**PEER TO PEER MUSIC TRANSFER**

***Course: Internetworking Protocol***

*Lecturer: Dr. Nguyen Tuan Nam.*

*Teaching Assistants:*

* *Le Quoc Hoa*
* *Nguyen Thanh Quan*

*Student:*

* *Vo Minh Thanh: 1251040*
* *Nguyen Ba Tuan Vu: 1251057*

# Introduction

The project developed a simple peer to peer application to transfer music file between nodes using UDP connection. The application is written in Java languages, providing with some key features:

* Using the peer to peer model to connect nodes and transfer file without the existing of server and client.
* Transfer file by UDP connection with no losing data guarantee.
* Automatic finding nodes connecting to system every 3 seconds.
* Manage the list music file.
* Manage the list connecting host.
* Play music.
* Search all music file between nodes.

# Structure of peer and the way of connecting peers:

## Structure of peer:

Every peer contains an IP and the port which it connected in. For the efficiency of program, we also store many further information in a node:

* + A list contains all the connected other peers.
  + A list contains the information of all added file.
  + A route interface helps to find the route between the current node to the others one
  + A handle interface helps to handle with all coming actions.
  + ….

## The way of connecting peers:

* Unlike the requirement which require a file nodes.txt for storing the initial connected peers before running, we wrote Depth First Search algorithm to finding all the connected peers in the system:
  + At first, node 1 is connected in isolated way that means node does not know any others connected node.
  + Input the node 2 which we want to connect to.
  + Establish the connection between two nodes.
  + The node 1 send request the 2 to send the list of node 2’s connected nodes.
  + Node 1 runs recursively and establish the connection with the given nodes.
* This is all the scenario for connecting peers:

|  |  |  |  |
| --- | --- | --- | --- |
| Index | Node 1 | Node 2 | Index |
| 1 | Input the port to connect node 1 |  | 1 |
| 2 |  | Input port to connect node 2 | 2 |
| 3 | Input the IP and port of node 2 to connect to |  | 3 |
| 4 | Node 1 send message to node 2 |  | 4 |
| 5 |  | Node 2 receives message from node 1, add node 1 to list, send reply | 5 |
| 6 | Node 1 receives the message (Reply), add node 2 to list |  | 6 |
| 7 | Node 1 send message to node 2 to get the list connected nodes from node 2 |  | 7 |
| 8 |  | Node 2 receives message, send back the list to node 1 | 8 |
| 9 | Node 1 receives the message and gets the list |  | 9 |
| 10 | With every node in list, repeat the step 3 |  | 10 |
| 11 |  | Node 2 send request node 1 to get the list from node 1 | 11 |
| 12 | Node 1 receives message, send back list to node 2 |  | 12 |
| 13 |  | Node 2 receives the message and gets the list | 13 |
| 14 |  | With every node in list, repeat the step 5 | 14 |

# Finding file between nodes

Every peer contains information of all the added file in the system so that we can find it right immediately.

How to add file:

* Node input the name of added file.
* Node broadcast to all the connected nodes to inform the name of new files.
* Other nodes get inform and added the name of file into their own list.

Scenario of finding nodes:

|  |  |
| --- | --- |
| Index | Node |
| 1 | Node input the name of file |
| 2 | Search in the list and return the connected node containing the file |

# Download file

Using the UDP connection to transfer file between two nodes with no losing data guarantee

Scenario of downloading file:

|  |  |  |  |
| --- | --- | --- | --- |
| Index | Node 1 | Node 2 | Index |
| 1 | Node 1 inputs the name of file which want to search |  | 1 |
| 2 | Returns the name of file, includes the address of node 2 |  | 2 |
| 3 | Node 1 sends request to get file on node 2 |  | 3 |
| 4 |  | Node 2 receives the request | 4 |
| 5 |  | Node 2 gets the data of file and send data back to node 1 | 5 |
| 6 | Node 1 receives data and save back to folder |  | 6 |

# Structure of message:

In the application we developed here, there are 9 type of message exchanged between nodes. Each message type is identified by a 4-character string and may be accompanied with additional message data:

* NAME: Requests a peer to reply with its "official" peer id.
* LIST: Requests a peer to reply with the list of peers that it knows about.
* JOIN pid host port: Requests a peer to add the supplied host/port combination, associated with the node identified by pid, to its list of known peers.
* QUER return-pid key ttl: Queries a peer to see if the peer has any record of a file name matching key. If so, send a RESP message back to the node identified by return-pid; if not, propagate the query to all known peers with a decreased ttl (time-to-live) value, unless ttl is already 0.
* RESP file-name pid: Notifies a peer that the node specified by pid has a file with the given name.
* FGET file-name: Request a peer to reply with the contents of the specified file.
* QUIT pid: Indicate to a peer that the node identified by pid wishes to be unregistered from the P2P system.
* REPL ...: Used to indicate an acknowledgement of the other message types above or to send back results of a successful request.
* ERRO msg: Used to indicate an erroneous or unsuccessful request.

**Handle message**:

**NAME**

When a peer receives a NAME message (and no additional data), it simply responds by sending back (in a REPL message) its peer id, of the form "host:port".

**LIST**

When a peer receives a LIST message (and no additional data), it responds by first sending a REPL message with the number of id's in its list of known peers. Then it sends that many additional REPL messages, including in the data portion of each message a string with the peer id, host address, and port number (all three separated by whitespace) associated with each known peer in its list.

**JOIN**

A JOIN message is accompanied with additional data specifying a sending peer id, IP address, and port number. The JOIN message is used to request a peer to insert the sending node into its list of known peers. If the peer's list is already full, or the specified id is already in the list (or incorrect arguments are specified with the JOIN message), than an ERRO message is sent back in response. Otherwise, the remote node's information is added to the list and a REPL acknowledgement is sent.

**QUER**

The QUER message is the most complex to handle in this application. Upon receipt of this type of message, if the correct number of additional data is provided (return-peer-id, key, and ttl value), then the peer responds simply with an acknowledgement REPL message; otherwise an ERRO message is returned. Before exiting the handler routine for this message type (and closing the socket connection), however, a separate thread is started that actually processes the query information.

**RESP**

A peer will receive a RESP message in response to a QUER message that it had previously sent out to others in the network. The RESP message includes a file name and the peer id of the node owning a copy of the file. Upon receiving a RESP message, a peer does not need to send any reply; if the file name is not already present in its file list, an entry will be added to the list indicating that the file may be found at the specified node.

**FGET**

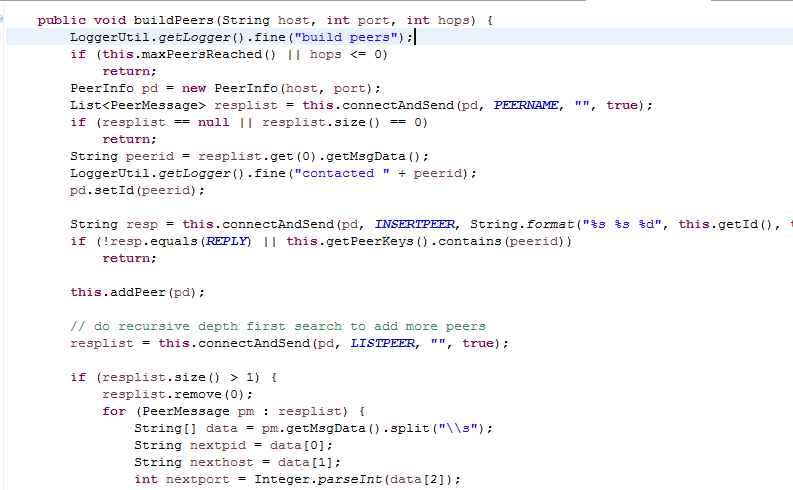
The data of an FGET message consists of a file name. If the peer receiving the FGET message does not own a copy of the specified file or if it is not readable for some reason, an ERRO reply is sent. Otherwise, the entire file is sent as the payload of a REPL message.

**QUIT**

A QUIT message, including a peer id in the data of the message, tells a peer to remove the specified node's information from its list of known peers as that node is preparing to leave the network. Note that, as is the case in most P2P protocols, node may also leave the network unexpectedly, so some sort of "stabilization" routine should be run by every peer node to periodically update its list of peers.

# Some crucial functions:

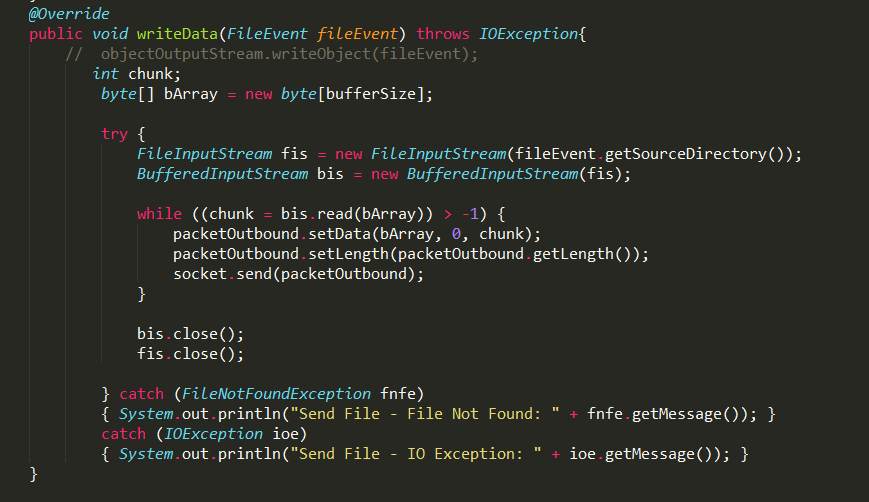
## Build peers list:

Run recursively to find all the connected peers: 

## Download file:

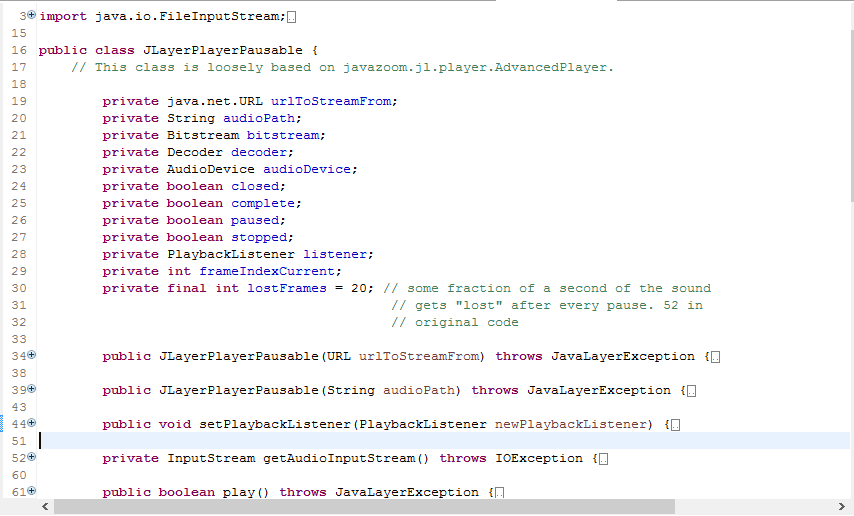


## Send file:

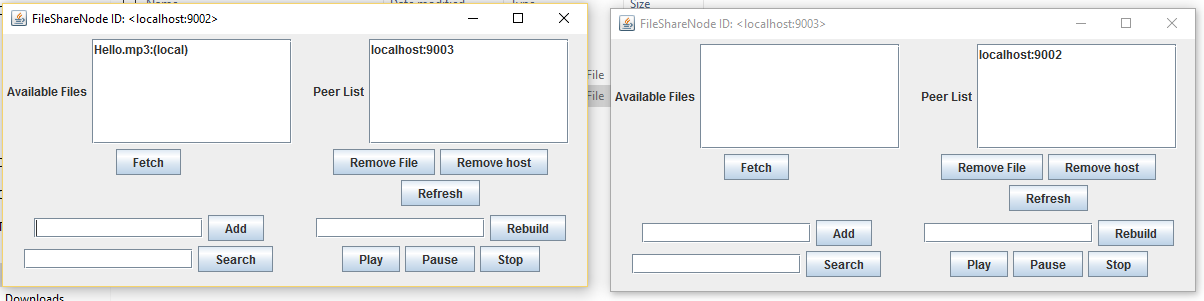


## Play music:

Application use the JLayer library which is an open source library for playing mp3 music and adapt from the tutorial “Playing an MP3 from Java Using JLayer” of author “This Could Be Better” website



# UI Interface:



Detail buttons:

* Rebuild: find the route between 2 nodes. Input the node which we want to connect: <IP:Port>
* Refresh: Update the peer list
* Remove host: remove a peer out the list peer
* Remove file: remove a music file out the list
* Add: add a music file to the list. Input the music file name: <music\_file\_name.mp3>
* Search: search a music file in the system. Input the name of music file: <music\_file\_name.mp3>
* Play/Pause/Stop: action button to play/pause/stop a specific music

# Reference:

* Peer-to-Peer Programming: <http://cs.berry.edu/~nhamid/p2p/index.html>
* Playing an MP3 from Java Using JLayer: <https://thiscouldbebetter.wordpress.com/2011/06/14/playing-an-mp3-from-java-using-jlayer/>