Progetto
Laboratorio Programmazione di rete
Mini-KaZaA

Andrea Di Grazia, Massimiliano Giovine

Anno Accademico 2008 - 2009

Indice

Capitolo 1

Introduzione

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.

[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.

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.

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 main 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)
6   UnicastRemoteObject.exportObject(bss, 2008);
```

[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.

```
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.

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

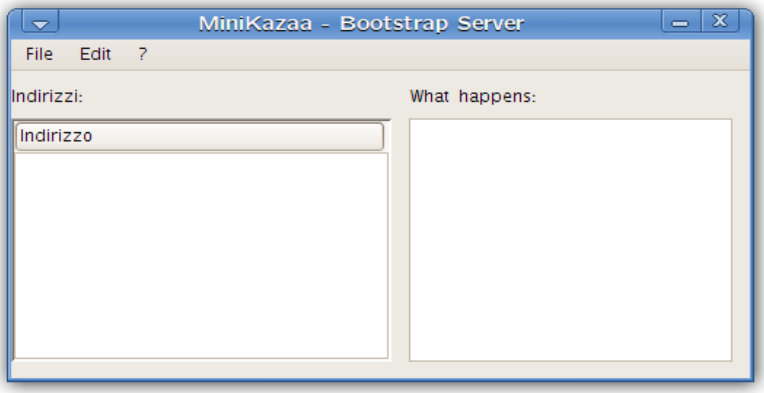

Figura 2.1: Interfaccia grafica del Bootstrap server

```
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 univoco 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.

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, che 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 loogicamente 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 a un supernodo per funzionare

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 particolareggiata 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 di queste classi.

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 ci sono 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 4.1.5.

La classe `Answer.java` viene utilizzata

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 del file da scaricare[1].

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] I download così come le ricerche vengono effettuati mediante l'hash univoco md5 che tratteremo nella Sezione 6.1

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 4.2.5

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 formdi cui parleremo nella Sezione 3.8. 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 5.2. 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[2]

Il metodo in questo listato

[2]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      }
```

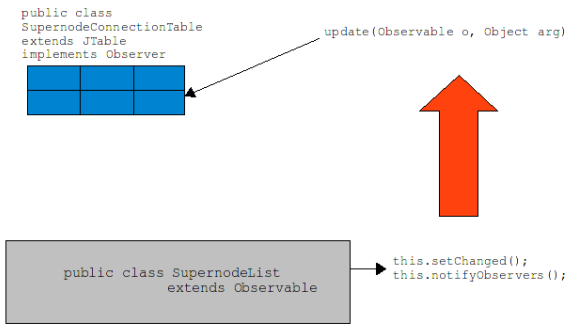

Figura 3.1: Paradigma Observable - Observer

```
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 paramtro 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.

Mini-KaZaA ha 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,
```

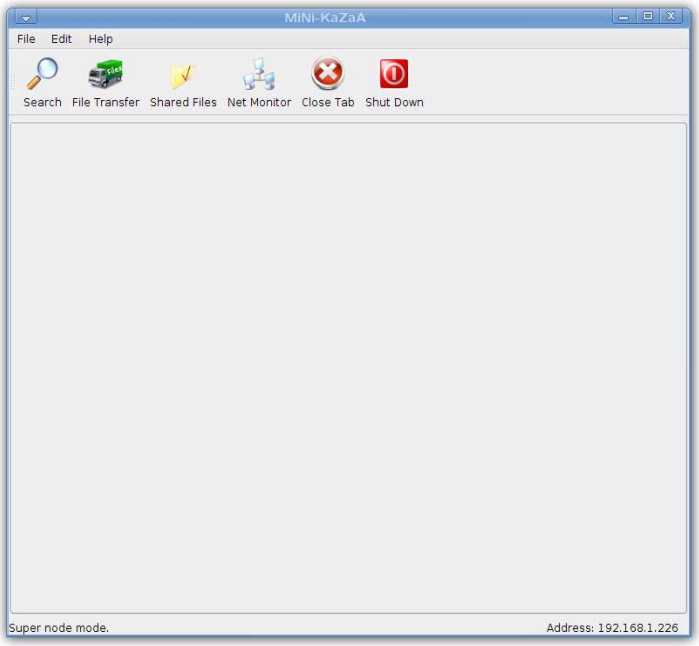

Figura 3.2: L'interfaccia grafica principale del client.

```
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[3]

3.8 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:

[3]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

```
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[4]

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.

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

Capitolo 4

Ordinary Node

4.1 Le classi del package lpr.minikazaaclient.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 lpr.minikazaaclient.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 OrdinarynodeDownloadMonitor.java

Parliamo di una classe che serve per indicizzare

4.1.2 OrdinarynodeFoundList.java

4.1.3 OrdinarynodeFriendRequest.java

4.1.4 OrdinarynodeQuestionList.java

4.1.5 OrdinarynodeFiles.java

4.2 Il cuore di un Ordinary Node

4.2.1 Engine

4.2.2 ON in ascolto sul socket TCP

4.2.3 ON e RMI

4.2.4 Scelta del SN al quale connettersi

4.2.5 Lo scambio di file

4.2.6 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 3.8

Capitolo 5

Super Node

5.1 L'interfaccia per le callback

5.2 Smistamento delle query

Capitolo 6

Il package Util

6.1 Calcolo dell'md5

Capitolo 7

Il package di grafica

7.1 Le tabelle di Java e le custom cells

Appendice A

Manuale d'uso