Università degli Studi di Bologna
Scuola di Ingegneria

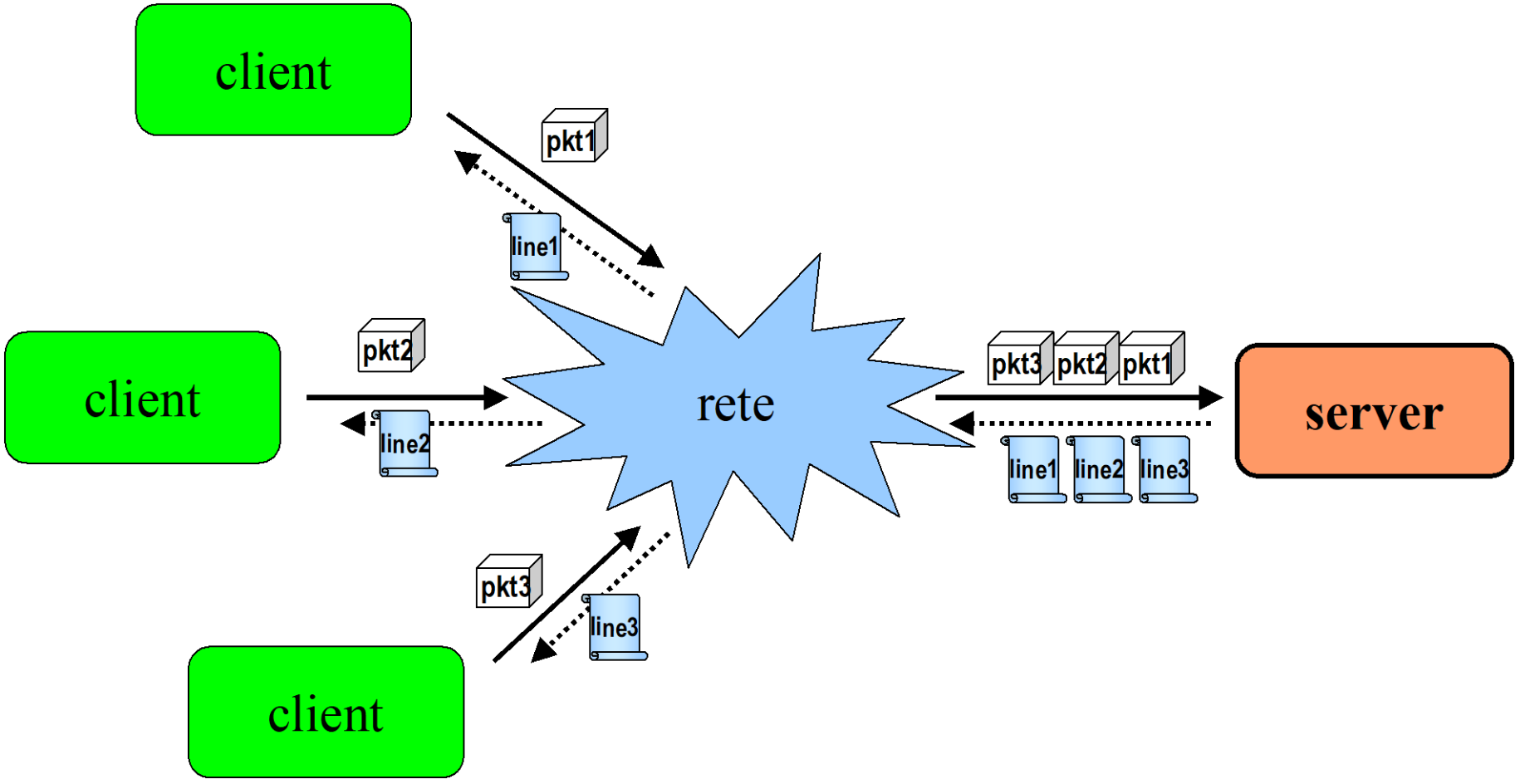
Corso di
Reti di Calcolatori T

Esercitazione 1 (svolta)
Socket Java senza connessione

Luca Foschini

Anno accademico 2020/2021

Architettura



Specifica: il Client

Sviluppare un'applicazione C/S in cui il **client**, invocato con **l'indirizzo IP e la porta** su cui il server è in ascolto, invia al server pacchetti contenenti **il nome del file testo e il numero della linea del file che vuole ricevere**, che sono richiesti dall'utente usando l'input da console. Si noti **l'invio di un unico datagramma per più informazioni** (perché non di due separati?)

Il **cliente è un filtro**

Successivamente il client **stampa il contenuto del datagram ottenuto in risposta**, che può essere la **linea richiesta** o una risposta negativa. Per evitare che, in caso di caduta del server o perdita di un pacchetto, il client si blocchi indefinitamente nell'attesa della risposta è previsto un timeout di 30 s.

> Per il settaggio del timeout sull'operazione di lettura si faccia uso della funzione **`setTimeout`**, invocata sulla socket dopo la costruzione della stessa. Attraverso tale funzione è infatti possibile evitare la condizione di blocco indefinito qualora si invochino operazioni bloccanti sul di una socket. Le primitive bloccanti saranno infatti sbloccate dopo un tempo pari al timeout settato.

Specifica: il Server

Il **server** riceve la richiesta inviata dal client

Il server **verifica l'esistenza del file richiesto e serve la richiesta**

Se il file non esiste, notifica l'errore, altrimenti tenta di estrarre la **linea richiesta**

Infine, **restituisce la risposta** al client: se la linea non esiste invia una notifica di errore; altrimenti invia al client (o ai client) un pacchetto contenente la **linea richiesta**

NOTA: il server è realizzato come **server sequenziale** (e quindi mono-processo), che esegue un **ciclo infinito** in cui compie i seguenti passi fondamentali:

1. attesa di richiesta;
2. generazione della risposta;
3. invio della risposta.

La Classe DatagramPacket

Classe con cui vengono **rappresentati i pacchetti datagram UDP da inviare** e ricevere sulle socket di tipo Datagram.

Si costruisce un **datagram packet** specificando:

- il contenuto del messaggio (i primi `ilength` bytes dell'array `ibuf`);
- l'indirizzo IP del destinatario;
- il numero di porta su cui il destinatario è in ascolto.

```
public DatagramPacket(byte ibuf[], int ilength,
                      InetAddress iaddr, int iport)
```

Se il pacchetto deve essere ricevuto basta definire il contenuto:

```
public DatagramPacket(byte ibuf[], int ilength)
```

La classe mette a disposizione una serie di metodi per estrarre o settare le informazioni:

```
getAddress(),    setAddress(InetAddress addr)
getPort(),       setPort(int port)
getData(),       setData(byte[] buf), etc.
```

La Classe InetAddress

Classe con cui vengono rappresentati gli indirizzi Internet, astraendo dal modo con cui vengono specificati (a numeri o a lettere)

→ **portabilità e trasparenza**

Non ci sono costruttori: utilizzo di tre metodi statici:

```
public static InetAddress getByName(String hostname);
```

restituisce un oggetto InetAddress rappresentante l'host specificato (come nome o indirizzo numerico); con il parametro **null** ci si riferisce all'indirizzo di default della macchina locale

```
public static InetAddress[] getAllByName(String hostname);
```

restituisce un array di oggetti InetAddress; utile in casi di più indirizzi IP registrati con lo stesso nome logico

```
public static InetAddress getLocalHost();
```

restituisce un oggetto InetAddress corrispondente alla macchina locale; se tale macchina non è registrata oppure è protetta da un firewall, l'indirizzo è quello di loopback: 127.0.0.1

Tutti possono sollevare l'eccezione **`UnknownHostException`**
se l'indirizzo specificato non può essere risolto (tramite il DNS)

Uso di strutture dati a livelli

La struttura dati di Java tendono a nascondere la memoria
In genere **il livello di sistema è nascosto**, pensate **ad indirizzi e rappresentazioni di aree di memoria** per i dati
Abbiamo risorse di sistema che rendono necessario potere fare i conti con la memoria e magari vederla con due viste o occhiali di diverso livello (che ci semplificano i compiti)
I datagram usano array di byte: byte [ ] bt

`ByteArrayOutputStream`
Una classe di **array di byte** su cui devono fare operazioni di basso livello e che dovrebbero usare operazioni pesanti per l'utente…

`DataOutputStream`
Stream di **dati di livello Java** che consentono operazioni su **tipi di alto livello di Java**, come readUTF e writeUTF per lavorare su stringhe

Basando i **due stream sulla stessa memoria possiamo lavorare sia a basso livello sia ad alto, secondo necessità**

```
ByteArrayOutputStream lowStream = new ByteArrayOutputStream;
DataOutputStream highStream = new DataOutputStream(lowStream);
```

Schema di soluzione: il Client

1. Creazione socket, set di eventuali opzioni socket e creazione DatagramPacket:

```
DatagramSocket socket =  new DatagramSocket();
socket.setTimeout(30000);
byte[] buf = new byte[256];
DatagramPacket packet =
        new DatagramPacket(buf, buf.length, addr, port);
```

2. Interazione da console con l'utente:

```
BufferedReader stdIn =
  new BufferedReader(new InputStreamReader(System.in));
System.out.print("\n^D(Unix)/^Z(Win) invio per uscire, "+
  "altrimenti inserisci nome file");
while ((nomeFile=stdIn.readLine())!=null){
  System.out.print("Numero linea? ");
  numLinea = Integer.parseInt(stdIn.readLine());
  String richiesta=nomeFile+" "+numLinea;
  …
}
```

Schema di soluzione: il Client (ancora)

3. Creazione del pacchetto di richiesta con le informazioni inserite dall'utente:

```
ByteArrayOutputStream boStream = new ByteArrayOutputStream();
DataOutputStream doStream = new DataOutputStream(boStream);
doStream.writeUTF(richiesta);
byte[] data = boStream.toByteArray();
```

4. Riempimento ed invio del pacchetto al server:

```
packet.setData(data);
socket.send(packet);
```

5. Inizializzazione e attesa del pacchetto di risposta:

```
packet.setData(buf);
socket.receive(packet);
```

6. Estrazione delle informazioni dal pacchetto ricevuto:

```
ByteArrayInputStream biStream =
  new ByteArrayInputStream(packet.getData(),0,packet.getLength());
DataInputStream diStream = new DataInputStream(biStream);
String risposta = diStream.readUTF();
```

Schema di soluzione: il Server

1. Creazione socket e DatagramPacket:

```
DatagramSocket socket = new DatagramSocket(port);
byte[] buf = new byte[256];
DatagramPacket packet = new DatagramPacket(buf,buf.length);
```

2. Inizializzazione e attesa del pacchetto di richiesta:

```
packet.setData(buf);
socket.receive(packet);
```

3. Estrazione delle informazioni dal pacchetto ricevuto:

```
ByteArrayInputStream biStream =
  new ByteArrayInputStream(packet.getData(),0,packet.getLength());
DataInputStream diStream = new DataInputStream(biStream);
richiesta = diStream.readUTF();
StringTokenizer st = new StringTokenizer(richiesta);
nomeFile = st.nextToken();
numLinea = Integer.parseInt(st.nextToken());
```

Schema di soluzione: il Server (ancora)

4. Preparazione della risposta con la linea richiesta:

```
String linea = LineUtility.getLine(nomeFile, numLinea);
ByteArrayOutputStream boStream = new ByteArrayOutputStream();
DataOutputStream doStream = new DataOutputStream(boStream);
doStream.writeUTF(linea);
data = boStream.toByteArray();
```

5. Riempimento e invio del pacchetto al client:

```
packet.setData(data, 0, data.length);
socket.send(packet);
```

Ovviamente, si deve sempre fare la primitiva close() di tutte le socket non più necessarie

LineUtility 1/2

```
public class LineUtility {

// Metodo per restituire una linea specificata in una posizione di un certo file testo
   static String getLine(String nomeFile, int numLinea)
   { String linea = null;
     BufferedReader in = null;
    try { in = new BufferedReader(new FileReader(nomeFile)); }
    catch (FileNotFoundException e)
     { e.printStackTrace(); return linea = "File non trovato"; }
    try
     { for (int i=1; i<=numLinea; i++)
       { linea = in.readLine();
          if ( linea == null)
          { linea = "Linea non trovata"; in.close(); return linea; }
       }
     }
     catch (IOException e)
       { e.printStackTrace(); return linea = "Linea non trovata"; }
     return linea;
   }
```

LineUtility 2/2

/* Metodo per recuperare la linea successiva di un file già aperto (servirà fra poco...): */

```
 static String getNextLine(BufferedReader in)
 { String linea = null;
  try
    { if ((linea = in.readLine()) == null)
      { in.close(); linea = "Nessuna linea disponibile";
      }
    }
   catch (FileNotFoundException e)
     {e.printStackTrace(); return linea = "File non trovato";}
   catch (IOException e)
     {e.printStackTrace(); linea = "Nessuna linea disponibile"; }

   return linea;
  }
}
```

Line Client 1/3

```
public class LineClient {
 public static void main(String[] args)
 { InetAddress addr=null; int port = -1;
  try
  {if (args.length == 2)
    {addr = InetAddress.getByName(args[0]);
     port = Integer.parseInt(args[1]);
    }
   else{ System.out.println("Usage: java LineClient addr port");
         System.exit(1);
   }   }
  catch(UnknownHostException e){...}
  DatagramSocket socket=null; DatagramPacket packet = null;
  byte[] buf = new byte[256];

  try{
    socket = new DatagramSocket(); socket.setSoTimeout(30000);
    packet = new DatagramPacket(buf, buf.length, addr, port);
  }
  catch (SocketException e) { e.printStackTrace();System.exit(2);}
```

Line Client 2/3

// parte di interazione con l'utente

```
BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
System.out.print("\n^D(Unix)/^Z(Win) invio per uscire, altrimenti nome file");
try{   // strutture dati varie
  ByteArrayOutputStream boStream = null; DataOutputStream doStream = null;
  byte[] data = null; String nomeFile = null; int numLinea = -1;
  String richiesta=null; String risposta = null;
  ByteArrayInputStream biStream = null; DataInputStream diStream = null;

  while ((nomeFile=stdIn.readLine())!=null) //Filtro ben fatto
  { try
   { System.out.print("Numero linea? ");
     numLinea = Integer.parseInt(stdIn.readLine());
     richiesta = nomeFile+" "+numLinea;
   }
  catch (Exception e) {
    System.out.println("Problemi nell'interazione da console: ");
    e.printStackTrace(); System.out.print("\n^D(Unix)/... ");
    continue;
  }
```

Line Client 3/3

```
 try{   // preparazione e spedizione richiesta
    boStream = new ByteArrayOutputStream();
    doStream = new DataOutputStream(boStream);
    doStream.writeUTF(richiesta);
    data = boStream.toByteArray();
    packet.setData(data); socket.send(packet);
  }
 catch (IOException e){...}
 try{   // inizializzazione e ricezione pacchetto
    packet.setData(buf); socket.receive(packet);
  }
 catch (IOException e){ /* come sopra */ }
 try{
   biStream = new ByteArrayInputStream( packet.getData(),0,packet.getLength());
   diStream = new DataInputStream(biStream);
   risposta = diStream.readUTF(); … // stampa risposta
  }
 catch (IOException e){... }
 System.out.print("\n^D(Unix)/^Z(Win...");
}
System.out.print("\n^D(Unix)/^Z(Win) invio per uscire, altrimenti nome file");
} //while
} // try
catch(Exception e){...}
socket.close(); System.out.println("LineClient:termino...");
}}
```

Line Server 1/3

```
public class LineServer {
 // Porta nel range consentito 1024-65535!
 private static final int PORT = 4445;

public static void main(String[] args)
{DatagramSocket socket = null; DatagramPacket packet = null;
 byte[] buf = new byte[256];


 // Controllo argomenti input: 0 oppure 1 argomento (porta)
 if ((args.length == 0)) { port = PORT; }
 else if (args.length == 1) {
   try {
      port = Integer.parseInt(args[0]);
     // controllo argomento e che la porta sia nel range consentito 1024-65535
      if (port < 1024 || port > 65535) {
         System.out.println("Usage: java LineServer [serverPort>1024]");
         System.exit(1);
     }
   } catch (NumberFormatException e) { … }
 } else {System.out.println("Usage: java LineServer [serverPort>1024]");  System.exit(1); }
```

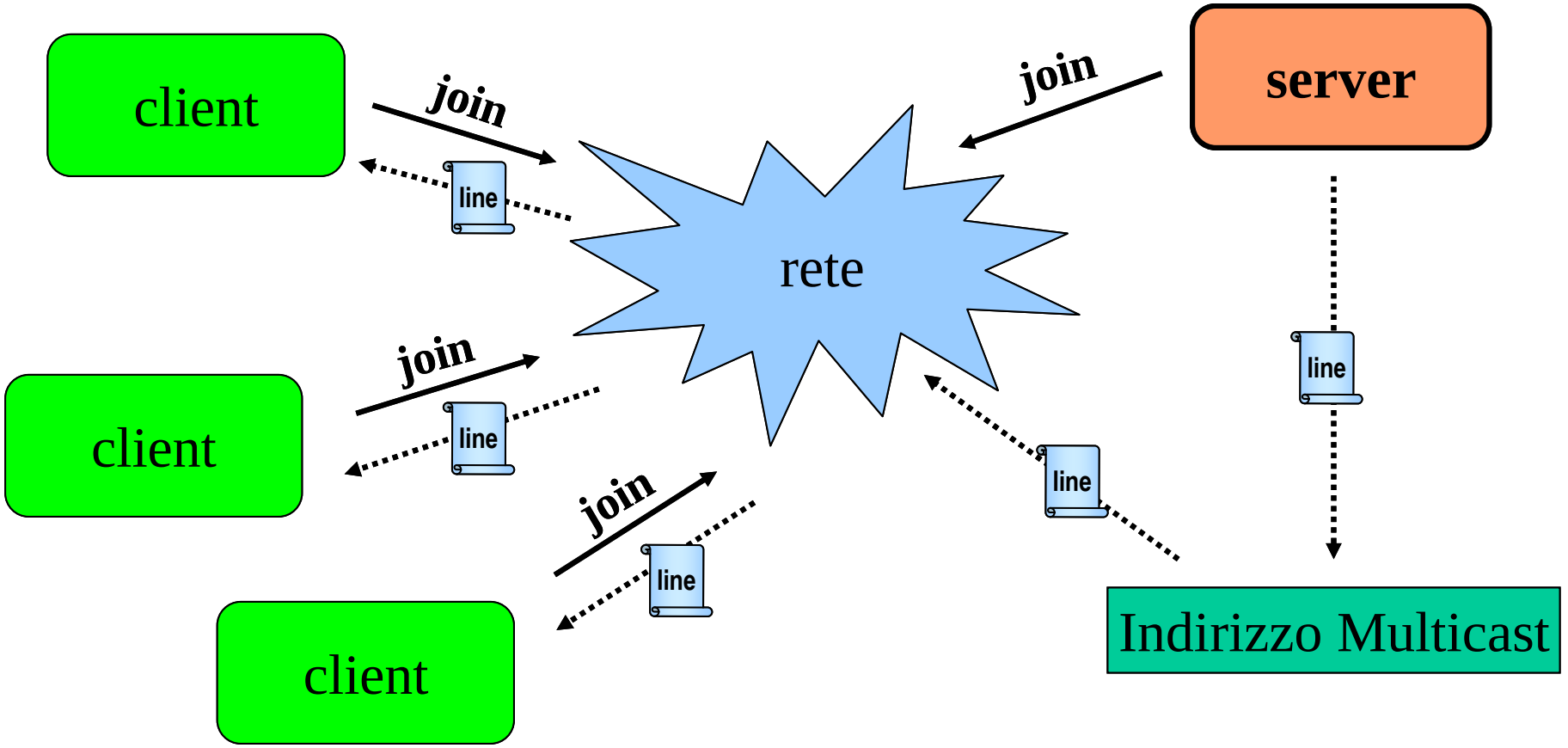
Line Server 2/3

```
try
 { socket = new DatagramSocket(port);
   packet = new DatagramPacket(buf, buf.length);
 } catch (SocketException e) {...}

 try
 { String nomeFile = null;
   int numLinea = -1;
   String richiesta = null;
   ByteArrayInputStream biStream = null;
   DataInputStream diStream = null;
   StringTokenizer st = null;
   ByteArrayOutputStream boStream = null;
   DataOutputStream doStream = null;
   String linea = null;
   byte[] data = null;

while (true)
{
try { packet.setData(buf); socket.receive(packet); }
     catch(IOException e){ ... continue; }
```

Line Server 3/3

```
try
 {  biStream = new ByteArrayInputStream(packet.getData(),0,
                                        packet.getLength());
    diStream=new DataInputStream(biStream);
    richiesta = diStream.readUTF();
    st = new StringTokenizer(richiesta);
    nomeFile = st.nextToken();
    numLinea = Integer.parseInt(st.nextToken());
 } catch(Exception e){/*… come sopra */}

try
  { String linea =  LineUtility.getLine(nomeFile, numLinea);
    boStream = new ByteArrayOutputStream();
    doStream = new DataOutputStream(boStream);
    doStream.writeUTF(linea);
    data = boStream.toByteArray();
    packet.setData(data); socket.send(packet);
  } catch(IOException e){/* come sopra */}
 }   // while
} catch(Exception e){...}
System.out.println("LineServer: termino..");
socket.close();
} // main
} // class
```

Socket Multicast: architettura



Specifica dettagliata Multicast

Modificare il programma sviluppato nella prima parte dell'esercitazione in accordo ai requisiti seguenti:

- I client **non inviano più nessun pacchetto di richiesta** al server, ma **si associano al gruppo** a cui il server invia periodicamente le linee di un certo file, **ne ricevono alcune** (per es. 20), **le stampano**, poi **si dissociano dal gruppo**; il programma client è lanciato con due argomenti: l'indirizzo multicast e la porta sulle quale il server trasmette.
- Il server **invia periodicamente** (per es. ogni 2 secondi) **una linea di un certo file** ad un prefissato indirizzo multicast (es. 230.0.0.1) e porta. Terminata la lettura, il server riprende dall'inizio la lettura del file e continua ad eseguire per tutta la propria esistenza questo ciclo infinito di letture e spedizioni.

Il server deve inoltre poter essere invocato da riga di comando con uno o due argomenti e cioè con la sola porta (in questo caso si utilizza l'indirizzo multicast di default, es. 230.0.0.1), oppure l'indirizzo multicast e la porta, presso cui effettuare gli invii.

Schema di soluzione: il Client Multicast

1. Creazione socket, set eventuali opzioni socket e creazione DatagramPacket:

```
MulticastSocket socket = new MulticastSocket(port);
socket.setTimeout(20000);
byte[] buf = new byte[256];
DatagramPacket packet = new DatagramPacket(buf, buf.length);
```

2. Adesione al gruppo multicast passato come argomento:

```
address = InetAddress.getByName(args[0]);
socket.joinGroup(address);
```

3. Inizializzazione e ricezione del pacchetto di risposta:

```
packet.setData(buf); socket.receive(packet);
```

4. Estrazione delle informazioni dal pacchetto ricevuto:

```
biStream = new ByteArrayInputStream(packet.getData(), 0,
  packet.getLength());
diStream = new DataInputStream(biStream); risposta = diStream.readUTF();
```

5. Uscita dal gruppo multicast

```
socket.leaveGroup(address);
```

Schema di soluzione: il Server Multicast

1. Creazione socket e DatagramPacket:

```
socket = new MulticastSocket(port);
byte[] data = new byte[256];
DatagramPacket packet =
    new DatagramPacket(data, data.length, group, port);
```

2. Adesione al gruppo multicast:

```
address = InetAddress.getByName(stringAddr);
socket.joinGroup(address);
```

3. Preparazione della risposta da inviare:

```
String linea = LineUtility.getNextLine(in);
boStream = new ByteArrayOutputStream();
doStream = new DataOutputStream(boStream);
doStream.writeUTF(linea);
data = boStream.toByteArray();
```

4. Riempimento e invio del pacchetto a tutto il gruppo:

```
packet.setData(data); socket.send(packet);
```

Line Multicast Client 1/2

```
public class MulticastClient {
public static void main(String[] args) {
InetAddress group = null; int port = -1;
try
{ if (args.length == 2) {
   group = InetAddress.getByName(args[0]);
   port = Integer.parseInt(args[1]);
  } else
  {  System.out.println("Usage: java MulticastClient MCastAddr MCastPort");
      System.exit(1);
  }
 } catch (Exception e){... System.exit(2);}
```

// creazione socket multicast e creazione datagram packet

```
MulticastSocket socket = null; DatagramPacket packet = null;
byte[] buf = new byte[256];

try {
  socket = new MulticastSocket(port);
  socket.setSoTimeout(20000); // 20 secondi
  packet = new DatagramPacket(buf,buf.length);
  System.out.println("Creata la socket: " + socket);
} catch (IOException e){... System.exit(3);}
```

Line Multicast Client 2/2

```
try
{ socket.joinGroup(group); // adesione al gruppo associato
  System.out.println("Adesione al gruppo " + addr);
} catch (IOException e){... System.exit(4);}

ByteArrayInputStream biStream = null;
DataInputStream diStream = null; String linea = null;
for (int i = 0; i < 20; i++) // ciclo di ricezione
{try
{packet.setData(buf); socket.receive(packet);
 } catch (SocketTimeoutException te)
 { System.out.println("Non ho ricevuto niente per 20 secondi, chiudo!"); System.exit(4);
 } catch (IOException e) {... continue;}
 try {
   biStream = new ByteArrayInputStream(packet.getData(),0,packet.getLength());
   diStream = new DataInputStream(biStream); linea = diStream.readUTF();
   System.out.println("Linea ricevuta: " + linea);
  } catch (IOException e) {... continue;}
}
System.out.println("\nUscita dal gruppo"); // uscita dal gruppo e chiusura della socket
try { socket.leaveGroup(addr); } catch (IOException e) { ... }
System.out.println("MulticastClient: termino...");
socket.close();
} }
```

Line Multicast Server 1/3

```
public class MulticastServer {
/* File di lettura per sempre, nella directory dove viene lanciato il server */
private static final String FILE = "saggezza.txt";
private static BufferedReader in = null;
private static boolean moreLines = true;

public static void main(String[] args) {
long WAIT = 1000; int port = -1;
InetAddress group = null; // configurazione Indirizzo del gruppo multicast e porta

try
{ if (args.length == 1)
  { group = InetAddress. getByName("230.0.0.1");
    port = Integer.parseInt(args[0]); }
  else if (args.length == 2)
  { group = InetAddress.getByName(args[0]);
    port = Integer.parseInt(args[1]); }
  else
  { System.out.println("Usage: ..."); System.exit(1); }
}
catch (Exception e) {... System.exit(1);}
```

Line Multicast Server 2/3

```
// Creazione datagram packet e socket
MulticastSocket socket = null; byte[] data = new byte[256];
DatagramPacket packet = new DatagramPacket(data, data.length, group, port);
try {
   socket = new MulticastSocket(port);
   socket.joinGroup(group);
    System.out.println("Socket: " + socket);
} catch (Exception e) { ... }
int count = -1;  // contatore per debug
ByteArrayOutputStream boStream = null;  DataOutputStream doStream = null;

while (true)  //  ciclo infinito di apertura file e lettura
{ count = 0; moreLines = true; // azzero count e indicatore linee

  try { // costruisco un nuovo buffered reader
    in = new BufferedReader(new FileReader(FILE));
   } catch (Exception e) {... continue; }
  while (moreLines) {
   count++; byte[] buf = new byte[256];
   // estrazione della linea
   String linea = LineUtility.getNextLine(in);
    if (linea.equals("Nessuna linea disponibile"))
     { moreLines = false; break;  //esco dal ciclo }
```

Line Multicast Server 3/3

```
 try
  { boStream = new ByteArrayOutputStream();
    doStream = new DataOutputStream(boStream);
    doStream.writeUTF(linea);
    data = boStream.toByteArray();
    packet.setData(data);
    socket.send(packet);
  } catch (Exception e) {... continue; }

  // attesa tra un invio e l'altro...
  try {
    Thread.sleep( 2 * WAIT );
  } catch (Exception e) {... continue; }

  } // while interno
  } // while esterno
 } // main
} // class
```

Problemi utilizzo MulticastSocket

Uso multicast in reti locali

In alcune reti locali il **multicast** potrebbe essere **disabilitato**.

Uso multicast in computer non connessi in rete

I problemi d'uso delle MulticastSocket con computer non connessi in rete, si possono superare impostando l'interfaccia di rete **PRIMA** della join.

Bisogna quindi sostituire questo codice:

```
socket.joinGroup(group);
```

Con il seguente:

```
// ottengo Inetaddress locale
InetAddress netInterface =
InetAddress.getByName("localhost");
// imposto l'interfaccia di rete
socket.setInterface(netInterface);
socket.joinGroup(group);
```

Per risolvere questo ed altri problemi realizzativi che si possono presentare durante il corso si tenga sempre monitorata **la sezione FAQ del sito!!!**