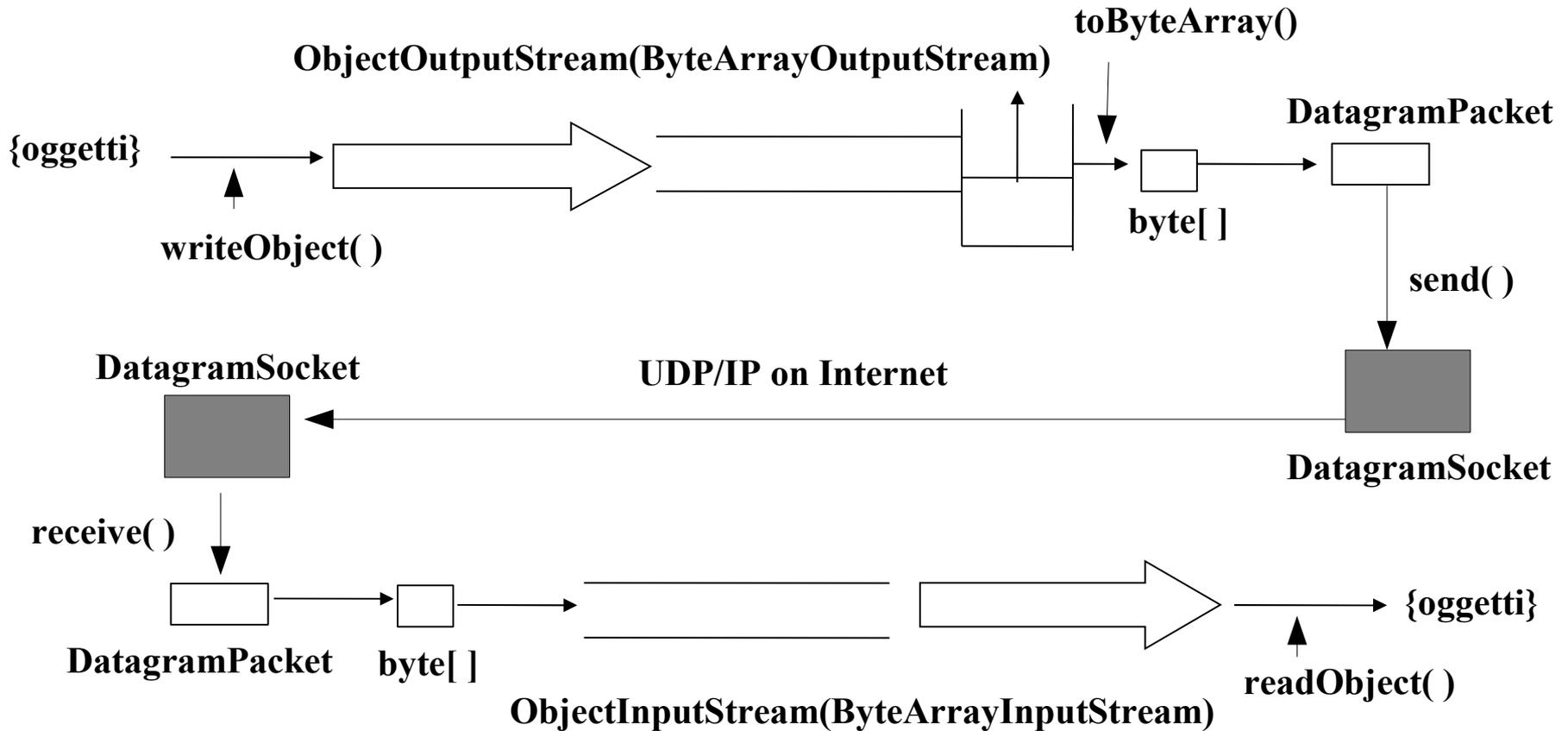
- `Magic Number` = identifica univocamente un object stream
- I Magic Number vengono utilizzati in diversi contesti. Ad esempio, ogni struttura contenente la definizione di una classe Java deve iniziare con un numero particolare (magic number), codificato mediante una sequenza di 4 bytes, che identificano che quella struttura contiene effettivamente una classe JAVA [3405691582, 0xCAFEBABE]
- Se l'header viene cancellato lo stream risulta corrotto e l'oggetto non può essere ricostruito. Infatti al momento della ricostruzione dell'oggetto si controlla innanzi tutto che l'header non sia corrotto

LA CLASSE `ObjectInputStream`

`public ObjectInputStream (InputStream in) throws Exception`

- L'header inserito dal costruttore di `ObjectOutputStream` viene letto e decodificato dal costruttore `ObjectInputStream`
- Se il costruttore `ObjectInputStream()` rileva qualche problema nel leggere l'header (ad esempio l'header è stato modificato o cancellato) viene segnalato che lo stream **risulta corrotto**
- L'eccezione sollevata è `StreamCorruptedException`

TRASMISSIONE DI OGGETTI, SCHEMATICAMENTE



UN ESEMPIO DI TRASMISSIONE DI OGGETTI

```
import java.io.*; import java.net.*; import java.util.Vector;
public class UDP_SendObject {
    public static void main(String[] args) throws IOException {
        // build interesting object
        Vector<Object> vett = new Vector<Object>();
        vett.add(new Integer(74));
        vett.add("Questa e' una stringa");
        vett.add(new java.awt.Rectangle(3,4,10,20));
        // prepare data to be sent using object stream
        ByteArrayOutputStream bout = new ByteArrayOutputStream();
        ObjectOutputStream oout = new ObjectOutputStream(bout);
        oout.writeObject(vett);
        byte[] arr = bout.toByteArray();
    }
}
```

UN ESEMPIO DI TRASMISSIONE DI OGGETTI

```
// continua la classe UDP_SendObject
// prepare Datagram socket and packet
DatagramSocket ds = new DatagramSocket();
InetAddress ia = InetAddress.getByName("localhost");
int port = 24309;
DatagramPacket dp = new DatagramPacket(arr,arr.length,ia,port);
// send
ds.send(dp);
}
}
```

UN ESEMPIO DI TRASMISSIONE DI OGGETTI

```
import java.io.*; import java.net.*; import java.util.Vector;
public class UDP_ReceiveObject {
    public static void main(String[] args)
        throws IOException,SocketException,ClassNotFoundException{
        DatagramSocket ds = new DatagramSocket(24309);
        DatagramPacket dp = new DatagramPacket(new byte[512],512);
        ds.receive(dp); // receive packet
        ByteArrayInputStream bais =
new ByteArrayInputStream(dp.getData(),dp.getOffset(),dp.getLength());
        ObjectInputStream ois = new ObjectInputStream(bais);
        Vector<Object> vett = (Vector<Object>) ois.readObject();
        for (Object o : vett){ System.out.println(o.getClass() + " : " + o); } } }
```

UN ESEMPIO DI TRASMISSIONE DI OGGETTI

L'output del programma mostra che il vettore e il suo contenuto sono arrivati correttamente al receiver:

```
class java.lang.Integer: 74
```

```
class java.lang.String: Questa e' una stringa
```

```
class java.awt.Rectangle: java.awt.Rectangle[x=3,y=4,width=10,height=20]
```

ATTENZIONE: UN ObjectOutputStream PER PACCHETTO!

```
public class Test {  
    public static void main(String args[ ]) throws Exception{  
        ByteArrayOutputStream bout = new ByteArrayOutputStream ( );  
        System.out.println (bout.size( )); // Stampa 0  
        ObjectOutputStream out= new ObjectOutputStream(bout);  
        System.out.println (bout.size( )); // Stampa 4: l'header è creato  
        out.writeObject("prova");  
        System.out.println (bout.size( )); //Stampa 12  
        // Spedisco il contenuto  
        bout.reset ( ); //Necessario per eliminare bytes già spediti  
        out.writeObject("prato");  
        System.out.println (bout.size( )); //Stampa 8 = 12-4.    }  
    }  
}
```

- la **reset()** ha distrutto l'header dello stream!!!

ESERCIZIO 1: UNA CLASSE AUSILIARIA PER TRASMETTERE OGGETTI

Scrivere la classe **Obj2DP** che fornisce due metodi statici:

- **public static** Object **dp2obj**(DatagramPacket dp)
che restituisce l'oggetto contenuto nel pacchetto passato per argomento, deserializzandolo
- **public static** DatagramPacket **obj2dp**(Object obj)
che restituisce un pacchetto contenente l'oggetto passato per argomento, serializzato, nel payload.
- Semplificare le classi **UDP_SendObject** e **UDP_ReceiveObject** viste prima usando i metodi della classe **Obj2DP**, senza più usare le classi **ObjectOutput/InputStream** e **ByteOutput/InputStream**.
- Usare la classe **Obj2DP** per i prossimi esercizi, trasmettendo oggetti serializzati con UDP invece di dati di tipi primitivi.

ESERCIZIO 2: MiniTalk Server e Client con UDP (1)

Si realizzino un Server e un Client UDP che realizzano un semplice Instant Messenger: ogni linea scritta nella shell del Server viene copiata nella shell del Client e viceversa. La trasmissione delle linee inizia dopo una semplice fase di handshaking, descritta di seguito.

- Il Server viene lanciato da shell, fornendo il numero di porta su cui ricevere i pacchetti UDP.
- Il Client viene lanciato da un'altra shell (eventualmente su un altro computer), fornendo host e porta del Server e porta locale su cui ricevere pacchetti UDP.
- Il Client manda una richiesta di connessione al Server, indicando nel messaggio la propria porta. Se non riceve un messaggio di ack entro 3 secondi, riprova a mandare la richiesta altre 5 volte, poi termina. Se invece riceve l'ack, inizia la trasmissione delle linee scritte nella shell locale e la ricezione e stampa delle linee scritte nella shell del Server.

ESERCIZIO 2: MiniTalk Server e Client con UDP (2)

- Quando il Server riceve una richiesta di connessione, recupera indirizzo e porta del Client dalla richiesta e manda un messaggio di ack al Client, quindi comincia la trasmissione e ricezione delle stringhe come descritto per il Client.
- Il Client decide quando disconnettersi in base a una condizione a vostra scelta (per esempio, allo scadere di un timeout, oppure se la linea da mandare è vuota, oppure se la stringa è "CLOSE",...).
- Per disconnettersi, il Client manda una richiesta di disconnessione al Server e aspetta un ack, quindi termina l'esecuzione.
- Quando il Server riceve una richiesta di disconnessione interrompe la trasmissione delle linee, manda un messaggio di ack, e si rimette in attesa di una richiesta di connessione.

Il Client e il Server devono scambiarsi **unicamente** oggetti della classe **TalkMsg**, usando i metodi della classe per crearne istanze e per ispezionare i messaggi arrivati.

ESERCIZIO 3: MiniTalk Messenger con UDP

Riusando il più possibile il codice sviluppato per l'esercizio precedente, realizzare un programma Messenger che offre le stesse funzionalità, ma in cui non si distinguono un Server e un Client.

Due istanze di Messenger devono essere lanciate in due shell diverse, fornendo ad ognuna tre dati: la porta locale, e l'host e la porta dell'altro Messenger. Ideare un opportuno protocollo di handshaking, che permetta di stabilire una connessione (concettuale) tra le due istanze di Messenger.

I messaggi scambiati devono essere tutti oggetti di una stessa classe. Usare la classe **TalkMsg**, oppure estenderla o definirne una analoga se necessario.

ESERCIZIO 4: TFTP con UDP

(Trivial File Transfer Protocol) [1]

Questa è la specifica di TFTP da WIKIPEDIA:

- L'host A invia un pacchetto RRQ (read request) o WRQ (write request) all'host B, contenente il nome del file e la modalità di trasferimento.
- B risponde con un ACK (acknowledgement) packet, che serve anche a dire ad A quale porta sull'host B dovrà usare per i restanti pacchetti.
- L'host di origine invia dei pacchetti DATA numerati all'host di destinazione, tutti tranne l'ultimo contenenti un blocco di dati completo. L'host di destinazione risponde con un pacchetto ACK numerato per ogni pacchetto DATA.
- Il pacchetto DATA finale deve contenere un blocco di dati non pieno ad indicare che si tratta dell'ultimo. Se la dimensione del file trasferito è un multiplo esatto della dimensione dei blocchi, la sorgente invia un ultimo pacchetto di dati contenente 0 byte di dati.

ESERCIZIO 4: TFTP con UDP

(Trivial File Transfer Protocol) [2]

Realizzare un Server TFTP che implementa il comportamento dell'host B e un Client TFTP che implementa l'host A. In particolare:

- Client e Server devono scambiarsi solo oggetti di una classe, **TFTPmsg**, usati sia per messaggi di servizio (RRQ, WRQ, ACK) che per i pacchetti DATA: definire opportunamente la classe **TFTPmsg**.
- Per il trasferimento dei file, considerarli come file binari, usando quindi opportuni Output/InputStreams (e non Writer/Reader).
- Inviare le porzioni di file in array di byte all'interno di un'istanza di **TFTPmsg**.

Per testare il programma:

- Confrontare il file originale spedito dal mittente con quello ricevuto dal destinatario e scritto nel file system.
- Usare la classe `UnreliableDatagramSocket` per controllare che i pacchetti persi vengano reinviati correttamente.