

Reti e Laboratorio III

Modulo Laboratorio III

AA. 2022-2023

docente: Laura Ricci

laura.ricci@unipi.it

Lezione 10

UDP

DATAGRAMPACKET E DATAGRAMSOCKET

24/11/2022

TCP ED UDP: CONFRONTO

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

“Timely, rather than orderly and reliable delivery”

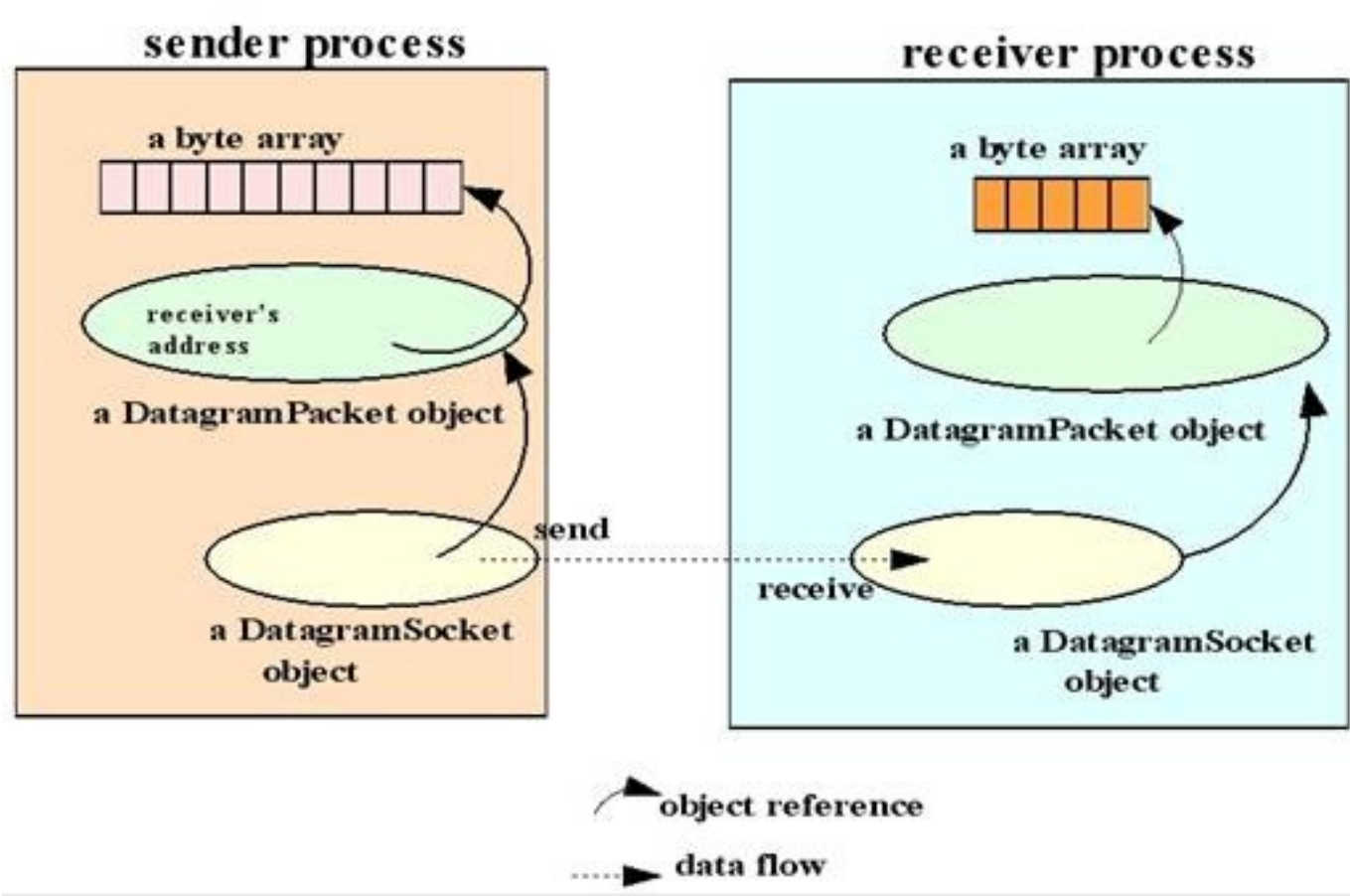
UDP: QUANDO USARLO?

- 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
- alcuni servizi su UDP: DNS, prime versioni di NFS, TFTP (retrivial file transfer protocol), alcuni protocolli peer-to-peer

CONNECTION ORIENTED VS. CONNECTIONLESS

- JAVA socket API: interfacce diverse per UDP e TCP
- TCP: Stream Sockets
 - “apertura di una connessione”, collegare il client socket al server
- UDP: Datagram Sockets
 - non è richiesto un collegamento prima di inviare una lettera
 - piuttosto è necessario specificare l'indirizzo del destinatario per ogni lettera spedita
 - lettera = pacchetto
 - ogni pacchetto, chiamato `DatagramPacket`, è indipendente dagli altri e porta l'informazione per il suo instradamento

UDP IN JAVA



IL SERVIZIO DAYTIME IN UDP

- svilupperemo un daytime client
- il client si collega ad un server noto che offre sulla porta nota il servizio daytime: client in JAVA, server su porta nota
- specifica del servizio: RFC 867
- successivamente svilupperemo il DayTimeServer, un server scritto in JAVA per il servizio di DayTime

DAYTIME UDP CLIENT: “HOW TO DO”

1. aprire il socket: se si sceglie la porta 0 il sistema sceglie una porta libera “effimera”

```
DatagramSocket socket = new DatagramSocket(0);
```

2. impostare un timeout sul socket, opzionale, ma consigliato

```
setSoTimeout(15000)
```

3. costruire due pacchetti: uno per inviare la richiesta al server, uno per ricevere la risposta

```
InetAddress host = InetAddress.getByName(HOSTNAME);
```

```
DatagramPacket request = new DatagramPacket(new byte[1], 1, host , PORT);
```

```
byte [] data = new byte[1024];
```

```
DatagramPacket response = new DatagramPacket(data, data.length);
```

4. mandare la richiesta ed aspettare la risposta

```
socket.send(request);
```

```
socket.receive(response);
```

5. estrarre i byte dalla risposta e convertirli in String

```
String daytime = new String(response.getData(),0,response.getLength(),"Us-ASCII");
```

```
System.out.println(daytime);
```

DAYTIME CLIENT

```
import java.io.*;
import java.net.*;

public class DayTimeUDPClient {
    // RFC-867
    private final static int PORT = 13;
    private static final String HOSTNAME = "test.rebex.net";
    public static void main(String[] args) {
        try (DatagramSocket socket = new DatagramSocket(0)) {
            socket.setSoTimeout(15000);
            InetAddress host = InetAddress.getByName(HOSTNAME);
            DatagramPacket request = new DatagramPacket(new byte[1], 1, host, PORT);
            DatagramPacket response = new DatagramPacket(new byte[1024], 1024);
            socket.send(request);
            socket.receive(response);
            String daytime = new String(response.getData(), 0, response.getLength(), "Us-ASCII");
            System.out.println(daytime);
        }
        catch (IOException ex) { ex.printStackTrace(); }}
```

try with resources

```
$Java DayTimeUDPClient
Wed, 23 Nov 2022 22:37:43 GMT
```


DAYTIME UDP SERVER: “HOW TO DO”

1. aprire un DatagramSocket su una porta “nota” (“well known port”)

```
DatagramSocket socket = new DatagramSocket(13))
```

- porta nota perchè i client devono inviare i packet a quella destinazione
- a differenza di TCP, stesso tipo di socket per il client e per il server

2. creare un pacchetto in cui ricevere la richiesta del client

```
DatagramPacket request = new DatagramPacket(new byte[1024], 1024);  
socket.receive(request);
```

3. creare un pacchetto di risposta

```
String daytime = new Date().toString();  
byte[] data = daytime.getBytes("US-ASCII");  
InetAddress host = request.getAddress();  
int port = request.getPort();  
DatagramPacket response = new DatagramPacket(data, data.length, host, port)
```

4. inviare la risposta usando lo stesso socket da cui si è ricevuto il pacchetto

```
socket.send(response);
```

DAYTIME UDP SERVER

```
import java.net.*; import java.util.Date; import java.io.*;

public class DatTimeUDPServer {
    private final static int PORT = 13;

    public static void main(String[] args) {
        try (DatagramSocket socket = new DatagramSocket(PORT)) {
            while (true) {
                try {
                    DatagramPacket request = new DatagramPacket(new byte[1024], 1024);
                    socket.receive(request);

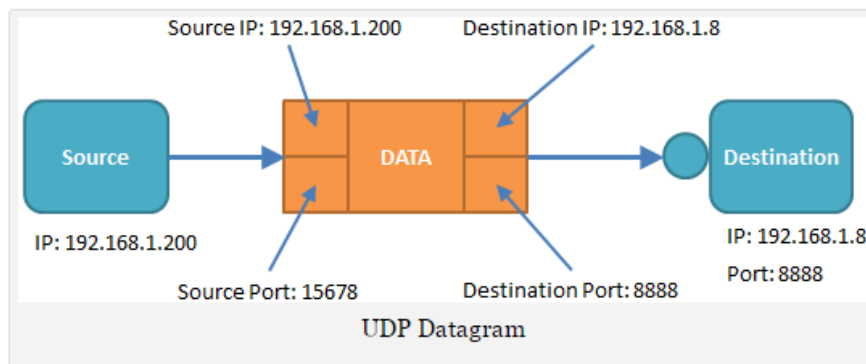
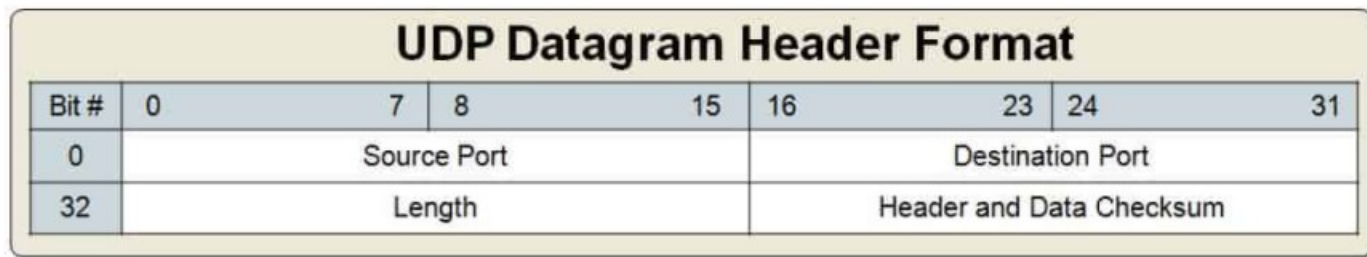
                    System.out.println("ricevuto un pacchetto da"+request.getAddress()+"
                                         "+request.getPort());

                    String daytime = new Date().toString();
                    byte[] data = daytime.getBytes("US-ASCII");
                    DatagramPacket response = new DatagramPacket(data, data.length,
                                                                    request.getAddress(), request.getPort());

                    socket.send(response);
                } catch (IOException | RuntimeException ex) {ex.printStackTrace();}
            }
        } catch (IOException ex) { ex.printStackTrace();}}
```

IL DATAGRAM UDP

- dimensione massima teorica di un pacchetto: 65597 bytes
 - IP header= 20 bytes, UDP header=8 bytes
 - molte piattaforme limitano la dimensione massima a 8192 bytes
- in JAVA un datagram UDP è rappresentato come una istanza della classe DatagramPacket



- il mittente deve inizializzare
 - il campo DATA
 - destination IP e destination port
- source IP inserito automaticamente
- source port può essere effimera

DATAGRAMPACKET: 6 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)
```

- in ogni caso un riferimento ad un vettore di byte buffer che contiene i dati da spedire oppure quelli ricevuti
- eventuali informazioni di addressing, se il DatagramPacket deve essere spedito

RICEVERE DATI: COSTRUZIONE DATAGRAM

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

- definisce la struttura utilizzata per memorizzare il pacchetto ricevuto.
- il buffer viene passato vuoto alla receive che lo riempie con il payload del pacchetto ricevuto
- se settato l'offset, la copia avviene a partire dalla posizione indicata
- il parametro length
 - indica il numero massimo di bytes che possono essere copiati nel buffer
 - deve essere minore di buffer.length, altrimenti viene sollevata eccezione
- la copia del payload termina quando
 - l'intero pacchetto è stato copiato
 - oppure quando length bytes sono stati copiati, se il payload è più grande
 - getLength restituisce il numero di bytes effettivamente copiati

INVIARE DATI: COSTRUZIONE DEL DATAGRAM

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

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

LA CLASSE DATAGRAM SOCKETS: COSTRUTTORI

- `DatagramSocket()` crea un socket e lo collega ad una qualsiasi porta libera disponibile sull'host locale

```
try {DatagramSocket client=new DatagramSocket();//send packets}
```

- utilizzato generalmente lato client, per spedire datagrammi
- `getLocalPort()` per reperire la porta allocata
- il server può inviare la risposta, prelevando l'indirizzo del mittente (IP+porta) dal pacchetto ricevuto
- `DatagramSocket(int p)` crea un socket e lo collega alla porta specificata, sull'host locale
 - il server crea un socket collegato ad una porta che rende nota ai clients.
 - la porta è allocata a quel servizio (porta non effimera)
 - solleva un'eccezione quando la porta è già utilizzata, oppure se non si hanno i diritti

LA CLASSE DATAGRAM SOCKETS: COSTRUTTORI

- DatagramSocket: crea un socket e lo collega ad una qualsiasi porta libera disponibile sull'host locale

```
try {  
    DatagramSocket client = new DatagramSocket();  
    //send packets }
```

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

DECIDERE LA DIMENSIONE DEL DATAGRAMPACKET

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

```
import java.net.*;

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

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

I METODI SET

```
void setData(byte[ ] buffer)
```

- setta il payload di “this” packet, prendendo i dati dal buffer

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

- setta il payload di “this” packet, prendendo dati da una parte del buffer
- utile quando si deve mandare una grande quantità di dati

```
int offset = 0;
```

```
DatagramPacket dp = new DatagramPacket(bigarray, offset, 512);
```

```
int bytesSent = 0;
```

```
while (bytesSent < bigarray.length) {
```

```
    socket.send(dp);
```

```
    bytesSent += dp.getLength( );
```

```
    int bytesToSend = bigarray.length - bytesSent;
```

```
    int size = (bytesToSend > 512) ? 512 : bytesToSend;
```

```
    dp.setData(bigarray, bytesSent, size);
```

```
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");}
```

```
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);
```

I METODI GET

`InetAddress getAddress()`

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

`int getPort()`

- restituisce il numero di porta sull'host remoto a cui il Datagram è stato inviato, oppure della macchina da cui è stato spedito

`byte[] getData()`

- restituisce un byte array contenente i dati del buffer associato al Datagram
- ignora offset e lunghezza

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

- restituiscono la lunghezza/offset del Datagram da inviare o da ricevere

`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

INVIARE E RICEVERE DATAGRAM

invio di pacchetti sock.*send* (dp)

- sock è il socket attraverso il quale voglio spedire il Datagram dp

ricezione di pacchetti sock.*receive*(dp)

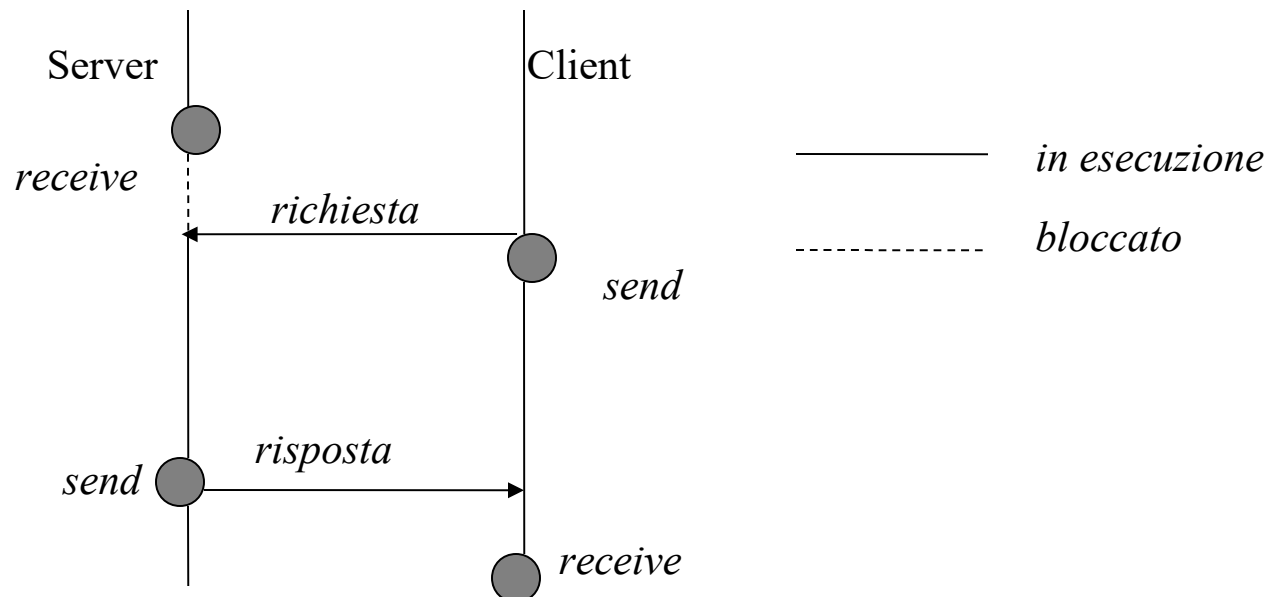
- sock è il socket attraverso il quale ricevo il Datagram dp
- riceve un Datagram dal socket
- riempie il buffer associato al socket con i dati ricevuti
- Il Datagram ricevuto contiene anche indirizzo IP e porta del mittente

COMUNICAZIONE UDP: CARATTERISTICHE

send non bloccante nel senso che il processo che esegue la send prosegue la sua esecuzione, senza attendere che il destinatario abbia ricevuto il pacchetto

receive bloccante il processo che esegue la receive si blocca fino al momento in cui viene ricevuto un pacchetto.

per evitare attese indefinite è possibile associare **al socket un timeout**. Quando il timeout scade, viene sollevata una **InterruptedIOException**



GESTIONE BUFFER RICEZIONE

```
import java.net.*;

public class Sender
{
    public static void main (String args[]) {
        try
        {
            DatagramSocket clientsocket = new DatagramSocket();
            byte[] buffer="1234567890abcdefghijklmnopqrstuvwxyz".getBytes("US-
                                                                    ASCII");

            InetAddress address = InetAddress.getByName("Localhost");

            for (int i = buffer.length; i >0; i--) {
                DatagramPacket mypacket = new DatagramPacket(buffer,i,address,
                                                                40000);

                clientSocket.send(mypacket);
                Thread.sleep(200); }

            System.exit(0);}

        catch (Exception e) {e.printStackTrace();}}}
```


DATI INVIATI

Dati inviati dal mittente:

Length 36 data 1234567890abcdefghijklmnopqrstuvwxyz

Length 35 data 1234567890abcdefghijklmnopqrstuvwxyz

Length 34 data 1234567890abcdefghijklmnopqrstuvwxyz

Length 33 data 1234567890abcdefghijklmnopqrstuvwxyz

Length 32 data 1234567890abcdefghijklmnopqrstuvwxyz

Length 31 data 1234567890abcdefghijklmnopqrstuvwxyz

.....

Length 5 data 12345

Length 4 data 1234

Length 3 data 123

Length 2 data 12

Length 1 data 1

GESTIONE BUFFER RICEZIONE

```
import java.net.*;

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

        while (true) {
            serverSock.receive(receivedPacket);
            String byteToString = new String(receivedPacket.getData(), "US-
                                            ASCII");

            int l=byteToString.length();
            System.out.println(l);
            System.out.println("Length " + receivedPacket.getLength() +
                               " data " + byteToString);}}}
}
```

DATI RICEVUTI

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

- per ricevere correttamente i dati
individuare i dati disponibili specificando offset e lunghezza

String byteToString =

new **String**(receivedPacket.getData(),0,receivedPacket.getLength(),"US-ASCII")

- sempre specificare lunghezza ed offset dei dati ricevuti, anche se si utilizzano stream (vedi slide successive)

RECEIVE CON TIMEOUT

- `SO_TIMEOUT` proprietà associata al socket, indica l'intervallo di tempo, in millisecondi, di attesa di ogni receive eseguita su quel socket
- nel caso in cui l'intervallo di tempo scada prima che venga ricevuto un pacchetto dal socket, viene sollevata una eccezione di tipo `InterruptedException`
- metodi per la gestione di time out

```
public synchronized void setSoTimeout(int timeout) throws  
SocketException
```

esempio: se ds è un datagram socket,

```
ds.setSoTimeout(30000)
```

associa un timeout di 30 secondi al socket ds.

COSTRUZIONE/LETTURA DI VETTORI DI BYTES

- i dati inviati mediante UDP devono essere rappresentati come **vettori di bytes**
- alcuni metodi per la conversione stringhe/vettori di bytes
 - `Byte[] getBytes()`
 - applicato ad un oggetto `String`
 - restituisce una sequenza di bytes che codificano i caratteri della stringa usando la codifica di default dell'host e li memorizza nel vettore
 - `String (byte[] bytes, int offset, int length)`
 - costruisce un nuovo oggetto di tipo `String` 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

DATI STRUTTURATI IN PACCHETTI UDP

```
public ByteArrayOutputStream ( )
```

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

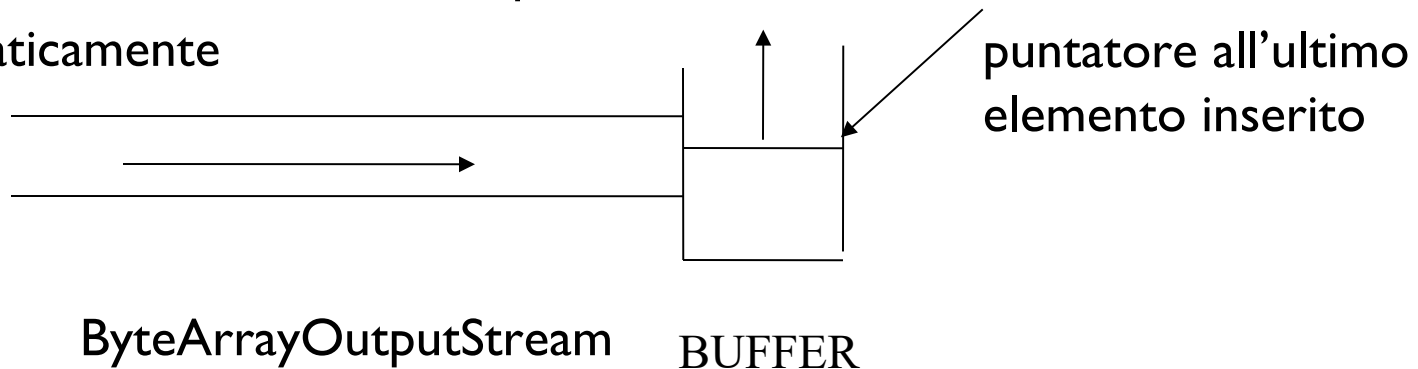
- gli oggetti istanze di questa classe rappresentano stream di bytes
- ogni dato scritto sullo stream viene riportato in un **buffer di memoria** a **dimensione variabile** (dimensione di default = 32 bytes).

```
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



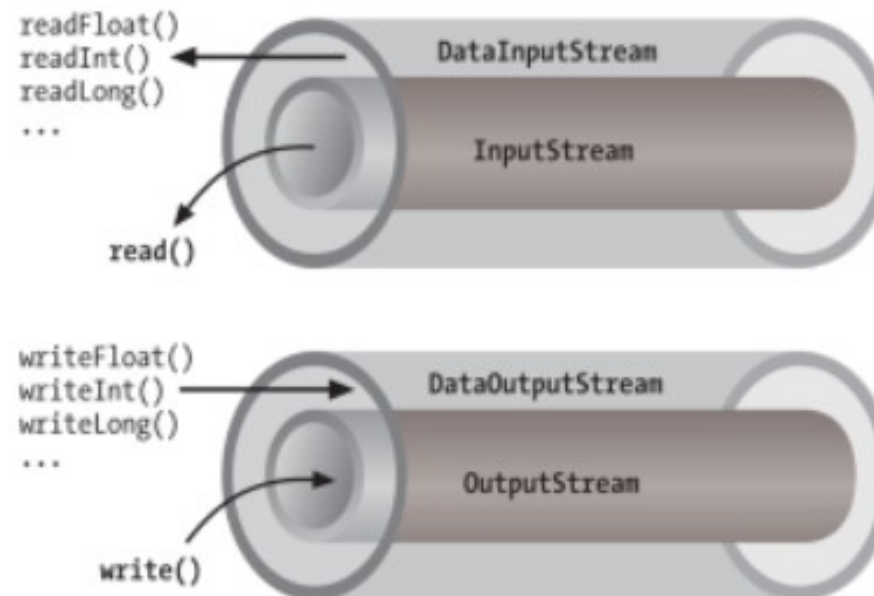
UDP E DATA STREAM

- è possibile collegare ad un `ByteArrayOutputStream` un altro filtro

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

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

- `DataOutput/InputStream` consente di scrivere dati primitivi sullo stream, la trasformazione in bytes è effettuata automaticamente



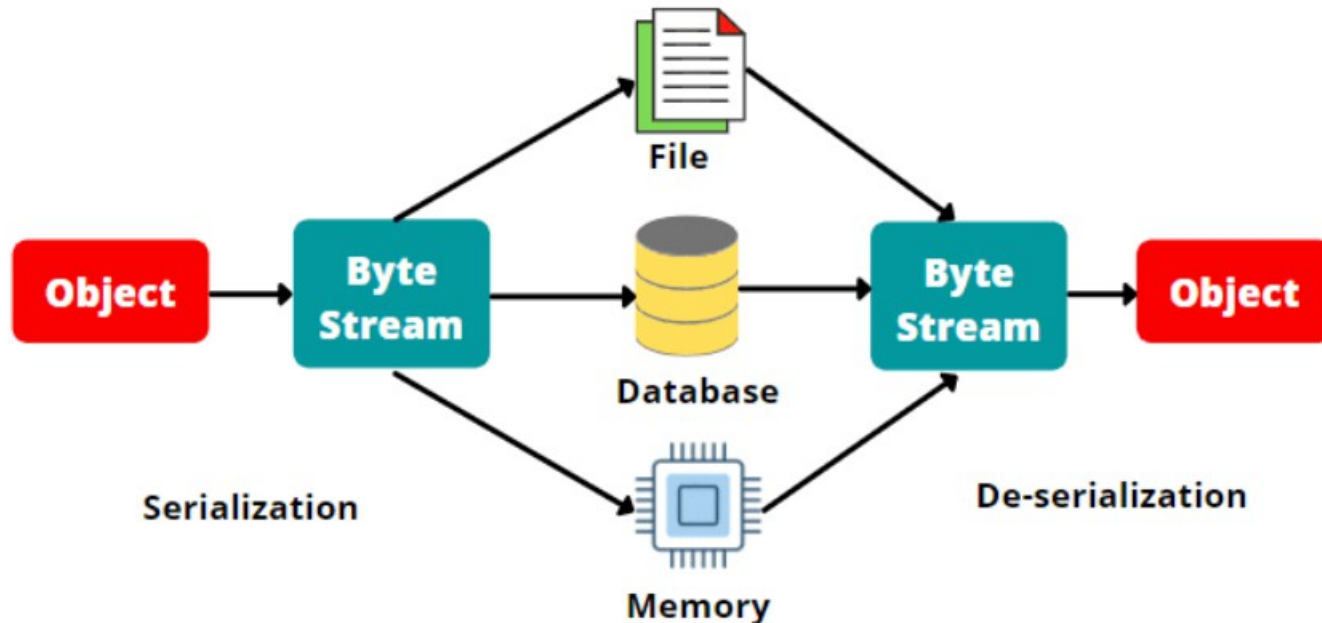
Data streams in java

UDP E SERIALIZAZIONE

- inviare oggetti in pacchetti?
- usare la serializzazione per generare uno stream di Byte
- collegare l'outputstream generato ad un `ByteArrayOutputStream`

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

```
ObjectOutputStream dos = new ObjectOutputStream(baos)
```



BYTE ARRAY OUTPUT STREAMS

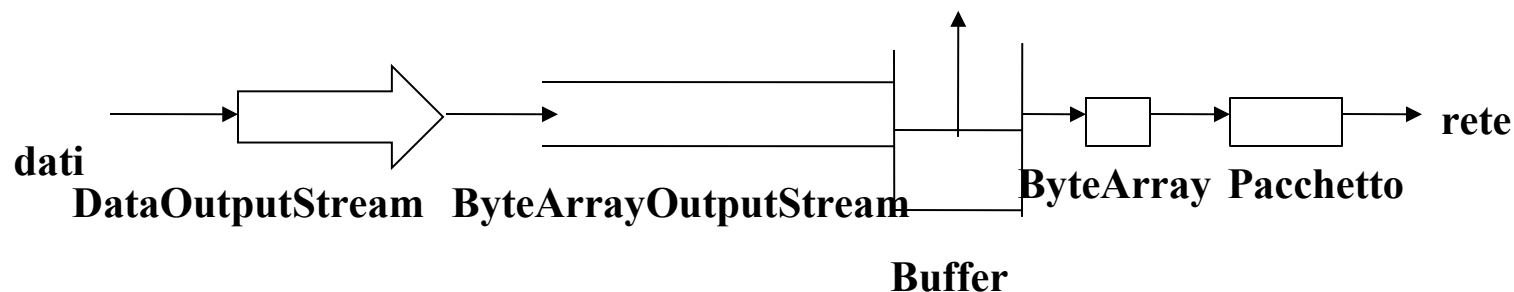
- ad un `ByteArrayOutputStream` può essere collegato un altro filtro

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

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

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

Flusso dei dati:

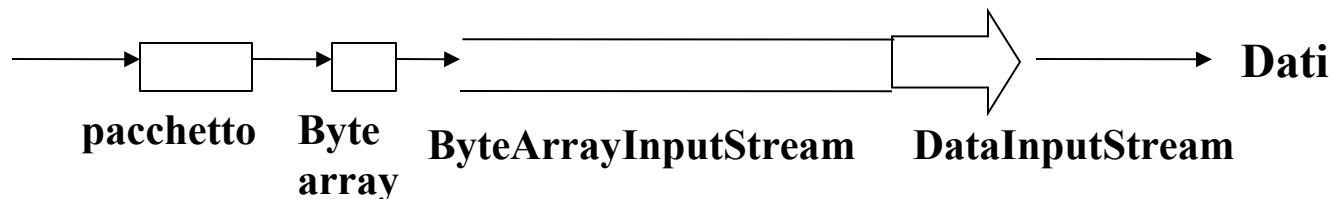


BYTE ARRAY INPUT STREAMS

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

- creano stream di byte a partire dai dati contenuti nel vettore di byte buf.
- il secondo costruttore copia length bytes iniziando alla posizione offset.
- è possibile concatenare un **DataInputStream**

Flusso dei dati:



LA CLASSE BYTEARRAYOUTPUTSTREAM

metodi per la gestione dello stream:

- **public int size()** restituisce count, cioè il numero di bytes memorizzati nello stream (non la lunghezza del vettore buf!)
- **public synchronized void reset()** svuota il buffer, assegnando 0 a count. tutti i dati precedentemente scritti vengono eliminati.

```
baos.reset ( )
```

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

BYTE ARRAY INPUT/OUTPUT STREAMS

Ipotesi semplificativa: non consideriamo perdita/riordinamento di pacchetti:

```
import java.io.*;
import java.net.*;
public class multidatastreamsender{
    public static void main(String args[ ]) throws Exception
    {
        // fase di inizializzazione
        InetAddress ia=InetAddress.getByName("localhost");
        int port=13350;
        DatagramSocket ds= new DatagramSocket();
        ByteArrayOutputStream bout= new ByteArrayOutputStream();
        DataOutputStream dout = new DataOutputStream (bout);
        byte [ ] data = new byte [20];
        DatagramPacket dp= new DatagramPacket(data,data.length, ia , port);
```

BYTE ARRAY INPUT/OUTPUT STREAMS

```
for (int i=0; i< 10; i++)  
    {dout.writeInt(i);  
    data = bout.toByteArray();  
    dp.setData(data,0,data.length);  
    dp.setLength(data.length);  
    ds.send(dp);  
    bout.reset( );  
    dout.writeUTF("***");  
    data = bout.toByteArray( );  
    dp.setData (data,0,data.length);  
    dp.setLength (data.length);  
    ds.send (dp);  
    bout.reset( ); } } }
```

BYTE ARRAY INPUT/OUTPUT STREAMS

Ipotesi semplificativa: non consideriamo perdita/riordinamento di pacchetti

```
import java.io.*;
import java.net.*;

public class multidatastreamreceiver
{
    public static void main(String args[ ]) throws Exception
    {
        // fase di inizializzazione
        FileOutputStream fw = new FileOutputStream("text.txt");
        DataOutputStream dr = new DataOutputStream(fw);
        int port = 13350;
        DatagramSocket ds = new DatagramSocket (port);
        byte [ ] buffer = new byte [200];
        DatagramPacket dp= new DatagramPacket
                                (buffer, buffer.length);
    }
}
```

BYTE ARRAY INPUT/OUTPUT STREAMS

```
for (int i=0; i<10; i++)
{ds.receive(dp);
    ByteArrayInputStream bin= new ByteArrayInputStream
                                (dp.getData(),0,dp.getLength());
    DataInputStream ddis= new DataInputStream(bin);
    int x = ddis.readInt();
    dr.writeInt(x);
    System.out.println(x);
    ds.receive(dp);
    bin= new ByteArrayInputStream(dp.getData(),0,dp.getLength());
    ddis= new DataInputStream(bin);
    String y=ddis.readUTF( );
    System.out.println(y); }}}
```


BYTE ARRAY INPUT/OUTPUT STREAMS

- nel programma precedente, la corrispondenza tra la **scrittura** nel mittente e la **lettura** nel destinatario potrebbe non essere più corretta
- esempio:
 - il mittente alterna la spedizione di pacchetti contenenti valori interi con pacchetti contenenti stringhe
 - il destinatario alterna la lettura di interi e di stringhe
 - ma se un pacchetto viene perso: il destinatario scritture/letture possono non corrispondere
- realizzazione di UDP affidabile: utilizzo di ack per confermare la ricezione + identificatori unici

CONCLUSIONI: STREAM ARE EVERYWHERE!

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

invio di dati scrittura sullo stream

ricezione di dati lettura dallo stream

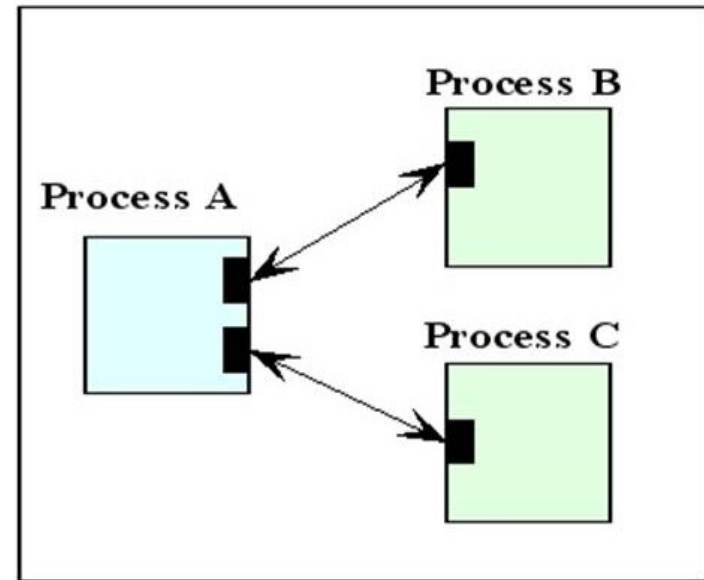
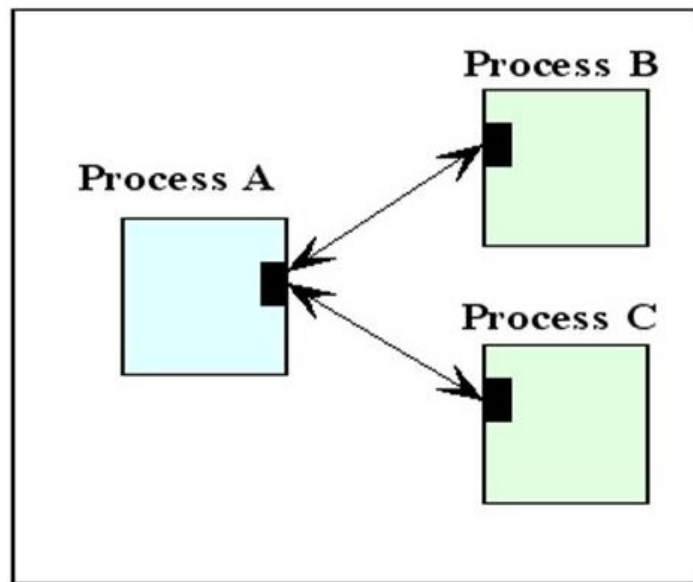
- trasmissione connectionless: stream utilizzati per la generazione dei pacchetti:

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

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

CONCLUSIONI: SOCKET UDP

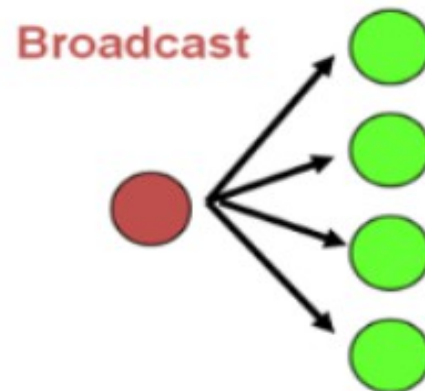
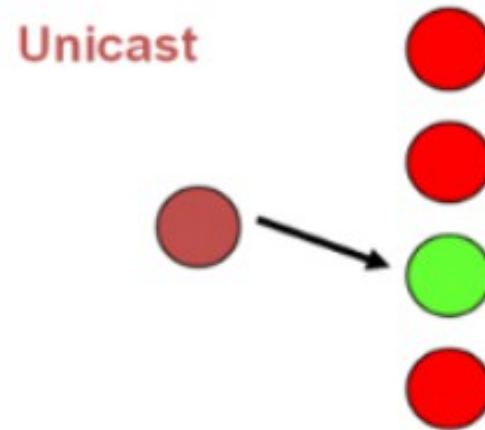
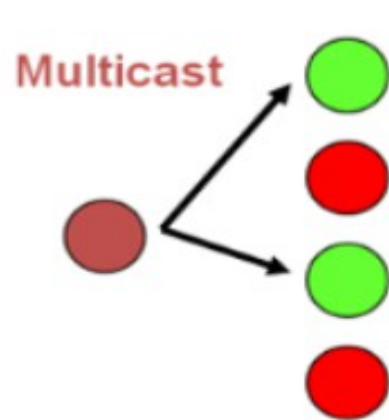
- possibile usare lo stesso socket per spedire pacchetti verso destinatari diversi
- processi (applicazioni) diverse possono spedire pacchetti sullo stesso socket in questo caso l'ordine di arrivo dei messaggi è non deterministico, in accordo con il protocollo UDP
- ...ma è possibile anche utilizzare socket diversi per comunicazioni diverse



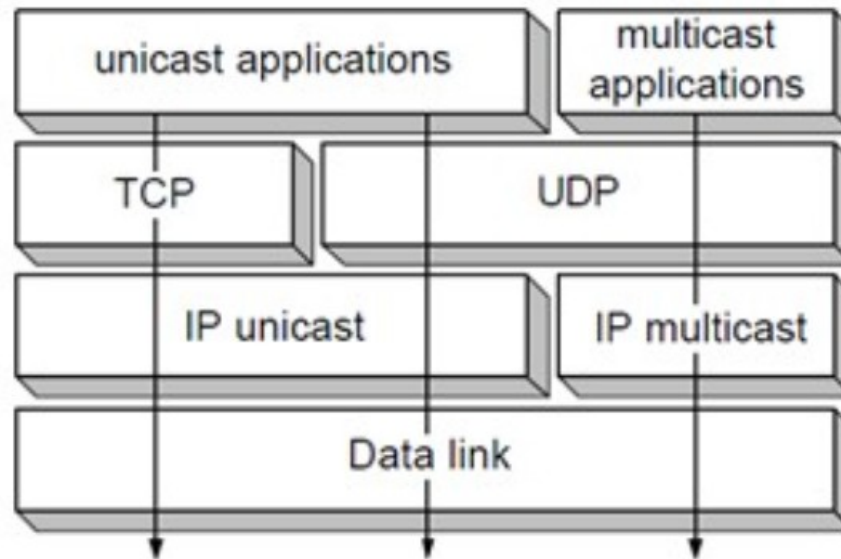
MULTICASTING

- inviare dati da un host ad un insieme di altri nodi
 - non tutti gli host della rete: solo quelli che hanno espresso interesse ad unirsi ad un gruppo di multicast
- esempi:
 - diverse applicazioni usano IP multicasting per notificare/scoprire dinamicamente i servizi in una rete, senza usare DNS o terze parti
- la maggior parte del lavoro viene svolto dai router ed è trasparente al programmatore
 - i router assicurano che il pacchetto spedito dal mittente sia consegnato a tutti gli host interessati
 - usa Time-To-Live IP: massimo numero di router che il datagramma può attraversare
 - il problema maggiore è che non tutti i router supportano il multicast

UNICASTING/MULTICASTING/BROADCASTING

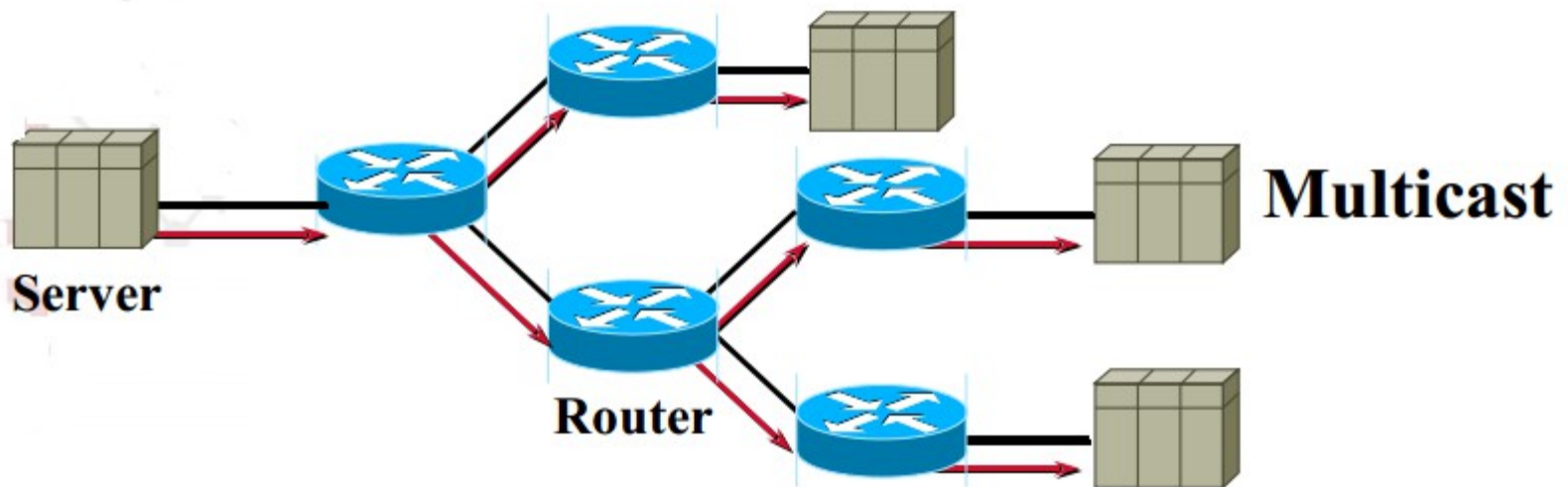
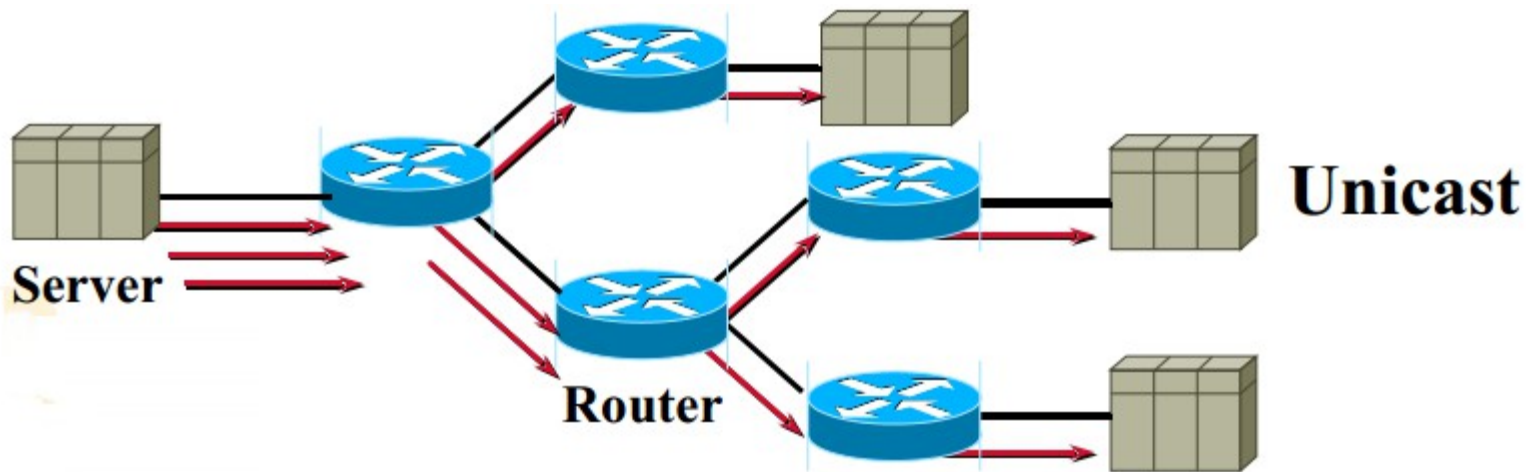


MULTICAST E PROTOCOL STACK

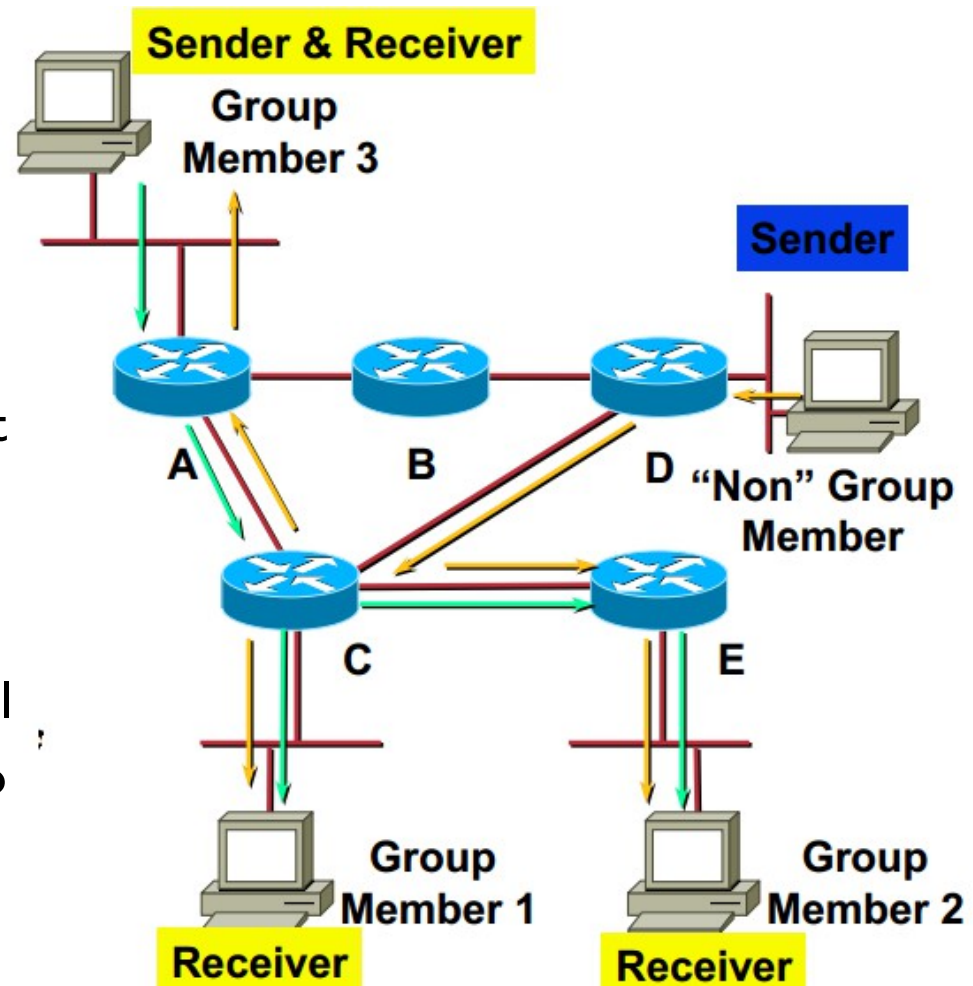


- IP multicasting utilizza UDP
- non esiste multicast TCP!

UNICAST VERSO MULTICAST



- IP multicast basato sul **concetto di gruppo**
 - insieme di processi in esecuzione su host diversi
- tutti i membri di un gruppo di multicast ricevono un messaggio spedito su quel gruppo
- mentre non occorre essere membri del gruppo per inviare i messaggi su di esso
- gestiti a livello IP dal protocollo IGMP



deve contenere almeno primitive per:

- **unirsi** ad un gruppo di multicast
- **lasciare** un gruppo
- **spedire** messaggi ad un gruppo
 - il messaggio viene recapitato a tutti i processi che fanno parte del gruppo in quel momento
- **ricevere** messaggi indirizzati ad un gruppo

Il supporto deve fornire

- uno **schema di indirizzamento** per identificare univocamente un gruppo.
- un meccanismo che registri la corrispondenza tra un gruppo ed i suoi partecipanti (IGMP)
- un meccanismo che ottimizzi l'uso della rete nel caso di invio di pacchetti ad un gruppo di multicast

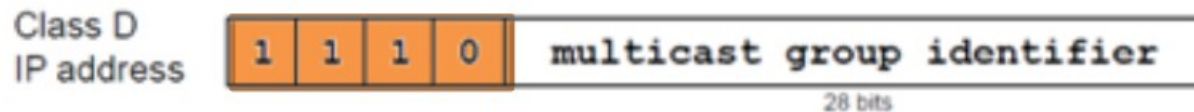
SCHEMA DI INDIRIZZAMENTO MULTICAST

- basato sull'idea di riservare un insieme di indirizzi IP per il multicast
- IPV4: indirizzo di un gruppo è un indirizzo in classe D
 - [224.0.0.0 – 239.255.255.255]

Class A :	0.0.0.0 to 127.255.255.255
Class B :	128.0.0.0 to 191.255.255.255
Class C :	192.0.0.0 to 223.255.255.255
Class D :	224.0.0.0 to 239.255.255.255
Class E :	240.0.0.0 to 255.255.255.255

→ Multicast range ←

*The IP Classes listed above are not all usable by hosts!
Here we are simply looking at the range each Class covers*



i primi 4 bit del primo ottetto = 1110

i restanti bit identificano il particolare gruppo

IPV6: tutti gli indirizzi di multicast iniziano con FF o **1111 1111** in binario

MULTICAST: CARATTERISTICHE

- utilizza il paradigma **connectionless** (UDP):
 - sarebbero richieste $n \times (n-1)$ **connessioni** per un gruppo di n applicazioni, se si usa la comunicazione **connection oriented**
- comunicazione **connectionless** adatta per il tipo di applicazioni verso cui è orientato il multicast
 - trasmissione di dati video/audio: invio dei frames di una animazione.
 - è più accettabile la **perdita occasionale** di un frame piuttosto che un **ritardo costante** tra la spedizione di due frames successivi
- si perde la affidabilità della trasmissione, ma esistono librerie JAVA non standard che forniscono multicast affidabile
- garantiscono che il messaggio venga recapitato una sola volta a tutti i processi del gruppo
- possono garantire altre proprietà relative all'ordinamento con cui i messaggi spediti al gruppo di multicast vengono recapitati ai singoli partecipanti

SOCKET MULTICAST

Java.net.MulticastSocket

- estende `DatagramSocket`
- socket su cui ricevere i messaggi da un gruppo di multicast
- effettua overriding dei metodi esistenti in `DatagramSocket` e fornisce nuovi metodi per l'implementazione di funzionalità tipiche del multicast

```
import java.net.*;
import java.io.*;
public class multicast
{
    public static void main (String [ ] args)
    {
        try
        {
            MulticastSocket ms = new MulticastSocket( );
        }
        catch (IOException ex) {System.out.println("errore"); }
    }
}
```

UNIRSI AD UN GRUPPO MULTICAST

```
import java.net.*;
import java.io.*;
public class multicast
{
    public static void main (String [ ] args)
    {
        try {MulticastSocket ms = new MulticastSocket(4000);
            InetAddress
                ia=InetAddress.getByAddress("226.226.226.226");
            ms.joinGroup (ia);
        }
        catch (IOException ex) {System.out.println("errore"); }}
```

- joinGroup necessaria nel caso si vogliano ricevere messaggi dal gruppo di multicast
- lega il multicast socket ad un gruppo di multicast: tutti i messaggi ricevuti tramite quel socket provengono da quel gruppo
- IOException sollevata se l'indirizzo di multicast è errato

RICEVERE PACCHETTI DA MULTICAST

```
import java.io.*; import java.net.*;

public class provemulticast {

public static void main (String args[]) throws Exception
{ byte[] buf = new byte[10];
  InetAddress      ia = InetAddress.getByName("228.5.6.7");
  DatagramPacket    dp = new DatagramPacket (buf,buf.length);
  MulticastSocket ms = new MulticastSocket (4000);
  ms.joinGroup(ia);
  ms.receive(dp);    } }
```

- se attivo due istanze di provemulticast sullo stesso host (la reuse socket settata true) non viene sollevata una BindException
- l'eccezione verrebbe invece sollevata se si utilizzasse un DatagramSocket
- servizi diversi in ascolto sulla stessa porta di multicast
- non esiste una corrispondenza biunivoca porta-servizio

JAVA API PER MULTICAST

- ogni socket multicast ha una proprietà, la `reuse socket`, che se settata a true, dà la possibilità di associare più socket alla stessa porta
- nelle ultime versioni è possibile impostarne il valore

```
try    {sock.setReuseAddress(true);}
catch (SocketException se) {...}
```
- nelle prime versioni di JAVA la proprietà era settata per default a true

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.
 - dati inviati dalla stessa send non possono essere ricevuti in receive diverse

ASSIGNMENT: JAVA PINGER

- PING è una utility per la valutazione delle performance della rete utilizzata per verificare la raggiungibilità di un host su una rete IP e per misurare il round trip time (RTT) per i messaggi spediti da un host mittente verso un host destinazione.
- lo scopo di questo assignment è quello di implementare un server PING ed un corrispondente client PING che consenta al client di misurare il suo RTT verso il server.
- la funzionalità fornita da questi programmi deve essere simile a quella della utility PING disponibile in tutti i moderni sistemi operativi. La differenza fondamentale è che si utilizza UDP per la comunicazione tra client e server, invece del protocollo ICMP (Internet Control Message Protocol).
- inoltre, poichè l'esecuzione dei programmi avverrà su un solo host o sulla rete locale ed in entrambe i casi sia la latenza che la perdita di pacchetti risultano trascurabili, il server deve introdurre un ritardo artificiale ed ignorare alcune richieste per simulare la perdita di pacchetti

PING CLIENT

- accetta due argomenti da linea di comando: nome e porta del server. Se uno o più argomenti risultano scorretti, il client termina, dopo aver stampato un messaggio di errore del tipo ERR -arg x, dove x è il numero dell'argomento.
- utilizza una comunicazione UDP per comunicare con il server ed invia 10 messaggi al server, con il seguente formato:

PING seqno timestamp

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

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

PING CLIENT

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

---- PING Statistics ----

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

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

PING SERVER

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

PING SERVER

```
java PingServer 10002 123
```

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

PING CLIENT

```
java PingClient localhost 10002
```

```
PING 0 1360792326564 RTT: 299 ms
```

```
PING 1 1360792326863 RTT: 183 ms
```

```
PING 2 1360792327046 RTT: 263 ms
```

```
PING 3 1360792327309 RTT: 22 ms
```

```
PING 4 1360792327331 RTT: 174 ms
```

```
PING 5 1360792327505 RTT: 45 ms
```

```
PING 6 1360792327550 RTT: 20 ms
```

```
PING 7 1360792327570 RTT: *
```

```
PING 8 1360792328571 RTT: *
```

```
PING 9 1360792329573 RTT: 263 ms
```

```
---- PING Statistics ----
```

```
10 packets transmitted, 8 packets received, 20% packet loss
```

```
round-trip (ms) min/avg/max = 20/158.62/299
```

JAVA PINGER

Invocazione corretta client/server:

```
java PingClient
```

```
Usage: java PingClient hostname port
```

```
java PingServer
```

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

Invocazione non corretta client/server:

```
java PingClient atria three
```

```
ERR - arg 2
```

```
java PingServer abc
```

```
ERR - arg 1
```