Appunti dei corsi di

Programmazione di Rete
Sistemi di elaborazione: Reti II

PROF. G. BONGIOVANNI

2) Sviluppo di applicazioni basate sullo scambio di messaggi

Vedremo ora come costruire, su uno stream quale quello offerto da una connessione TCP, una comunicazione basata su un flusso di messaggi anziché, come fatto finora, su un flusso di byte.

Un ***messaggio*** è una quantità arbitraria di informazioni (in genere tali da avere una propria autosufficienza logica e funzionale) che vengono corredate da ***informazioni di controllo*** (***CI***) e quindi inviate come un'unità a se stante che prende il nome di ***pacchetto***.

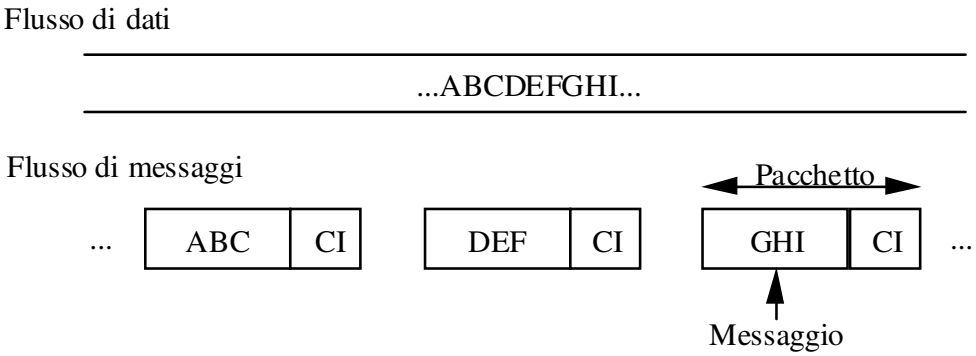

Figura 2-1: Dati, messaggi e pacchetti

Si noti come, per una comunicazione di questo tipo, non sia strettamente necessaria una connessione basata su TCP.

Anche UDP sarebbe adeguato, se non fosse per il fatto che non è un protocollo affidabile. Di conseguenza, noi ricaveremo il supporto alla gestione di messaggi a partire da stream affidabili quali quelli offerti da TCP.

I principali benefici derivanti dall'uso di messaggi sono i seguenti:

1. facilità di multiplexing e demultiplexing. Se su una singola connessione incidono più trasmissioni indipendenti, è più facile gestire tale situazione incapsulando i dati da trasmettere. Il server può estrarli indipendentemente gli uni dagli altri, senza entrare nel merito dei contenuti;
2. gli errori sono confinati all'interno di un messaggio. Per la stessa ragione di prima, eventuali errori presenti in un messaggio non influenzano i messaggi successivi. Senza l'incapsulamento l'errore potrebbe rendere impossibile determinare il confine fra due messaggi;
3. facilità di trasmettere dati di natura diversa sulla stessa connessione. Ciò è strettamente legato al punto 1. Infatti, dato che ogni messaggio è corredato di informazioni di controllo,

esse consentono di comunicare metainformazioni sulla natura dei dati contenuti nel messaggio.

Tipicamente i messaggi possono essere usati per:
- corredare le informazioni trasmesse di una minimale informazione di controllo, ad esempio la lunghezza del messaggio (sarà la prima cosa che vedremo);
- consentire a parti diverse di una stessa applicazione di condividere lo stesso stream per la comunicazione (la seconda cosa che vedremo);
- consentire a un'applicazione di specificare, per mezzo delle informazioni di controllo, una lista di destinatari. Ciò consente trasmissioni di tipo ***multicast***. Questa tecnica non la vedremo in dettaglio, ma sarà chiaro come possa essere implementata.

2.1) Classi di base per la gestione di messaggi

2.1.1) Classe MessageOutput

E' la superclasse astratta da cui derivano tutti gli stream per la gestione di messaggi (che indicheremo col termine generico di ***message stream***) di output.

E' un `FilterOutputStream` (e quindi si deve attaccare a un `OutputStream`) e, per convenienza, estende `DataOutputStream`.

Ai metodi di quest'ultima classe aggiunge tre metodi `send()`.

Su un message stream di output si opera in questo modo:
- usando i normali metodi di `DataOutputStream` si scrive un messaggio, finché esso non è completato (tipicamente usando un buffer interno);
- usando uno dei metodi `send()` lo stream effettua le seguenti operazioni:
  - incapsula il messaggio in un pacchetto, corredandolo delle informazioni di controllo;
  - invia il pacchetto sul canale di comunicazione;
  - azzera il buffer interno.

Figura 2-2: Funzionamento di un message stream di output

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.          *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;

/**
 * The superclass for all message output streams.
 * <p>Extends <tt>DataOutputStream</tt> and adds <tt>send()</tt>
 * methods that send the current message to the attached channel.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.MessageInput
 */
public abstract class MessageOutput extends DataOutputStream {
  /**
   * Creates a new <tt>MessageOutput</tt>.
   * @param out Stream to which to write message data.
   */
  public MessageOutput (OutputStream out) {
    super (out);
  }

  /**
   * Sends the current message to the attached channel.
   * <p>Subclasses will extend this class and implement this
   * method as appropriate for a particular communications
   * channel.
   * @exception IOException Occurs if there is a problem sending
   * a message.
   */
  public abstract void send () throws IOException;

  /**
   * Sends the current message to the attached channel with
   * a routing header that indicates a list of recipients.
   * <p>Subclasses that support this method will override it
   * with an appropriate implementation. The default implementation
   * is to throw an exception.
   * @param dst The list of intended recipients
   * @exception IOException Occurs if there is a problem sending
   * a message or this method is not supported.
   */
  public void send (String[] dst) throws IOException {
    throw new IOException ("send[] not supported");
  }

  /**
   * Sends the current message to the attached channel with
   * a routing header that indicates a single recipient.
   * <p>The default implementation of this method calls the
   * previous method with a single-element array.
   * @param dst The intended recipient
   * @exception IOException Occurs if there is a problem sending
   * a message or targeted sending is not supported.
   */
```

```
  public void send (String dst) throws IOException {
    String[] dsts = { dst };
    send (dsts);
  }
}
```

Note

- Si estende `DataOutputStream`, ereditando quindi i suoi metodi.
- Il costruttore accetta un `OutputStream` a cui attaccarsi e lo passa al costruttore della superclasse. I vari metodi di scrittura della superclasse scriveranno direttamente su tale stream. Esso, nelle classi derivate, tipicamente sarà un `ByteArrayOutputStream`, ossia uno stream che scrive in un buffer (costituito da un array di byte) in memoria.
- `send()` trasmette materialmente il messaggio (dopo averlo incapsulato) sul canale di comunicazione, cioé scrive l'intero pacchetto sull'`OutputStream` a cui è attaccato.
- Le altre due varianti del metodo `send(…)`, che noi non useremo, predispongono alla trasmissione multicast.

2.1.2) Classe MessageInput

E' la superclasse astratta da cui derivano tutti i message stream di input.

E' un `FilterInputStream` (e quindi si deve attaccare a un `InputStream`) e, per convenienza, estende `DataInputStream`.

Ai metodi di quest'ultima classe aggiunge un metodo `receive()`.

Con un message stream di input opera in questo modo:

- chiamando `receive()` ci si blocca in attesa che un pacchetto sia ricevuto dal canale di comunicazione. Quando ciò accade, il messaggio viene estratto dal pacchetto e diviene disponibile;
- chiamando i vari metodi di lettura di `DataInputStream` si leggono i dati del messaggio;
- chiamando nuovamente `receive()` si attende un nuovo pacchetto, il cui contenuto sovrascrive il messaggio precedente (anche se quest'ultimo non è stato completamente letto).

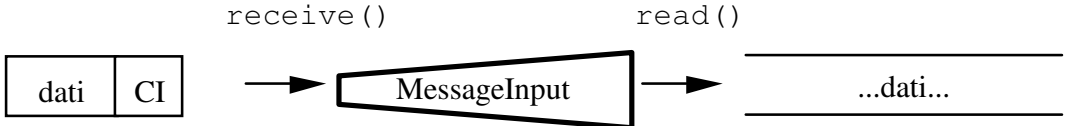

Figura 2-3: Funzionamento di un message stream di input

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.          *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;

/**
 * The superclass for all message input streams.
 * <p>Extends <tt>DataInputStream</tt> and adds a <tt>receive()</tt>
 * method that receives a new message from the attached channel.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.MessageOutput
 */
public abstract class MessageInput extends DataInputStream {
  /**
   * Creates a new <tt>MessageInput</tt>.
   * @param in Stream from which to read message data
   */
  public MessageInput (InputStream in) {
    super (in);
  }

  /**
   * Receives a new message from the attached channel and makes
   * it available to read through the standard <tt>DataInputStream</tt>
   * methods.
   * <p>Subclasses will extend this class and implement this method
   * as appropriate for a particular communications channel.
   * @exception IOException Occurs if there is a problem receiving
   * a new message.
   */
  public abstract void receive () throws IOException;
}
```

Note

- Si estende `DataInputStream`, ereditando i suoi metodi.
- Il costruttore accetta un `InputStream` a cui attaccarsi e lo passa al costruttore della superclasse. I vari metodi di lettura della superclasse leggeranno direttamente da tale stream. Esso, nelle classi derivate, sarà tipicamente uno stream che legge da un buffer in memoria.
- `receive()` si blocca finché non arriva un pacchetto, estrae il messaggio e lo rende disponibile nel buffer di cui sopra.

2.1.3) Classe MessageOutputStream

Questa è la prima implementazione di `MessageOutput` che vedremo, e si attacca a un `OutputStream`.

Questa classe, in particolare, incapsula i messaggi in pacchetti la cui Control Information è costituita dalla lunghezza in byte del messaggio. Ciò consente al destinatario di sapere in anticipo quanto è grande il messaggio, senza doverne analizzare alcuna parte.

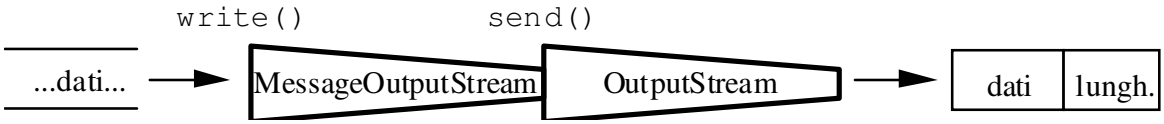

Figura 2-4: Funzionamento di `MessageOutputStream`

Internamente, il meccanismo di buffering si ottiene per mezzo di un `ByteArrayOutputStream`.

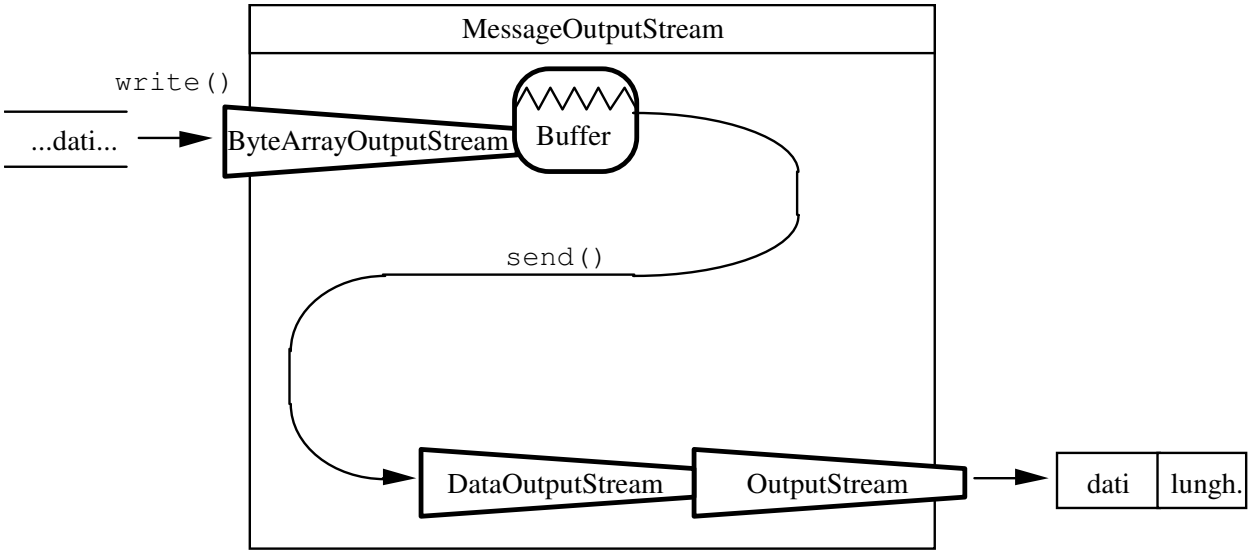

Figura 2-5: Implementazione di `MessageOutputStream`

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.          *
 * See the file legal.txt in the txt directory for details. */
```

```
package prominence.msg;

import java.io.*;

/**
 * A <tt>MessageOutput</tt> that writes messages to an attached
 * <tt>OutputStream</tt>.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.MessageInputStream
 */
public class MessageOutputStream extends MessageOutput {
  /**
   * The attached <tt>OutputStream</tt>.
   */
  protected OutputStream o;
  /**
   * A <tt>DataOutputStream</tt> attached to <tt>o</tt>.
   */
  protected DataOutputStream dataO;
  /**
   * A <tt>ByteArrayOutputStream</tt> used to buffer the current message.
   */
  protected ByteArrayOutputStream byteO;

  /**
   * Creates a new <tt>MessageOutputStream</tt>. Message data is
   * buffered internally until <tt>send()</tt> is called.
   * @param o The <tt>OutputStream</tt> to which to write messages
   */
  public MessageOutputStream (OutputStream o) {
    super (new ByteArrayOutputStream ());
    byteO = (ByteArrayOutputStream) out;
    this.o = o;
    dataO = new DataOutputStream (o);
  }

  /**
   * Sends a message to the attached stream. The message body length
   * is written as an <tt>int</tt>, followed by the message body itself;
   * the internal message buffer is then reset.
   * <p>This method synchronizes on the attached stream.
   * @exception IOException Occurs if there is a problem writing to
   * the attached stream.
   */
  public void send () throws IOException {
    synchronized (o) {
      dataO.writeInt (byteO.size ());
      byteO.writeTo (o);
    }
    byteO.reset ();
    o.flush ();
  }
}
```

Note

- la classe estende `MessageOutput` e quindi `DataOutputStream`;
- i metodi di `DataOutputStream` scrivono nel buffer interno;

- `send()` invia un pacchetto, formato dalla lunghezza dati più il contenuto del buffer, sullo stream di output a cui il `MessageOutputStream` è attaccato.

Costruttore

- Chiama quello di `MessageOutput`, il quale a sua volta chiama quello di `DataOutputStream`. Però, al costruttore della superclasse non viene passato il vero `OutputStream` a cui ci si deve attaccare, ma un nuovo `ByteArrayOutputStream`, creato per l'occasione, che costituirà il buffer interno. Di conseguenza, la superclasse sarà attaccata a tale buffer, e i suoi metodi di scrittura opereranno su di esso.
- Ricordiamo che `FilterOutputStream` tiene in una variabile `out` una reference all'`OutputStream` a cui è attaccato (cioè quello che gli viene passato nel costruttore). Usiamo tale variabile per ricavare una reference al buffer interno, `byteO`, che serve per usare i metodi di `ByteArrayOutputStream`.
- Dopodiché attacchiamo al vero `OutputStream`, cioè ad `o`, un `DataOutputStream` (per poterne usare i metodi) e utilizziamo `dataO` come reference ad esso.
- Chiudere con `close()` il MessageOutputStream, che non è attaccato al vero `OutputStream`, non chiude quest'ultimo ma il buffer interno `byteO`, il che non ha alcun effetto.

Metodi

- Tutti i normali metodi di `DataOutputStream` scriveranno dunque dentro `byteO`, il buffer interno.
- Il metodo `send()`, invece, compie le seguenti azioni (bloccandosi se necessario):
  - si sincronizza sul vero `OutputStream`, garantendo così una corretta gestione del canale di comunicazione nel caso vi fossero attacati molteplici `MessageOutputStream`;
  - scrive sull'`OutputStream` un intero pari alla dimensione corrente del buffer;
  - scarica il contenuto del buffer sull'`OutputStream`;
  - resetta il buffer interno;
  - chiama `flush()` per inviare subito il pacchetto (lunghezza più dati).

2.1.4) Classe MessageInputStream

Questa classe, che estende `MessageInput`, si attacca a un `InputStream` e va usata per leggere i dati che provengono da un `MessageOutputStream`.

Essa, col metodo `receive()`, estrae il messaggio dal pacchetto (eliminando la CI, costituita dalla dimensione del messaggio) e quindi rende disponibile il contenuto del messaggio ai normali metodi di lettura (quelli di `DataInputStream`), memorizzandolo in un buffer interno.

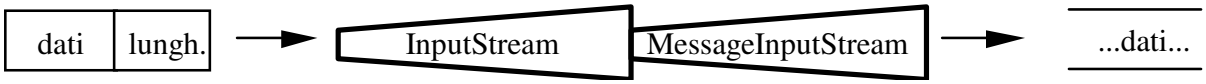

Figura 2-6: Funzionamento di `MessageInputStream`

Internamente, si usa un ByteArrayInputStream per implementare il meccanismo di buffering.

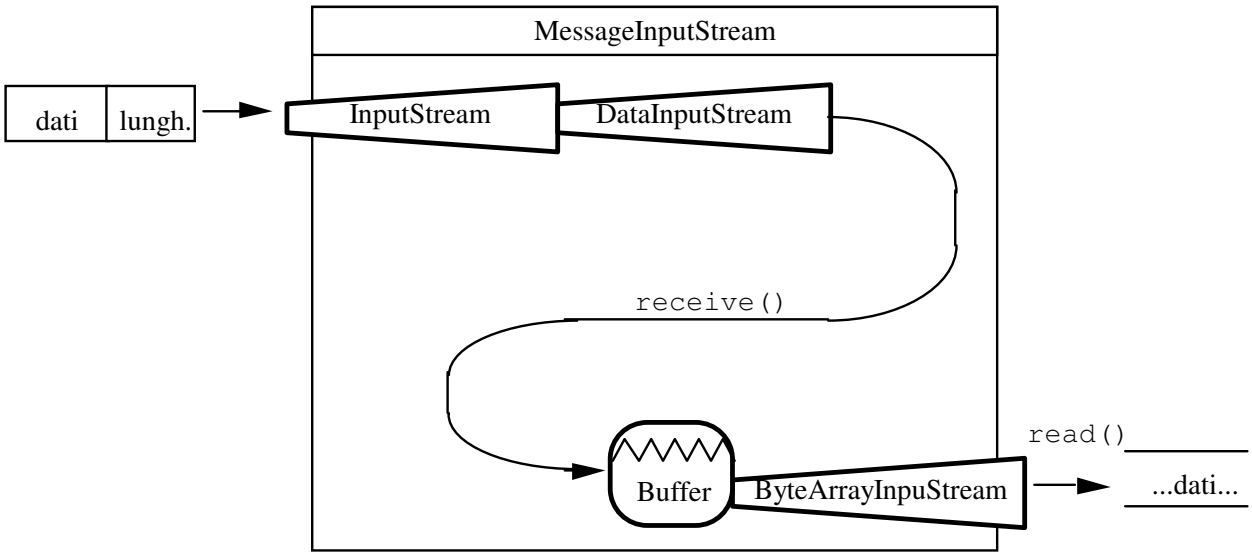

Figura 2-7: Implementazione di `MessageInputStream`

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.        *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;

/**
 * A <tt>MessageInput</tt> that reads messages from an
 * attached <tt>InputStream</tt>.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.MessageOutputStream
 */
public class MessageInputStream extends MessageInput {
  /**
   * The attached <tt>InputStream</tt>.
   */
  protected InputStream i;
  /**
   * A <tt>DataInputStream</tt> attached to <tt>i</tt>.
```

```
   */
  protected DataInputStream dataI;

  /**
   * Creates a new <tt>MessageInputStream</tt>.
   * @param i The <tt>InputStream</tt> from which to read messages.
   */
  public MessageInputStream (InputStream i) {
    super (null);
    this.i = i;
    dataI = new DataInputStream (i);
  }

  /**
   * A buffer containing the most recently received message.
   * <p>This variable is exposed to permit potential optimizations.
   */
  byte[] buffer;

  /**
   * Receives a message from the attached stream. An <tt>int</tt>
   * header is read, which indicates the length of the message body.
   * The message body is then read and made available through the
   * usual superclass <tt>read()</tt> methods.
   * <p>This method synchronizes on the attached stream.
   * @exception IOException Occurs if there is a problem reading
   * from the attached stream.
   */
  public void receive () throws IOException {
    synchronized (i) {
      int n = dataI.readInt ();
      buffer = new byte[n];
      dataI.readFully (buffer);
    }
    in = new ByteArrayInputStream (buffer);
  }
}
```

Note

- la classe estende `MessageInput` e quindi `DataInputStream`;
- i metodi di `DataInputStream` leggeranno dal buffer interno (se è stata fatta almeno una `receive()`);
- `receive()` crea un buffer, lo riempie col messaggio e lo attacca al `MessageInputStream`.

Costruttore

- Chiama quello di `MessageInput`, e quindi quello di `DataInputStream`, che però (come prima) non vengono attaccati al vero `InputStream`. In questo caso anzi vengono attaccati a un `null`, perché non ha senso fare normali letture prima che sia arrivato un pacchetto: in tal caso si genera un'eccezione.
- crea in `dataI` un nuovo `DatainputStream` attaccato al canale di comunicazione, per poterne usare i metodi.

Metodi

- Tutti i normali metodi di `DataInputStream` leggeranno da `in` e quindi dal buffer interno.

- Il metodo `receive()` compie le seguenti operazioni (bloccandosi finché non arriva un messaggio):
    - si sincronizza sul canale di comunicazione;
    - legge un intero, che indica la dimensione del messaggio;
    - crea un array di byte di pari dimensioni;
    - legge dal canale di comunicazione un corrispondente numero di byte;
    - crea un nuovo `ByteArrayInputStream`, attaccato all'array appena riempito;
    - attacca il `MessageInputStream` al `ByteArrayInputStream` appena creato, assegnando quest'ultimo a `in` (la variabile di `FilterInputStream` nella quale si memorizza l'`InputStream` a cui il `FilterInputStream` è attaccato).

2.2) Un'applicazione client-server per la gestione di transazioni

Si deve notare che:

- molti `MessageOutputStream` possono essere attaccati a un singolo `OutputStream`
- molti `MessageInputStream` possono essere attaccati a un singolo `InputStream`.

I message stream opereranno comunque correttamente, anche se pilotati da corrispondenti thread in concorrenza, grazie all'incapsulamento e alla sincronizzazione.

Ciò è utile, ad esempio, in un'applicazione per la gestione di transazioni, che si svolgono ciascuna secondo il seguente schema:

1. una richiesta del client;
2. una elaborazione conseguente effettuata dal server (che può richiedere anche molto tempo);
3. una risposta del server.

Pensiamo a un server multithreaded, in cui ogni thread gestisce sequenzialmente una transazione dopo l'altra.

I vari thread operano in concorrenza, per cui mentre uno elabora una transazione, un altro ne accetta una nuova, e così via.

Naturalmente, molte richieste viaggiano assieme in una direzione, e molte risposte viaggiano assieme nell'altra direzione.

Grazie alla struttura di messaggi, queste trasmissioni non si disturbano a vicenda.

Il server:

- contiene una `HashTable` che mappa attributi in valori;

- attiva un numero di thread deciso dall'utente per gestire un corrispondente numero massimo di transazioni in concorrenza.

Il client:
- può leggere (con `get()`) o modificare (con `put()`) il valore di un attributo (queste sono le uniche due possibili transazioni).

2.2.1) Classe TransactionClient

E' il cliente che si connette al corrispondente server.

Usa `MessageOutputStream` e `MessageInputStream` per comunicare. Presenta all'utente due campi testo e due bottoni:
- col bottone "get" si chiede il valore di un attributo;
- col bottone "put" si cambia il valore di un attributo.

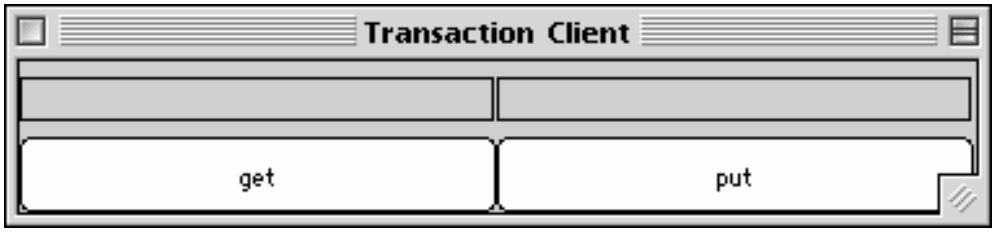

Figura 2-8: Interfaccia utente del client

La definizione della classe è la seguente; si noti che rispetto alla versione del libro ("Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997) c'è una variazione sostanziale, in quanto la ricezione dei dati del server avviene in un thread separato, altrimenti sia il client che il server si bloccano su `receive()`.

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

import prominence.msg.MessageInputStream;
import prominence.msg.MessageOutputStream;

public class TransactionClient extends Frame implements Runnable {
  protected MessageInputStream mI;
  protected MessageOutputStream mO;
  protected Button get, put;
  protected TextField attr, value;
```

```
  public TransactionClient (InputStream i, OutputStream o) {
    super ("Transaction Client");
    mI = new MessageInputStream (i);
    mO = new MessageOutputStream (o);
    attr = new TextField (24);
    value = new TextField (24);
    get = new Button ("get");
    put = new Button ("put");
    setLayout (new GridLayout (2, 2));
    add (attr);
    add (value);
    add (get);
    add (put);
    pack ();
    show ();
  }

  public void run () {
    while (true) {
      try {
        mI.receive ();
        System.out.print ("attr: " + mI.readUTF ());
        System.out.println (" value: " + mI.readUTF ());
      } catch (IOException e) {
            e.printStackTrace();
      }
    }
  }

  public boolean handleEvent (Event e) {
    if ((e.id == e.ACTION_EVENT) && (e.target instanceof Button)) {
      try {
        if (e.target == get) {
          mO.writeUTF ("get");
          mO.writeUTF (attr.getText ());
        } else if (e.target == put) {
          mO.writeUTF ("put");
          mO.writeUTF (attr.getText ());
          mO.writeUTF (value.getText ());
        }
        mO.send ();
      } catch (IOException ex) {
        ex.printStackTrace ();
      }
    }
    return super.handleEvent (e);
  }

  static public void main (String args[]) throws IOException {
    if (args.length != 2)
      throw new RuntimeException ("Syntax: TransactionClient <server> <port>");
    Socket s = new Socket (args[0], Integer.parseInt (args[1]));
    InputStream i = s.getInputStream ();
    OutputStream o = s.getOutputStream ();
    TransactionClient c = new TransactionClient (i, o);
    //c.listen ();
    new Thread(c).start();
  }
}
```

Note

- Il costruttore riceve i due stream per la comunicazione, vi attacca due message stream e costruisce l'interfaccia utente.
- Il `main()` crea il socket per la connessione al server, estrae i due stream, e crea il `TransactionClient`. Quindi avvia un thread separato per la ricezione dei dati.
- Il metodo `run()`, e dunque il thread separato, è un ciclo infinito che:
  - riceve un pacchetto spedito dal server (contenente una coppia di stringhe attributo-valore) con `receive();`
  - stampa sullo standard output la coppia attributo e valore.
- Il metodo `handleEvent()` gestisce la interazione coll'utente, dando via via inizio alle transazioni:
  - bottone get: si invia un pacchetto il cui messaggio è costituito da due stringhe UTF:
    - la prima è "get"
    - la seconda è il nome dell'attributo di cui si vuole conoscere il valore;
  - bottone put: si invia un pacchetto il cui messaggio è costituito da tre stringhe UTF:
    - la prima è "put"
    - la seconda è il nome dell'attributo che si vuole modificare;
    - la terza è il valore che si vuole assegnare all'attributo.

2.2.2) Classe TransactionServer

Questo è il server che gestisce le transazioni. Anche lui usa gli stessi tipi di message stream del client, e grazie ad essi può attivare molteplici thread che lavorano in concorrenza per gestire molte transazioni contemporaneamente. Un ritardo simula artificialmente il tempo necessario a gestire una transazione.

Implementa l'interfaccia `Runnable` per poter lanciare molti thread sullo stesso oggetto.

Il costruttore riceve i due stream per la comunicazione e crea una `HashTable` (essenzialmente una classe che implementa una memoria associativa, contenente coppie chiave-valore e dotata di metodi per l'inserimento e la ricerca di elementi) per memorizzare e ricercare le coppie attributo-valore.

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.        *
 * See the file legal.txt in the txt directory for details. */

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

```
import java.net.*;

import prominence.msg.MessageOutput;
import prominence.msg.MessageInputStream;
import prominence.msg.MessageOutputStream;

public class TransactionServer implements Runnable {
  protected Hashtable h;
  protected InputStream i;
  protected OutputStream o;

  public TransactionServer (InputStream i, OutputStream o) {
    this.i = i;
    this.o = o;
    h = new Hashtable ();
  }
  public void run () {
    MessageInputStream mI = new MessageInputStream (i);
    MessageOutputStream mO = new MessageOutputStream (o);
    try {
      while (true) {
        mI.receive ();
        try {
          Thread.sleep (1000);
        } catch (InterruptedException ex) {
        }
        String cmd = mI.readUTF ();
      System.out.println (Thread.currentThread () + ": command " + cmd);
        if (cmd.equals ("get")) {
          get (mI, mO);
        } else if (cmd.equals ("put")) {
          put (mI);
        }
      }
    } catch (IOException ex) {
      ex.printStackTrace ();
    }
  }

  void get (DataInputStream dI, MessageOutput mO) throws IOException {
    String attr = dI.readUTF ();
    mO.writeUTF (attr);
    if (h.containsKey (attr))
      mO.writeUTF ((String) h.get (attr));
    else
      mO.writeUTF ("null");
    mO.send ();
  }

  void put (DataInputStream dI) throws IOException {
    String attr = dI.readUTF ();
    String value = dI.readUTF ();
    h.put (attr, value);
  }

  static public void main (String args[]) throws IOException {
    if (args.length != 2)
      throw new RuntimeException ("Syntax: TransactionServer <port> <threads>");
    ServerSocket server = new ServerSocket (Integer.parseInt (args[0]));
    Socket s = server.accept ();
    server.close ();
    InputStream i = s.getInputStream ();
    OutputStream o = s.getOutputStream ();
    TransactionServer t = new TransactionServer (i, o);
```

```
    int n = Integer.parseInt (args[1]);
    for (int j = 0; j < n; ++ j)
      new Thread (t).start ();
  }
}
```

Note

- Il `main()` opera come segue:
  - crea un `ServerSocket`;
  - accetta una connessione e chiude subito il `ServerSocket`;
  - deriva i due stream per la comunicazione;
  - crea il `TransactionServer`;
  - lancia su di esso un certo numero (passato come parametro) di thread.
- Il metodo `run()` di ogni thread è un ciclo infinito, in cui:
  - si creano i due message stream, locali al thread, per la comunicazione;
  - si aspetta un messaggio dal client, con `receive()`;
  - appena arriva il messaggio, che è relativo a una nuova transizione, la si gestisce:
    - si simula un ritardo di 1 secondo;
    - si legge il comando ("get" o "put");
    - si chiama il corrispondente metodo di gestione (`get()` o `put()`):
      - se il comando è "get", il thread spedisce un messaggio di risposta costituito da una coppia di stringhe (attributo e valore);
      - se il comando è "put", il thread cambia il valore dell'attributo (se c'è nella tavola) o inserisce una nuova coppia (se non c'è l'attributo). Questo è ottenuto automaticamente col metodo `put()` di `HashTable`.

Lo scopo fondamentale di questa applicazione client-server è mostrare come l'uso dei message stream ci consenta facilmente di condividere un unico canale di comunicazione fra molti thread concorrenti. L'incapsulamento ci protegge dal mescolamento dei dati.

2.3) Accodamento di messaggi

Avendo la possibilità di gestire stream di messaggi anziché di byte, è possibile crearsi ulteriori strumenti di utilità:

- una coda di messaggi;
- degli stream di messaggi (di input e output) che estraggono (e inseriscono) messaggi dalla/nella coda anziché da una connessione di rete.

Ciò significa avere la possibilità di disaccoppiare fra loro:
- il canale di comunicazione fisico sul quale viaggiano i dati (che partono e arrivano da/nella coda);
- la componente applicativa che utilizza o produce i dati (che legge/scrive da/nella coda).

Le principali conseguenze positive di questo disaccoppiamento sono:
- isolamento dell'applicazione dagli errori di comunicazione;
- isolamento dell'applicazione da eventuali ritardi di trasmissione: le scritture dell'applicazione avvengono nella coda e quindi sono rapide. Eventuali ritardi nell'invio sulla rete penalizzano solo il thread (di norma separato) che li gestisce.

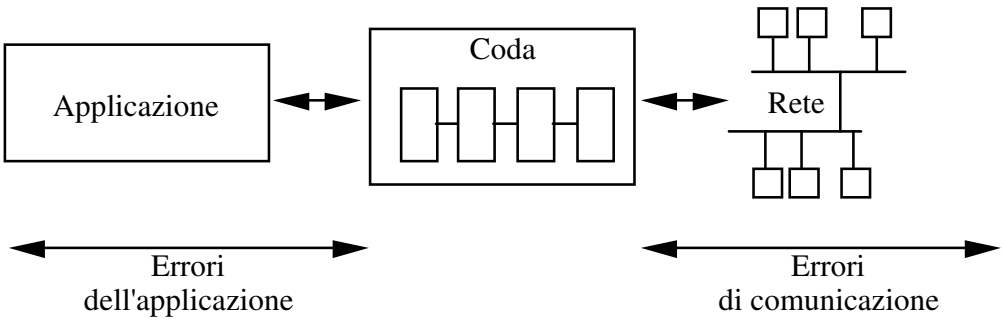

Figura 2-9: Coda di messaggi

2.3.1) Classe Queue

E' una implementazione della ***coda*** (intesa come struttura dati), i cui elementi sono messaggi.

Si noti che nella coda ci sono messaggi e non pacchetti, perché la coda provvede automaticamente a tenere separati tali elementi e quindi non c'è bisogno dell'informazione di controllo.

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.          *
 * See the file legal.txt in the txt directory for details. */

package prominence.util;

import java.util.Vector;

/**
```

```
 * A FIFO (first in, first out) dat1-structure; the opposite of a
<tt>Stack</tt>.
 * Objects are added to the front of the <tt>Queue</tt> and removed from the
back.
 * <p>This implementation blocks the caller who attempts to remove an object
from
 * an empty queue until the queue is non-empty again.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 */
public class Queue {
  /**
   * A <tt>Vector</tt> of the queue elements.
   */
  protected Vector queue;

  /**
   * Creates a new, empty <tt>Queue</tt>.
   */
  public Queue () {
    queue = new Vector ();
  }

  /**
   * Attempts to remove an object from the queue; blocks if there are no objects
   * in the queue. This call will therefore always return an object.
   * @returns The least-recently-added object from the queue
   */
  public Object remove () {
    synchronized (queue) {
      while (queue.isEmpty ()) {
        try {
          queue.wait ();
        } catch (InterruptedException ex) {}
      }
      Object item = queue.firstElement ();
      queue.removeElement (item);
      return item;
    }
  }

  /**
   * Adds an item to the front of the queue, wakes a caller who is waiting for
   * the queue to become non-empty.
   * @param item The object to be added
   */
  public void add (Object item) {
    synchronized (queue) {
      queue.addElement (item);
      queue.notify ();
    }
  }

  /**
   * Returns whether the queue is empty.
   * @returns Whether the queue is empty
   */
  public boolean isEmpty () {
    return queue.isEmpty ();
  }
}
```

Note

- La coda è implementata per mezzo di un `Vector`, classe che offre due metodi, `add()` e `remove()`, per inserire e recuperare un elemento. Va notato che un thread che cerca di estrarre un elemento da una coda vuota viene bloccato, e si risveglierà quando ci sarà qualcosa nella coda.
- Il costruttore crea un `Vector` vuoto, nel quale verrà mantenuta la coda.
- Il metodo `remove()` elimina un elemento dalla coda e lo restituisce al chiamante:
  - contiene un blocco di codice sincronizzato sul `Vector`, al fine di poter gestire la coda per mezzo di thread multipli;
  - si blocca con `wait()`, rilasciando quindi il lucchetto, se la coda è vuota.
- Il metodo `add()` aggiunge un elemento alla coda:
  - contiene un blocco di codice sincronizzato sul `Vector`;
  - provvede a risvegliare con `notify()` uno degli eventuali thread in attesa di un elemento.

2.3.2) Classe QueueOutputStream

Questa classe è un `MessageOutput` che, invece di scrivere messaggi su un `OutputStream`, li inserisce in una coda.

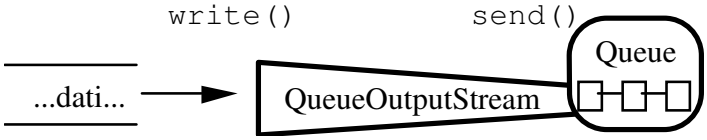

Figura 2-10: `QueueOutputStream`

`send()` aggiunge alla coda un messaggio, costituito dal corrente contenuto del buffer.

Come si noterà, in questo caso non si incapsula il messaggio in un pacchetto, in quanto la coda è una struttura costituita di elementi separati, ciascuno dei quali è un array che internamente ha l'informazione sulle proprie dimensioni. Il contenuto di ogni array coincide con quello del corrispondente messaggio.

Internamente, il buffer si implementa con un ByteArrayOutputStream.

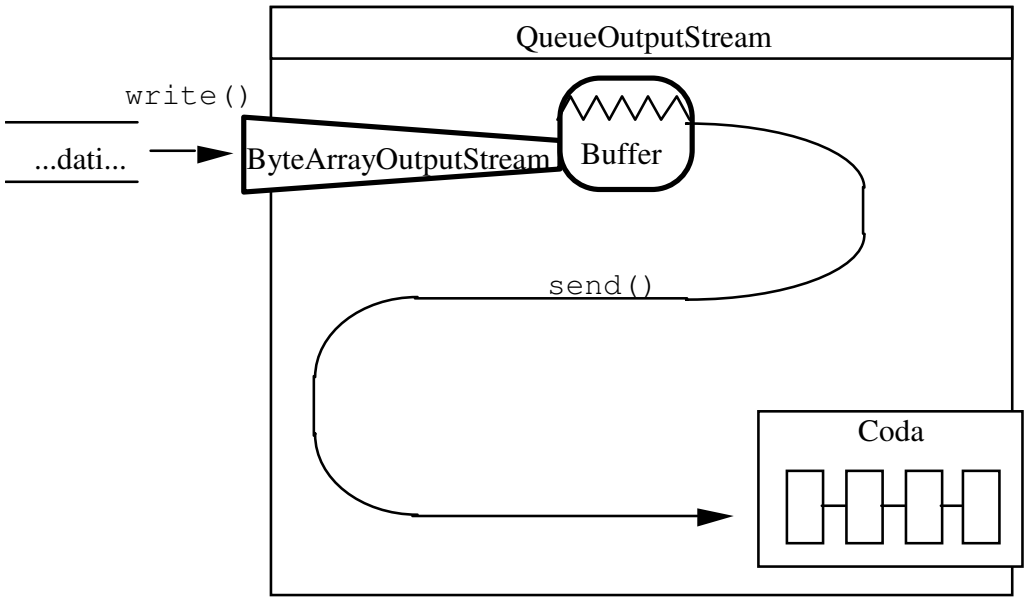

Figura 2-11: Implementazione di `QueueOutputStream`

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.        *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;
import prominence.util.Queue;

/**
 * A <tt>MessageOutput</tt> that inserts messages into a <tt>Queue</tt>
 * of byte arrays.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.QueueInputStream
 */
public class QueueOutputStream extends MessageOutput {
  /**
   * A <tt>ByteArrayOutputStream</tt> used to buffer the current message
   * contents.
   */
  protected ByteArrayOutputStream byteO;
  /**
   * The <tt>Queue</tt> of messages.
   */
  protected Queue q;

  /**
   * Creates a new <tt>QueueOutputStream</tt>.
   * @param q A <tt>Queue</tt> into which messages will be written
```

```
   */
  public QueueOutputStream (Queue q) {
    super (new ByteArrayOutputStream ());
    byteO = (ByteArrayOutputStream) out;
    this.q = q;
  }

  /**
   * Inserts the current message buffer into the <tt>Queue</tt>.
   */
  public void send () {
    byte[] buffer = byteO.toByteArray ();
    byteO.reset ();
    q.add (buffer);
  }
}
```

Note

- Il costruttore riceve come parametro la coda a cui si deve attaccare, e crea un buffer interno su cui avverranno le scritture (analogamente a quanto visto per `MessageOutputStream`). Inoltre, mantiene una reference alla coda cui è attaccato.
- Il metodo `send()` estrae il contenuto attuale del buffer (ossia il messaggio costruito con le scritture) e lo inserisce come nuovo elemento nella coda. Quindi resetta il buffer in modo che le prossime scritture possano costruire il prossimo messaggio.
- Si noti che molti `QueueOutputStream` possono essere attaccati alla stessa coda. Grazie al fatto che l'aggiunta di elementi avviene in un blocco sincronizzato, non si verificano interferenze fra eventuali thread concorrenti, che tipicamente gestiscono i vari `QueueOutputStream`.

2.3.3) Classe QueueInputStream

Questa classe è un `MessageInput` che, invece di leggere messaggi da un `InputStream`, li estrae da una coda.

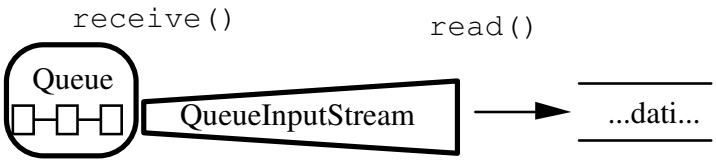

Figura 2-12: `QueueInputStream`

`Receive()` estrae un messaggio dalla coda e rende disponibile il suo contenuto per le letture. Se non ci sono messaggi, blocca il chiamante finché non ce n'è uno disponibile.
Internamente il buffer si implementa con un `ByteArrayInputStream`.

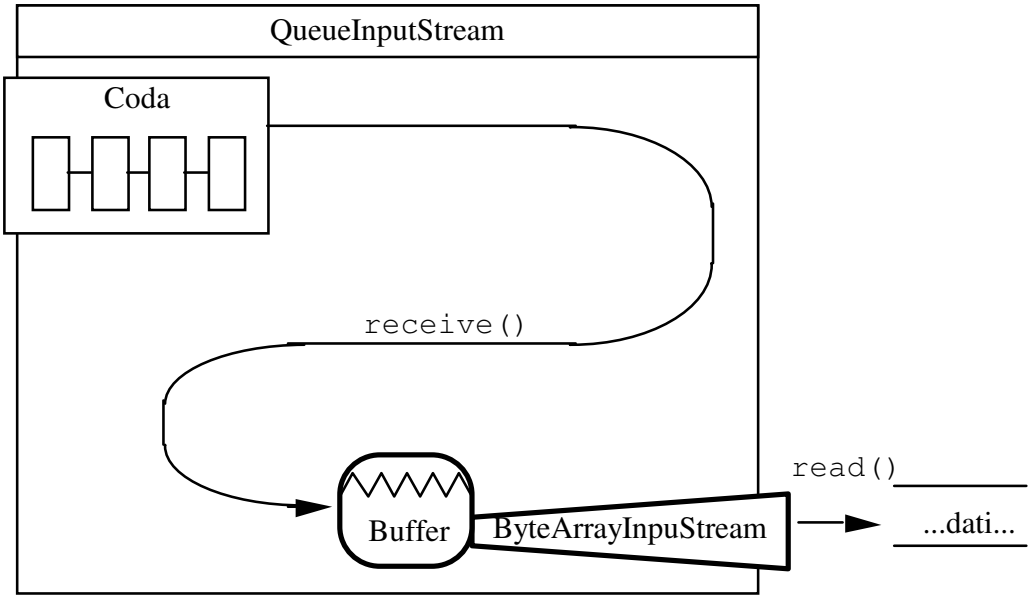

Figura 2-13: Implementazione di `QueueInputStream`

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.          *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;
import prominence.util.Queue;

/**
 * A <tt>MessageInput</tt> that reads messages from a <tt>Queue</tt> of
 * byte arrays.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.QueueOutputStream
 */
public class QueueInputStream extends MessageInput {
  /**
   * The <tt>Queue</tt> of messages.
   */
  protected Queue q;

  /**
   * Creates a new <tt>QueueInputStream</tt>.
   * @param q A <tt>Queue</tt> out of which messages will be read
   */
  public QueueInputStream (Queue q) {
    super (null);
    this.q = q;
```

```
  }

  /**
   * A buffer containing the most recently received message.
   * <p>This variable is exposed to permit potential optimizations.
   */
  byte[] buffer;

  /**
   * Extracts a message from the attached <tt>Queue</tt> and makes
   * it available to read through the usual superclass <tt>read()</tt>
   * methods.
   */
  public void receive () {
    buffer = (byte[]) q.remove ();
    in = new ByteArrayInputStream (buffer);
  }
}
```

Note

- Il costruttore riceve come parametro la coda a cui attaccarsi. Chiama il costruttore della superclasse passandogli `null` (non c'è alcun messaggio da leggere) e mantiene una reference alla coda.
- Il metodo `receive()` estrae un messaggio dalla coda (bloccandosi se non ce ne sono) e lo copia in un array di byte interno. Quindi, analogamente a `MessageInputStream`, crea un `ByteArrayInputStream` attaccato al buffer e si aggancia a tale `ByteArrayInputStream` per le successive letture.
- Anche in questo caso, molti `QueueInputStream` possono essere attaccati alla stessa coda senza problemi di interferenze.

2.3.4) Utilizzo tipico degli stream per l'accodamento di messaggi

Come abbiamo detto, l'accodamento dei messaggi permette di:

- isolare l'applicazione dagli errori di comunicazione;
- isolare l'applicazione dagli eventuali ritardi di trasmissione.

Entrambi questi obiettivi possono essere raggiunti con sistemi analoghi a quelli sotto esposti.

Uso di code in input

Si dedica un thread separato, che chiameremo ***thread copiatore***, alla lettura dei pacchetti dalla connessione di rete. Tale thread non fa altro che ricevere pacchetti dalla rete (attraverso un `MessageInput`) e ricopiarli in una coda (attraverso un `QueueOutputStream`).

Una applicazione che legga i dati da tale coda (attaccandovi un `QueueInputStream`) è quindi isolata dalla rete.

Il codice di tale thread copiatore sarà semplicemente un ciclo infinito di copiatura:

```
...
try{
    while (true) {
        mi.receive();
        byte[] buffer=new byte[mi.available()];
        mi.readFully(buffer);
        mo.write(buffer);
        mo.send();
    }
} catch (IOException e) {
    e.printStackTrace();
}
...
```

dove:

- `mi` è il `MessageInput`;
- `mo` è il `QueueOutputStream`.

Uso di code in output
Si dedica un thread copiatore alla scrittura dei pacchetti sulla connessione di rete. Tale thread preleva i pacchetti da una coda e li ricopia su una connessione di rete.

Una applicazione che scriva i dati nella coda (attaccandovi un `QueueOutputStream`) è quindi isolata dalla rete, sia per quanto riguarda errori che possibili ritardi.

Il codice del thread copiatore è uguale a prima, solo che ora si avrà che:

- `mi` è il `QueueInputStream`;
- `mo` è il `MessageOutput`.

Vedremo più avanti una classe (`Demultiplexer`) che fra le sue funzioni ha anche quella di thread copiatore per isolare una applicazione dalla rete.

2.4) Multiplexing di messaggi

Finora abbiamo sviluppato stream di messaggi piuttosto semplici, che non aggiungono particolari funzionalità ai normali stream, pur presentando già alcuni vantaggi quali:

- facilità di condivisione di un unico canale di comunicazione da parte di più stream di messaggi;
- disaccoppiamento fra rete e applicazione.

Comunque, gli stream visti finora formano la base necessaria sulla quale svilupparne di più sofisticati.

In particolare, ora vedremo come sviluppare degli stream di messaggi che siano in grado di effettuare il ***multiplexing***, su un unico canale di comunicazione, di vari flussi di dati eterogenei fra loro, tipicamente generati da componenti diverse di una applicazione in maniera del tutto trasparente a tali componenti, che in genere ignorano l'esistenza le une delle altre.

Lo scopo si raggiunge incapsulando i messaggi dentro pacchetti nei quali la Control Information è costituita da un' ***etichetta*** che caratterizza l'origine dei dati.

Quando tali pacchetti giungono a destinazione, i messaggi vengono instradati alla corretta componente grazie a un meccanismo corrispondente di ***demultiplexing***, che opera sulla base di tale Control Information.

In particolare, dopo aver introdotto le necessarie classi di utilità, svilupperemo un'applicazione client-server che implementa una chatline più raffinata di quella vista precedentemente.

Il client infatti è costituito da due parti indipendenti fra loro, che consentono l'una di inviare testo e l'altra disegni condividendo lo stesso canale fisico di comunicazione.

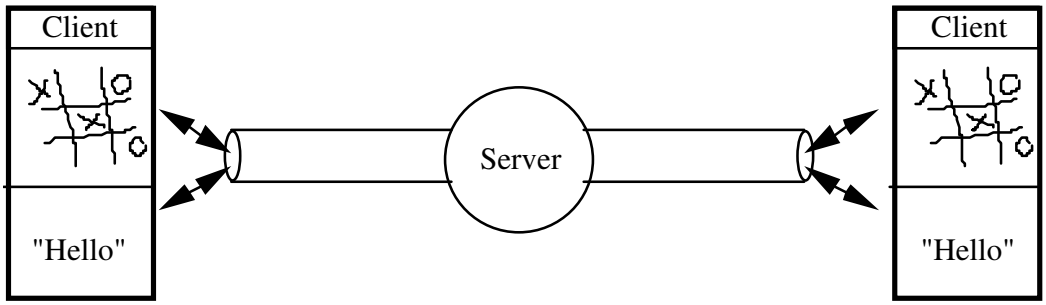

Figura 2-14: Chatline grafica e testuale

Il server ha il compito di effettuare il broadcast dei pacchetti che riceve da un client a tutti gli altri.

Come vedremo, esso non entra nel merito di ciò che riceve, e quindi rimarrà inalterato anche in caso si aggiungano ulteriori moduli funzionali ai client.

2.4.1) Classe MultiplexOutputStream

Questa classe estende `MessageOutput`, però si attacca a un altro `MessageOutput` e non a un semplice `OutputStream`.

Ciò consente di effettuare il multiplexing su un qualunque stream di messaggi (ad esempio su un `QueueOutputStream` oppure un `MessageOutputStream`).

Questo meccanismo ci consente di gestire più livelli di incapsulamento, uno dentro l'altro.

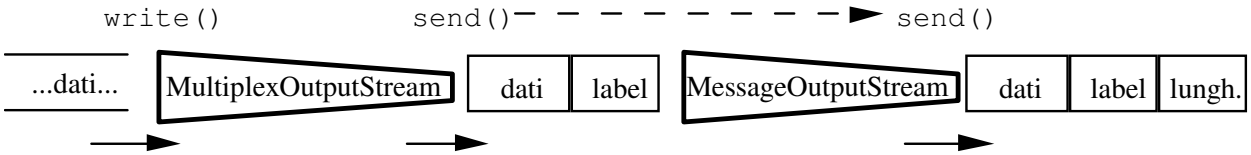

Figura 2-15: Incapsulamento multiplo

In sostanza, aggiungiamo un meccanismo di etichettatura all'`OutputStream` attaccato.

Dall'altra parte del canale, tale etichezza sarà usata per determinare l'origine dei dati.

Nel client che vedremo fra breve useremo questa classe per fare il multiplexing di due flussi di dati (che provengono dalle due componenti separate dell'applicazione) su un unico canale di comunicazione.

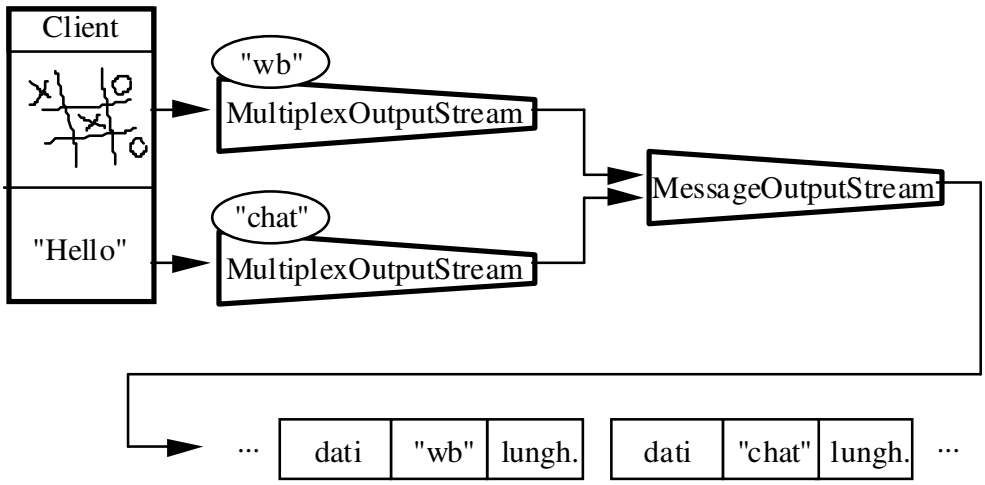

Figura 2-16: Multiplexing dei messaggi della chatline

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.          *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;

/**
 * A <tt>MessageOutput</tt> that attaches to an existing <tt>MessageOutput</tt>
 * and attaches a multiplexing label to the header of each message that
 * is transmitted.
 * <p>The label is specified in the constructor and so one stream always
 * attaches the same label.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.MultiplexInputStream
 */
public class MultiplexOutputStream extends MessageOutput {
  /**
   * The <tt>MessageOutput</tt> to which this is attached.
   */
  protected MessageOutput o;
  /**
   * A <tt>ByteArrayOutputStream</tt> used to buffer the current message
   * contents.
   */
  protected ByteArrayOutputStream byteO;
  /**
   * The multiplexing label.
   */
  protected String label;

  /**
   * Creates a new <tt>MultiplexOutputStream</tt>.
   * @param o The <tt>MessageOutput</tt> to which to send messages
   * @param label The multiplexing label to be used by this stream
   */
  public MultiplexOutputStream (MessageOutput o, String label) {
    super (new ByteArrayOutputStream ());
    byteO = (ByteArrayOutputStream) out;
    this.o = o;
    this.label = label;
  }

  /**
   * Sends the current message with a multiplexing label header to the
   * attached <tt>MessageOutput</tt>.
   * @exception IOException Occurs if there is a problem sending the
   * message.
   */
  public void send () throws IOException {
    synchronized (o) {
      o.writeUTF (label);
      byteO.writeTo (o);
      o.send ();
    }
    byteO.reset ();
```

```
  }

  /**
   * Sends the current message with a multiplexing label header to the
   * attached <tt>MessageOutput</tt>.
   * <p>If the attached <tt>MessageOutput</tt> supports targeted sending
   * then this method will succeed; otherwise an appropriate
   * <tt>IOException</tt> will be thrown.
   * @param dst The list of intended recipients
   * @exception IOException Occurs if there is a problem sending the
   * message or the targeted <tt>send()</tt> method is not supported by
   * the attached <tt>MessageOutput</tt>.
   */
  public void send (String[] dst) throws IOException {
    synchronized (o) {
      o.writeUTF (label);
      byteO.writeTo (o);
      o.send (dst);
    }
    byteO.reset ();
  }
}
```

Note

- Questo stream ci offre il vantaggio (rispetto a inserire "manualmente" un'etichetta) di offrire tale funzione in modo trasparente al chiamante, che usa i normali metodi di un `MessageOutput`. Solo al momento di chiamare il costruttore si è consapevoli di aver a che fare con un `MultiplexOutputStream`.
- Il metodo `send(String[] dst)` è implementato per predisporsi al caso in cui il `MessageOutput` a cui si attacca supporti tale metodo (spedizione ***multicast***).

Costruttore

- Riceve come parametri il `MessageOutput` (naturalmente si passerà una sua classe derivata) a cui attaccarsi e l'etichetta da usare come CI.
- Per il resto è analogo agli altri stream di messaggi di output: crea un `ByteArrayOutputStream` su cui convogliare le scritture in attesa della spedizione.

Metodi

- `send()` scrive sul `MessageOutput` l'etichetta (cioè la CI) e quindi il contenuto del buffer interno (cioè il messaggio). Quindi chiama il metodo `send()` del `MessageOutput`, il che fa sì che quest'ultimo invii sul canale di comunicazione:
  - la sua propria CI (se c'è, come nel caso di `MessageOutputStream`);
  - ciò che ha nel buffer, ossia nell'ordine:
    - l'etichetta;
    - il messaggio vero e proprio;
- E' necessario assicurarsi, dento la `send()`, che le scritture sul `MessageOutput` e la chamata del metodo `send()` di quest'ultimo non possano essere interrotti, per evitare, se qualcun altro condivide il `MessageOutput`, che la costruzione e l'invio del pacchetto siano interrotti prima di essere completati. Ciò si ottiene sincronizzando tale codice sul `MessageOutput`.

2.4.2) Classe MultiplexInputStream

Questo è lo stream di input corrispondente a `MultiplexOutputStream`.

Questa classe estende `MessageInput`, si attacca a un altro `MessageInput` (non a un semplice `InputStream`) ed estrae l'etichetta da ogni messaggio che viene ricevuto.

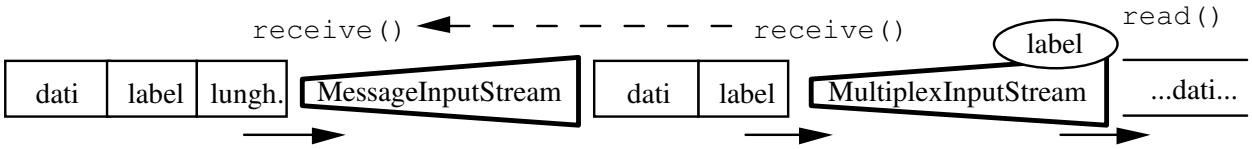

Figura 2-17: `MultiplexInputStream`

L'etichetta è resa accessibile all'esterno, rendendo così possibile determinare come deve essere elaborata l'informazione contenuta nel messaggio.

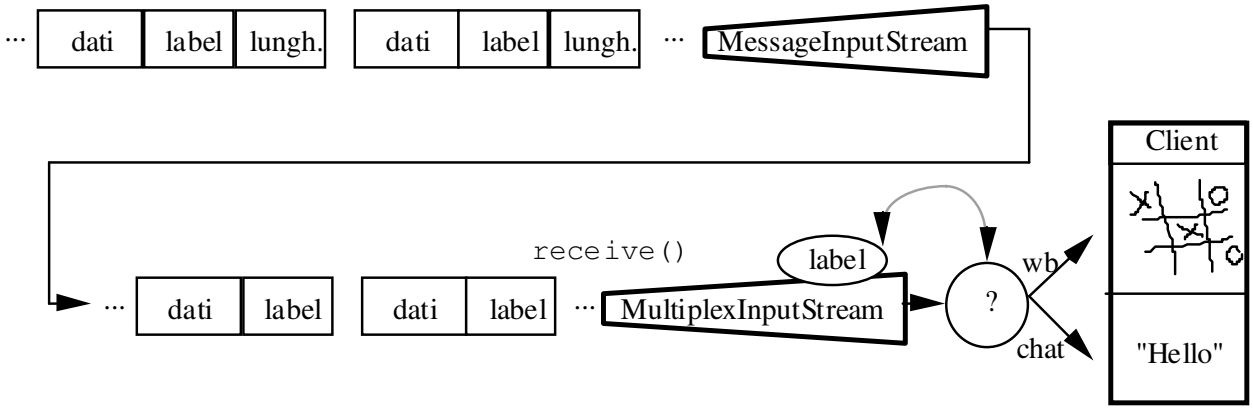

Figura 2-18: Demultiplexing dei messaggi della chatline

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.          *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;

/**
 * A <tt>MessageInput</tt> that attaches to an existing <tt>MessageInput</tt>
 * and strips the multiplexing label from each message that is received.
 * <p>The label is made publicly accessible in the <tt>label</tt> variable.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.MultiplexOutputStream
 */
public class MultiplexInputStream extends MessageInput {
  /**
   * The multiplexing label of the most recently received message.
   */
  public String label;
  /**
   * The <tt>MessageInput</tt> to which this is attached.
   */
  protected MessageInput i;

  /**
   * Creates a new <tt>MultiplexInputStream</tt>.
   * @param i The <tt>MessageInput</tt> from which messages should
   * be received
   */
  public MultiplexInputStream (MessageInput i) {
    super (i);
    this.i = i;
  }

  /**
   * Receives a new message from <tt>i</tt> and strips the multiplexing
```

```
    * label. The label is accessible in the <tt>label</tt> variable; the
    * contents of the message can be read through the usual superclass
    * <tt>read()</tt> methods.
    * @exception IOException Occurs if there is a problem receiving a
    * message or extracting the multiplexing label.
    */
  public void receive () throws IOException {
    i.receive ();
    label = i.readUTF ();
  }
}
```

Note

- Questa classe non effettua il demultiplexing. Si limita ad estrarre l'etichetta e a renderla accessibile dall'esterno. Il demultiplexing vero e proprio sarà effettuato da qualcun altro sulla base di tale informazione.

Costruttore

- Riceve come parametro il `MessageInput` (si passerà di fatto una sua sottoclasse) a cui attaccarsi.
- Chiama il costruttore della superclasse (cioè `MessageInput` e quindi `DataInputStream`) passandogli il `MessageInput` a cui si è attaccato, per cui i normali metodi di lettura della superclasse saranno direttamente passati a `i`.

Metodi

- `receive()` chiama innanzitutto il metodo `receive()` dell'`InputStream` a cui è attaccato (che provvederà a rimuovere la propria CI). Quindi provvede a leggere l'etichetta (ossia la CI di multiplexing) che viene memorizzata nella variabile pubblica `label`.
- A questo punto, i normali metodi di lettura (`read()`, ecc.) leggeranno dal corpo del messaggio.

2.4.3) Classe Demultiplexer

Questa è la classe che si incarica di effettuare il demultiplexing vero e proprio.

Riceve i messaggi da un `MultiplexInputStream` e, sulla base della loro etichetta, li consegna ad uno dei `MessageOutput` a cui è connesso.

In sostanza, è un thread copiatore che ha in più la capacità di effettuare il demultiplexing dei messaggi in arrivo. Include dei metodi per registrare i (e cancellare la registrazione dei) `MessageOutput` che devono essere associati, in qualità di destinatari, alle etichette.

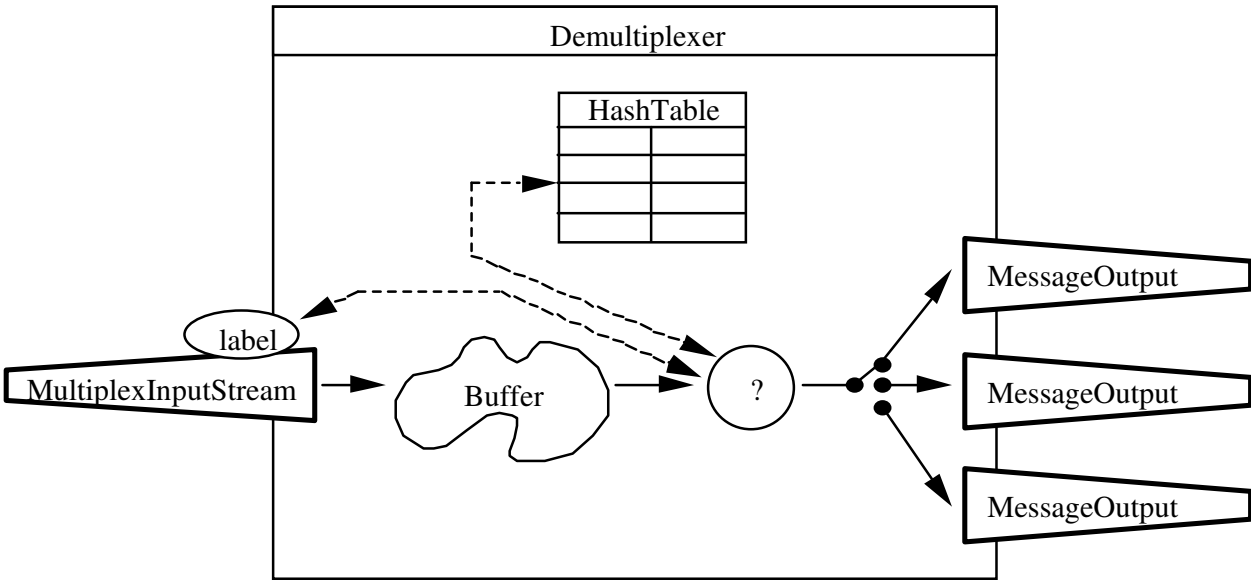

Figura 2-19: `Demultiplexer`

Tipicamente i `MessageOutput` sono dei `QueueOutputStream`, per le ragioni che abbiamo esposto in precedenza, oppure dei `DeliveryOutputStream` (che vedremo fra poco).

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.          *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;
import java.util.*;
/**
 * A class that reads messages from a <tt>MultiplexInputStream</tt>
 * and forwards them on to the <tt>MessageOutput</tt> identified by the
 * message label.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.MessageCopier
 */
public class Demultiplexer extends Thread {
  /**
   * The <tt>MultiplexInputStream</tt> from which messages are read.
   */
  protected MultiplexInputStream i;
  /**
   * The message routing table. Maps from message labels to
   * <tt>MessageOutput</tt>s.
   */
  protected Hashtable routes;

  /**
```

```
   * Current <tt>Demultiplexer</tt> ID.
   */
  static private int plexerNumber;
  /**
   * Assigns unique <tt>Demultiplexer</tt> ID's.
   * @return An unique <tt>Demultiplexer</tt> ID.
   */
  static private synchronized int nextPlexerNum () { return plexerNumber ++; }

  /**
   * Creates a new <tt>Demultiplexer</tt> reading from a specified
   * stream.
   * @param i The <tt>MultiplexInputStream</tt> from which mess. should be read
   */
  public Demultiplexer (MultiplexInputStream i) {
    super ("Demultiplexer-" + nextPlexerNum ());
    this.i = i;
    routes = new Hashtable ();
  }

  /**
   * Registers a <tt>MessageOutput</tt> as the destination for messages
   * with a particular label.
   * @param label The message label that is to be routed
   * @param o The destination for such messages
   */
  public void register (String label, MessageOutput o) {
    routes.put (label, o);
  }

  /**
   * Deregisters a particular message label.
   * @param label The label that is to be deregistered
   */
  public void deregister (String label) {
    routes.remove (label);
  }

  /**
   * Routes messages from the <tt>MultiplexInputStream</tt> to the
   * <tt>MessageOutput</tt> identified by their labels. <pre>This method
   * is called by a
   * new thread when the superclass <tt>start()</tt> method is called.
   * @see java.lang.Thread#start
   */
  public void run () {
    try {
      while (true) {
        i.receive ();
        MessageOutput o = (MessageOutput) routes.get (i.label);
        if (o != null) {
          byte[] message = new byte[i.available ()];
          i.readFully (message);
          synchronized (o) {
            o.write (message);
            o.send ();
          }
        }
      }
    } catch (IOException ex) {
      ex.printStackTrace ();
    }
  }
}
```

Costruttore

- Chiama il costruttore della superclasse, passandogli come nome una stringa concatenata con un numero progressivo (ottenuto con il metodo statico `nextPlexerNum()`).
- Crea una `HashTable` che servirà per correlare le etichette ai `MessageOutput`.
- Mantiene una reference al `MultiplexInputStream` a cui è attaccato.

Metodi

- Il metodo statico `nextPlexerNum()` restituisce il valore della variabile statica `plexerNumber` e successivamente lo incrementa di 1, fornendo così il numero progressivo per i nomi dei vari `Demultiplexer` che si volessero creare.
- Il metodo `register(...)` riceve come parametri un'etichetta e un `MessageOutput`, e inserisce un corrispondente nuovo elemento nella `HashTable`.
- Il metodo `deregister(...)` riceve come parametro un'etichetta e rimuove dalla `HashTable` il corrispondente elemento.
- Il metodo `run()` è un ciclo infinito che realizza le funzioni di:
    - thread copiatore;
    - demultiplexing dei messaggi.
- Il thread effettua nel ciclo i seguenti passi:
    - riceve un messaggio da `MultiplexInputStream`;
    - cerca nella `HashTable` l'`OutputStream` corrispondente all'etichetta del messaggio;
        - se esso esiste:
            - crea un buffer (array di byte) e lo riempie col messaggio;
            - scrive il buffer sul `MessageOutput` trovato prima e chiama `send()` di tale `MessageOutput`;
            - queste ultime due operazioni sono sincronizzate sul `MessageOutput` per evitare che il messaggio si possa mescolare con quelli di altri thread che condividono il `MessageOutput` prima di essere spedito.
        - se invece l'etichetta non è registrata nella `HashTable`, il messaggio viene scartato.

2.4.4) Classe DeliveryOutputStream e Interfaccia Recipient

Consegnare i messaggi in una coda e aspettarsi che l'applicazione cerchi attivamente di prelevarli da lì, il che richiede in genere un thread separato, non è talvolta necessario, sopratutto per una piccola applicazione.

Una alternativa è definire un diverso tipo di `MessageOutput`, che consegna i messaggi attivamente al destinatario invece di scriverli su uno stream (che nel caso sopracitato è la coda).

Ciò si può implementare facendo sì che questo nuovo tipo di `MessageOutput`, quando deve spedire un messaggio con `send()`, chiami direttamente un apposito metodo `receive()` (che deve quindi essere presente) del destinatario.

In tal modo, quando un messaggio è spedito viene immediatamente consegnato al destinatario per l'elaborazione.

La contropartita è che il thread che invia il messaggio con `send()` (tipicamente, un thread copiatore quale il `Demultiplexer`) deve aspettare il ritorno della `receive()` del destinatario per poter proseguire la propria elaborazione e ricevere quindi il prossimo messaggio.

La classe `DeliveryOutputStream` implementa un `MessageOutput` del tipo descritto.

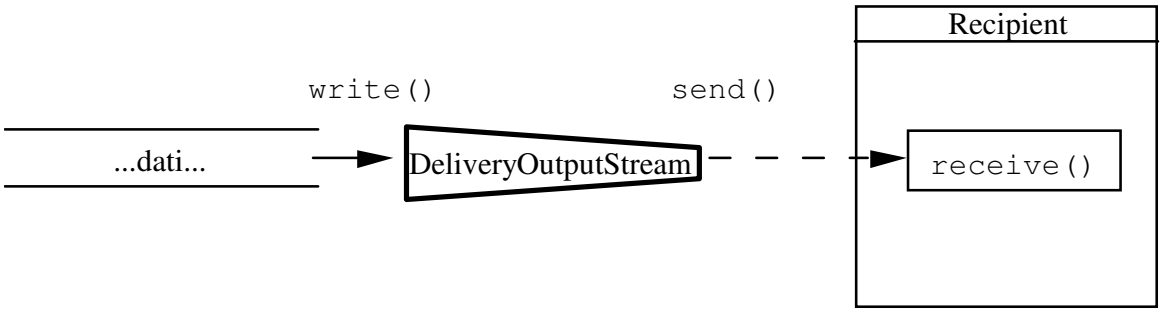

Figura 2-20: `DeliveryOutputStream`

La interfaccia `Recipient` dichiara il metodo `receive()`, e deve essere implementata da ogni destinatario che si voglia attaccare ad un `DeliveryOutputStream`.

2.4.4.1) Classe DeliveryOutputStream

E' un `MessageOutput` che si attacca a un oggetto che implementa l'interfaccia `Recipient`.

I messaggi sono costruiti dentro un `ByteArrayOutputStream` e sono consegnati al destinatario immediatamente, dopo averli spostati in un `ByteArrayInputStream` dal quale il destinatario provvederà a leggerli.

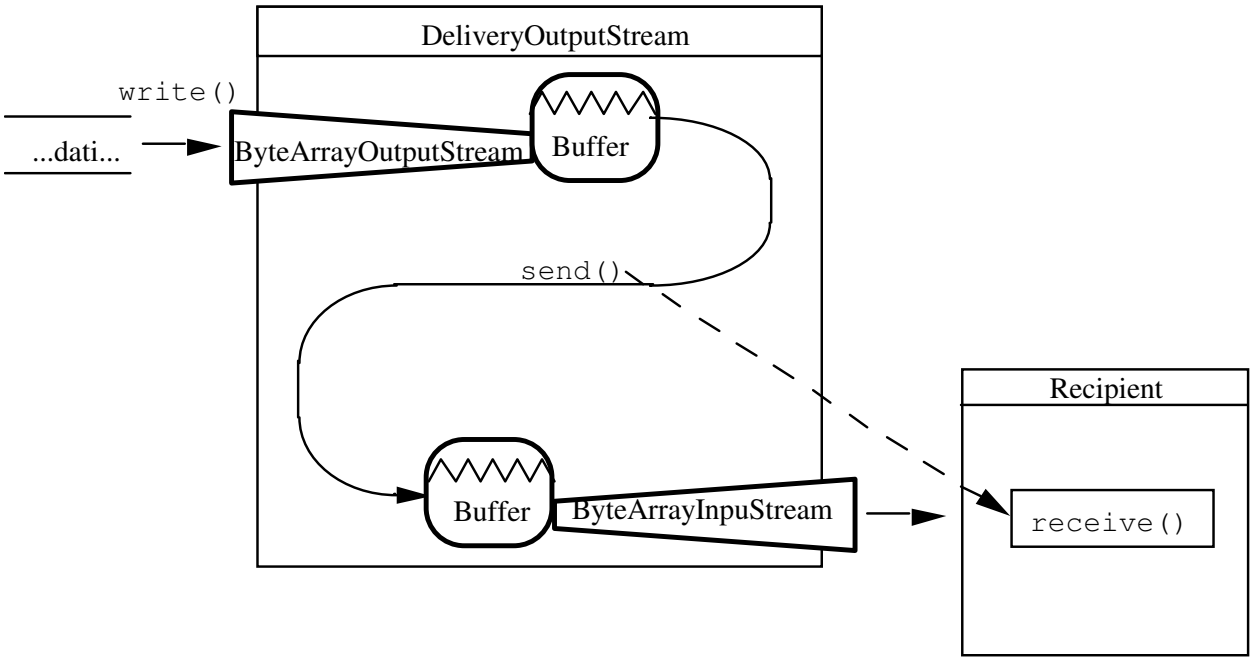

Figura 2-21: Implementazione di `DeliveryOutputStream`

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.          *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;

/**
 * A <tt>MessageOutput</tt> that immediately delivers its
 * contents to a recipient specified in the constructor when
 * send() is called.
 * <p>The message is delivered through the <tt>Recipient</tt> interface.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.Recipient
 */
public class DeliveryOutputStream extends MessageOutput {
  /**
   * A <tt>ByteArrayOutputStream</tt> used to buffer the message contents.
   */
  protected ByteArrayOutputStream byteO;
  /**
   * The recipient of messages sent to this stream.
   */
  protected Recipient r;

  /**
```

```
    * Creates a new <tt>DeliveryOutputStream</tt> with a specified recipient.
    * @param r The recipient for messages sent to this stream.
    */
  public DeliveryOutputStream (Recipient r) {
    super (new ByteArrayOutputStream ());
    byteO = (ByteArrayOutputStream) out;
    this.r = r;
  }

  /**
   * Delivers the current message contents to the designated recipient.
   */
  public void send () {
    byte buffer[] = byteO.toByteArray ();
    ByteArrayInputStream bI = new ByteArrayInputStream (buffer);
    r.receive (new DataInputStream (bI));
    byteO.reset ();
  }
}
```

Costruttore

- Chiama il costruttore della superclasse, attaccandolo a un `ByteArrayOutputStream` interno.
- Mantiene una reference in `byteO` a tale `ByteArrayOutputStream` ed un'altra, in `r`, all'oggetto che implementa l'interfaccia `Recipient`, che viene passato come parametro.

Metodi

- Il metodo `send()` effettua le seguenti operazioni:
  - crea, in un array di byte, una copia del contenuto attuale del `ByteArrayOutputStream` (cioè una copia del messaggio);
  - costruisce un `ByteArrayInputStream` attaccato alla copia del messaggio;
  - chiama il metodo `receive()` del destinatario, passandogli per convenienza un `DataInputStream` attaccato al `ByteArrayInputStream` e quindi al messaggio;
  - resetta il `ByteArrayOutputStream` per consentire la costruzione del prossimo messaggio.

2.4.4.2) Interfaccia Recipient

E' una semplice interfaccia che dichiara un solo metodo, `receive()`.

Esso ha come parametro un `DataInputStream` (che di fatto costituisce il corpo del messaggio) dal quale, con opportune letture, potrà essere acquisito il messaggio.

La definizione dell'interfaccia è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.        *
 * See the file legal.txt in the txt directory for details. */

package prominence.msg;

import java.io.*;

/**
 * The interface through which the <tt>DeliveryOutputStream</tt> delivers
 * messages.
 *
 * @version 1.0 1 Nov 1996
 * @author Merlin Hughes
 * @see prominence.msg.DeliveryOutputStream
 */
public interface Recipient {
  /**
   * Delivers a new message to the recipient.
   * @param in A <tt>DataInputStream</tt> from which the message contents can
   * be read
   */
  public void receive (DataInputStream in);
}
```

2.4.5) Client per la chatline grafica e testuale

Vediamo ora come è fatto il client per la chatline grafica e testuale; in seguito vedremo il server, che peraltro è molto semplice.

Come abbiamo già detto, il client è composto di due componenti distinte ed indipendenti, che comunicano con le loro controparti (grazie al server) per mezzo di multiplexing di messaggi.

Il client consiste di tre classi:

- un frame che contiene i componenti (`CollabTool`);
- la componente grafica (`WhiteBoard`);
- la componente testuale (`ChatBoard`).

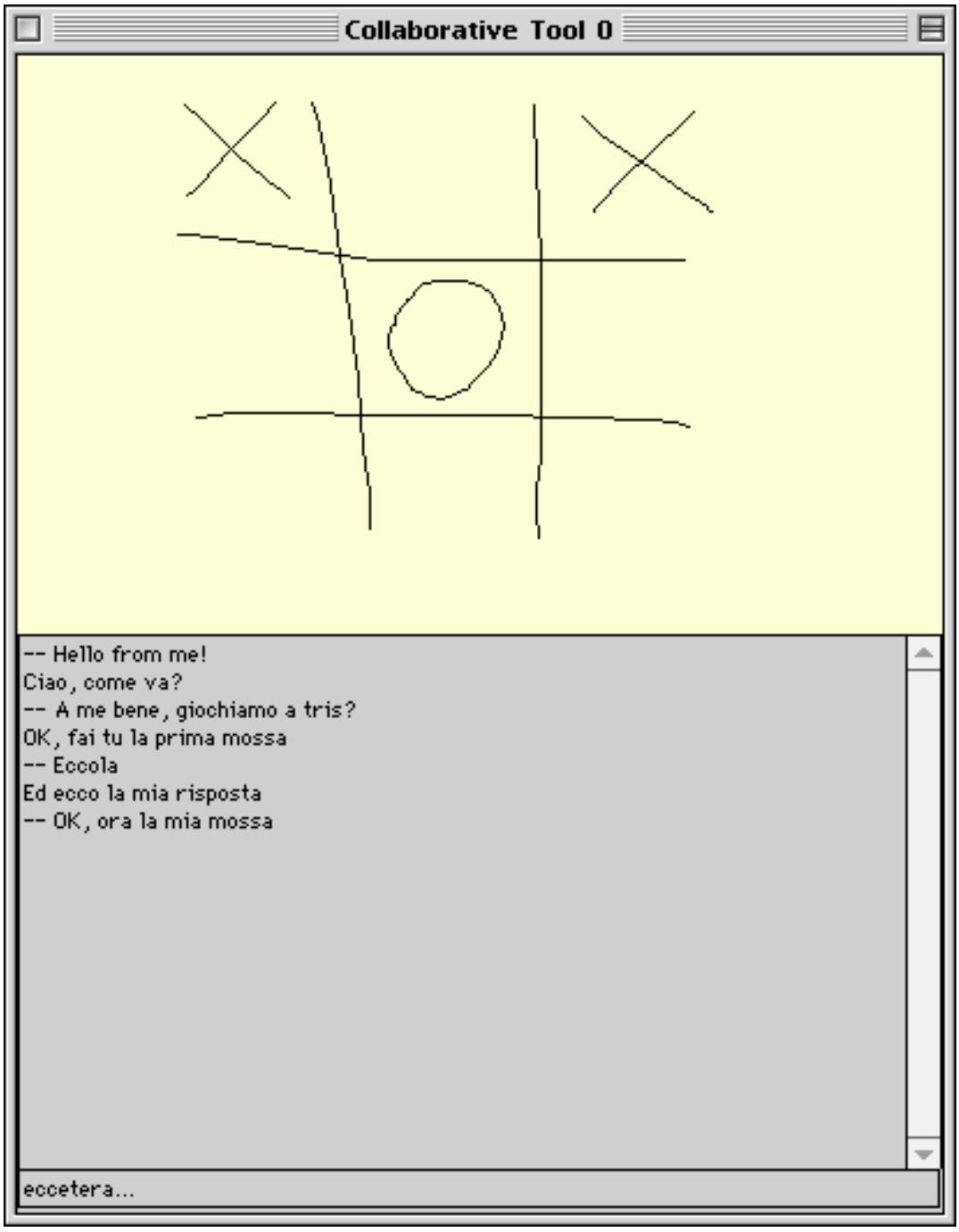

Figura 2-22: Interfaccia utente del client

2.4.5.1) Classe CollabTool

E' la classe principale (contiene il `main()`) ed ha diverse incombenze:

- aprire la connessione con il server;
- creare le due componenti (`ChatBoard` e `WhiteBoard`), inserirle in un frame e mostrare il tutto;
- predisporre tutti i necessari stream di messaggi (di input e output) per la comunicazione.

La definizione della classe è la seguente.

```
import java.io.*;
import java.awt.*;
import java.net.*;
import prominence.msg.*;

public class CollabTool extends Frame {
  protected static int id = 0;

  public CollabTool (InputStream i, OutputStream o) {
    super ("Collaborative Tool " + (id ++));
    setResizable(false);
    Whiteboard wb = new Whiteboard ();
    Chatboard cb = new Chatboard ();
    setLayout (new GridLayout (2, 1));
    add (wb);
    add (cb);
    resize (400, 500);

    MessageOutputStream mO = new MessageOutputStream (o);
    MessageInputStream mI = new MessageInputStream (i);

    cb.setMessageOutput (new MultiplexOutputStream (mO, "chat"));
    wb.setMessageOutput (new MultiplexOutputStream (mO, "wb"));

    Demultiplexer d = new Demultiplexer (new MultiplexInputStream (mI));
    d.register ("chat", cb.getMessageOutput ());
    d.register ("wb", wb.getMessageOutput ());
    d.start ();
  }

  static public void main (String args[]) throws IOException {

  Socket socket = new Socket(args[0], 5000);

    new CollabTool (socket.getInputStream(),socket.getOutputStream()).show ();
  }
}
```

Note

- Il metodo `main()` apre la connessione di rete con il server (il cui indirizzo IP o nome DNS deve essere passato come parametro) sul port 5000.
- Quando la connessione di rete è stabilita, da essa si derivano i due stream per la comunicazione che vengono passati al costruttore di `CollabTool`.

Costruttore

- Chiama il costruttore di `Frame` passandogli un titolo per la finestra.
- Crea `cb` (un `ChatBoard`) e `wb` (un `WhiteBoard`) e li aggiunge al `Frame`;
- Crea due message stream attaccati ai due stream esistenti da/verso il server:
  - `mO` è un `MessageOutputStream`;
  - `mI` è un `MessageInptStream`.
- Chiamando il metodo `setMessageOutput (...)` di `wb` e `cb`, aggancia il loro output ad un corrispondente `MultiplexOutputStream`.
- A questo punto l'output delle due componenti è sistemato, ora si deve occupare dell'input. Innanzitutto fa queste operazioni:
  - crea un `MultiplexInputStream` agganciato a `mI`;
  - crea un `Demultiplexer` agganciato a tale `MultiplexInputStream`.
  - Registra nel `Demultiplexer` gli stream di output ai quali dovranno essere inviati i messaggi di competenza di ognuna delle due componenti. Questo si fa chiedendo a loro quale dev'essere lo stream di output, mediante il loro metodo `getMessageOutput()` che, appunto, restituisce il `MessageOutput` che ognuna delle due componenti avrà provveduto a creare secondo le proprie inclinazioni.
- Infine, avvia il thread "copiatore" del `Demultiplexer`.

2.4.5.2) Classe ChatBoard

Questa classe implementa la componente testuale della chatline.

E' costituita da una `TextArea` in cui appaiono i messaggi precedenti e da un `TextField` in cui si immette di volta in volta un nuovo messaggio. Esso viene spedito non appena si preme il tasto Return dentro il `TextField`.

Nel contesto del client, questa componente dialoga direttamente con le corrispondenti componenti degli altri client collegati, ed ignora completamente l'esistenza della componente grafica.

Implementa l'interfaccia `Runnable` perché un thread gestisce l'interazione con l'utente (quello principale) mentre un thread separato (avviato con `run()`) si occupa di ricevere i messaggi che arrivano dalla connessione di rete.

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.          *
 * See the file legal.txt in the txt directory for details. */

import java.io.*;
import java.awt.*;
import prominence.msg.*;
import prominence.util.Queue;

public class Chatboard extends Panel implements Runnable {
  protected TextArea output;
  protected TextField input;
  protected Queue q;
  protected Thread exec;

  public Chatboard () {
    setLayout (new BorderLayout ());
    add ("Center", output = new TextArea ());
    output.setEditable (false);
    add ("South", input = new TextField ());
    q = new Queue ();
    exec = new Thread (this);
    exec.start ();
  }

  protected MessageOutput o;

  public void setMessageOutput (MessageOutput o) {
    this.o = o;
  }

  public MessageOutput getMessageOutput () {
    return new QueueOutputStream (q);
  }

  public boolean action (Event e, Object arg) {
    if (e.target == input) {
      try {
        o.writeUTF (input.getText ());
        o.send ();
      } catch (IOException ex) {
        ex.printStackTrace ();
      }
      output.appendText (input.getText () + "\n");
      input.setText ("");
      return true;
    }
    return super.action (e, arg);
  }

  public void sendMessage (String s) {
      try {
        o.writeUTF (s);
        o.send ();
      } catch (IOException ex) {
        ex.printStackTrace ();
      }
  }
```

```
  public void run () {
    QueueInputStream qI = new QueueInputStream (q);
    while (true) {
      try {
        qI.receive ();
        String msg = qI.readUTF ();
        output.appendText ("-- " + msg + "\n");
      } catch (IOException ex) {
        ex.printStackTrace ();
      }
    }
  }
}
```

Costruttore

- Crea i componenti dell'interfaccia utente e li dispone opportunamente.
- Crea una coda `q` a cui verranno attaccati i 2 stream per la comunicazione col `Demultiplexer`:
  - lo stream di output attaccato alla coda serve al `Demultiplexer` per depositarvi i messaggi arrivati;
  - lo stream di intput attaccato alla coda serve al `Chatboard` per prelevare (e quindi leggere) i messaggi arrivati.
- Avvia il thread separato per lettura dei messaggi in arrivo.

Metodi

- I metodi `setMessageOutput(MessageOutput o)` e `getMessageOutput()`, che vengono chiamati da `ColabTool` in fase di creazione della `ChatBoard`, predispongono gli stream di input e output. In particolare:
  - `setMessageOutput(MessageOutput o)` si aggancia al `MultiplexOutputStream`, creato da `CollabTool`, che serve in output;
  - `getMessageOutput()` invece crea un `QueueOutputStream` attaccato a `q` e lo passa a `CollabTool`, il quale così sa dove inviare i dati prodotti dal `Demultiplexer` per il `ChatBoard`.
- Il metodo `run()`, che fa partire il thread separato di lettura dei dati in arrivo, prima di tutto crea un `QueueInputStream` attaccato a `q`, e poi entra in un ciclo infinito di estrazione dei messaggi da tale stream.
- Il metodo `action(...)` gestisce l'evento di tipo `action` nel solo `TextField`, inviando un nuovo messaggio; rimanda alla superclasse la gestione di altri tipi di eventi.

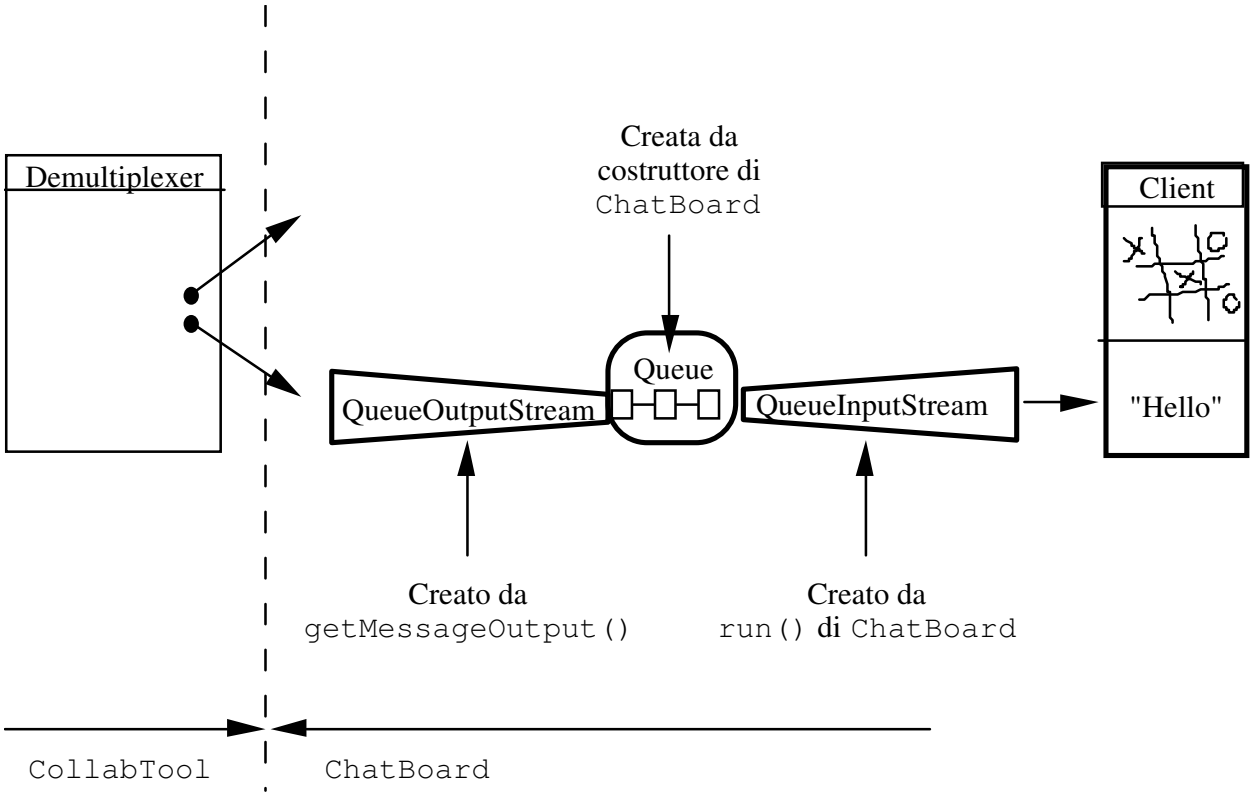

Figura 2-23: Collegamento degli stream per l'input di `ChatBoard`

2.4.5.3) Classe WhiteBoard

Questa classe implementa la componente grafica della chatline.

Offre una superficie sulla quale l'utente può disegnare a mano libera. Quando l'utente:

- preme il pulsante del mouse: inizia una nuova porzione del disegno;
- sposta il mouse (a bottone premuto, ***drag***): la porzione del disegno viene man mano costruita;
- rilascia il mouse: la porzione di disegno appena completata viene spedita sotto forma di messaggio.

Nel contesto del client, anche questa componente dialoga con le corrispondenti componenti degli altri client ed ignora l'esistenza della componente `ChatBoard`.

Implementa l'interfaccia `Recipient`, e quindi implementa un metodo `receive()` per la ricezione immediata dei messaggi.

La definizione della classe è la seguente (dal libro "Java Network Programming" di Merlin Hughes et al., Mannig Publications Co, Greenwich CT, 1997).

```
/* Copyright (c) 1996, 1997 Prominence Dot Com, Inc.          *
 * See the file legal.txt in the txt directory for details. */

import java.io.*;
import java.awt.*;
import prominence.msg.*;

public class Whiteboard extends Canvas implements Recipient {
  public Whiteboard () {
    setBackground (new Color (255, 255, 204));
  }

  protected MessageOutput o;

  public void setMessageOutput (MessageOutput o) {
    this.o = o;
  }

  public boolean mouseDown (Event e, int x, int y) {
    transmit (x, y);
    return super.mouseDown (e, x, y);
  }

  public boolean mouseDrag (Event e, int x, int y) {
    scribble (x, y);
    transmit (x, y);
    return super.mouseDrag (e, x, y);
  }

  public boolean mouseUp (Event e, int x, int y) {
    scribble (x, y);
    transmit (x, y);
    try {
      o.send ();
    } catch (IOException ex) {
      ex.printStackTrace ();
    }
    return super.mouseUp (e, x, y);
  }

  protected int oX, oY;

  protected void transmit (int x, int y) {
    try {
      o.writeInt (x);
      o.writeInt (y);
    } catch (IOException ex) {
      ex.printStackTrace ();
    }
    oX = x;
    oY = y;
  }

  protected void scribble (int x, int y) {
    Graphics g = getGraphics ();
    g.drawLine (oX, oY, x, y);
    g.dispose ();
  }
```

```
  public MessageOutput getMessageOutput () {
    return new DeliveryOutputStream (this);
  }

  public void receive (DataInputStream dI) {
    Graphics g = getGraphics ();
    try {
      int x0 = dI.readInt (), y0 = dI.readInt ();
      while (dI.available () > 0) {
        int x1 = dI.readInt (), y1 = dI.readInt ();
        g.drawLine (x0, y0, x1, y1);
        x0 = x1;
        y0 = y1;
      }
    } catch (IOException ex) {
      ex.printStackTrace ();
    }
    g.dispose ();
  }
}
```

Costruttore

- `WhiteBoard` è costituito da un unico `Canvas` (area di disegno), istanziato automaticamente dato che `WhiteBoard` estende proprio `Canvas`. Il costruttore si limita a cambiare il colore di fondo.

Metodi

- Come in `ChatBoard`, i metodi `setMessageOutput(MessageOutput o)` e `getMessageOutput()`, che vengono chiamati da `CollabTool` in fase di creazione della `WhiteBoard`, predispongono gli stream di I/O. In particolare:
  - `setMessageOutput(MessageOutput o)` si aggancia al `MultiplexOutputStream`, creato da `CollabTool`, che serve in output;
  - `getMessageOutput()` invece crea un `DeliveryOutputStream`, avente come target il `WhiteBoard` stesso, e lo passa a `CollabTool`, il quale così sa dove inviare i dati prodotti dal `Demultiplexer` e destinati al `WhiteBoard`.
- Il metodo `scribble()` disegna un trattino dall'ultima posizione del mouse (`ox, oy`) a quella corrente (`x, y`), e per far questo crea un contesto grafico che poi elimina.
- Il metodo `transmit()` scrive nel corpo del messaggio corrente le coordinate (`x, y`) attuali, che così si aggiungono a quelle precedenti; aggiorna quindi le coordinate (`ox, oy`) con i valori correnti.
- La gestione degli eventi si articola su tre metodi:
  - `mouseDown(...)` inizia a costruire il prossimo messaggio, chiamando `transmit(...);`
  - `mouseDrag(...)` mostra l'ultimo tratto disegnato (con `scribble(...)`) e poi inserisce sul messaggio le nuove coordinate finali, chiamando `transmit(...);`

- `mouseUp(...)` disegna l'ultimo tratto (con `scribble(...)`), completa il messaggio (con `transmit(...)`) e poi invia effettivamente il messaggio con il metodo `send()` del `MultiplexOutputStream` a cui è attaccato.

- Il metodo `receive()` legge il contenuto di un messaggio (cioè una successione di coppie (`x, y`)) e provvede ad effettuare il corrispondente disegno.

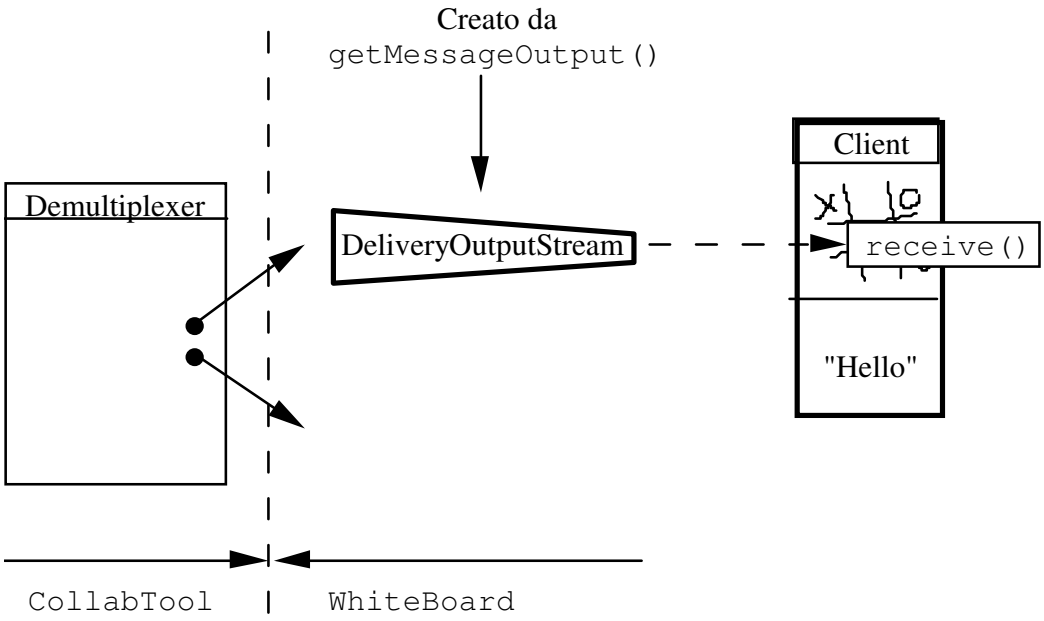

Figura 2-24: Collegamento degli stream per l'input di `WhiteBoard`

2.4.6) Server per la chatline grafica e testuale

E' il server a cui tutti i client si connettono. Invia in broadcast a tutti (tranne che al mittente) tutto ciò che riceve da uno qualunque di essi.

Concettualmente (e anche strutturalmente) è identico al server dell'esempio 9, solo che comunica attraverso l'uso di messaggi.

Si noti che i messaggi sono a loro volta pacchetti composti da:
- etichetta;
- dati.

Questo però il server lo ignora, e non ne preoccupa. Quindi, tale server continuerà a funzionare correttamente anche anche nel caso in cui ai client si dovessero aggiungere

ultreriori componenti funzionali, purché anch'esse comunichino fra loro facendo uso di stream per il multiplexing dei messaggi.

Il codice è costituito da due classi. La prima, `CollabServer`, accetta richieste di connessione sul port 5000 e, ogni volta che ne arriva una, istanzia un oggetto della classe `CollabHandler` che si occupa di gestirla.

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

public class CollabServer {

//-------------------------------------------------

  public CollabServer() throws IOException {

    ServerSocket server = new ServerSocket(5000);
    System.out.println ("Accepting connections...");
    while(true) {
      Socket client = server.accept();
      System.out.println ("Accepted from " + client.getInetAddress());
      new CollabHandler(client).start();
    }
  }

//-------------------------------------------------

  public static void main(String args[]) throws IOException {

    new CollabServer();
  }

}
```

La seconda si occupa della gestione di una singola connessione e dell'invio a tutte le altre, in broadcast, dei dati provenienti da tale connessione.

```
import java.net.*;
import java.io.*;
import java.util.*;
import prominence.msg.*;

public class CollabHandler extends Thread {

  protected static Vector handlers = new Vector();
  protected Socket socket;
  protected MessageInputStream is;
  protected MessageOutputStream os;

//-------------------------------------------------

  public CollabHandler(Socket socket) throws IOException {
```

```
    this.socket = socket;
    is = new MessageInputStream(socket.getInputStream());
    os = new MessageOutputStream(socket.getOutputStream());
  }

//-------------------------------------------------

  public void run() {

    try {
      handlers.addElement(this); //e' un metodo sincronizzato
      while (true) {
        is.receive();
        byte[] buffer = new byte[is.available()];
        is.readFully(buffer);
        broadcast(this, buffer);
      }
    } catch(IOException ex) {
      ex.printStackTrace();
    } finally {
      handlers.removeElement(this); //e' un metodo sincronizzato
      try {
        socket.close();
      } catch(Exception ex) {
        ex.printStackTrace();
      }
    }
  }

//-------------------------------------------------

  protected static void broadcast(CollabHandler sender, byte[] buffer) {

    synchronized (handlers) { //ora nessuno puo' aggiungersi o abbandonare
      Enumeration e = handlers.elements();
      while (e.hasMoreElements()) {
        CollabHandler c = (CollabHandler) e.nextElement();
        if (c != sender) {
          try {
            c.os.write(buffer);
            c.os.send();
          } catch(Exception ex) {
            c.stop();
          }
        }
      }
    }
  }
}
```

Note

- `CollabTool` avvia mediante il `main()` il thread principale che:
  - rimane in ascolto sul port 5000;
  - attiva un thread separato (`CollabHandler`) per la gestione di ogni nuova connessione.
- Ogni nuovo `CollabHandler` attacca due message stream ai due canali di comunicazione, e quando si avvia:
  - si registra nel `Vector` statico delle connessioni attive;

    - riceve messaggi in input e li ritrasmette col metodo `broadcast();`
    - quando termina, si rimuove dal `Vector` delle connessioni.
- Il metodo `broadcast()` di `CollabHandler` si sincronizza sul `Vector` delle connessioni e spedisce a tutti, tranne che a se stesso, il messaggio passato come parametro.

2.5) Ulteriori estensioni di funzionalità tramite messaggi

Abbiamo visto come l'uso e la concatenazione di message stream costituisca un potente e versatile meccanismo per lo sviluppo di applicazioni di rete.

Con tali conoscenze in mente, è relativamente semplice procedere ad ulteriori estensioni di funzionalità, quali ad esempio una trasmissione di tipo ***multicast***, ossia una trasmissione nella quale si specifica il sottoinsieme dei destinatari che deve ricevere il messaggio.

Tale funzionalità può essere ottenuta con una classe `RoutingOutputStream`, che avrà il compito di incapsulare ogni messaggio dotandolo di una CI costituita da un vettore di stringhe che specificano ciascuna un destinatario.

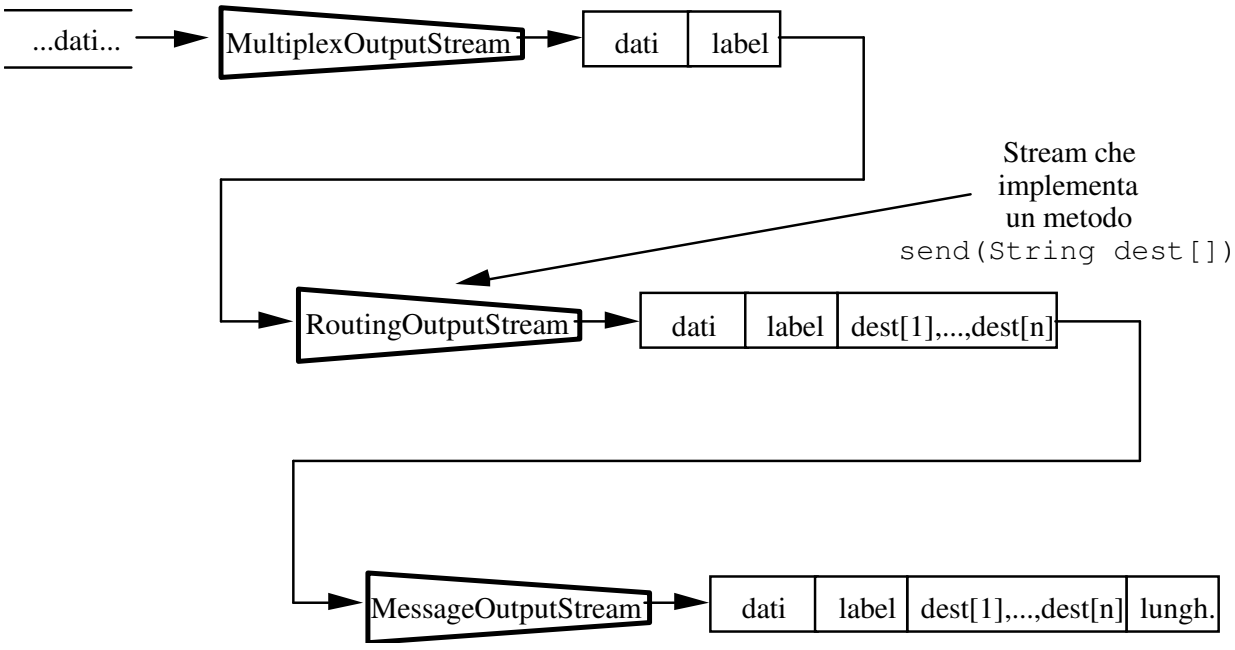

Figura 2-25: Uso di `RoutingOutputStream` per trasmissioni multicast,

All'altra estremità, tipicamente sul server, a valle di un `MessageInputStream` si potrà predisporre una corrispondente classe `RoutingInputStream` che estrae la lista di destinatari e

la rende disponibile all'esterno, e una classe `Router` (che ha la funzione di thread copiatore) incaricata di inviare una copia di ogni messaggio solamente ai corrispondenti destinatari.

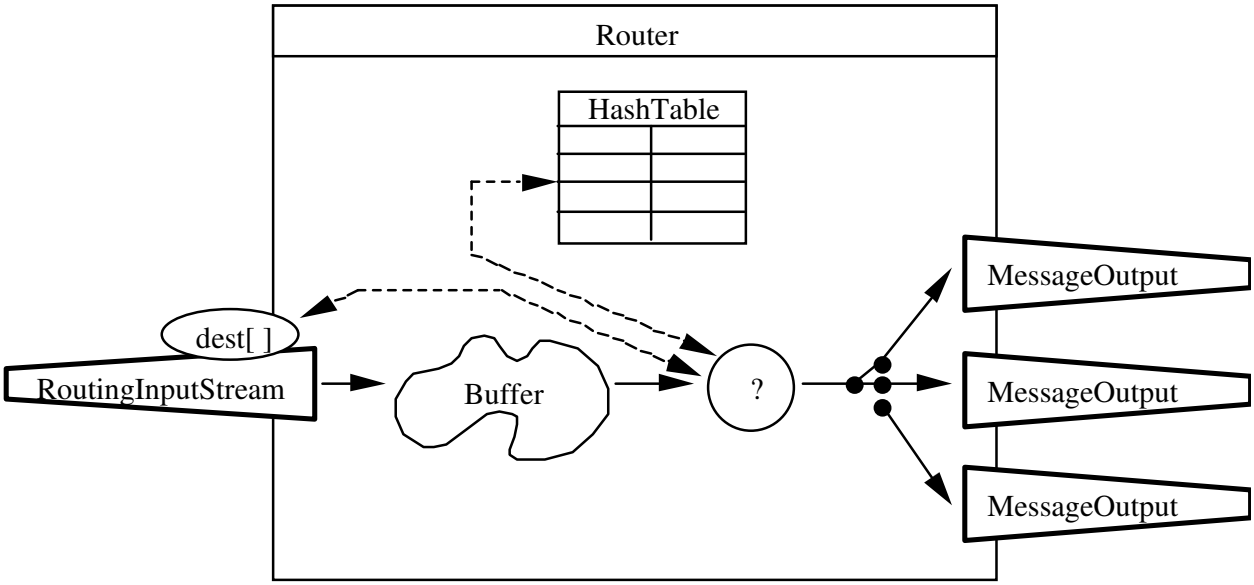

Figura 2-26: `Router`

Con riferimento alla specifica concatenazione illustrata nella figura 2-25, il messaggio che viene instradato dal `Router` è in realtà un pacchetto contenente la CI del `MultiplexOutputStream`, e cioè l'etichetta della componente applicativa che deve ricevere i dati.

Di conseguenza, a valle di ogni `MessageOutput` in uscita dal `Router` ci potrà essere un `MessageInput`, quindi un `MultiplexInputStream` ed infine un `Demultiplexer` che esamina tale etichetta e smista il messaggio alla corretta componente applicativa.

In tal modo è possibile costruire applicazioni di tipo client-server costituite da componenti indipendenti capaci di effettuare trasmissioni multicast, il che consente di soddisfare praticamente qualunque esigenza applicativa.

Torna all'indice | Vai avanti