Programmazione di reti
Corso B

19 Ottobre 2016
Lezione 6

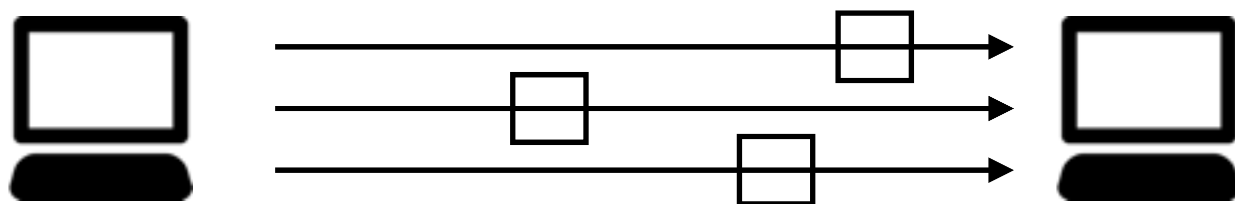
Contenuti

- Protocollo UDP
- *Socket* UDP
- UDP *multicasting*

Protocollo UDP

- Molto più veloce di TCP

- Molto meno affidabile
- Non si sa se pacchetti arrivano ne in quale ordine arrivano
- I pacchetti sono indipendenti
- Non esiste una connessione né uno stream continuo di dati come in TCP
- Corrispondenza tra `send` e `receive`: i dati inviati in un solo `send()` saranno ricevuti in un solo `receive()`. Questo non è valido per TCP.



Protocollo UDP

- non adatto per applicazioni quali ftp, http
- adatto per audio/video streaming, DNS, ping
- si possono fare correzioni/verifiche a livello applicazione
  - se non è arrivata una risposta, forse la richiesta si è persa
  - pacchetti possono essere ordinati - quelli mancanti inviati di nuovo

UDP in Java

- Due classi
  - `DatagramPacket`
    - usato per riempire e leggere pacchetti (datagram)
    - il pacchetto contiene la destinazione
  - `DatagramSocket`
    - invia e riceve `DatagramPacket`
    - conosce solo il *port* locale su quale inviare/ricevere
    - un socket può inviare pacchetti a più destinazioni (non c'è una connessione tra due applicazioni)
    - non esiste uno stream tra applicazioni

DatagramPacket

- Un pacchetto contiene *header* e dati.
- In teoria un pacchetto può contenere fino a 65507 *byte* di dati
- In pratica molti sistemi accettano solo fino a 8kb di dati (il resto vengono scartati)
- Per massima certezza è meglio usare fino a 512 *byte* di dati

DatagramPacket

- Il *header* include
  - *header* UDP:
    - *port* per mittente e destinazione
    - dimensione con UDP *header* incluso
    - *checksum* opzionale (se *checksum* non torna il pacchetto viene scartato)
  - *header* IP:
    - IP per mittente e destinazione
    - altri campi (versione, TTL, etc)

DatagramPacket

- La costruzione degli oggetti `DatagramPacket` dipende dalla modalità di uso: per inviare o ricevere dati

- Per ricevere:

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

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

DatagramPacket

- Per inviare:

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

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

DatagramPacket(byte[] data, int length,
SocketAddress destination)

DatagramPacket(byte[] data, int offset, int length,
SocketAddress destination)
```

- i dati non vengono copiati in un buffer locale, quindi cambiando i contenuti dell'array `data` dopo aver creato il datagram si cambia anche il dato da inviare

DatagramPacket

- Per lavorare con `DatagramPacket`, bisogna:
  - trasformare i dati in `byte[]`
  - trasformare da `byte[]` in altro tipo strutturato di dati
  - usando metodi di alcune classi:
    - `String: byte[] getBytes(String encoding)`, costruttore `String(byte[] bytes)`
  - usando `ByteArrayOutputStream` e `ByteArrayInputStream`

```java
public class ByteArrayExample {

   public static void main(String[] args) {

      Student[] students={new Student("Robert","Brown","Dawson",12),
            new Student("Michael","Reds","Pearse",40),
            new Student("Joanna","Moore","Collins",62),
            new Student("Ann","Brown","Buffallo",132)};

      ArrayList<Student> student_list= new ArrayList<Student>();
      Vector<Student> student_vector= new Vector<Student>();
      LinkedList<Student> student_linked_list= new LinkedList<Student>();

      for (Student s : students){
         student_list.add(s);
         student_vector.add(s);
         student_linked_list.add(s);
      }

      ArrayList<Object> all_lists=new ArrayList<>();
      all_lists.add(student_list);
      all_lists.add(student_vector);
      all_lists.add(student_linked_list);
```

```
    try {
        for (Object list : all_lists){

            ByteArrayOutputStream byte_stream= new ByteArrayOutputStream();

            ObjectOutputStream out = new ObjectOutputStream(byte_stream);

            out.writeObject(list);

            byte[] bytes= byte_stream.toByteArray();

            System.out.format("the %s uses %d bytes to store %d students %n",
                    list.getClass(),bytes.length,students.length);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
  }
}
```

```
the class java.util.ArrayList uses 292 bytes to store 4 students
the class java.util.Vector uses 397 bytes to store 4 students
the class java.util.LinkedList uses 282 bytes to store 4 students
```

```java
ByteArrayInputStream in_byte_stream= new ByteArrayInputStream(bytes);

ObjectInputStream in = new ObjectInputStream(in_byte_stream);

Collection<Student> read_list=(Collection<Student>) in.readObject();

System.out.format("read back from byte[] %d students: %n",
      read_list.size());

for (Student s : read_list){
   System.out.println(s);
}
```

```
read back from byte[] 4 students:
Robert Brown living at 12 Dawson street.
Michael Reds living at 40 Pearse street.
Joanna Moore living at 62 Collins street.
Ann Brown living at 132 Buffallo street.
```

DatagramPacket

- Leggere il *header*

`InetAddress getAddress()`
contiene l'indirizzo della parte remota della comunicazione: destinatario se pacchetto e da inviare, mittente se pacchetto e stato ricevuto

`int getPort()`
numero di *port* remoto

`SocketAddress getSocketAddress()`
indirizzo e *port* remoti

`int getLength()`
numero di *byte* di dati nel pacchetto

`int getOffset()`
dove cominciano i dati nel pacchetto

DatagramPacket

- Leggere i dati

`byte[] getData()`

L'*array* di *byte* ottenuto può essere trasformato in altro tipo di dati usando `ByteArrayInputStream` come da esempio precedente.

DatagramSocket

- Oggetto che rappresenta un *socket* UDP
- Usato da cliente e *server*
- Costruttori:

`public DatagramSocket() throws SocketException`
crea un socket associato ad un *port* aleatorio - di solito per clienti

`public DatagramSocket(int port) throws SocketException`
crea un socket associato ad un *port* conosciuto - di solito usato da *server*

i port UDP e TCP sono diversi - un `Socket` ed un `DatagramSocket` possono usare lo stesso numero di *port* allo stesso tempo

`public DatagramSocket(int port, InetAddress interface) throws SocketException`
`public DatagramSocket(SocketAddress interface) throws SocketException`
sceglie un'interfaccia di rete su cui aprire il *port*

`DatagramSocket`

- Usato per inviare e ricevere pacchetti
- Lo stesso `DatagramSocket` può inviare e ricevere più pacchetti da o per applicazioni remoti multipli.

`void send(DatagramPacket dp) throws IOException`
Invia un pacchetto alla destinazione inclusa nel `DatagramPacket`

`void receive(DatagramPacket dp) throws IOException`
Si blocca fin che un pacchetto arriva. Il byte array del `DatagramPacket` viene riempito. Se i dati sono di più, vengono scartati, se i dati sono di meno, la dimensione viene aggiustata.

DatagramSocket

```
public void receive(DatagramPacket dp) throws IOException
```

Lo stesso oggetto `DatagramPacket` può essere usato per ricevere più pacchetti

Ogni volta che il pacchetto arriva, la dimensione dei dati viene fissata alla dimensione del pacchetto in arrivo.

Prima di ricevere un nuovo pacchetto, dobbiamo resettare la dimensione, altrimenti i dati possono essere troncati.

```
while (condizione) {
   server.receive(packet);
   //usa i dati
   packet.setLength(MAX_LENGTH);
}
```

DatagramSocket

`public void close()`

Chiude il socket. `DatagramSocket` implementa `AutoCloseable`

```
public int getLocalPort()
public InetAddress getLocalAddress()
public SocketAddress getLocalSocketAddress()
```

Metodi per trovare il *port* e indirizzo locali su cui il *socket* lavora.

`DatagramSocket`

- Anche se può essere utile inviare o ricevere da più host alla volta, di solito la comunicazione occorre tra *peer* conosciuti.
- Si può controllare il *peer* con cui un `DatagramSocket` comunica:
  - sono inviati solo pacchetti a questo indirizzo, altrimenti viene lanciata un'eccezione `InvalidArgumentException`
  - i pacchetti con provenienza diversa vengono scartati

DatagramSocket

```
void connect(InetAddress host, int port)
```

non si stabilisce una connessione nel senso TCP, si aggiunge solo un controllo dell'origine e della destinazione dei pacchetti

```
void disconnect()
```

rimuove il controllo dell'origine e della destinazione dei pacchetti

```
int getPort()
InetAddress getInetAddress()
InetAddress getRemoteSocketAddress()
```

restituiscono l'indirizzo con cui il socket è connesso (-1, null, null se socket non è mai stato connesso)

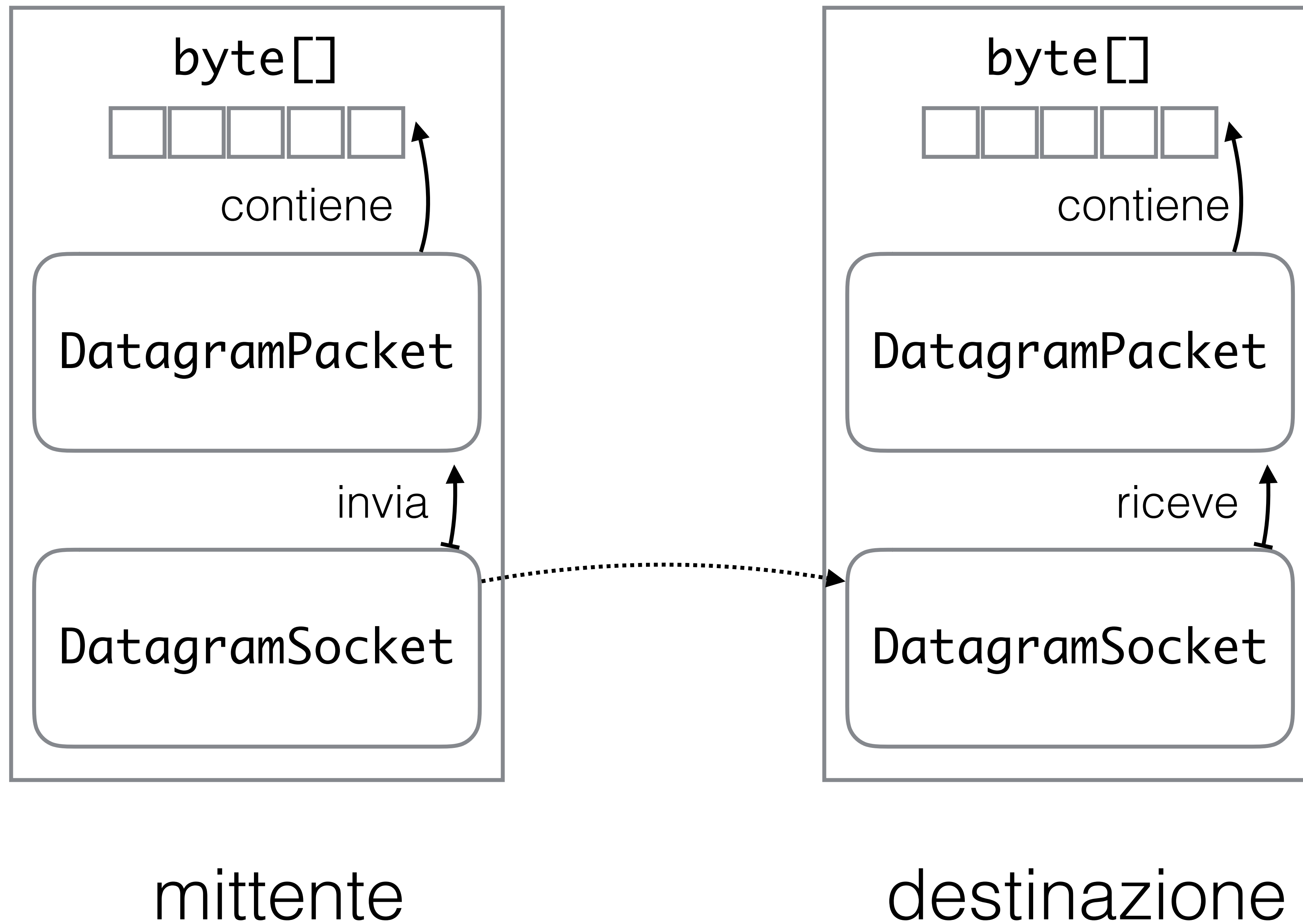
Opzioni

`void setSoTimeout(int timeout) throws SocketException`
Imposta un timeout per metodo `receive()`

`void setReceiveBufferSize(int size) throws SocketException`
`void setSendBufferSize(int size) throws SocketException`
Suggerimenti per dimensione dei buffer

`void setReuseAddress(boolean on) throws SocketException`
Lo stesso port può essere usato da `DatagramSocket` multipli. Quando un pacchetto arriva sul *port*, questo viene inviato a tutti i `DatagramSocket` che aspettano su quel *port*.

`void setBroadcast(boolean on) throws SocketException`
Permette al `DatagramSocket` di inviare/ricevere messaggi di *broadcast* (inviati ad indirizzi che finiscono in .255)


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

Cliente UDP

- Apre un `DatagramSocket`

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

- Impostazioni

```
socket.setTimeout(10000);
```

Timeout molto importante, a causa dell’inaffidabilità della trasmissione - non si sa se il server è online almeno che non risponde

Cliente UDP

- Invia una richiesta con `DatagramPacket`
    - Crea `DatagramPacket` - Contiene *host* e *port* della destinazione, e un `byte[]` con i dati

      ```
      InetAddress host = InetAddress.getByName(“time.nist.gov”);
      byte[] data= new byte[1];
      DatagramPacket request = new DatagramPacket(data, 1, host, 13);
      ```

    - Invia richiesta

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

Cliente UDP

- Riceve una risposta con `DatagramPacket`
    - Crea un `DatagramPacket` per risposta

```
byte[] data = new byte[1024];
DatagramPacket response = new DatagramPacket(data,
data.length);
```

- Aspetta risposta

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

Server UDP

- Si usa la stessa classe `DatagramSocket`, la differenza è che il server legge prima di scrivere

- Aspetta una richiesta

```
DatagramPacket request = new DatagramPacket(new byte[1024],
0, 1024);
socket.receive(request);
```

- Invia risposta: deve estrarre la destinazione dal pacchetto appena ricevuto

```
byte[] data;
//… add data to the byte array
InetAddress host = request.getAddress();
int port = request.getPort();
DatagramPacket response = new DatagramPacket(data,
data.length, host, port);
socket.send(response);
```

Server Echo
public class EchoServer {
static final int MAX_LENGTH=512;
static final int PORT=2000;
public static void main(String[] args) {
try (DatagramSocket server= new DatagramSocket(PORT);){
DatagramPacket request= new DatagramPacket(new byte[MAX_LENGTH],MAX_LENGTH);
while(true){
try {
System.out.println("Waiting for clients…");
server.receive(request);
Costruttore simile anche per ByteArrayInputStream
System.out.println("Received: “+
new String(request.getData(),0,request.getLength(),”UTF-8"));
DatagramPacket response= new DatagramPacket(
request.getData(), request.getLength(),request.getSocketAddress());
server.send(response);
} catch (IOException e) {
System.err.println("Error in communication: "+e.getMessage());
}
}
} catch (SocketException e) {
System.err.println("Error creating socket: "+e.getMessage());
}
}
}

Cliente Echo

```
public class EchoClient {
   public static void main(String[] args) {
      String message=null;
      try(DatagramSocket client= new DatagramSocket();
            BufferedReader reader= new BufferedReader(
                  new InputStreamReader(System.in))){
         System.out.println("Enter a message to send or 'exit' to finish.");
         DatagramPacket response= new DatagramPacket(
               new byte[EchoServer.MAX_LENGTH], EchoServer.MAX_LENGTH);
         SocketAddress server= new InetSocketAddress(InetAddress.getLocalHost(),
                     EchoServer.PORT);
         while(!(message=reader.readLine()).trim().toLowerCase().equals("exit")){
            byte[] bMessage=message.getBytes("UTF-8");
            DatagramPacket request= new DatagramPacket(bMessage,
                     bMessage.length,server);
            try {
               client.send(request);
               client.receive(response);
               System.out.println("Received response: “+
                     new String(response.getData(),0,response.getLength(),"UTF-8"));
            } catch (IOException e) {
               System.err.println("Error in communication: "+e.getMessage());
            }
            System.out.println("Enter a message to send or 'exit' to finish.");
         }
         System.out.println("Bye!");
```

```java
        } catch (SocketException e) {
            System.err.println("Error creating socket: "+e.getMessage());
        } catch (UnsupportedEncodingException e) {
            System.err.println("Wrong encoding: "+e.getMessage());
        } catch (IOException e) {
            System.err.println("Error reading from stdin: "+e.getMessage());
        }
    }
}
```

```
Waiting for clients...
Received: hello
Waiting for clients...
Received: how are you?
Waiting for clients...
Received: fine
Waiting for clients...
```

```
Enter a message to send or 'exit' to finish.
hello
Received response: hello
Enter a message to send or 'exit' to finish.
how are you?
Received response: how are you?
Enter a message to send or 'exit' to finish.
fine
Received response: fine
Enter a message to send or 'exit' to finish.
exit
Bye!
```

Esempio servizio “Echo”

- Il programma funziona in un sistema perfetto
- problemi:
    - Cosa succede se un pacchetto si perde (UDP non è affidabile)?
        - Il server - non è influenzato
        - Il cliente - si blocca
        - Soluzione: cliente con 2 thread, uno scrive, uno legge
    - Dimensione del pacchetto per messaggi in arrivo
        - Il Javadoc del metodo `receive()` della classe `DatagramSocket`: “The length field of the datagram packet object contains the length of the received message. If the message is longer than the packet's length, the message is truncated.”
        - Nel mio caso questo non succede (sembra che il funzionamento è cambiato dopo Java 1.4, però la documentazione non ci aiuta a sapere se vale per tutte le piattaforme)
        - Meglio ressettare la dimensione, come abbiamo detto prima.
    - Server non è multithreaded

Server “Echo” multithreaded

```
public class EchoServer {
   static final int MAX_LENGTH=512;
   static final int PORT=2000;
   static final int THREADS=5;
   public static void main(String[] args) {

      try (DatagramSocket server= new DatagramSocket(PORT);){
         ExecutorService es = Executors.newFixedThreadPool(THREADS);
         ArrayList<EchoServerTask> tasks= new ArrayList<>();
         for (int i=0;i<THREADS;i++){
            tasks.add(new EchoServerTask(server));
         }
         es.invokeAll(tasks);
      } catch (SocketException e) {
         System.err.println("Error creating socket: "+e.getMessage());
      } catch (InterruptedException e) {
         System.err.println("This should never happen: "+e.getMessage());
      }
   }
}
```

```java
public class EchoServerTask implements Callable<Integer>{

    DatagramSocket socket;

    EchoServerTask(DatagramSocket ds){
        this.socket=ds;}

    public void print(String message){
        System.out.println(Thread.currentThread().getName()+message);}

    public Integer call() {
        DatagramPacket request= new DatagramPacket(new byte[EchoServer.MAX_LENGTH],
                EchoServer.MAX_LENGTH);
        while(true){
            try {
                this.print(" Waiting for clients...");
                socket.receive(request);
                this.print(" Received: “+
                            new String(request.getData(),0,request.getLength(),"UTF-8"));
                DatagramPacket response= new DatagramPacket(
                        request.getData(), request.getLength(),request.getSocketAddress());
                socket.send(response);
                request.setLength(EchoServer.MAX_LENGTH);
            } catch (IOException e) {
                System.err.println("Error in communication: "+e.getMessage());
            }
        }}}
```

Reset dimensione del pacchetto

Cliente “Echo” con 2 thread

```
class EchoClientThreaded {

    public static void main(String[] args) {
        ExecutorService es= Executors.newSingleThreadExecutor();
        try(DatagramSocket client= new DatagramSocket();
                BufferedReader reader= new BufferedReader(
                        new InputStreamReader(System.in))){
            es.execute(new ReceiverThread(client));
            SocketAddress server= new InetSocketAddress(
                    InetAddress.getLocalHost(), EchoServer.PORT);
            String message;
            System.out.println("Enter a message to send or 'exit' to finish.");
            while(!(message=reader.readLine()).trim().toLowerCase().equals("exit")){
                byte[] bMessage=message.getBytes("UTF-8");
                DatagramPacket request= new DatagramPacket(bMessage, bMessage.length,server);
                try {
                    client.send(request);
                } catch (IOException e) {
                    System.err.println("Error in communication: "+e.getMessage());
                }
                System.out.println("Enter a message to send or 'exit' to finish.");
            }
            es.shutdownNow();
            System.out.println("Bye!");
        } catch (IOException e) {
            System.err.println("Error with socket: "+e.getMessage());
        }}}
```

Avviamo thread di receive

Questo thread fa solo send

Alla fine interrompiamo anche l’altro thread

```java
public class ReceiverThread implements Runnable{

    DatagramSocket socket;
    ReceiverThread(DatagramSocket sa){
        this.socket=sa;
    }
    @Override
    public void run() {
        try{
            this.socket.setSoTimeout(10000);
            DatagramPacket response= new DatagramPacket(
                    new byte[EchoServer.MAX_LENGTH], EchoServer.MAX_LENGTH);
            while (!Thread.interrupted()){
                try{
                    this.socket.receive(response);
                    System.out.println("Received response: "+
                            new String(response.getData(),0,response.getLength(),"UTF-8"));
                    response.setLength(EchoServer.MAX_LENGTH);
                } catch(IOException e){}
            }
        } catch(SocketException e){
            System.err.println("Error with socket: "+e.getMessage());
        }
    }
}
```

Task per thread di receive

Timeout necessario per gestire interruzione

Si ferma quando interrotto

Receive

Reset dimensione del pacchetto

```
Enter a message to send or 'exit' to finish.
hello
Enter a message to send or 'exit' to finish.
Received response: hello
how are you?
Enter a message to send or 'exit' to finish.
Received response: how are you?
fine thanks and you?
Enter a message to send or 'exit' to finish.
Received response: fine thanks and you?
bye
Enter a message to send or 'exit' to finish.
Received response: bye
exit
Bye!
```

```
pool-1-thread-1 Waiting for clients...
pool-1-thread-3 Waiting for clients...
pool-1-thread-2 Waiting for clients...
pool-1-thread-5 Waiting for clients...
pool-1-thread-4 Waiting for clients...
pool-1-thread-1 Received: hello
pool-1-thread-4 Received: how are you?
pool-1-thread-5 Received: fine thanks and you?
pool-1-thread-2 Received: bye
```

Mettiamo un delay nella risposta del server:

Output del cliente diventa:

```
Enter a message to send or 'exit' to finish.
hello
Enter a message to send or 'exit' to finish.
how are you?
Enter a message to send or 'exit' to finish.
everything ok?
Enter a message to send or 'exit' to finish.
Received response: hello
Received response: how are you?
bye
Enter a message to send or 'exit' to finish.
Received response: everything ok?
exit
Bye!
```

Output del server rimane lo stesso:

```
pool-1-thread-5 Waiting for clients...
pool-1-thread-1 Waiting for clients...
pool-1-thread-4 Waiting for clients...
pool-1-thread-2 Waiting for clients...
pool-1-thread-3 Waiting for clients...
pool-1-thread-5 Received: hello
pool-1-thread-3 Received: how are you?
pool-1-thread-2 Received: everything ok?
pool-1-thread-4 Received: bye
```

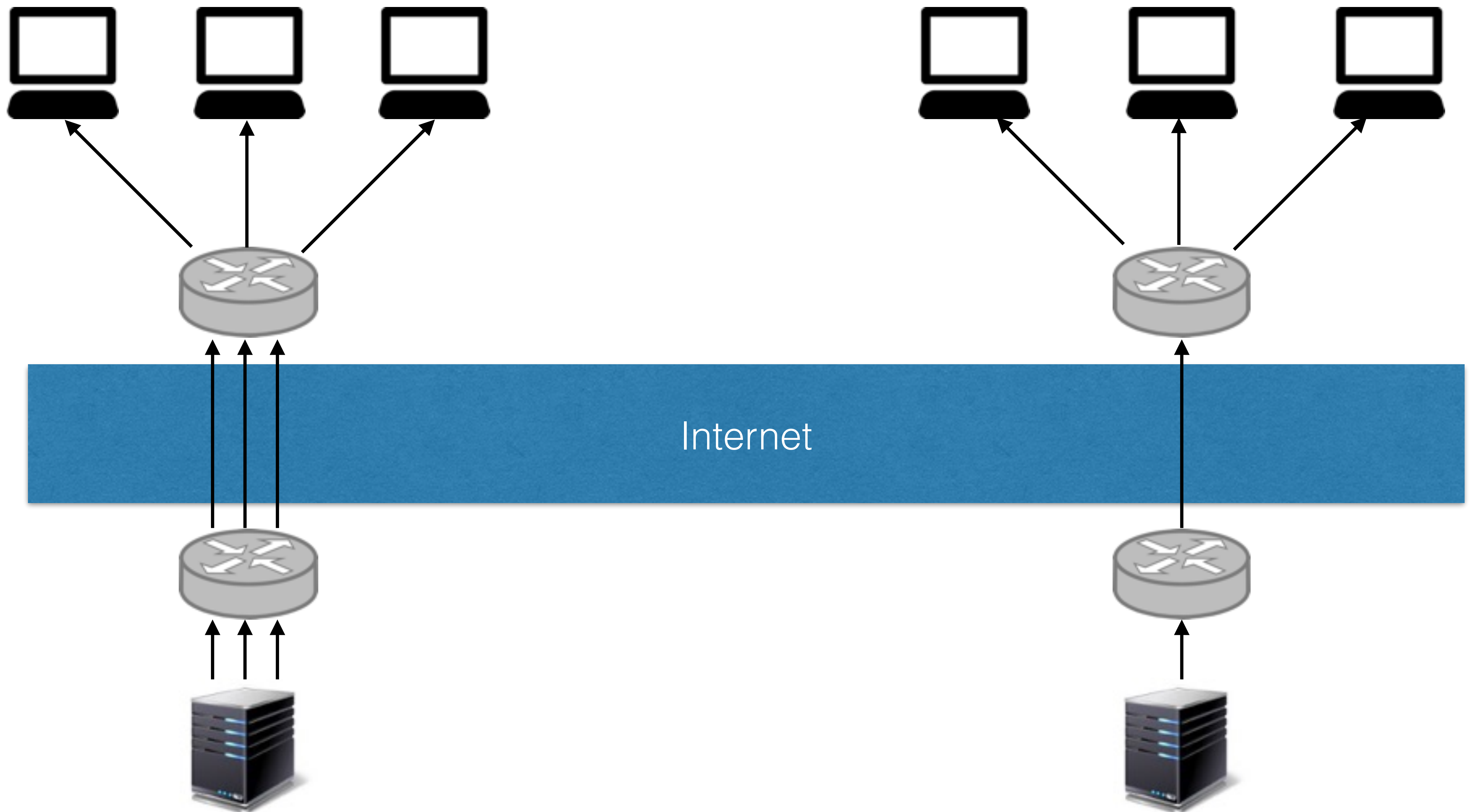
IP *multicast*

- I `DatagramSocket` sono *unicast* - sempre un mittente e una destinazione
- A volte è utile fare *broadcast*: un mittente invia un messaggio a tutti gli altri
    - E.g. TV *broadcast* - segnale raggiunge tutti, anche quelli che non lo usano
    - *Broadcast* diventa molto costoso per reti grandi, quindi è di solito limitato a reti locali - un *broadcast* su tutto Internet bloccherebbe tutto
- *Multicast* - metodo di mezzo - un mittente invia un messaggio a più destinazioni, però non a tutte.
    - Gli interessati si devono registrare per ricevere il messaggio
    - Potrebbe essere implementato come *multi-unicast* - tanti *socket* che inviano a tutti i clienti registrati - l'informazione viene moltiplicata quindi non è efficiente - se i clienti sono troppi anche il *server* soffre

IP *multicast*

- Si usano *socket multicast*
- Un *socket multicast* invia una copia del messaggio verso i clienti
- Questa copia viene moltiplicata in punti specifici della rete, dai *router*
- Nella vicinanza dei clienti, i *router* fanno delle copie addizionali, una per ogni cliente.
- In questo modo i dati non viaggiano più volte sulla stessa strada verso i clienti
- Gruppo *multicast:* l'insieme di host registrati per ricevere messaggi multicast da un *server* - identificato con un indirizzo IP

Multi-Unicasting vs multicasting



IP *multicast*

- Applicazioni
  - Audio e video for TV casting e conferencing
  - Giochi multiplayer, file system distribuiti, calcolo distribuito, basi di dati distribuiti, scoperta di servizi

IP *multicast*

- grande parte del lavoro viene fatto dai *router*
- come programmatori : inviamo dati ad un indirizzo *multicast* (UDP) - non è tanto diverso da inviare ad un indirizzo normale
- attenzione al valore TTL *(time-to-live*) del pacchetto

Indirizzo *multicast*

- Identifica un gruppo *multicast*: una lista di *host*
- Una *host* può aggiungersi o rimuoversi dal gruppo quando vuole
- IPv4: tra 224.0.0.0 e 239.255.255.255 (tutti gli indirizzi che cominciano con 1110)
- IPV6: tutti gli indirizzi che cominciano con 0xFF (11111111)
- Può essere associato con un host name
    - e.g. Network Time Protocol: 224.0.1.1, ntp.mcast.net.

Indirizzo *multicast*

- Per creare un gruppo *multicast*: scegliere un indirizzo *multicast* e cominciare ad inviare dati
- Di solito i gruppi non sono permanenti - vivono tanto quanto hanno clienti
- Qualche indirizzo permanente è stato fissato per qualche servizio
  - i router usano indirizzi 224.0.0.* per protocolli di routing o scoperta di gateway
  - 224.1.*.* e 224.2.*.* per altri servizi
  - iana.org - per la lista completa di servizi con indirizzo permanente
- Tutti gli altri indirizzi sono disponibili
- I *router multicast* (*mrouter*) garantiscono che non ci sono due servizi che usano lo stesso indirizzo di gruppo

TTL

- Ogni volta che un pacchetto passa per un *router*, il TTL viene decrementato
- Originariamente usato per evitare pacchetti che vanno in circuiti chiusi durante il *routing*
- In multicasting, TTL è usato per limitare geograficamente l'invio degli pacchetti
    - TTL=0 - host
    - TTL=1 - *subnet* locale
    - TTL=16 - *campus* locale (anche con più *subnet*)
    - TTL=48 - paese
    - TTL=64 - continente
    - TTL=255- mondo

Funzionamento del *multicasting*

server

se TTL>1

i ‘verdi’ sono parte del gruppo *multicast*

Processo dipende dall’esistenza di *router multicast*
Multicast dentro la rete locale è di solito consentito

Multicasting in Java

- `MulticastSocket`: classe che estende `DatagramSocket` - un *multicast socket* è un *socket* UDP
- Simile a `DatagramSocket`: si usa `DatagramPacket` per inviare o ricevere pacchetti, e gli stessi metodi di `DatagramSocket`
- Differenze:
  - si usa un indirizzo *multicast* per inviare
  - per ricevere si deve aderire ad un gruppo *multicast*

MulticastSocket

- Per inviare:
  - Creare `MulticastSocket` (o anche `DatagramSocket`)

`MulticastSocket(int port)`

`MulticastSocket(SocketAddress bindaddr)`

  - Opzioni:

`void setTimeToLive(int ttl) throws IOException`

Imposta il TTL dei pacchetti inviati usando `send()`. Il valore implicito e 1.

`void setLoopbackMode(boolean disable) throws SocketException`

`boolean getLoopbackMode() throws SocketException`

se `false`, i pacchetti inviati sono anche ricevuti dallo stesso *socket* (se parte del gruppo multicast)

  - Creare `DatagramPacket` con indirizzo multicast e inviare usando `send()` (simile a DatagramSocket)

MulticastSocket

- Per ricevere:
  - Creare `MulticastSocket` e aderire a un gruppo *multicast*

    `void joinGroup(InetAddress address) throws IOException`

    `void joinGroup(SocketAddress address, NetworkInterface interface) throws IOException`

  - Creare `DatagramPacket` e richiamare metodo `receive()` (simile a `DatagramSocket`)
  - Lasciare il gruppo *multicast*

    `void leaveGroup(InetAddress address) throws IOException`

    `void leaveGroup(SocketAddress multicastAddress, NetworkInterface interface) throws IOException`

```
public class MulticastServer {

   public static final int PORT=2000;
   public static final String IP="225.1.1.1";


   public static void main(String[] args) {
      String c="Current time: ";

      try(MulticastSocket server = new MulticastSocket(MulticastServer.PORT);){

         server.setTimeToLive(1);
         server.setLoopbackMode(false);
         server.setReuseAddress(true);

         InetAddress multicastGroup=InetAddress.getByName(MulticastServer.IP);
```

```
    for (int i=0;i<10;i++){
      if (i==9)
        c="finish";

      ByteArrayOutputStream byteStream= new ByteArrayOutputStream();
      DataOutputStream out= new DataOutputStream(byteStream);
      out.writeUTF(c);
      out.writeLong(System.currentTimeMillis());

      byte[] data= byteStream.toByteArray();
      DatagramPacket packet= new DatagramPacket(data, data.length,
          multicastGroup, MulticastServer.PORT);

      server.send(packet);

      System.out.println("Sent packet");

      Thread.sleep(20000);
    }
  } catch (IOException e) {
    System.out.println("Some error appeared: "+e.getMessage());
  } catch (InterruptedException e) {
    System.out.println("This should not happen: "+e.getMessage());
  }}}
```

```java
public class MulticastClient {

   public static void main(String[] args) {

      try(MulticastSocket client = new MulticastSocket(MulticastServer.PORT);){
         InetAddress multicastGroup=InetAddress.getByName(MulticastServer.IP);
         client.joinGroup(multicastGroup);
         DatagramPacket packet= new DatagramPacket(new byte[512], 512);
         String header;
         while(true){
            client.receive(packet);

            DataInputStream in = new DataInputStream(new ByteArrayInputStream(
                    packet.getData(),packet.getOffset(),packet.getLength()));

            System.out.println((header=in.readUTF())+" “+
                    new Date(in.readLong()));
            if (header.equals("finish"))
               break;
         }
      } catch (IOException e) {
         System.out.println("Some error appeared: "+e.getMessage());
         e.printStackTrace();
      }
   }
}
```

```
Current time:  Fri Apr 01 20:17:35 CEST 2016
Current time:  Fri Apr 01 20:17:55 CEST 2016
Current time:  Fri Apr 01 20:18:15 CEST 2016
Current time:  Fri Apr 01 20:18:35 CEST 2016
Current time:  Fri Apr 01 20:18:55 CEST 2016
Current time:  Fri Apr 01 20:19:15 CEST 2016
Current time:  Fri Apr 01 20:19:35 CEST 2016
Current time:  Fri Apr 01 20:19:55 CEST 2016
finish Fri Apr 01 20:20:15 CEST 2016
```

```
Sent packet
Sent packet
Sent packet
Sent packet
Sent packet
Sent packet
Sent packet
Sent packet
Sent packet
Sent packet
```

```
Current time:  Fri Apr 01 20:18:15 CEST 2016
Current time:  Fri Apr 01 20:18:35 CEST 2016
Current time:  Fri Apr 01 20:18:55 CEST 2016
Current time:  Fri Apr 01 20:19:15 CEST 2016
Current time:  Fri Apr 01 20:19:35 CEST 2016
Current time:  Fri Apr 01 20:19:55 CEST 2016
finish Fri Apr 01 20:20:15 CEST 2016
```

Opzione JVM:
-Djava.net.preferIPv4Stack=true

Esercizio: Java Ping

- Implementare un servizio Ping : verifica la qualità della linea verso un server, e la reattività del server.
- 2 misure: pacchetti persi e round trip time (RTT)

Cliente

- Invia 10 pacchetti UDP verso il *server*, con formatto:

PING seqno timestamp

dove **seqno** va da 0 a 9 e **timestamp** e il momento della spedizione

- Dopo ogni pacchetto aspetta una risposta, con *timeout* di 2 secondi
- Se risposta arriva deve memorizzare il tempo trascorso tra l'invio del pacchetto originale e la ricezione della risposta, e scrivere un messaggio sullo schermo.
- Se risposta non arriva entro il *timeout*, scrive "*" sullo schermo.
- Alla fine scrive un riassunto:

---- PING Statistics ----

10 packets transmitted, 7 packets received, 30% packet loss

round-trip (ms) min/avg/max = 63/190.29/290

Server

- Simile ad un *echo server*: una volta ricevuto un pacchetto rinvia il contenuto al mittente.
- Deve simulare la latenza della rete usando `sleep` con tempo aleatorio (tra 0 e 1 secondo)
- Deve simulare la perdita dei pacchetti: con probabilità 25% non risponde al messaggio ricevuto
- L'azione scelta viene scritta sullo schermo (dopo l'IP e il *port* del cliente): "PING not sent" o "PING delayed by n ms"

Ultimo esempio

- Implementiamo insieme la chat room usando UDP e multicast