DIFFSEARCH ALGORITHM TO EXPLOIT FILE SHARING HETEROGENEITY

A PROJECT REPORT

Submitted by

DHEEPIKA.K **71205104005**

GOMATHI.D **71205104009**

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

KUMARAGURU COLLEGE OF TECHNOLOGY, COIMBATORE

ANNA UNIVERSITY: CHENNAI 600025

APRIL 2009

ANNA UNIVERSITY: CHENNAI 600025

BONAFIDE CERTIFICATE

Certified that this project report **"DIFFSEARCH ALGORITHM TO EXPLOIT FILE SHARING HETEROGENEITY"** is the bonafide work of **"DHEEPIKA. K, GOMATHI. D"** who carried out the project work under my supervision.

SIGNATURE

Dr. S Thangasamy
HEAD OF THE DEPARTMENT

Department of Computer Science and Engineering,
Kumaraguru College of Technology,
Coimbatore-641006

SIGNATURE

Ms. S Rajini
SUPERVISOR
Senior Lecturer,
Department of Computer Science and Engineering,
Kumaraguru college of Technology,
Coimbatore-641006

The candidates with University Register Nos. **71205104005**, **71205104009** were examined by us in the project viva-voice examination held on....28/4/09

INTERNAL EXAMINER

EXTERNAL EXAMINER

ACKNOWLEDGEMENT

We are extremely grateful to, **Professor R.Annamalai M.E.**, Vice Principal, Kumaraguru College Of Technology for having given us this opportunity to embark on this project.

We are deeply obliged to **Dr. S. Thangasamy, Ph.D.,** Head of the Department of Computer Science and Engineering for his valuable guidance and useful suggestions during the course of this project.

We wish to express our heartiest thanks to **Mrs. P. Devaki**, Assistant Professor and Project Co-ordinator who helped us to overcome the perplexity while choosing the project.

We thank our guide, **Ms. S. Rajini**, Senior Lecturer, Department of Computer Science and Engineering, for her excellent guidance in each and every step of our project and been with us to complete the project.

We thank the **teaching and non-teaching staff** of our department for providing us the technical support during the course of our project.

We also thank all our **parents and friends** who helped us to complete this project successfully.

ABSTRACT

Although the original intent of the peer to peer (P2P) concept is to treat each participant equally, heterogeneity widely exists in deployed P2P networks. Peers are different from each other in many aspects, such as bandwidth, CPU power and storage capacity. Early search approaches have high indexing cost. But in this project the query answering heterogeneity to directly improve the search efficiency of P2P networks is suggested. It is proposed Differentiated Search (Diff Search) algorithm, the peers with high query answering capabilities will have high priority to be queried. In the DiffSearch algorithm, a query consists of three rounds of searches. In the first round of search, the query is only sent to cache and if that fails the search is resorted to the ultra peer overlay peers with high query answering capabilities. If the second round of search fails in the ultra peer overlay, the third round of search will be evoked to query the entire network. The DiffSearch uses the number of shared effective files as the criterion to select ultra peers and forms the content-rich ultra peer overlay. Based on this ultra peer overlay, a query can be answered by a small portion of peers with high probability before it is broadcast to the entire network. The algorithm converges very fast because of the file sharing heterogeneity in P2P networks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ABBREVIATIONS	DESCRIPTION
P2P	Peer to Peer
DIFFSEARCH	Differentiated Search
DNS	Domain Name Server
RFC	Request For Comment
NAT	Network Address Translation

CHAPTER 1
INTRODUCTION

PEER-TO-PEER (P2P) networks are booming in today's network community because their fully distributed design makes them good candidates to build fault-tolerant file sharing systems which can balance the load of file storage and transfer. Nevertheless, the basic flooding search approach in unstructured P2P networks raises enormous concerns about network scalability. Many approaches have been proposed to replace or optimize the flooding search mechanism.

The structured P2P networks avoid the flooding search by tightly coupling data or indices of data with sharing peers such that a query can be directly routed to matched peers.

Instead of replacing the flooding method, the hierarchical P2P networks try to reduce the flooding traffic by limiting the search scope within a small area of supernodes. Since the indices of leaf nodes are uploaded into supernodes, the search in KaZaA can achieve good coverage of an entire network by only querying a small number of supernodes.

The design of the hierarchical P2P networks improve search efficiency at the cost of performing extra index operations such that a query can either be directed to a host previously registered with possible matched indices or flooded within a small search space which aggregates the indices of an entire network since, in a spontaneously formed P2P network where both peers hosting indices and peers pointed to by indices are unreliable and may join and leave the network arbitrarily, the cost of frequent index updates cannot be ignored. Accordingly, how to balance the costs between searching and indexing becomes a critical design issue in such a dynamic environment. An intriguing question is: "Can we improve the search efficiency of P2P networks with zero or minimal indexing cost?"

The observation on file sharing heterogeneity shows that some peers are more willing to share files than others. For example, some new musicians regard P2P networks as free platforms to distribute their productions and therefore build their reputations.P2P networks, compared with traditional client server architectures such as Web systems, provide ideal infrastructures for those content publishers because of several reasons:

1) It does not require publicized DNS names with fixed IP addresses and stable online time.
2) The bandwidth demand on a service provider in P2P networks is not as critical as that on a server since the load of file transfer can be balanced to other service providers or leaf nodes with replicated data.
3) Service providers in P2P networks can retain better anonymity than well-known servers. In summary, P2P networks create a low cost and easily accessible publishing environment.

1.1 PROBLEM DEFINITION

In this project, the Differentiated Search (DiffSearch) algorithm has been proposed to improve the search efficiency of unstructured P2P networks by giving higher querying priority to peers with high querying reply capabilities. This proposal is based on the observation that query reply capabilities are extremely unbalanced among peers: Seven percent of peers in the Gnutella network share more files than all of those other peers can offer and 47 percent of queries are responded to by the top 1 percent of peers. The remarkable heterogeneity of peers' reply capabilities intrigues to revisit the nature of P2P networks. Rather than all peers actively participating in file sharing, only a small portion of volunteering peers provide the majority of the service in P2P networks.

1.2 STATE OF THE ART

DiffSearch algorithm further improves this environment by promoting content-rich publishers to ultrapeers and making them more visible for other visitors. The heterogeneity of file sharing motivates us to redraw the picture of P2P networks: A small portion of volunteers provide file sharing service to a large number of visiting peers. In such a model, the basic flooding approach is like looking for a needle in a bale of hay since queries are forwarded by a large number of visiting peers without any contributions to search results. Instead of flooding, it is proposed to use the DiffSearch algorithm, which gives service providers higher priorities to be queried.

1.3 SYSTEM STRUCTURE

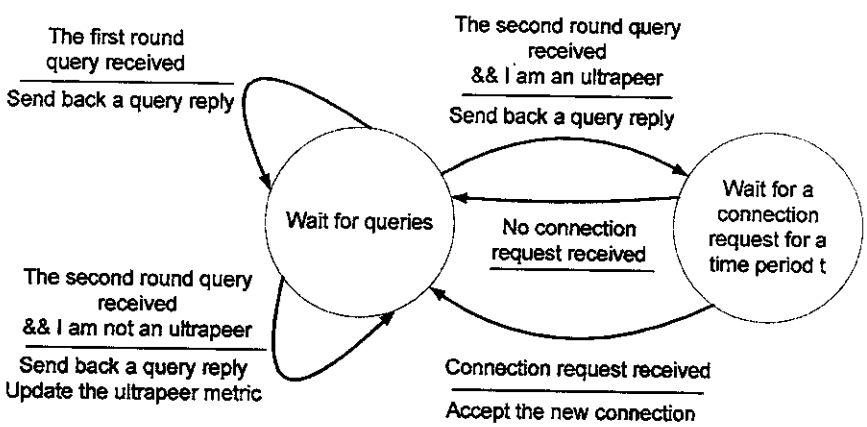

Fig.1.1. Query reply operation of an individual peer

Fig.1.1 gives the query reply operation of an individual peer. The upcoming chapter describes the overview of the DiffSearch algorithm. Chapter 3 describes the proposed design principle. Chapter 4 describes the implementation details that include forming of the ultrapeer overlay and maintenance of cache memory. Chapter 5 describes the implementation tools and techniques. Chapter 6 describes the experimentation results and Chapter 7 gives the conclusion and future extensions.

CHAPTER 2

DIFFSEARCH ALGORITHM

Current hierarchical designs select the ultrapeers by emphasizing their computing capabilities such as band- width, CPU power, and memory spaces. It is argued that the content capacity, i.e., the number of files shared in a node, is also an important factor to decide if a hosting peer is an ultrapeer. After aggregating peers sharing a large number of files into the ultrapeer overlay, good search performance can be achieved by only flooding a query within a small search space.

In the DiffSearch algorithm, a query consists of two rounds of searches. In the first round search, the query is only sent to the ultrapeer overlay. If the first round search fails in the ultrapeer overlay, the second round search will be evoked to query the entire network. The ultrapeer overlay construction is illustrated as follows: If a DiffSearch query fails in the first round search, but succeeds in the second round, this implies two possible cases:

- The queried object is not shared by any ultrapeer such that it can only be found by querying all the networks.
- The queried object is shared by ultrapeers and some of the replies in the second round search are responses from the ultrapeers, which is specified in Fig 2.1.

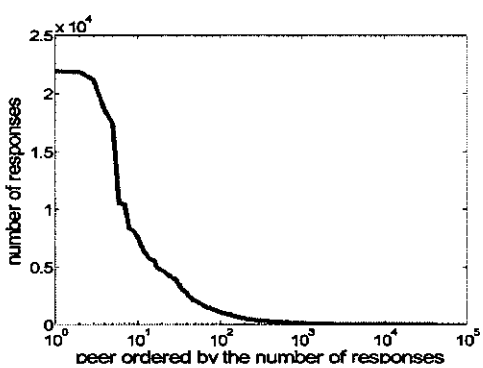

Fig. 2.1. Response distribution.

2.1 STEPS IN DIFFSEARCH ALGORITHM

Using the metric of effective files discussed above, the ultrapeers are self-aware by simply counting the number of shared files which have been visited before. As long as the number of shared effective files reaches the threshold, a peer can promote itself as an ultrapeer. In our DiffSearch algorithm, those peers should form an ultrapeer overlay and have higher priority to be queried. Fig 2.2 gives the steps in the DiffSearch algorithm. The challenge is how to find the ultrapeers and connect them into an overlay efficiently. The basic objectives of our topology creation algorithm are:

1) Each leaf node has at least one ultrapeer as a neighbour and
2) The ultrapeer overlay is a connected graph.

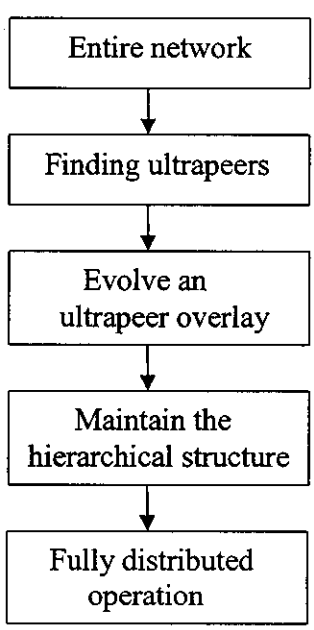

Fig. 2.2. Steps in Diff Search Algorithm

CHAPTER 3

DESIGN PRINCIPLES

3.1 HETEROGENEITY OF FILE SHARING

While the primary intent of the P2P concept is to blur the border between service providers and consumers and treat each participant equally, heterogeneity is an inherent characteristic of P2P systems. Peers vary in many aspects, such as network bandwidth, CPU power, and storage capacity. It is a natural choice that this heterogeneity is taken into account when designing P2P systems, such as the flow control and topology adaptation algorithm. Those algorithms try to balance the query forwarding load among peers according to their bandwidth resources to maximize the system capacity. In contrast to the heterogeneity of the query process capability, the heterogeneity of query reply capability has not been emphasized in P2P system design yet.

To investigate the heterogeneity of peers query answering capabilities in detail, a Gnutella trace collector based on the Limewire servant is implemented. By hooking the log functionality to the Limewire servant, all of the queries and responses passing by the trace collector are recorded. Aiming at collecting enough data within a short period of time, set the collector as an ultrapeer with 100 neighbors.

According to the RFC 1918 responses from IP addresses prefixed by 10/8,172.16/12, and 192.168/16 are filtered from the response trace file because those IP addresses are allocated to private networks and widely used in Network Address Translation (NAT). The responses coming from the IP addresses within those ranges cannot be guaranteed to be from the same host even if they have the same private IP address. Based on the analysis of the refined traces, the response distribution among peers and metrics of selecting ultrapeers are investigated.

3.2 EXISTING SYSTEM

In the existing system, the query that has to be searched is sent only to the neighboring peers that are very close to the querying peer and the indexing cost is very high. The actual concept is "can we search the files with minimal or zero indexing cost?" In the existing system bandwidth, CPU power, and storage capacity are only taken as vital parameters for selecting the ultrapeer

3.3 EFFECTIVE FILES

Peers vary in their query reply capabilities because of their file sharing heterogeneity. The trace analysis shows that very few peers share a large number of files, because query answering is the process of matching keywords with all the shared file names. However, the investigation shows that there is a weak correlation between query responses and the number of shared files. Our explanation is that some useless files make no contribution to the query answering, i.e., some files are never used to answer the queries. If we only count the files which have been used to answer the queries, the number of responses does show a strong correlation with the number of shared files. To distinguish those files from useless files, we define the files which have been used to answer the queries as Effective Files. The correlation between the number of responses and the number of effective files is illustrated in Fig.3.1. As we can see, the peers sharing more effective files have a greater tendency to answer queries. This correlation makes the number of effective files shared by a peer a good criterion to determine if the peer should be selected as an ultrapeer.

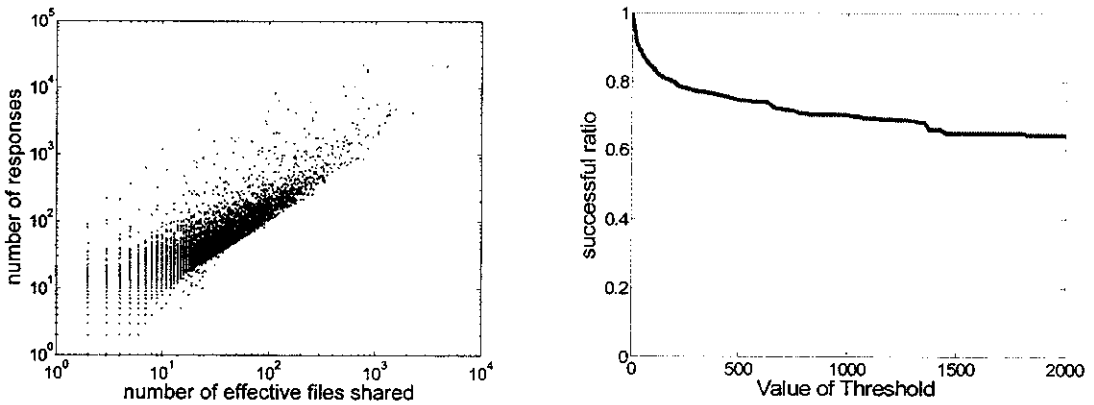

Fig. 3.1. Correlation between query response and number of files.

3.4 PROPOSED SYSTEM

In our proposed system we focused on solving the software heterogeneity and reducing the indexing cost. By setting a threshold of 100 effective files, the top 2 percent of peers are selected from 10,000 peers to form the ultrapeer overlay. More than 80 percent of queries can be replied to from the ultrapeer overlay, which is contrasted by the fact that less than 20 percent of queries can be answered by a cluster consisting of the same number of peers picked randomly as referred in Fig.3.2.

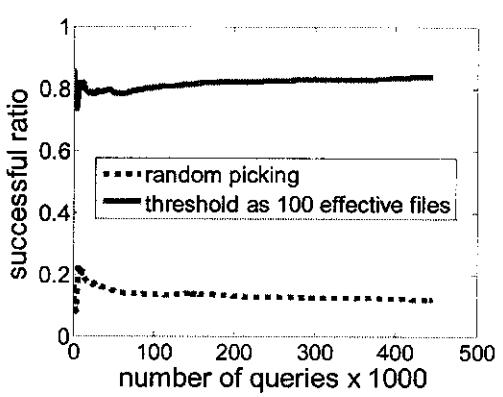

Fig. 3.2. Success ratio in ultrapeer overlay.

If peer "a" is disconnected from the ultra peer overlay, it will initiate the second round search, which may be replied to by some ultra peers with their IP addresses carried back in the query reply messages. In such a case, the

disjointed clusters are detected and the notification will be sent back to querying peer.

To hitchhike the overlay construction to the search messages, three bits need to be appended to the original query and reply messages. One bit is used in the query message to show whether the query is in the first round or second round. Two bits are used in the reply message to show whether the reply is from an ultra peer and to which round search the reply is responding.

CHAPTER 4

IMPLEMENTATION DETAILS

4.1 DIFFSEARCH ALGORITHM

Using the metric of effective files discussed above, the ultrapeers are self-aware by simply counting the number of shared files which have been visited before. As long as the number of shared effective files reaches the threshold, a peer can promote itself as an ultrapeer. In our DiffSearch algorithm, those peers should form an ultrapeer overlay and have higher priority to be queried. The challenge is how to find the ultrapeers and connect them into an overlay efficiently.

The basic objectives of our topology creation algorithm are:

1) Each leaf node has at least one ultrapeer as a neighbor and
2) The ultrapeer overlay is a connected graph.

4.2 FINDING ULTRAPEERS

For the first objective, the peer isolated from ultrapeers needs to find an ultrapeer as a neighbor. There are two basic approaches to finding an ultrapeer. The first one is a passive approach in which the isolated peer sends out an "ultrapeer search" message in the way of flooding and the ultrapeers respond back with their IP addresses. The second one is an active approach in which ultrapeers periodically publicize themselves to the P2P network and the isolated peers can find ultrapeers by overhearing the publicizing messages.

In the first approach, both the "ultrapeer search" message and response messages will cause tremendous overhead if all the isolated peers repeat the same process. Furthermore, the isolated peers will be overwhelmed by the response messages sent back from thousands of ultrapeers. In the second approach, the periodically publicized messages will also cause remarkable overhead, and the peer being publicized may be overwhelmed by the connection requests from a huge number of isolated peers, which will result in some heavily loaded ultrapeers.

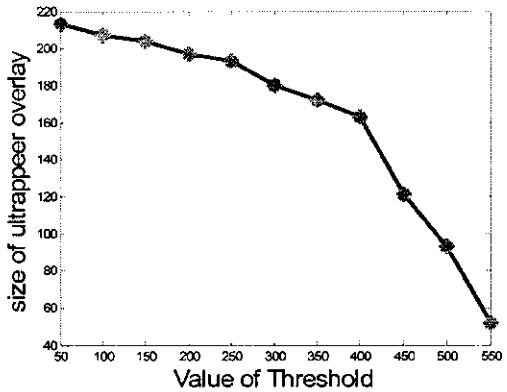

Fig. 4.1. Size of the ultrapeer overlay under different threshold values.

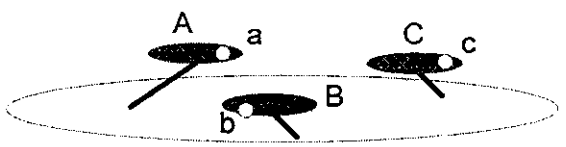

Fig. 4.2 A hierarchical P2P network.

One bit of data is appended to the reply message to indicate if the respondent is an ultrapeer. All of the replies received by isolated peers will be checked and the IP addresses will be extracted from the message sent from ultrapeers. Because ultrapeers have high query answering capabilities, the isolated peers can find ultrapeers with high probability after sending out several queries. In addition, no extra messages are required for the isolated peers to find the ultrapeers and the overhead is minimized by using only one bit of data in query replies.

4.3 EVOLVE AN ULTRAPEER OVERLAY:

To guarantee that each peer in the ultrapeer overlay can be reached by the first round search in DiffSearch, all the peers in the ultrapeer overlay should form a connected topology. The basic approach is to detect all the separated clusters consisting of ultrapeers and connect them with each other. Again, in a fully decentralized environment, a careful design is necessary to avoid too much

overhead caused by the cluster detection.

All of the separated clusters can be found if we flood the cluster detection message to the entire P2P network. However, this flooding may be initiated by multiple peers due to the lack of a synchronization mechanism. Since each peer is uncertain if other peers will start the cluster detection, all the peers must repeat the same operations. Furthermore, the cluster detection messages must be periodically flooded since the connected overlay will be disjointed by the dynamic movement of the P2P network.

To minimize the overhead of cluster detection, we follow the same design principles as finding ultrapeers, i.e., the peers' search operations will be reused by cluster detection. In our design, the cluster detection data hitchhikes to DiffSearch query messages. We use an example in Fig. 4.2 to illustrate the cluster detection, where the ultrapeer overlay is divided into three clusters. Suppose peer a fails to search keyword k in cluster A during the first round search of DiffSearch, which is caused by two possible reasons:

1) The keyword k is not shared by ultrapeers and

2) the file k is shared by ultrapeers, but they are located in separated clusters B or C.

For any case, peer "a" will initiate the second round search to the whole network. If the second round search is received by ultrapeers "b" and "c" in possession of file k, both "b" and "c" can make the conclusion that they are disconnected from the cluster A. Otherwise, the first round search would have received a successful response from peers "b" and "c", and it is unnecessary for peer "a" to initiate the second round query. As long as peers "b" and "c" detect the cluster disconnection, the notification messages will be carried back in the query replies made by peers "b" and "c". Based on the notification and the IP addresses in the query replies, peer "a" will set up new connections with peers "b" and "c" such that all the three disjointed clusters will be connected together. If the source peer "a" itself is not an ultrapeer, an ultrapeer from its neighbors will be picked up to set connections with peers "b" and "c". If peer "a" cannot find any ultrapeer in its neighbor list, peer an itself will be connected to peers "b" and "c". In

this case, peers "b" and "c" cannot be reached in the first round search just because peer "a" is isolated from the ultrapeer overlay.

One problem of the above algorithm is that clusters B and C may be connected with each other. In this case, one connection from "a" to either "b" or "c" is enough to connect all the clusters. We hope the number of newly added connections can be minimized so that the duplicated query messages in the ultrapeers overlay can be reduced. To eliminate the redundant connections, peer "a" has to judge if peers "b" and "c" are in a connected graph. Again, the testing approach for peer "b" to flood a message to peer "c" is not encouraged. An alternative method is that peer "a" only makes one connection with one of the two responding peers. This conservative approach will slow down the convergence speed but reduce redundant connections.

4.4 MAINTAINING THE HIERARCHICAL STRUCTURE:

From the above analysis, we show that all of the overlay constructing operations can be finished by only using the search messages. If peer a is disconnected from the ultrapeer overlay, it will initiate the second round search, which may be replied to by some ultrapeers with their IP addresses carried back in the query reply messages. In such a case, the disjointed clusters are detected and the notification will be sent back to querying peer a.

Due to the highly dynamic nature of P2P networks, the ultrapeer overlay may be broken by peer's ungraceful departures. However, as illustrated above, the incomplete- ness of the two-tier structure can be detected and repaired by ongoing queries of the DiffSearch. We show in our experiment that the DiffSearch approach is a self-maintained as described in Fig.4.3.

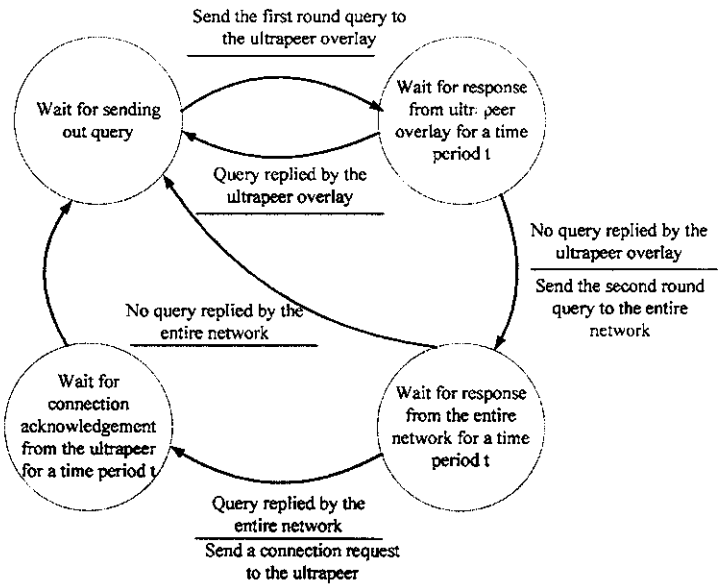

Fig. 4.3. Two round query operation of an individual peer.

4.5 FULLY DISTRIBUTED OPERATIONS:

The state transition of a peer's query operation is illustrated in Fig. 4. When a peer initiates a query, it will send the first round query to the ultrapeer overlay. If the first round search fails, it will send the second round search to the entire network. After it receives query replies to the second round search, the peer will pick up an ultrapeer from the reply messages and request a new connection to that ultrapeer. The entire operations above are completed locally by the querying peer without global information.

CHAPTER 5
IMPLEMENTATION TOOLS

5.1 IMPLEMENTATION TECHNIQUE

Our proposed DiffSearch algorithm is orthogonal to many other P2P search improvement solutions and, therefore, they can be combined together. Each peer overhears query responses and caches file names together with IP addresses of peers who share the queried files. When a peer receives a query, it will look up its local indices and cached indices as well. A response will be sent back if the query matches one of the two indices.

The cache can help reduce the query traffic because the cache indices increase the possibility for a query of being answered by nearby peers. In DiffSearch algorithm actually the query is sent to the ultra peer overlay and then to the entire network, but since we use the cache response concept first the query is searched in the cache and then to the ultrapeer overlay and finally to the entire network. Fig.5.1 shows the steps in our modified DiffSearch algorithm.

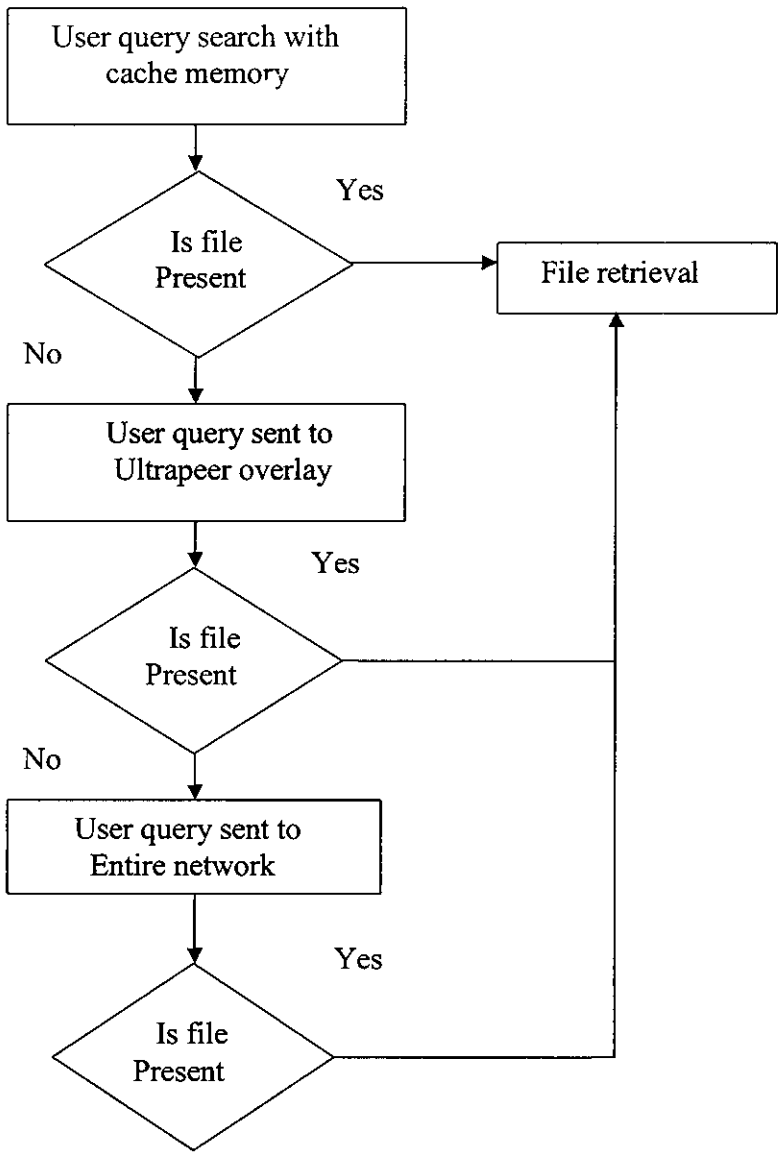

Fig.5.1. Steps in modified DiffSearch algorithm

CHAPTER 6

EXPERIMENTATION PROCEDURE AND RESULTS

This chapter describes our experimentation procedure and results. The following snapshots explain our implementation details.

6.1 TRANSFER OF FILES WHOSE IP IS NOT IN THE CACHE MEMORY

Fig 6.1 explains that initially the cache memory has the following data. It tells us that 1.txt is present in the 150.150.2.20 IP.

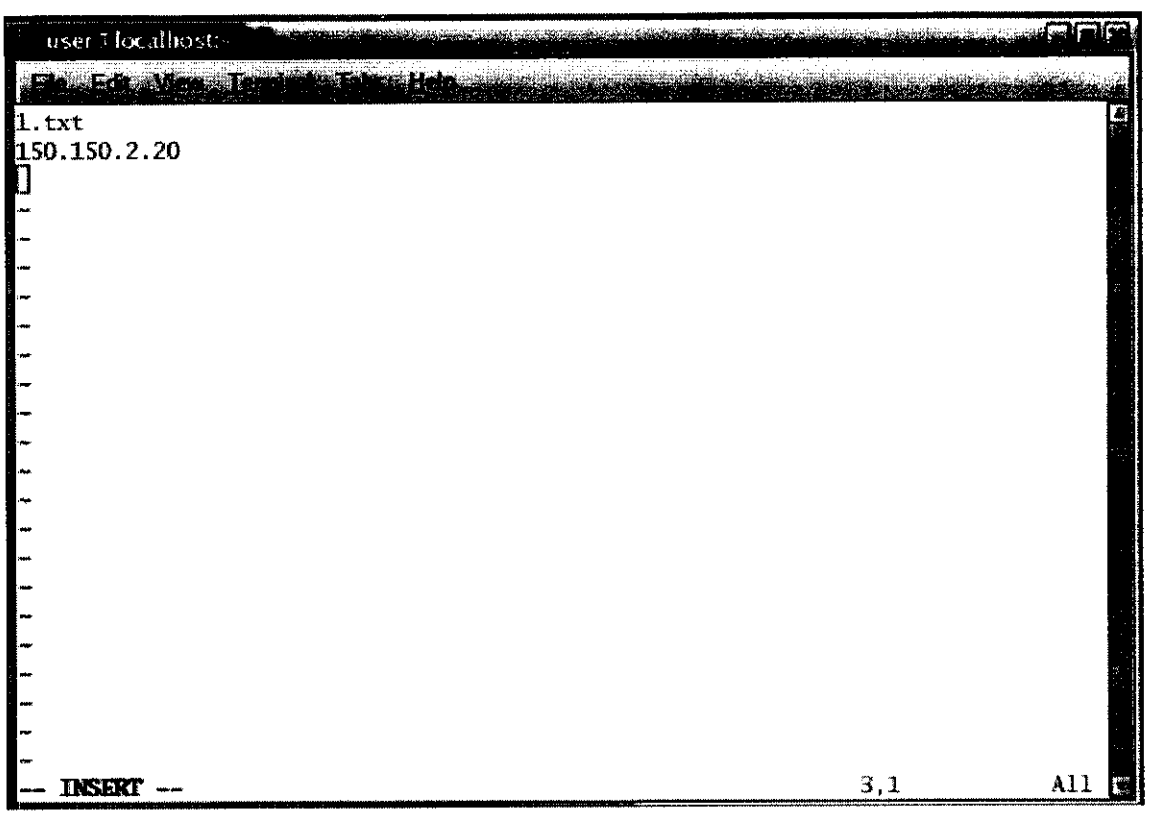

Fig.6.1. Initial contents in cache.

The Fig .6.2 shows that one of the ultrapeer is requesting for the file 2.txt and hence this is now query peer. The query is searched in the cache memory for the IP address and since the IP address of the peer is not present in the cache, the searching is evolved to the ultrapeer overlay. The Fig 6.3 and Fig 6.4 show that they are the other two ultrapeers and the Fig.6.5 shows that it is a peer under any one of the ultapeers.

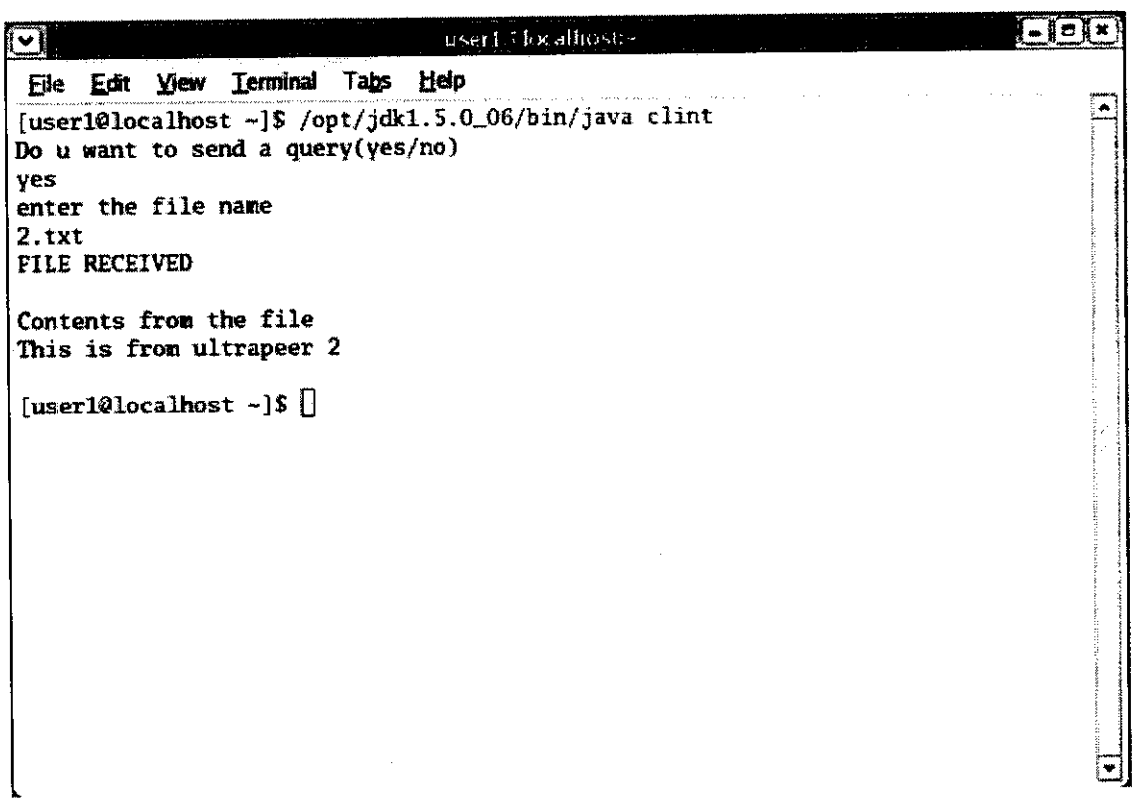

Fig.6.2.Query peer (2.txt)

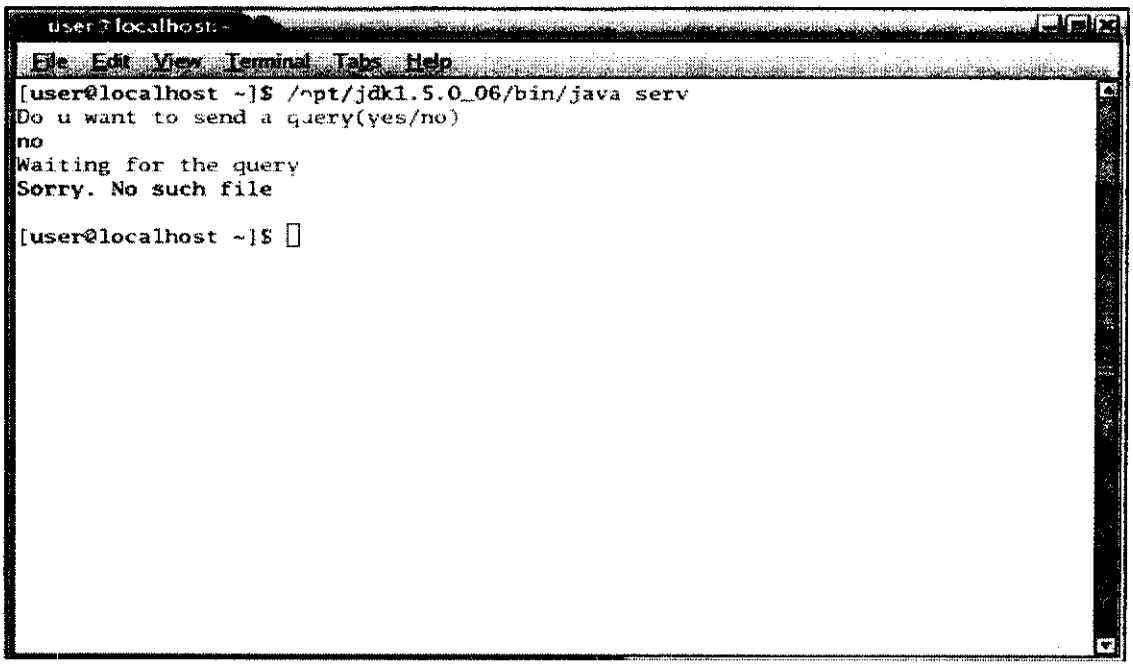

Fig.6.3.Response peer1

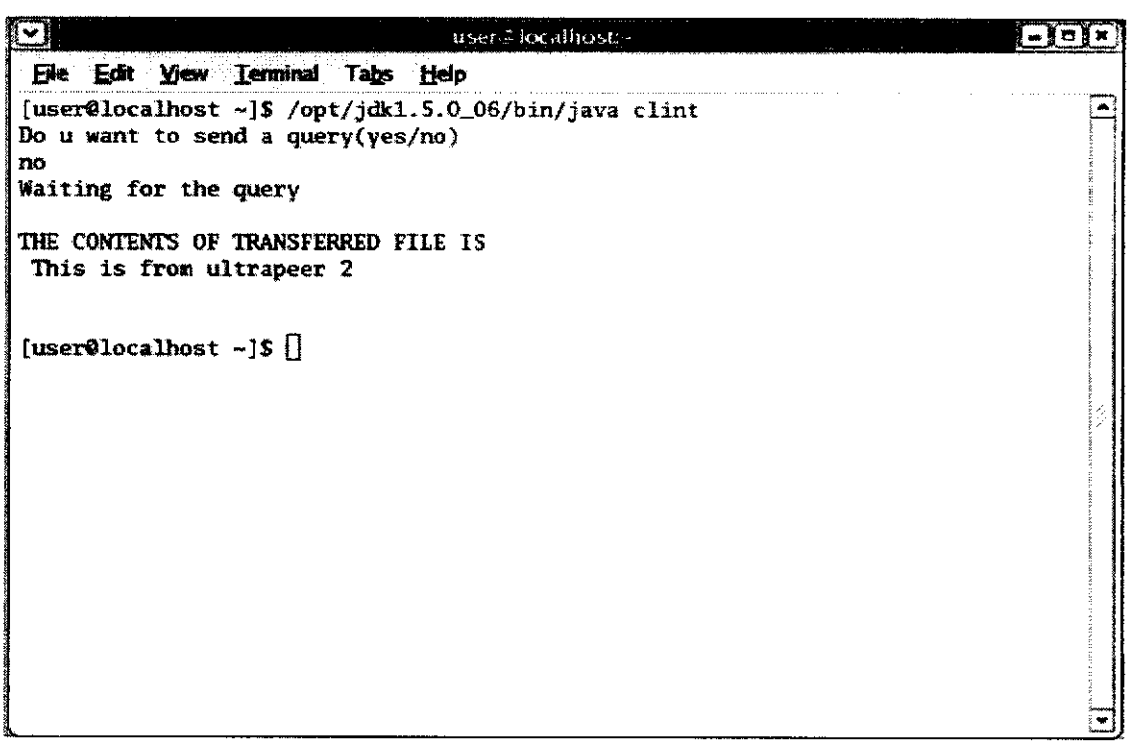

Fig.6.4. Response peer2

Here searching is not done, because this peer's supernode does not contain the specified file and so the searching is not done in this leafnode.The above structure is known as the two-tier hierarchical structure.

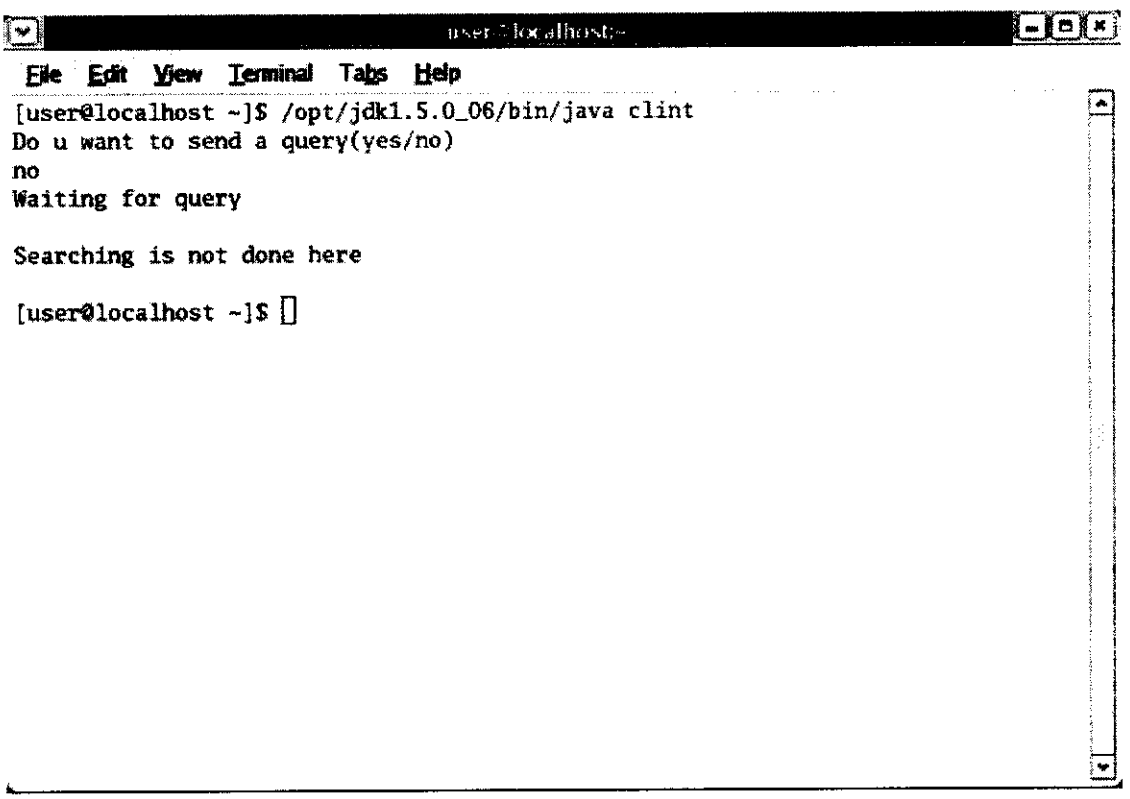

Fig.6.5 Response peer under Hierarchical structure

6.2 TRANSFER OF FILES WHOSE IP IS IN THE CACHE MEMORY

The Fig.6.6 shows that the following two files IP address are present in the cache memory.

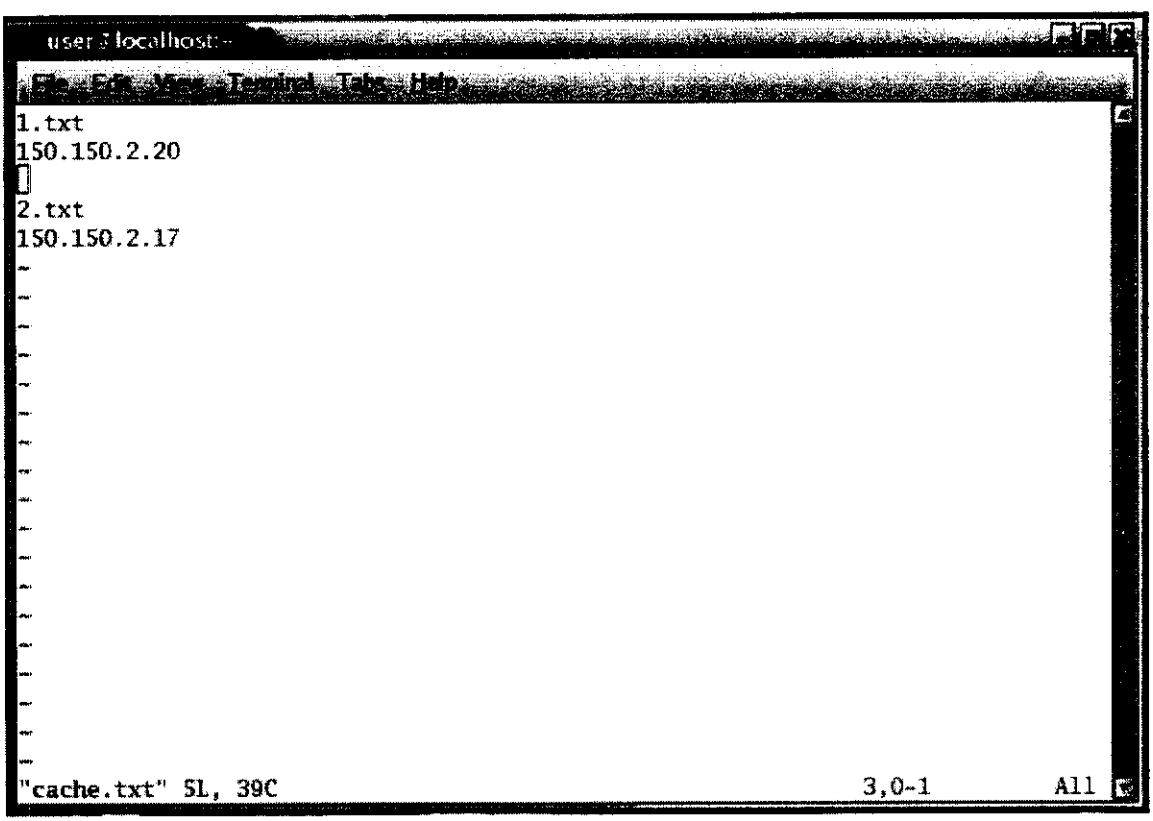

Fig.6.6.Cache content after first query response

Fig.6.7 shows that, it is requesting for the file 1.txt and since it is present in the cache memory the file present in 150.150.2.20 IP is searched directly from the cache and returned to the query peer. Fig.6.9 and Fig 6.10 show that searching is not done there.

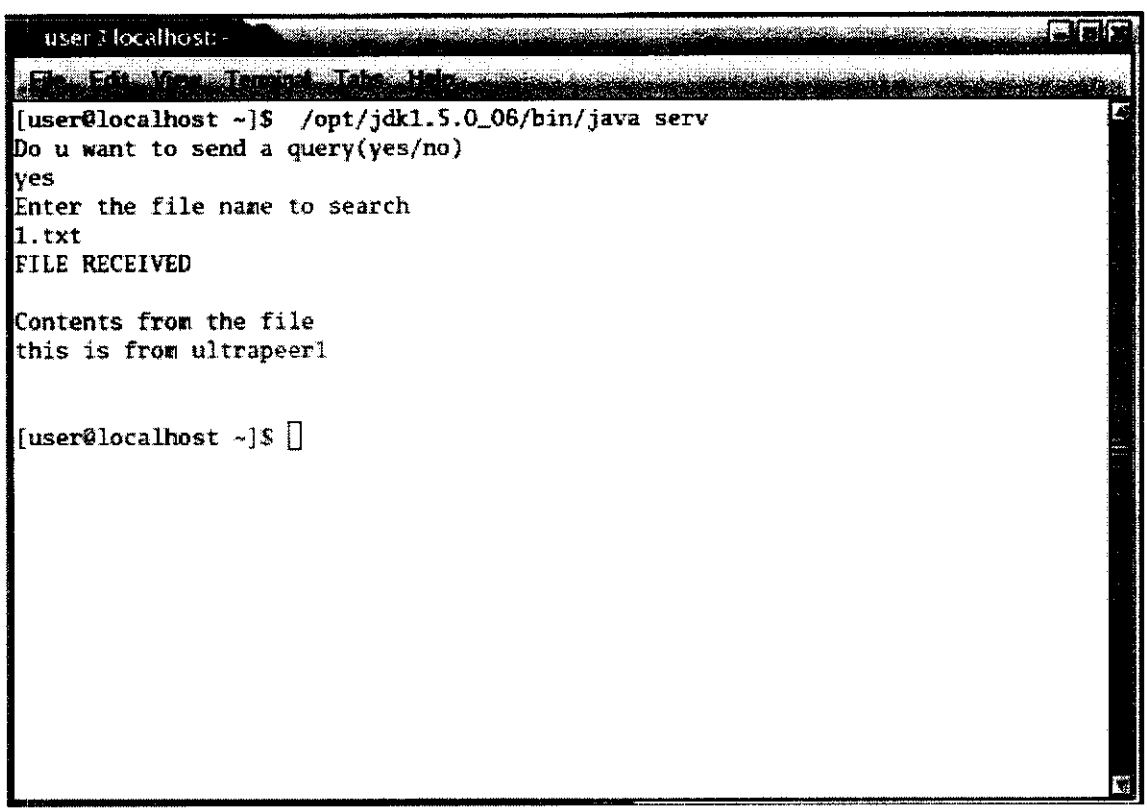

Fig.6.7 Query peer (1.txt)

Fig.6.8 is the peer with the IP address 150.150.2.20.

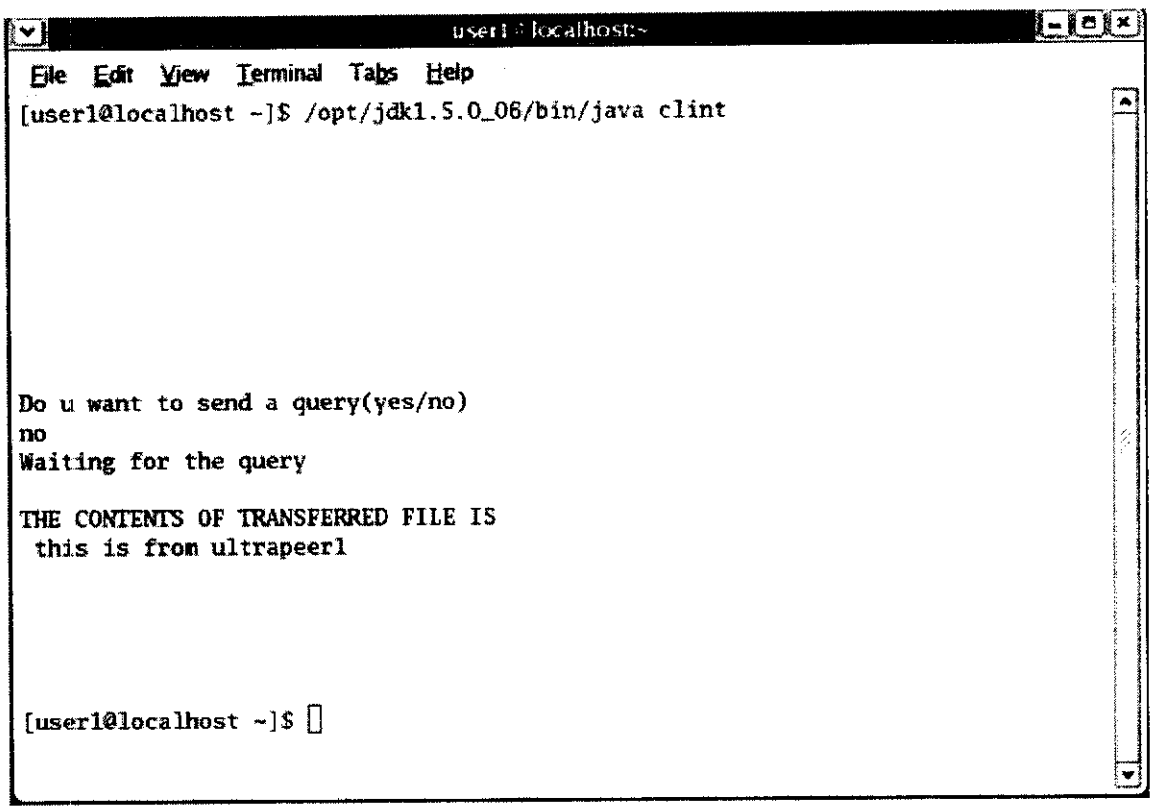

Fig.6.8.Response peer1

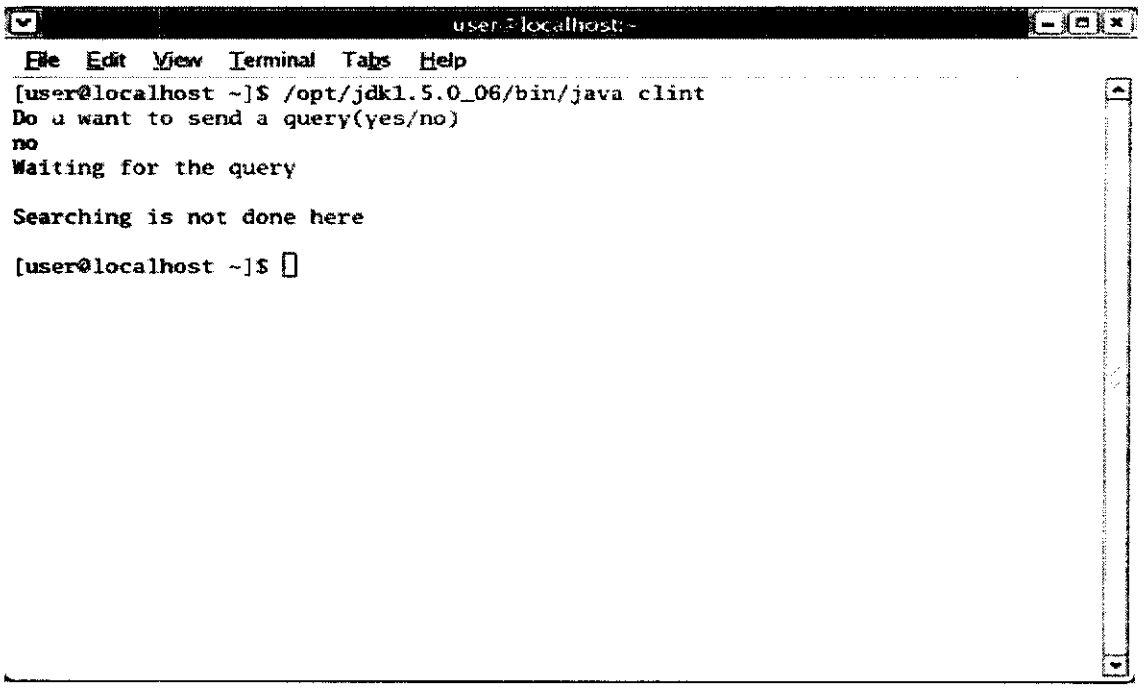

Fig.6.9.Response peer2

Fig.6.10 Response peer3 under hierarchical structure

6.3 TRANSFER OF FILES FROM HIERARCHICAL STRUCTURE

Fig.6.11 shows that the following two files IP address are present in the cache memory.Fig.6.12 shows the query peer and requests for the file 'hier.txt' and since it is in the leaf node of one of the ultrapeers, the leaf node which has the ultrapeer informs them to wait for few seconds since it needs to allow for the leaf node peer to transfer the specified file.

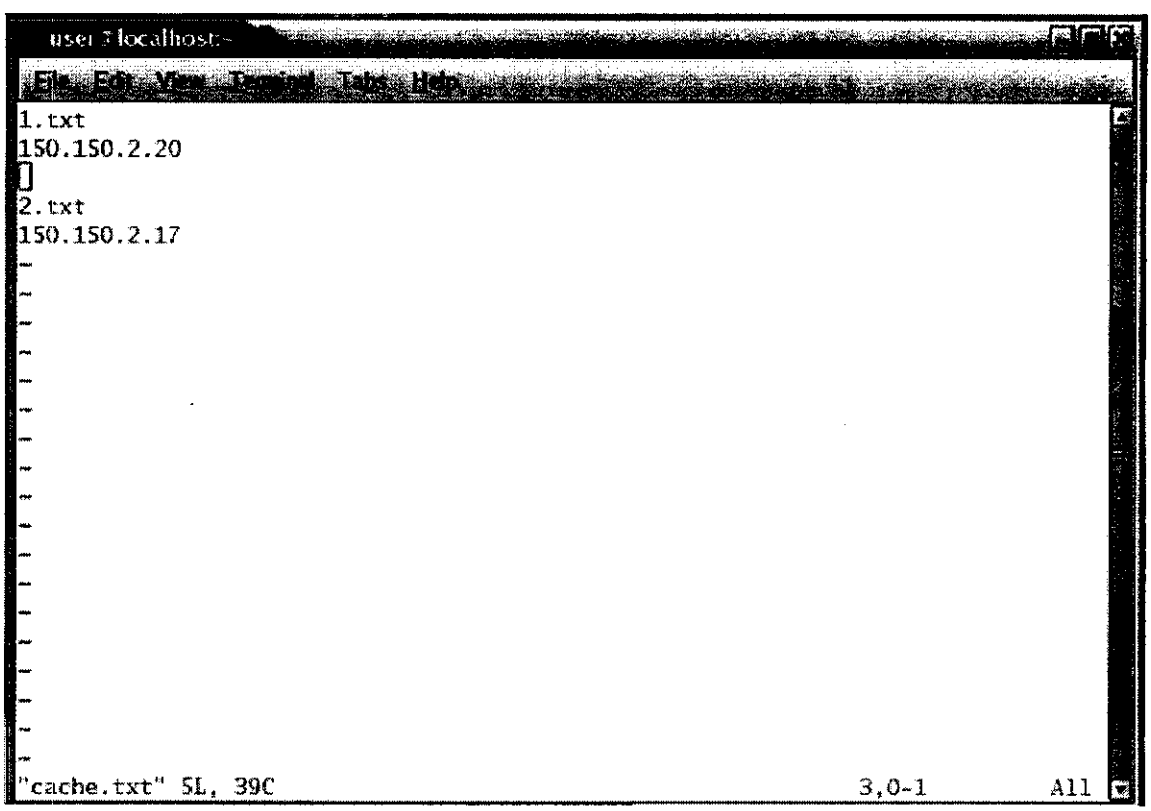

Fig.6.11. Cache content after query response

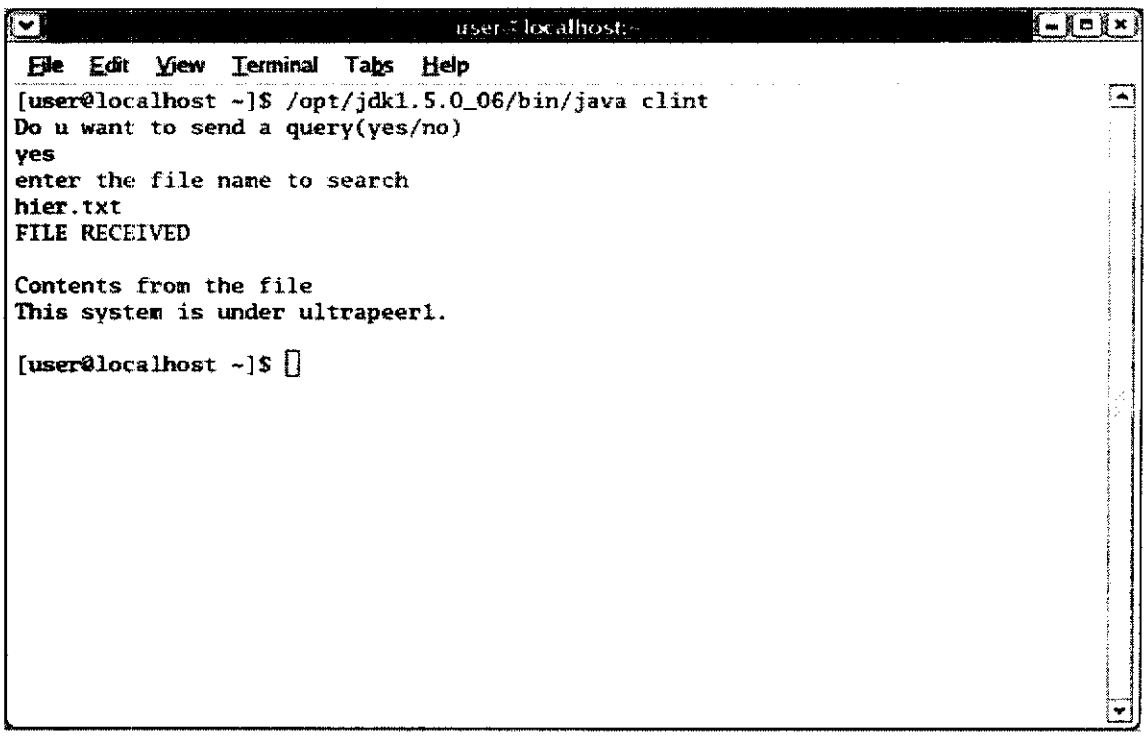

Fig.6.12. Query peer (hier.txt)

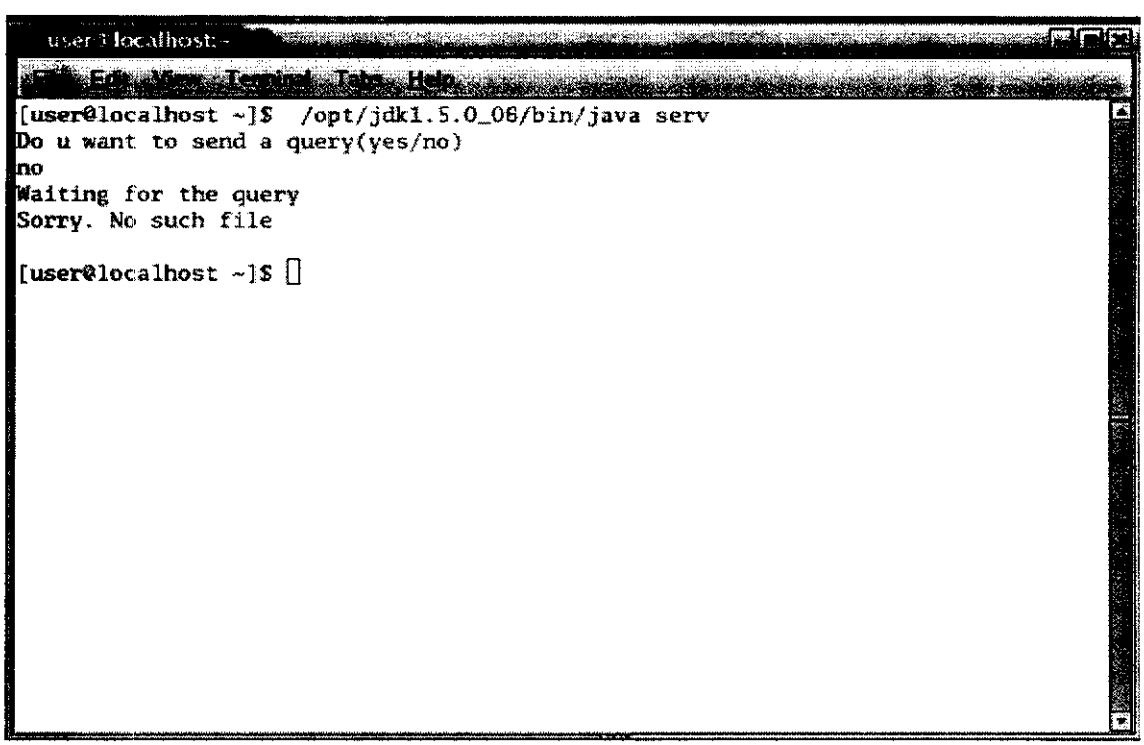

Fig.6.13.Response peer1

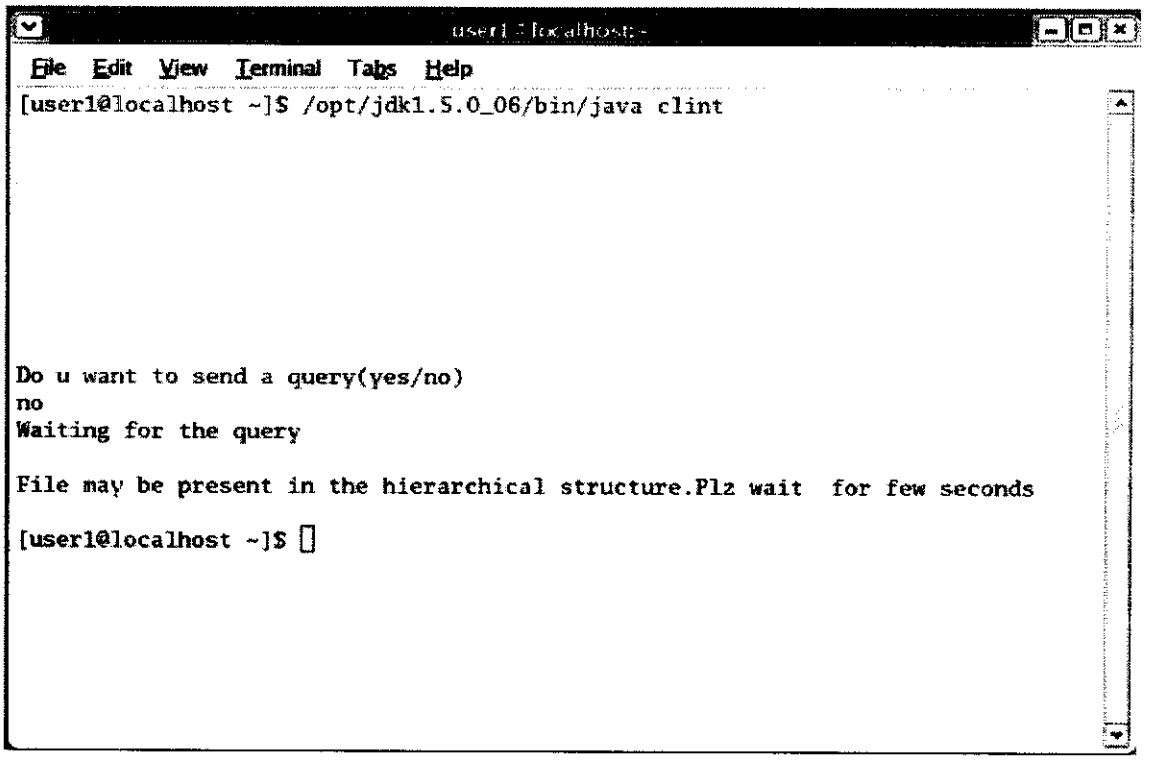

Fig.6.14.Response peer2

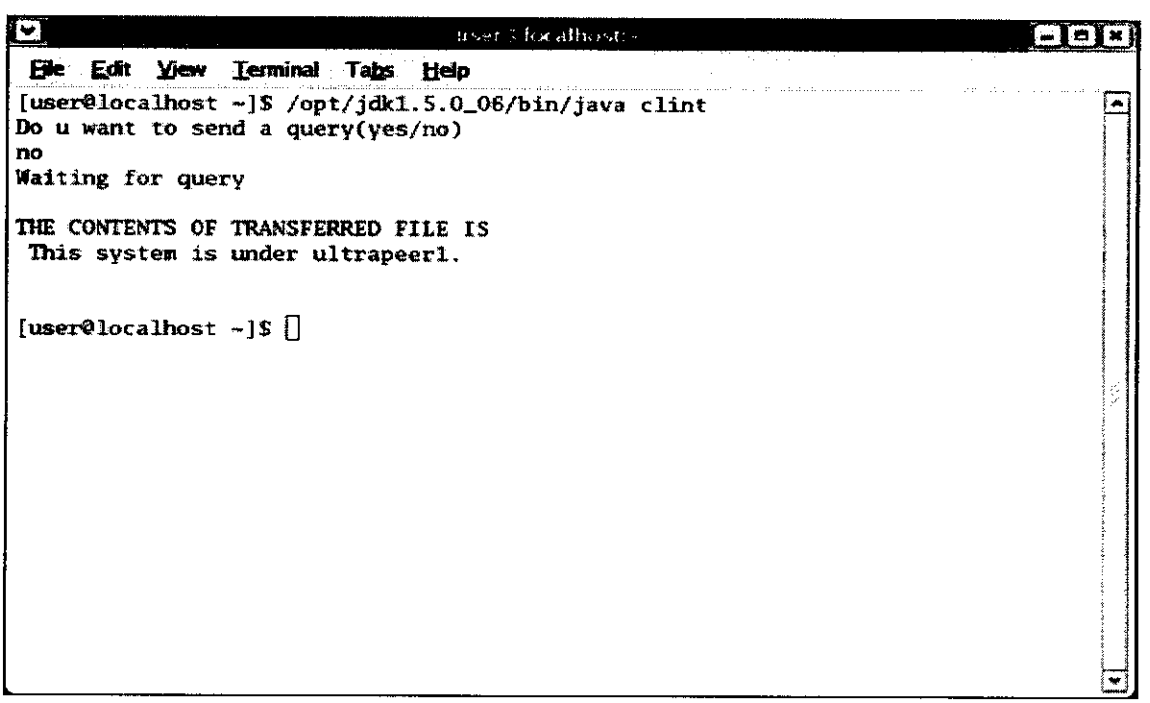

Fig.6.15. Response peer3 (from hierarchical structure)

Table.6.1: Sample Output

The following table shows the sample output of our implementation.

Hierarchy	**Ultrapeer1**	**Ultrapeer2**	**Isolatedpeer3 (under ultrapeer4)**	**Ultrapeer4**	**Files in Cache memory**
File present	1.c	2.c	3.c	4.c	1.c,4.c
Query request	No	Yes (1.c)	No	No	1.c,4.c
Response	File transferred	File 1.c received	Search not done	Search not done	1.c,4.c
Query request	No	Yes(4.c)	No	No	1.c,4.c
Response	Search not done	File 4.c received	Search not done	file transferred	1.c 4.c
Query request	No	Yes(3.c)	No	No	1.c 4.c
Response	File not found	File 3.c received	File transferred (its from hierarchical structure)	Please wait for few seconds	1.c,4.c,3.c

6.4 PERFORMANCE ANALYSIS:

We have compared the following three algorithms such as flooding search algorithm, Diff search algorithm and Diff search with cache memory. Fig 6.16 shows the performance analysis graph.In the graph we take average file retrieval time along Y-axis and number of ultrapeer along X-axis. This graph shows that our algorithm i.e., DiffSearch with Cache memory will retrieve the files quickly when compared with other algorithms.

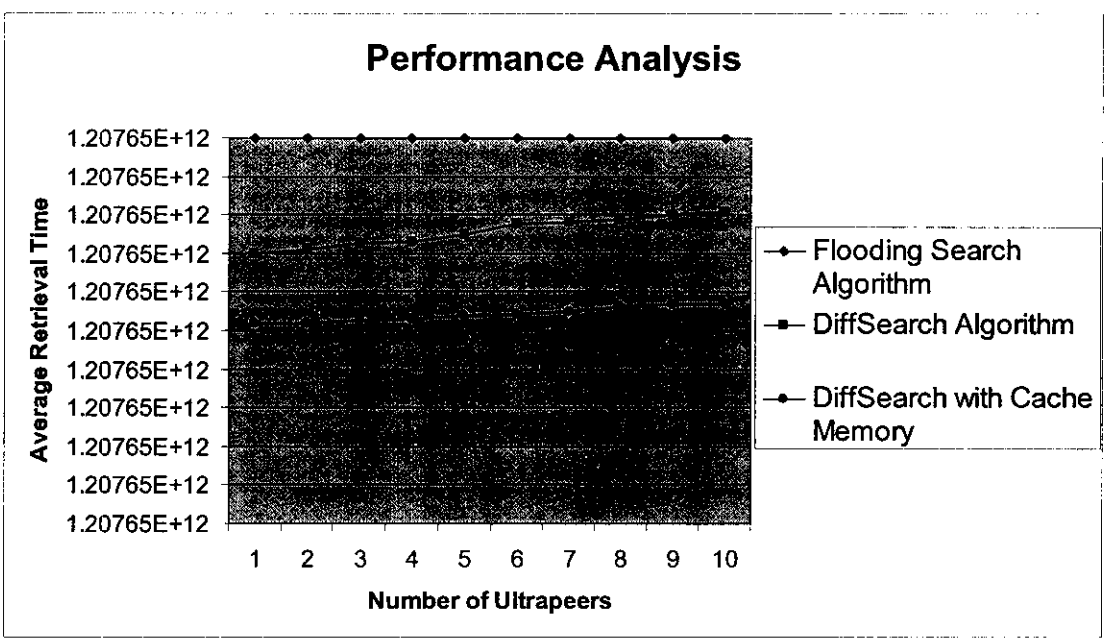

Fig.6.16. Performance analysis

7. APPENDICES

7.1 SOURCE CODE FOR ULTRAPEER 1

```
import java.io.*;
import java.net.*;
class P2P
{
public static DatagramSocket ds;
public static byte buffer1[]=new byte[1024];
public static byte buffer2[]=new byte[1024];
public static byte buffer3[]=new byte[1024];
public static void main(String args[])throws Exception
{
String
fully="",str="",str1="",query="",ser_file_name="",cache="cache.txt",dont="
dont";
Socket cl=null;
int serverport=2424,temp3=1,i=1;
File f1;
DataInputStream in=null,din=null;
DataOutputStream dot=null;
ds=new DatagramSocket(2323);
InetAddress ia=InetAddress.getByName("90.0.0.64");
try
{
cl=new Socket("90.0.0.64",2233);
in= new DataInputStream(System.in);
din=new DataInputStream(cl.getInputStream());
dot=new DataOutputStream(cl.getOutputStream());
}
```

```
catch(Exception e)
{
System.out.println(e);
}
if(cl!=null)
{
  fully="";
  FileWriter f2=new FileWriter("cache.txt");
  while((str1=din.readLine())!=null)
  {
   if(str1.compareTo("end")!=0)
   {
   fully=fully.concat(str1);
   fully=fully.concat("\n");
   }
   else
   {
     break;
   }
  }
  char buffer4[]=new char[fully.length()];
  fully.getChars(0,fully.length(),buffer4,0);
  f2.write(buffer4);
  f2.close();
 System.out.println("THIS PEER SHARE MORE NUMBER OF
EFFECTIVE FILES. SO IT IS A ULTRAPEER\n");
 System.out.println("Do u want to send a query(yes/no)");
 query=in.readLine();
 if(query.compareTo("no")==0)
 {
 buffer2=query.getBytes();
```

```
ds.send(new DatagramPacket(buffer2,query.length(),ia,serverport));
System.out.println("Waiting for the query\n");
DatagramPacket p=new DatagramPacket(buffer1,buffer1.length);
ds.receive(p);
ser_file_name=new String(p.getData(),0,p.getLength());
if(ser_file_name.compareTo("dont")!=0)
{
fl=new File(ser_file_name);
if(fl.exists())
{
dot.writeBytes("present");
dot.writeBytes("\n");
FileReader fr=new FileReader(ser_file_name);
BufferedReader br=new BufferedReader(fr);
System.out.println("THE CONTENTS OF TRANSFERRED FILE IS" );
while((str=br.readLine())!=null)
{
 System.out.println(" "+str);
 System.out.println("\n");
 dot.writeBytes(str);
 dot.writeBytes("\n") ;
}
}
else
{
  dot.writeBytes("absent");
  System.out.println("Sorry. No such file\n") ;
}
}
else
```

```
{
 System.out.println("Searching is not done here\n") ;
}
}
else
{
buffer2=query.getBytes();
ds.send(new DatagramPacket(buffer2,query.length(),ia,serverport));
System.out.println("enter the file name to search");
str=in.readLine();
FileReader fr=new FileReader("cache.txt") ;
BufferedReader br=new BufferedReader(fr);
String str2;
fully="";
while((str2=br.readLine())!=null)
{
 if(str2.compareTo(str)==0)
 {
  temp3=0;
  str2=br.readLine();
  String name=str;
  if(str2.compareTo("90.0.0.64")==0)
  {
   buffer3=str.getBytes();
   ds.send(new DatagramPacket(buffer3,str.length(),ia,serverport));
   buffer2=dont.getBytes();
   ds.send(new DatagramPacket(buffer2,dont.length(),ia,serverport));
   while((str1=din.readLine())!=null)
   {
    fully=fully.concat(str1);
```

```
    fully=fully.concat("\n");
   }
  }
  else if(str2.compareTo("90.0.0.66")==0)
  {
   buffer3=dont.getBytes();
   ds.send(new DatagramPacket(buffer3,dont.length(),ia,serverport));
   buffer2=str.getBytes();
   ds.send(new DatagramPacket(buffer2,str.length(),ia,serverport));
   while((str1=din.readLine())!=null)
   {
    fully=fully.concat(str1);
    fully=fully.concat("\n");
   }
  }
  System.out.println("FILE RECEIVED\n");
  System.out.println("Contents from the file");
  System.out.println(fully);
  char buffer[]=new char[fully.length()];
  fully.getChars(0,fully.length(),buffer,0);
  FileWriter f0=new FileWriter(str);
  f0.write(buffer);
  f0.close();
 }
}
if(temp3!=0)
{
buffer3=str.getBytes();
ds.send(new DatagramPacket(buffer3,str.length(),ia,serverport));
buffer2=str.getBytes();
```

```
ds.send(new DatagramPacket(buffer2,str.length(),ia,serverport));
while((str1=din.readLine())!=null)
{
fully=fully.concat(str1);
fully=fully.concat("\n");
}
System.out.println("FILE RECEIVED\n");
System.out.println("Contents from the file");
System.out.println(fully);
char buffer[]=new char[fully.length()];
fully.getChars(0,fully.length(),buffer,0);
FileWriter f0=new FileWriter(str);
f0.write(buffer);
f0.close();
}
}
}
din.close();
dot.close();
in.close();
cl.close();
}
}
```

7.2 SOURCE CODE FOR ULTRAPEER 2

```
import java.io.*;
import java.net.*;
class P2P
{
```

```
public static DatagramSocket ds,ds1;
public static void main(String args[])throws Exception
{
String
fully="",file_name="",query="",dont="dont",cache="cache.txt",ip1="90.0.0.
63",ip="90.0.0.66",ip2="90.0.0.64";
int temp1=1,temp2=1,clientport=2323,clientport1=2525,temp3=1;
File f1;
ServerSocket se=null;
ServerSocket se1=null;
Socket c1=null;
Socket c2=null;
DataInputStream din=null,din1=null;
DataInputStream in=null;
DataOutputStream dot=null,dot1=null;
byte buffer1[]=new byte[1024];
byte buffer2[]=new byte[1024];
byte buffer3[]=new byte[1024];
ds=new DatagramSocket(2121);
ds1=new DatagramSocket(2424);
BufferedReader dis=new BufferedReader(new
InputStreamReader(System.in));
InetAddress ia=InetAddress.getByName("90.0.0.63") ;
InetAddress ia1=InetAddress.getByName("90.0.0.66") ;
try
{
se=new ServerSocket(2222);
c1=se.accept();
se1=new ServerSocket(2233);
c2=se1.accept();
din= new DataInputStream(c1.getInputStream());
```

```
in= new DataInputStream(System.in);
dot= new DataOutputStream(cl.getOutputStream());
din1= new DataInputStream(c2.getInputStream());
dot1= new DataOutputStream(c2.getOutputStream());
}
catch(Exception e)
{
System.out.println(e);
}
try
{
 FileReader fr=new FileReader("cache.txt");
 BufferedReader br=new BufferedReader(fr);
 String str;
 while((str=br.readLine())!=null)
 {
  dot1.writeBytes(str);
  dot1.writeBytes("\n") ;
 }
dot1.writeBytes("end");
dot1.writeBytes("\n") ;
FileReader fr1=new FileReader("cache.txt");
BufferedReader br1=new BufferedReader(fr1);
while((str=br1.readLine())!=null)
{
 dot.writeBytes(str);
 dot.writeBytes("\n") ;
}
dot.writeBytes("end");
dot.writeBytes("\n") ;
```

```
System.out.println("THIS PEER SHARE MORE NUMBER OF EFFECTIVE FILES. SO IT IS A ULTRAPEER\n");
System.out.println("Do u want to send a query(yes/no)");
query=in.readLine();
if(query.compareTo("no")==0)
{
 System.out.println("Waiting for the query");
 DatagramPacket p=new DatagramPacket(buffer1,buffer1.length);
 ds.receive(p);
 String psx=new String(p.getData(),0,p.getLength());
 DatagramPacket p1=new DatagramPacket(buffer1,buffer1.length);
 ds1.receive(p1);
 String psx1=new String(p1.getData(),0,p1.getLength());
 if(psx.compareTo("yes")==0)
 {
  temp1=0;temp2=1;
 }
 else if(psx1.compareTo("yes")==0)
 {
  temp1=1;temp2=0;
 }
 if( (psx.compareTo("no")==0) && (psx1.compareTo("no")==0) )
 {
  temp1=0;temp2=1;
 }
}
}
catch(Exception e)
{
System.out.println(e);
}
```

```
if(cl!=null && temp1==0)
{
DatagramPacket p3=new DatagramPacket(buffer1,buffer1.length);
ds.receive(p3);
String m=new String(p3.getData(),0,p3.getLength());
DatagramPacket p2=new DatagramPacket(buffer1,buffer1.length);
ds.receive(p2);
String psx2=new String(p2.getData(),0,p2.getLength());
FileWriter f3=new FileWriter(cache,true);
if(m.compareTo(psx2)==0)
{
 char buffer4[]=new char[m.length()];
 m.getChars(0,m.length(),buffer4,0);
 f3.write(buffer4);
 f3.write("\n");
}
if(m.compareTo("null")==0)
{
}
else
{
buffer1=psx2.getBytes();
ds1.send(new DatagramPacket(buffer1,psx2.length(),ia,clientport));
if(m.compareTo("dont")!=0)
{
f1=new File(m);
if(f1.exists())
{
  if(m.compareTo(psx2)==0)
  {
```

```
   char buffer5[]=new char[ip2.length()];
   ip2.getChars(0,ip2.length(),buffer5,0);
   f3.write(buffer5);
   f3.write("\n");
  }
FileReader fr=new FileReader(m);
BufferedReader br=new BufferedReader(fr);
String str;
System.out.println("THE CONTENTS OF TRANSFERRED FILE IS" );
while((str=br.readLine())!=null)
{
 System.out.println(" "+str);
 dot.writeBytes(str);
 dot.writeBytes("\n") ;
}
}
else
{
 String str1;
 System.out.println("Sorry. No such file\n") ;
 str1=din1.readLine();
while((str1=din1.readLine())!=null)
 {
  dot.writeBytes(str1);
  dot.writeBytes("\n");
 }
if(m.compareTo(psx2)==0)
  {
   char buffer5[]=new char[ip1.length()];
   ip1.getChars(0,ip1.length(),buffer5,0);
```

```
   f3.write(buffer5);
   f3.write("\n");
  }
}
}
else
{
 System.out.println("Searching is not done here\n") ;
  String str1;
 str1=din1.readLine();
 while((str1=din1.readLine())!=null)
 {
  dot.writeBytes(str1);
  dot.writeBytes("\n");
 }
}
}
f3.close();
}
if(c2!=null && temp2==0)
{
DatagramPacket p3=new DatagramPacket(buffer1,buffer1.length);
ds1.receive(p3);
String m=new String(p3.getData(),0,p3.getLength());
DatagramPacket p2=new DatagramPacket(buffer1,buffer1.length);
ds1.receive(p2);
String psx2=new String(p2.getData(),0,p2.getLength());
FileWriter f3=new FileWriter(cache,true);
if(m.compareTo(psx2)==0)
{
```

```
 char buffer4[]=new char[m.length()];
 m.getChars(0,m.length(),buffer4,0);
 f3.write(buffer4);
 f3.write("\n");
}
if(m.compareTo("null")==0)
{
}
else
{
buffer1=psx2.getBytes();
ds1.send(new DatagramPacket(buffer1,psx2.length(),ia1,clientport1));
if(m.compareTo("dont")!=0)
{
f1=new File(m);
if(f1.exists())
{

  if(m.compareTo(psx2)==0)
  {
   char buffer5[]=new char[ip2.length()];
   ip2.getChars(0,ip2.length(),buffer5,0);
   f3.write(buffer5);
   f3.write("\n");
  }
FileReader fr=new FileReader(m);
BufferedReader br=new BufferedReader(fr);
String str;
System.out.println("THE CONTENTS OF TRANSFERRED FILE IS" );
while((str=br.readLine())!=null)
```

```
{
 System.out.println(" "+str);
 dot1.writeBytes(str);
 dot1.writeBytes("\n") ;
}
}
else
{
 System.out.println("Sorry. No such file\n") ;
 String str1;
 str1=din.readLine();
 while((str1=din.readLine())!=null)
 {
  dot1.writeBytes(str1);
  dot1.writeBytes("\n");
 }
 if(m.compareTo(psx2)==0)
   {
    char buffer5[]=new char[ip.length()];
    ip.getChars(0,ip.length(),buffer5,0);
    f3.write(buffer5);
    f3.write("\n");
   }
}
}
else
{
  System.out.println("Searching is not done here\n") ;
  String str1;
 str1=din.readLine();
```

```
while((str1=din.readLine())!=null)
{
 dot1.writeBytes(str1);
 dot1.writeBytes("\n");
}
}
}
f3.close();
}
if(temp1==1 && temp2==1)
{
System.out.println("Enter the file name to search");
file_name=dis.readLine();
FileReader fr=new FileReader("cache.txt") ;
BufferedReader br=new BufferedReader(fr);
String str;
while((str=br.readLine())!=null)
{
 if(str.compareTo(file_name)==0)
 {
  temp3=0;
  str=br.readLine();
  String name=str;
  buffer1=file_name.getBytes();
  buffer2=dont.getBytes();
  String str1;
  if(name.compareTo("90.0.0.63")==0)
  {
   ds.send(new DatagramPacket(buffer1,file_name.length(),ia,clientport));
   ds1.send(new DatagramPacket(buffer2,dont.length(),ia1,clientport1));
```

```
String psx3=din1.readLine();
if(psx3.compareTo("present")==0)
{
 while((str1=din1.readLine())!=null)
 {
  fully=fully.concat(str1);
  fully=fully.concat("\n");
 }
}
}
else if(name.compareTo("90.0.0.66")==0)
{
 ds.send(new DatagramPacket(buffer2,dont.length(),ia,clientport));
 ds1.send(new
DatagramPacket(buffer1,file_name.length(),ia1,clientport1));
 String psx2=din.readLine();
 if(psx2.compareTo("present")==0)
 {
  while((str1=din.readLine())!=null)
  {
   fully=fully.concat(str1);
   fully=fully.concat("\n");
  }
 }
}
System.out.println("FILE RECEIVED\n");
System.out.println("Contents from the file");
System.out.println(fully);
char buffer[]=new char[fully.length()];
fully.getChars(0,fully.length(),buffer,0);
FileWriter f0=new FileWriter(file_name);
```

```
 f0.write(buffer);
 f0.close();
 }
 }
 if(temp3!=0)
 {
 buffer1=file_name.getBytes();
 ds.send(new DatagramPacket(buffer1,file_name.length(),ia,clientport));
 ds1.send(new
DatagramPacket(buffer1,file_name.length(),ia1,clientport1));
 String str1;
 String psx2=din.readLine();
 String psx3=din1.readLine();
 FileWriter f2=new FileWriter(cache,true);
 char buffer4[]=new char[file_name.length()];
 file_name.getChars(0,file_name.length(),buffer4,0);
 f2.write(buffer4);
 f2.write("\n");
 if(psx2.compareTo("present")==0)
 {
  while((str1=din.readLine())!=null)
  {
   fully=fully.concat(str1);
   fully=fully.concat("\n");
  }
   char buffer5[]=new char[ip.length()];
   ip.getChars(0,ip.length(),buffer5,0);
   f2.write(buffer5);
   f2.write("\n");
 }
 if(psx3.compareTo("present")==0)
```

```
{
 while((str1=din1.readLine())!=null)
 {
  fully=fully.concat(str1);
  fully=fully.concat("\n");
 }
  char buffer6[]=new char[ip1.length()];
  ip1.getChars(0,ip1.length(),buffer6,0);
  f2.write(buffer6);
  f2.write("\n");
 }
 System.out.println("FILE RECEIVED\n");
 System.out.println("Contents from the file");
 System.out.println(fully);
 char buffer[]=new char[fully.length()];
 fully.getChars(0,fully.length(),buffer,0);
 FileWriter f0=new FileWriter(file_name);
 f0.write(buffer);
 f0.close();
 f2.close();
 }
 }
 din.close();
 dot.close();
 se.close();
 cl.close();
 din1.close();
 dot1.close();
 se1.close();
 c2.close();
```

```
}
}
```

7.3 SOURCE CODE FOR ISOLATED PEER 3 (UNDER ULTRAPEER 4)

```
import java.io.*;
import java.net.*;
class P2P
{
public static DatagramSocket ds;
public static byte buffer1[]=new byte[1024];
public static void main(String args[])throws Exception
{
String fully="",str="",query="";
Socket cl=null;
File f1;
int serverport=2929;
DataInputStream din=null,in=null;
DataOutputStream dot=null;
ds=new DatagramSocket(2828);
InetAddress ia=InetAddress.getByName("90.0.0.64");
try
{
cl=new Socket("90.0.0.66",3232);
in= new DataInputStream(System.in);
din=new DataInputStream(cl.getInputStream());
dot=new DataOutputStream(cl.getOutputStream());
}
catch(Exception e)
{
System.out.println(e);
```

```
}
if(cl!=null)
{
System.out.println("THIS PEER IS UNDER ULTRAPEER 1\n");
System.out.println("Do u want to send a query(yes/no)");
query=in.readLine();
dot.writeBytes(query);
dot.writeBytes("\n");
if(query.compareTo("no")==0)
{
System.out.println("Waiting for query\n");
DatagramPacket p=new DatagramPacket(buffer1,buffer1.length);
ds.receive(p);
String ser_file_name=new String(p.getData(),0,p.getLength());
fl=new File(ser_file_name);
if(ser_file_name.compareTo("not")==0)
{
 System.out.println("Searching is not done here\n");
}
else
{
if(fl.exists())
{
FileReader fr=new FileReader(ser_file_name);
BufferedReader br=new BufferedReader(fr);
System.out.println("THE CONTENTS OF TRANSFERRED FILE IS" );
while((str=br.readLine())!=null)
{
 System.out.println(" "+str);
 System.out.println("\n");
```

```
dot.writeBytes(str);
dot.writeBytes("\n") ;
}
}
else
{
 System.out.println("Sorry. No such file\n") ;
}
}
}
else
{
System.out.println("enter the file name");
str=in.readLine();
dot.writeBytes(str);
dot.writeBytes("\n");
String str1;
while((str1=din.readLine())!=null)
{
fully=fully.concat(str1);
fully=fully.concat("\n");
}
System.out.println("FILE RECEIVED\n");
System.out.println("Contents from the file");
System.out.println(fully);
char buffer[]=new char[fully.length()];
fully.getChars(0,fully.length(),buffer,0);
FileWriter f0=new FileWriter(str);
f0.write(buffer);
f0.close();
```

```
}
}
din.close();
dot.close();
in.close();
cl.close();
}
}
```

7.4 SOURCE CODE FOR ULTRAPEER 4

```
import java.io.*;
import java.net.*;
class P2P
{
public static DatagramSocket ds,ds1;
public static byte buffer1[]=new byte[1024];
public static byte buffer2[]=new byte[1024];
public static byte buffer3[]=new byte[1024];
public static void main(String args[])throws Exception
{
String
fully="",str="",str1="",query="",ser_file_name="",hierar="not",dont="dont";
Socket cl=null,cc1=null;
ServerSocket ss1=null;
char a='h';
File f1;
int serverport=2121,clientport=2828,temp3=1;
DataInputStream in=null,din=null,din1=null;
DataOutputStream dot=null,dot1=null;
```

```
ds=new DatagramSocket(2525);
ds1=new DatagramSocket(2929);
InetAddress ia=InetAddress.getByName("90.0.0.64");
InetAddress ia1=InetAddress.getByName("90.0.0.65");
try
{
cl=new Socket("90.0.0.64",2222);
in= new DataInputStream(System.in);
din=new DataInputStream(cl.getInputStream());
dot=new DataOutputStream(cl.getOutputStream());
ss1=new ServerSocket(3232);
cc1=ss1.accept();
din1=new DataInputStream(cc1.getInputStream());
dot1=new DataOutputStream(cc1.getOutputStream());
}
catch(Exception e)
{
System.out.println(e);
}
if(cl!=null)
{
  fully="";
  FileWriter f2=new FileWriter("cache.txt");
  while((str1=din.readLine())!=null)
  {
   if(str1.compareTo("end")!=0)
   {
   fully=fully.concat(str1);
   fully=fully.concat("\n");
   }
```

```
      else
      {
        break;
      }
     }
     char buffer4[]=new char[fully.length()];
     fully.getChars(0,fully.length(),buffer4,0);
     f2.write(buffer4);
     f2.close();
   System.out.println("THIS PEER SHARE MORE NUMBER OF
EFFECTIVE FILES. SO IT IS A ULTRAPEER\n");
   System.out.println("Do u want to send a query(yes/no)");
   query=in.readLine();
   if(query.compareTo("no")==0)
   {
    buffer2=query.getBytes();
    ds.send(new DatagramPacket(buffer2,query.length(),ia,serverport));
   System.out.println("Waiting for the query\n");
   String from_hier=din1.readLine();
   if(from_hier.compareTo("yes")==0)
   {
    str=din1.readLine();
    FileReader fr=new FileReader("cache.txt") ;
   BufferedReader br=new BufferedReader(fr);
   String str2;
   fully="";
   while((str2=br.readLine())!=null)
   {
    if(str2.compareTo(str)==0)
    {
    temp3=0;
```

```
 str2=br.readLine();
 String name=str;
 if(str2.compareTo("90.0.0.64")==0)
 {
  buffer3=str.getBytes();
  ds.send(new DatagramPacket(buffer3,str.length(),ia,serverport));
  buffer2=dont.getBytes();
  ds.send(new DatagramPacket(buffer2,dont.length(),ia,serverport));
 }
 else if(str2.compareTo("90.0.0.63")==0)
 {
  buffer3=dont.getBytes();
  ds.send(new DatagramPacket(buffer3,dont.length(),ia,serverport));
  buffer2=str.getBytes();
  ds.send(new DatagramPacket(buffer2,str.length(),ia,serverport));
 }
 }
}
if(temp3==1)
{
 buffer3=str.getBytes();
 ds.send(new DatagramPacket(buffer3,str.length(),ia,serverport));
 buffer2=str.getBytes();
 ds.send(new DatagramPacket(buffer2,str.length(),ia,serverport));
}
 while((str1=din.readLine())!=null)
 {
   dot1.writeBytes(str1);
   dot1.writeBytes("\n");
 }
```

```
}
else
{
DatagramPacket p=new DatagramPacket(buffer1,buffer1.length);
ds.receive(p);
ser_file_name=new String(p.getData(),0,p.getLength());
char ch=ser_file_name.charAt(0);
if(ser_file_name.compareTo("dont")!=0)
{
f1=new File(ser_file_name);
if(f1.exists())
{
dot.writeBytes("present");
dot.writeBytes("\n");
FileReader fr=new FileReader(ser_file_name);
BufferedReader br=new BufferedReader(fr);
System.out.println("THE CONTENTS OF TRANSFERRED FILE IS" );
while((str=br.readLine())!=null)
{
 System.out.println(" "+str);
 System.out.println("\n");
 dot.writeBytes(str);
 dot.writeBytes("\n") ;
}
buffer2=hierar.getBytes();
ds1.send(new DatagramPacket(buffer2,hierar.length(),ia1,clientport));
}
//else if(ser_file_name.compareTo("hier.txt")==0)
else if(ch==a)
{
```

```
    System.out.println("File may be present in the hierarchical structure.Plz
wait  for few seconds\n");
     buffer2=ser_file_name.getBytes();
     ds1.send(new
DatagramPacket(buffer2,ser_file_name.length(),ia1,clientport));
     dot.writeBytes("present");
     dot.writeBytes("\n");
     while((str1=din1.readLine())!=null)
     {
     dot.writeBytes(str1);
     dot.writeBytes("\n");
     }
   }
   else
   {
    dot.writeBytes("absent");
     buffer2=hierar.getBytes();
     ds1.send(new DatagramPacket(buffer2,hierar.length(),ia1,clientport));
   System.out.println("Sorry. No such file\n") ;
   }
   }
   else
   {
    buffer2=hierar.getBytes();
    ds1.send(new DatagramPacket(buffer2,hierar.length(),ia1,clientport));
    System.out.println("Searching is not done here\n") ;
   }
   }
   }
   else
   {
```

```
buffer2=hierar.getBytes();
ds1.send(new DatagramPacket(buffer2,hierar.length(),ia1,clientport));
buffer2=query.getBytes();
ds.send(new DatagramPacket(buffer2,query.length(),ia,serverport));
System.out.println("enter the file name");
str=in.readLine();
FileReader fr=new FileReader("cache.txt") ;
BufferedReader br=new BufferedReader(fr);
String str2;
fully="";
while((str2=br.readLine())!=null)
{
 if(str2.compareTo(str)==0)
 {
 temp3=0;
 str2=br.readLine();
 String name=str;
 if(str2.compareTo("90.0.0.64")==0)
 {
  buffer3=str.getBytes();
  ds.send(new DatagramPacket(buffer3,str.length(),ia,serverport));
  buffer2=dont.getBytes();
  ds.send(new DatagramPacket(buffer2,dont.length(),ia,serverport));
  while((str1=din.readLine())!=null)
  {
   fully=fully.concat(str1);
   fully=fully.concat("\n");
  }
 }
 else if(str2.compareTo("90.0.0.63")==0)
```

```
{
 buffer3=dont.getBytes();
 ds.send(new DatagramPacket(buffer3,dont.length(),ia,serverport));
 buffer2=str.getBytes();
 ds.send(new DatagramPacket(buffer2,str.length(),ia,serverport));
 while((str1=din.readLine())!=null)
 {
   fully=fully.concat(str1);
   fully=fully.concat("\n");
 }
}
System.out.println("FILE RECEIVED\n");
 System.out.println("Contents from the file");
 System.out.println(fully);
 char buffer[]=new char[fully.length()];
 fully.getChars(0,fully.length(),buffer,0);
 FileWriter f0=new FileWriter(str);
 f0.write(buffer);
 f0.close();
}
}
if(temp3!=0)
{
buffer3=str.getBytes();
ds.send(new DatagramPacket(buffer3,str.length(),ia,serverport));
buffer2=str.getBytes();
ds.send(new DatagramPacket(buffer2,str.length(),ia,serverport));
while((str1=din.readLine())!=null)
{
fully=fully.concat(str1);
```

```
fully=fully.concat("\n");
}
System.out.println("FILE RECEIVED\n");
System.out.println("Contents from the file");
System.out.println(fully);
char buffer[]=new char[fully.length()];
fully.getChars(0,fully.length(),buffer,0);
FileWriter f0=new FileWriter(str);
f0.write(buffer);
f0.close();
}
}
}
din.close();
dot.close();
in.close();
cl.close();
}
}
```

CHAPTER 8
CONCLUSION AND FUTURE WORK

8.1 CONCLUSION

In the DiffSearch algorithm, a fully distributed approach which can evolve a two-tier hierarchical structure from a spontaneously formed ad hoc P2P network is proposed. By hitchhiking the topology operations to query/ reply messages and prompting content-rich peers to the ultrapeer overlay, the DiffSearch algorithm can achieve significant performance improvement with a little overhead on topology maintenance and index operations.

8.2 FUTURE WORK

The future work includes extending the number of peers and threshold values to more than 100. It is notable that the DiffSearch algorithm is orthogonal to many existing P2P optimization algorithms. For example, the random walk or DHT can still be applied in the ultrapeer overlay to further reduce the search traffic, which could be investigated in future.

REFERENCES

1) Cooper. B and Garcia-Molina. H, "SIL: Modeling and Measuring Scalable Peer-to-Peer Search Networks," http://dpubs.stanford. Edu: 8090/pub/2003-12, 2003.

2) Patro S and Hu Y. C, "Transparent Query Caching in Peer-to-Peer Overlay Networks," Proc. 17th Int'l Parallel and Distributed Processing Symp. (IPDPS), 2003.

3) Wang Chen, and Xiao Li, "An Effective P2P Search Scheme to Exploit File Sharing Heterogeneity" IEEE Transactions on Parallel and Distributed Computing, vol.18, No. 2, PP 145 – 156, February 2008.

4) Yang B and Garcia-Molina. H, "Efficient Search in Peer-to-Peer Networks," Proc.Int'l Conf. Distributed Computing Systems (ICDCS), 2002.

5) Zhao.B. Y , Huang. L, Stribling. J, Rhea. S. C, Joseph.A . D, and Kubiatowicz., "Tapestry: A Resilient Global-Scale Overlay for Service Deployment," J. Selected Areas in Comm., 2004.