Team Peerless

A Distributed Peer-to-Peer File Sharing System

Peerless-Final-Documentation

Hee Won Lee
Nachiketa Gadiyar
Deepak Angathil
Hemant Imudianda

1. Document Revisions

Revision	Date	Author(s)	Comments
1.0	10/24/2009	Hee Won Lee	Initial Creation Requirement, Architecture, Project Organization added.
1.1	10/26/2009	Nachiketa Gadiyar	Introduction, Basic Project Implementation, Packet Structure added
1.2	10/26/2009	Deepak Angathil	Registration, Login/Logout added
1.3	10/26/2009	Hemant Imudianda	Signature Generation, Signature Comparison and Ranking added
1.4	10/28/2009	Hee Won Lee	Editing
2.0	12/14/2009	Hee Won Lee, et al	Final Version

2. Table of Contents

1. Introduction

This report describes the design and implementation of a commercial-quality distributed Peer-to-Peer file sharing system named "DGOOGLE", in which peers are capable of efficiently searching for files in a system using string variables as inputs. Java is our language of choice for implementing this project. All the peers use well known port numbers for communication.

The proposed system will implement the following two architectures :

(1) Centralized
(2) Distributed

1.1. Centralized Architecture

The centralized architecture has the following features:-

(1) Text files are contributed to the server by peers.

(2) When a peer needs a file, it has to send digested strings, containing file information to the bootstrap server.

(3) A queried file will be searched by the server in its database of file digests, and information such as file name, abstract, and IP addresses of the peers where the files are located will be send back to the requesting peer. The search will be ranked.

(4) Based on the information sent by the server, the requesting peer can directly download the file from the peer which has the file.

1.2. Distributed Architecture

The distributed architecture has the following features:-

(1)A peer will login into the boot strap server to find the list of peers which are active.

(2)A requesting peer has to send digested strings, containing file information to his fellow peer.

(3) The fellow peer searches its database and sends back information such as as file name, abstract, and IP Addresses of the peers where the files are located to the requesting peer. The search will be ranked.

(4) The peer who received the query then forwards the request to another fellow peer to look up his database and send back any information it has about the queried file back to the requesting peer. Such forwarding will continue for a decided number of times.

1.3. Other Key Features

Along with the above core features our project will also provide the following add on features:

(1) Performance Evaluation Results, which will indicate which architecture is better, centralized or distributed and why.

(2) User interface at each peer for very user friendly interaction with the system.

2. System Requirements

The following table shows system requirements. Our system consists of three tiers: Server, Peer, and Super Peer. Each tier has its own requirements. These are shown in Figure 1. After developing Server, Peer, and Super Peer, our system will be evaluated for centralized vs. distributed architecture. Evaluation is also one of the requirements.

	Level 1	Level 2
Server	Registration Provider	
	Loigin/Logoff	
	Session Management	Timeout
	Entry: ID, email address, User ID, password	
	password: AES(Advanced Encyrption Standard) hashing	
	login: ID, password	
	Remind users of password forgotten	Birth place,Best friend name, mother's maiden name
	Data: File name, short abstract, file size, IP address, digest of the file	
	Peers query searches for file digests in its own database	
	In distributed implementation, the server relays the query to other peers	
	Maintain digest information that comes from peers	
	Search files with the digest information	
	Provide the requesting peer with search service for digested files	(1) A list of of matching files (2) IP addresses of peers storing the files
	Digest the abstract of each file	
	Ranking methods -> research, compare them	
Peers	Maintains files in a separate directory	
	Communication among peers and the server	
	Registration	
	Login/logoff	
	Advertise its file to the bootstrap server	

	Search for files in the server	
	Search for files in other peers	
	Provide other peers with search service for files	
	Search text is digested	
	Provide the server with the digest	
	Request other peers to send a list of matching files, which received from server	
	Request the server to send active IP addresses periodically	
Super Peer	Maintain information about the files and links to peers storing the files	
	A method to distribute the above information --> paper 2	
	Provide search service for digested files with peers	(1) A list of of matching files (2) IP addresses of peers storing the files
	Forward the query to other peers	
Performance Evaluation	Centralized vs distributed architecture	
	(1) efficiency: time to conduct popular vs unpopular queries	
	(2) accuracy: - % of correct results about queries - % of false positive - % of false negative	
	(3) accuracy of ranking mechanism	

Figure 1. Requirements Specification

3. Design

3.1. Basic Project Implementation

We are using the Apache MINA network application framework for implementation of the server and client. APACHE MINA is a networking library, which operates above the socket layer and provides API over various transport level protocols such as TCP/IP and UDP/IP. The APACHE MINA network library provides various functions such as, functions for creating new sockets, functions for binding sockets to IP/Port Address and functions to bind a filter to the socket based on the type of encryption required for communication. Following are the details about the client and server we have designed.

1.1.1 SERVER

The server code creates a socket and binds it to the port address its listening on, and the server IP address. It also binds a filter with the socket, which decides the encryption method that would be used for communication between the client and server. It then places the socket on passive mode, where the socket listens to incoming connections. The moment an incoming connection is intercepted, the listening socket creates another socket and binds the remote peer's IP/Port Address and the server IP/Port Address to the that socket, in other words a new thread is created to deal with each connection. Also to the basic server code we have added additional code to facilitate handling new connections, registration of new connections, file searching and other features required by the project.

1.1.2 CLIENT

The client creates a socket (server connection request) and binds the server IP address and port address with it to connect to the server. We have designed different packet structures in the client code based on the type of communication required between the client and server. At the server, we have added code for decoding packets and treatment of the packets based on the commands they contain. This will be explained in detail further in the report.

The report now continues to describe the various features, we have implemented or we are in process of implementing to the system under design.

3.2. System Architecture

Figure 2 shows our overall system architecture. In the server, Sever Acceptor takes charge of low-level communication functions. The functions for thread pool and logging are provided in the form of filters. Above the filters, Protocol Handler handles our designed protocols. In addition, we will use file systems for our data storage. In the client, Client Connector manages all the low-level communication functions. Client Handler handles our protocols. Above that, there are specific dialog windows.

We will use TCP for communication between the server and the peer. However, when we search a file from other peers or super peers, we will use UDP. For file transfer, we will use TCP.

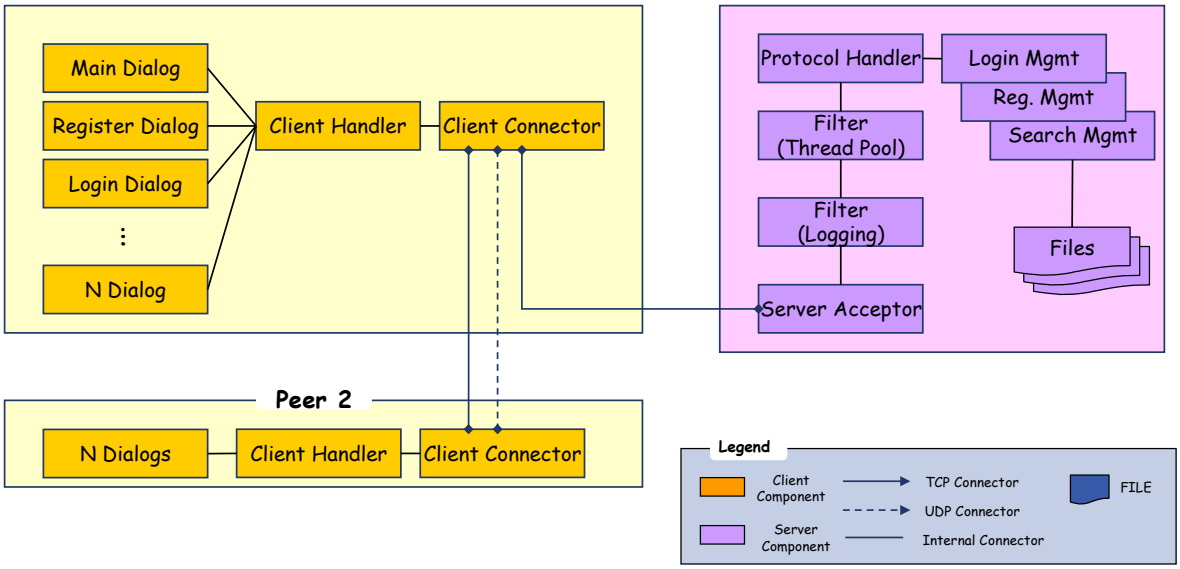


Figure 2. Component and Connector View of Peerless System Architecture

3.3. Component Diagram

Figure 3 shows the component diagram of our system.

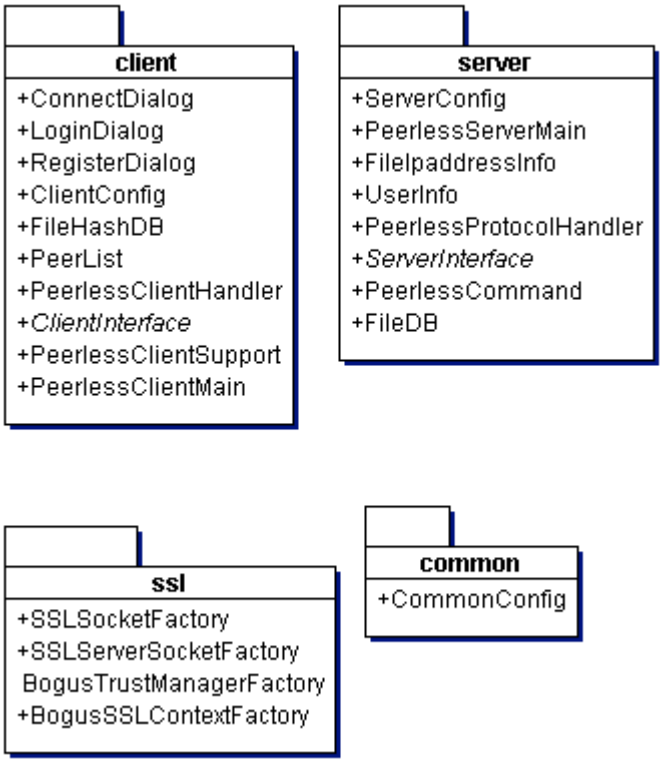


Figure 3. Component Diagram

3.4. Class Diagram

Figure 4 and Figure 5 shows the class diagrams of the bootstrap server and the peer.

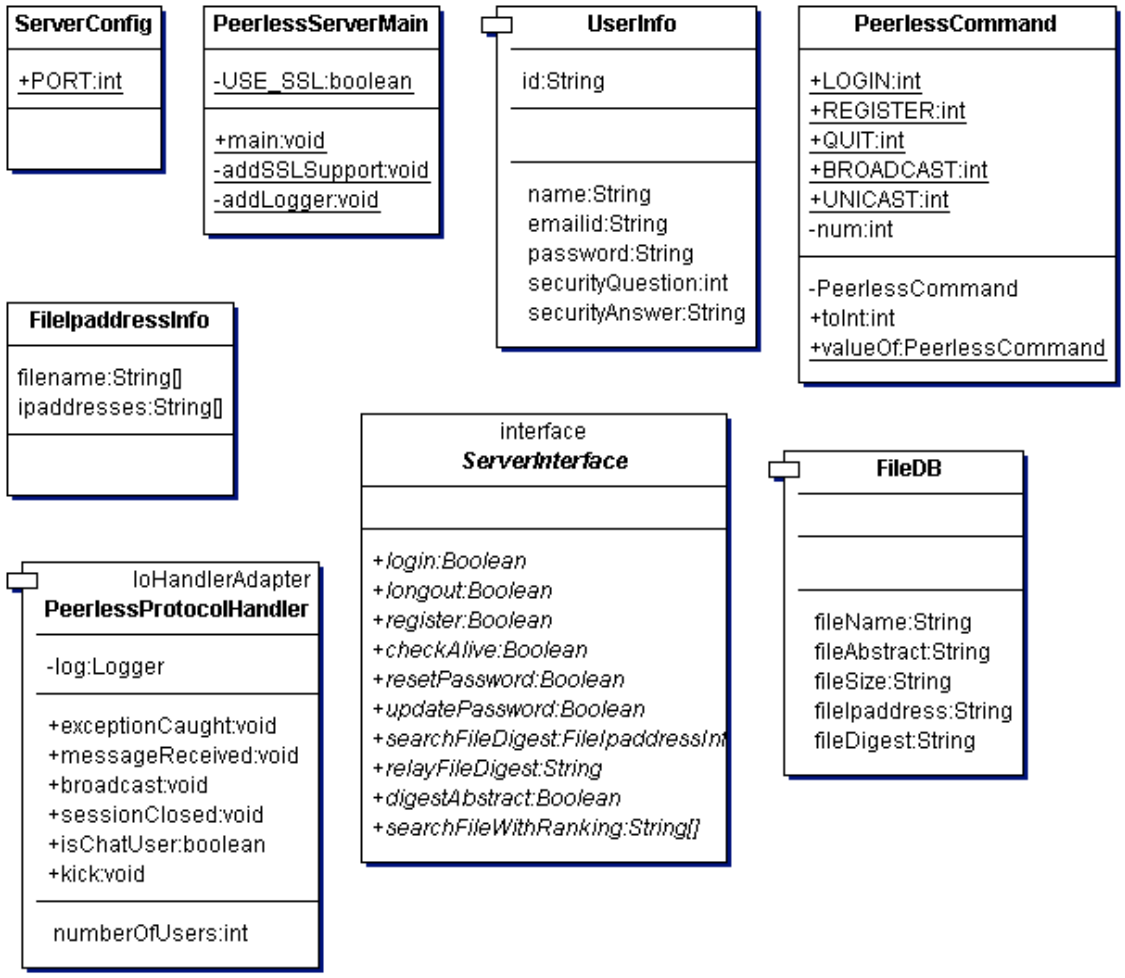


Figure 4. Class Diagram of Bootstrap Server

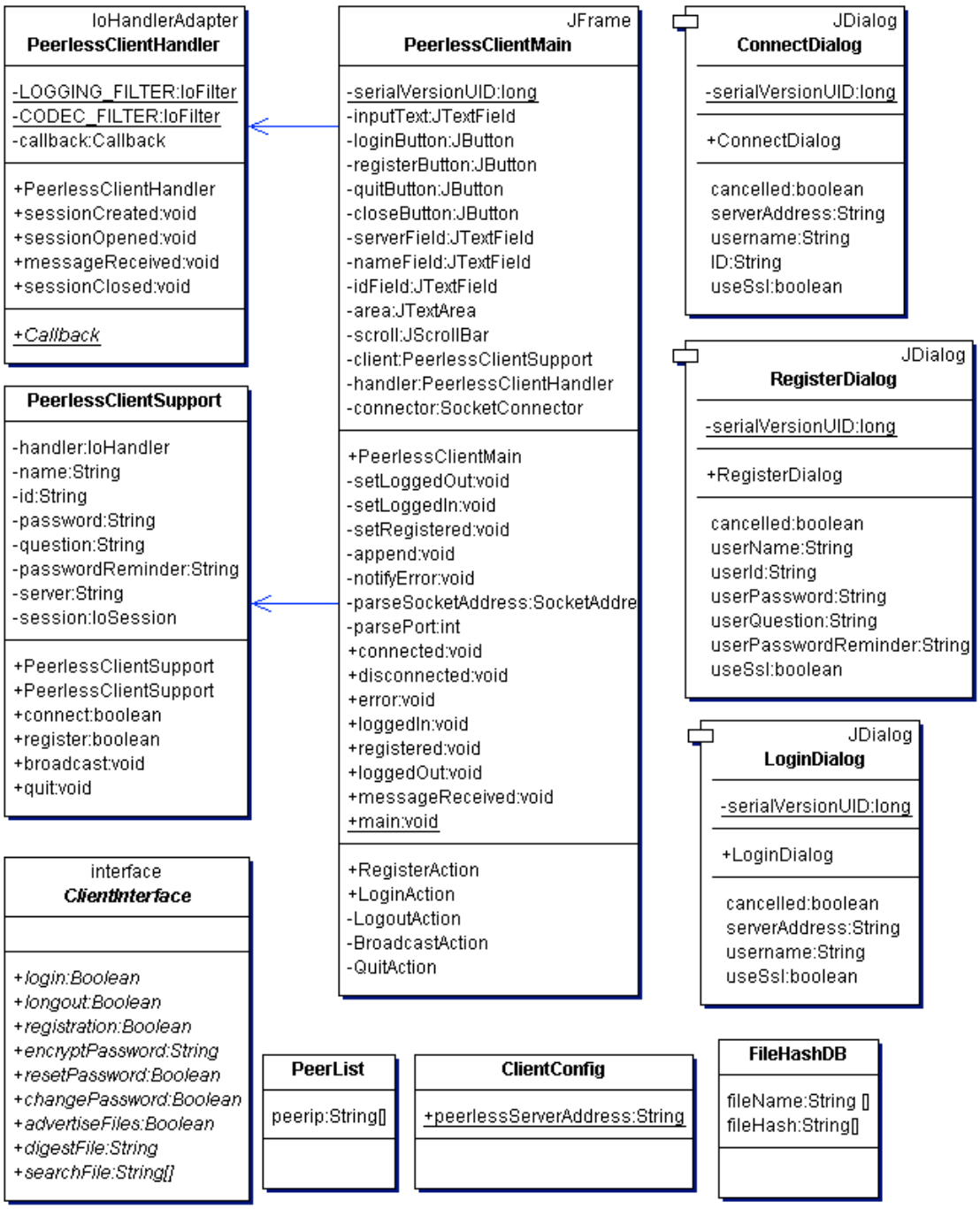


Figure 5. Class Diagram of Peers

3.5. Programming Interfaces

Server and client interfaces are designed. These are shown in Figure 6 and Figure 7.

```
      Boolean login(String id, String password);

      Boolean longout(String id, String password);

      Boolean register(String id, String password, int questionNum,
String answer);

      Boolean checkAlive(int ipaddress, int port);

      Boolean resetPassword(String id, int Question, String Answer);

      Boolean updatePassword(String id, String currPassword, String
newPassword);

      FileIpaddressInfo searchFileDigest(String fileDigest); // return
value: files & ip addresses

      String relayFileDigest(String fileDigest);

      Boolean digestAbstract(String filename);        // one person
takes this part, paper 1

      String[] searchFileWithRanking(String filename); // retrun value:
IP lists --> This is research part
```

Figure 6. Server Interface

```
      Boolean login(String ID, String Password);

      Boolean longout();

      Boolean registration(String ID, String encPassword, int Question,
                  String Answer);

      String encryptPassword(String rawPassword);

      Boolean resetPassword(int Question, String Answer);

      Boolean changePassword(String currPassword, String newPassword);

      Boolean advertiseFiles(String[] digestedfile);

      String digestFile(String filename); // return value: digestedfile

      String[] searchFile(String ipaddress, String filename); // return
value: List of ip addresses
```

Figure 7. Client Interface

3.6. Packet Structure

The different Packets used for communication are created using objects of different classes. The Communication between client and server is being done using these packets, which are nothing but objects of different classes. When the server receives an object for a particular session, it uses the class to decide which type of object has been received and performs the corresponding action. Now I will put down the various Packet Structures with its different fields.

Packet Structure for Registration :-

Command Type (String Variable)	User ID	Password	Password Question	Hint Answer
String "register" Used	*	*	*	*

* :- User Defined . All the fields of the packet are string variables. A Java class is being used to bind the above variables together. Packet for Registration is nothing but a object of the class register. Command Type, UserID etc are the member variables of that class. A packet of the type register is sent from client to server when he wants to login.

Packet Structure for Login :-

Command Type (String Variable)	User ID	Password	IPAddress
String "login" Used	*	*	*

* :- User Defined . All the fields of the packet are string variables. A Java class is being used to bind the above variables together. IPAddress is the Pvt IP address of the host trying to login. A packet of the type Login is sent from client to server when a client want to login.

Packet Structure for Broadcasting :-

Command Type (String Variable)	Message
String "broadcast" Used	*

* :- User Defined . All the fields of the packet are string variables. A Java class is being used to bind the above variables together. The broadcast packet is used when the Server is trying to send a message to all the peers who have logged into the Peerless Application.

Packet Structure for Quit :-

Command Type (String Variable)
String "Quit" Used

This packet is sent to the Server by the Client when it has and it wants to close the connection and logoff from the application.

Packet Structure Requesting a Client to become Super Peer :-

Command Type (String Variable)
String "Request Super" Used Request

This packet is sent by the Server to the Client when it requests the client to be a Super Peer. The Java Class used for this Packet is ReqSuperInv.

Packet Structure to become a Super Peer:-

Command Type (String Variable)
String "Accept" Used Request

This packet is sent to the Server by the Client when it accepts the servers and decided to become a Super Peer. Java Class used for this Packet is AcceptSuperInv.

Packet Structure for Super Peer IP List Broadcast:-

Public IP Address List	Private IP Address List
List of Public IP Address of all Super Peers in the System	List of Private IP Address of all Super Peers in the System

This packet is sent from the Server to a Peer/Super Peer, so that the destination knows gets information about all the super peers in the system.

Packet Structure to search for a File:-

File Signature (Byte Array)
*

This packet is sent to the Server by the Client/Super Peer when it is looking for a particular file. The search string is converted to a File Signature (a Byte Array) stored in an Object and transmitted to the server.

Packet Structure used to send back Result to peer :-

Filename (String Variable)	File Signature (Byte Array)	Public IP Address (String)	Private IP Address (String)	File Abstract (String)
*	*	*	*	*

*-Value defined at server or Super peer. This the Packet is sent by the server or the super peer to the peer, it contains the result of the search conducted at the server/Superpeer. If multiple results are present multiple packets are made and sent back to the client.

Packet Structure to Indicate Null Result:-

No Result (int)
*

*-Value of the No Result variable is set to 0 and sent to the client by the server/ super peer to indicate no results were found for the search.

3.7. Registration

When a client wants to register on to the server, it uses the register dialogue box provided in the GUI to send registration information the server. The client encapsulates the registration information in a registration packet and sends it over the client, using the encryption scheme decided by the user. At the server, the command field of the registration packet is extracted to find what kind of treatment is required for the packet. Once the server realizes that a client is trying to register, it first compares the user-id sent with all the existing user-ids in the database. In this case we have used a file to write the details of all existing users for our application. If the server finds that the user-id is already in use, it doesn't register the user-id and sends a registration failed message to the client, and allows the client to try another user-id. If the user-id of the client is unique then the server registers the user into the database and also assigns a session to the client. The client is now free to access the resources of the server. It can send digested strings to the server, which the server will use to compare against its file digests to locate the details of the requested file.

3.8. Login/logout

The Login/Logout feature is used by a returning user who is already registered with the client. The user uses the login dialogue box at the client to login into the server. The client sends the user information using the Login packet to the server. In this case as well the server extracts the command portion of the packet to determine the treatment for the packet. Once the server realizes this is a login packet, it compares the user-id and password with the values stored database. If the they don't match, the server sends a incorrect password message to the client. If the client details match the details stored at the server, then the server looks the session table, which holds the details about the sessions currently serviced by the server and the clients which are using them. If the requesting user-id is already bound to a session, then the server refuses a session to the client, if not then the user is granted a session and is free to utilize the server resources.

3.9. Signature Generation, Signature Comparison and Ranking

To provide a Google-like text search mechanism, the server should maintain information about all the files in the system. The necessary information that need to be maintained by the server include file name, abstract of the file, address of the user maintaining the file and a digest of the file which will be used by the server to search for matching files and to return a ranked list containing information (file name, location and abstract) of these matching files. All the participating clients must make use of some sort of hashing technique to generate the digest (signature) for all its files that it wishes to share and should advertize this information to the server. Additionally, the clients searching for files in the system should also hash its query string using the same hashing technique used for creating file digests. The following section provides details of the hashing technique that we plan to implement.

We plan to implement a 3-signature hashing technique using the English dictionary as our reference. Here, we maintain a string array 'Ref[]' where every element is the first three alphabets of the words in the dictionary. To generate the signature of a file, for every element in the string array Ref[], we parse the file to see if it contains a match corresponding to Ref[i] element and if there is a match we make the corresponding bit in the files binary string (Signature) as one. That is, every file is represented by a binary signature S1S2S3……Sn where a value of 1 for Si indicates that the file contains at least one element that matches with Ref[i]. Now as mentioned before, the server maintains the binary signature of every file that is shared in the system. Therefore, when a client queries the server for a particular file, the server matches the binary signature of the query with the signature of all the files that it holds. A query is said to match a particular file if the signature of the query has ones in most of the locations where the file signature has ones. Higher the correspondence of ones between the query signature and the file signature better is the match. This additional information obtained as the result of comparison between the query and file signature can be used for the purpose of ranking. That is, files whose signature has a better match with the query signature could be ranked higher in the response list provided by the server.

We plan to implement three objects for the purpose of signature generation, comparison and ranking:

1) Hash Function:
-Part of the Client program
- int hashFunction(int fileHandle);

2) Signature Comparision:
-Part of the Server program
-int compareSignature(int Signature1, int Signature2);

3) Ranking:
-Part of the Server program
-String[] Ranking(int querySignature);

4. Project Organization

4.1. Personnel and Roles

Peerless Team consists of four members. Every team member has their own roles.

Name	Student ID	Unity ID	Roles
Hee Won Lee	000961896	hlee16	Architect, Developer
Nachiketa V Gadiyar	000916724	Nvgadiya	Process Manager, Developer
Deepak Angathil	000921812	dangath	Configuration Manager, Developer
Hemant E. Imudianda	000921064	heimudia	Quality Manager, Developer

4.2. Implementation Responsibilities

Each member is taking charge of their responsibilities in implementing Peerless Systems.

Name	Responsibilities
Hee Won Lee	Requirement Analysis Design and Implementation - System Framework - Interface Design - File Transfer Documentation
Nachiketa V Gadiyar	Requirement Analysis Design and Implementation - Login/Registration - Protocol Design - Graphic User Interface Documentation
Deepak Angathil	Requirement Analysis Design and Implementation - Login/Registration - UDP Communication - Performance Evaluation Documentation
Hemant E. Imudianda	Requirement Analysis Design and Development

	- Hashing, Signature Comparison, Ranking Documentation

4.3. Deliverables

Deliverable	Date
Preliminary Design Document	Oct 28, 2009
Final Project Document - 12-pages single-spaced document (PDF format) - Performance measurements/analysis - Test cases	Dec 10, 2009
Source Code	Dec 10, 2009
Readme File - Installation procedure - Bug report - Platform specific information	Dec 10, 2009

5. Implementation

5.1. Database Structure :- The database implementation is different for the Centralized and the Distributed case.

Centralized Case:- In the centralized case, the database used is installed at the server, its name is "peerlessdb". It contains two tables which are essential for the working of this application, they are namely:- ACTIVEPEERS and FILELOCATION.

i) ACTIVEPEERS:- This Table stores the userid, public IP address, private IP address and user type of all the peers logged into the server. User type is used to indicate if the peer is a normal peer or a super peer. The schema for this table is as follows:-

```
CREATE TABLE ACTIVEPEERS
(
      PvtIP               varchar(20) NOT NULL,
      ipAddress           varchar(20) NOT NULL,
      userId              varchar(20) NOT NULL,
      userType             varchar(10) NOT NULL
);
ALTER TABLE ACTIVEPEERS
   ADD CONSTRAINT ACTIVEPEERS_PK Primary Key (ipAddress);
```

ii) FILELOCATION:- This table stores the filename, file signature, public & private IP address, and file abstract of all the files published on the server. The schema of this table is as follows:-

```
CREATE TABLE FILELOCATION
(
      fileName                varchar(50),  NOT NULL,
      fileSignature           varchar(1024) NOT NULL,
      ipAddress               varchar(20) NOT NULL,
      fileAbstract            varchar(128) NOT NULL,
      PvtIPAdd                varchar(20) NOT NULL
);
```

Distributed Case:- In the distributed case, the database used is installed at the super peer, its name is "peerlessdb". It contains two tables which are essential for the working of this application, they are namely:- ACTIVEPEERS and FILELOCATION.

5.2. Hash Space

Centralized Implementation:

When a peer publishes its file signatures, they are transmitted into the centralized server. In the server, the file signatures are stored in Hashtable. The coordinate of a file signature in the hash space of Hashtable is stored at the same time. The values of the location are stored in the type of int[] in Hashtable. The structures of Hashtable and FileSignature are shown below.

```
Hashtable<FileSignature, int[]> sigHashTable;

public class FileSignature extends AbstractMessage implements Comparable<FileSignature>{

        private byte[] fileSignature = new byte[CommonConfig.fileSignatureSize];;

}
```

You can find the coordinate of a file signature with the method of findCoordinate, as shown below.

```
int[] findCoordinate(FileSignature fSig)
```

Distributed Implementation:

With a file signature and the number of hash spaces, you can find the index of the hash space. The method of findHashSpace() provides this function.

```
public static int findHashSpace(FileSignature fSig, int numOfHashSpaces)
```

The whole hash space is divided by the number of the hash spaces. According to the divided hash spaces, file signatures are distributed. Super peers will take the part of the hash space in their own hash table.

5.3. Publishing

Publishing message format:

Command Type (String Variable)	FileSignature	FileAbstract	Client Private IP address
"PUBLISH"	*	*	*

Centralized Implementation:

In our implementation, the client publishes the following file information to the server

1) 1024 byte file signature which is generated by applying hashing function on the file
2) A 128 character file abstract
3) Private IP address of the client. However, the public client IP address is automatically derived by the server from the client session

Our implementation allows client to publish its files to the server in two ways

- During login
- Using a push button at the client GUI

During client login or when the publish button is pressed, the PublishAll function is called which reads all the files from the default shared directory using a file array object and generates the corresponding fileSignature and fileAbstract for each file.

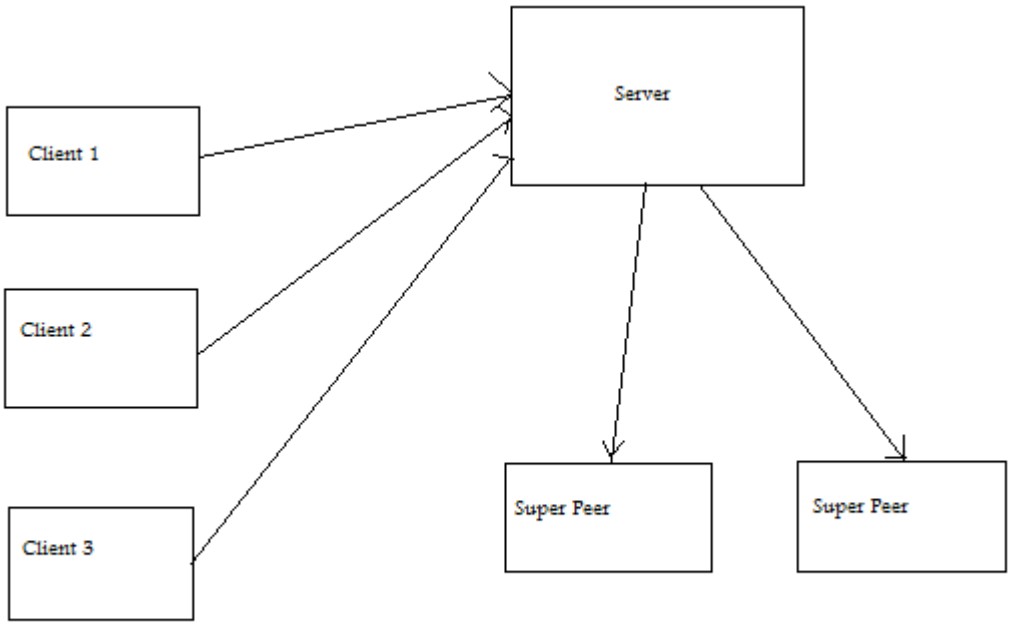
void publishFileAll()

void publishFile(FileSignature fileSignature, FileAbstract fileAbstract);

The PublishAll function then repeatedly calls the publishfile function for each file stored in the directory. The publishfile function in-return creates a Publish message, fills it with the passes information and then sends it to the server over the session formed during login.

At the server, the Publish message is received in the messageReceived(IoSession session, Object message) function which is defined in the PeerLess Protocol Handler class. Once the server receives the message, it removes the file signature and file abstract from the messafe and then stores it in the File Location table in the database and also in the fileSignatures hash table at the server itself.

Distributed Implementation:



Concept Diagram for distributed case

For the distributed case, the client part of the implementation is same as that for the centralized case. At the server, in case of distributed implementation, the server will first extract the file

signature obtained from the publish message received from the client and then it will call superPeerManager.findHashSpace(signature, superusersIP.size()) function to find out the hash space to which the input signature belongs. Then, the server will connect to the superpeer in charge of that hash space and will forward the request to that superpeer. The superpeer, in turn, will save the publish client file information in its own database.

5.4. Searching

Centralized Implementation:

After logging in when a client wishes to search for a file, he or she writes the text into the text box and hits enter. In centralized case this string is first converted into a signature i.e. a byte stream of length 1024 and it is appended to the RequestSignatureMessage along with number of search results required and send across to the server using a UDP connection. At the server end this is received as an object. If the object is found to be of the format RequestSignatureMessage, the file signature is first extracted. Then using this signature, all the signatures which are closely related to it are obtained from the hash space. For all these signatures search is individually performed in the data base to find any match. If for a signature a match is found in the database, the private IP address, public IP address, file name, file abstract and signature are immediately sent back to the client using the ResultSignatureMessage packet. This in turn is displayed on the search window. If no result is found the information is conveyed back to the client with a different message. Upon the reception of this message the client displays that no match was found for that particular query.

Distributed Implementation:

Distributed is very similar to the centralized case except for few changes. Even here after logging in when a client wishes to search for a file, he or she writes the text into the text box and hits enter. After which the client first obtains the list of super peers on the network. For load balancing, based on the number of super peers we equally divide the number of search results required for each super peer. Also we convert the string into signature i.e. a byte stream of length 1024 and then add to the RequestSignatureMessage along with number of search results required and send it across to each super peer using a UDP connection. At each super peer end this is received as an object. If the object is found to be of the format RequestSignatureMessage, the file signature is first extracted. Then using this signature, all the signatures which are closely related to it are obtained from its own hash space. For all these signatures search is individually performed in the data base to find any match. If for a signature a match is found in the database, the private IP address, public IP address, file name, file abstract and signature are immediately sent back to the client using the ResultSignatureMessage packet. This in turn is displayed on the search window. If no result is found the information is conveyed back to the client with a different message. Upon the reception of this message the client displays that no match was found for that particular query.

Message Formats

RequestSignatureMessage

File Signature	Number of results required

ResultSignatureMessage

File name	File Signature	Public IP address	Private IP address	File Abstract

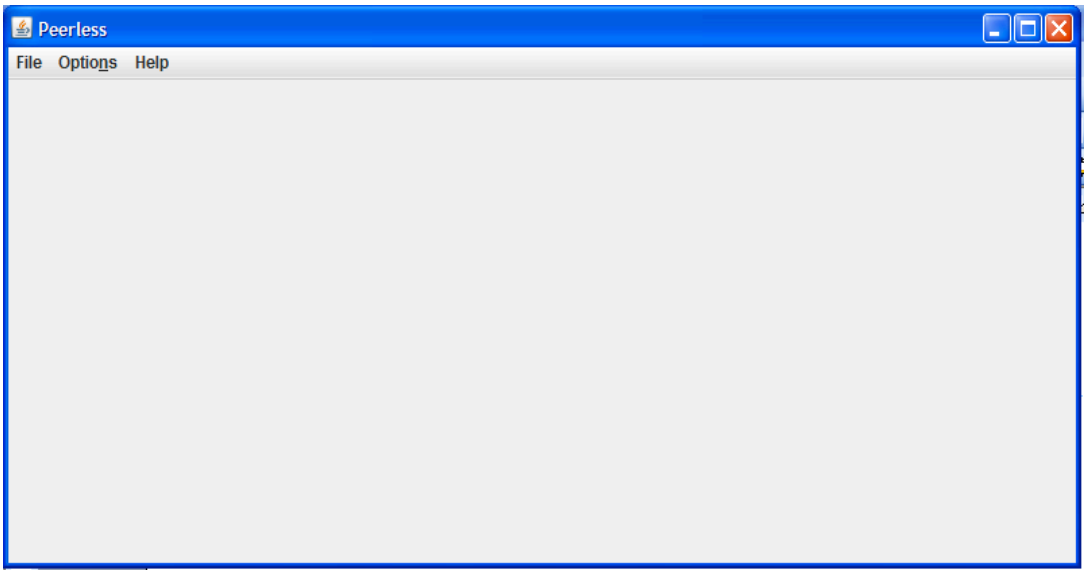
5.5. File Transfer

We first set up the default download folder. When a file is requested to be downloaded, we take the IP address and the default port number of 2221 and open an FTP connection using the apache mina with the target client using the built in class FlientTransferClient. And then using the download function defined in the class FlientTransferClient, the download is initiated by passing the filename and the folder path to which the file needs to be downloaded.

5.6. Graphic User Interface

Our implementation provides a GUI for giving the user a pleasant and easy experience with this application. Now I will walk you through the usage of our GUI and its features.

i) The moment a user runs a application the Main GUI window opens up which has a menu bar with 4 menu items namely File, Options and Help. These menu items again have sub menu items. The main GUI window which pops up moment a host runs our application is shown below:-



Now I will talk about the sub menu items of each of the menu items.

- **File Menu** :- The File menu item has four sub menu items namely the login, register, quit and close.

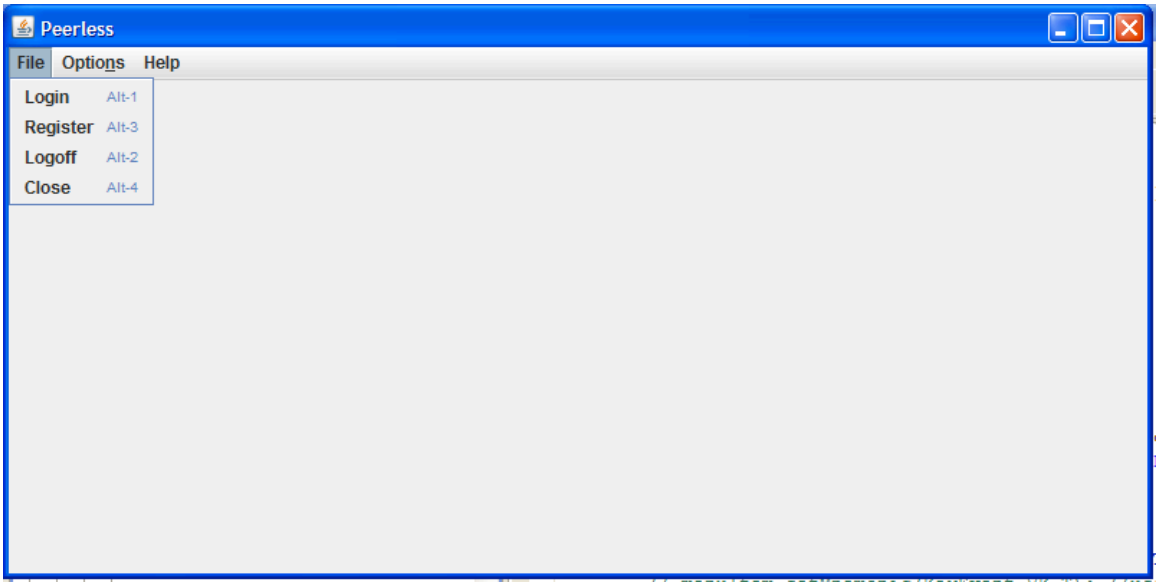
    - Login :- The Login sub menu item can be activated using the mouse to produce the login dialog box. The Login sub menu can also be activated using the short cut key Alt + 1.

    - Register:- The Register sub menu can be activated using the mouse to produce the register dialog box. The register sub menu can also be activated using the short cut key Alt + 3.

    - Logoff:- The Logoff submenu can be activated using the mouse and causes the user to log off from the application. The logoff sub menu can also be activated using the short key Alt + 2.

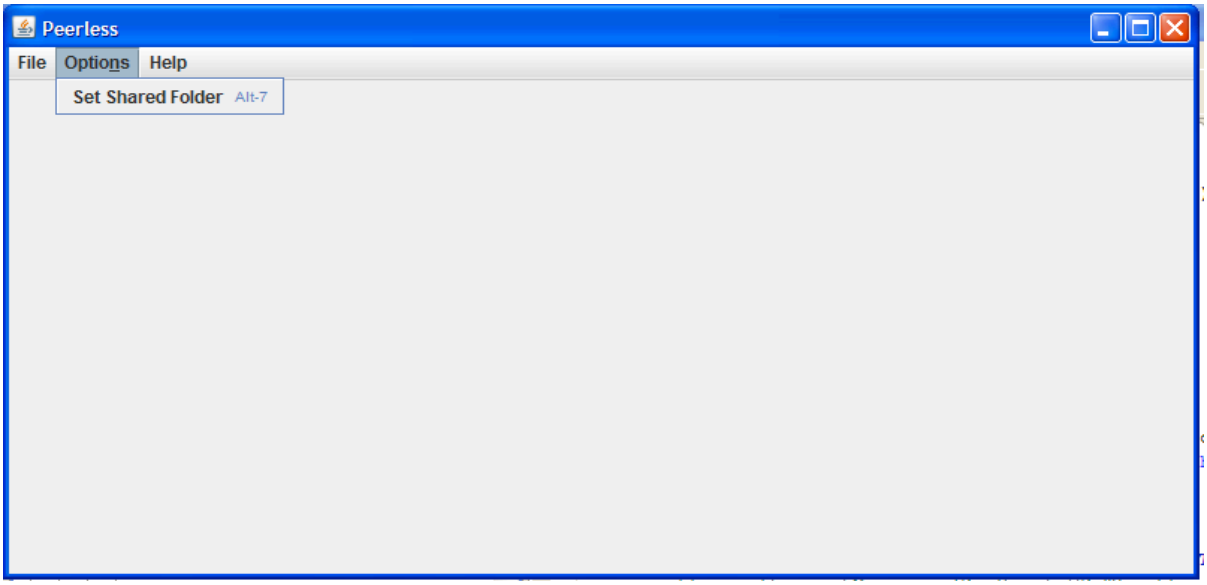
    - Close:- The close submenu can be activated using the mouse and causes the GUI to close. The logoff sub menu can also be activated using the short key Alt + 4.

  Here is a view of the above Description:-

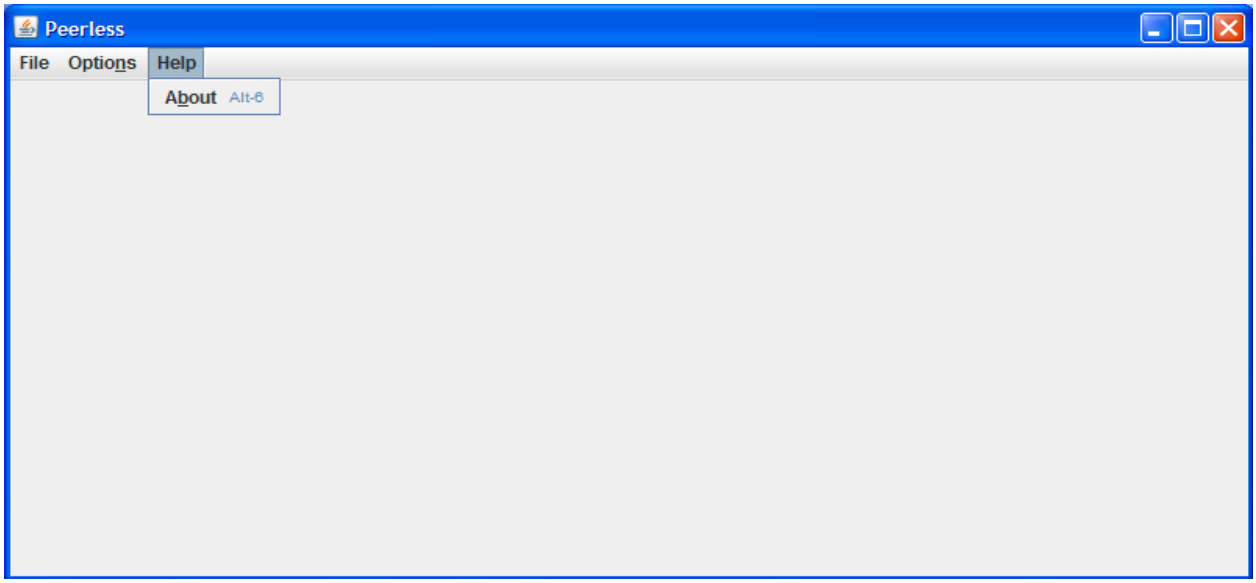


- **Options Menu**:- The options menu has one sub menu item namely Set Shared Folder.

    - Set Shared Folder:- The Set Shared Folder can be activated using the mouse and helps the user to set the folder whose files he wants to publish to the server / super peer.

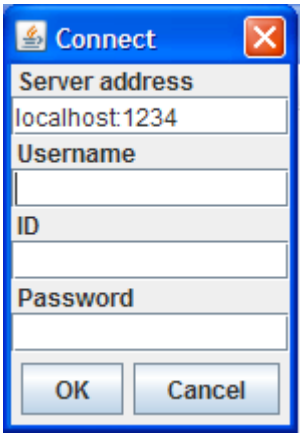


- **Help Menu**:- The help menu has one sub menu item namely about.
    - About:- The about can be activated using the mouse and gives the user information about our team who developed this application.

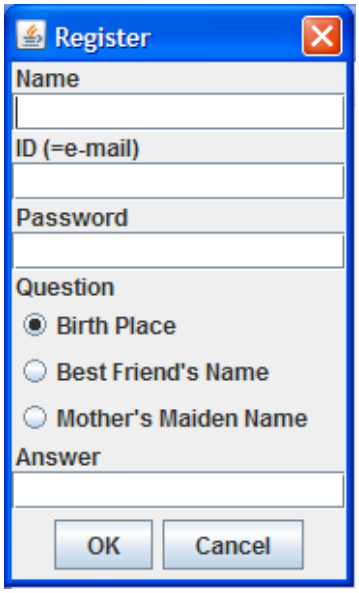


Now I will walk you through the various dialog boxes which open up when the sub menu items are activated.

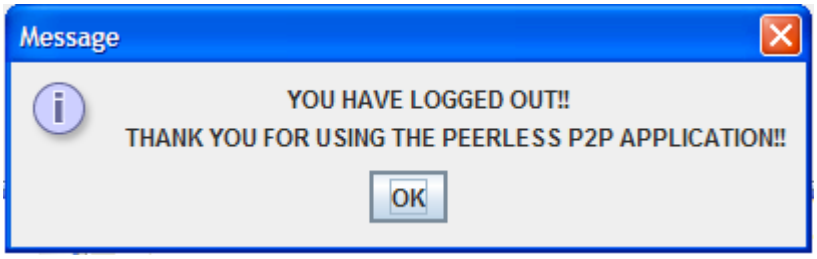
- **Login Dialog Box**:- The login dialog box pops up the moment the login sub menu item is activated. It provides text fields such as username, user id and password so that the user can pass authentication information to the server and login into the application. The server IP address and port number populates in the login register box, so that user is saved of typing the IP address and port number every time, but can change it easily in case he/she needs to.



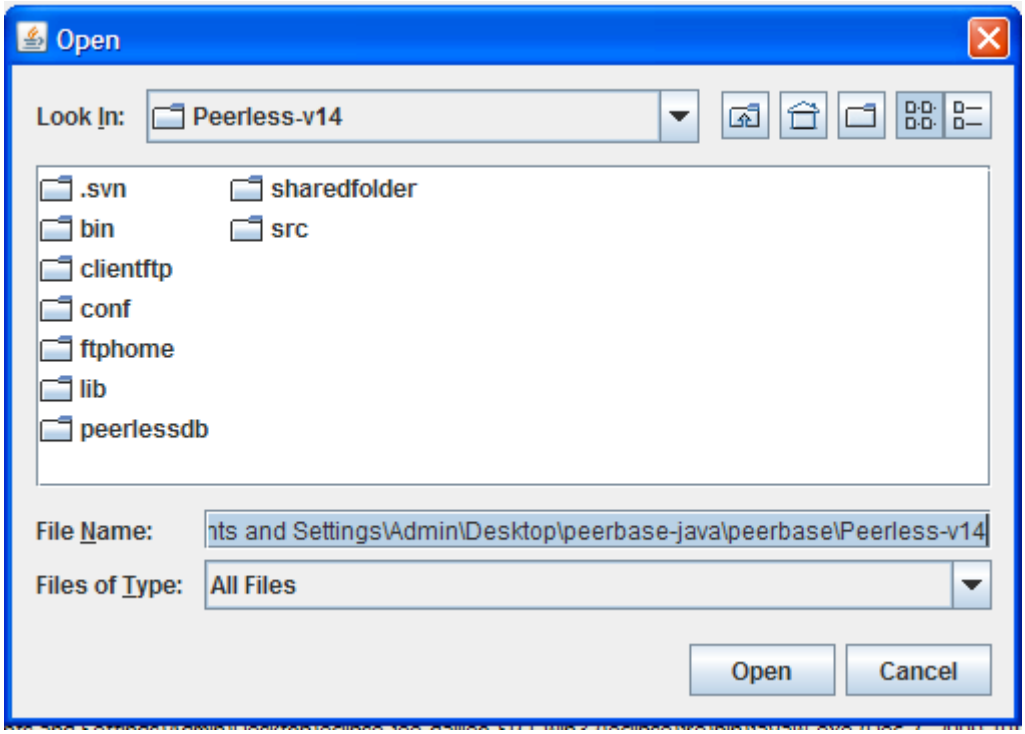
- **Register Dialog Box:-** The register dialog box pops up the moment the register sub menu item is activated. The register dialog box provides text fields such as user name, user id, password, Hint question and hint answer so that user can pass registration information to the server, and register himself with the application.



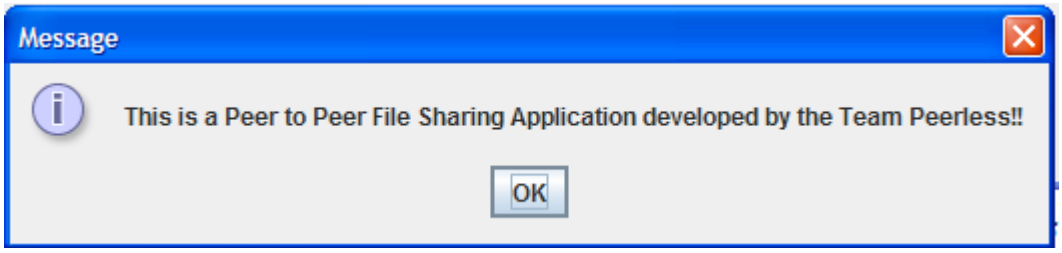
- **Logoff Box:-** Th log off dialog box is used to inform the user that he has logged out of the application.



- **Set Shared Folder Box :-** The Set Shared Folder box pops up when the set shared folder sub menu item is activated. It helps the user to set the folder whose files he plans to publish.



- **About Box:-** The about sub menu item can be activated to have a brief introduction about our team who developed this application.

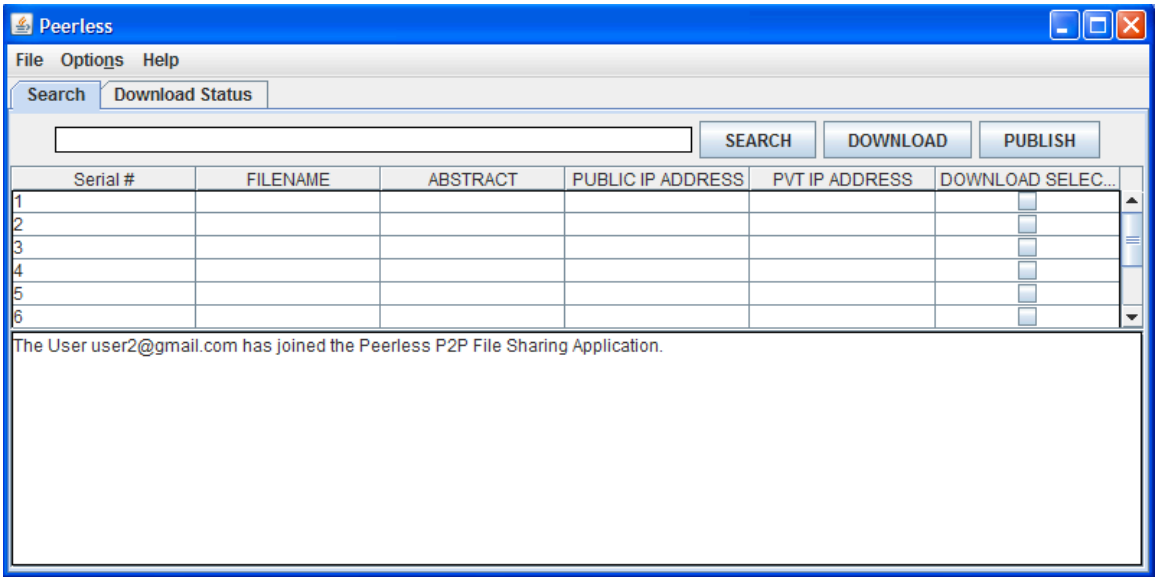


- **Final Search Dialog Box**:- This dialog opens up after the user is authenticated and he logs into the application. This box gives him access to all the resources of the application.

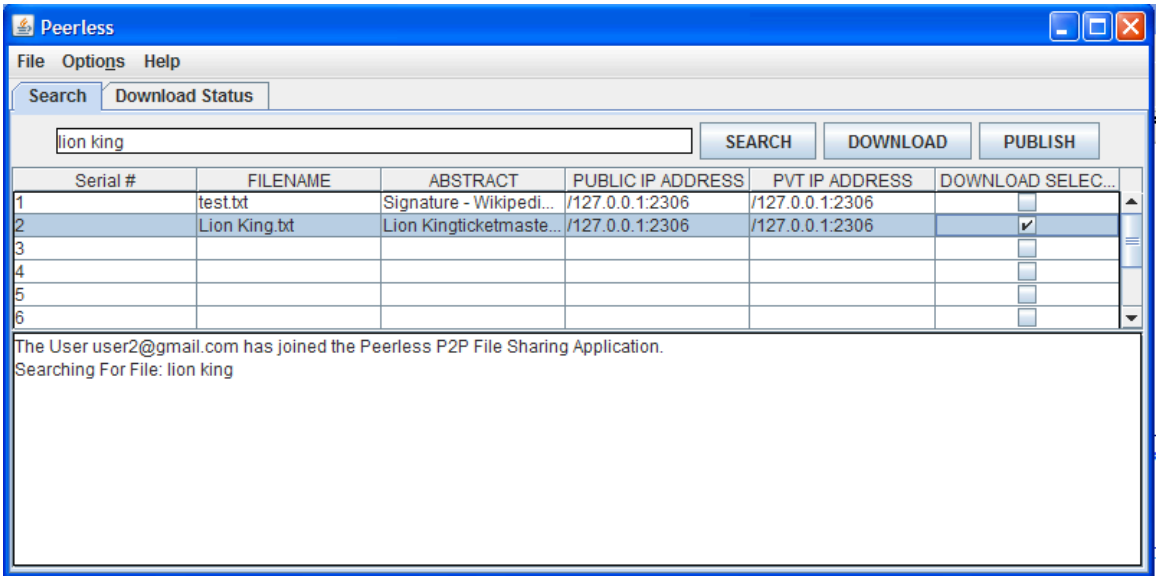
- This dialog box has two panes. One is the **search pane** and another is the **download pane.**

- **Search Pane:-** is used to search for files by passing string inputs. The string input should be the content he is looking for in the files that would be returned in the search result. He can use the check boxes provided to select the files and download them. The search result displayed by this dialog box consists of the filename, public IP address, private IP address,

Here is a glimpse of the Search pane moment the user logs in:-



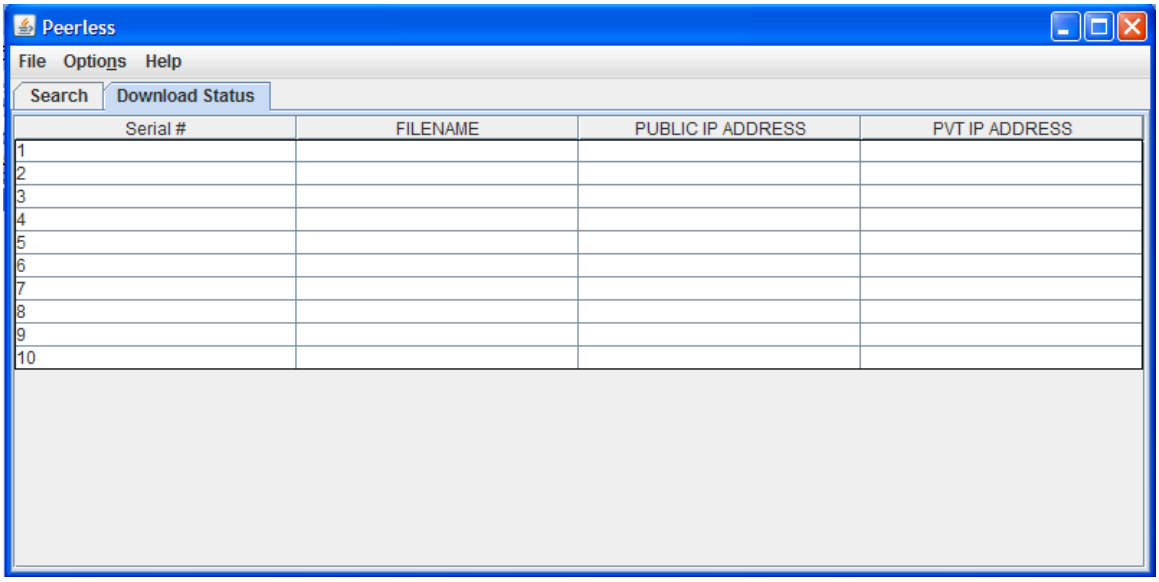
Here is a glimpse of the Search Pane after the server returns the result for the searched string input..



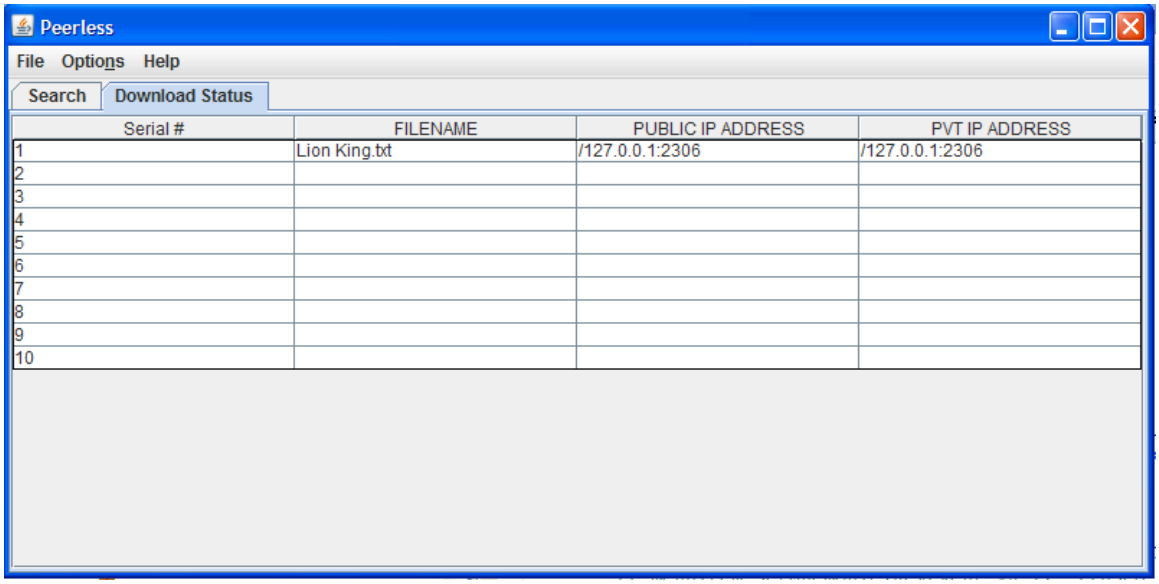
The figure shows the files returned for the search with its details and the desired file selected, so that it can be downloaded.

- **Download Pane:-** is used to display all the files downloaded by the user.

Here is a glimpse of the download pane moment the user logs in:-



Here is a glimpse of the download Pane after the user finished downloading the file selected in the search pane.



6. Document Ranking

Hash Space is divided by the number of the hash space. However, most of file signatures are located in the upper left area, so that the hash space shouldn't be divided equally. In our implementation, among the divided hash spaces the upper left hash space has smallest area, because it contains the majority of file signatures. Hash spaces that are located towards the lower right area has bigger area. That's because they have smaller number of file signatures in their hash space. Figure 8 shows the hash space in our implementation.

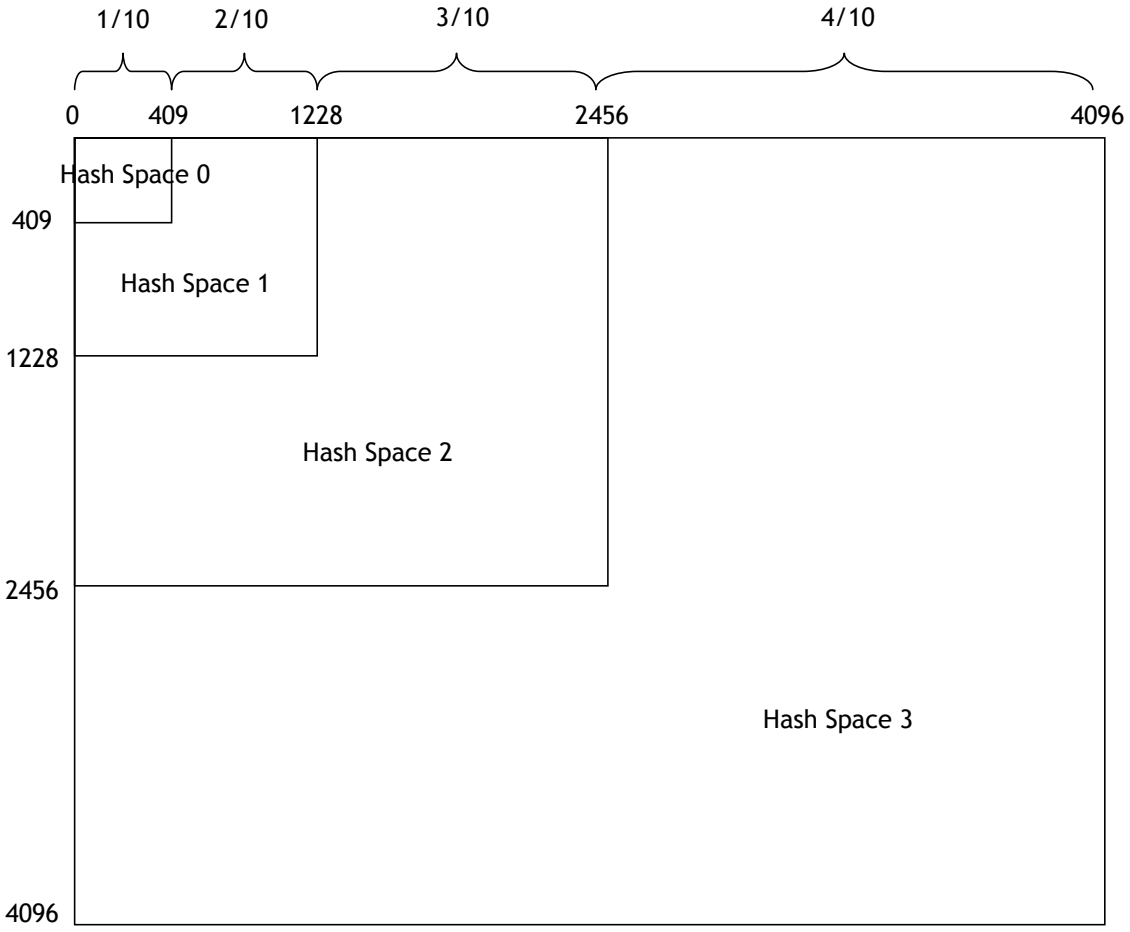


Figure 8. Hash Space

Centralized Case:

When you search a file in a peer client, it is changed into a file signature and transmitted to the centralized server. Then, the server finds matches with the signature in the hash space that contains signature information. The server can search a file according to the minimum number of signature bit match. The method is shown as follows.

```
public List<FileSignatureMatch> searchSignature(FileSignature inputSig, int
maxNumOfSearch, int minNumOfMatch)
```

For example, if you take 3 as a minimum number of match, then you can get signatures that has more than 3 bit match. Then, a file signature that has the more bit match comes first. By adjusting this value, you can rank results.

Distributed Case:

In distributed implementation, when you search a file, you should first find the IP addresses of super peers that contain file information, and make a query to the super peers. With a file signature of a query, you can find a coordinate in (int[0], int[1]) of the whole hash space, using the following method.

```
public static int[] findCoordinate(FileSignature fSig)
```

The following method finds IP addresses of super peers based on the method of findCoordinate() above. Since every peer maintains the IP addresses of the super peers from the server, each peer can find them with a file signature.

```
public PeerIpAddress[] findHashSpaceAddress(FileSignature fSig)
```

Among multiple super peers, a peer starts to search a file from a super peer that maintains the hash space that has the biggest coordinates. After that the peer makes a search form a super peer that manages the next hash space that contains the smaller coordinates. In this way, a peer continues to search a file from super peers, until it meets a super peer that manages a query signature. With this ranking system, you can first find a file that contains the longest ones in its signature and matches with the query. On the other hand, you divide the total number of search requests by the number of hash spaces that manage coordinates bigger than the coordinate of the query signature. You make a query to each super peer according to the number of search request. For example, if you are searching for 30 file matches and the query signature belongs to Hash Space 1, then you first starts to search a file from Hash Space 3, expecting 10(=30/3) results from the super peer that manages Hash Space 3. Next, you will search for 10 results from the super peer that has Hash Space 2. Finally, you will get 10 results from the super peer from the super peer that maintains Hash Space 1.

7. Performance Evaluation

7.1. Popular vs. Unpopular Query

The first test we used to evaluate the performance of our P2P implementation is to compare the time required for popular compared to a unpopular queries. This test was run for both centralized as well as distributed case and the results for these architectures have been compared.

For the files published by us in both the centralized and distributed case we found that string "buffer" and "acknowledgement" were popular queries and the string "test data" was a unpopular query. Popular Query is defined as a query which results in many matches from the P2P implementation while a unpopular Query is defined as a query which results in very few matches.

Here are our test results for the above for the Centralized Case.

# of Files Published	Search Time(ms)	Query(String)	Query Type
5	350	test data	Un-Popular
5	180	Acknowledgement	Popular
5	190	Buffer	Popular

# of Files Published	Search Time(ms)	Query(String)	Query Type
10	410	test data	Un-Popular
10	182	Acknowledgement	Popular
10	250	Buffer	Popular

# of Files Published	Search Time(ms)	Query(String)	Query Type
20	571	test data	Un-Popular
20	185	Acknowledgement	Popular
20	260	Buffer	Popular

# of Files Published	Search Time(ms)	Query(String)	Query Type
60	630	test data	Un-Popular
60	205	Acknowledgement	Popular
60	290	Buffer	Popular

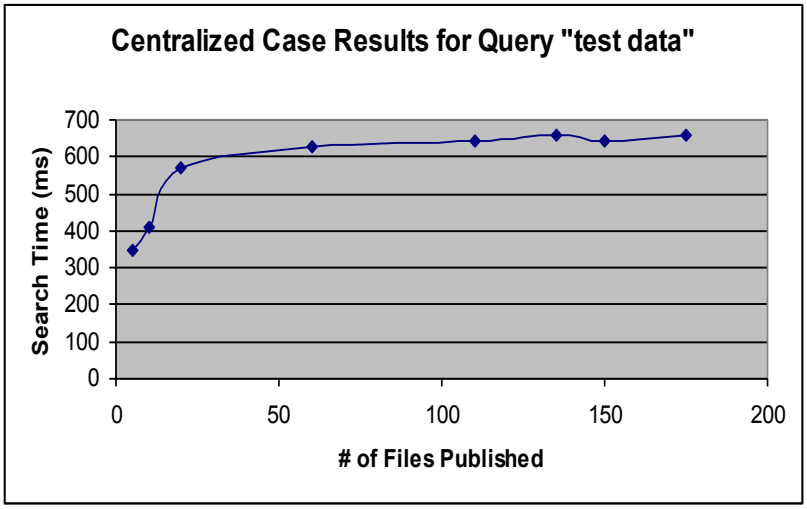
# of Files Published	Search Time(ms)	Query(String)	Query Type
110	645	test data	Un-Popular
110	215	Acknowledgement	Popular
110	295	Buffer	Popular

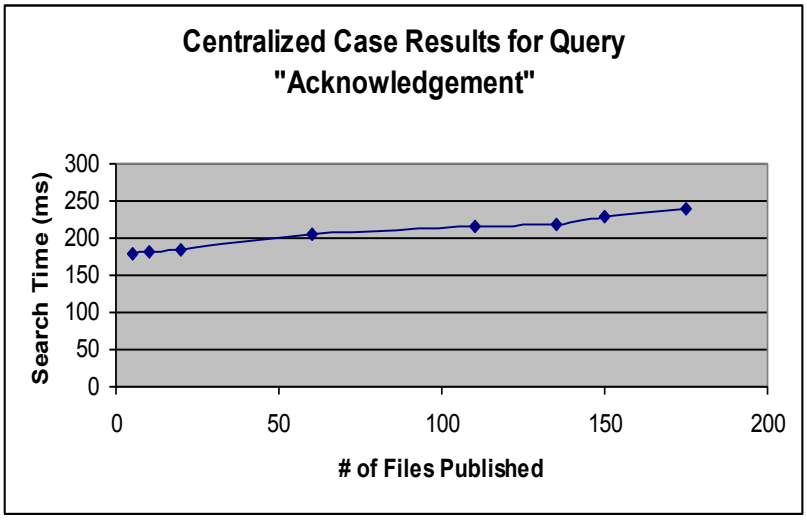
# of Files Published	Search Time(ms)	Query(String)	Query Type
135	661	test data	Un-Popular
135	218	Acknowledgement	Popular
135	298	Buffer	Popular

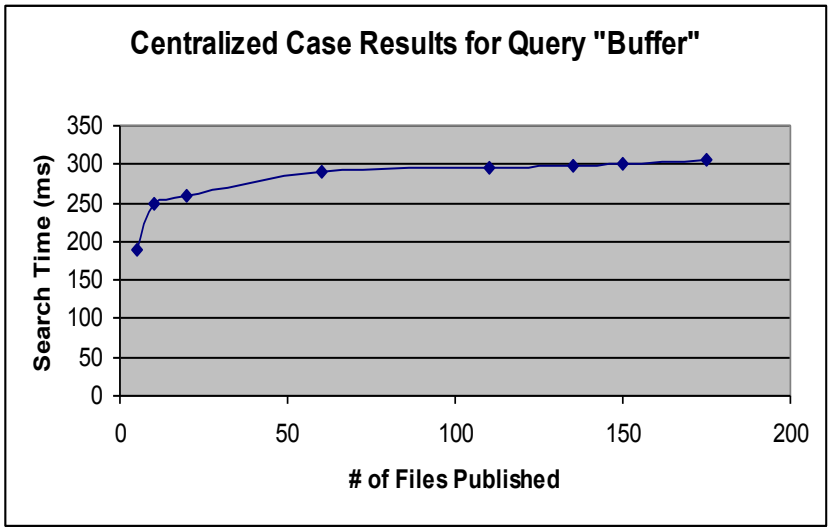
# of Files Published	Search Time(ms)	Query(String)	Query Type
150	641	test data	Un-Popular
150	230	Acknowledgement	Popular
150	300	Buffer	Popular

# of Files Published	Search Time(ms)	Query(String)	Query Type
175	660	test data	Un-Popular
175	240	Acknowledgement	Popular
175	305	Buffer	Popular

In the centralized case we observe that "test data" the un-popular query takes a nearly double the time than the popular queries. Also it is seen that when the number of files published increases the time required to search for the result increases. The above results were observed with a bootstrap server and 1 peer in place. The above results have been plotted below.







Here are our test results for the above for the Distributed Case:-

# of Files Published	Search Time(ms)	Query(String)	Query Type
25	498	test data	Un-Popular
25	551	Acknowledgement	Popular
25	564	Buffer	Popular

# of Files Published	Search Time(ms)	Query(String)	Query Type
50	621	test data	Un-Popular
50	591	Acknowledgement	Popular
50	601	Buffer	Popular

# of Files Published	Search Time(ms)	Query(String)	Query Type

75	641	test data	Un-Popular
75	600	Acknowledgement	Popular
75	620	Buffer	Popular

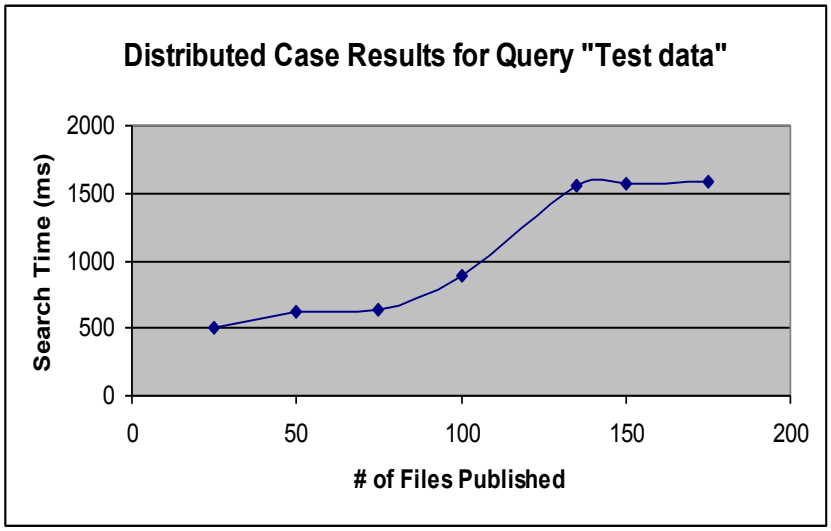
# of Files Published	Search Time(ms)	Query(String)	Query Type
100	891	test data	Un-Popular
100	610	Acknowledgement	Popular
100	630	Buffer	Popular

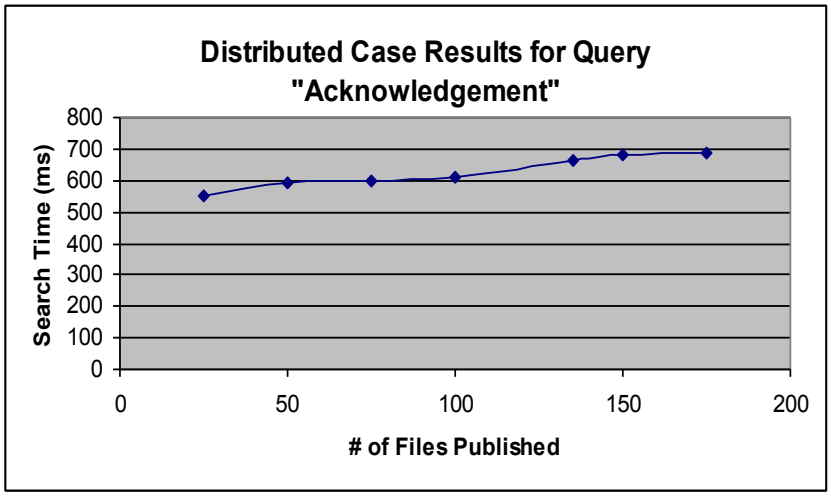
# of Files Published	Search Time(ms)	Query(String)	Query Type
135	1552	test data	Un-Popular
135	661	Acknowledgement	Popular
135	700	Buffer	Popular

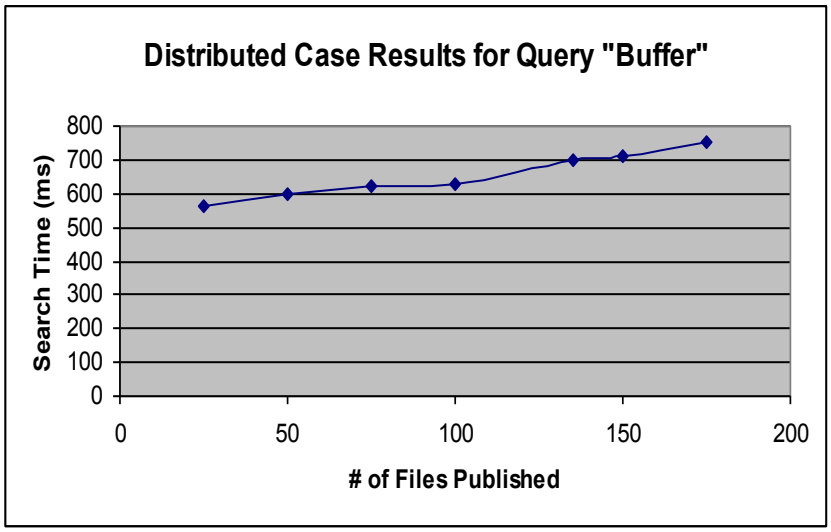
# of Files Published	Search Time(ms)	Query(String)	Query Type
150	1572	test data	Un-Popular
150	680	Acknowledgement	Popular
150	710	Buffer	Popular

# of Files Published	Search Time(ms)	Query(String)	Query Type
175	1590	test data	Un-Popular
175	690	Acknowledgement	Popular
175	750	Buffer	Popular

In the Distributed case it is observe that the delay nearly doubles for all the search queries, popular as well as unpopular ones. The relation between the search time for popular and un-popular queries still remains the same. The above results were observed with 1 bootstrap server and 2 super peers. The distributed case took double the time as the centralized case in this case the query has to go to go to the 1st super peer first return a result and then propagate to the second one, hence increasing the delay.


Distributed Case Results for Query "Test data"
Search Time (ms)
2000
1500
1000
500
0
0
50
100
150
200
# of Files Published


Distributed Case Results for Query "Acknowledgement"
Search Time (ms)
800
700
600
500
400
300
200
100
0
0
50
100
150
200
# of Files Published


Distributed Case Results for Query "Buffer"
Search Time (ms)
800
700
600
500
400
300
200
100
0
0
50
100
150
200
# of Files Published

7.2. Accuracy of Search

The procedure followed to test the accuracy of our search is to first find a list of lesser used words like archipelago and then to the repository of files add few files which specifically contain these words. This will make sure that we have few files in the repository with exactly matching words and few with no such word in it. Then publish all the files and search for each of these words. Then count the no of accurate result that was produced and the total number of results that was given. The ratio of correct no of accurate results to the total number of results that is displayed on the GUI will give us the percentage of accuracy of the search.

These were the results found for different set of published files

Published files: 45

Searched Word	No of accurate results	Total no of results	Accuracy
nanotubes	7	11	63.33
Fortuitous	6	9	66.67
archipelago	4	7	71.42
pithy	4	5	75.00
Bonhomie	7	12	58.33

Published files: 80

Searched Word	No of accurate results	Total no of results	Accuracy
nanotubes	11	15	73.33
Fortuitous	8	12	66.67
archipelago	5	7	71.42
pithy	6	8	75.00
Bonhomie	8	14	57.14

Published: 250

Searched Word	No of accurate results	Total no of results	Accuracy
nanotubes	16	20	80
Fortuitous	18	20	90
archipelago	17	20	85
pithy	17	20	85
Bonhomie	16	20	80

The search results can be interpreted this way.

It was made sure that for all the above cases maximum number of results displayed on the GUI was made 20. From the first two tables we can see that the search results displayed were less than 20 since the number of files published was less. And the accuracy was observed to be between 60% and 75%. There were few false results because of the false positive values which got generated. Then the number of published files was increased to around 250 to make sure that the number of results shown on the GUI reaches 20, as it is the max we are displaying. Once the number files published are more, the result set returned contains a mix both accurate matches as well closely matching results which is more than 20. If no ranking was implemented and the first 20 results were arbitrarily displayed on the GUI, it is possible that the accuracy will remain almost same as what was before. But with ranking implemented, from the result, 20 best matches got displayed on the GUI with lesser matching results getting filtered out. As a result the accuracy improved.