P2P File Sharing Application

Acc. No. (FMS) T-1135

T01135

Developed by
Muhammad Rashid

Supervised by
Prof. Dr. Khalid Rashid

Department of Computer Science
International Islamic University, Islamabad
(2004)

Acc. No. (DMS) T-1135

بسم الله الرحمن الرحيم

In the name of ALMIGHTY ALLAH, The most Beneficent, the most Merciful.

Department of Computer Science,
International Islamic University, Islamabad.

16 Aug, 2004

Final Approval

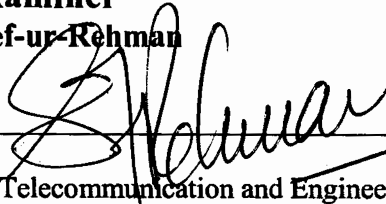
It is certified that we have read the thesis, titled **"P2P File Sharing Application"** submitted by **Muhammad Rashid** under University Reg. No. **11-CS/MS/01**. It is our judgment that this thesis is of sufficient standard to warrant its acceptance by the International Islamic University, Islamabad, for the Degree of **Master of Science**.

Committee

External Examiner
Dr. Abdus Sattar

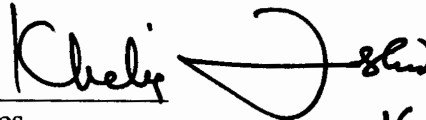
Consultant Multimedia Center,
Department of Computer Science,
Allam Iqbal Open University, Islamabad

Internal Examiner
Dr. S. Tauseef-ur-Rehman



Incharge,
Department of Telecommunication and Engineering.
International Islamic University, Islamabad.

Supervisor
Prof. Dr. Khalid Rashid



Dean, Faculty of Management Sciences,
Dean, Faculty of Applied Sciences,
International Islamic University, Islamabad.

Dedications

To my father, for giving me the courage to undertake this task, my mother, for giving me the patience when things got tough, my sister, for her wisdom which proved invaluable through out the project, and to my brother for making this whole experience a lot of fun.

A dissertation submitted to the Department of Computer Science,
International Islamic University, Islamabad in partial
fulfillment of the requirements for the degree of
Masters of Science in Computer Science.

Declaration

I hereby declare that this software, neither as a whole nor as a part thereof has been copied out from any source. It is further declared that I have developed this software entirely on the basis of my personal efforts made under the sincere guidance of my teachers and supervisor. No portion of the work presented in this thesis has been submitted in support of any application for any other degree or qualification of this or any other university or institute of learning.

Muhammad Rashid
11-CS/MS-01

Acknowledgement

All praise is to Allah Almighty, the most merciful, the most gracious, without whose help and blessings, I was unable to complete the project.

Thanks to Prof. Dr. Khalid Rashid, my project supervisor, whose sincere efforts helped me to complete my project successfully. Without his great help, motivation and righteous guidance it would have been impossible to complete this project.

And a special thanks to my family who helped me during my most difficult times and it is due to their unexplainable care and love that I am at this position today.

Muhammad Rashid

Project in Brief

Project Title:	P2P File Sharing Application
Undertaken by:	Muhammad Rashid
Supervised by:	Prof. Dr. Khalid Rashid Dean, Faculty of Management Sciences Dean, Faculty of Applied Sciences International Islamic University, Islamabad
Started in:	September 2002
Completed in:	August 2003
Software used:	Java™ 2 SDK, Standard Edition, Version 1.3.0 Allaire Kawa Java IDE, Professional Edition, Version 5.0 Microsoft® Office XP
Operating System used:	Microsoft® Windows XP Professional
System used:	Intel® Celeron™ CPU 1 GHz

Abstract

This thesis focuses on peer-to-peer (P2P) technology, which is recently so much spoken about in both the business and the academic world. During the thesis an application layer protocol capable of sharing resource among peer systems was developed. This protocol is specifically capable of sharing files among peers but can also be extended so that other resources can be shared (e.g. processor). An application was also developed which by using the above mentioned protocol is capable of sharing files among peers. This report gives an overview of peer-to-peer technology, defining it and describing the architecture. Also the most successful file-sharing implementations from the P2P world are presented in detail; each is followed by an analysis, pointing out the advantages and disadvantages. This document also contains the research, design, and implementation details of the above mentioned peer to peer protocol and application for resource sharing.

Table of Contents

Ch. No. | Contents | Page No.

1. Introduction 1
 1.1 Objectives 1
 1.2 Peer to Peer Networks 1
 1.3 Resource Sharing 4
  1.3.1 File Sharing 4
  1.3.2 Grid Computing 5
2. Existing Implementations 6
 2.1 Napster 6
  2.1.1 Napster Protocol Details 8
   2.1.1.1 Client-Server protocol 8
   2.1.1.2 Client-Client Protocol 8
  2.1.2 Analysis 10
 2.2 Gnutella 12
  2.2.1 Gnutella Protocol Details 13
  2.2.2 Analysis 16
3. Requirement Analysis 20
 3.1 Protocol Requirements 20
  3.1.1 Platform independence 20
  3.1.2 Extendable 20
  3.1.3 Scalable 21
 3.2 Application Requirements 21
  3.2.1 Implement the Protocol 21
  3.2.2 Implement the file sharing message 22
  3.2.3 User defined file management 22
  3.2.4 Peer management 22
  3.2.5 Network Mapping 23
  3.2.6 Usability 23
4. System Design 24
 4.1 Protocol Design 24
 4.2 Application Design 28
5. Implementation 32
 5.1 General Implementation notes 32
  5.1.1 Language Choice 32
  5.1.2 Network Choice 32
 5.2 Application Details 32
  5.2.1 Miscellaneous classes 33
   5.2.1.1 The Message Class 33
   5.2.1.2 MessageIDWrapper class 34
   5.2.1.3 MessagePeerIdWrapper class 35
   5.2.1.4 Result class 35
   5.2.1.5 TransferInfo class 36
   5.2.1.6 Util class 37
  5.2.2 The GUI Element 39
   5.2.2.1 Gui class 39
   5.2.2.2 SearchPanel class 43
   5.2.2.3 SearchPanelTableModel class 44
   5.2.2.4 TransfersPanel 44

5.2.2.5 TransfersPanelTableModel 45
5.2.2.5 SetupPanel 45
5.2.2.6 FileLister 47
5.2.2.7 FileRemover 48
5.2.2.8 PeerPanel 49
5.2.2.9 NetworkMapPanel 50
5.2.2.10 NetworkMap 51
5.2.2.11 Node 51
5.2.2.12 Edge 52
5.2.3 The Download Section 52
5.2.3.1 DownloadManager 53
5.2.3.2 Download 54
5.2.3.3 Upload 55
5.2.4 The Network Section 55
5.2.4.1 Control class 56
5.2.4.2 PeerListener class 58
5.2.4.3 Peer class 58
5.2.4.4 MessageHandler class 60
5.2.4.5 PeerMessageHandler class, QueryHandler class, and QueryHitHandler class 61
6. The System in Operation 62
6.1 Search Tab 62
6.2 Transfers Tab 63
6.3 Setup tab 63
6.4 Peer List Tab. 65
6.4 Network Map Tab. 66
7. Results and Conclusion 67
7.1 Study at least one existing peer to peer file sharing application 67
7.2 Specify a peer to peer protocol for resource sharing 67
7.3 Create an application to share resources (files) between machines 67
7.4 Advantages of Shadow Protocol 67
7.5 Scope for future work 68
References and Bibliography 69
Appendix A A-1
Appendix B B-1

Chapter 1
Introduction

1. Introduction

This document contains the research, design, and implementation details of a peer to peer application for resource sharing. In this section the problem outline and the objectives of the project are presented..

In many professional environments there is an abundance of personal computers (PCs). These PCs are high powered machines which often do not reach anywhere near their maximum throughput, with processors often running idle cycles. There are also many situations where large amounts of processing power is needed, but the price of a 'super computer' matching the requirements needed make such a machine unfeasible.

This project looks into specifying a protocol, and then building an application that will allow machines on a network to set up connections between themselves and allow for resources to be shared between neighbors. We will also need to look at the way in which neighboring machines are chosen, so that optimum resource usage can be obtained.

1.1 Objectives

Following objectives must be fulfilled in order to claim that the project is a success.

i) Study at least one existing peer to peer file sharing application

Two most popular applications namely Napster and Gnutella have been studied in detail (Section 2) before developing a peer to peer file sharing application.

ii) Specify a peer to peer protocol that can be used to coordinate resource sharing

The protocol we need to design must be able to communicate between peers, querying each other about what resources they can share. Although we cannot say for sure what resources people will want to share, but we will make the protocol extendable, efficient and scalable.

iii) Create an application to share resources between machines.

In order to test our protocol we need to create an application that uses it to share resources. We will use this application to share computer files.

1.2 Peer to Peer Networks

This section of the report deals with the background research into the design of peer-to-peer networks and applications. It will look at the underlying protocols of such applications and identify key features.

Peer to peer technology has been used for almost as long as there have been computer networks. The internet itself can be seen as a large peer to peer network, with data passing from router to router. In general, a peer to peer system can fall into one of

the two categories. The first is a full distributed (pure) peer to peer system. The second is a central server peer to peer system.

A pure peer to peer system can be described as "any application or process that uses a distributed architecture and direct bi-directional communication between autonomous resources without central coordination and management" [1]. The nodes in the network are equal in functionality, i.e. each node is both a client and a server (a servent). When a node joins the network, it connects to one or more other nodes. These nodes are its neighbouring peers. Communication between nodes is done through message passing between neighbours. For example in Figure 1.1, to communicate with node I, node A must pass a message to its neighbours (B and/or C), who will then pass the message to their neighbours; and so on until node I receives the message. The method used to forward data between nodes is dependent on the system being used. The Gnutella network floods the data to all of its neighbours, excluding the one that sent it. IP routers use routing tables to decide which peer is the most appropriate recipient for the message, only flooding if a decision cannot be made.

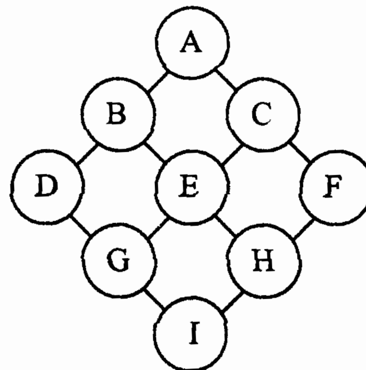


Figure 1.1 Example Peer to Peer Network

Pure peer to peer networks have the advantage that there is no central point of failure. This means that the network should be able to loose some nodes/ connections without collapsing the entire system. Take the of example network in Figure 1.1. Suppose nodes D and F become disconnected, and node E looses its connection to node C. This gives us a network similar to Figure 1.2. Despite the loss of these two nodes, and node E's connection to node C, it is still possible for all the remaining nodes to communicate with each other.

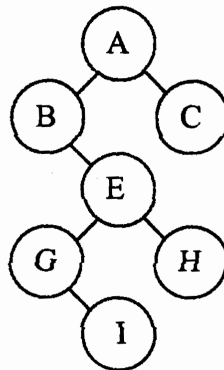


Fig. 1.2 The p2p network after loosing nodes/ connections.

Another advantage of pure peer to peer networks is the scalability of these systems. Each node sees only its neighbours, and so only needs to scale in proportion to them. For example, suppose nodes can have at most two neighbours. The nodes could then form a long chain of connections with less cost to any individual node. Pure peer to peer networks do have disadvantages though. The main disadvantage is the same as the main advantage, no central control. This means that any work being done between nodes must be coordinated by the nodes themselves. This leads to more complex applications. Also, without central control it is difficult to control data distribution. If we take file sharing applications, it is very hard to stop the sharing of copyright material. This is because several nodes can download the same file at once and the file can spread across the network very quickly. Whereas it is relatively easy and cost efficient to take one user to court and have them remove their copy of the file, it become more expensive when you are taking many hundred users to court.

The ease and speed with which messages pass between nodes also causes a security risk to users. Suppose node E in Figure 1.1 is infected with a virus that automatically copies itself to neighbouring peers. It would very quickly infect the entire network. Viruses like this do exist. The VBS.Gnutella virus was discovered in May 2000[2]. This virus was fairly harmless, creating and sharing both a copy of itself and a file containing the victim's name. The method of transmission was also fairly basic, with the virus creating a shared file with an appealing name (such as Gladiator.vbs, or Battlefield Earth.vbs). It was hoped that users would think that they were downloading the named file but actually would be downloading the virus. Since then several advanced viruses have been created. W32.Gnuman.Worm was discovered in February 2001 [3]. Unlike VBS.Gnutella this virus changes its filename to match any query that has been made, thus increasing the potential number of contractors. This virus also provides a greater security risk because it opens a port on the infected machine.

On the whole, peer to peer networks are being seen as 'another vector of delivery for malicious code' [4]. Because many peer to peer applications involve transferring of files from a possibly unknown source, users can never be sure whether the file they are downloading is actually the file they want. Even if the file they downloaded is the file they want, it still may be infected with a virus. It is also feared that a virus could connect a user to a peer to peer network without his/his/her knowledge [4]. The virus could be spread by any method, and provided it doesn't set off a virus protector it could start up a servent to a peer to peer network without the user realising. Many firewall setups wouldn't detect this attack because they tend only to block incoming connections. In contrast to pure peer to peer systems, other type of peer to peer technology employs the use of a central server. Consider the network in Figure 1.3:

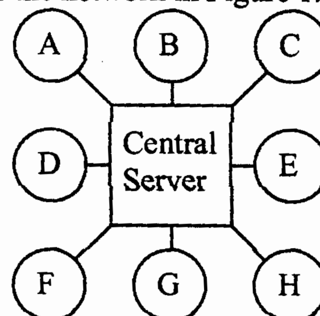


Fig. 1.3 An Example of Peer to Peer Network

Rather than passing messages between peers, each message is sent to the central server. The central server then decides who should receive the message, and then forwards it on to them. At first glance this looks like the classic client server network model, and in many ways it is similar. The difference between the two is that the messages sent through the central server are often only very simple, with any other data transfers being done directly between peers. For example, the Napster network when it still operated used to use the central server to hold a database of which music files were being shared by which peers [5]. Once a peer had the list of files another peer was sharing, all other communications (such as transfer requests) were sent directly between the two peers.

The advantages and disadvantages of central server peer to peer networking are opposite to the pros and cons of pure peer to peer networks. The biggest pro of the central server is its central control. As mentioned above, it is easier to co-ordinate tasks between nodes if there is a central server. Also, it is easier for a company to monitor the use of system thus creating a more secure environment, and more importantly for a company, it provides a base for billing peers for use of the network.

Although many of the security floors that affect pure peer to peer networks would also affect central server networks, several can be eliminated. In case of the W32.Gnuman.Worm, the file transmitted may have any name, but it is always a Windows executable (exe) file of size 8192 Bytes [3]. With this knowledge, the central server could be designed to identify requests for these files and block them.

The main disadvantage of central server peer to peer networks is the scalability issues. Suppose the central server in Figure 1.3 can only support the 8 nodes shown. If a ninth node was to be added either another central server would have to be added, or the server would have to be upgraded. Because the server would only be upgraded a certain number of times, eventually another server would need to be used. This then poses problems of synchronising the servers, so that a node connected to server A could connect to a node connected to server B.

In conclusion, the style of peer to peer networking used is dependent on the application being developed and the requirements of the company producing the product. If the application is to be deployed on a large scale, it is cost effective to use a pure peer to peer solution. If the application requires any element of central control, and the scalability issues are not seen as a problem, then the central server approach is more effective.

1.3 Resource Sharing

Many of the peer to peer technologies available today are used to share resources between peers. The following two subsections discuss two of the most commonly shared resources, files and processor cycles.

1.3.1 File Sharing

File sharing applications are the most well known of all of the peer to peer technologies. It is common to see peer to peer technologies described as 'Napster/

Gnutella like' due to the popularity of these file sharing applications. The popularity and the publicity of these services stem from illegal sharing of copyright material, such as music and movies. Users are attracted to the file sharing applications because of their ability to download entire movies and CDs at no cost. This theft of copyright material has lead to many high profile court cases, which in turn advertises the technology to more users. It is important to note that there are many people who only download legal files, and there are also some applications that bill the users for copyright material.

The next chapter will discuss two of the most famous file sharing applications, Napster (a central server peer to peer system), and Gnutella (a pure peer to peer system).

1.3.2 *Grid Computing*

Grid computing is the name given to applications that use several computers to complete a task. These applications are becoming more and more popular due to the increasing power/ price ratio of desktop PC's and the high performance of the networks that connect them. We will be taking grid computing as our example for non- file resource sharing.

Grid computing solutions are best suited to applications where large volumes of raw data need to be processed. It works effectively only when this large data can be segmented into smaller independent data units (often called work units). One such application is seti@home [15]. Seti@home uses data recorded by the Arecibo telescope in Puerto Rico to search for extra-terrestrial intelligence. The problem with this is that the data produced per day is too large to be processed by a single machine in any reasonable amount of time. The solution to this is to break down the data into work units, and allow people to download these and process them at home. The seti@home application runs as an idle process on many machines around the world, automatically downloading the work units as needed. This is an example of processor sharing.

There are several problems with Grid computing solutions. Firstly, the end users must be trusted enough to perform the correct processing on the data received without falsifying results. Another major problem is that of lost work units. If the application sends out each unit one at a time, the system would experience problems if any of this data was corrupted on the network, or simply never returned. To work around this problem most grid computing systems store all of the data sent out until a response is returned. If after a certain time out period no response has been heard the work unit is sent out to another end user. In all, grid computing solutions are being used in environments where there is often a lot of wasted computer power (e.g. large offices, universities), or via voluntary systems such as seti@home. The types of applications are often medical or scientific where large amounts of data are to be processed.

Chapter 2
Existing Implementations

2. Existing Implementations

The implementations presented in this chapter are the best known from the file sharing type of P2P applications. The documentation and technical specifications of the available systems help to give a detailed description and analysis of their architecture and protocols. In order to analyze the systems from the user point of view, all the described applications have been installed and used for at least half a year.

2.1 Napster

Napster is a program for MP3 (MPEG-1 Layer 3) files exchange [5]. This is a peer-to-peer application that made many people look differently at Internet. Before looking into Napster's details, let us have a look at the history of Napster.

The recording industry that is closely related to the Napster boom was always in control of its business, even when the well known high-quality digital sound format MPEG-1 appeared. The new format gave the possibility to store songs of a near CD quality in relatively small files, and to distribute these files. There began to appear more and more websites having big lists of MP3 files being offered for download. But it was possible for the recording industry to shut them down, or at least to ask the people behind these services to shorten their lists. Some of them were even brought to court. But the possibility to have any song we like at hand, to make our own collection, to exchange songs with our friends, was very attractive and there was a way to get around recording industry control.

Shawn Fanning, a student at Northeastern University (Boston, MA),in January 1999 at the age of 19 decided to create an application that gave the possibility to campus students to share MP3 files. By the end of the same year this application was launched on the campus of Indiana University. It spread very quickly despite the problems and fights with university administration, and the Napster Community grew by leaps and bounds. Napster service was introduced to the public through www.download.com and shortly became the most downloaded software on the site. Napster has been named the fastest growing application in the history of the Internet by Media Metrix, an Internet research company. Nearly 5 million people used Napster in July 2000 compared to 1.1 million in February, the first month the Napster use was measured [19].

It turned out to be not so easy to shut the Napster service down, as it was in the case of web servers with copyright content. Already in December 1999 RIAA (Recording Industry Association of America) sued Napster for copyright infringement and related state law violations. Then in April 2000 the heavy metal rock group, Metallica, sued Napster for copyright infringement and racketeering. Rapper Dr. Dre followed suit two weeks later. In October 2000 Napster announced it was partnering with the German media giant, Bertelsmann AG, to develop a new membership-based distribution system that would guarantee payments to artists. Under the deal, Bertelsmann would drop its lawsuit against Napster, make its music catalog available and gain the right to buy a stake in Napster.

To explain why it was not that easy to judge the Napster creators, and finally shut down the service by March 2001, let us take a look at how Napster works, at a very superficial level.

First of all there is a client-part piece of software we have to install in order to use Napster. Most of the peer-to-peer implementations have their own client side application. Then there is a set of computers that are intended for Napster network organization. They are keeping track of the registered users, the content the users share, type of connection, user presence online and some other things. We cannot say they are servers in the context we define a "server" in the client server model. They do not store the contents that users are searching for, they just store the index of filenames and where the files are located. These computers are located at Napster's Headquarters and use distributed techniques for index and user management. As we shall see, they are acting more as routers in the peer-to-peer environment, than as servers. We will still use the "server" term when talking about these special-purpose computers.

After the user installs and launches the client-side software, it automatically connects to one such server. First time, the user has to register by entering a username and password. Usernames are unique in the Napster network, and the reason is explained below. After registration the user specifies the directory of files open for sharing. The username, the password, details about the connection, user's current IP and port number, as well as the information about the shared files is sent to the server where it is stored for future reference.

When a registered user launches the client, it connects to one of the servers (actually we can see them all as one, as this is transparent for the user) and it sends the information that might have changed from last login, such as IP address and port number, the list of shared files. The server also internally updates the user's status as "online". At this moment the user becomes an active part of the Napster community, and can benefit from services, as well as provide the same services to others. For example, the user wants to find a certain song. He/she enters the search parameters, presses the search button and the query is sent to the server for execution. The search results are sent back to the user, together with the IP address, port number and other connection details for each successful search hit. These search results are presented in a user-friendly way, so that the user can sort the results by different parameters. When the user finds the desired song and clicks on the link, a direct connection is made between this user's computer (peer) and the computer (peer) that is holding the song (file) itself. A peer-to-peer connection is established and the download process begins.

For the second peer the described process looks a little bit different. Let us say the user does not do anything, but the Napster client software is running. Besides the above described functionality, that is clearly client-oriented, the same application has a kind of server-oriented functionality. There is a "process" that is listening on a specific port for incoming connections from other peers. Therefore, after receiving a request and after the connection is established, for this user the described process will look like an upload, or more simple, as a provider of service by giving the shared file. The user can specify how many simultaneous downloads and uploads he/she wants to have, in such a way specifying both his/her interest in the services, and his/her contribution to the services.

After each successful download the client reports to the server some information, both for logging and for catalog (index) updating.

2.1.1 Napster Protocol Details

In order to make an analysis of Napster protocol and architecture, we will go into some technical details about structure of the messages and some significant examples of messages. They exemplify the basics of the protocol and exhibit advantages and disadvantages. The complete protocol specifications can be found in [10] and [18].

2.1.1.1 *Client-Server protocol*

Napster uses TCP for client to server communication. Typically the servers run on ports 8888 and 7777. This is different from the `metaserver' (or redirector) which runs on port 8875.

Each message to/from the server is in the form of:

<length><type><data>

Where <length> and <type> are 2 bytes each. <length> specifies the length in bytes of the <data> portion of the message. Detail of the messages can be found in Appendix A.

2.1.1.2 *Client-Client Protocol*

File transfer occur directly between clients without passing through the server. There are four transfer modes, upload, download, firewalled upload, firewalled download. The normal method of transfer is that the client wishing to download a file makes a TCP connection to the client holding the file on their data port. However, in the case where the client sharing the file is behind a firewall, it is necessary for them to "push" the data by making a TCP connection to the downloader's data port.

Normal Downloading

Regardless of which mode, the downloading client will first issue either a search (200) or browse (211) command to the server. This returns a list of files and information on the client sharin the file. To request a download, a get (203) request is sent to the server. *The server will respond with a get ack (204) containing more detailed* information.

This is the point at which the different methods diverge. If the 204 get ack says that the remote client's data port is 0, you must send a 500 request to the server requesting that the remote client send the data to you. In this case you wait for the remote client to connect to your own data port.

In the case where the sharing client is not firewalled, you make a TCP connection to the data port specified in the 204 message from the server. The remote client should accept the connection and immediately send one ASCII char, `1' (ASCII 49). Once you read this char, you send a request for the file you wish to download. First send the string "GET" in a single packet, and then send

<mynick> "<filename>" <offset>

Where <mynick> is your napster user name, <filename> is the file you wish to download, and <offset> if the byte offst in the file to begin the transfer at (if you are downloading for the first time, and not resuming a prior transfer, you should uses 0 to start at the beginning of the file).

The remote client will then return the file size, or an error message such as "INVALID REQUEST" or "FILE NOT SHARED". Note that the file size is not terminated in any special way, and the best way to figure out the size is to keep reading until you hit a character that is not a digit (it will usually be 0xff which is the start of the MP3 frame sync header, but if a ID3v2 tag is present it might look different). Immediately following the file size is where the data stream for the file begins.

Once the data transfer is initiated, the downloader should notify the server that they are downloading a file by sending the 218 message. Once the transfer is complete, you send a 219 message to indicate you have finished the download. Note that this is cumulative, so that if you are downloading multiple files, you send one 218/219 pair for EACH concurrent download—this is how the server knows how many transfers you have going on. Likewise, the up loader should send one 220 message for each upload, and one 221 when each upload has finished.

Firewalled Downloading

As described above, when the file needs to be pushed from a client behind a firewall, the downloader sends a 500 message to the server. This causes a 501 message to be sent to the up loader, which is similar to the 204 message for a normal download.

Once the uploader receives the 501 message from the server, they should make a TCP connection to the downloader's data port (given in the 501 message). Upon connection, the downloader's client will sent one byte, the ASCII character `1'. The uploader should then send the string "SEND" in a single packet, and then the information:

<mynick> "<filename>" <size>

where <mynick> is the uploader's napster user name, <filename> is the file being sent, and <size> is the size of the file in bytes.

Upon receipt, the downloading client will either send the byte offset at whcih the transfer should start, or an error message such as "INVALID REQUEST". The byte offset should be sent as a single packet in plain ASCII digits. Just as with above, a 0 byte offset indicates the transfer should begin at the start of the file.

Each client should notify the server that they are uploading or downloading with the 218/219 (downloading) or 220/221 (uploading) command pairs.

Client to Client Browsing

Napster 2.0 BETA 8 adds a feature which allows direct client-to-client browsing of file lists. To request a browse, a client uses the

640 <nick>

command. The server then sends a

640 <requester>

to the client which is getting browsed with the nick of the client that is requesting the browse. If the client accepts the browse request, it sends back a

641 <requestor>

to the server with the nick of the client requesting the browse. The server then sends a

641 <nick> <ip> <port>

to the requesting client. In the case of an error, the server will send a 642 command in response to the 640 command.

The browsing client then makes a TCP conection to the remote client's data port. After getting the "1" character, the browsing client sends a

GETLIST

At which point the remote client sends its nick followed by a linefeed (\n) by itself in a single packet (ie, one write() call)

<nick> LF

followed by the list of files being shared (the format being the same as the data of the "share" command). Each line is terminated by a linefeed char. At the end of the list, an additional linefeed char is sent and then the client closes the connection.

In the case that the remote client is firewalled, the browse list will have to be pushed to the requesting client. The remote client makes a TCP connection to the requesting client, then sends

SENDLIST <nick>\n

followed by the list of files, as with the "GETLIST" command response.

2.1.2 Analysis

Although Napster became so popular and spread all over the world in a very short time because of the services it offered, and not because of its architecture and protocol specification, it still made a lot of IT professionals think about the technology behind it. One reason is Napster's ability to cope with scalability in an extreme rate of growth, the huge number of users that joined the community, without shuting down the Napster network, without overloading the Napster servers (as might happen if a client-server model was used), without causing big problems to the Internet traffic in general. In other words Napster and the technology behind it showed a previously unseen degree of scalability, and this was the start for different companies and enthusiasts to begin developing their own implementations of peer-to-peer. Also in the academic world it has attracted attention, leading to the introduction of more research papers on this technology.

With respect to the peer-to-peer concept, Napster is a hybrid model of P2P. It has a central computer (or a distributed network of such computers) that performs some special tasks for the network to function. Napster's scalability is due to the fact that even if there is a server, to which the clients connect, and which is a central point for the whole network, its functions are quite different from those a server in a client-server model has. Its role in the network is more organizational than for providing of services.

In the Napster case the main purpose of this server is to keep the catalog of shared files in the network, keep the information where these files are located, perform search through this catalog on behalf of the user, and report all the results back together with all the information needed to establish a direct peer-to-peer connection between the requesting peer and the one that holds the requested data. It acts as a router to the

requester, as the client is given the contact address where he/she can find what he/she is looking for. It still seems that there is a lot of work that the server performs, especially if looking into protocol details, but this still takes much less time than performing uploads of large media files to the client in client/server architectures.

From another point of view the existence of this server makes the network dependent on it, as the crash of the server would mean the crash of the whole network. Besides, as the server stores the content index of all the registered users, we cannot speak about anonymity and privacy in such a network.

A great concern in a peer-to-peer network are the modem users, with unreliable and slow connections.

First of all, all the computers that connect to Internet using a modem have dynamic IP addresses (as usual), and second, most of them are behind a firewall. The problem of the permanently changing IP address is solved by Napster in a well known way – the users are registered using a unique nickname, thus bypassing the DNS system and permanent IP addresses. The same method is used by ICQ for example, with the difference that User Identification Numbers (*UIDs*) are used as identifiers instead of nicknames. In Napster the nickname is what identifies a user in the network, and not its IP address or Domain Name. Every time a user connects to the network, the client sends to the server its current address, port number, and these are used later for direct connection between clients. The firewall problem is solved by Napster based on the possibility of the client behind the firewall to make outgoing TCP connections, being unable to receive messages. This is known as "pushing" data (see Napster protocol details, messages 500/501). Actually, the same method is used by other implementations to solve the firewall problem.

Despite the fact that Napster became so popular, and it was used by a lot of users before it was shut down, it has some drawbacks that we want to point out. One of the main drawbacks we have already mentioned – that is the presence of a central server. This makes the network easier to shutdown and more vulnerable to external attacks. It is enough to shutdown the server, or to find a security hole in the server, and the whole network will be affected. Another limitation of Napster is the exchange of just MP3 files. Some can regard this as a good thing, because there is more order in the network, and there is less threat of viruses, which have not yet found their way into MP3 format. So, it is not a general purpose peer-to-peer network, it is rather specialized. But even for MP3 files exchange, the search engine is quite weak. The search is performed only as substrings of filenames, not distinguishing between the author and songname, despite the fact that there are two separate fields for them in the client-side search window. Therefore, if the search is quite general, or the search string or keyword is often encountered in song names, then there are plenty of results, from which only the first 100 are displayed, ending in the possibility that you will not find the song you seek even if it is available in the network.

Another problem that is not addressed in Napster is corrupted or broken files that can very quickly spread among a lot of nodes. This is because in the case of a transfer interrupt, the file is written in the shared directory and the index on the server is updated, whether the file was completely downloaded or not.

It is worth mentioning that connection speed information does not always correspond to the real speed. Some users claim to have a slow connection while they have a fast one, so that it is not so attractive for other users to connect to them. That is why, from the practice of using Napster, we can say that the ping time value is more informative. Napster has a lot of successors that try to improve on the Napster model by making changes and additions to the Napster protocol.

2.2 Gnutella

Gnutella is another representative of peer-to-peer networking. It was originally developed by programmers inside America Online's Nullsoft (developers of Winamp, the popular MP3 player), but without the knowledge of AOL's top executives. When Gnutella was ready to deploy, the developers put it for public download on the Nullsoft website. Gnutella was there for a few hours only before it was taken down, but during that time several thousands downloads occurred. Using these downloads programmers reverse-engineered the software and created their own Gnutella software packages.

Gnutella works much like Napster, but at the same time it is quite different. The main difference is that in Gnutella's variant of peer-to-peer there is no central server at all. We can speak about Gnutella as a pure peer-to-peer network with a completely decentralized protocol. The only component parts of the network are the peers, and the logical network is formed dynamically every time you log into the network. Your computer connects directly to a couple of other computers, each of which are connected to some other computers, creating a tree chain of members that quickly becomes enormous. You can even specify how big you would like the network to be, by setting some protocol specific information that will be used by the piece of software that you have to install in your computer in order to be able to become a part of the network. The information you provide is used to build the virtual network that you think better suits you taking into account your own network parameters and the character of the information you share.

Once attached to the network, peers interact with each other by means of messages. Peers will create and initiate a broadcast of messages as well as rebroadcasting others (receiving and transmitting to neighbors). The messages allowed in the network are [25]:

PING MESSAGE – message directed at a host to check its presence in the network, as well as to announce peer's own presence (the presence of peer that issues the message).

PONG MESSAGE – a reply to a ping. The pong message contains information about the peer such as their IP address and port as well as the number of files shared and the total size of those files. Peers forward this kind of message to their neighbors so that it is possible to later find other peers. This is needed in case there is a disconnect in the network.

QUERY MESSAGE – these are messages sent when searching for certain information, and can get forwarded throughout the entire network. Query messages are uniquely identified, but their source is unknown.

QUERY RESPONSE MESSAGE – these are replies to query messages, and they include the information necessary to download the file (IP, port, and other location information). Responses also contain a unique client ID associated with the replying peer. These messages are propagated backwards along the path that the query message originally took. Since these messages are not broadcast it becomes impossible to trace all query responses in the system.

GET/PUSH MESSAGE – get messages are simply a request for a file returned by a query. The requesting peer connects to the serving peer directly and requests the file. Certain hosts, usually located behind a firewall, are unable to directly respond to requests for files. For this reason the Gnutella protocol includes push messages. Push messages request the serving client to initiate the connection to the requesting peer and upload the file. However, if both peers are located behind a firewall a connection between the two will be impossible.

2.2.1 Gnutella Protocol Details

Next we consider how Gnutella protocol works, and then we discuss the most important issues of this kind of implementation. This more or less detailed and useful information is provided by Jerome Kuptz [22].

1. The Gnutella application on your desktop is actually a peer, acting as both client and server in interactions with a network of similar peers. Unlike Napster, Gnutella has no central servers to which it can connect for information. Before it can begin swapping files, your peer must be told (by the user or from its own database) the IP address of one other peer to which it can connect.

2. Your Gnutella peer transmits a handshake message ("GNUTELLA CONNECT/0.4\n\n") to the other peer. The handshake identifies you to the other peer, which in return sends back a confirmation ("GNUTELLA OK\n\n").

3. Your peer sends a ping request to the other peer, announcing its presence on the network. The ping request includes a time-to-live (TTL) count, which states how many times the request can be forwarded to other computers. The default for most Gnutella peers is 7.

>>> CONTENTS OF HEADER

0-15 GUID (Globally Unique Identifier)
16 Messaging Identifier
  0x00 Ping
  0x01 Pong
  0x41 Push
  0x80 Query
  0x81 Query results
17 TTL
18 Hops
19-22 Payload length

>>> CONTENTS OF PING

No payload. Just the header is sent as the "I'm here" message.

4. The other peer replies to your ping with a pong, which contains its IP address and file sharing information (total files and kilobytes shared).

>>> CONTENTS OF PONG

0-1 Port
2-5 IP address
6-9 Number of shared files on the host
10-13 Total kilobytes of shared files on the host

5. The other peer also forwards your ping to additional Gnutella peers it knows about, after first reducing the TTL count by 1, from 7 to 6. Each peer that receives the packet similarly subtracts 1 from the TTL and forwards the packet to others. Many peers end up forwarding your ping to one another over and over. Gnutella relies on fat bandwidth to overcome this inefficiency. Users raising their TTLs past 7 could flood the Net with trillions of pings. To keep Gnutella efficient, other peers will adjust high TTLs before forwarding them.

>>> RULES FOR FORWARDS

Hops > 7 – drop the message, because it has exceeded the maximum TTL of 7.

TTL > 50 – drop the message.

7 < TTL < 50 – possibly overzealous users. Adjust TTL to 7.

TTL + hops > 7 – adjust to TTL = 7–hops (maximum 7).

6. Each peer that receives your ping sends back a pong to your peer, routing the pong back along the path of the ping.

7. As pongs arrive, your hostcatcher collects the IP addresses of available peers. They may be anywhere on the Internet, but all are at most seven degrees of separation from you. This network of peers known to your own node is your "horizon".

8. A typical radius includes 2,000 to 10,000 other peers, with 500,000 to 1 million files. Gnutella's open architecture means you can also share files with users of compatible programs such as Gnutella or Gnucleus [29].

9. To find a file, you enter a search term into the Gnutella interface on your screen. Your peer then sends a query directly to every peer known to your peer.

>>> CONTENTS OF QUERY

0-1 minimum connection speed of responding servers
2+ NULL terminated string of search criteria

10. Each peer searches its local files for matches to your query. If it does not find any, it does not reply. This prevents your computer from being bombarded with "no results" messages.

11. If there are one or more matches, a query results message is routed to your peer, containing the IP address of the sender and the matching file name. Unlike Napster or a Web search engine, your peer does not know when the search process is complete: Peers that have not replied either have found no results, or are still working on a reply. Newer implementations allow the user to set the duration of the search.

>>> CONTENTS OF RESULTS

0	Number of hits
1-2	Port
3-6	IP address
7-10	Speed in kilobits/sec of responding host
11+	Results start (there are n of these)
	Index
	Size of file in bytes
8+	File name terminated with a double NULL
	Last 16 bytes GUID identifying the client to be used during a push request

12. When you select one of the query results for downloading, your peer creates a standard HTTP request (the kind used by browsers to request Web pages) from the IP address and filename in the results message. It sends this request directly to the peer, which returns the file via HTTP. This is part of what makes Gnutella networks hard to shut down, as their file transfers look just like ordinary Web traffic.

13. If the file you are after is hidden behind a firewall, your peer will issue a push request – a broadcast message that winds its way around the network until it gets to the recipient, which responds by connecting to your peer and transmitting the file. An estimated 50 percent of Gnutella traffic is across firewalls.

>>> CONTENTS OF PUSH

0-15	Client identifier from the results message
16-19	File index
20-23	IP address to push to
24-25	Port to push to

14. Peers on low-bandwidth networks will miss (or "drop") messages, causing pings, pongs, queries, and replies to be lost. This happens not only to messages to and from the low-bandwidth computer, but to any to which it is trying to forward packets. In other words, a huge portion of your radius can "go dark," becoming unreachable and unusable (see Figure 2.1, 2.2 and 2.3). This is another inefficiency inherent in Gnutella's serverless structure.

2.2.2 Analysis

We can now evaluate Gnutella's network organization, taking into account the way the protocol works and the experience of using Gnutella?

Because there is no central point that monitors the connections or the traffic on this network, and no company is responsible for the software that all the computers are using, it is quite difficult to keep track of what happens inside the network. Therefore, it is essentially anonymous. This decentralized architecture also means there is no company against which to file a copyright-infringement lawsuit. The anonymity is caused also by the file-transfer method. As mentioned above, all traffic looks like normal HTTP traffic, which makes it difficult or even impossible to track Gnutella traffic.

Gnutella offers the possibility to share any type of files. This could sound nice, but it also raises some security problems, as malicious users can use the network for spreading viruses. It is true that such systems as Gnutella, Scour, Freenet do allow viruses to be spread more easily. But that is the case with any system in which two people exchange files, whether it is people sharing a disk, as in Gnutella's case, or the Internet itself [26] [21]. A good example is electronic mail exchange. That is a perfect environment for a virus to spread, and some well known cases have proven that. Nevertheless people still use email. You just have to be conscious when clicking with your mouse on the attachment. With little care and knowledge about viruses, unwanted surprises can be avoided. The same is true for the file exchange networks. You have to take responsibility when downloading and launching files. And the extent that a virus can spread in a peer-to-peer network is much smaller than it is with e-mail, taking into account the number of users that use e-mail (all Internet users, actually), and the intensity of mail exchange against file transfers.

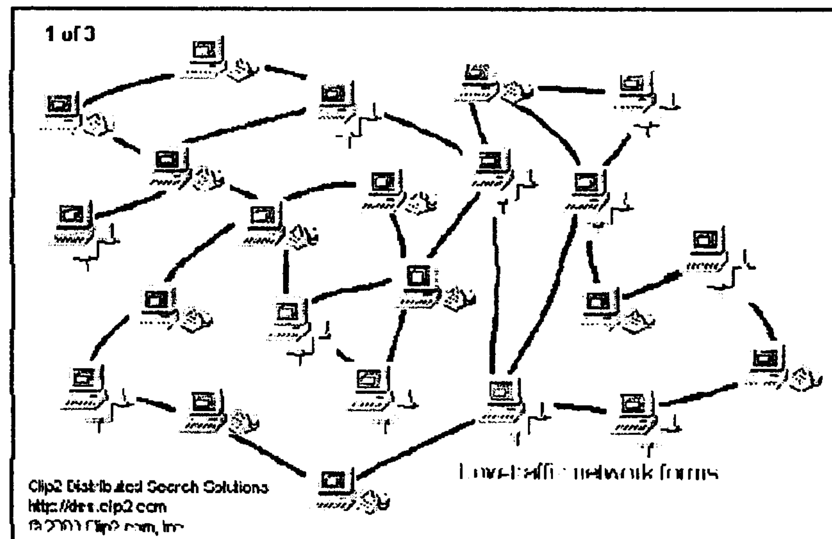


Fig.2.1 Low-traffic network forms

But there are some drawbacks of Gnutella architecture that merit more attention here. We mention two main problems that are related to each other – network overload, and the resulting bad scalability. Message traffic in Gnutella network is quite high, and it consumes a lot of bandwidth. Even if theoretically the number of participating peers can

be unlimited, the practice shows that such architecture has limited scalability. This is the reason messages have a TTL value, that specifies the "horizon" beyond which peers can not see, as well as the small cache of routed messages to prevent re-broadcasting.

Another big problem with Gnutella is the slow-speed peers, connected via a slow connection. Let us examine what happens in such a case. At the beginning we have a consistent network of peers, a part of the nodes being connected via modems, these connections being slow and unreliable (Figure 2.1).

With time, more and more unreliable nodes go offline, or are over flooded with messages. They become unresponsive (Figure 2.2).

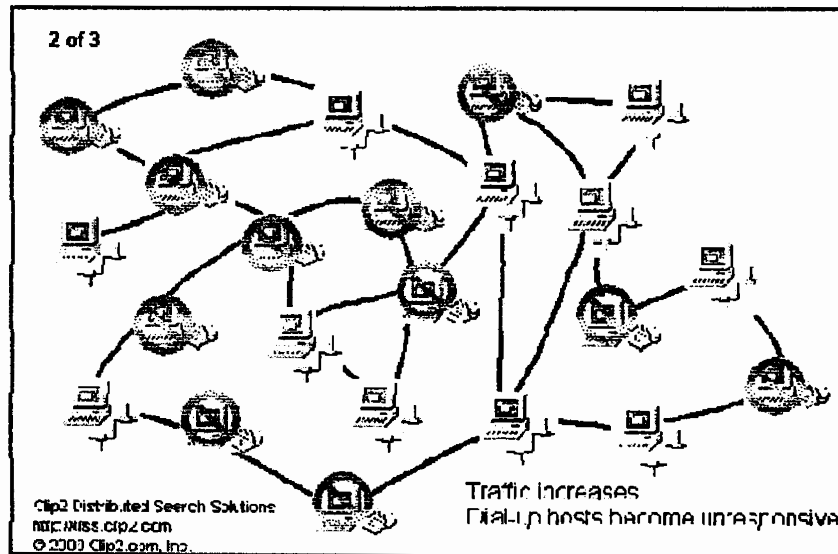


Fig. 2.2 Traffic increases dial-up hosts become unresponsive

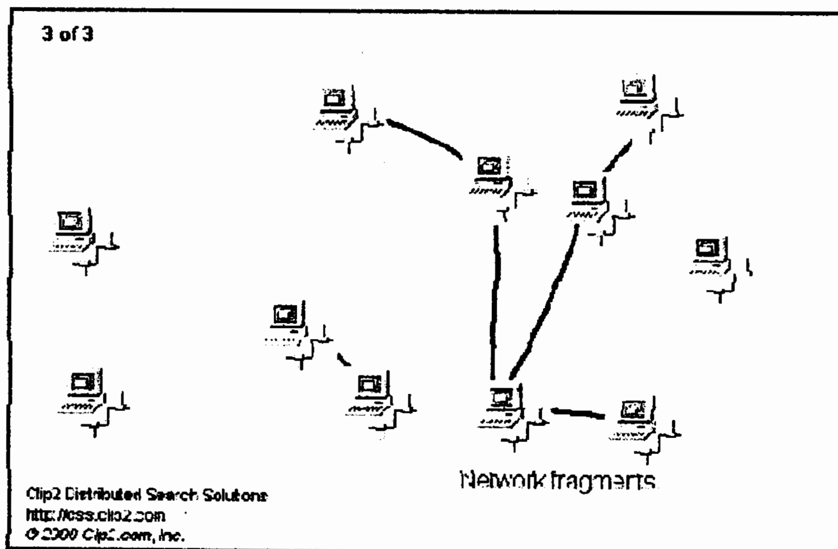


Fig. 2.3 Network fragments

This leads to a situation where instead of having a consistent network of peers we have just separated segments of peers, that can not communicate between them (Figure 2.3). From the practical using of Gnutella we can say that the network of nodes changes

constantly. You can have 10k hosts, and in a minute their number can drop to less than 1k. And this does not happen because suddenly 9k hosts went offline, but because your peer "neighbors", through which you accessed the network, went offline.

As specialists at dss.clip2.com, which made an analysis on Gnutella traffic, affirm about 67,000 bits/sec are necessary for a peer just for message communication in case the number of query messages in the network reaches the value of 10 per second [23]. And this means that peers connected via modems (even with 56 kbps) cannot deal with the high traffic, therefore delaying or just dropping messages. This leads to a situation where instead of having a consistent network of peers we have just separated segments of peers (Figure 2.1, 2.2 and 2.3) [24]. As the same source shows, this barrier of 10 queries per second was reached at the end of July, during the "Napster Flood", and then by middle of August, staying around this value until now [23].

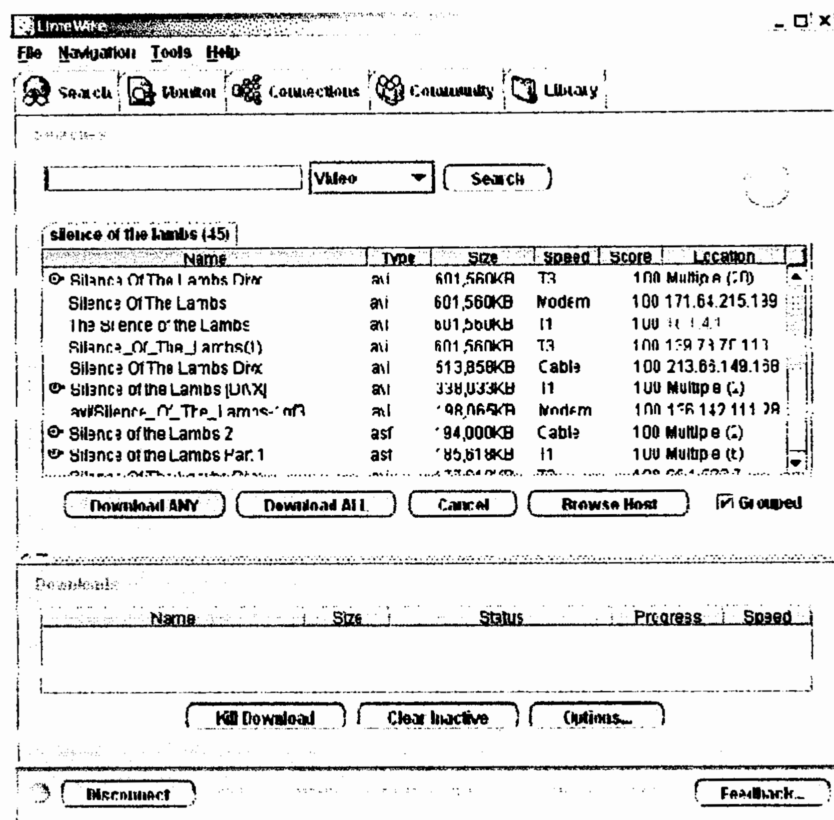


Fig. 2.4 LimeWire screen shot

In turn, these problems lead to issues in search problems. In the Napster case the search is done on one server, at a high speed, and for all currently connected peers. All the results are presented at once to the peer that issued the search request. In Gnutella the search request propagates from one peer to the other, and the results come one by one. An important detail is that those peers that do not have any files that match the search pattern do not reply at all. This is done to minimize traffic, but from another point of view, the requesting peer does not know the actual reason of silence – is it because no files were found, or because of network problems? If a slow peer, as described above, just drops such a search request message, the entire network tree beyond that peer is not even aware of the search request. This indicates that the search actually was done in a part (or several parts) of the network, and not on the entire network. Also related to this, a lot of files

have the same name and completely different sizes (music or video, for example). The screenshot in Figure 2.4 represents a search result for the movie “Silence of the lambs”, taken from the LimeWire Gnutella client [27]. You can see in the search results several files that have same name, but different sizes. That indicates incomplete files, or even if complete, of a worse quality. There is no description that would carry useful information about the file. So the user could end up spending several hours downloading something, and then realize that it is of no use, especially if it is a type of file that does not load at all if it is not complete (MPEG-4). It seems there are no means to continue the download, as the file is sent via HTTP that does not support a reget method.

The ad hoc character and decentralized architecture of Gnutella led also to a different kind of spam. First it occurs because "beta or badly written versions of the Gnutella software send poorly formed results onto the network, using precious bandwidth" [20]. But the main spam attacks are from those who spread different kind of advertisements, sending the desired information for any search request. Developers are looking now for methods to stop spam before it gets out of control. One idea, originally posted pseudonymously on the Slashdot.org community site, is of considerable interest among developers. Under that model, the Gnutella software would send a test search composed entirely of random characters. Any computer that responded to that meaningless string would then be filtered out of the next, real query [20]. First of all, it actually doubles the search requests traffic, and second, methods will be found to overcome this anyway. Another idea is to set up trusted third-party sites, which would review files on the Gnutella network and approve them as spam free. Software could be configured to accept only files that had been approved by one of these groups [20]. This will involve third-party sites, that will have to keep an eye permanently on the network content and files flow. In this way Gnutella becomes not so anonymous as it now is.

Despite all these drawbacks, the Gnutella architecture is quite powerful in the P2P world. It is much discussed and is looked upon as a basis for developing an architecture and protocol for the future.

Chapter 3
Requirement Analysis

3. Requirement Analysis

This section of the report deals with the requirements of the project. We will first detail the requirements of the protocol, and then we will detail the requirements of the application.

3.1 Protocol Requirements

This section will detail the requirements of the resource sharing protocol. These requirements are platform independence, extendibility, and scalability.

3.1.1 *Platform independence*

A key requirement for any communication protocol is to be platform independent. By this we mean that the protocol should be defined in a format that can be interpreted by any system using any programming language. The reason why this is an important requirement is because we are developing a protocol for use in resource sharing applications, not a protocol for use in *our* resource sharing application. This is a subtle difference, but an important one. It is envisioned that our implementation of the protocol will be the first of many implementations, and so we should define it to be independent of any technology we may be using.

An example of this is the use of bytes in the message headers. Suppose we were to have one byte field. We should define exactly how the eight bits in this byte are used. Some technologies will have bytes defined as unsigned numbers between 0-255 both inclusive, others have signed versions of bytes e.g. sign and magnitude, Diminished Radix Complement, Radix Complement, etc.

3.1.2 Extendable

An important requirement for the protocol is that it should be extendable. At this time we know of several resources that we wish to be able to share, such as file sharing, CPU cycles, and memory. If the protocol was defined with only the three resources above and no way of adding more we would have to re-write the entire protocol and possibly a large number of applications if we decide to add another resource. For this reason the protocol needs to be extendable. Another reason why it would be beneficial for the protocol to be extendable is to help with scalability. For example, suppose we define one message type for file sharing, with a field for file name. Suppose after a period of time in operation we study the contents of this field and find that in a high percentage of the messages the search is for a particular file type, such as avi. If there is a user on the network that only shares mp3 files, they would still have to process all query messages despite the fact that the vast majority of them will not return a result. With an extendable protocol we could introduce an 'avi file query' message, which would specifically deal with searches for avi files. If a user has no avi files the client could be told to pass the message without reading it.

3.1.3 Scalable

The final requirement of the protocol is that it should be scalable. One of the floors in the Gnutella protocol is that the main header that accompanies every descriptor is 22-bytes long. Although this is still quite small, it is still several times longer than other protocol headers such as the Napster header. When dealing with one or two messages, a long header is not a serious problem, but if we scale up the number of messages the header length becomes more of an issue.

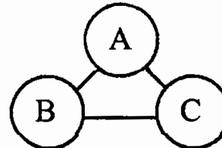


Fig. 3.1 An example of peer to peer network

Small message sizes are the first step in a scalable protocol. Another step is to limit the number of messages that are on the network. This can be done by including time to live (TTL) field. With a TTL field we can set a horizon for the message, the point where the message is no longer forwarded. This stops the message from exiting on the network forever. Another way to limit the messages on the network is to have a unique id for each message, and have the nodes remove any message that they have already seen. Suppose we have a network set up as in Figure 3.1.

Suppose node A sends a message to node B with a TTL of 10. Node B decrements the TTL, and forwards the message to node C. Node C decrements the TTL and forwards the message to node A. If there was no way of identifying the message, node A would have no way of knowing that it has sent the message, and so it would decrement the TTL and forward the message to node B. Node B would have no idea that it had already seen the message, and so would decrement the TTL and forward the message to C. This would happen until the TTL reaches zero. Suppose that there is an id file in the message, the message would go from node A to node B to node C and back to node A. Once at node A, it would recognize that it has sent the message, and so would not forward it anymore.

3.2 Application Requirements

In this section we will talk about the requirements for the resource sharing application we build. These requirements are to implement the protocol, to enable file sharing, to allow for user defined file management, to allow user defined peer management, to allow the network to be mapped, and for the system to be usable.

3.2.1 Implement the Protocol

The first and possibly most important requirement for the application is that it implements the protocol that we will design. It must be able to read in the entire message, process the header, handle the contents of the message if appropriate, and forward the message. Reading in the entire message should be a simple case of reading from an input stream. Processing the header will include tasks such as decrementing the TTL and

determining what type of message this is. Handling the contents of the message is a stage that may not occur in many of the messages received. For example, upon processing the header we may find that the message type is not supported by our application. In this case the message contents will not need to be processed. Finally, forwarding of the message will be decided using several rules specified by the protocol. The application must also ensure that the messages it sends comply with the message formats specified by the protocol.

3.2.2 Implement the file sharing message

We envision that the protocol we design will be used to request many different types of resources to be shared. This does not mean that the application needs to be able to share many different types of resources. In many cases it may not be suitable to have an 'all in one' application that can handle all types of resource sharing requests. Because of this, and because of time constraints, the application should only handle the sharing of files.

3.2.3 User defined file management

The application should allow the user to decide which files they share with the network, along with the location where downloaded files are placed. The first part of this requirement is a security consideration. The user may have confidential files on their system, such as usernames and passwords, or files that they cannot legally share, such as mp3 files, and system files. The application should allow the user to decide which files to share.

Choosing of a download location is mainly a house keeping option. Most users will want their downloads in a separate directory where they can quickly find them and access them.

3.2.4 Peer management

The application should allow the user to decide how many peers it wants to connect to, along with deciding how these peers are. This means the application must allow the user to add and remove peers at will.

There are several reasons for this. First is security. Suppose the application is sharing sensitive information for use within a company. The users will want to specify who connects to them (other members of the company), whilst blocking any other connections.

Another reason for this requirement is performance. Suppose the application starts up and immediately has 5000 peers connected to it. Even if these peers only send one message a second this is still enough to cause significant strain on the system. This could result in poor performance by the user leading to lost messages. This would compromise the performance of the network as a whole.

3.2.5 Network Mapping

The application should provide the user with some information of how the network is constructed. By this we mean it should display the users' peers, those peers' peers, and so on up to the message horizon. This should be done in a scalable manner.

3.2.6 Usability

A general requirement for the system is that it is easy to use. This requirement is a rather subjective one because what one person finds easy, another does not. For example, expert users might find it easy for all operations to be executed through keyboard commands, whereas novice users might prefer a visual interface. For clarity, we will describe usability as an ease with which a novice user can learn to use the system.

Chapter 4
System Design

4. System Design

This chapter describes the design of the system. The first section describes the protocol design, including message formats and suggestions as to how applications should handle messages. The second section will discuss the design of the file sharing application, including architecture design and class break down.

4.1 Protocol Design

The resource sharing protocol is an application layer protocol that passes one of several message types between peers on the network. Although it is envisioned that the protocol will be run on top of TCP and IP, this is not part of the specification. The protocol is to be used in a pure peer to peer environment, and has been designed to be a more optimised version of the Gnutella protocol [14]. The message itself consists of two parts. The first part is a fixed length header used mainly for the routing of the message. The second part is message body, a variable length field depending on the type of message being passed. The message header is shown in Figure 4.1.

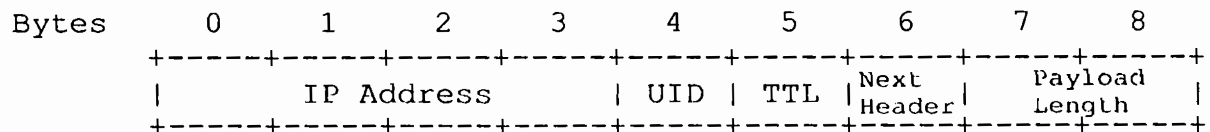


Fig. 4.1 The main header of the resource sharing protocol

The first field in the header is the IP address of the node that originally sends the message. The address is stored in IPv4 dotted format, with each value between the dots being stored in its own byte.

The UID field is a field when coupled with the IP address uniquely identifies the message on the network. This field is an 8 bit unsigned integer. It is up to the client to specify how the UID is chosen. It may seem that 256 possibilities for the UID would make unique identification impossible if a node is used for a long period of time, with 256 values being used within a matter of minutes. This is not the case however because the clients only remember the values for a certain length of time before they timeout. Because of this it is recommended that the clients repeat the sequence of UIDs once they run out.

The third field is the TTL or time to live field. This field is actually split into two sections. The first four bits are an unsigned integer value for the original value of the time to live. These four bits are never changed throughout the life of the message. The last four bits are an unsigned integer representing the actual time to live of the message. These four bits are decremented at every node. This means that for a time to live of 4, the TTL will look as follows:

`0100 0100`	time to live = 4
`0100 0011`	time to live = 3
`0100 0010`	time to live = 2

```
0100 0001        time to live = 1
0100 0000        time to live = 4
```

The reason for this split value for the time to live is to save space in the header whilst ensuring that any response being sent to a message will only have a horizon up to the sending node. Suppose the second node in the above example has a response to send, it will subtract the second 4 bits of the TTL field from the first four bits to set the response TTL to an appropriate value. This is shown below:

```
Original message = 0100 0011       time to live = 3

                   0100
                  -0011
                   0001

Reply = 0001 0001      time to live = 1
```

It may seem that for a file sharing application a horizon of 16 is rather low, considering many nodes may only have a small number of peers. This is a valid argument when looking at a file sharing application, but this protocol is designed to be used for many different types of resource sharing. The reason why this makes a difference is due to the way people will share resources. If a company is sharing processor cycles, it will most likely only want to share them within the company. This can be done by limiting the peers that are connected. All that needs to be done is connect all the nodes within a company. Suppose each node connects to three other nodes, and these nodes are connected to further three nodes. We can see how the network size increases with horizon distance in table 4.2.

T-1/35

Horizon Distance	Number of Nodes
0	0
1	3
2	9
3	21
4	45
5	93
6	189
7	381
8	765
9	1533
10	3069
11	6141
12	12285
13	24573
14	49149
15	98301
16	196605

Table 4.2 Showing how the number of nodes increases with horizon (each node having 3 peers)

We can see, the problem of having a horizon of 16 can be solved using sensible peer selection. The next field is the nextHeader field. This is an unsigned integer that represents the message type that follows this header. The final field is the payload length field. This is 16 bit unsigned integer that represents the number of bytes that follow this header. This means that it is possible to have a message up to 65545bytes ≈512Kbytes long (2^{16} payload bytes + 9 header bytes).

The first of the message type we will discuss is the query message. This message has the nextHeader value of 1 (`00000001`) in the main header. Its format is shown in Figure 4.3.

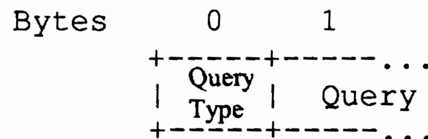


Fig. 4.3 The query message

The query message has a very simple format. The first byte is an unsigned integer specifying the query type. Currently the only query specified is query type 1 (`00000001`), the file query. The file query's query field is an ASCII string containing the elements of the file name to search for.

It is envisioned that queries for other resources (such as spare CPU cycles) will use this message with a separate query type. Having the queries specified in such a way means that we can have up to 256 different query types, satisfying the extendibility requirement.

The second message type we will discuss is the query hit message sent in response to the query message. It has the next header value of 2 (`00000010`) in the main header. Its format is shown in Figure 4.4.

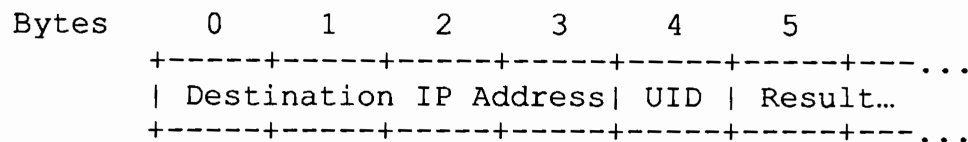


Fig. 4.4 The query hit message

The first two fields of the query hit message are the same as the first two fields of the main header that contained the original query. They are included so that nodes can check to see if the message is destined for them. The next field is the result field. The format of this field is dependent on the original query that was made. Again, the only specified result field is for file queries. Its format is as follows:

```
1st Filename
1st File size (in bytes)
IP address of file owner
2nd Filename
2nd File size
IP address of file owner
⋮ ⋮ ⋮
nth Filename
nth file size
IP address of file owner
```

Each of these fields are ASCII strings, separated by the new line (\n) character. As you can see the IP address of the owner is repeated for every file result. This at first appears to be a waste of bandwidth, but there is a reason for it. Suppose a remote node caches every query hit message it sees. It will soon have a long list of file names/ sizes and the IP addresses of the machines they reside on. If someone queries this machine it can search the list of cached responses to return results faster, or possibly return results that due to horizon size would not have been returned.

When looking at the query hit message it is a valid question to ask why there is not a query hit type field similar to the query type field. The reason for this is to keep the message short and simple. The client should remember all queries it sends using the IP address/ UID combination, and from that remember what type of query was issued.

The final message type that is specified is the network admin message type. This has the nextHeader value of 0 (00000000) in the main header. Its format is similar to the query message (Figure 4.3) with a one byte unsigned integer field to specify the type of network admin message follows. Two such messages have been specified. The first is message type 1 (0000001) the peer query message.

The peer query message does not contain a body. It is simple the main header followed by the admin type field. When a node sees this field it knows that the query originator is requesting a list of all peers that are connected to this node. To reply there are two methods. First (if supported), the node can make a separate connection to the query originator and send an ASCII string of the nodes IP address followed by a space separated list of its peers.

The second method is to send a response along the peer to peer network in a similar fashion to a query hit message. The entire admin message has the format shown in Figure 4.5.

```
Bytes     0     1     2     3     4     5
        +-----+-----+-----+-----+-----+-----+---...
        | type|Destination IP Address | Result...
        +-----+-----+-----+-----+-----+-----+---...
```

Fig. 4.5 The admin message for a peer query response

The admin message type for a peer query response is 256 (11111111). The destination IP address is the address of the node that sends the peer query. Finally, the results are sent as an ASCII string in the format mentioned above.

Out of the two response types, the first is favoured more because it keeps the load off the peer to peer network. The inclusion of the peer query response along the peer to peer network is to respond to users behind a firewall, or those who are limited by the number of connections that can be made.

Now we have the message formats we can discuss how messages are passed. When a node receives a message it checks to see if it has ever seen the message before. It does this by storing the IP address and UID fields of every message processed, and checking all subsequent messages against this list. This list needs to be dynamic, with new message ID's being added, and old ones timing out after a few seconds. If the message has been seen before it means that there is a loop in the network. Rather than process the message again it is just dropped. If the message hasn't been seen before, a copy is made for the node to process, while the TTL of the original is decremented. If the TTL is greater than zero, the message is forwarded. If not, the message is dropped.

Forwarding a message depends on the message type. If the message is a query, each node floods the message along all connections to all peers except the incoming connection. If the message is a query hit, the client can either flood the response along all network connections, or the more preferred method is for the client to forward the query hit along the connection that the original query came down.

4.2 Application Design

In this section we will discuss the design of the file sharing application. The application was designed using an exploratory programming methodology, with the final version being discussed here. An architecture diagram of the application can be found in Figure 4.6.

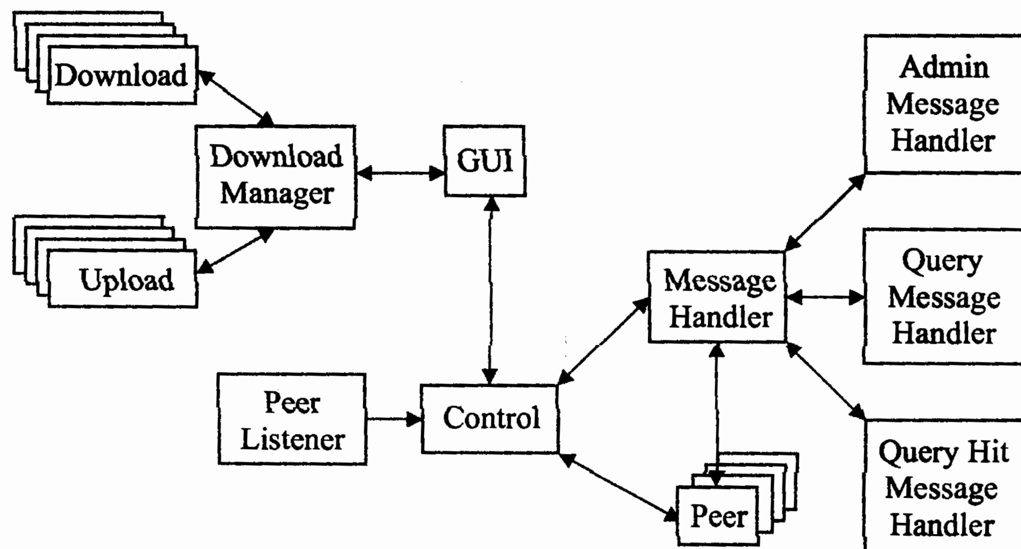


Fig. 4.6 File Sharing Application Architecture

To describe the architecture of the application, we will start at the entry point of the application, the GUI element. We will then branch off and discuss various other sections of the architecture.

The GUI element is the graphical user interface to the application. The main elements it must contain are as follows:

- an area where the user can enter a search string
- an area where results can be displayed
- an area where the user can view status of any current transfers
- an area where the user can specify the maximum number of file transfers
- an area where the user can view a list of current peers/ manage peers
- an area where the user can add a new peer
- an area where the user can specify the maximum number of peers
- an area where the user can view a list of shared files/ manage shared files
- an area displaying the network topology

The GUI element must be able to convert the user input into a message that the remote peers will understand, as well as display the messages sent by the remote peer to this node. Finally, the GUI must be able to pass enough information to the Download Manager element in order for it to start a download.

The Download Manager element is used to hold instances of the Download and Upload elements- end points for all communication that does not occur on the peer to peer network. It stores these instances, along with providing ways to cancel the transfer they represent. The Download Manager creates the Download instances that are needed when the user requests a file to be downloaded. It also listens for any connection attempts, creating an Upload instance to handle the connection.

The Upload and Download elements are the endpoints of the file transfer connections. These elements communicate with each other in the way shown below;

Download	Upload
Attempt connection	Accept connection
Send request	Read request
Read response	Send response
If positive response	*If positive response*
Read file	Write file
Close connection	Close connection

The request that is sent has the format shown in Figure 4.7.

```
Bytes     0     1     3
       +-----+-----+-----+-----+-----+-----+--...
       |RequestSize| File name \n file size
       +-----+-----+-----+-----+-----+-----+--...
```

Fig. 4.7 The file request

The file request is a simple message containing the name and the size (in bytes) of the file in an ASCII string, separated by the new line (\n) character. Preceding this is the size of this string in bytes, stored as a 16 bit unsigned integer.

The response to the request follows a similar format to the original request. The difference is that the string contains an error message instead of a file request. If the response is not a negative response the message is simple the two byte size field with the value 0 (00000000 00000000).

To the other side of the GUI element is the Control element. This element stores instances of the Peer element, the end point of all peer to peer communication. As such, it must provide methods to add and remove peers following user requests to the GUI. The control element also acts as a mid point for all messages that are passed either up from the peer (via the Message handler), or down from the GUI.

The Peer listener element listens for new connections from nodes wishing to become a peer. It simply listens for a connection attempt, and once made passes the socket to the Control element to create a Peer element.

The Peer element is perhaps one of the most important elements of the application. It acts as an end point for the peer to peer communications. It has two major tasks. The first is to send any messages passed to it from the Control element. The second task is to read in messages from the remote peer. This is done by reading in the fixed length header and identifying the length of the payload attached. The payload can then be read in. Once the entire message has been received the Peer element sends it to the Message Handler element.

The Message Handler element reads all messages that have been sent to the application and then decides how they should be treated. Upon receiving a message, the Message Handler users the IP address and UID fields from the header to see if the message has already been seen. This is done by storing these fields from all new messages it receives, with the older ones timing out after a defined time. If the message has been seen before it is dropped, and no more processing is done on it. If the message has not been seen the Message Handler reads the nextHeader field from the message header. If the nextHeader field is one, a copy of the message is passed to the Query Message Handler. If the field is two, a copy of the message is passed to the Query Hit Handler. It the field is zero, a copy of the message is passed to the Admin Message Handler. If the field is of any other value no copies of it are sent anywhere, but the message is passed to the Control element to be forwarded to all connected peers.

The Admin Message Handler checks the payload of the admin message to see what action is required. If the admin message is a Peer Query, the Control element is asked to return a list of the IP addresses of all connected peers. A Peer Query response message is then created and passed to the Control element to be forwarded to the node that made the original request. The original message then has its TTL decremented and (if still valid) is then passed to the Control element (via the Message Handler) to be forwarded to all peers except the one that sent it originally.

If the admin message is a Peer Query Response the Admin Message Handler checks to see if the destination IP address is equal to the applications IP address. If it is, the contents of the message are retrieved and passed to the GUI element via the Message Handler and Control elements. If the destination IP address is not equal to the local IP address, the TTL of the message is decremented and if still valid the message is passed to the Control class via the Message Handler, to be forwarded as appropriate.

The Query Message Handler is used to process file queries. The first task is to check the query message to see if the message is a file query. If not, the TTL is decremented and the message sent for forwarding (as described above). If the message is a file query the query string is retrieved. This string is then searched for against files specified by the user. If any files are found, a query hit message is created and sent for forwarding in the appropriate manner. Once any responses have been sent, the original message has its TTL decremented and it too is forwarded as appropriate.

Finally, the Query Hit Message Handler is used to process query hits. The first task is to see if the query hit is destined for this node using the destination IP address field in the message. If the message is not for this node, the TTL is decremented and the message forwarded. If the message is for this node, the results are extracted from the message and passed to the GUI (via the classes in between) to be displayed to the user.

Chapter 5
Implementation

5. Implementation

This section will discuss the implementation of the file sharing application. We will first of all discuss some general decisions about the implementation before discussing the implementation in more depth.

5.1 General Implementation notes

This section details the choice of language used to implement the application, along with details of the network connections we will use.

5.1.1 Language Choice

The language we used for the implementation of the file sharing application is Java. Java is a platform independent object oriented programming language developed by Sun Microsystems. Originally designed for use in set top boxes, the growth in the World Wide Web opened up a new field for Java applications. It was then seen that the main use for Java could be in adding interactivity to web pages. It is now used mainly as an enterprise business solution, with an emphasis on distributed systems.

5.1.2 Network Choice

There are two main network technologies that can be used for the peer to peer networks connections, TCP sockets and UDP datagram sockets.

TCP, or transmission control protocol, is a way of reliable transition in a network enabled by creating a virtual circuit. The protocol ensures that all data sent by one machine is received by the other machine by through a process of sending acknowledgements for all data received. If the sender does not receive an acknowledgement, it presumes that the data was lost and therefore it retransmits it.

UDP, or user datagram protocol, is a connectionless transport protocol. This can be thought of in the terms of the postal service. If we imagine all the data packets that are sent are letters to be posted, then it no matter what order we send them in we cannot guarantee that they will arrive, or what order they will arrive in. This is in stark contrast to TCP's 'virtual circuit' paradigm where data appears not to get lost or appear in the wrong order.

Because the effectiveness of the peer to peer network depends on the guaranteed arrival of messages sent between nodes, we will use TCP sockets for the connections.

5.2 Application Details

This section deals with the actual implementation of the file sharing application. We will take each of the elements in the design of the system and discuss their translation into actual Java Classes. Firstly, we look at the design of some miscellaneous classes that are used in the application.

5.2.1 Miscellaneous classes

All but one of the miscellaneous classes are data types created to be sent between different classes in the application. The exception is the Util class that is used to perform common operations found in many of the classes. We will discuss each of these classes individually.

5.2.1.1 The Message Class

The message class is a useful data type that is used to store a representation of the message that will be sent between peers. Its object model is shown in Figure 5.1.

The variables in the message class represent the fields in the file sharing protocol. There are three constructors that can be used to create an instance of the Message class. Message() creates a blank message where all of the variables are set to zero. In the case where the variable is an array, the correct length of the array is given, but its contents are all zero.

```
Message
--------------------------------------------------
byte[]        mainHeader;
byte[]        ipAddress;
byte          uId;
byte          ttlField;
byte          nextHeader;
byte[]        payloadLength;
byte[]        payload;
--------------------------------------------------
              Message()
              Message(byte[] b, byte[] pl)
              Message(byte[] ip, byte uid, byte ttl)
void          setIpAddress(byte[] ip)
void          setUId(byte id)
void          setTTLField(byte b)
void          setNextHeader(byte b)
void          setPayloadLength(byte[] b)
void          setPayloadLength(int i)
void          setPayload(byte[] b)
byte[]        getIpAddress()
byte          getUId()
byte          getTTLField()
byte          getNextHeader()
byte[]        getPayloadLength()
byte[]        getPayload()
int           length()
byte[]        toBytes()
String        toString()
byte[]        getQuery()
MessageIdWrapper  getMessageId()
byte[]        getQueryHitPayload()
String        getSearchString()
```

Fig. 5.1 The object model for the Message class

Message(byte[] b, byte[]pl) creates a message where the bytes for the main header are stored in the first byte array argument, and the bytes for the payload are stored in the second byte array.

The Message(byte[] ip, byte uid, byte ttl) constructor allows a message instance to be created with a specified ip address (first argument), UID (second argument), and TTL (third argument).

The methods in the Message class are used to either get or set the variables stored in the class. The exceptions to this are length(), toBytes(), toString(), getQuery(), getMessageId(), getQueryHitPayload(), getSearchString().

The length() method returns an integer value representing the entire length of the message in bytes. This will be the length of the payload variable plus 9 (number of bytes in the header).

The toBytes() method compiles all of the variables data into one byte array, returning this value.

The toString() method is used to obtain a textual representation of the message. Each value of the header is listed separately with its value, as is the payload, which is translated into whichever message type it represents.

The getQuery() method is used to return the all of the bytes in the payload except the first one. This will be the query string in byte form, if the message is a query message.

The getMessageId() method is used to return an instance of the MessageIdWrapper class created using this messages ipAddress and UID variables, along with the current time.

The getQueryHitPayload() method returns all of the bytes in the payload except the first 5. If this is a query hit message, these bytes will be the results (in byte form).

The getSearchString() method returns a string representation of the results of the getQuery() method. This will be the query string.

5.2.1.2 MessageIDWrapper class

The MessageIDWrapper class is a class that stores the message's ID (IP address and UID) along with a value for the time. Its object model is shown in Figure 5.2.

This is a very simple class with only two variables. The messageId variable is a byte array with 5 elements. The first four elements are an IP address, and the last byte is the UID of a message. The long time variable is a long representation of the time when the MessageIDWrapper was created.

The methods in this class simply either return the variables, or in the case of toString() return a string representation of this class.

MessageIdWrapper	
byte[]	messageId;
long	time;
	MessageIdWrapper(byte[] b, long l)
byte[]	getByteArray()
long	getTime()
String	toString()

Fig. 5.2 The object model for the MessageIDWrapper class

5.2.1.3 MessagePeerIdWrapper class

The MessagePeerIdWrapper class is a data type used to store an instance of the message along an integer value used to identify which peer the message arrived from. The object model for this class is shown in Figure 5.3.

MessagePeerIdWrapper	
Message	message
int	peerId
	MessagePeerIdWrapper()
	MessagePeerIdWrapper(Message m, int i)
void	setMessage(Message m)
void	setPeerId(int i)
Message	getMessage()
int	getPeerId()
MessageIdWrapper	getMessageId()

Fig. 5.3 The MessagePeerIdWrapper object model

There are two constructors for this class. The first constructor initiates the message variable to null, and the peerId variable to zero. The second constructor has two arguments used to set the message and integer variables. The methods in the class are used simply to either set the variables or get access to them.

5.2.1.4 Result class

The Result class is used to represent a result returned in the body of a Query Hit Message. Its object model is shown in Figure 5.4.

The three variables in the Result class are all strings representing the three fields returned as part of a result, the file name, the file size, and the owner of the file (IP address).

There are three constructors in the Result class. The first takes no arguments, setting the three variables to empty string values. The next two constructors both take three arguments used to set the variables. The difference between them is that one uses String arguments, while the other uses Object arguments which need to be cast to strings.

```
Result
-------------------------------------------------------------
String      fileName;
String      fileSize;
String      fileOwner;
-------------------------------------------------------------
            Result()
            Result(String s1, String s2, String s3)
            Result(Object s1, Object s2, Object s3)
void        setFileName(String s)
void        setFileSize(String s)
void        setFileOwner(String s)
String      getFileName()
String      getFileSize()
String      getFileOwner()
String      toString()
```

Fig. 5.4 The Result object model

The methods in the result class are simply used to either get or set the three variables, or in the case of toString() the method returns all the results at once.

5.2.1.5 TransferInfo class

The TransferInfo class is used to store information about a file transfer. Its object model is shown in Figure 5.5.

The transferId variable is an integer used by the DownloadManager to identify the particular file transfer. The fileName variable is a string variable that stores the name of the file that this transfer id represents. The progress variable is a string that is used to hold the progress details of the transfer. This is most likely to be either a percentage value, a number representing the number of bytes downloaded/ remaining, or a representation of the time left. Finally, the completed variable is a Boolean used to represent if the transfer has been completed.

```
TransferInfo
-------------------------------------------------------------
int         transferId;
String      fileName;
String      progress;
boolean     completed;
-------------------------------------------------------------
            TransferInfo()
            TransferInfo(String s1, String s2)
            TransferInfo(int i,String s1, String s2, boolean b)
            TransferInfo(int i, String s1, String s2)
void        setTransferId(int i)
void        setFileName(String s)
void        setProgress(String s)
void        setCompleted(boolean b)
int         getTransferId()
String      getFileName()
String      getProgress()
boolean     getCompleted()
```

Fig. 5.5 The TransferInfo object model

The class has for constructor methods associated with it. The first constructor has no arguments, setting the variables of the class to zero, an empty string, or false depending on the variable type.

The TransferInfo(String s1, String s2) constructor is used to set the fileName variable (first argument) and the progress of the transfer (second argument). The body of the constructor also sets the completed variable to false.

The TransferInfo(int i,String s1, String s2, boolean b) and TransferInfo(int i, String s1, String s2) constructors are both used to set the variables in the class to those values specified as arguments. The difference between the two constructors is that the first one takes an argument to specify the completed variable, whereas the second one automatically sets it to false.

5.2.1.6 Util class

Unlike the previous classes in this section the Util class is not used as a data type to hold information to be passed between classes. It is used to provide methods that many classes use, often in relation to bit manipulation. The object model for the Util class is shown in Figure 5.6.

```
Util
-----------------------------------------------------------
-----------------------------------------------------------
boolean     contains(String s1, String s2)
String      byteAddressToStringAddress(byte[] b)
String      intToBinary(int in)
Int         binaryToInt(String s)
byte[]      toByte(int in)
int         toInt(byte[] b)
int         toInt(byte b)
int[]       splitByte(byte b)
byte        combineInts(int[] in)
```

Fig. 5.6 The Util object model

The first method contains(String s1, String s2) is used to see if string s2 is contained in s1. For example, if s2 was 'ace' and s1 was 'space', the method would return true. The pseudo code algorithm for how this is done is shown in Figure 5.7.

```
int n = the length of string s1
int index = 0

for(n times)
    if after the index^th char of s1 the string starts with s2
        return true
    index ++
```

Fig. 5.7 The contains method algorithm

The next method, byteAddressToStringAddress(byte[] b), converts a 4 element byte array into a string representation of an IPv4 address. It does this by calling the toInt(byte b) method, returning the outputs separated by the '.' character.

```
int integerInput
String output

while(integerInput != 0)
    output = (integerOutput % 2) + output
    integerInput = integerInput / 2
```

Fig. 5.8 The intToBinary(int in) method algorithm

The intToBinary(int in) method takes an integer argument and returns a string representation of its unsigned value. For example, if `in` was 25, the method would return '00011001'. If `in` was 300, the method would return '0000000100101100'. The *pseudo code algorithm for how this is done is shown in Figure 5.8.*

binaryToInt(String s) is a method that does the reverse of the above method. It takes in a string representation of a binary number and returns its integer value. The algorithm for how this is done is shown in Figure 5.9.

```
String inputString
Int result
int index = 1

for(all the characters in the inputString)
    if the character = '1'
        result = result + 2^index
```

Fig. 5.9 The binaryToInt(int in) method algorithm

The toByte(int in) method is used to convert the integer argument into a series of bytes, returned in a byte array. This method is useful in places where the payload length field of the protocol's main header needs to be completed. For example, if the method was given the argument 300, the byte array returned would have 2 elements, one would have the value '1', and the other would have the value '44'. The pseudo code algorithm for this method is shown in Figure 5.10.

```
Convert the input int into a binary string
Split the binary string into a set of 8 character strings
Convert these strings into ints
Convert these ints into bytes
Return a byte array containing the bytes.
```

Fig. 5.10 The toByte(int in) method algorithm

The toInt(byte[] b) method is the reverse of the above method. It converts an array of bytes into one integer value. It is important to note that for each of these methods the bytes are taken to be an 8 bit unsigned number. In the case where an array of bytes are used, *the integer returned is not the integer values of all of the bytes added together, but* the integer value of the number if the bytes formed one long binary string.

Similar to the above method, the toInt(byte b) method converts a byte into an integer. The difference between the methods is that this one only deals with single bytes.

The splitByte(byte b) method is used mainly when processing the protocol main header's TTL field. If we remember the TTL byte is actually two 4 bit numbers, this method takes the byte as an argument and returns an integer array containing these two numbers.

The combineInts(int[] in) method is used as the reverse of the above method. When provided with an array of two integers the method converts them into two binary strings, concatenates them, and then converts this string into a byte.

5.2.2 The GUI Element

As shown before, the GUI element is responsible for presenting information to the user, taking user commands and converting them into messages, and sending the correct information to the download element. The GUI element is implemented using thirteen classes as shown in Figure 5.11. We will discuss each class in turn.

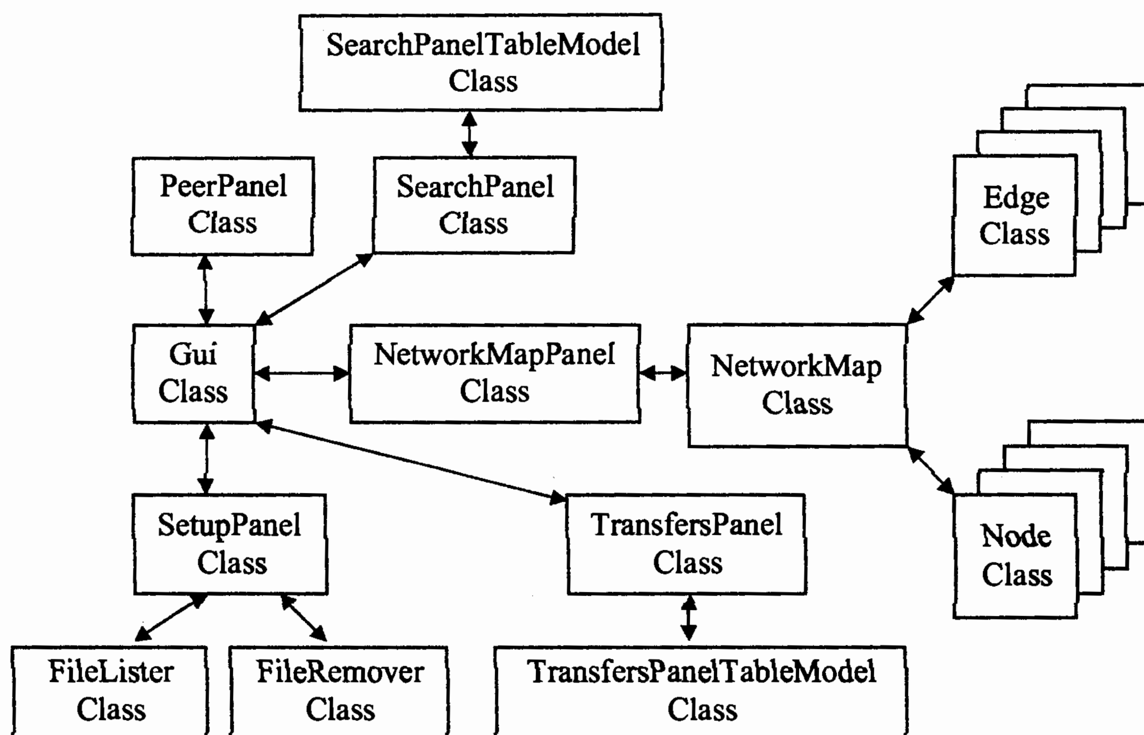


Fig. 5.11 The GUI element class diagram

5.2.2.1 Gui class

The Gui class builds a JTabbedPane to display the main interface to the user. The object model diagram is shown in Figure 5.12.

Most of the variables are Swing components whose function can be inferred by their name. There are also several variables of objects whose names end in 'Panel'. These objects will be discussed in later sections. The String variable 'localAddress' is used to store the IP address of the machine running the application. The Vector variable 'sharedFiles' is a vector that contains instances of the File class that represents the shared files. The File variable 'downloadLocation' is used to store the location of the directory

where downloaded files will be placed. Finally, the in variable 'numberOfDownloads' stores the number of file transfers allowed at any one time.

```
Gui

JPanel            contentPane;        JTabbedPane       jTabbedPane;
JMenuBar          menuBar;            JMenu             menuFile;
JMenuItem         menuFileExit;       JMenuItem         menuFileSave;
PeerPanel         peerPanel;          SearchPanel       searchPanel;
SetupPanel        setupPanel;         TransfersPanel    transfersPanel;
NetworkMapPanel   networkMapPanel;    Control           control;
DownloadManager   downloadManager;    GridLayout        gridLayout;
String            localAddress;       Vector            sharedFiles;
File              downloadLocation;
int               numberOfDownloads;

          Gui()
void      init()
void      build()
void      menuFileSavePressed()
void      menuFileExitPressed()
void      addResults(Vector v)
void      download(Result[] r)
void      search(String s)
void      setDownloadLocation(File f)
void      addTransferInfo(TransferInfo t)
void      setLocalAddress(String s)
void      updateProgress(TransferInfo t, String s)
void      sendPeerQuery()
void      cancelRemoveTransfer(TransferInfo[] t)
void      addSharedFiles(Vector v)
String    getDownloadLocation()
Vector    getSharedFiles()
void      setSharedFiles(Vector v)
void      setNumberOfDownloads(int i)
int       getNumberOfDownloads()
void      setMaxNumberOfPeers(int i)
int       getMaxNumberOfPeers()
int       getNumberOfPeers()
boolean   canAddMorePeers()
boolean   addPeer(String addressToAdd)
Vector    getPeerList()
void      updatePeerList()
void      removePeer(int rI)
String    getLocalAddress()
void      addEdgeToNetworkMap(String s1, String s2)
void      main(String[] args)
```

Fig. 5.12 The Gui object model

The Constructor for the Gui class does not take any variables. It is designed to be the constructor called to start the application. The constructor calls two methods, init(), and then build().

The init() method is used to initialise all of the variables in the Gui class to their appropriate default values.

The build() method is used to construct the user interface. Its first task is to set the size of the application window to 400x400 pixels. Once this has been done, a menu bar is created with two menu items, menuFileSave and menuFileExit. Action listeners are assigned to each of these menu items. After this has been done, the panels are added to the jTabbedPane, and then the jTabbedPane is added to the contentPane.

The menuFileSavePressed() method is called whenever the menuFileSave menu item is pressed. Currently this method has no body, but would be the place where the status of the application could be saved.

The menuFileExitPressed() method is called whenever the menuFileExit menu item is pressed. This method causes the system to exit.

The addResults(Vector v) method is used to pass a Vector of Results to the SearchPanel class' addResult(Vector v) class.

The download(Result[] r) method is used to pass an array of Results to the DownloadManager class' download(Result[] r) class.

The search(String s) method is used to create a query message to send along the network. It first of all create an instance of the Message class via the control class' getEmptyMessage() method. It then sets the next header variable of the message to 1 (query message). Once this has been done it takes the String argument provided and converts it into a byte array. The size of this byte array is used to set the payload length variable in the message. This byte array is then set as the payload variable in the message. Finally, the instance of the message class is passed to the Control class via the sendMessage(Message m) method in the Control class.

The setDownloadLocation(File f) method is used to set the downloadLocation variable in the client to the value specified in this methods argument.

The addTransferInfo(TransferInfo t) method is used to pass instances of the TransferInfo class to the TransferPanel class using its addTransferInfo(TransferInfo t) method.

The setLocalAddress(String s) method sets the value of the localAddress method in the Gui to that supplied by the methods argument.

The updateProgress(TransferInfo t, String s) passes these arguments to the TransferPanel using its updateProgress(TransferInfo t, String s) method.

The sendPeerQuery() method is used to create a Peer Query Message to be passed to the Control class. The first task is to create an instance of the message class using the Control class' getEmptyMessage() method. The next header variable of the message is set to 0, and the payload length of the message is set to 1. The payload of this message is simply a one byte field containing the value 1. This is created and set. The message is then passed to the control class using its sendMessage(Message m) method.

The cancelRemoveTransfer(TransferInfo[] t) is used to pass an array of TransferInfo's to the DownloadManager class using its cancelRemoveTransfer(TransferInfo[] t) method.

The addSharedFiles(Vector v) method is used to append the files stored in the Vector provided as the argument to the sharedFiles variable in the Gui.

The getDownloadLocation() method returns the string representation of the Gui variable downloadLocation.

The getSharedFiles() method is used to access the sharedFiles variable in the Gui.

The setSharedFiles(Vector v) method is used to set the sharedFiles variable to the variable specified as the argument.

The setNumberOfDownloads(int i) method is used to set the numberOfDownloads variable to the value specified as the argument.

The getNumberOfDownloads() method is used to access the numberOfDownloads variable.

The setMaxNumberOfPeers(int i) method is used to set the maxNumberOfPeers variable in the Control class. It does this by passing the argument here to the control class via its setMaxNumberOfPeers(int i) method.

The getMaxNumberOfPeers() method returns the value returned when the Control method getMaxNumberOfPeers() is called.

The getNumberOfPeers() method returns the value returned when the Control method getNumberOfPeers() is called.

The canAddMorePeers() method returns the value returned when the Control method canAddMorePeers() is called.

The addPeer(String addressToAdd) method returns the value returned when the Control method addPeer(String addressToAdd) is called.

The getPeerList() method returns the value returned when the Control method getPeerList() is called.

The updatePeerList() method calls the updatePeerList() method in the PeerPanel class.

The removePeer(int rI) method calls the method removePeer(int rI) in the Control class, using the same argument supplied here.

The getLocalAddress() method returns the String variable localAddress, used to store the IP address of the local machine.

The addEdgeToNetworkMap(String s1, String s2) method is used to call the method of the same name in the NetworkMapPanel class, using the same arguments.

The *main(String[] args) method is the method used to start the application.* Its first task is to set the look and feel of the application to 'Metal'. If this look and feel cannot be found the system will use the default look and feel. Next, a new instance of the Gui class is created. This is then packed and validated before the location of the window can be set to the centre of the screen.

5.2.2.2 SearchPanel class

The SearchPanel class is the first of the 'Panel' classes we shall look at. They all have a similar structure, and they all extend Java's JPanel class. They are used to draw the elements in the particular panel of the user interface. The object model is shown for the *SearchPanel is shown in Figure 5.13.*

```
SearchPanel
---------------------------------------------------------
Gui                       gui;
JLabel                    searchLabel;
JTextField                searchTextField;
JScrollPane               searchScrollPane;
JTable                    searchTable;
SearchPanelTableModel     searchPanelTableModel;
JButton                   searchButton;
JButton                   downloadButton;
GridBagLayout             gridBagLayout;
---------------------------------------------------------
                          SearchPanel(Gui g)
void                      init()
void                      build()
void                      addResults(Vector v)
void                      searchButtonPressed()
void                      downloadButtonPressed()
```

Fig. 5.13 The SearchPanel object model

Most of the variables in the SearchPanel class are Swing components used in the user interface. The two exceptions are gui, the instance of the Gui class that created this SearchPanel, and searchPanelTableModel, the SearchPanelTableModel instance used in the JTable.

The constructor for this class has one argument, Gui g. This is used to set the gui variable to that of the parent Gui. The constructor then calls the init() and the build() methods.

The init() method initialises all of the variables in the class to a suitable value.

The build() method is used to lay the interface components out in the format required. It also adds action listeners to the two JButtons used.

The addResults(Vector v) method is used to pass a vector of Result classes to the SearchPanelTableModel.

The searchButtonPressed() method is called whenever the search button is pressed. This causes the text from the searchTextField to be copied and sent to the Gui using its search(String s) method. If the search string was empty, a message prompting the user to enter a search string is displayed.

The downloadButtonPressed() method is called whenever the download button is pressed. This method requests the row/ rows of the selected results from the SearchPanelTableModel, passing them to the Gui using its download(Result[] r) method.

5.2.2.3 SearchPanelTableModel class

The SearchPanelTableModel class is an extension of the AbstractTableModel class, and is used to display Result classes in the SearchPanel's JTable. The object model for this class is shown in Figure 5.14.

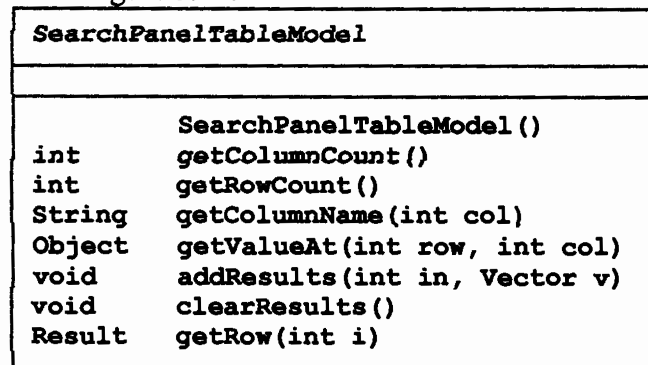


Fig. 5.14 The SearchPanelTableModel object diagram

Most of the methods in this class are relatively simple get methods, used to access data in the table. There are also methods to add data, and also to clear the entire table. The only method of any interest to us is the getRow(int i) method. This method returns an instance of the Result class for the particular row specified in the argument.

5.2.2.4 TransfersPanel

The TransfersPanel is another of the 'panel' classes and so has a similar structure to the SearchPanel. Its object model can be seen in Figure 5.15.

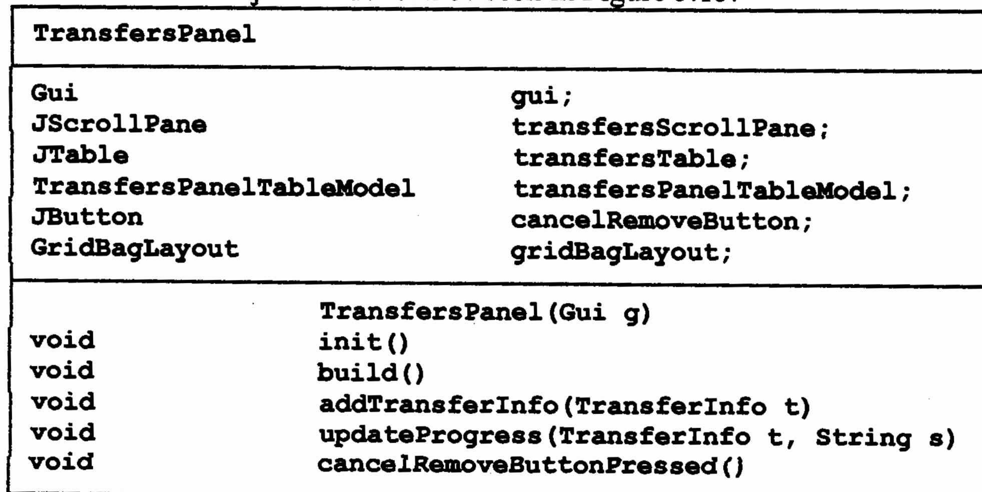


Fig. 5.15 The TransfersPanel object model

The variables and methods in this have similar functions to those found in the other 'panel' classes. The ones that are different are addTransferInfo(TransferInfo t), updateProgress(TransferInfo t, String s), and cancelRemoveButtonPressed() methods.

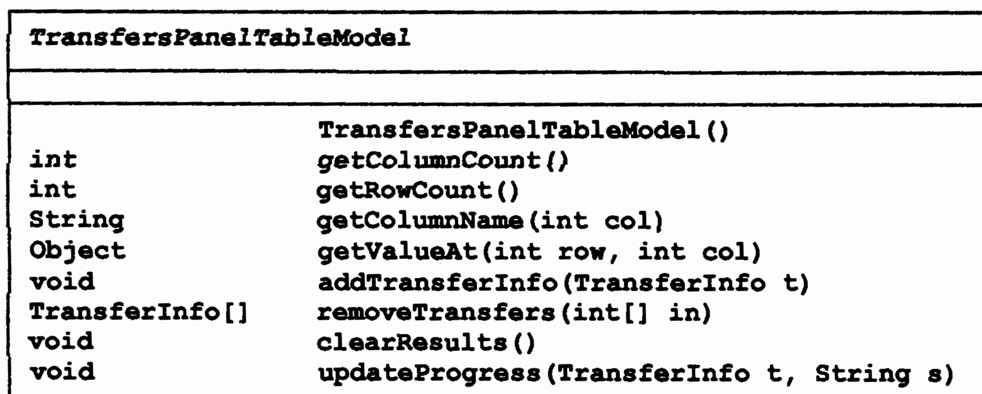
The updateProgress(TransferInfo t, String s) method passes a copy of the original TransferInfo along with the new value for its progress variable to the TransfersPanelTableModel class using its updateProgress(TransferInfo t, String s) method.

The addTransferInfo(TransferInfo t) method simply passes the TransferInfo variable to the TransfersPanelTableModel class using its addTransferInfo(TransferInfo t) method.

The cancelRemoveButtonPressed() method is called when the cancelRemoveButton is pressed. It uses the selected index in the JTable to get a copy of the TransferInfo class that is to be cancelled, passing it to the Gui class using its cancelRemoveTransfer(TransferInfo t) method.

5.2.2.5 TransfersPanelTableModel

The TransfersPanelTableModel is almost exactly the same as the SearchPanelTableModel class. Its object model is shown in Figure 5.16.



TransfersPanelTableModel	
	TransfersPanelTableModel()
int	getColumnCount()
int	getRowCount()
String	getColumnName(int col)
Object	getValueAt(int row, int col)
void	addTransferInfo(TransferInfo t)
TransferInfo[]	removeTransfers(int[] in)
void	clearResults()
void	updateProgress(TransferInfo t, String s)

Fig. 5.16 The TransfersPanelTableModel object diagram

Due to the similarities to the SearchPanelTableModel class, the implementation details will not be repeated here.

5.2.2.5 SetupPanel

The SetupPanel class is another 'panel' class, this one being used to draw the tab used to set the shared files, download location, number of downloads etc. The object model for this class is shown in Figure 5.17.

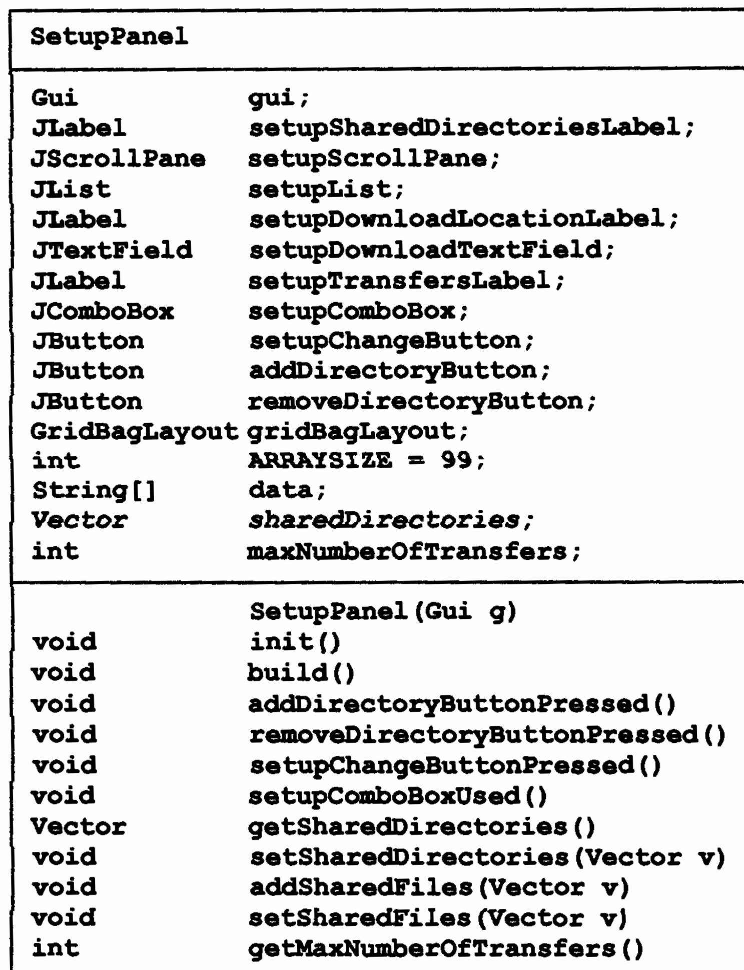


Fig. 5.17 The SetupPanel object diagram

Most of the variables in this class are elements of the user interface, the exceptions are ARRAYSIZE, data, sharedDirectories, and maxNumberOfTransfers. Array size is a fixed integer value. It represents the size of the data variable. The data variable is an array of length ARRAYSIZE containing the strings of all the numbers from zero to ARRAYSIZE – 1. This is used as the data for the setupComboBox. The sharedDirectories variable is a Vector containing Files representing the directories that the application is sharing with the peer to peer network. Finally, the maxNumberOfTransfers variable is an integer used to store the maximum number of file transfers this application will allow.

The constructor, init() and build() methods are similar to that of all the panel classes. The getSharedDirectories() and getMaxNumberOfTransfers() methods simply return the appropriate variable, while the addSharedFiles(Vector v) and setSharedFiles(Vector v) pass their argument to the Gui class using methods of the same name. The setSharedDirectories(Vector v) method is used to set the data that is used in the JList.

The addDirectoryButtonPressed() method is called when the addDirectoryButton button is pressed. It is used to add another directory and its contents to the list of shared files. It does this by creating a JFileChooser that is set to only select directories. If the use

selects a valid directory the method reads in the selected file, passing it to a new instance of the FileLister class.

The removeDirectoryButtonPressed() is used to remove the directory highlighted in the JList when the removeDirectoryButton is pressed. It first of all checks to see if one or more directories have been selected in the JList. If none are selected a warning is given to the user asking to select a directory. If one or more directories are chosen the index of the selection is used to gain access to the corresponding File in the sharedDirectories variable. This is then passed to a new instance of the FileRemover class.

The setupChangeButtonPressed() is used to set the download location to a new directory. The method is fired whenever the setupChangeButton is pressed. The method starts by creating a JFileChooser that is set to only select directories. Once a valid directory is chosen it is passed to the Gui class using its setDownloadLocation(File f) method. It then sets the text of the setupDownloadTextField to the toString() value of the selected file.

The setupComboBoxUsed() is called whenever the user changes the option in the setupComboBox, used for setting the maximum number of file transfers. This method gets the newly selected number and passes it to the Gui class using its setNumberOfDownloads(int i) method.

5.2.2.6 FileLister

The FileLister class is used to obtain a list of all files within a directory. If a directory contains a sub directory the user is prompted as to whether or not they wish to add the contents of it. The object model for this class is shown in Figure 5.18.

```
FileLister
-------------------------------------------
SetupPanel  setupPanel;
Vector      directories;
Vector      files;
File        sourceFile;
Vector      currentDirectories;
-------------------------------------------
            FileLister(SetupPanel sp, File s)
void        getFiles(File f)
boolean     isAlreadySelected(File f)
void        setSharedDirectories()
void        setSharedFiles()
void        run()
```

Fig. 5.18 The FileLister object model

The setupPanel variable is an instance of the SetupPanel class set to the SetupPanel that created this FileLister. The Vectors directories and files are used to store instances of the File class that represent the shared directories and files. The sourceFile variable is a variable used to store the directory whose contents we are sharing. The Vector currentDirectories is used to hold the list of current shared directories.

The constructor of this class takes two arguments, a SetupPanel instance used to tell the FileLister the SetupPanel that created this instance, and a File instance used as the source directory whose contents we are listing. The constructor sets both of these arguments to the appropriate variables along with initialising the directories and files Vectors to new Vectors. The constructor then sets the currentDirectories variable to the list of directories currently shared by calling the getSharedDirectories() method in setupPanel. Finally, the constructor calls the start() method used to run this thread.

The run() method is used to start the thread running. It first of all adds the sourceFile variable to the directories vector, then calls the getFiles(File f) method. Once this has completed, the method calls the setSharedDirectories() and the setSharedFiles() method.

The getFiles(File f) method is the main method used to get a list of the files inside of a directory. The pseudo code algorithm that does this is shown in Figure 5.19.

```
Use the directory to make an array of its contents

For all of the elements in the array
    If the file hasn't already been selected
        If the file is a directory
            Ask the user if they want the directory contents
              If yes
                Add the file to the directories vector
                Call the getFile method on this file
        Else if the file isn't a directory
            Add the file to the files vector
```

Fig. 5.19 The pseudo code algorithm for the getFiles(File f) method

The isAlreadySelected(File f) method is used to check if the directory we are adding has previously been added. It does this by comparing the argument with all the elements in the directories Vector. If any of the files match, the method returns true, otherwise it returns false.

Finally, the setSharedFiles() and setSharedDirectories() methods are used to call the methods in the setupPanel class to set the shared files and shared directory vectors.

5.2.2.7 FileRemover

```
FileRemover
-----------------------------------------------
SetupPanel  setupPanel;
File        fileToRemove;
Vector      files;
Vector      currentDirectories;
-----------------------------------------------
            FileRemover(SetupPanel s, File f)
void        getDirectoryContents(File f)
void        run()
```

Fig. 5.20 The FileRemover object model

The FileRemover class is used to remove a directory and the files it contains from the list of shared directories. It does not remove any sub-directories automatically, meaning that the user will need to manually ask for this to be done. The object model for this class is shown in Figure 5.20.

The variables in the FileRemover class match those specified in the FileLister class. The FileRemover constructor also works in a similar fashion, setting the variables from the appropriate arguments and calling the start() method.

The run() method is called when the constructor calls the start() method. It does three main tasks. Firstly it searches through the list of current directories until it finds the directory specified to remove. It then removes this from the list. Now it has done this the second task is to take the list of directories and re-build the list of shared files by getting the contents of each directory. The final task is to use the setSharedDirectories(Vector v) and setSharedFiles(Vector v) methods in the setupPanel class to store the new file lists.

The getDirectoryContents(File f) is the method called by the run() method to obtain the list of files within the directory specified as the argument.

5.2.2.8 PeerPanel

The PeerPanel is the 'panel' class used to draw all interface components used to manage the peers connected to this application. Its object model is shown in Figure 5.21.

```
PeerPanel
-----------------------------------------
Gui             gui;
JLabel          peerLabel;
JScrollPane     peerScrollPane;
JList           peerList;
JTextField      peerTextField;
JButton         addPeerButton;
JButton         removePeerButton;
JButton         discoverButton;
JLabel          maxPeerLabel;
JComboBox       maxPeersComboBox;
GridBagLayout   gridBagLayout;
int             ARRAYSIZE = 99;
String[]        data;
-----------------------------------------
                PeerPanel(Gui g)
void            init()
void            build()
void            addPeerButtonPressed()
void            removePeerButtonPressed()
void            updatePeerList()
```

Fig. 5.21 The PeerPanel object model.

As with the other 'panel' classes most of the variables are interface components. The ARRAYSIZE and data variables are the same as those found in the SetupPanel class. Most of the methods in this class are also the same as those found in the other panel

classes, the difference being the addPeerButtonPressed(), removePeerButtonPressed(), and updatePeerList() methods.

The addPeerButtonPressed() method is used to handle the addPeerButton being pressed. Its task is to check to see if anything was entered into the peerTextField and if anymore peers can be added. If the answer to both of these is yes, the method calls the addPeer(String s) method in the Gui class to attempt to add the peer specified in the peerTextField. If any errors are encountered the user is notified.

The removePeerButtonPressed() method is similar to the addPeerButtonPressed() method except its task is to remove the peer highlighted in the peerList. It does this by retrieving the index of the selected item and passing it to the Gui using its removePeer(int i) method.

The updatePeerList() method is used to set the data in the JList. It does this using the vector returned when calling the Gui's getPeerList() method.

5.2.2.9 NetworkMapPanel

The NetworkMapPanel class is the last of the 'panel' classes. It is used to hold the NetworkMap instance, as well as the interface components used to interact with it. The object model can be found in Figure 5.22.

As with the previous four panel classes, the variables and the constructor, init(), and build() methods are the same as before.

```
NetworkMapPanel
--------------------------------------------------------------
Gui             gui;
NetworkMap      networkMap;
JButton         updateButton;
JButton         clearButtoni;
GridBagLayout   gridBagLayout;
--------------------------------------------------------------
                NetworkMapPanel(Gui g)
void            init()
void            build()
void            addEdgeToNetworkMap(String s1, String s2)
void            updateButtonPressed()
void            clearButtonPressed()
```

Fig. 5.22 The NetworkMapPanel object model

The addEdgeToNetworkMap(String s1, String s2) method is used to add a new node to the network map (s1), along with the connection to an existing node (s2). This method simple passes the arguments to the NetworkMap class.

The updateButtonPressed() method is called when the update button is pressed. It is used to clear the NetworkMap, add the central node (i.e. this instance of the application) and call the method in the Gui responsible for sending out an Peer Query message (sendPeerQuery()).

The clearButtonPressed() method is used to clear the networkMap by calling its clearMap() method.

5.2.2.10 NetworkMap

The NetworkMap class is used to draw a representation of the peers in the network and the connections between them. It is based on the Graph and GraphPanel classes developed by Sun Microsystems as examples of the power of java applets[17]. The object model of this class is shown in Figure 5.23.

NetworkMap	
NetworkMapPanel	networkMapPanel;
Thread	relaxer;
Vector	nodes;
Vector	edges;
Image	offscreen;
Dimension	offscreensize;
Graphics	offgraphics;
Node	pick;
boolean	pickfixed;
	NetworkMap(NetworkMapPanel nmp)
void	run()
Node	findNode(String s)
void	relax()
void	update(Graphics g)
void	start()
void	stop()
void	clearMap()
void	paintNode(Graphics g, Node n)
boolean	isNode(String nn)
void	addFixedNode(String nn)
void	addNode(String nn)
void	addEdge(String f, String t)
void	mouseClicked(MouseEvent e)
void	mousePressed(MouseEvent e)
void	mouseReleased(MouseEvent e)
void	mouseEntered(MouseEvent e)
void	mouseExited(MouseEvent e)
void	mouseDragged(MouseEvent e)
void	mouseMoved(MouseEvent e)

Fig. 5.23 The object model for the NetworkMap class

Due to this class being a cut down/optimised version of the Sun Microsystems class, the details of the implementation will not be detailed here.

5.2.2.11 Node

The node class is used by the NetworkMap class to represent a node on the network. Again this is a class created by Sun Microsystems[17]. The object model is shown here for the sake of completion (Figure 5.24).

```
Node
---------------------------------------------
String      nodeName;
double      xPosition;
double      yPosition;
double      dx;
double      dy;
boolean     fixed;
---------------------------------------------
void        Node()
void        setNodeName(String n)
void        setXPosition(double d)
void        setYPosition(double d)
void        setDx(double d)
void        setDy(double d)
void        setFixed(boolean b)
String      getNodeName()
double      getXPosition()
double      getYPosition()
double      getDx()
double      getDy()
boolean     getFixed()
```

Fig. 5.24 The object model of the Node class

5.2.2.12 Edge

The Edge class is used by the NetworkMap class to represent the link between two nodes. Again this class was created by Sun Microsystems and so is shown here only for the sake of completion (Figure 5.25).

```
Edge
---------------------------------------------
String      from;
String      to;
double      length;
---------------------------------------------
            Edge()
            Edge(String f, String t, double d)
String      getFrom()
String      getTo()
double      getLength()
void        setFrom(String s)
void        setTo(String s)
void        setLength(double d)
```

Fig. 5.25 The Edge object model

5.2.3 The Download Section

The download section of the application consists of three classes, the DownloadManager, Upload, and Download classes (Figure 5.26).

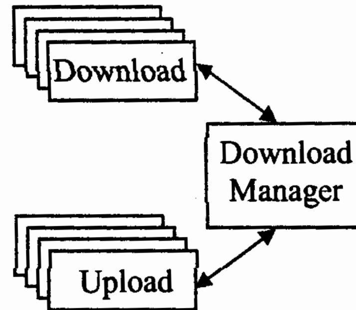


Fig. 5.26 The download section of the application

5.2.3.1 DownloadManager

The DownloadManager class is used to initiate downloads, via creation of Download instances, listen and accept upload attempts, via Upload instances, and also to manage connections once they have started. The object model for the DownloadManager is shown in Figure 5.27.

```
DownloadManager
-------------------------------------------------
Gui          gui;
Vector       downloads;
Vector       uploads;
int          PORTNUMBER = 7660;
int          idNumber = 0;
-------------------------------------------------
             DownloadManager(Gui g)
void         download(Result[] r)
void         error(String s)
void         addTransferInfo(TransferInfo t)
void         updateProgress(TransferInfo t, String s)
void         removeUpload(int in)
void         cancelRemoveTransfer(TransferInfo[] t)
void         removeDownload(int in)
boolean      canAddMoreTransfers()
Vector       getSharedFiles()
void         run()
```

Fig. 5.27 The DownloadManager object model

The Gui variable is used to store the instance of the Gui class that created this DownloadManager. The downloads and uploads vectors are used to store instances of the Download and Upload classes. The PORTNUMBER integer is the port number that the class will listen on for download attempts. The idNumber variable will be used to give instances of the Upload/ Download class an id number.

The DownloadManager constructor takes one argument, a Gui instance. This instance is used to set the gui variable. The constructor also initialises all of the variables to appropriate values before calling the start() method.

The run() method is called when the start() method is run. It is used to listen for connection attempts. It creates a server socket on port PORTNUMBER and blocks until a connection is made. This is then used to create an instance of the Upload class which is

stored in the uploads vector. The body of the method loops continually while the application is run.

The download(result[] r) method is used to create instances of the Download class using the Results in the argument. Before each instance is created the canAddMoreTransfers() method is called to ensure the transfer limit has not been reached. The Downloads are stored in the downloads vector.

The error(String s) method is used to display error messages provided as the argument to the user.

The addTransferInfo(TransferInfo t), updateProgress(TransferInfo t, String s), getSharedFiles(), and getDownloadLocation() all call the corresponding methods in the Gui class.

The removeUpload(int i) and removeDownload(int i) methods are similar in that they both remove either an Upload or a Download instance from the vectors using its id number, provided as the argument.

The cancelRemoveTransfer(TransferInfo[] t) method is used to remove a transfer similar to the above two methods, but this method uses TransferInfo to identify them.

The canAddMoreTransfers() method is used to decide if the uploads and downloads vector sizes combined have exceeded the maximum number of transfers allowed. It returns false if transfers are still allowed.

5.2.3.2 Download

The Download class is the endpoint of a download attempt. Its object model can be seen in Figure 5.28.

```
Download
-----------------------------------------------------------------
DownloadManager  downloadManager;
Result           result;
int              id;
int              PORTNUMBER = 7660;
-----------------------------------------------------------------
                 Download(DownloadManager dm, Result r, int i)
int              getId()
void             run()
```

Fig. 5.28 The Download object model

The Download class has 4 variables. downloadManager is the instance of the DownloadManager class that created this download. result is the instance of the Result class that contains the information about the file to do downloaded. The id variable is used to store the transfer id for this transfer, and the PORTNUMBER contains the port number that the connection will be made on.

The constructor for the Download class takes three arguments used to initialise the variables. The constructor then calls the start() method.

Once the start() method is called the run() method is accessed. This method creates the connection to the remote node and downloads the file. This is done in the same way as specified in before.

The getId() method is used to return the id variable.

5.2.3.3 Upload

The Upload class is the opposite end to the Download class. It is very similar to the Download class. The object model for this class is shown in Figure 5.29.

```
Upload
------------------------------------------------------------
DownloadManager   downloadManager;
Socket            socket;
int               id;
Vector            downloadableFiles;
File              file;
------------------------------------------------------------
                  Upload(DownloadManager dm, Socket s, int i, Vector v)
void              run()
boolean           canDownload(String s)
int               getId()
```

Fig. 5.29 The Upload object model

The downloadManager and id variables are the same as in the Download class. socket is the instance of the Socket class made by the DownloadManager class. The downloadableFiles vector contains a list of all the files currently shared. File is used to store the details of the file that the remote user requests to download.

The constructor for the Upload class uses its arguments to initialise the variables, and to start the thread running.

The run() method that is started when the thread is run uses the socket to listen for the file request, and if available writes the file.

The canDownload(String s) method is used to check whether a file with the filename 's' is one of the shared files. If so the method return true, otherwise the method returns false.

Finally, the getId() method returns the id variable of this instance of the Upload class.

5.2.4 The Network Section

The final part of the implementation is the section dealing with creating and maintaining a working peer to peer network. The classes involved in doing this are shown in Figure 5.30. Each of these classes will be discussed in turn.

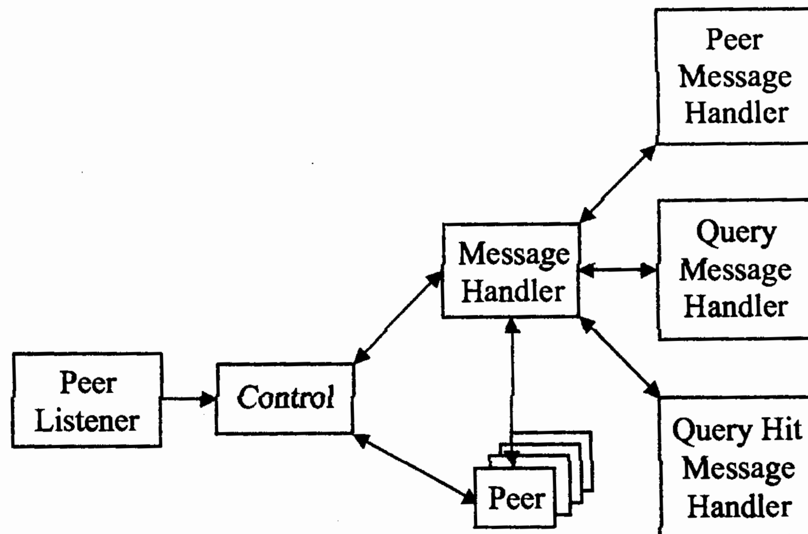


Fig. 5.30 The classes in the networking section of the application

5.2.4.1 Control class

The Control class acts as a central point for the networking section. It acts as the intermediary between the user interface and the three key sections of the network section, the PeerListener, the Peers, and the MessageHandler. The object model for this class is shown in Figure 5.31.

The Gui variable is the instance of the Gui class that created this Control instance, with PeerListener and MessageHandler being the instances that this control class creates for the application to use. The sharedFiles vector is used to store a copy of all the files that are shared by this system. The peers vector is used to store instances of the peer class that will represent each peer connection. The maxNumberOfPeers variable is used to determine the maximum number of peers this application is allowed to have, with the PORTNUMBER being the port number on which the peers connect on. The idNumber field is an integer that will be used to give the instances of the peer class an id number, with uid acting as the same but for use within message classes. Finally, localAddress is used to store the IPv4 address of the local machine, and TTL is the default time to live value for the messages.

The Control constructor takes one argument, the Gui that created it. The constructor sets this to the appropriate variable, whilst initialising the rest of the variables. The PeerListener and MessageHandler classes are also created by the constructor.

The getMaxNumberOfPeers(),getNumberOfPeers(), and the getLocalAddress() methods all return the values which the method name insinuates.

Conversely the setMaxNumberOfPeers(int i) and the setSharedFiles(Vector v) set the two values insinuated.

The two addPeer methods are used to create a new instance of the peer class and add it to the vector of peers, The method that takes a socket as an argument is used by the

peerListener to accept incoming connection attempts, with the other method being used by the gui class.

```
Control
----------------------------------------------------------
Gui                 gui;
PeerListener        peerListener;
Vector              sharedFiles;
Vector              peers;
MessageHandler      messageHandler;
int                 maxNumberOfPeers;
int                 PORTNUMBER = 7659;
int                 idNumber;
int                 uid;
byte[]              localAddress;
byte                ttlField;
----------------------------------------------------------
                    Control(Gui g)
int                 getNumberOfPeers()
int                 getMaxNumberOfPeers()
boolean             canAddMorePeers()
void                setMaxNumberOfPeers(int i)
boolean             addPeer(String s)
void                addPeer(Socket s)
void                removePeer(Peer p)
void                removePeer(int index)
Vector              getPeerInfo()
String[]            getPeerAddresses()
void                setSharedFiles(Vector v)
void                sendMessage(Message m)
void                reply(int in, Message m)
void                forwardMessage(Message m, int id)
Message             getEmptyMessage()
void                setResults(Vector v)
byte[]              getLocalAddress()
void                addEdgeToNetworkMap(String f, String t)
```

Fig. 5.31 The Control class object model

The two removePeer methods are used to find and remove the specified peer from the vector of peers, thus disconnecting the remote machine. The peer can either be identified by its peer id number, or by passing a copy of itself to the method.

The getPeerInfo() method is used to obtain a vector of strings containing the addresses of the peers. This method is used by the Gui class for use in displaying the information about the peers.

The getPeerAddresses() method is similar to the getPeerInfo() method, however this one returns the results as a string array.

The sendMessage(Message m) method is used to pass along a message to all of the nodes connected to the application. It simply calls the send(Message m) method of all the peers in the peers vector.

The replyMessage(int i, Message m) method is used to send the specified message to just one peer, specified by the peer ID, i.

The forwardMessage(int i, Message m) method is the reverse of the replyMessage method. Rather than just reply to the peer specified, this method replies to all peers except the one specified.

The getEmptyMessage() method is used to return a copy of the message class with the IP address, UID, and TTL fields set to the correct values.

Finally, the setResults(Vector v) and the addEdgeToNetworkMap(String f, String t) methods pass their arguments to the corresponding methods in the Gui class.

5.2.4.2 PeerListener class

The PeerListener class is used to listen for connection attempts by other nodes. Once a socket connection has been established it is passed to the control class in order to create an instance of the peer class. The object model of the PeerListener is shown in *Figure* 5.32.

PeerListener	
Control	control;
int	PORTNUMBER = 7659;
	PeerListener(Control c)
void	run()

Fig. 5.32 The PeerListener object model.

The control variable is used to store the instance of the Control class that created this PeerListener. The PORTNUMBER variable is used to store the port number that the communication will be made on.

The constructor for the PeerListener sets the control variable to that of its argument, then starts the thread running.

The run() method creates a serverSocket and sits in a loop waiting for someone to connect. Once a connection has been made, the socket is passed to the control class using its addPeer(Socket s) method.

5.2.4.3 Peer class

The Peer class is used as the end point for all peer to peer communications. The Control class will store one of these classes for each peer it has. The class provides methods to read messages from the network, and also to send messages. Figure 5.33 shows the object model for this class.

The peerId variable is an integer that uniquely identifies this peer to the control class. The inetAddress variable is the IPv4 address of the remote node connected to this

peer, with the stringAddress variable storing the same information but in string format. The socket variable stores the Socket that is used to connect to the remote node, with the DataInputStream and DataOutputStream variables holding the readers and writers to actually send the bytes down. The control variable is used to store the instance of the control class that created this peer, with the messageHandler storing the instance of that class where the messages are passed to.

```
Peer
---------------------------------------------------------------
int                 peerId;
InetAddress         inetAddress;
String              stringAddress;
Socket              socket;
Control             control;
boolean             connected;
DataInputStream     in;
DataOutputStream    out;
MessageHandler      messageHandler;
---------------------------------------------------------------
             Peer(Control c, Socket s, int i, MessageHandler mh)
String       getAddress()
byte[]       getByteAddress()
void         disconnect()
void         internalDisconnect()
void         send(Message m)
void         run()
int          getId()
```

Fig. 5.33 The Peer Object model

The constructor for the Peer class takes a control, socket, integer, and mesageHandler instance as arguments. These are all self explanatory with the exception of the integer, which is used to set the id number of this peer. Once the constructor has set all of these variables it creates the dataOutputStream and starts the thread running.

The getAddress() and getByteAddress() methods both return the IPv4 address of the remote node. The difference between the classes is that the first method returns a string whilst the second method returns a byte array.

The internalDisconnect() and disconnect() methods both do the same tasks. They close all readers/ writers and the socket before removing the peer from the control class. The difference between the methods is that the disconnect method is called by the control class when the local node wished to remove the connection, whilst the internalDisconnect method is used when the remote node terminated the connection.

The send(Message m) method is used to send the message to the remote node. It converts the message into a byte array using its toBytes() method, and then uses the dataOutputStream to send it.

The run() method contains a continuing loop where messages are read in and passed to the message handler. It does this by first reading in 9 bytes (the message header) and creating a new instance of the message class. It then uses this class to find the

length of the payload still waiting to be read in. Once this is discovered it is read in a stored in the message class. The message is then passed to the messageHandler.

The final method is the getId() method. This simply returns the peerId variable.

5.2.4.4 MessageHandler class

The MessageHandler class is the first place messages are sent once they have been received by the peer. The class first ensures that the message has never been seen before by this node before passing it on to the appropriate message handler type. Figure 5.34 shows the classes object model.

```
MessageHandler

Control                  control;
PeerMessageHandler       peerMessageHandler;
QueryHandler             queryHandler;
QueryHitHandler          queryHitHandler;
Vector                   messages;
Vector                   messageIds;
Vector                   sharedFiles;
long                     MESSAGE_TIMEOUT_VALUE = 60000;
String                   localAddress;

                         MessageHandler(Control c)
void                     addMessage(MessagePeerIdWrapper m)
void                     setResults(Vector v)
void                     setSharedFiles(Vector v)
boolean                  haveRecieved(MessageIdWrapper miw)
void                     handleMessage(MessagePeerIdWrapper m)
void                     forwardMessage(Message m, int i)
String                   search(String s)
Message                  getEmptyMessage()
void                     reply(int peerId, Message toSend)
void                     run()
void                     checkTimeOut()
void                     removeBefore(int in)
void                     addEdgeToNetworkMap(String f, String t)
String[]                 getPeerAddresses()
```

Fig. 5.34 The MessageHandler object model

The control, peerMessageHandler, queryHandler, and queryHitHandler all refer to variables that store instances of those classes. The message Vector is used to store all messages that are waiting to be processed by the class, while the messageIds vector stores instances of the MessageIDWrapper class created by the Messages. sharedFiles keeps a copy of the vector containing all of the files shared by the system, while MESSAGE_TIMEOUT_VALUE and localAddress store their obvious values.

The constructor for this class initialises all of the variables in it, and starts the thread running.

The addMessage(MessagePeerIdWrapper m) method adds the argument to the messages vector.

The addEdgeToNetworkMap(String f, String t), getPeerAddresses(), getEmptyMessage(), reply(int peerId, Message toSend), forwardMessage(Message m, int i), and setResults(Vector v) methods all call their corresponding methods in the control class with their argument.

The haveRecieved(MessageIdWrapper miw) method is used to check whether or not this node has received the message beforehand. It searches through the messageIds vector to see if the MessageIdWrapper supplied as the argument is found. If it is, the method returns true, otherwise false.

The handleMessage(MessagePeerIdWrapper m) method is used to send messagePeerIdWrappers to the correct message handler. It does this by looking at the message's nextHeader field, sending them to the appropriate class based on its value.

The search(String s) method is used to see if the String in the argument is found in the filenames of any of the shared files. It steps through each shared file, testing it against the string using the contains(String s1, String s2) method found in the Util class.

The run() method is used to call the handleMessage(MessagePeerIdWrapper m) method for all of the elements in the messages vector. When there are no messages waiting the method calls the checkTimeOut() method.

The checkTimeOut() method is used to time out the messageIdWrappers in the messageIds vector. It does this by subtracting the time when the message id was stored from the current time. If the value is greater than the timeout value the ID is removed from the vector.

5.2.4.5 PeerMessageHandler class, QueryHandler class, and QueryHitHandler class

These three classes are very similar to each other. They each run a thread that reads their particular message type and performs an action depending on the result. The QueryHandler does a search on the filename, replying with a query hit if a result is found. The PeerMessageHandler returns a peerQueryHit message to the response for a peer list. Both of these classes decrement the TTL of the original message and forward it on to the other peers.

The query hit handler checks to see if the result it has received is destined for it. If it is, the results are passed up to the gui via the intermediary classes. If not, the class forwards the message to the other peers.

Chapter 6
The System in Operation

6. The System in Operation

This section shows the system as it looks when it is being used. We will split this into five sections, one for each of the tabbed panes.

6.1 Search Tab

The search pane initially presents the user with a text field to enter a search string and an empty table where the results will be displayed (Figure 6.1).

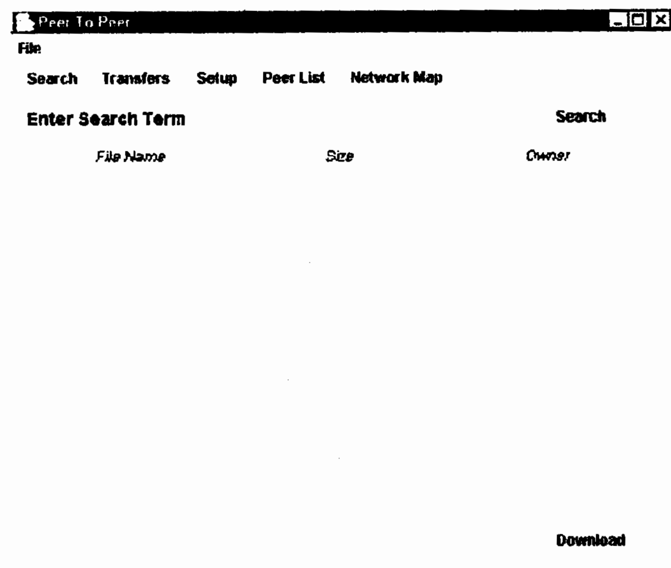


Fig. 6.1 The Search Tab at start up

Entering a search term in the text field and clicking search/ pressing enter will perform a search amongst peers for files matching the search term. The results are displayed in the table (Figure 6.2).

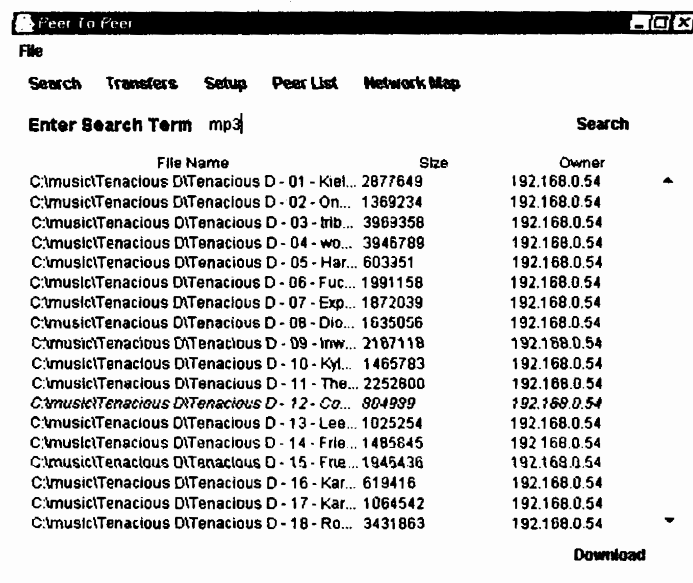


Fig. 6.2 The Search Tab after a search

The user can then highlight a file/ files and press the download button.

6.2 Transfers Tab

Upon start-up, the transfers tab has an empty table where details of downloads will be placed (Figure 6.3).

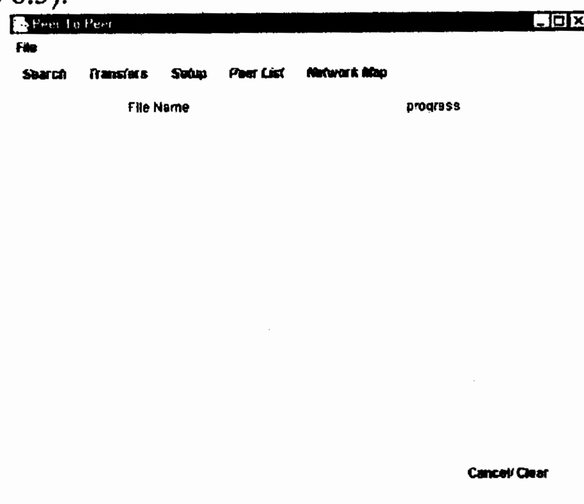


Fig. 6.3 The Transfers Tab at start up

Once a download is started on the search tab, or a remote download is requested, the transfers tab lists that info (Figure 6.4).

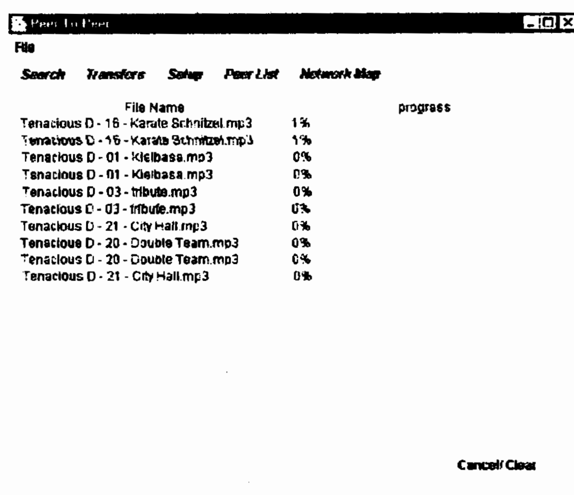


Fig. 6.4 The Transfers Tab during a file transfer

Note that the reason why the above screen shot has the file names listed twice is due to the application downloading the file from itself (done for the purposes of the example).

6.3 Setup tab

The setup tab is used to add/ remove shared directories, to specify the maximum number of downloads, and to specify the location where the downloaded files will be placed. The tab is shown in Figure 6.5.

Upon pressing the Add... button, a file chooser window is created (Figure 6.6). The user uses this to select the directory to share.

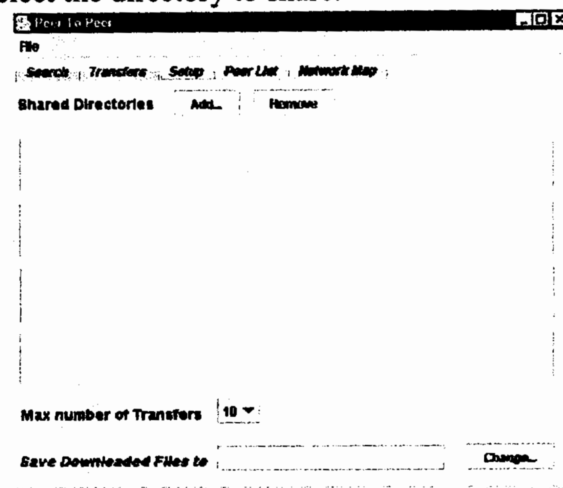


Fig. 6.5 The Setup Tab at start up

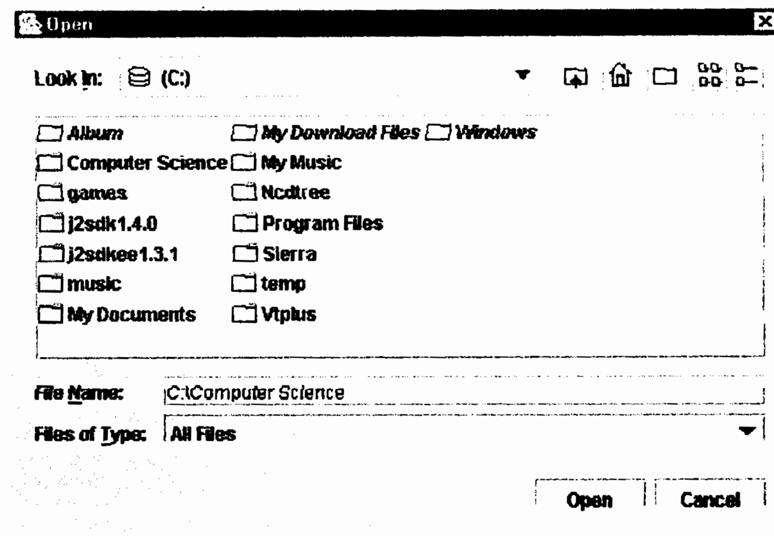


Fig. 6.6 The directory chooser

Once the Open button is pressed the directory is shared. If a sub-directory is found, the application prompts the user as to whether or not it should also be shared (Figure 6.7).

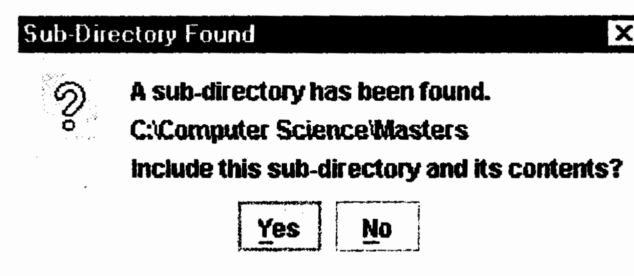


Fig. 6.7 The Sub-directory prompt

A shared directory can be removed by highlighting it and pressing the Remove button. The combo-box towards the bottom of the tab is used to select the number of

concurrent uploads/ downloads that can occur. This can have a value between 1 and 99. Finally, pressing the Change... button brings up a directory choose as shown in Figure 6.6. The directory chosen in this case sets the download location for the files.

6.4 Peer List Tab.

Upon start up, the peer list tab displays an empty list where the peer details will go once one has been connected, a combo box to set the maximum number of peers, and buttons to add, remove and discover peers (Figure 6.8).

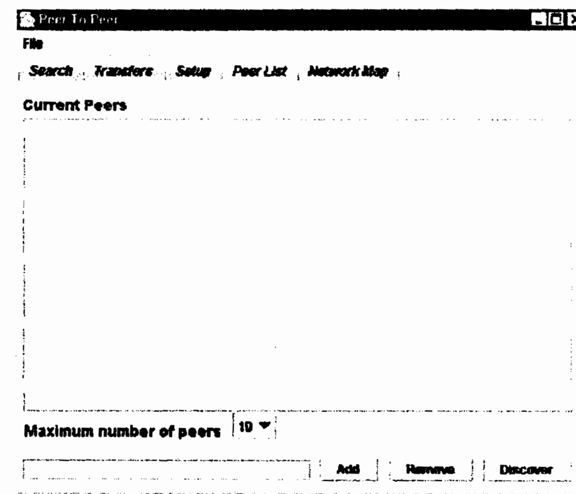


Fig. 6.8 The Peer List Tab at start up

To connect to a peer, the address is entered into the text area at the bottom left of the screen and either the enter key or Add button is pressed. If the connection is accepted, the peer details are listed in the Current Peers list (Figure 6.9).

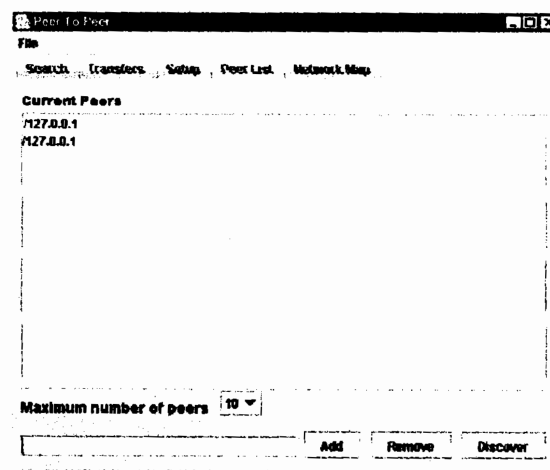


Fig. 6.9 The Peer List Tab after adding peers

Removing a peer is done by highlighting it in the peer list and clicking the remove button.

6.4 Network Map Tab.

Upon start-up, the network map tab shows an empty map and two buttons (Figure 6.10). The map also looks like this after the Clear button has been pressed.

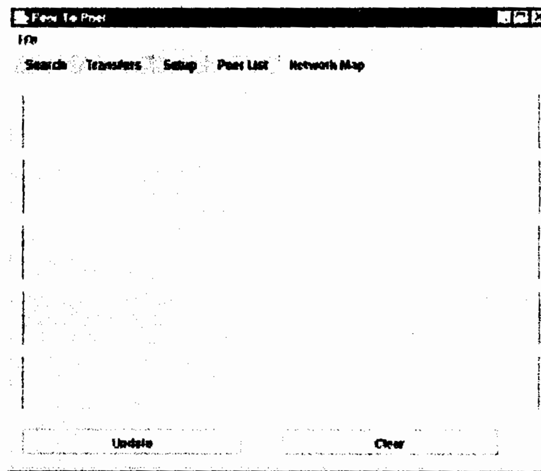


Fig. 6.10 The Network Map Tab at start up

Pressing the Update button after one or more peers have been added causes the map to display the current topology of the peer to peer network (Figure 6.11).

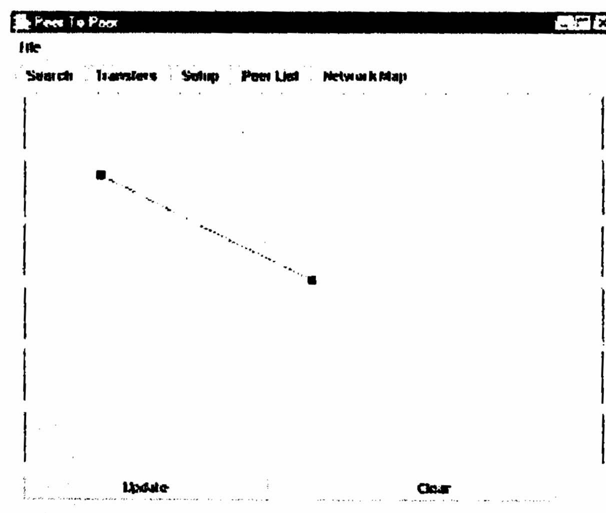


Fig. 6.11 The Network Map tag after a node has joined the network

Chapter 7
Results and Conclusion

7. Results and Conclusion

This final section of the report will discuss the results and conclusions of the project. First we will discuss each of the objectives from chapter one in turn and detail whether or not they were reached. Then we will discuss future developments of the system, and what new features could be added.

The main aim of this project was to create a peer to peer application for resource sharing. When looking at the project at this level of abstraction the project has been a success. The application we have can be used to share a resource (files) in a peer to peer environment. When looking at the project through the aims specified in chapter 1, the success is put into perspective.

7.1 Study at least one existing peer to peer file sharing application

This objective was completed successfully. Two of the most popular peer to peer file sharing applications were studied in great detail. The study of both of these applications (i.e. Napster and Gnutella) gave us insight into the working of pure peer to peer system and client server peer to peer systems. Also many advantages and disadvantages were identified, which proved very valuable when developing our own application.

7.2 Specify a peer to peer protocol for resource sharing

This objective was completed successfully. Our protocol allows for peers to be queried about shared resources, and for results to be returned. The protocol can also be seen as a success when looking at the issue of efficiency. The main header of the protocol is 9 bytes in length, less than half the size of some file sharing applications. This means that a network will be able to carry a higher number of messages at any one point, helping scalability issues.

7.3 Create an application to share resources (files) between machines

Like the protocol objective, this object too has been met. Our application allows for the user to specify which files they wish to share, where they wish to download the files to, and who their peers are. The user can also search for and download files, the main part of this objective.

7.4 Advantages of Shadow Protocol

In February of 2001 Jordan Ritter published a paper by the name of "Why Gnutella Can't Scale. No, Really"[30]. This is a mathematical research paper in which Jordan Ritter has determined the amount of bandwidth that is generated by a Gnutella network. Suppose the Gnutella network is organized in such a way that it is well balanced and each node is connect to 8 other nodes and the query message has a TTL of 8 for all

nodes then the amount of traffic that will be generated for one node to issue a query of 18 bytes is approximately 6.3 GB. Now if the node is performing 10 queries per second which is the average query per second during rush hour then the Gnutella network will need to transfer data at a rate of 507.2 Gbps (63.4 GBps) (See Appendix B for calculations).

Since the Shadow protocol is an optimized version of the Gnutella protocol Jordan Ritter formula for calculating bandwidth can also be applied to the Shadow protocol with slight modification. Now if the Shadow network is arranged in the same way as the Gnutella network described previously and one node issues a query of 18 bytes then approximately 5.2 GB traffic is generated. If the node performs 10 queries per second then the bandwidth required will be 420 Gbps (52.5 GBps) (See Appendix B for calculations). Which means that the Shadow network incurred approximately 89.6 Gbps (11.2 GBps) less overhead then Gnutella performing the same query under same circumstances.

The structure of the Shadow network is decentralized, which has the advantage that there is no one point of failure. This makes the Shadow network very stable. Also when performing a search in the Shadow network it is not possible to determine exactly which node initiated the query. Therefore the shadow network is anonymous. The Shadow protocol currently supports file sharing but has the capability to be extended to other forms of peer-to-peer computing.

7.5 Scope for future work

There are many areas of this project that could be improved should further work be done. Time could be spent specifying further message types to allow for the sharing of such resources as disk space, and CPU cycles. Also, messages could be developed to request the 'pushing' of files from behind firewalls in the same way that the Napster and Gnutella protocols feature.

Many improvements to the application could be made as well. Caching, both of query hits and, in the case of file sharing the files themselves could increase the efficiency of the system, reducing the number of messages on the network at any one time. As a whole, the efficiency of the application could be improved by using a more efficient programming language such as C or C++. And many of the searching algorithms that currently just step through the options one at a time could be replaced with more efficient algorithms.

References and Bibliography

References and Bibliography

[1] Enterprise P2P: Flexibility and ROI
Christine Axton
http://www.zdnet.com/enterprise

[2] Symantec Security Response- vbs.gnutella
http://securityresponse.symantec.com/avcenter/venc/data/vbs.gnutella.html

[3] Symantec Security Response- w32.gnuman.worm
http://securityresponse.symantec.com/avcenter/venc/data/w32.gnuman.worm.html

[4] Malicious Threats of Peer-to-Peer Networking
Eric Chien
http://securityresponse.symantec.com/avcenter/reference/p2pnetworking.pdf

[5] Napster Web site
www.napster.com

[6] Napster timeline
http://www.personal.psu.edu/users/j/i/jid102/timeline.html

[7] Napster timeline
http://www.idg.net/english/crd_napster_497926.html

[8] History of Napster
http://holly.colostate.edu/~cass859/allbegan.html

[9] Napster Face Liquidation
http://news.bbc.co.uk/1/hi/business/2234947.stm

[10] Napster Protocol
http://opennap.sourceforge.net/napster.txt

[11] MD5 Homepage (unofficial)
http://userpages.umbc.edu/~mabzug1/cs/md5/md5.html

[12] LimeWire: What is Gnutella?
http://www.limewire.com/index.jsp/what_gnut

[13] LimeWire: DEVELOPER RESOURCES
http://www.limewire.com/index.jsp/developer

[14] Gnutella Protocol
http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf

[15] Seti@home
http://setiathome.ssl.berkeley.edu/

[17] Graph Layout Example Source code

http://java.sun.com/applets/jdk/1.0/demo/GraphLayout/Graph.java

[18] The Napster Protocol, David Weekly's protocol page, February 23, 2000
http://david.weekly.org/code/napster.php3

[19] Napster makes Internet history, Business Journal, September 11, 2000
http://sanjose.bcentral.com/sanjose/stories/2000/09/11/daily8.html

[20] Gnutella girds against spam attacks, John Borland, Staff Writer, CNET News.com, August 10, 2000
http://news.cnet.com/news/0-1005-200-2489605.html?tag=st.ne.1002.tgif.ni

[21] Gnutella viruses weaker than email bugs, John Borland, Staff Writer, CNET News.com, June 5, 2000
http://news.cnet.com/news/0-1005-200-2020899.html?tag=rltdnws

[22] Gnutella: Unstoppable by Design, Jerome Kuptz, a programmer who contributes to the Gnutella protocol, October 10, 2000
http://www.wirednews.com/wired/archive/8.10/architecture.html

[23] Bandwidth Barriers to Gnutella Network Scalability, Clip2 DSS, September 8, 2000
http://dss.clip2.com/dss_barrier.html

[24] Gnutella: To the Bandwidth Barrier and Beyond, Clip2 DSS, November 6, 2000
http://dss.clip2.com/gnutella.html#q3

[25] *Free Riding on Gnutella by Eytan Adar and Bernardo A. Huberman, October 10, 2000*
http://firstmonday.org/issues/issue5_10/adar/index.html

[26] The Freenet Project
http://freenet.sourceforge.net/

[27] *LimeWire: Running on the Gnutella Network*
http://www.limewire.com/

[29] SourceForge.net: Project Info - Gnucleus
http://sourceforge.net/projects/gnucleus/

[30] "Why Gnutella Can't Scale. No, Really." By Jordan Ritter
http://www.darkridge.com/~jpr5/doc/gnutella.html

Appendix A

Appendix A

Napster protocol Messages

The following section describes the format of the <data> section for each specific message type. Each message to/from the server is in the form of:

<length><type><data>

Where <length> and <type> are 2 bytes each. <length> specifies the length in bytes of the <data> portion of the message. Each field is denoted with <>. The fields in a message are separated by a single space character (ASCII 32). Where appropriate, examples of the <data> section for each message are given. The number on the left represent the message type code (e.g. 0 is for error messages, sent by the server). <type> can be one of these codes (converted to big-endian).

0 *error message* [SERVER]

Format: <message>

2 login [CLIENT]

Format: <nick> <password> <port> "<client-info>" <link-type> [<num>]

<port> is the port the client is listening on for data transfer. If this value is 0, it means that the client is behind a firewall and can only push files outward. It is expected that requests for downloads be made using the 500 message
<client-info> is a string containing the client version info
<link-type> is an integer indicating the client's bandwidth
 0 unknown
 1 14.4 kbps
 2 28.8 kbps
 3 33.6 kbps
 4 56.7 kbps
 5 64K ISDN
 6 128K ISDN
 7 Cable
 8 DSL
 9 T1
 10 T3 or greater
<num> build number for the windows client [optional]

Example:

foo badpass 6699 "nap v0.8" 3

5 "auto-upgrade" [SERVER]

Format: <version> <hostname:filename>

Napster is out of date, get a new version. Or also known as gaping security hole.

<version> = string, the new version number.
<hostname> = string, hostname of upgrade (http) server
<filename> = string, filename

Connections are always made to port 80.

The HTTP Request:

GET <filename> HTTP/1.0
Connection: Keep-Alive
Host: <hostname>

Expected HTTP Response.

Content-Length: <size>
Content-Type: <doesn't matter> <data>

Upgrade file is saved as "upgrade.exe".
And executed as: upgrade.exe UPGRADE "<current.exe>"

No confirmation is requested by Napster when it receives this message. And immediately start to "auto-upgrade". To keep users informed that Napster is doing something potentially very harmful to their computer it displays a message saying its "auto-upgrading".

100 (0x64) client notification of shared file [CLIENT]

Format: "<filename>" <md5> <size> <bitrate> <frequency> <time>

<md5> see section "MD5"
<size> is bytes
<bitrate> is kbps
<frequency> is Hz
<time> is seconds

Example:

"generic band - generic song.mp3" b92870e0d41bc8e698cf2f0a1ddfeac7 443332 128 44100 60

200 (0xc8) client search request [CLIENT]

Format: [FILENAME CONTAINS "artist name"] MAX_RESULTS <max> [FILENAME CONTAINS "song"] [LINESPEED <compare> <link-type>] [BITRATE <compare> "<br>"] [FREQ <compare> "<freq>"] [WMA-FILE] [LOCAL_ONLY]

The artist name and the song name are, obviously, treated the same by the server.

<max. is a number; if it is greater than 100, the server will only return 100 results.

<compare> is one of "AT LEAST" "AT BEST" "EQUAL TO"
<link-type> see 0x02 (client login) for a description

 is a number, in kbps
<freq> is a sample frequency, in Hz
LOCAL_ONLY causes the server to only search for files from users on the same server rather than all linked servers.

The windows client filters by ping time inside the client. It pretty much has to, and it's easy to see the result by setting ping time to at best 100 ms or so, and max search terms to 50. You'll get back like 3 results, but the client will still tell you that it found "50 results".

Examples:
FILENAME CONTAINS "Sneaker Pimps" MAX_RESULTS 75 FILENAME CONTAINS "tesko suicide" BITRATE "AT LEAST" "128"
MAX_RESULTS 100 FILENAME CONTAINS "Ventolin" LINESPEED "EQUAL TO" 10

201 (0xc9) search response [SERVER]

Format: "<filename>" <md5> <size> <bitrate> <frequency> <length> <nick> <ip> <link-type> [weight]

<md5> see secton "MD5"
<size> is file size in bytes
<bitrate> is mp3 bit rate in kbps
<frequency> is sample rate in hz
<length> is the play length of the mp3 in seconds
<nick> the person sharing the file
<ip> is an unsigned long integer representing the ip address of the user with this file
<link-type> see message client login (2) message for a description.
[weight] a weighting factor to allow the client to sort the list of results. Positive values indicate a "better" match; negative values indicate a "worse" match. If not present, the weight should be considered to be 0.

Example:

"random band - random song.mp3" 7d733c1e7419674744768db71bff8bcd 2558199 128 44100 159 lefty 3437166285 4

203 (0xcb) download request [CLIENT]

Format: <nick> "<filename>"

Client requests to download <filename> from <nick>. Client expects to make an outgoing connection to <nick> on their specified data port.

Example:

mred "C:\Program Files\Napster\generic cowboy song.mp3"

SEE ALSO: 500 alternate download request

204 (0xcc) download ack [SERVER]

Format: <nick> <ip> <port> "<filename>" <md5> <linespeed>

Server sends this message in response to a 203 request.

<nick> is the user who has the file
<ip> is an unsigned long integer representing the ip address
<port> is the port <nick> is listening on
<filename> is the file to retrieve
<md5> is the md5 sum
<linespeed> is the user's connection speed (see login(2))

Example:

lefty 4877911892 6699 "generic band - generic song.mp3" 10fe9e623b1962da85eea61df7ac1f69 3

205 (0xcd) private message to/from another user [CLIENT, SERVER]

Format: <nick> <message>

The same type is used for a client sending a msg or recieving one

206 (0xce) get error [SERVER]

Format: <nick> "<filename>"

the server sends this message when the file that the user has requested to download is unavailable (such as the user is not logged in).

212 (0xd4) browse response [SERVER]

Format: <nick> "<filename>" <md5> <size> <bitrate> <frequency> <time>

<nick> is the user contributing the file
<filename> is the mp3 file contributed
<md5> is the has of the mp3 file
<size> is the file size in bytes
<bitrate> is the mp3 bitrate in kbps
<frequence> is the sampling frequency in Hz
<time> is the play time in seconds

Example:

foouser "generic band - generic song.mp3" b92870e0d41bc8e698cf2f0a1ddfeac7 443332 128 44100 60

218 (0xda) downloading file [CLIENT]

No body.

Client sends this message to the server to indicate they are in the process of downloading a file. This adds 1 to the download count which the server maintains.

219 (0xdb) download complete [CLIENT]

No body.

Client sends this message to the server to indicate they have completed the file for which a prior 218 message was sent. This subtracts one from the download count the server maintains

500 (0x1f4) alternate download request [CLIENT]

Format: <nick> "<filename>"

Requests that <nick> make an outgoing connection to the requester's client and send <filename>. This message is for use when the person sharing the file can only make an outgoing tcp connection because of firewalls blocking incoming messages. This message should be used to request files from users who have specified their data port as 0 in their login message

501 (0x1f5) alternate download ack [SERVER]

Format: <nick> <ip> <port> "<filename>" <md5> <speed>

This message is sent to the up loader when their data port is set to 0 to indicate they are behind a firewall and need to push all data outward. The up loader is responsible for connecting to the downloader to transfer the file.

603 (0x25b) whois request [CLIENT]

Format: <nick>

604 (0x25c) whois response [SERVER]

Format: <nick> "<user-level>" <time> "<channels>" "<status>" <shared> <downloads> <uploads> <link-type> "<client-info>" [<total downloads> <total_uploads> <ip> <connecting port> <data port> <email>]

<user-level> is one of "User", "Moderator", "Admin" or "Elite"
<time> is seconds this user has been connected

<channels> is the list of channels the client is a member of, each separated by a space (ASCII 32)
<status> is one of "Active", "Inactive" (offline) or "Remote" (on a different server)
<shared> is number of files user has available for download
<downloads> is the current number of downloads in progress
<uploads> is the current number of uploads in progress
<link-type> see 0x02 (client login) above
<client-info> see 0x02 (client login) above

The following fields are displayed for user level moderator and above:

<total uploads>
<total downloads>
<ip> note: can be "unavailable"
<connecting port>
<data port>
<email> note: can be "unavailable"

Example:

lefty "User" 1203 "80's " "Active" 0 0 0 3 "nap v0.8"

605 (0x25d) whowas response [SERVER]

Format: <user> <level> <last-seen>

if the user listed in a 603 request is not currently online, the server sends this message.

<user> is the user for which information was requested
<level> is the user's last known user level (user/mod/admin)
<last-seen> is the last time at which this user was seen, measured as seconds since 12:00am on January 1, 1970 (UNIX time_t).

619 (0x26b) queue limit [CLIENT]

Format: <nick> "<filename>" <n>

A client may limit the number of downloads from a particular client. Once the limit for a particular user has been reached, the uploading client can send this message to notify the downloader that they have hit the limit and can't have any more simultaneous downloads. *<nick> is the user who hit the limit, <filename> is the file they were trying to download* when they hit the limit, and <n> is the number of simultaneous downloads allowed.

Example:

joebob "C:\MP3\Generic Band - Generic Song.mp3" 3

620 (0x26c) queue limit [SERVER]
Format: <nick> "<filename>" <filesize> <digit>

This message is sent by the server in response to the 619 client request, when one user needs to notify another that they have reached the maximum allowed simultaneous downloads. When the server receives the 619 request from the uploading client, it sends the 620 message to the downloading client. The only difference in format is the addition of the <nick> field in the 620 message which specifies the username of the uploading agent which is notifying the downloader that the queue is full.

Example:

joebob "C:\MP3\Generic Band - Generic Song.mp3" 1234567 3

640 direct browse request [CLIENT, SERVER]

Client: <nick>
Server: <nick> [ip port]

Client: request files for <nick>
Server: <nick> is requesting your files. Optionally, <ip> and <port> are given if the client getting browsed is firewalled.

This message is sent to initiate a direct client to client browsing of shared files.

641 direct browse accept [CLIENT, SERVER]

Client: <nick>
Server: <nick> <ip> <port>

The client to be browsed sends this message back to the server to indicate it is willing to accept a direct browse from <nick>. The server sends this message to the requestor of the browse to indicate where it should connect in order to do a direct browse from <nick>.

748 login attempt [SERVER]

The server sends this message to a logged in client when another client attempts to log in with the same nickname.

753 (0x2f1) change password for another user [CLIENT]

Format: <user> <password> "<reason>"

Allows an administrator to change the password for another user.

821 (0x335) redirect client to another server [CLIENT, SERVER]

Client: <user> <server> <port>
Server: <server> <port>

This command allows an administrator to redirect clients to another server.

Appendix B

Appendix B

Bandwidth Usage Calculations for Gnutella and Shadow Protocols

P	The number of users connected to the GnutellaNet.
N	*The number of connections held open to other servents in the network. In* the default configuration of the original Gnutella client, this is 4.
T	Our TTL, or Time To Live, on packets. TTL's are used to age a packet and ensure that it is relayed a finite number of times before being discarded.
B	The amount of available bandwidth, or alternatively, the maximum capacity of the network transport.
f(n, x, y)	A function describing the maximum number of reachable users that are at least ***x*** hops away, but no more than ***y*** hops away. ***f(n, x, y) = Sum[((n-1)^(t-1))*n, t = x->y]***
G(n, t)	A function describing the maximum number of reachable users for any given ***n*** and ***t***. ***g(n, t) = f(n, 1, t)***
H(n, t, s)	A function describing the maximum amount of bandwidth *generated* by relaying a transmission of ***s*** bytes given any ***n*** and ***t***. *Generation* is defined as the formulation and outbound delivery of data. ***h(n, t, s) = n*s + f(n, 1, t-1)*(n-1)*s***
i(n, t, s)	A function describing the maximum amount of bandwidth *incurred* by relaying transmission of ***s*** bytes given any ***n*** and ***t***. *Incurrence* is defined as the reception or transmission of data across a unique connection to a network. ***i(n, t, s) = (1 + f(n, 1, t-1))*n*s + f(n, t, t)*s***
A	Mean percentage of users who typically share content.
B	Mean percentage of users who typically have responses to search queries.
R	Mean number of search responses the typical respondent offers.
L	Mean length of search responses the typical respondent offers.
R	A function representing the Response Factor, a constant value that describes the product of the percentage of users responding and the amount of data generated by each user. R = (a*b) * (88 + r*(10 + l))
j(n, T, R)	A function describing the amount of data generated in response to a search query by tier T, given any n and Response Factor R. j(n, T, R) = f(n, T, T) * R
k(n, T, R)	A function decsribing the maximum amount of bandwidth generated in response to a search query, including relayed data, given any n and t and Response Factor R. k(n, t, R) = Sum[j(n, T, R) * T, T = 1->t]

Table B.1 Variables and Equations

IP header	20 bytes
TCP header	20 bytes
Gnutella header	23 bytes
Minimum Speed	1 byte
Search string	18 bytes + 1 byte *(trailing null)*
Total	83 bytes

Table B.2 Gnutella Search Query Packet Makeup

IP header	20 bytes
TCP header	20 bytes
Gnutella header	23 bytes
Number of hits	1 byte
Port	1 byte
IP Address	4 byte
Speed	3 byte
Result Set	$r * (8 + l + 2)$ bytes
Servent Identifier	16 bytes
Total	$88 + r*(10 + l)$ *bytes*

Table B.3 Gnutella Search Response Packet Makeup

	T = 1	T = 2	T = 3	T = 4	T = 5	T = 6	T = 7	T = 8
N = 2	355	899	1,633	2,555	3,667	4,968	6,457	8,136
N = 3	533	2,165	6,566	17,635	44,314	106,748	249,773	572,133
N = 4	710	3,976	17,176	66,990	247,069	879,219	3,051,406	10,395,177
N = 5	888	6,330	35,665	183,261	894,685	4,224,529	19,480,490	88,250,692
N = 6	1,065	9,229	64,231	410,161	2,494,411	14,688,661	84,524,911	478,031,161
N = 7	1,243	12,672	105,075	802,470	5,844,688	41,245,083	284,529,996	1,929,534,709
N = 8	1,420	16,659	160,398	1,426,040	12,101,847	99,546,651	800,659,396	6,331,442,417

Table B.4 Bandwidth Generated in Bytes (S=83, R=94.56) Gnutella

IP header	20 bytes
TCP header	*20 bytes*
Gnutella header	23 bytes
Minimum Speed	*1 byte*
Search string	18 bytes + 1 byte *(trailing null)*
Total	83 bytes

Table B.2 Gnutella Search Query Packet Makeup

IP header	20 bytes
TCP header	20 bytes
Gnutella header	23 bytes
Number of hits	1 byte
Port	1 byte
IP Address	4 byte
Speed	3 byte
Result Set	$r * (8 + l + 2)$ bytes
Servent Identifier	16 bytes
Total	$88 + r*(10 + l)$ bytes

Table B.3 Gnutella Search Response Packet Makeup

	T = 1	T = 2	T = 3	T = 4	T = 5	T = 6	T = 7	T = 8
N = 2	355	899	1,633	2,555	3,667	4,968	6,457	8,136
N = 3	533	2,165	6,566	17,635	44,314	106,748	249,773	572,133
N = 4	710	3,976	17,176	66,990	247,069	879,219	3,051,406	10,395,177
N = 5	888	6,330	35,665	183,261	894,685	4,224,529	19,480,490	88,250,692
N = 6	1,065	9,229	64,231	410,161	2,494,411	14,688,661	84,524,911	478,031,161
N = 7	1,243	12,672	105,075	802,470	5,844,688	41,245,083	284,529,996	1,929,534,709
N = 8	1,420	16,659	160,398	1,426,040	12,101,847	99,546,651	800,659,396	6,331,442,417

Table B.4 Bandwidth Generated in Bytes (S=83, R=94.56) Gnutella

	T = 1	T = 2	T = 3	T = 4	T = 5	T = 6	T = 7	T = 8
N = 2	295	747	1,356	2,122	3,044	4,124	5,361	6,755
N = 3	442	1,798	5,451	14,642	36,789	88,617	207,342	474,929
N = 4	590	3,301	14,261	55,617	205,110	729,874	2,533,011	8,628,968
N = 5	737	5,257	29,612	152,146	742,738	3,506,924	16,170,937	73,256,069
N = 6	885	7,664	53,330	340,520	2,070,770	12,193,520	70,164,770	396,808,520
N = 7	1,032	10,523	87,242	666,217	4,852,037	34,238,786	236,190,215	1,601,684,430
N = 8	1,180	13,834	133,174	1,183,907	10,046,478	82,636,583	664,632,068	5,255,653,701

Table B.8 Bandwidth Generated in Bytes (S=69, R=78.48) Shadow

	T = 1	T = 2	T = 3	T = 4	T = 5	T = 6	T = 7	T = 8
N = 2	2,950	7,469	13,558	21,216	30,444	41,242	53,609	67,546
N = 3	4,424	17,982	54,515	146,416	367,888	886,172	2,073,424	4,749,289
N = 4	5,899	33,014	142,613	556,166	2,051,102	7,298,736	25,330,114	86,289,677
N = 5	7,374	52,566	296,118	1,521,462	7,427,382	35,069,238	161,709,366	732,560,694
N = 6	8,849	76,637	533,297	3,405,197	20,707,697	121,935,197	701,647,697	3,968,085,197
N = 7	10,324	105,227	872,416	6,662,166	48,520,374	342,387,856	2,361,902,147	16,016,844,304
N = 8	11,798	138,336	1,331,741	11,839,066	100,464,778	826,365,830	6,646,320,682	52,556,537,011

Table B.9 Bandwidth rates for 10 qps (S=69, R=78.48) Shadow