Progetto
Laboratorio Programmazione di rete
Mini-KaZaA

Andrea Di Grazia, Massimiliano Giovine

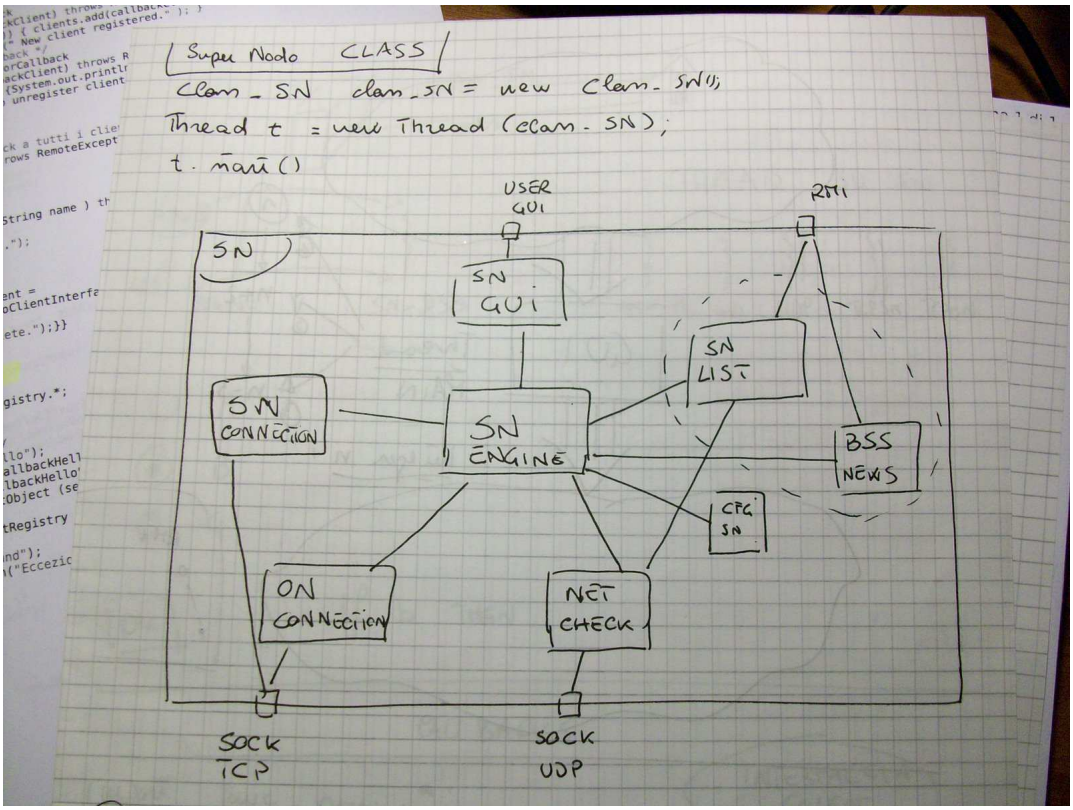
Anno Accademico 2008 - 2009

Indice

Capitolo 1

Introduzione

E tutto cominciò così .....



1.1 Una panoramica generale.

Il progetto Mini-KaZaA mira allo sviluppo di un sistema p2p per lo scambio di file su WAN, ispirato alla più famosa rete p2p KaZaA. Ogni peer[1] partecipante alla rete Mini-KaZaA condivide un insieme di files con gli altri peer connessi e può ricercare file all'interno della rete e effettuarne il download.

Mini-KaZaA prevede due tipi diversi di peer:

- **Super Nodes (SN):** i SN hanno il compito di gestire le comunicazioni all'interno della rete;
- **Ordinary Nodes (ON):** gli ON hanno responsabilità più limitate, condividono e cercano file nella rete.

Nella rete Mini-KaZaA è prevista anche un'altra entità chiamata **Bootstrap Servers** che contiene la lista di tutti i peer connessi alla rete e dalla quale ogni nodo che desidera entrare a far parte della rete può scaricare la lista aggiornata di tutti i SN presenti.

La rete si costruisce automaticamente dai vari peer secondo un preciso schema e si mantine stabile grazie a processi automatizzati che lavorano in background, completamente trasparenti all'utente.

1.2 La rete Mini-KaZaA.

Ogni peer della rete Mini-KaZaA viene configurato esplicitamente dall'utente al primo avvio come SN o come ON. Successivamente non sarà possibile cambiare tale configurazione.

Al momento della connessione alla rete ogni peer, SN o ON, contatta un Bootstrap server che gli fornisce la lista aggiornata di SN presenti in quel momento all'interno della rete.

Un ON sceglie il migliore SN per lui e si connette ad esso. Un SN mantiene in memoria la lista di riferimenti a SN che gli servirà, in un secondo momento, per smistare le interrogazioni. Gli SN, inoltre, avendo un sistema dinamico di connessione ai pari SN esplorano a ogni interrogazione porzioni nuove della rete in modo tale che vi siano il meno possibile porzioni isolate della rete.

[1] Ogni nodo della rete è un pari all'interno del network poichè funziona sia da client, per ciò che concerne la ricerca e il download dei file, sia da server per la condivisione dei file o lo smistamento delle ricerche nella rete p2p.

Capitolo 2

Bootstrap Server

2.1 Il Bootstrap server in generale

Il Bootstrap server ha il compito di tenere un indice di tutti i SN presenti nella rete che abbiano una certa affidabilità. Per poter fare questo fornisce un servizio di RMI[1] tramite il quale i SN si possono iscrivere alla rete Mini-KaZaA e richiedere liste aggiornate.

Gli aggiornamenti vengono spediti a ogni SN presente nella lista del Bootstrap server tramite un sistema di *callbacks*[2]

Il Bootstrap server deve fornire questo servizio anche agli ON che vogliono entrare nella rete per poter individuare il "miglior"[3] SN al quale potersi connettere. Per questa ragione si è reso necessario indicizzare anche gli ordinary node all'interno del bootstrap server.

2.2 Entriamo nel dettaglio

Il bootstrap server si avvia dal programma principale situato all'interno del file `BootstrapService.java` e subito crea il servizio RMI sulla porta 2008 da mettere a disposizione per i vari nodi della rete con le seguenti istruzioni:

```
1  Registry registry = LocateRegistry.createRegistry(2008);
2  BootStrapServer bss = new BootStrapServer(g,sn_list);
3
4  BootStrapServerInterface stub =
5    (BootStrapServerInterface)
```

[1] Remote Method Invocation, tramite questo servizio è possibile invocare metodi che si trovano su una macchina diversa da quella in cui si trova la chiamata a procedura.

[2] Ogni nodo mette a disposizione del Bootstrap server alcune chiamate di procedura che, richiamate, consentono di inviare aggiornamenti.

[3] Spiegeremo nella sezione 4.2.4 i parametri secondo i quali ogni ON sceglie un SN al quale connettersi.

```
6     UnicastRemoteObject.exportObject(bss, 2008);
7
8
9   SupernodeCallbacksImpl client_impl =
10    new SupernodeCallbacksImpl(
11    new SupernodeList(),
12    new NodeConfig());
13
14  SupernodeCallbacksInterface client_stub =
15    (SupernodeCallbacksInterface)
16    UnicastRemoteObject.exportObject( client_impl,2008);
```

Con le prime istruzioni il Bootstrap server mette a disposizione tutti i metodi della classe `BootStrapServerInterface` che sono i seguenti:

```
1   public boolean
2     addSuperNode(NodeInfo new_node) throws RemoteException;
3
4   public boolean
5     removeSuperNode(NodeInfo new_node) throws RemoteException;
6
7   public boolean
8     addOrdinaryNode(NodeInfo new_node) throws RemoteException;
9
10  public boolean
11    removeOrdinaryNode(NodeInfo new_node) throws RemoteException;
12
13  public ArrayList<NodeInfo>
14    getSuperNodeList() throws RemoteException;
```

Con le istruzioni alla riga 9 e 14 registra l'interfaccia di callback, definita nel package del Super Node, `SupernodeCallbacksInterface` che ha i seguenti metodi:

```
1   public void
2     notifyMeAdd(NodeInfo new_node) throws RemoteException;
3
4   public void
5     notifyMeRemove(NodeInfo new_node) throws RemoteException;
```

Per poter trasmettere e ricevere le informazioni riguardanti i vari nodi della rete, il client Mini-KaZaA e il Bootstrap server usano una classe serializzabile che si chiama `NodeInfo` che analizzamo nella sezione 2.3

2.3 La classe NodeInfo

La classe `NodeInfo` si trova nel package `lpr.minikazaa.bootstrap` ma viene utilizzata da tutto il pacchetto Mini-KaZaA per poter inviare nella rete le informazioni relative ai nodi.

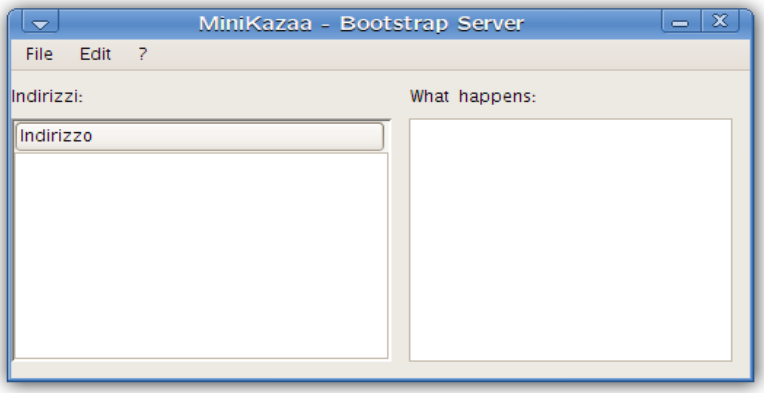


Figura 2.1: Interfaccia grafica del Bootstrap server

Questa classe rappresenta le informazioni utili di un nodo con le sue variabili private.

```
1 private InetAddress ia_node;
2 private int door;
3 private String id_node;
4 private String username;
5 private SupernodeCallbacksInterface stub;
6 private long ping;
7 private boolean is_sn;
```

Le prime tre variabili private riguardano tutte le informazioni di rete dei nodi, ovvero l'indirizzo IP, la porta di connessione e un *id* [4] ottenuto facendo la concatenazione della rappresentazione decimale dell'indirizzo IP.

La variabile privata `private SupernodeCallbacksInterface stub` rappresenta l'interfaccia per le callbacks che viene messa a disposizione dal nodo. In questo modo il bootstrap server può richiamare direttamente l'interfaccia delle callback di ogni nodo interessato all'aggionrnamento.

L'ultima variabile, `private boolean is_sn` indica se il nodo al quale si riferiscono le informazioni, all'interno della rete ricopre il ruolo di SN o di ON.

La classe contiene tutti i metodi `set` e `get` per poter assegnare valori alle variabili private e per poterne ricavare il contenuto in qualsiasi momento.

2.4 L'interfaccia grafica.

L'interfaccia grafica fornisce le informazioni riguardo a ciò che avviene all'interno della rete.

[4] codice identificativo univoco

Uno screenshot dell'interfaccia grafica principale del Bootstrap server si può vedere in Figura 2.1.

Nella parte a sinistra dell'interfaccia vengono inseriti gli *id* di tutti i nodi che si connettono alla rete.

Nella parte destra, invece, vengono visualizzati dei messaggi che spiegano cosa avviene all'iterno della rete.

Capitolo 3

Mini-KaZaA Client

Oltre che di un Bootstrap server, la rete Mini-KaZaA si basa su un client che gli utenti possono usare per accedere alla rete e poter condividere e scaricare file.

3.1 Mini-KaZaA Client in generale

Mini-KaZaA client presenta tutte le funzionalità che consentono una condivisione peer to peer dei contenuti. Ogni client al primo avvio chiede all'utente, tramite un comodo pannello, di scegliere il *ruolo* da interpretare all'interno della rete.

Chi ha più risorse da mettere a disposizione e una banda di comunicazione più ampia può scegliere di essere un Super Node, il quale oltre a condividere e scaricare, ha la funzione di smistare le query nella rete e accettare richieste direttamente dagli Ordinary Node *figli*. Chi ha meno risorse da mettere a disposizione può scegliere di essere un semplice Ordinary Node.

3.2 Il codice di Mini-KaZaA client

Il codice di Mini-KaZaA client è distribuito in tre diverse librerie:

- **lpr.minikazaa.minikazaaclient**: questa libreria contiene classi comuni a tutti e due i tipi di client dal punto di vista logico. L'esempio più evidente è la classe `MainGui.java`.
- **lpr.minikazaa.minikazaaclient.ordinarynode**: questa libreria contiene le classi che logicamente appartengono al tipo di client Ordinary Node, ma che, all'occorrenza, possono essere importate anche da un Super Node.
- **lpr.minikazaa.minikazaaclent.supernode**: questa libreria, infine contiene tutte le classi che servono ad un supernodo per funziona-

re e che appartengono a questo logicamente. Alcune di queste classi, come per esempio `SupernodeCallbacksInterface.java`, vengono utilizzate anche dagli Ordinary Node.

Questa suddivisione è puramente logica visto che i due tipi di client differiscono solo per alcune caratteristiche.

Si è preferito dividere anche le classi che contengono gli stessi task per i SN e per gli ON per poter meglio gestire il codice e renderlo più modulare. Un esempio è rappresentato dalle classi `OrdinarynodeWorkingThread.java` e `SupernodeWorkingThread.java` che hanno lo stesso compito, ma, che piuttosto che complicare con una serie di

```
if <condizione> then
<blocco>
else
<blocco>|
```

si è preferito separare in due classi distinte.

Passiamo ora a una presentazione più dettagliata del codice comune a Super Node e Ordinary Node.

3.3 Le strutture dati comuni

Per lo sviluppo di Mini-KaZaA è stato necessario predisporre una serie di strutture dati che tutto il software utilizzi per condividere informazioni.

All'interno del package `lpr.minikazaa.minikazaaclient` troviamo le seguenti classi che rappresentano strutture dati comuni a SN e ON:

- `NodeConfig.java`
- `Query.java`
- `Answer.java`
- `SearchField.java`
- `Download.java`
- `DownloadRequest.java`
- `DownloadResponse.java`

Guardiamo cosa si nasconde all'interno di ognuna delle classi sopracitate.

3.3.1 NodeConfig.java

La classe `NodeConfig.java` contiene i seguenti attributi:

```
1  private String user_name;
2  private int port;
3  private String bootstrap_address;
4  private int max_conn;
5  private int ttl;
6  private boolean is_sn;
7
8  //Calcolato all'avvio
9  private String my_address;
```

Questi attributi sono i campi che l'utente inserisce nel form al primo avvio del programma e contengono le informazioni di configurazione del nodo.

3.3.2 Query.java

La classe `Query.java` viene utilizzata dal client Mini-KaZaA per l'invio di richieste di file nella rete.

Contiene diversi attributi per i quali sono disponibili i metodi `set` e `get`. Questa classe inoltre implementa le interfacce `Serializable` e `Cloneable`. La prima serve per poter inviare su rete come flusso di byte l'oggetto `Query`. La seconda invece serve per poter copiare un'istanza dell'oggetto `Query` in una seconda istanza.

```
1  //Espressione regolare della query
2  private String body_q;
3
4  //Query di risposta
5  private Answer body_a;
6
7  //Sorgente di una query
8  private NodeInfo id_origin;
9
10 //NodeInfo del mittente
11 private NodeInfo sender;
12
13 //NodeInfo del destinatario
14 private NodeInfo receiver;
15
16 //Time to live della query
17 private int ttl;
18
19 //Id della query attribuito dall'origine
20 private int id;
```

La classe `Query` ha tre gruppi di attributi.Un primo gruppo descrive il contenuto della query e di conseguenza il tipo di query. Un secondo gruppo

serve per identificare i soggetti coinvolti nello scambio della query stessa. Il terzo gruppo contiene invece parametri per l'identificazione della query.

Analizziamo uno ad uno questi parametri per capire meglio come funzionano le query in Mini-KaZaA.

- `body_q`: il vero corpo della query di richiesta di un file. È una stringa che contiene un espressione regolare che il client Mini-KaZaA riesce a interpretare;
- `body_a`: la parte dell'oggetto `Query` che contiene la risposta a una determinata richiesta. Analizzaremo la classe `Answer` successivamente;
- `id_origin`: per ogni query deve essere nota l'origine dalla quale proviene la query stessa per poi poterla correttamente fermare al punto giusto e farla ritornare al mittente. Questo è il compito del campo `id_origin`;
- `sender`: questo campo indica uno dei due soggetti che sono impegnati in un singolo scambio di query, il nodo da cui parte;
- `receiver`: questo campo indica il nodo a cui deve arrivare la query in uno scambio;
- `ttl`: questo campo sta per *Time To Live* e indica il numero di scambi per il quale la query deve continuare a esistere. Serve principalmente per evitare che si creino dei cicli infiniti di scambio della query ottenendo quindi una valanga di dati ridondanti con conseguente intasamento della rete;
- `id`: ogni nodo può inviare più query alla volta nella rete e il compito di questo campo è di identificare univocamente la query presso il suo nodo origine.

3.3.3 Answer.java

La classe `Answer.java` contiene i file che possono corrispondere ai criteri di una ricerca.

È una classe molto semplice ma molto utile per indicizzare rapidamente i file.

Ecco il codice nel quale vengono dichiarati gli attributi della classe.

```
1 //File che corrispondono a una query
2 private ArrayList <OrdinarynodeFiles> files;
3
4 //Id della query assegnato dall'origine
5 private int id;
```

L'attributo `files` è una lista di OrdinarynodeFiles, Sezione 4.1.1.

L'attributo `id` richiama semplicemente l'id univoco della query di cui fa parte l'oggetto `Answer`.

3.3.4 SearchField.java

La classe `SearchField.java` viene utilizzata dal client Mini-KaZaA per tenere in memoria tutti i risultati associati a una richesta di file. Da uno di questi campi poi vengono estratte le informazioni per eventuali download di file.

Anche questa classe è piuttosto semplice poichè funziona da appoggio alla rappresentazione grafica e per snellire la quantità di informazioni da tenere in memoria per l'utente.

Il codice che descrive gli attributi della classe è il seguente:

```
1  //File owner
2  private NodeInfo owner;
3
4  //File descriptor
5  private MKFileDescriptor file;
```

Con queste due semplici informazioni è possibile sia risalire al proprietario, compreso l'indirizzo ip da contattare per il download, sia ottenere tutti i metadati[1] del file da scaricare[2].

3.3.5 Download.java

La classe `Download.java` serve al client Mini-KaZaA per indicizzare i downloads che si effettuano. Questa classe si distingue da `SearchField.java` perchè mantiene anche il numero di byte già scaricati.

```
1  private MKFileDescriptor file_to_download;
2  private long downloaded_bytes;
3  private String downloader_path;
4
5  public Download(MKFileDescriptor file){
6
7    this.file_to_download = file;
8    this.downloaded_bytes = 0;
9
10   //Directory di default
11   this.downloader_path = "./downloads/";
12 }
```

[1] Quando parliamo di metadati ci riferiamo alle informazioni che riguardano i file, come la dimensione, il path assoluto ecc. Queste informazioni si possono sempre ricavare perchè salvate all'interno del file.

[2] I download così come le ricerche vengono effettuati mediante l'hash univoco md5 che tratteremo nella Sezione 6.2

Gli attributi della classe `Download.java` sono quelli elencati nel listato mostrato appena sopra. `file_to_download` identifica il file che si sta scaricando tramite il codice hash *md5*. Viene anche usato il nome del file, salvato all'interno di `MKFileDescriptor`, per poter comporre il path assoluto con l'attributo `downloader_path`. I `downloaded_bytes` invece indicano, quanta parte di file è stata già scaricata dalla rete.

3.3.6 DownloadRequest.java

La classe `DownloadRequest.java` viene utilizzata dal client Mini-KaZaA per richiedere file da scaricare al nodo che lo possiede. Si compone dei seguenti attributi:

```
1 //File da scaricare
2 private String file_request;
3
4 //Sorgente della richiesta
5 private NodeInfo request_source;
```

3.3.7 DownloadResponse.java

Il client Mini-KaZaA fa un uso particolare della classe `DownloadResponse.java`. Essa viene infatti usata sia per inizializzare e terminare una comunicazione, sia come "mezzo di trasporto" delle varie parti di un file.

Vediamo innanzitutto quali attributi include questa classe:

```
1 //byte che compongono una parte di
2 //un file
3 private byte [] part;
4
5 //File inviato
6 private String file;
```

Mini-KaZaA "spezzetta" il file da inviare in piccoli pacchetti da 4Kb che vengono inseriti all'interno dell'array `part`. `file` invece contiene l'md5 del file che viene inviato. Questa classe, e la combinazione dei suoi paramtri, permettono a Mini-KaZaA di controllare l'inizio di una comunicazione, la fine della stessa, e tutti gli invii intermedi di byte. Vedremo meglio come funziona lo scambio di file all'interno della Sezione 3.8

3.4 Il percorso di una query

Nella sezione 3.3.2 abbiamo visto di cosa si compone la classe `Query.java`. Ora vediamo come viene utilizzata dal client nello scambio di richieste.

Ogni query di ricerca di file comincia con l'input dell'utente tramite un apposita form di cui parleremo nella Sezione 3.9. Viene così generato un oggetto di tipo query con il seguente frammento di codice:

```
1 Query q = new Query();
2 q.setId(this.my_num);
3 q.setSender(this.my_infos);
4 q.setOrigin(this.my_infos);
5 q.setAskingQuery(this.search_tf.getText());
6 q.setTTL(this.my_conf.getTimeToLeave());
```

Questo oggetto viene così inviato nella rete attraverso i SN. Ogni SN leva un'unità di *TTL* alla query e la rimbalza nella rete.

La gestione del percorso di una query è affidata completamente alla classe `SupernodeTCPWorkingThread.java` e ne parleremo in Sezione 5.3. Quest'operazione è comune ai due tipi di client, ma contiene delle ovvie differeze, dovute alla natura dei vari nodi, che spiegheremo più avanti.

3.5 La classe SupernodeList.java

La classe `SupernodeList.java` è una delle classi principali del progetto. Essa infatti mantiene un elenco dei SN presenti sulla rete. Essa in realtà è molto utile per indicizzare insiemi di nodi di qualsiasi tipo, difatti il Bootstrap server utilizza proprio questa classe per le sue liste di nodi.

Come si può vedere dal seguente listato:

```
1 private ArrayList<NodeInfo> sn_list;
2 private ArrayList<NodeInfo> sub_set_list;
3
4 public SupernodeList() {
5   this.sn_list = new ArrayList();
6   this.sub_set_list = null;
7 }
```

la classe `SupernodeList.java` è composta da due attributi:

- `sn_list`: un'*ArrayList* di `NodeInfo` utilizzata per tenere in memoria *tutti* i nodi della rete;
- `sub_set`: una seconda *ArrayList* di `NodeInfo` utilizzata di volta in volta per memorizzare un sottoinsieme di nodi vicini e convenienti da contattare.

La classe fornisce anche tutti i metodi per gestire al meglio l'elenco di nodi.

Per prima cosa ci sono due overload per quanto riguarda il calcolo della latenza di un nodo[3]

Il metodo in questo listato

[3]La latenza viene calcolata in ms tramite una procedura che prende il nome di *ping* che vedremo in Sezione 3.7

```
1  public synchronized void
2  refreshPing(InetAddress ia, int port, long new_ping) {
3    for (NodeInfo n : sn_list) {
4      //Confrontiamo l'indirizzo del nodo estratto
5      //con quello passato come parametro del metodo
6      if (n.getIaNode().toString().equals(ia.toString())) {
7        if (n.getDoor() == port) {
8          n.setPing(new_ping);
9        }
10     }
11   }
12 }
```

viene usato solo per cambiare il valore della latenza a un nodo specifico.

Il secondo overload del metodo mostrato in precedenza consente invece di effettuare una passata di tutti i nodi che abbiamo nella nostra lista e di aggiornarne il valore della latenza.

```
1  public synchronized void refreshPing() {
2    //Thread pool
3    ThreadPoolExecutor my_thread_pool =
4      new ThreadPoolExecutor(
5        10, 15, 50000L,
6        TimeUnit.MILLISECONDS,
7        new LinkedBlockingQueue<Runnable>());
8
9    if (this.sn_list.size() >= 1) {
10     for (NodeInfo n : sn_list) {
11       NodePing pinging =
12         new NodePing(
13           n.getIaNode(),
14           n.getDoor(),
15           this);
16
17       my_thread_pool.execute(pinging);
18     }
19   }
20   my_thread_pool.shutdown();
21
22   this.setChanged();
23   this.notifyObservers();
24 }
```

Questo metodo invece non vuole parametri poichè effettua l'aggiornamento su tutto il set di nodi.

Mini-KaZaA crea un overlay network dinamico grazie al metodo `subSet()` che seleziona un insieme di SN valutati come "vicini", Sezione 3.7, a seconda della loro latenza. Per esplorare nuove porzioni della rete ne sceglie due sopra i 50ms di latenza e tutti gli altri al di sotto. Di seguito riportiamo la porzione di codice che si occupa di questa scelta.

```
1  public synchronized void
2  subSet(int set_size, long threshold) {
```

```
3     ArrayList<NodeInfo> neighbors = new ArrayList();
4
5     for (NodeInfo n : this.sn_list) {
6       if (n.getPing() != -1) {
7
8         if (n.getPing() <= threshold) {
9           neighbors.add(n);
10          if (neighbors.size() == set_size) {
11            this.sub_set_list = neighbors;
12          }
13        }
14      }
15    }
16
17    this.sub_set_list = neighbors;
18  }
19
20  public synchronized ArrayList<NodeInfo> getSubSet() {
21
22    if (this.sub_set_list == null) {
23      subSet(10, 100);
24    }
25
26    return this.sub_set_list;
27  }
```

Per gli Ordinary Node che devono scegliere il loro Super Node di riferimento la classe `SupernodeList.java` mette a disposizione un metodo che seleziona il nodo "migliore" al quale connettersi. Il metodo si chiama `getBest()` e il codice che lo riguarda è riportato di seguito.

```
1   public synchronized NodeInfo getBest() {
2     NodeInfo best = new NodeInfo();
3
4     for (NodeInfo candidate : this.sn_list) {
5       if (best.getIaNode() == null) {
6         best.setInetAddress(candidate.getIaNode());
7         best.setCallbacksInterface(
8           candidate.getCallbackInterface());
9         best.setDoor(candidate.getDoor());
10        best.setId(candidate.getId());
11        best.setUsername(candidate.getUsername());
12        best.setPing(candidate.getPing());
13      } else {
14        if (candidate.getPing() < best.getPing()) {
15          best.setInetAddress(candidate.getIaNode());
16          best.setCallbacksInterface(
17            candidate.getCallbackInterface());
18          best.setDoor(candidate.getDoor());
19          best.setId(candidate.getId());
20          best.setUsername(candidate.getUsername());
21          best.setPing(candidate.getPing());
22        }
23      }
```

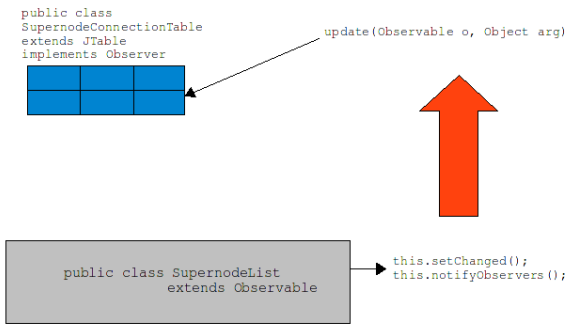


Figura 3.1: Paradigma Observable - Observer

```
24    }
25
26    return best;
27  }
```

3.6 Il paradigma Observable-Observer

Il linguaggio Java fornisce due classi molto utili per gestire in modo asincrono i cambiamenti di stato di determinate strutture dati.

È molto utile, infatti, avere un modo agile e integrato nel linguaggio di nodificare ogni cambiamento che subisce una struttura dati. Esempi che riguardano Mini-KaZaA sono molto frequenti: la struttura dati che memorizza i file trovati e la tabella che li deve mostrare in grafica, la lista di SN e la tabella di Net Monitor e così via.

Java mette a disposizione la classe `Observable`,con la quale è possibile mettere "sotto osservazione" una determinata classe che estende Observable con il comando `public class MyClass extends Observable`, e la classe `Observer`, che va implementata con il comando `public class MyClass implements Observer`.

La situazione che si viene a creare è mostrata in Figura 3.1. La prima classe fornisce due metodi che vanno richiamati ogni qual volta si effettua una modifica all'oggetto che estende `Observable`:

```
1  this.setChanged();
2  this.notifyObservers();
```

Questi due metodi svegliano il metodo `update()`, di cui deve essere fatto *obbligatoriamente* l'override, che è contenuto nella classe che implementa `Observer`. La firma del metodo è sempre la solita:

```
1  public void update(Observable o, Object arg)
```

Il primo parametro rappresenta l'oggetto che ha chiamato il metodo `update()`. Per generalizzare il metodo, questo parametro rappresenta un *Observable* sul

quale poi dovrà essere fatto un'operazione di *cast* per convertirlo nell'oggetto corretto e poter utilizzare le sue funzioni. Il secondo parametro rappresenta i parametri aggiuntivi che possono essere utili al metodo update. Il metodo update definirà quindi tutte le operazioni che dovranno essere fatte a ogni aggiornamento della struttura dati sotto osservazione.

3.7 Ping dei nodi

Ogni client Mini-KaZaA ha bisogno di sapere quanto dista dalla rete e, di conseguenza, quanto i nodi della rete distano da lui. Come unità di misura per la distanza da un nodo all'altro vengono usati i millisecondi che intercorrono dall'invio di un particolare pacchetto alla ricezione del pacchetto di risposta.

In Mini-KaZaA sono stati implementati due *Task*: NodePing e NodePong. Il primo task crea un pacchetto Datagram con le seguenti istruzioni:

```
1  DatagramSocket my_datagram_socket = null;
2
3  try {
4    my_datagram_socket = new DatagramSocket();
5  } catch (SocketException ex) {
6    //Log
7  }
8
9  byte [] data = new byte[32];
10
11 DatagramPacket pack =
12 new DatagramPacket(
13   data,
14   data.length,
15   host_ia,
16   host_port);
17
18 //Preparazione del pacchetto package
19 pack.setData(data,0,data.length);
20 pack.setLength(data.length);
```

dopo di che lo invia al nodo destinatario e fa partire un timer.

```
1  long start_time = System.currentTimeMillis();
2
3  try {
4    my_datagram_socket.send(pack);
5  } catch (IOException ex) {
6    //Log
7  }
```

Non appena arriva il medesimo pacchetto di risposta viene calcolato il tempo in millisecondi intercorsi fra l'invio e la ricezione del pacchetto e questa sarà la stima della distanza fra i due nodi. Le istruzioni che si occupano della ricezione del pacchetto sono le seguenti:

```
1  try {
2
3    my_datagram_socket.receive(pack);
4  } catch (IOException ex) {
5    //Log
6  }
7
8  long arrive_time = System.currentTimeMillis();
9
10 long ping = arrive_time - start_time;
```

Il secondo task invece funziona con il procedimento opposto. Ovvero, inizialmente predispone un socket di ascolto sulla porta scelta dall'utente al primo avvio del programma.

```
1 int port = this.my_conf.getPort();
2 DatagramSocket pong_sock = null;
3 try {
4   pong_sock = new DatagramSocket(port);
5 } catch (SocketException ex) {
6   //Log
7 }
```

Poi entra in un ciclo infinito, che verrà interrotto solo dall'uscita del programma, il cui unico compito è quello di rispondere il più velocemente possibile alle richieste di "ping" che arrivano alla porta del socket.

```
1  while (true) {
2
3    byte[] packet = new byte[32];
4
5    DatagramPacket pack =
6      new DatagramPacket(
7        packet,
8        packet.length);
9
10   try {
11
12     pong_sock.receive(pack);
13   } catch (IOException ex) {
14     //Log
15   }
16   packet = pack.getData();
17   DatagramSocket send_sock = null;
18   try {
19
20     send_sock = new DatagramSocket();
21   } catch (SocketException ex) {
22     //Log
23   }
24
25   byte [] send_byte = packet;
26   DatagramPacket send_pack =
27     new DatagramPacket(
28       send_byte,
```

```
29          send_byte.length,
30          pack.getAddress(),
31          pack.getPort());
32
33      try {
34
35        send_sock.send(send_pack);
36      } catch (IOException ex) {
37        //Log
38      }
39
40  }
```

Ovviamente ogni errore o eccezione che è sollevata in questi frammenti di codice viene loggata per future analisi[4]

3.8 Invio di file e divisione del file in parti.

L'attività principale che Mini-KaZaA prevede è lo scambio di file. I file vengono trasmessi sulla rete, anche su grandi distanze. I file possono anche essere molto grandi quindi non conviene inviare tutto il file in un unico momento. È molto più conveniente dividere il file in parti. Di questo si occupa la parte che invia il file nel *TCPWorkingThread*. Il codice che esegue queste operazioni è comune a tutti i tipi di nodi, dato che sia SN che ON possono inviare file sulla rete, ed è esposto nel seguente listato.

```
1   if (incoming_obj instanceof DownloadRequest) {
2     //Devo spedire il file richiesto
3     DownloadRequest request =
4       (DownloadRequest) incoming_obj;
5
6     MKFileDescriptor file_to_send =
7       this.my_files.getFileList(request.getFile());
8
9     Socket send_sock = new Socket(
10        request.getSource().getIaNode(),
11        request.getSource().getDoor());
12
13    ObjectOutputStream output =
14      new ObjectOutputStream
15      (send_sock.getOutputStream());
16
17    //Avvisa l'inizio del trasferimento.
18    DownloadResponse init_response =
19      new DownloadResponse(null, file_to_send.getMd5());
20
21    output.writeObject(init_response);
22
```

[4]In questi frammenti di codice abbiamo omesso la chiamata allaclasse di log per concentraci maggiormente sulle istruzioni che sono più strettamente legate alle misurazioni delle latenze tramite socket UDP

```
23    File file_pointer = new File(file_to_send.getPath());
24
25    FileInputStream in_file =
26      new FileInputStream(file_pointer);
27
28    byte[] buffer = new byte[4096];
29
30    while (true) {
31      try {
32        int letti = in_file.read(buffer);
33        if (letti > 0) {
34          DownloadResponse filepart =
35            new DownloadResponse(buffer, null);
36          output.writeObject(filepart);
37        } else {
38          break;
39        }
40      } catch (IOException e) {
41        break;
42      }
43    }
44
45    //Chiude la comunicazione
46    DownloadResponse stop_sending =
47      new DownloadResponse(null, file_to_send.getMd5());
48    output.writeObject(stop_sending);
49
50    in_file.close();
51    output.flush();
52    output.close();
53
54    send_sock.close();
55
56
57  }
```

Il procedimento è standard. Ovvero, prima si inizializza la connessione con una prima comunicazione, poi si inviano le varie parti di file e infine si chiude la comunicazione.

Dall'altro lato si troverà chi ha inviato la richiesta di download e che ora si prepara alla ricezione. Il listato che riceve il file è il duale di quanto appena mostrato ed è il seguente.

```
1   if (incoming_obj instanceof DownloadResponse){
2     DownloadResponse response =
3       (DownloadResponse) incoming_obj;
4
5     //Inizio dell'invio di un file.
6     if (response.getPart() == null) {
7       Download file_dl =
8         this.my_dl_monitor.getDownload(response.getFile());
9
10      File file = new File(
11        file_dl.getDownloaderPath() +
```

```
12          file_dl.getFile().getFileName());
13
14        FileOutputStream file_downloading =
15          new FileOutputStream(file);
16
17        while (true) {
18          Object read_part = input_object.readObject();
19
20          if (read_part instanceof DownloadResponse) {
21            DownloadResponse part = (DownloadResponse) read_part;
22            if (part.getFile() == null) {
23              //Posso inserire la parte sia
24              //nel monitor che nel file effettivo.
25              byte[] buffer = part.getPart();
26
27              if (buffer.length > 0) {
28                file_downloading.write(buffer, 0, buffer.length);
29                part.setFile(file_dl.getFile().getMd5());
30                this.my_dl_monitor.addBytes(part);
31              }
32            }
33            else{
34              break;
35            }
36          }
37        }
38
39        file_downloading.flush();
40        file_downloading.close();
41      }
```

3.9 La grafica del client Mini-KaZaA

Il client Mini-KaZaA è dotato di un front-end grafico per facilitarne l'utilizzo agli utenti finali. La grafica di Mini-KaZaA è scritta usando le librerie Swing, che mette a disposizione il linguaggio Java, con un unica differenza:

```
1 try {
2   UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
3 } catch (Exception ex) {
4 }
```

con queste operazioni si vuole che la grafica Swing ottenga il rendering dall'environment di sistema. Questo comando funziona, per il momento, solamente in ambiente grafico Win e Gnome (GTK+2), mentre non viene attivato, e quindi vengono visualizzati con il rendering originale i componenti Swing, su KDE[5]

[5]Sistemi utilizzati per questo tipo di test: Windows Vista(Win), Ubuntu(Gnome), Xandros OS (KDE).

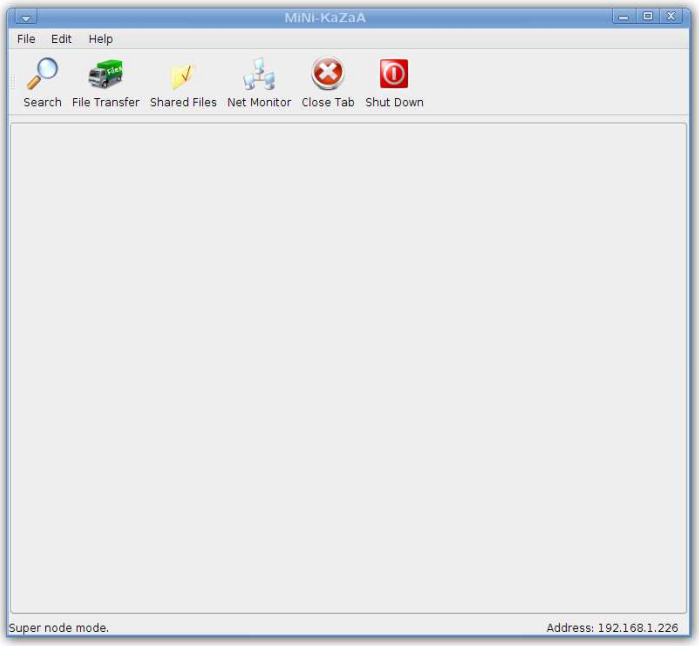


Figura 3.2: L'interfaccia grafica principale del client.

L'interfaccia principale (Figura 3.2), comune a tutti i tipi di nodi si divide in quattro parti principali:

1. **Barra dei menù:** la prima barra in alto dove è possibile trovare il menù:
   (a) *File*: per le operazioni sul client;
   (b) *Edit*: per il controllo delle configurazioni;
   (c) *Help*: per ottenere aiuto e informazioni su Mini-KaZaA.
2. **Barra dei pannelli:** contiene dei grossi pulsanti che attivano dei pannelli per le varie operazioni;
3. **Panel Box:** il riquadro centrale nel quale compaiono e scompaiono i vari pannelli attivabili dalla barra;
4. **Barra di stato:** posizionata in fondo alla finestra contiene informazioni di sistema riguardante il client Mini-KaZaA.

Nella *barra dei pannelli* troviamo sei pulsanti, ognuno dei quali ha una precisa funzione:

1. **Search:** apre un pannello per la ricerca dei file nella rete;

2. **File transfer:** apre un pannello che consente di monitorare lo stato dei download in corso;

3. **Shared Files:** apre un pannello nel quale è possibile aggiungere o rimuovere file da condividere nella rete;

4. **Net Monitor:** se si è un SN questo pulsante è abilitato e consente di vedere quali super nodi ci sono nella rete, mentre se si è un ON non sarà possibile utilizzarlo;

5. **Close Tab:** ogni nuovo pannello viene aperto in un *tab* a se stante, per questo tramite questo bottone sarà possibile chiudere i *tab* aperti;

6. **Shut Down:** consente di chiudere il programma.

Capitolo 4

Ordinary Node

4.1 Le classi del package ordinarynode

All'interno del progetto Mini-KaZaA ogni package contiene delle classi che sono state scritte non solo per il tipo di nodo specifico, ma che, grazie alla *modularità* e alla *genericità* dei metodi, sono utili a tutti i tipi di nodi. Di seguito quindi presentiamo le varie classi del package ordinarynode, ma non vanno pensate come dedicate esclusivamente al tipo di nodo ON. Vanno piuttosto legate agli ON da un punto di vista concettuale, ma nulla vieta di utilizzare queste pratiche classi nei SN.

4.1.1 OrdinarynodeFiles.java

La classe `OrdinarynodeFiles.java` si occupa di indicizzare i file che il client decide di condividere all'interno della rete. Questa classe implementa *Observable*, quindi è sempre possibile monitorare tutte le informazioni sui file condivisi nella rete da parte del client. Per essere utilizzata il più possibile, la classe utilizza gli attributi che sono descritti, assieme al costruttore della classe, nel seguente frammento di codice.

```
1 private ArrayList<MKFileDescriptor> file_list;
2 private NodeInfo my_info;
3
4 public OrdinarynodeFiles(NodeInfo infos) {
5   this.my_info = infos;
6   this.file_list = new ArrayList();
7 }
```

L'*ArrayList* di *MKFileDescriptor*(Sezione 6.1), `file_list` consente di contenere in memoria tutti i file che si è deciso di condividere tramite l'apposito form. Fra gli attributi compare anche `my_info` che viene utilizzato dal client per segnalare nella rete che un determinato *set* di file appartiene a un certo nodo che ha come *NodeInfo*, Sezione 2.3.

La classe predispone anche una serie di metodi che consentono di manipolare la lista di file per aggiungere, modificare, estrapolare le informazioni.

addFiles

```
1  public synchronized void
2  addFiles(MKFileDescriptor[] new_files) {
3    for (int i = 0; i < new_files.length; i++) {
4
5      if (!isIn(new_files[i])) {
6        this.file_list.add(new_files[i]);
7      }
8    }
9    this.setChanged();
10   this.notifyObservers();
11 }
```

Questo metodo consente di inserire all'interno della lista un *array* di *MKFileDescriptor*. La scelta di ricevere come parametro un array di *MKFileDescriptor* è dettata da come Java interagisce con il file system. Java, infatti, quando si effettua la selezione di un insieme di file dal file system, restituisce un array di *File*. È quindi più semplice, quindi preferibile, effettuare una conversione fra *File* e *MKFileDescriptor*.

removeFiles

```
1  public synchronized void
2  removeFiles(MKFileDescriptor[] old_files) {
3
4    for (int i = 0; i < old_files.length; i++) {
5      int index = 0;
6      for (MKFileDescriptor file : this.file_list) {
7
8        if ((old_files[i].
9          getFileName().equals(file.
10         getFileName())) &&
11         old_files[i].getMd5().
12           equals(file.getMd5()) &&
13         old_files[i].getPath().
14           equals(file.getPath())) {
15
16
17         this.file_list.remove(index);
18         break;
19       }
20       index ++;
21     }
22
23   }
24   this.setChanged();
25   this.notifyObservers();
26 }
```

Questo metodo consente di rimuovere un *Array* di *MKFileDescriptor* dalla lista di file condivisi. La scelta di avere come parametro un array di *MKFileDescriptor* è già stata spiegata in Sezione 4.1.1. È un metodo piuttosto semplice che scorre la lista di file e si interrompe quando il doppio controllo[1] su *md5* e *path* del file segnala che il file è stato individuato. Una volta individuato il file, viene rimosso e il metodo viene interrotto.

searchFiles

```
1  public synchronized
2  ArrayList <OrdinarynodeFiles>
3  searchFiles(String regex){
4
5  ArrayList <OrdinarynodeFiles> l =
6    new ArrayList();
7    Pattern pattern = Pattern.compile(regex);
8    OrdinarynodeFiles files_found =
9      new OrdinarynodeFiles(this.my_info);
10
11   MKFileDescriptor [] new_array = null;
12   for(MKFileDescriptor file : file_list){
13     ArrayList <MKFileDescriptor> found_list =
14       new ArrayList();
15
16       Matcher matcher =
17         pattern.matcher(file.getFileName());
18
19       while(matcher.find()){
20         found_list.add(file);
21         new_array =
22           new MKFileDescriptor[found_list.size()];
23
24         int index = 0;
25
26         for(MKFileDescriptor file_just_found : found_list){
27           new_array[index] = file_just_found;
28           index ++;
29         }
30
31         files_found.addFiles(new_array);
32         l.add(files_found);
33       }
34
35   }
36
37   return l;
38 }
```

Questo metodo deve ritornare al chiamante un *ArrayList* di *Ordinarynode* in cui andranno inserite i risultati della ricerca.

[1] Abbiamo inserito un doppio contorllo su md5 (Sezione 6.2) e sul path del file per evitare qualsiasi tipo di problema dovuto a malfunzionamenti o scritture di memoria eseguite in maniera scorretta.

La ricerca viene effettuata prendendo come parametro una stringa, `regex`, che è un'espressione regolare. Con questa espressione regolare il metodo invoca due diverse classi: `Pattern`, che contiene tutti i pattern che l'espressione regolare contiene, `Matcher`, che si occupa di controllare eventuali *match*[2] con i file della lista. Una volta individuate delle corrispondenze, i file vengono inserite nella lista da ritornare, `l`, assieme alle *NodeInfo* relative al nodo proprietario dei file. Infine viene fatto un `return` della lista, anche se dovesse risultare vuota. Ritornale comunque la lista anche se fosse vuota è stata una decisione presa per un' implementazione a noi piu' congeniale del progetto.

4.1.2 OrdinarynodeDownloadMonitor.java

Parliamo di una classe che serve per indicizzare tutti i download di una determinata sessione. Questa classe implementa Observable, in modo che sia possibile far vedere in grafica in ogni momento lo stato dei download.

`OrdinarynodeDownloadMonitor.java` ha un solo attributo che consiste in un *ArrayList* di Download, Sezione 3.3.5. Questo attributo nel costruttore viene inizializzato con un nuovo ArrayList, quindi ogni nodo che partirà avrà un suo *DowloadMonitor*.

```
1  ArrayList <Download> downloads;
2
3  public OrdinarynodeDownloadMonitor(){
4    this.downloads = new ArrayList();
5  }
```

I file vengono divisi in parti come ved(remo) in Sezione 3.8. A ogni parte ricevuta è quindi necessario andare ad aggiornare il numero di byte scaricati per quel file. Questo è il compito del seguente metodo:

```
1  public synchronized boolean addBytes(DownloadResponse part){
2    //Individuo il download e agiungo i byte
3    for(Download d : downloads){
4      if(d.getFile().
5           getMd5().
6           equals(part.getFile())){
7        d.updateDownloadBytes(
8           part.getPart().length);
9
10     }
11   }
12
13   //Notifico il cambiamento
14   this.setChanged();
15   this.notifyObservers();
16
17   return true;
18 }
```

[2]Si ricerca una corrispondenza fra gli oggetti passati e i vari pattern dell'espressione regolare.

Questo metodo, quindi richiama l'attenzione di tutti gli *Observer* per aggiornare lo stato dei download. Infine, i metodi `set` e `get` sono piuttosto semplici e non meritano particolare attenzione.

4.1.3 OrdinarynodeFoundList.java

La classe `OrdinarynodeFoundList.java` consente di enumerare molto facilmente i risultati delle ricerche che si sono effettuate. Si compone dei seguenti attributi:

```
1  private int id;
2  private ArrayList<SearchField> found;
3
4  public OrdinarynodeFoundList(int n) {
5    this.id = n;
6    found = new ArrayList();
7  }
```

I due attributi descritti nel frammento di codice rendono molto semplice la gestione dei risultati delle ricerche. `int id` indica l' indice corrispondente alla query di ricerca che è stata lanciata. `ArrayList<SearchField> found` è una lista di tutti i risultati che corrispondono a una determinata ricerca, che vengono convertiti in oggetti di tipo `SearchField`, Sezione 3.3.4.

I campi che costituiscono la lista `found` vengono costruiti a partire dall' *Answer* che arriva con indice corrispondente a `id`. Il frammento di codice che si occupa di inserire nella lista i campi correttamente costruiti è il seguente:

```
1  public void add(Answer k) {
2    ArrayList <OrdinarynodeFiles> list =
3      k.getFilesList();
4
5    for(OrdinarynodeFiles of : list){
6      ArrayList <MKFileDescriptor>
7        answer_files = of.getFileList();
8
9      for(MKFileDescriptor files : answer_files){
10       SearchField field =
11         new SearchField(files,of.getOwner());
12
13       found.add(field);
14     }
15   }
16
17   this.setChanged();
18   this.notifyObservers();
19 }
```

Il metodo per prima cosa estrae la lista di *OrdinarynodeFiles*, Sezione 4.1.1, dopo di che scorre tutti gli *OrdinarynodeFiles* ed estrae i vari descrittori di files, MKFileDescriptor 6.1.

Il tipo *MKFileDescriptor* però non consente un indicizzazione per proprietario, quindi, sempre per mantenere modularità nel codice, Mini-KaZaA

converte i *MKFileDescriptor* in *SearchField* e li inserisce nell'*ArrayList*. Infine il metodo notifica tutti gli *Observer* che stanno monitornando l'oggetto.

I vari metodi `get` sono molto semplici, quindi non sono necessari ulteriori chiarimenti.

4.1.4 OrdinarynodeQuestionList.java

La classe `OrdinarynodeQuestionList.java` consente di effettuare uno storage di tutte le ricerche che si stanno effettuando con il client Mini-KaZaA.

Questa classe ha solo un attributo, che è una *List* di *OrdinarynodeFoundList*.

```
1  private List <OrdinarynodeFoundList> my_res_list;
2
3  public OrdinarynodeQuestionsList(){
4    this.my_res_list = new ArrayList();
5  }
```

In `my_res_list` vengono aggiunti di volta in volta i risultati delle varie ricerche. Ogni volta che arriva il risultato di una ricerca sarà quindi necessario individuare a quale ricerca si riferisce. Questo si può fare grazie all'indice `id` che è contenuto sia nell'*Answer* sia nella *OrdinarynodeFoundList*.

```
1  public synchronized void add(Answer a){
2    for(OrdinarynodeFoundList l : this.my_res_list){
3      if(a.getID() == l.getId()){
4        l.add(a);
5        return;
6      }
7    }
8  }
```

La classe `OrdinarynodeQuestionList.java` mette anche a disposizione un metodo che consenta di individuare uno specifico file, tramite il codice md5, Sezione 6.2.

```
1  public SearchField getFile(String md5){
2
3    for(OrdinarynodeFoundList list : this.my_res_list){
4      ArrayList <SearchField> file_list =
5        list.getFoundList();
6
7      for(SearchField file : file_list){
8        if((file.getFile().
9          getMd5()).equals(md5)){
10
11         return file;
12       }
13     }
14   }
15
```

```
16    return null;
17  }
```

4.1.5 OrdinarynodeFriendRequest.java

La classe `OrdinarynodeFriendRequest.java` è la classe su cui si basa l'*overlay network* che si crea fra ON e SN. Questa classe serve infatti per comunicare a un SN che l'ON, mittente della richiesta, ha scelto come SN di riferimento proprio lui.

`OrdinarynodeFriendRequest.java` è un *Java bean*, Sezione 8.1 che contiene solo i metodi `set` e `get` che si possono visualizzare nel seguente frammento di codice.

```
1  private boolean want_to_be_friend;
2  private NodeInfo friend;
3
4  public OrdinarynodeFriendRequest(){ }
5  //Metodi set
6  public void
7  setRelationship(boolean rel)
8    {this.want_to_be_friend = rel;}
9
10 public void
11 setInfo(NodeInfo info)
12   {this.friend = info;}
13
14 //Metodi get
15 public boolean getRelationship()
16   {return this.want_to_be_friend;}
17
18 public NodeInfo getInfo()
19   {return this.friend;}
```

4.2 Il cuore di un Ordinary Node

Oltre a tutte le classi che sono state descritte nelle sezioni precedenti, un Ordinary Node ha anche un motore che combina tutte le classi in modo che siano realmente operative. Come è possibile vedere il diagramma di disposizione logica di Mini-KaZaA, Sezione 8.2 un Ordinary Node ha un motore principale e quattro interfacce principali con il mondo esterno:

- **Grafica utente:** che viene mostrata in modo chiaro e completo nel Capitolo 7, e nella Sezione 3.9;
- **Socket UDP:** utilizzato solo per la misurazione delle latenze e mostrata nel Capitolo 8 in Sezione 3.7;
- **Socket TCP:** utilizzato per la maggior parte delle comunicazioni nella rete, mostrato anch' esso nel Capitolo 8;

- **Comunicazioni RMI:** utilizzato per l'interazione con il Bootstrap Server per gli aggiornamenti sui nodi della rete. RMI che tratteremo nel dettaglio nel Capitolo 8

Tutti questi componenti sono coordinati da un "motore" principale che ora andremo a descrivere.

4.2.1 Engine

Il motore di un Ordinary Node è il task

```
public class OrdinarynodeEngine implements Runnable
```

`OrdinarynodeEngine` ha il compito di inizializzare le variabili e gli oggetti che poi utilizzaranno i *thread* dell'ON e di far partire tutti i vari task che controllano le varie interfacce.

Guardiamo ora, in un piccolo frammento di codice, le variabili e gli oggetti istanziati dall'*engine* dell'Ordinary Node.

```
1  NodeInfo my_infos = new NodeInfo();
2
3  SupernodeList sn_list = new SupernodeList();
4
5  OrdinarynodeQuestionsList found_list =
6    new OrdinarynodeQuestionsList();
7
8  OrdinarynodeFiles my_file_list =
9    FileUtil.loadMySharedFiles(my_infos);
10
11 OrdinarynodeDownloadMonitor dl_monitor =
12   new OrdinarynodeDownloadMonitor();
13
14 BootstrapRMIWrapper rmi_stub =
15   new BootstrapRMIWrapper();
16
17 OrdinarynodeRefSn my_ref_sn =
18   new OrdinarynodeRefSn();
19 sn_list.addObserver((OrdinarynodeRefSn) my_ref_sn);
```

Fa quindi partire i *thread* che possiamo vedere nel seguente listato.

```
1  //Init TCP listener
2  OrdinarynodeTCPListener on_tcp =
3  new OrdinarynodeTCPListener(
4    this.my_conf,
5    found_list,
6    dl_monitor,
7    my_file_list);
8  Thread tcp_thread = new Thread(on_tcp);
9  tcp_thread.start();
10
11 //Init main GUI of supernode
12 MainGui main_gui = new MainGui(
```

```
13      this.my_conf,
14      my_file_list,
15      found_list,
16      sn_list,
17      my_infos,
18      dl_monitor,
19      rmi_stub,
20      my_ref_sn);
21  main_gui.setLocationRelativeTo(null);
22  main_gui.setVisible(true);
23
24  //Init RMI manager
25  OrdinarynodeRMIManager on_rmi =
26  new OrdinarynodeRMIManager(
27      this.my_conf,
28      my_infos,
29      sn_list,
30      rmi_stub,
31      my_ref_sn);
32  Thread rmi_thread = new Thread(on_rmi);
33  rmi_thread.start();
34
35  //Init ping service to receive pings
36  NodePong pong = new NodePong(this.my_conf);
37  Thread ping_service = new Thread(pong);
38  ping_service.start();
```

I *thread* dei quali viene eseguito il comando `.start()` sono quelli che poi andranno a gestire le varie interfacce che un client Mini-KaZaA deve avere.

4.2.2 ON in ascolto sul socket TCP

Ogni client Mini-KaZaA deve stare costantemente in ascolto sul socket TPC poichè da esso giungono la maggior parte delle comunicazioni. Ogni client, pertanto, dedica un *thread* alla funzione di ascolto su tale socket.

`OrdinarynodeTCPListener.java` contiene un task con firma

```
public class OrdinarynodeTCPListener implements Runnable
```

Questo task ha il compito di accettare le richieste che provengono dalla rete e creare un *sotto-thread* che si occupi della richiesta specifica. La classe `OrdinarynodeTCPListener.java` ha i seguenti attributi.

```
1  private NodeConfig my_conf;
2  private OrdinarynodeQuestionsList my_found_list;
3  private OrdinarynodeDownloadMonitor my_dl_monitor;
4  private OrdinarynodeFiles my_files;
5
6  public OrdinarynodeTCPListener(
7      NodeConfig conf,
8      OrdinarynodeQuestionsList list,
9      OrdinarynodeDownloadMonitor dl_monitor,
10     OrdinarynodeFiles files){
```

```
11
12    this.my_conf = conf;
13    this.my_found_list = list;
14    this.my_dl_monitor = dl_monitor;
15    this.my_files = files;
16  }
```

Gli attributi appena mostrati occorrono al *TCPListener* per poterli poi far ereditare ai *sotto-thread*. Essi poi li utilizzeranno per rispondere alle richieste che arrivano dalla rete. Diamo uno sguardo al codice che si occupa di ricevere le richieste e far partire i *sotto-thread*.

```
1  ServerSocket listen_sock = null;
2  Socket incoming_sock = null;
3
4  ThreadPoolExecutor answer_pool = new ThreadPoolExecutor
5      (10,15,50000L,
6      TimeUnit.MILLISECONDS,
7      new LinkedBlockingQueue <Runnable>());
8
9  try{
10   listen_sock =
11     new ServerSocket(this.my_conf.getPort());
12 }
13 catch(IOException ex){
14   //Log
15 }
16
17 while(true){
18   try {
19     incoming_sock = listen_sock.accept();
20
21     OrdinarynodeTCPWorkingThread tcp_job =
22         new OrdinarynodeTCPWorkingThread(
23         incoming_sock,
24         this.my_found_list,
25         this.my_dl_monitor,
26         this.my_files);
27
28     answer_pool.execute(tcp_job);
29   } catch (IOException ex) {
30     //Log
31   }
32 }
```

Il codice mostrato è piuttosto semplice ma utilizza una tecnologia Java molto utile: *Thread Pool*8.7. Ogni richiesta che il client riceve sul socket TCP viene infatti delegata a un *sotto-thread* che la elabora, lasciando libero il thread principale di ricevere altre richieste sul socket.

4.2.3 ON e RMI

La connessione di un Ordinary Node al Bootstrap server avviene tramite protocollo RMI che consente di richiamare metodi di classi che stanno su una macchina remota e di elaborare i dati di ritorno.

La connessione tramite protocollo RMI viene gestita dal task

`public class OrdinarynodeRMIManager implements Runnable`

Questa classe ha i parametri descritti nel frammento di codice che viene riportato sotto.

```
1  private NodeConfig my_conf;
2  private NodeInfo my_infos;
3  private SupernodeList sn_list;
4  private BootstrapRMIWrapper rmi_stub;
5
6  public OrdinarynodeRMIManager(
7      NodeConfig conf,
8      NodeInfo info,
9      SupernodeList list,
10     BootstrapRMIWrapper rmi) {
11
12   this.my_conf = conf;
13   this.my_infos = info;
14   this.sn_list = list;
15   this.rmi_stub = rmi;
16 }
```

L'attributo `my_conf` e `my_infos` vengono utilizzati dalla classe per recuperare informazioni riguardanti il nodo come il tipo di nodo o l'indirizzo IP del nodo. Gli attributi `sn_list`, `rmi_stub` e `my_infos` vengono utilizzati all'interno di tutto il client ma vengono istanziati all'interno del task `OrdinarynodeRMIManager` poichè le informazioni necessarie alla loro istanziazione sono recuperabili esclusivamente dal Bootstrap Server.

Diamo anche uno sguardo a come l'*RMIManager* utilizza il protocollo RMI per recuperare le informazioni utili.

```
1  Registry bootstrap_service;
2
3  BootStrapServerInterface callbacks_remote;
4
5  SupernodeCallbacksInterface callbacks_stub;
6
7  try {
8    bootstrap_service =
9      LocateRegistry.getRegistry
10     (my_conf.getBootStrapAddress(), 2008);
11
12   //Divisione logica delle chiamate a procedure remote
13   rmi_stub.setStub((BootStrapServerInterface)
14     bootstrap_service.lookup("BootStrap"));
15
```

```
16    callbacks_remote = (BootStrapServerInterface)
17      bootstrap_service.lookup("BootStrap");
18
19    ArrayList<NodeInfo> ni_list =
20      rmi_stub.getStub().getSuperNodeList();
21
22    sn_list.refreshList(ni_list);
23
24    //Managing callbacks.
25    SupernodeCallbacksImpl callback_obj =
26      new SupernodeCallbacksImpl(
27        this.sn_list,
28        this.my_conf);
29
30    callbacks_stub =
31      (SupernodeCallbacksInterface)
32      UnicastRemoteObject.exportObject(callback_obj, 0);
33
34    //Modifica degli attributi di NodeInfo
35    try {
36      my_infos.setInetAddress(
37        InetAddress.getByName(my_conf.getMyAddress()));
38
39      my_infos.setDoor(
40        my_conf.getPort());
41
42      my_infos.setCallbacksInterface(
43        callbacks_stub);
44
45      my_infos.setIsSn(
46        my_conf.getIsSN());
47
48      my_infos.setId(
49        this.my_conf.getMyAddress()+
50        ":"+this.my_conf.getPort());
51
52      this.my_sn_ref.setMyInfo(this.my_infos);
53
54    } catch (UnknownHostException ex) {
55      //Log
56    }
57
58    callbacks_remote.addOrdinaryNode(my_infos);
59
60    sn_list.refreshList(ni_list);
61    sn_list.refreshPing();
62
63  } catch (RemoteException ex) {
64    //Error message
65  } catch (NotBoundException ex) {
66    //Error message
67  }
```

Dal listato possiamo notare come in un primo momento l'*RMIManager* si

connette al Bootstrap Server tramite protocollo RMI e successivamente, una volta ottenute le informazioni riguardanti i SN presenti nella rete, inizializza tutti gli attributi della classe *NodeInfo* poi richiama la procedura remota per aggiungere se stesso alla lista degli Ordinary Node presenti nella rete.

Le eccezioni catturate si riferiscono all'impossibilità di creare una connessione con il Bootstrap Server per errori nell'inserimento dell'indirizzo IP. La gestione di tali eccezioni è stata oscurata per rendere più leggibile il codice. In realtà viene presentato un *Dialog Panel* che consente di modificare *on the fly* l'indirizzo del Bootstrap Server.

4.2.4 Scelta del SN al quale connettersi

Un nodo di tipo ON che si connette alla rete ha bisogno di connettersi a un SN per poter accedere alla rete e cominciare a condividere ed effettuare ricerche all'interno della rete. Scegliere però un nodo è uno dei problemi principali che un ON incontra appena inizia la sua attività. Mini-KaZaA client ha quindi, grazie al paradigma *Observer-Observable*, Sezione 3.6, un sistema che monitora i SN nella rete e sceglie il migliore.

Ciò avviene attraverso la classe `OrdinarynodeRefSn.java` che contiene i senguenti attributi con il costurttore, descritti all'interno del seguente frammento di codice.

```
1  private Socket my_sn;
2  private NodeInfo best_sn;
3  private int num_query;
4  private ObjectOutputStream output_object;
5  private NodeInfo my_info;
6
7  public OrdinarynodeRefSn() {
8    this.num_query = 0;
9    this.my_sn = null;
10   this.best_sn = null;
11 }
```

Gli attributi descritti sono quelli che servono per instanziare una connessione tra l'ON e il SN. A fini statistici contiene anche il numero di query che vengono effettuate tramite questa connessione. Una prima domanda che potrebbe sorgere guardando questi attributi è: perchè creare un *wrapper* per un socket? La risposta sta nella dinamicità delle connessioni di rete che caratterizzano il progetto Mini-KaZaA.

Poichè è possibile che appena un ON entra nella rete non ci siano SN ai quali connettersi[3] Mini-KaZaA si affida ancora una volta al paradigma *Observer-Observable*, Sezione 3.6, per risolvere eventuali inconsistenze.

La firma della classe è

```
public class OrdinarynodeRefSn implements Observer
```

[3]La spiegazione al perchè è stato creato un *wrapper* per un socket, Sezione 8.3.

quindi è interessante andare a vedere il metodo `public synchronized void update(Observable o, Object arg)` di cui la classe ha dovuto fare un *override*.

```
1  if (o instanceof SupernodeList) {
2    SupernodeList list = (SupernodeList) o;
3
4    if ((this.my_sn == null) &&
5      (list.getList().size() > 0)) {
6
7      NodeInfo best = list.getBest();
8
9      this.setSocket(
10       best.getIaNode(),
11       best.getDoor());
12
13     this.best_sn = best;
14
15     try {
16
17       this.output_object =
18         new ObjectOutputStream(
19           this.my_sn.getOutputStream());
20
21       OrdinarynodeFriendRequest friend_request =
22         new OrdinarynodeFriendRequest();
23
24       friend_request.setRelationship(true);
25
26       friend_request.setInfo(my_info);
27
28       output_object.writeObject(friend_request);
29
30     } catch (IOException ex) {
31       //Log
32     }
33   }
34 }
```

Innanzitutto il metodo verifica che l'oggetto che ha chiamato il metodo sia effettivamente di tipo *SupernodeList* e quindi esegue un *cast* per poterne sfruttare tutti i metodi. Fondamentale è, quindi, l'istruzione `NodeInfo best = list.getBest();` che estrae dalla lista `list` di tipo *SupernodeList* il nodo "migliore"[4] e lo salva all'interno della variabile `best`. Successivamente instanzia un *stream* di dati verso il nodo `best` e invia un oggetto di tipo *OrdinarynodeFriendRequest*, Sezione 4.1.5, con il valore `true`, che indica l'inizio di un "amicizia".

Va fatto notare che con le istruzioni

```
1  if ((this.my_sn == null) &&
2      (list.getList().size() > 0)) {
```

[4]Rimandiamo alla misurazione delle latenze, Sezione 3.7

l'ON evita di creare confusione nella rete istanziando più di una connessione.

4.2.5 Condivisione di file

La condivisione di file avviene non appena l'utente di un ON decide di condividere file tramite l'apposito pannello descritto in Sezione 3.9 Se si è un ON va però comunicata la selezione al proprio SN

Capitolo 5

Super Node

Il package `lpr.minikazaa.supernode` non contiene un gran numero di classi peculiari, ma attinge molto dal package per gli Ordinary Node. Fornisce però l'interfaccia per le *callbacks* RMI che viene utilizzata da tutto il client. Di seguito mostriamo le caratteristiche principali del package `supernode`.

5.1 L'interfaccia per le callback

L'interfaccia per le callback fonisce due metodi semplici ma fondamentali.

```
1  public void notifyMeAdd(NodeInfo new_node)
2    throws RemoteException;
3
4  public void notifyMeRemove(NodeInfo new_node)
5    throws RemoteException;
```

Questi due metodi servono al client per mantenersi aggiornato sui SN presenti nella rete. L'implementazione di questi due metodi è molto semplice poichè opera solo delle `add` o `remove` alla struttura *SupernodeList*. Non è pertanto interessante mostrarne il codice.

5.2 Indicizzamento dei file degli ON collegati

Come abbiamo già detto, gli Ordinary Node condividono i propri file attraverso il Super Node che scelgono come "amico". Un Super Node quindi ha bisogno di uno strumento per indicizzare le informazioni sui file condivisi che gli arrivano dalla rete.

Per questo motivo è stata predisposta una classe il cui obiettivo è tenere in memoria tutti i file condivisi dagli Ordinary Node. `SupernodeOnFileList.java` svolge proprio questo compito. È composta da un unico attributo che viene inizializzato nel costruttore come segue.

```
1  private ArrayList <OrdinarynodeFiles> file_list;
2
3  public SupernodeOnFileList(){
4    this.file_list = new ArrayList();
5  }
```

Questa classe mette a disposizione anche diversi metodi utili a manipolare le informazioni che si trovano all'interno dell'*ArrayList*.

5.2.1 addNewOnFileList

Un metodo che serve all'inserimento di nuove liste di files. Questo metodo si accorge se è già presente una lista di file per un determinato nodo e, nel caso, sostituisce la lista vecchia con la lista nuova. Ciò semplifica la gestione delle liste riducendo le operazioni di add, update e remove a una sola operazione, e di conseguenza gli errori in cui si può incorrere. Nel caso non esista già una lista per il nodo mittente, si aggiunge un nuovo campo all'*ArrayList* `file_list`. Mostriamo quindi il frammento di codice che si occupa di queste operazioni.

```
1  for(OrdinarynodeFiles o : this.file_list){
2    if(o.getOwner().getId().
3      equals(new_file_list.getOwner().getId())){
4
5      //Abbiamo gia' informazioni per il
6      //nodo owner
7      o.resetList(new_file_list.getFileList());
8      return;
9    }
10 }
11
12 this.file_list.add(new_file_list);
```

5.2.2 searchFiles

La funzione di ricerca dei file viene fatta come per la classe OrdinarynodeFiles, Sezione 4.1.1. La differenza in questo caso è che si lavora su liste di OrdinaryNodeFiles, quindi bisogna distinguere gli *owner* di ogni lista di file. Il codice è uguale a quello di Sezione 4.1.1 per cui non vale la pena riportarlo nuovamente.

5.2.3 Altri metodi

La classe prevede altri metodi per la rimozione delle liste o per verificare se un file è già presente all'interno della struttura dati, ma sono metodi molto semplici composti da poche righe, quindi li citiamo, ma non li mostriamo completamente.

5.3 Smistamento delle query

Il compito principale di un Super Node, che poi è anche quello che lo differenzia principalemente da un Ordinary Node, è quello di smistare le *query* nella rete, ovvero spargere le richieste nella rete.

Per fare questo però un Super Node non solo deve inviare le *query* ai nodi che ritiene "più vicini", ma anche far ritornare le *query* al mittente una volta ricevute le risposte.

Questo compito viene svolto dal *TCPWorkingThread* sfruttando una struttura dati che si chiama `SupernodeQueryList`. Prima di tutto vediamo come questa struttura dati è costrutita.

Si compone di un solo attributo.

```
1  private ArrayList <Query> query_list;
```

Questo attributo viene inizializzato nel costruttore a una lista vuota. La classe mette anche a disposizione:

- **getRelativeQuery**: data una query di risposta ricevuta in input restituisce la relativa che è stata ricevuta in precedenza;
- **generateQueryList**: prende in input una query di ricerca e una lista di node info e genera n query per gli n nodi contenuti nella lista passata come parametro;

Questa classe viene utilizzata dal Super Node nel suo *TCPWorkingThread* che, una volta riconosciuto che l'oggetto che sta arrivando sul socket è una query, controlla che tipo di query è e poi agisce di conseguenza. Vediamo questo procedimento nel dettaglio.

Un primo caso è che la query entrante sia una query di richiesta. Si riconosce perchè ha solo il `body_q` diverso da `null`.

```
1  if (peer_query.getBodyQ() != null &&
2    peer_query.getBodyA() == null &&
3    peer_query.getBodyF() == null)
```

Dopo aver inviato le risposte al nodo che ha inviato la query, Sezione 8.8, si richiamano le funzioni della class mostrata sopra per poter smistare la query.

```
1  //Propagazione query
2  if (peer_query.getTTL() >= 1) {
3    //Propaga la query
4    ArrayList<Query> out_query_queue =
5        this.my_q_list.
6        generateQueryList(peer_query,
7        this.my_list.getSubSet());
8
9    for (Query q : out_query_queue) {
10     NetUtil.sendQuery(q);
11   }
12 }
```

Ovviamente le operazioni di smistamento vengono effettuate se alla query non è ancora scaduto il TTL.

Il secondo caso è che la query che è arrivata è una risposta a una query precedentemente inviata.

```
1  if (peer_query.getBodyA() != null &&
2    peer_query.getBodyF() == null)
```

In questo caso è stato inserito un meccanismo per vedere se è una query di una nostra richiesta, onde evitare di rimbalzare query in maniera errata.

```
1  if (origin_id.equals(my_id)) {
2    //La risposta e' a una mia query,
3    //quindi interrompo il cammino
4    this.my_found_list.add(peer_query.getBodyA());
5  }
```

Altrimenti viene ricercata nella classe mostrata prima la query d'origine e inviata al risposta al nodo mittente della richiesta precedentemente salvata.

```
1  //E' arrivata al nodo una risposta di un altro peer,
2  //bisogna dunque inoltrarla al peer richiedente.
3  Query other_answer =
4    this.my_q_list.getRelativeQuery(peer_query);
5
6  other_answer.setAnswerQuery(peer_query.getBodyA());
7
8  //Scambio mittete e destinatario.
9  NodeInfo temp_sender = other_answer.getSender();
10 other_answer.setSender(other_answer.getReceiver());
11 other_answer.setReceiver(temp_sender);
12
13 //Send query.
14 NetUtil.sendQuery(other_answer);
```

Molto semplicemente vengono scambiati il *sender* e il *receiver* all'interno della query che precedentemente era arrivato al nodo sena ulteriori sprechi di informazioni memorizzate. Questo metodo ci ha permesso di semplificare notevolmente lo scambio delle query all'interno della rete, che come è possibile vedere, viene gestito da poche righe di codice. Le informazioni sui nodi attraversati dalla query potevano essere salvate all'interno della query, ma si sarebbe immessa nella rete una query sempre più grossa, quindi impegnativa da trasmettere. Abbiamo ritenuto questo metodo più agile ed elegante.

5.4 Il cuore di un Super Node

Oltre che dalle classi appena viste, un Super Node è composto anche da un motore principale che fa in modo che il SN, così come accade per un ON, "respiri". Questo concetto di motore è chiaramente visibile nel diagramma, Sezione 8.2, che decrive come sono relazionate le componenti di un Super Node.

5.4.1 Engine

Il motore di un Super Node è il task

```
public class SupernodeEngine implements Runnable
```

`SupernodeEngine` ha il compito di inizializzare le variabili e gli oggetti che poi utilizzaranno i *thread* dell'ON e di far partire tutti i vari task che controllano le varie interfacce.

Guardiamo ora, in un piccolo frammento di codice, le variabili e gli oggetti istanziati dall'*engine* dell'Ordinary Node.

```
1  NodeInfo my_infos =
2    new NodeInfo();
3
4  SupernodeList sn_list =
5    new SupernodeList();
6
7  SupernodeOnFileList on_files =
8    new SupernodeOnFileList();
9
10 OrdinarynodeFiles my_file_list =
11   FileUtil.loadMySharedFiles(my_infos);
12
13 OrdinarynodeQuestionsList found_list =
14   new OrdinarynodeQuestionsList();
15
16 OrdinarynodeDownloadMonitor dl_monitor =
17   new OrdinarynodeDownloadMonitor();
18
19
20 BootstrapRMIWrapper rmi_stub =
21   new BootstrapRMIWrapper();
```

Fa quindi partire i *thread* che possiamo vedere nel seguente listato.

```
1  //Inizializza la main GUI del supernode
2  MainGui main_gui = new MainGui(
3      this.my_conf,
4      my_file_list,
5      found_list,
6      sn_list,
7      my_infos,
8      dl_monitor,
9      rmi_stub,
10     null);
11 main_gui.setLocationRelativeTo(null);
12 main_gui.setVisible(true);
13
14
15 //Inizializza il servizio ping per ricevere i ping
16 NodePong pong = new NodePong(this.my_conf);
17 Thread ping_service = new Thread(pong);
18 ping_service.start();
```

```
19
20 //Inizializza RMI manager Thread.
21 SupernodeRMIManager sn_rmi = new SupernodeRMIManager(
22     my_conf,
23     sn_list,
24     my_infos,
25     rmi_stub);
26 Thread rmi_manager = new Thread(sn_rmi);
27 rmi_manager.start();
28
29 //Inizializza TCP requests manager
30 SupernodeTCPListener listener_tcp = new SupernodeTCPListener(
31     this.my_conf,
32     sn_list,
33     on_files,
34     my_file_list,
35     dl_monitor,
36     found_list);
37 Thread tcp_listen = new Thread(listener_tcp);
38 tcp_listen.start();
```

I *thread* dei quali viene eseguito il comando `.start()` sono quelli che poi andranno a gestire le varie interfacce che un client Mini-KaZaA, in particolare un Super Node, deve avere.

5.4.2 SN e RMI

Parliamo ora di come un Super Node sfrutta il protocollo RMI per recuperare informazioni sulla rete. Un Super Node dedica all'interazione con il Bootstrap Server tramite protocollo RMI un thread che esegue il task

`public class SupernodeRMIManager implements Runnable`

L'*RMIManager* del Super Node si compone degli attributi, passati dall'*Engine*, e vengono inizializzati, dal costruttore, come mostrato nel listato seguente.

```
1  private NodeConfig my_conf;
2  private SupernodeList sn_list;
3  private NodeInfo my_infos;
4
5  private BootstrapRMIWrapper rmi_stub;
6
7
8  public SupernodeRMIManager(
9      NodeConfig conf,
10     SupernodeList list,
11     NodeInfo infos,
12     BootstrapRMIWrapper rmi) {
13   this.my_conf = conf;
14   this.sn_list = list;
15   this.my_infos = infos;
16   this.rmi_stub = rmi;
17 }
```

Quando viene eseguito il `.start()` di questo task vengono eseguite nell'ordine le seguenti operazioni.

```
1  Registry bootstrap_service;
2  BootStrapServerInterface callbacks_remote;
3  SupernodeCallbacksImpl callback_obj = null;
4  SupernodeCallbacksInterface callbacks_stub = null;
5
6  try {
7
8    bootstrap_service =
9      LocateRegistry.getRegistry(
10     my_conf.getBootStrapAddress(),2008);
11
12   rmi_stub.setStub(
13     (BootStrapServerInterface)
14     bootstrap_service.lookup("BootStrap"));
15
16   callbacks_remote =
17     (BootStrapServerInterface)
18     bootstrap_service.lookup("BootStrap");
19
20   ArrayList<NodeInfo> ni_list =
21     rmi_stub.getStub().getSuperNodeList();
22
23
24   sn_list.refreshList(ni_list);
25
26   //Eseguo un refresh dei ping
27   sn_list.refreshPing();
28
29   //Gestione delle callbacks
30   callback_obj =
31     new SupernodeCallbacksImpl(
32     this.sn_list, this.my_conf);
33
34   callbacks_stub =
35     (SupernodeCallbacksInterface)
36     UnicastRemoteObject.
37     exportObject(callback_obj, 0);
38
39   //Creazione delle mie informazioni
40   try {
41     my_infos.setInetAddress(
42       InetAddress.getByName(my_conf.getMyAddress()));
43
44     my_infos.setDoor(my_conf.getPort());
45
46     my_infos.setCallbacksInterface(callbacks_stub);
47
48     my_infos.setIsSn(my_conf.getIsSN());
49
50     my_infos.setId(this.my_conf.getMyAddress()+
51             ":"+this.my_conf.getPort());
```

```
52      } catch (UnknownHostException ex) {
53        //Log
54      }
55      try {
56        //Aspetto di essere reputato affidabile
57        Thread.sleep(15000);
58      } catch (InterruptedException ex) {
59        //Log
60      }
61
62      callbacks_remote.addSuperNode(my_infos);
63
64    } catch (RemoteException ex) {
65      //Log
66    } catch (NotBoundException ex) {
67      //Log
68    }
```

L'*RMIManager* di un Super Node inizializza le classi per le callback, si da effettuare presso il Bootstrap Server, sia quelle che il Bootstrap Server può richiamare sul Super Node.

Appena ottenuta la lista dei Super Nodi effettua un *ping* su di loro, per poter essere subito disponibile a interrogarli anche se lui non è ancora registrato come SN da poter interrogare. Infatti un Super Node, contrariamente a quanto accade per gli Ordinary Node, ha un tempo di attesa prima di potersi registrare sul Bootstrap Server, richiamando l'apposita. Questo tempo di attesa viene dettato da una *sleep* che esegue il thread per un valore in millisecondi cablato all'interno del codice. Passato questo tempo il Super Nodo si iscrive al servizio di callback presso il Bootstrap server e fornisce il suo servizio di smistamento query a tutti gli effetti.

Abbiamo omesso le operazioni di log in caso di errori ma non sono particolarmente interessanti ai fini della descrizione della funzionalità.

5.4.3 SN in ascolto su socket TCP

Ogni client Mini-KaZaA deve stare costantemente in ascolto sul socket TPC poichè da esso giungono la maggior parte delle comunicazioni e un Super Node non fa eccezione. Ogni client, pertanto, dedica un *thread* alla funzione di ascolto su tale socket. In un Super Node il thread dedicato a questa funzione è contenuto nella classe `SupernodeTCPListener.java`. Questa classe contiene un task che ha la seguente firma

```
public class SupernodeTCPListener implements Runnable
```

e i seguenti attributi.

```
1  private NodeConfig my_conf;
2  private SupernodeList my_list;
3  private SupernodeOnFileList on_files;
4  private OrdinarynodeFiles my_files;
```

```
5  private OrdinarynodeDownloadMonitor my_dl_monitor;
6  private OrdinarynodeQuestionsList my_found_list;
7
8  public SupernodeTCPListener(
9      NodeConfig conf,
10     SupernodeList list,
11     SupernodeOnFileList file_list,
12     OrdinarynodeFiles sn_files,
13     OrdinarynodeDownloadMonitor dl_monitor,
14     OrdinarynodeQuestionsList found_list) {
15   this.on_files = file_list;
16   this.my_conf = conf;
17   this.my_list = list;
18   this.my_files = sn_files;
19   this.my_dl_monitor = dl_monitor;
20   this.my_found_list = found_list;
21 }
```

Gli attributi appena mostrati occorrono al *TCPListener* per poterli poi far ereditare ai *sotto-thread*. Essi poi li utilizzeranno per rispondere alle richieste che arrivano dalla rete. Diamo uno sguardo al codice che si occupa di ricevere le richieste e far partire i *sotto-thread*.

```
1  //Socket che ascolta tutte le richieste
2  //provenienti dalla rete
3  ServerSocket listen_sock = null;
4  Socket client_socket = null;
5  SupernodeQueryList query_list =
6      new SupernodeQueryList();
7
8  ThreadPoolExecutor answer_pool =
9      new ThreadPoolExecutor(
10     10,
11     this.my_conf.getMaxConnection(),
12     50000L,
13     TimeUnit.MILLISECONDS,
14     new LinkedBlockingQueue<Runnable>());
15 try {
16   listen_sock =
17     new ServerSocket(this.my_conf.getPort());
18 } catch (IOException ex) {
19 }
20
21 while (true) {
22   try {
23     client_socket = listen_sock.accept();
24     SupernodeTCPWorkingThread answer =
25         new SupernodeTCPWorkingThread(
26         client_socket,
27         this.my_conf,
28         this.my_list,
29         query_list,
30         this.on_files,
31         this.my_files,
```

```
32              this.my_dl_monitor,
33              this.my_found_list);
34          answer_pool.execute(answer);
35      } catch (IOException ex) {
36          //Log
37      }
38
39  }
```

Il codice mostrato utilizza una tecnologia Java molto utile: *Thread Pool*8.7. Ogni richiesta che il client riceve sul socket TCP viene infatti delegata a un *sotto-thread* che la elabora, lasciando libero il thread principale di ricevere altre richieste sul socket.

Capitolo 6

Il package Util

Durante lo sviluppo del client Mini-KaZaA sono state individuate una serie di funzioni di utilità varia che logicamente non appartenevano a nessun package visto il compito piuttosto particolare che dovevano svolgere.

6.1 Descrittore di file custom

Mini-KaZaA utilizza come descrittore dei file una classe che ha definito al suo interno. Questa classe contiene i metadati utili alla manipolazione dei file e consente di inviare sulla rete un oggetto piuttosto "agile" per scambiare informazioni sui files. `MKFileDescriptor.java` ha proprio queste caratteristiche, a cominciare dai suoi attributi.

```
1  private String file_name;
2  private String md5;
3  private long size;
4  private String absolute_owner_path;
5
6  public MKFileDescriptor(
7      String fn,
8      String code,
9      long s,
10     String path
11     ){
12
13   this.file_name = fn;
14   this.md5 = code;
15   this.size = s;
16   this.absolute_owner_path = path;
17 }
```

I parametri che vediamo in questo frammento riguardano tutti la descrizione del file nel computer del suo proprietario. Vedremo come viene calcolato l'md5 in Sezione 6.2.

Mentre i metodi `set` e `get` non sono molto interessanti da illustrare, è importante mostrare come vengono comparati fra di loro gli *MKFileDescriptor*.

```
1  @Override
2  public boolean equals(Object obj){
3    if(obj instanceof MKFileDescriptor){
4      MKFileDescriptor other = (MKFileDescriptor) obj;
5
6      if(this.getMd5().equals(other.getMd5()))
7        return true;
8    }
9    return false;
10 }
```

Da questo piccolo frammento si può notare quanto sia importante l'md5 e che, grazie alla sua bontà, possiamo comparare i vari file con un buon livello di certezza nel confronto.

6.2 Calcolo dell'md5

> L'acronimo MD5 (Message Digest algorithm 5) indica un algoritmo crittografico di hashing realizzato da Ronald Rivest nel 1991 e standardizzato con la RFC 1321.
>
> Questo tipo di codifica prende in input una stringa di lunghezza arbitraria e ne produce in output un'altra a 128 bit (ovvero con lunghezza fissa di 32 valori esadecimali, indipendentemente dalla stringa di input) che può essere usata per calcolare la firma digitale dell'input. La codifica avviene molto velocemente e si presuppone che l'output (noto anche come MD5 Checksum o MD5 Hash) restituito sia univoco (ovvero si ritiene che sia impossibile, o meglio, che sia altamente improbabile ottenere con due diverse stringhe in input una stessa firma digitale in output) e che non ci sia possibilità, se non per tentativi, di risalire alla stringa di input partendo dalla stringa di output (la gamma di possibili valori in output è pari a 16 alla 32esima potenza).

Questa definizione è stata estratta da http://it.wikipedia.org/wiki/Md5 e chiarifica piuttosto bene che cosa intendiamo per *md5*.

In Mini-KaZaA il calcolo di questo particolare codice è affidato alla classe `md5.java`. Questa classe contiene un solo metodo che prende in input un oggetto di tipo `File`[1].

Di seguito riportiamo il codice che salva i risultati in un MessageDigest e legge i primi 1024 byte di un file. Questo limite è stato imposto perchè

[1] Il tipo di dato standard che viene utilizzato da Java per manipolare i dati sul file system.

calcolare l'*md5* di file molto grossi richiede molto tempo. Inoltre, 1024 byte, assicurano un buon livello di diversità fra i vari codici md5. Successivamente viene convertito il `BigInteger` calcolato in una stringa e restituito. Se, durante il calcolo, si verifica un errore il metodo ritorna `null`.

```
1  final int MAX_BYTE = 1024;
2
3  try {
4    MessageDigest digest =
5      MessageDigest.getInstance("MD5");
6
7    InputStream is = new FileInputStream(file);
8
9    byte[] buffer = new byte[Constants.MAX_BYTE];
10
11   int read = 0;
12   while ((read = is.read(buffer)) > 0) {
13     digest.update(buffer, 0, read);
14   }
15
16   BigInteger bigInt =
17     new BigInteger(1, digest.digest());
18
19   return bigInt.toString(16);
20
21 } catch (Exception ex) {
22   return null;
23 }
```

6.3 Manipolazione delle stringhe

Il client Mini-KaZaA si trova molto spessoa dover manipolare delle stringhe, soprattutto la loro rappresentazione. Per questo è stata fornita una classe che consente di richiamare metodi pratici per la manipolazione delle stringhe.

Le funzioni di cui si è sentito il bisogno sono state principalmente due. La prima consente di affermare se una determinata stringa è o meno un indirizzo IP valido. La seconda invece fornisce una rappresentazione molto più leggibile delle dimensioni dei file traducendo i byte nei vari ordini superiori a seconda della grandezza del file.

Vediamo di seguito i listati dei due metodi con un breve commento.

isInetAddress

Ecco il metodo che controlla se un indirizzo, `String addr`, passato è veramente un indirizzo IP valido.

```
1  StringTokenizer tokenizer = new StringTokenizer(addr,".");
2  boolean result = true;
```

```
3
4  while(tokenizer.hasMoreTokens()){
5    String piece = tokenizer.nextToken();
6
7    int pic;
8    try{
9      pic= Integer.parseInt(piece);
10   }
11   catch(NumberFormatException ex){return false;}
12   if(pic< 0 || pic >255) result = false;
13 }
14 return result;
```

Questo metodo seziona la stringa di origine in quattro sotto-stringhe e cerca di convertirle in numero. Se ciò non è possibile significa già che l'indirizzo non è valido. Se la conversione è riuscita allora bisogna controllare che il numero non superi 255, che è il massimo numero consentito dagli indirizzi ip[2], o non sia inferiore di 0.

getRappresentableSize

Questo è invece il metodo che, preso in input un numero di byte (`long file_size`), restituisce una rappresentazione in stringa più leggibile e facilmente interpretabile.

```
1  String size = null;
2  //Bytes
3  if(file_size < 1024) {
4    size = file_size+" bytes";
5    if(file_size == 1)
6      size = file_size+" byte";
7  }
8  else if(file_size < 1024*1024){
9    size =
10   ((float)file_size / 1024) + " Kb";
11 }
12 else if(file_size < 1024*1024*1024){
13   size =
14   ((float)file_size /
15   (1024*1024)) + " Mb";
16 }
17 else if(file_size < 1024*1024*1024*1024){
18   size =
19   ((float)file_size /
20   (1024*1024*1024)) + " Gb";
21 }
22 else
23   size =
24   ((float)file_size /
25   (1024*1024*1024*1024)) + " Tb";
```

[2]Un indiritto IPv4 è formato da 4 byte separati da un punto e rappresentati in forma decimale $2^8.2^8.2^8.2^8$è*appunto*256.

```
26
27  return size;
```

Questo metodo è molto semplice. Guarda che soglia supera il numero di byte e poi converte il numero in una stringa molto compatta con l'unità di misura vicino.

6.4 Interazione con i file e con i metadati

Il client Mini-KaZaA ha una forte interazione con i file. Primo motivo fra tutti, il fatto che sia un programma di *file sharing*. È stato quindi indispensabile raccogliere in un unico punto tutte le funzioni che riguardano l'interazione dei file, sia a livello di trasmissione sulla rete, sia a livello di salvataggio di informazioni utili al client.

Guardiamo ora che funzioni mette a disposizione.

6.4.1 getFilesIntoDirectory

Questo metodo prende in input un oggetto di tipo `File` nominato `dir` che identifica una directory e restituisce i file che sono nella directory passata come parametro codificati con l'oggetto *MKFileDescriptor*. Il metodo si divide in due parti. Nella prima parte esclude dalle rilevazioni sulla directory tutte le sotto-directory creando un array di dimensione *numero elementi nella directory - numero directory*.

```
1  MKFileDescriptor[] file_array = null;
2
3  String[] file_list = dir.list();
4
5  int directory = 0;
6  for (int i = 0; i < file_list.length; i++) {
7    File f = new File(file_list[i]);
8    if (f.isDirectory()) {
9      directory++;
10   }
11 }
12
13 file_array =
14 new MKFileDescriptor[file_list.length - directory];
```

Nella seconda parte invece, converte tutti e soli i file, da `File` a `MKFileDescriptor`.

```
1  file_array = new MKFileDescriptor[file_list.length - directory];
2  int index = 0;
3  for (int i = 0; i < file_list.length; i++) {
4    File f = new File(file_list[i]);
5
6    if (!f.isDirectory()) {
7      MKFileDescriptor file = new MKFileDescriptor(
8          f.getName(),
9          md5.getMD5(f),
```

```
10            f.length(),
11            f.getAbsolutePath());
12        file_array[index] = file;
13        index++;
14      }
15    }
16    return file_array;
```

Alla fine il metodo ritorna l'array calcolato con tutti e soli i file della directory indicata.

6.4.2 transformFileToMKFile

Questo metodo prende in input un array di *File*, `files_array`, e si propone di restituire un array di *MKFileDescriptor*. Il corpo del metodo fa una semplice operazione: scorre tutti i *File* che ci sono in `files_array` e estrae le informazioni che occorrono agli attributi di un oggetto di tipo *MKFileDescriptor*. Di seguito il codice che esegue queste operazioni.

```
1   MKFileDescriptor[] mk_files =
2     new MKFileDescriptor[files_array.length];
3
4   for (int i = 0; i < files_array.length; i++) {
5     String file_name = files_array[i].getName();
6     String code = md5.getMD5(files_array[i]);
7     long size = files_array[i].length();
8     String abs_path =
9       files_array[i].getAbsolutePath();
10
11    mk_files[i] = new MKFileDescriptor(
12        file_name,
13        code,
14        size,
15        abs_path);
16
17  }
18
19  return mk_files;
```

Esiste anche un altro overload di questo metodo che prende in input un solo *File* e restituisce un solo *MKFileDescriptor* ma le operazioni sono le medesime, quindi non riportiamo più di quanto già visto.

6.4.3 saveMySharedFiles

Il metodo `saveMySharedFiles` prende come parametro un oggetto di tipo *OrdinarynodeFiles*, `shared_files`, e "fotografa" il suo stato all'interno del file `shared.mk`.

Queste operazioni sono state molto semplificate da Java semplicemente dichiarando *Serializable* la classe da scrivere nel file. Quindi è stato sufficiente scrivere le poche righie, che presentiamo nel listato mostrato succes-

sivamente, per scrivere tutto il contenuto dell'oggetto *OrdinarynodeFiles*, in un dato momento, sul file.

```
1 FileOutputStream file_stream = null;
2 ObjectOutputStream object_stream = null;
3
4 try{
5   file_stream = new FileOutputStream(sh_files_save);
6   object_stream = new ObjectOutputStream(file_stream);
7
8   object_stream.writeObject(shared_files);
9 }
```

6.4.4 loadMySharedFiles

Il duale del metodo presentato sopra è `loadMySharedFiles` che prende in input un oggetto di tipo *NodeInfo*, `my_infos`, e restituisce l'*OrdinarynodeFiles* contenuto all'interno del file `shared.mk`.

Le affermazioni fatte sopra valgono anche in questo caso. Tutto è stato decisamente semplificato dal Java che si è occupato della fase a *basso livello* di lettura die dati dal file.

Non riportiamo il codice perchè è il duale di quello presentato prima.

Capitolo 7

Il package di grafica

La parte grafica del MiniKazaa è stata implementata con la grafica che mette a disposizione il linguaggio Java e con l' ausilio dello strumento di programmazione NetBeans. Nella parte che segue andiamo a vedere un pochino più di preciso cosa abbiamo usato per creare le interfaccie.

7.1 Il campo di testo

Il JTextField è un componente campo di testo, usabile per scrivere e visualizzare una riga di testo. Il campo di testo può essere editabile o no, il testo al suo interno è accessibile con getText() e modificabile con setText(). Ogni volta che il testo in esso contenuto cambia si genera un DocumentEvent nel documento che contiene il campo di testo. Se invece è sufficiente registrare i cambiamenti solo quando si preme INVIO, basta gestire semplicemente l' ActionEvent.

7.2 I bottoni

Quando viene premuto, un bottone genera un evento di classe ActionEvent. Questo evento viene inviato dal sistema allo specifico ascoltatore degli eventi per quel bottone. L'ascoltatore degli eventi deve implementare l' interfaccia ActionListener:

- può essere un oggetto di un'altra classe al di fuori del pannello;
- .. o può essere anche il pannello stesso nel quale (this).

Tale ascoltatore degli eventi deve implementare il metodo definito nella interfaccia actionListener void actionPerformed(ActionEvent ev); che gestisce l'evento, nel senso che reagisce all'evento con opportune azioni.

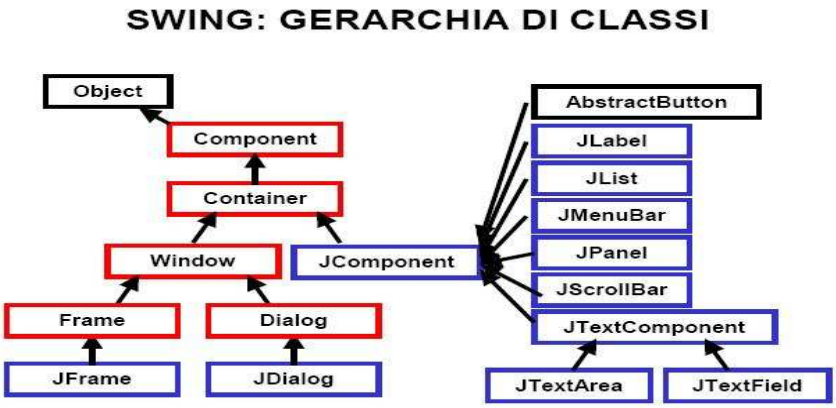


Figura 7.1: La grafica con le librerie Swing.

7.3 Gestore degli eventi

Una volta generato l'oggetto evento, questo viene inviato ad un oggetto ascoltatore degli eventi (event listener). L'event listener deve essere definito e creato da noi e deve essere associato al componente attivo, cosicché quando si genera un evento, la JVM sappia a chi inviare l'oggetto evento. L'event listener gestisce l'evento mediante un opportuno metodo, che non è altro che l'implementazione di un particolare metodo di un'interfaccia associata a tali tipi di eventi.

7.4 Le tabelle di Java

Una tabella è un componente che visualizza righe e colonne di dati. È possibile trascinare il cursore sui bordi delle colonne per ridimensionarle; è anche possibile trascinare una colonna in una nuova posizione. Le tabelle sono implementate dalla classe JTable. Uno dei suoi costruttori è mostrato di seguito: JTable(Object dati[][],Object intCol[]) dove dati è un array bidimensionale delle informazione da presentare e intCol è un array monodimensionale con le intestazione delle colonne. Ecco i passi per utilizzare una tabella in un frame:

- Creare un oggetto JTable
- Creare un oggetto JScrollPane dove l'argomento del costruttore specifica la JTable appena creata. In questo modo la tabella verrà aggiunta al pannello
- Aggiungere il pannello di scorrimento al pannello dei contenuti del

JFrame (per intendere quello ottenuto tramite il metodo getContentPane()della classe)

Un JScrollPane è un pannello di scorrimento che presenta un'area rettangolare nella quale si può vedere un componente. Se il componente ha dimensione maggiore del pannello vengono fornite barre di scorrimento orizzontali e/o verticali.

Capitolo 8

Scelte di progetto e cenni di teoria

8.1 Java Bean

Le JavaBean sono usate per incapsulare molti oggetti in un singolo oggetto (il bean), così da poter passare il bean invece degli oggetti individuali. Al fine di funzionare come una classe JavaBean, una classe comune di un oggetto deve obbedire a certe convenzioni in merito ai nomi, alla costruzione e al comportamento dei metodi. Le convenzioni richieste sono:

- La classe deve avere un costruttore senza argomenti.
- Le sue proprietà devono essere accessibili usando get, set e altri metodi (così detti metodi accessori).
- La classe dovrebbe essere serializzabile (capace di salvare e ripristinare il suo stato in modo persistente).
- Non dovrebbe contenere alcun metodo richiesto per la gestione degli eventi.

Utilizzando queste proprietà, i Java Bean sono risultati molto utili sia per creare strutture facilmente inviabili via rete, sia per utilizzare le classi XMLEncoder e XMLDecoder per scrivere e leggere Java Bean da file XML.

8.2 UML Logica

La logica del programma è visualizzabile in Figura 8.1. Il diagramma UML si riferisce un nodo di tipo SN, ma, visto che possiamo considerare i SN come degli ON con alcune funzionalità in più, questo diagramma vale anche per gli ON.

Possiamo vedere come ogni nodo abbia due tipi di componenti:

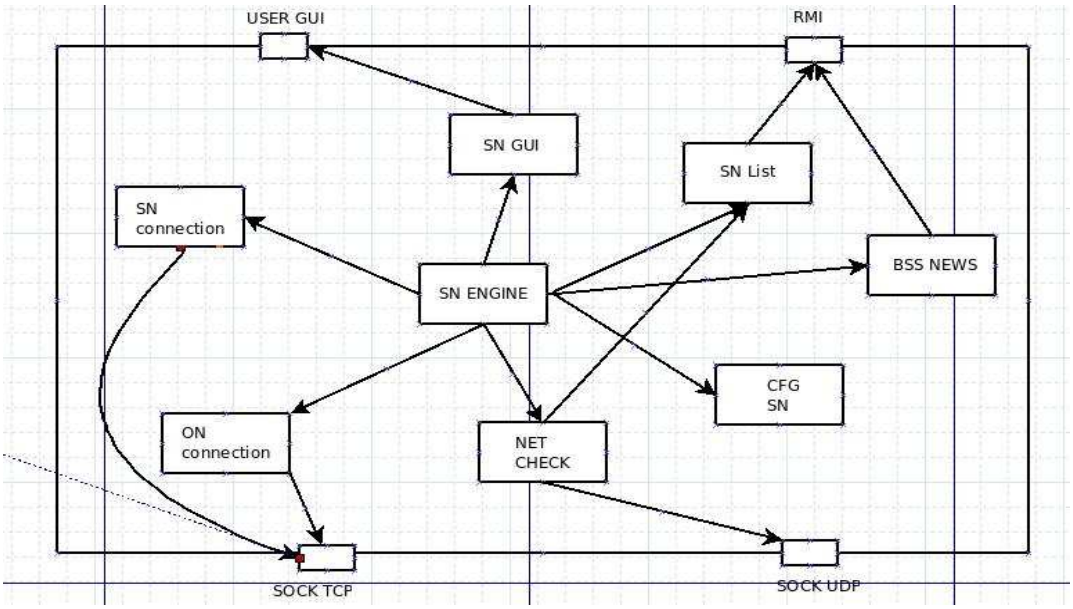


Figura 8.1: Il diagramma UML che rappresenta la logica di Mini-KaZaA.

- componenti logica interna;
- componenti di interfaccia con il mondo esterno.

Tutti questi componenti vengono gestiti da un motore centrale che richiama opportunamente le strutture dati utilizzate e i thread che devono svolgere i vari compiti.

8.3 Classe Wrapper per il socket

Il fine del Wrapper è di fornire una soluzione astratta al problema dell'interoperabilità tra interfacce differenti. Il problema si presenta ogni qual volta nel progetto di un software si debbano utilizzare sistemi di supporto (come per esempio librerie) dotati di interfaccia non perfettamente compatibile con quelle richieste da applicazioni già esistenti. Invece di dover riscrivere parte del sistema, oneroso e non sempre possibile se non si ha a disposizione il codice sorgente, può essere comodo scrivere un Adapter che faccia da tramite tra le diverse interfacce, rendendole così compatibili.

Il wrapper inoltre ci ha consentito di poter inizializzare un oggetto che ci serviva all'interno di un *sotto-thread* all'interno dell'*engine* per poi modificarne il contenuto all'interno dei vari sotto-thread non appena si rendevano disponibili le informazioni per completarne la struttura.

8.4 Il protocollo TCP

Il TCP nacque nel 1970 come frutto del lavoro di un gruppo di ricerca del dipartimento di difesa statunitense. I suoi punti di forza sono l'alta affidabilità e robustezza. Il protocollo TCP serve a creare degli stream socket, cioè una forma di canale di comunicazione che stabilisce una connessione stabile fra due stazioni, in modo che queste possano scambiarsi dei dati.

Il servizio offerto da TCP è il trasporto di un flusso di byte bidirezionale tra due applicazioni in esecuzione su host differenti. Il protocollo permette alle due applicazioni di trasmettere contemporaneamente nelle due direzioni, quindi il servizio può essere considerato Full Duplex anche se non tutti i protocolli applicativi basati su TCP utilizzano questa possibilità. Il flusso di byte viene frazionato in blocchi per la trasmissione dall'applicazione a TCP (che normalmente è implementato all'interno del sistema operativo), per la trasmissione all'interno di segmenti TCP, per la consegna all'applicazione che lo riceve, ma questa divisione in blocchi non è per forza la stessa nei diversi passaggi. TCP è un protocollo orientato alla connessione, ovvero prima di poter trasmettere dati deve stabilire la comunicazione, negoziando una connessione tra mittente e destinatario, che viene esplicitamente chiusa quando non più necessaria. Esso quindi ha le funzionalità per creare, mantenere e chiudere una connessione. TCP garantisce che i dati trasmessi, se giungono a destinazione, lo facciano in ordine e una volta sola. Questo è realizzato attraverso vari meccanismi di acknowledgment e di ritrasmissione su timeout. TCP possiede funzionalità di controllo di flusso e di controllo della congestione sulla connessione, attraverso il meccanismo della finestra scorrevole. Questo permette di ottimizzare l'utilizzo della rete anche in caso di congestione.

8.5 Il protocollo UDP

A differenza del TCP, l'UDP è un protocollo di tipo connectionless, inoltre non gestisce il riordinamento dei pacchetti né la ritrasmissione di quelli persi, ed è perciò generalmente considerato di minore affidabilità. È in compenso molto rapido ed efficiente per le applicazioni leggere o time-sensitive. Ad esempio, è usato spesso per la trasmissione di informazioni audio o video. Dato che le applicazioni in tempo reale spesso richiedono un ritmo minimo di spedizione, non vogliono ritardare eccessivamente la trasmissione dei pacchetti e possono tollerare qualche perdita di dati, il modello di servizio TCP può non essere particolarmente adatto alle loro caratteristiche. L'UDP fornisce soltanto i servizi basilari del livello di trasporto, ovvero:

- multiplazione delle connessioni, ottenuta attraverso il meccanismo delle porte

- verifica degli errori mediante una checksum, inserita in un campo dell'intestazione del pacchetto.

mentre TCP garantisce anche il trasferimento affidabile dei dati, il controllo di flusso e il controllo della congestione. L'UDP non tiene nota dello stato della connessione, dunque ha rispetto al TCP informazioni in meno da memorizzare. Un server dedicato ad una particolare applicazione che scelga UDP come protocollo di trasporto pu supportare molti pi client attivi.

8.6 Remote Method Invocation

In informatica, e in particolare nel contesto del linguaggio di programmazione object-oriented Java, Remote Method Invocation (invocazione remota di metodi) o RMI una tecnologia che consente a processi Java distribuiti di comunicare attraverso una rete. Questa tecnologia include una API (application programming interface) il cui scopo esplicito quello di rendere trasparenti al programmatore quasi tutti i dettagli della comunicazione su rete. Essa consente infatti di invocare un metodo di un oggetto remoto (cio appartenente a un diverso processo, potenzialmente su una diversa macchina) quasi come se tale oggetto fosse locale (ovvero appartenente allo stesso processo in cui viene eseguita l'invocazione). In questo senso, la tecnologia RMI può essere ricondotta, da un punto di vista concettuale, all'idea di chiamata di procedura remota riformulata per il paradigma object-oriented (in cui, appunto, le procedure sono sostituite da metodi). L'utilizzo di un meccanismo di invocazione remota di metodi in un sistema object-oriented comporta notevoli vantaggi di omogeneit e simmetria nel progetto, poich consente di modellare le interazioni fra processi distribuiti usando lo stesso strumento concettuale che si utilizza per rappresentare le interazioni fra i diversi oggetti di una applicazione, ovvero la chiamata di metodo.

8.7 Utilizzo dei ThreadPool

In molte applicazioni vengono creati thread il cui stato è quasi sempre sospeso, in attesa che si verifichi un evento. Altri thread potrebbero entrare in uno stato di inattività ed essere attivati solo periodicamente alla ricerca di informazioni sullo stato di una modifica o di un aggiornamento. Il polling del thread consente di utilizzare i thread in modo pi efficiente fornendo all'applicazione un pool di thread di lavoro gestiti dal sistema. Un unico thread consente di controllare lo stato di varie operazioni di attesa in coda nel pool di thread. Quando un'operazione di attesa viene completata, la funzione di callback corrispondente viene eseguita da un thread di lavoro contenuto nel pool di thread.

Nel progetto Mini-KaZaA abbiamo usato i ThreadPool, e in particolare *ThreadPoolExecutor*, nel task che si occupava di elaborare le richieste

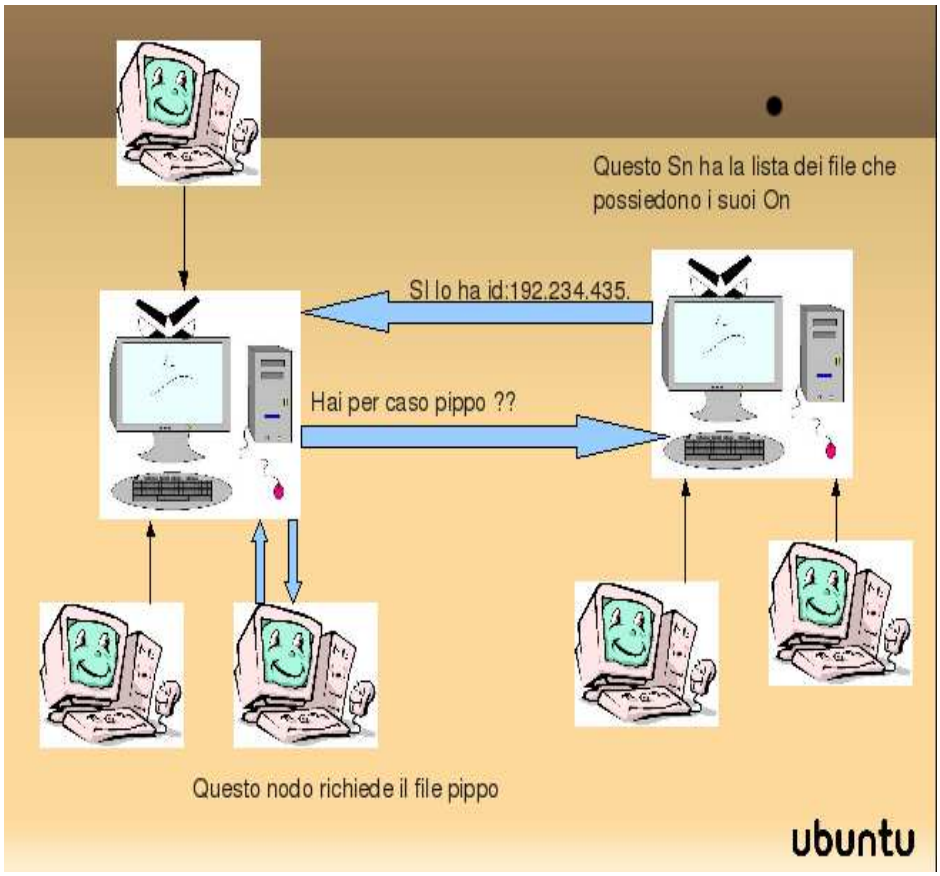


Figura 8.2: Richiesta file.

provenienti dal socket TCP. Qui infatti si possono creare un numero elevato di thread, visto che a ogni richiesta da elaborare ne viene assegnato uno. Per ottimizzare le risorse, quindi, stato conveniente demandare a Java la gestione della schedulazione dei thread utilizzando questo particolare costrutto.

8.8 Come funzionano le query

Per aiutare a comprendere il meccanismo che st dietro al nostro sistema di comunicazione per la ricerca di un file abbiamo creato una semplice vignetta che facilita il tutto. Analiziamo insieme come avvengono le comunicazioni. Come possiamo notare dalla prima figura un utente che ricerca un file effettua la richiesta al SuperNode al quale connesso direttamente. Il SuperNode possiede la lista di tutti i file in possesso dei suoi OrdinaryNode, se il file richiesto appartiene a quelli in suo possesso risponde al mittente con l' indirizzo del nodo che possiede in file. Se non lo possiede invece chiede

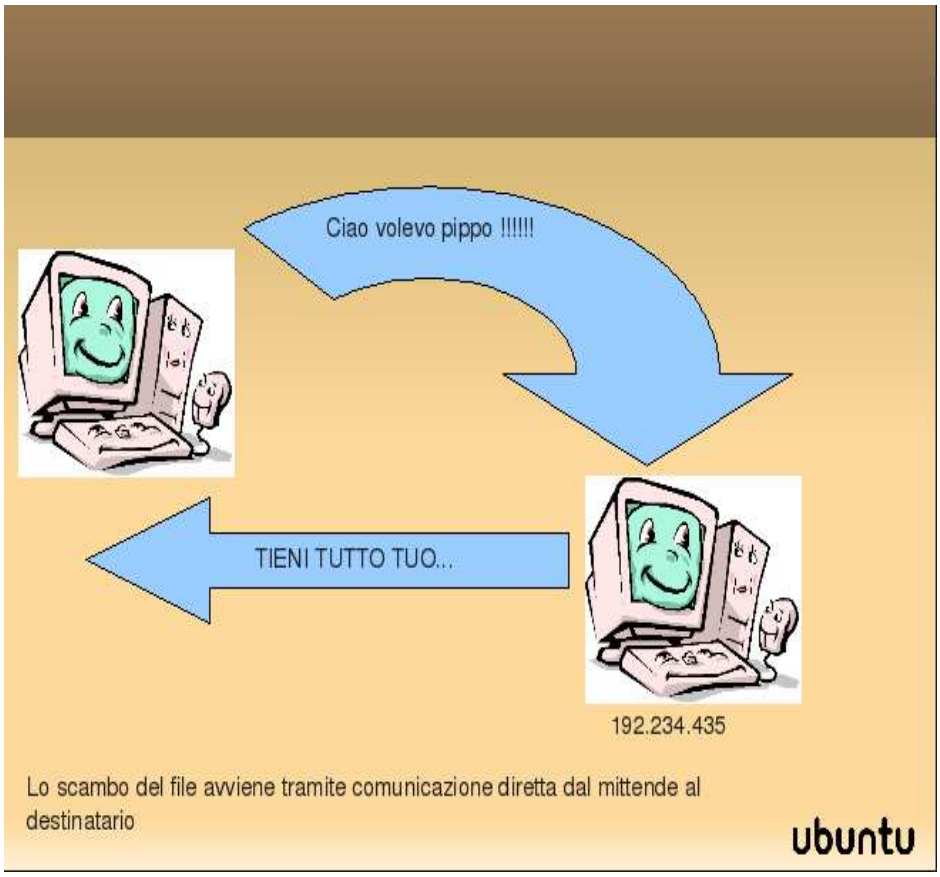


Figura 8.3: Download.

agli altri SuperNode ai quali lui collegato nella rete e attende la risponsta. La risposta che gli viene data comprende oltre a bit di controllo inseriti dai programmatori per verificare che la query che ritorna sia quella giusta e non una che abbia sbagliato percorso nella rete, l' indirizzo del nodo che possiede il file. Tale risposta viene girata al nodo che inizialmente ha richiesto il file. A questo punto come viene mostrato nella seconda figura avviene la comunicazione diretta tra il richiedente ed il possessore del file. Comunicazione che avviene tramite un protocollo TCP. Interessante sottolineare che stata rispettata la richiesta del testo di avere un percorso a ritroso delle query dal mittente al destinatario.

Una cosa che pu sicuramente saltare all'occhio guardando nel codice come vengono smistate le query che non appena viene ricevuta una query di richiesta il Super Node invia due risposte, una contenente i riferimenti ai file che condivide il SN e una con le informazioni riguardanti i file che gli ON che sono connessi all'SN condividono. Queste due risposte vengono inviate indipendentemente dal loro contenuto, quindi pu capitare, anzi molto

spesso capita, che vengano inviate delle risposte vuote. Questo sicuramente aumenta il traffico di rete ma impedisce che rimangano per la rete risposte pendenti con conseguenti malfunzionamenti dei vari client.

8.9 La classe di log

Per salvare i codici di errore che il client Mini-KaZaA pu sollevare durante la sua esecuzione, ci siamo avvalsi di una classe gi presente nelle API Java: Logger. Questa classe interagisce direttamente con i logger del componente che gestisce l'applicazione, nel nostro caso il terminale Linux o il prompt DOS. Una volta passato il comando il messaggio di log viene salvato utilizzando la procedura standard di sistema.

Appendice A

Manuale d'uso

Affrontiamo ora la parte piu' pratica del progetto.

A.1 Installazione

Anzitutto e' importante sapere che per poter utilizzare il prodotto da noi creato e' necessaria almeno una rete LAN o una rete INTERNET. Un computer della rete deve fornire il servizio di bootstrap Server e quindi deve essere avviato come tale. All'avvio del programma l' utente può decidere se essere un SuperNode o un OrdinaryNode. Ricordiamo che questa decisione non sarà possibile modificarla nel seguito. A seconda della scelta appariranno a video le schermate da utilizzare ed il programma è pronto a funzionare.

A.2 Primo avvio

Per semplicità spieghiamo il primo avvio supponendo di lanciae la nostra applicazione in modalità SuperNode. Le cose che verranno scritte nella parte sottostante sono analoghe anche se si lanciasse la modalità OrdinaryNode. All'avvio appare la schermata mostrata in Figura A.1. Cliccando sul tasto che corrisponde alla creazione di supernode appare un nuovo pannello mostrato in Figura A.2. E' necessario ora riempire i campi vuoti per la configurazione del peer. Inseriamo quindi un nikname che preferiamo, il numero della porta di comunicazione, l' indirizzo del bootstrap server, il massimo numero di nodi al quale verrà inviata la richiesta di un file ed infine il time to live. Il settaggio del time to live lo sconsigliamo ad utenti poco esperti dato che potrebbe, se impostato male, rendere l' applicazione poco funzionante. Consigliamo inoltre ad utenti esperti che intendano settare il time to live di farlo in modo oculato. Per l'applicazione utilizzata su reti di grosse dimensioni è preferibile un time to live maggiore di quello impostato di default, mentre al contrario se si lancia l' applicativo su reti di piccola

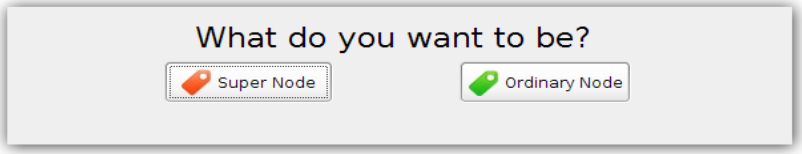


Figura A.1: Pannello di decisione SuperNode o OrdinaryNode

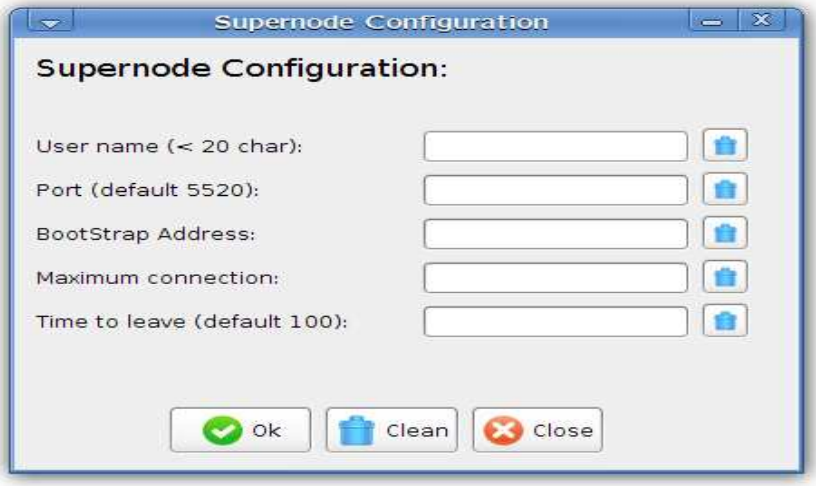


Figura A.2: Il pannello di configurazione del SuperNode.

dimensione è preferibile un time to live decisamente più piccolo di quello di default. Riempiti tutti i campi correttamente e premuto il tasto ok, se tutto è andato a buon fine apparirà ora la finestra in figura A.3 In caso di campi compilati in maniera non corretta il client provvede a segnalare l'errore con un'apposita *dialog*. L' installazione è terminata, il programma è funzionanante non vi resta che divertirvi con Mini-KaZaA.

A.3 Come funziona

Bene se siamo di fronte al programma correttamente installato ora non ci rimane che usarlo. Se abbiamo deciso di essere SuperNode noi vogliamo oltre che poter cercare e scaricare sulla rete fornire il nostro servizio alla comunica globale. Se abbiamo decido di essere OrdinaryNode noi vogliamo usufruire solamente del servizio di ricerca e download che offre Minikazaa.

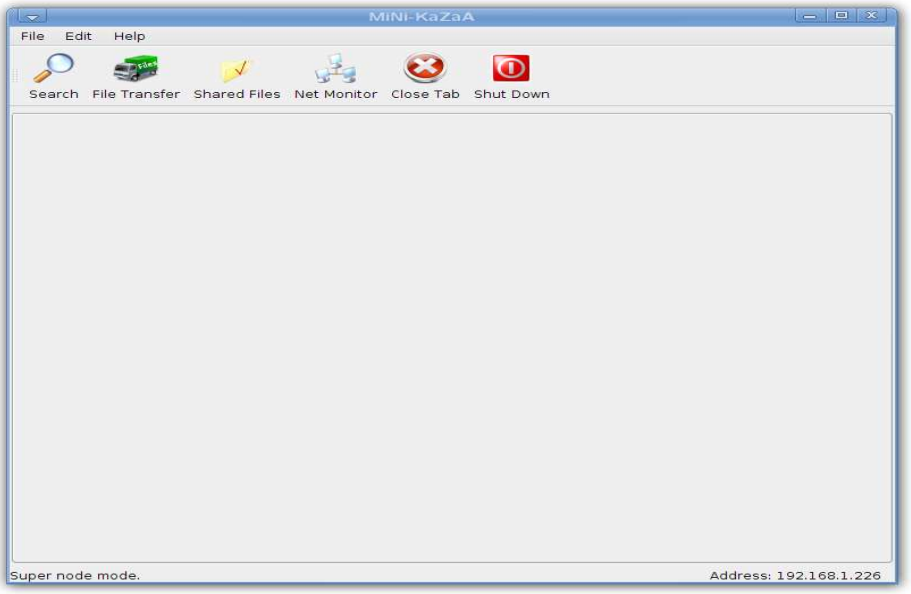


Figura A.3: L' interfaccia principale dell' ordinary node.

A.3.1 Cercare e scaricare un file

Per cercare e scaricare un file, è necessario inserire nella casella vuota il titolo del file che interessa trovare nella rete. La nostra casella di ricerca interpreta la parola chiave inserita della casella come una $*esempio*$. Questo significa che tutti i file che contengono nel titolo la parola esempio verranno restituiti come canditati per lo scaricamento. Nella tabellina sotto il campo dove è stata inserita la parola ora appariranno i riferimenti ai file che sono presenti sulla rete, i quali possono essere facilmente scaricati cliccando su comodo pulsate download Il download che parte automaticamente è visibile e controllabile nella sezione chiamata FILETRANSFERT.

A.3.2 Aggiungere un file nella lista dei file condivisi

Come tutti i servizi di file sharing minikazaa permette di aggiornare la lista dei file che è possibile scaricare. Cliccando il bottone di Sharing viene caricato automaticamente un pannello che indica la lista dei file che sono stati messi in condivisione nella rete. E' inoltre possibile aggiungere file alla lista cliccando su tasto add e rimuoverne altri cliccando sul tasto remove.

A.3.3 Controllare lo stato dei download

Cliccando la sezione FileTranfert si apre un pannello che permette di controllare lo stato dei download dei file che desideriamo scaricare. Nella

tabella che si è appena aperta è possibile vedere il nome del file, la dimensione e il progressivo stato di scaricamento nella barra.

A.3.4 NetMonitor

Questo tasto risulta disabilitato se avete deciso di essere OrdinaryNode e invece è attivo se avete deciso di essere SuperNode. Prendiamo ora il caso di essere SuperNode, e di avere aperto il pannello corrispondente a NetMonitor. Nel suddetto pannello appare la situazione della rete al momento, piu' precisamente la lista dei SuperNode ai quali si è connessi.

A.3.5 Chiudere le schede

I restanti tasti visibili nella schermata permettono di chiudere le schede aperte in precedenza.

A.4 Consigli degli autori per l' utilizzo

Al fine di rendere un migliore servizio possibile diamo alcuni consigli per gli utilizzatori del nostro software. Consigliamo quindi ad utilizzatori dotati di grosse quantità di file, ma soprattutto di grandi capacità di linea di offrire il loro servizio alla comunità globale e quindi di avviare l' applicazione come supernode. Consigliamo invece agli utenti di piccole dimensione, che hanno difficolta ed un valore di upload limitato di preferire la modalità da ordinaryNode.