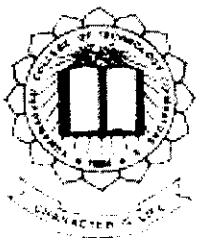
P-1614



i SHARE – A PEER TO PEER FILE SHARING NETWORK

A PROJECT REPORT

Submitted by

RAJIV R LUND 71202104031

RAJKUMAR.N 71202104032

VENKATESAN.P 71202104047

In partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

KUMARAGURU COLLEGE OF TECHNOLOGY,

COIMBATORE-641006

ANNA UNIVERSITY : CHENNAI 600 025

MAY 2006

ANNA UNIVERSITY : CHENNAI 600 025

BONAFIDE CERTIFICATE

Certified that this project "**i-SHARE –A PEER TO PEER FILE SHARING NETWORK**" is the bonafide work of **RAJIV R LUND, RAJKUMAR.N, VENKATESAN.P** who carried out the project work under my supervision.

SIGNATURE

Dr THANGASAMY.S

DEAN

Department of Computer Science & Engg.,

Kumaraguru College of Technology

Coimbatore-641006

SIGNATURE

Mr MOHANAVEL.S

SUPERVISOR

SENIOR LECTURER

Department of Computer Science & Engg.,

Kumaraguru College of Technology

Coimbatore-641006

Submitted for viva-voce examination held on 02.05.2006

Internal Examiner

External Examiner

DECLARATION

We here by declare that the project entitled "i-SHARE –A PEER TO PEER FILE SHARING NETWORK", is a record of original work done by us and to the best of our knowledge, a similar work has not been submitted to Anna University or any other institution for fulfillment of the requirement of the course study

This report is submitted in partial fulfillment of the requirements for the award of the degree of bachelor of computer science and engineering of Anna University, Chennai

Place: Coimbatore

Date: 02.05.2006

(Rajiv R Lund)

(Rajkumar.N)

(Venkatesan.P)

ACKNOWLEDGEMENT

We are extremely grateful to Dr.Padmanaban.K.K, B.Sc.(Engg.), M.Tech., Ph.D., Principal, Kumaraguru College of Technology, Coimbatore for having given us a golden opportunity to embark on this project.

We are deeply obliged to Dr.Thangasamy.S., B.E.(Hons), Ph.D., Dean, Department of Computer Science and Engineering, Kumaraguru College of Technology, Coimbatore for his valuable guidance and useful suggestions.

We are grateful to Prof Mrs.Devaki.P B.E.,M.S., Project Coordinator, Department of Computer Science and Engineering, Kumaraguru College of Technology, Coimbatore for her encouragement and support at various levels of this project work.

We also express our sincere thanks to Mr Mohanavel.S B.E,M.B.A,.Project Guide, Department of Computer Science and Engineering, Kumaraguru College of Technology, Coimbatore, for his valuable guidance and encouragement at every stage of this project.

Most of all, we thank our parents and friends for their blessings and support, without which we would not be able to do anything.

Abstract

The project entitled "i SHARE – A Peer To Peer File Sharing Network" relates to the creation of a Gnutella servent (application that acts as both client and server) capable of connecting to and downloading from the Gnutella network, from both single and multiple sources. The initial code base was a set of Java classes. From there, single and multi source download classes were created and tested to make an attempt to show whether multi sourced downloads provide faster downloads for users while not overly congesting the network.

This project focuses on the development and evaluation of a solution to multi-source downloads for Gnutella. The project investigates different strategies for implementing these multi-source downloads in terms of the number of sources used for any particular download; whether a part-file download should be dropped if it proves to be slow, whether or not the speed of a particular servent proves to be inaccurate etc. Comparative performance analysis is performed on these different strategies to investigate if download rates are increased.

The system itself was written using Sun's Java language. It was developed using the Eclipse IDE Project, which is itself undergoing continual development by IBM and the freeware version of the IBM Eclipse SWT Designer plug-in. This project runs in all machines which has Java run time environment. i-share is able to transfer all types of files like txt, mp3, dat, wmv, avi, exe, zip, rar.

CONTENTS

List of Tables:

List of Figures:

List of Abbreviations:

GUID - Globally Unique Identifier

TTL -Time-To-Live

GDF - Gnutella Developers Forum

INTRODUCTION

CHAPTER - 1

INTRODUCTION

A **peer-to-peer** (or **P2P**) computer network is a network that relies on the computing power and bandwidth of the participants in the network rather than concentrating it in a relatively low number of servers. P2P networks are typically used for connecting nodes. Such networks are useful for many purposes. Sharing content files containing audio, video, data or anything in digital format is very common.

A pure peer-to-peer network does not have the notion of clients or servers, but only equal peer nodes that simultaneously function as both "clients" and "servers" to the other nodes on the network. This model of network arrangement differs from the client-server model where communication is usually to and from a central server. Some networks and channels, such as Napster, OpenNAP, or IRC @find, use a client-server structure for some tasks (e.g., searching) and a peer-to-peer structure for others. Networks such as Gnutella or Freenet use a peer-to-peer structure for all purposes, and are sometimes referred to as true peer-to-peer networks, although Gnutella is greatly facilitated by directory servers that inform peers of the network addresses of other peers.

An important goal in peer-to-peer networks is that all clients provide resources, including bandwidth storage space, and computing power. Thus, as nodes arrive and demand on the system increases, the total capacity of the system also increases. This is not true of a client-server architecture with a fixed set of servers, in which adding more clients could mean slower data transfer for all users. The distributed nature of peer-to-peer networks also increases robustness in case of failures by replicating data over multiple peers, and in pure P2P systems by enabling

peers to find the data without relying on a centralized index server. In the latter case, there is no single point of failure in the system.

This Project therefore details the creation of a Gnutella servent capable of making single and multi-sourced downloads from the Gnutella P2P network. The reasoning behind the project was originally that no current Gnutella servents supported multi-sourced downloading and that no tests had attempted to prove that it was better.

CONTEXT AND REQUIREMENTS

CHAPTER – 2

CONTEXT AND REQUIREMENTS

2.1 Context

Gnutella, has an open specification available online and so anyone can create a Gnutella client. This is probably one of the main reasons for its popularity, as there is a client out there to suit nearly any ones tastes and for many different Operating Systems. Gnutella is therefore an ideal choice for the network to connect to for this project. It is open source, so the specification could be read online and understood, and because of this there are many different servents available to test the servent created for this project against.

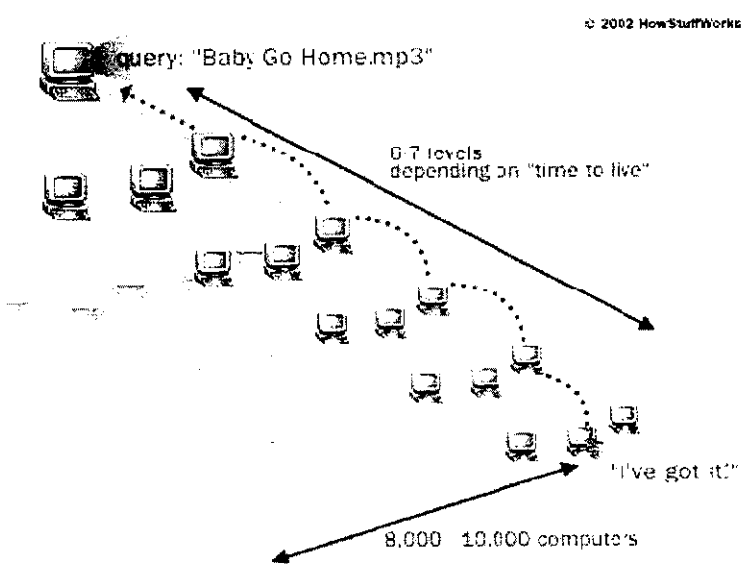


Figure 2.1: How the Gnutella file sharing network works

2.2 Requirements

It has already been explained why the Gnutella network was chosen as the P2P network to be used in this project. The aim of the project is to examine whether or not multi-sourced downloads offer an improved download time over single-sourced downloads, while not overly crippling the host network.

The requirements for this project are therefore as follows:

- To create a servent program capable of using the classes to connect to the Gnutella network.
- To make the servent capable of searching for specific files on the Gnutella network using the classes.
- To make the servent capable of downloading a file from a remote host on the Gnutella network.
- To test whether or not multi-sourced downloads do in fact provide an improved download time, on average, over single-sourced downloads.
- To examine whether the process of downloading a file from multiple sources in parallel causes excessive network congestion above that caused by single-sourced downloads.
- To test whether the servent compares favorably with other well known servents.

Software Requirements:

JAVA, ECLIPSE/SWT.dll.

Hardware Requirements:

Processor speed	- 1 GHz
RAM	- 128 MB
Disk Space	- 1 GB

All systems must be capable of running JVM and have an active internet connection.

2.3 Project schedule

S.no	Task Description	DEC 2005	JANUARY 2006	FEBRUARY 2006	MARCH 2006	APRIL 5th 2006
1	Planning					
2	Design strategy list and break into a sub-list					
3	Implement system adding new functionality					
4	Test system and record details					
5	Develop system by further tests					
6	Finalise software and write the report					

Table 2.1 Project schedule

BACKGROUND

CHAPTER - 3

BACKGROUND

3.1 Overview of Gnutella

The Gnutella network is a fully decentralized, peer-to-peer application layer network that facilitates file sharing and is built around an open protocol developed to enable host discovery, distributed search and file transfer. It consists of the collection of Internet connected hosts on which Gnutella protocol enabled applications are running.

The Gnutella protocol makes possible all host-to-host communication through the use of messages.

GUID	Type	TTL	Hops	Payload Size
16 bytes	1 byte	1 byte	1 byte	4 bytes
←----------23 bytes----------→				

Table 3.1 - IP message format expressed in bytes

A message consists of the following five fields, GUID (Globally Unique Identifier), TTL (Time-To-Live), Hops, and Payload Size. The GUID field provides a unique identifier for a message on the network; the Type field indicates which type of message is being communicated; the TTL field enumerates the maximum number of hops that this message is allowed to traverse; the Hops field provides a count of the hops already traversed; and the Payload Size field provides a byte count of all data expected to follow the message.

A host communicates with its peers by receiving, processing, and forwarding messages, but to do so it must follow a set of rules which help to ensure reasonable message lifetimes. The rules are as follows:

1. Prior to forwarding a message, a host will decrement its TTL field and increment its Hops field. If the TTL field is found to be zero following this action, the message is not forwarded.
2. If a host encounters a message with the same GUID and Type fields as a message already forwarded, the new message is treated as a duplicate and is not forwarded a second time.

At present there are only five types of messages: Ping, Pong, Query, Query-Hit and Push. Ping and Pong messages facilitate host discovery, Query and Query-Hit messages make possible searching the network, and Push messages ease file transfer from fire walled hosts.

Because there is no central index providing a list of hosts connected to the network, a disconnected host must have offline knowledge of a connected host in order to connect to the network. Once connected to the network, a host is always involved in discovering hosts, establishing connections, and propagating messages that it receives from its peers, but it may also initiate any of the following six voluntary activities: searching for content, responding to searches, retrieving content, and distributing content. A host will typically engage in these activities simultaneously.

Typically, a host will search for other hosts and establish connections so as to satisfy a maximum requirement for active connections as specified by the user, or to replace active connections dropped by it or its peer. Consequently, a host tends to always maintain the maximum requirement for active connections as specified by

the user, or the one connection that it needs to remain connected to the network. To engage in host discovery, a host must issue a Ping message to the host of which it has offline knowledge. That host will then forward the ping message across its open connections, and optionally respond to it with a Pong message. Each host that subsequently receives the Ping message will act in a similar manner until the TTL of the message has been exhausted. Furthermore, a Pong message may only be routed along the reverse of the path that carried the Ping message to which it is responding. Lastly, after having discovered a number of other hosts, the host that issued the initial ping message may begin to open further connections to the network.

In searching for content, a host propagates search queries across its active connections. Those queries are then processed and forwarded by its peers. When processing a query, a host will typically apply the query to its local database of content and respond with a set of URLs pointing to the matching files. The propagation of Query and Query-Hit messages is identical to that of Ping and Pong messages. First, a host issues a Query message to the hosts to which it is connected. The hosts receiving that Query message will then forward it across their open connections, and optionally respond with a Query-Hit message.

Each host that subsequently receives the Query message will act in a similar manner until the TTL of the message has been exhausted. Again, as with Pong messages, Query-Hit messages may only be routed along the reverse of the path that carried the Query message to which it is a response. Lastly, the sharing of content on the Gnutella network is accomplished through the use of the HTTP protocol.

Due to the decentralized nature of the architecture, each host participating in the Gnutella network plays a key role in the organization and operation of the network. Both the sharing of host information, as well as the propagation of search requests and responses are the responsibility of all hosts on the network rather than a central

index. As a result, however, the Gnutella network architecture avoids creating the single-point-of-failure that, theoretically, threatens other networks. Consequently, the Gnutella network exhibits a great deal of fault-tolerance, continuing to function even as subsets of hosts become unavailable.

Additionally, each host in the Gnutella network is capable of acting as both a server and client, allowing a user to distribute and retrieve content simultaneously. This spreads the provision of content across many hosts, and helps to eliminate the bottlenecks that typically occur when placing enormous loads on a single host. Additionally, in the Gnutella network, the amount and variety of content available to a user scales with the number of users participating in the network.

3.1.1 The Gnutella Protocol – 0.4

To implement the previously discussed ways of connecting and searching there are certain Gnutella Specification Standards which must be upheld. As previously commented, to connect to the Gnutella network the address of another servent must be known otherwise a connection is not possible. If a remote host's IP Address is known then a TCP connection to this known servent can be attempted by sending the Gnutella specification "Connection String" to the remote host.

```
GNUTELLA CONNECT/0.4\n\n
```

Assuming that the remote host which the connection request was sent to receives it, they are responsible for sending back the Response String:

```
GNUTELLA OK\n\n
```

If however the remote host does not reply with this string then the connection can not be established. Various reasons for this are possible, these being that the remote host's incoming connection slots which are available are full or perhaps that

the remote servent does not support the same version of the Gnutella Protocol as the one the connecting servent does.

Assuming that a successful connection is made between the servent and the remote host, they communicate over the Gnutella network via Gnutella Protocol Descriptor Headers. These descriptors are specific to Gnutella and can be identified by their characteristics.

For example, if the IP Address 10.36.152.157 was to be represented in the IPv4 byte format, then it would be shown as:-

	0x00	0x11	0x32	0x04
Byte offset	byte 0	byte 1	byte 2	byte3

Table 3.2 - IP Address Expressed in Byte Format

3.1.2 Descriptor Headers

The descriptor headers contain the information required to communicate with other servents.

Descriptor ID	Payload Descriptor	TTL	Hops	Payload Length
Byte offset 0 15	16	17	18	19 22

Table 3.3 - Generic Gnutella Descriptor Header

Descriptor ID 16-byte String that identifies the descriptor when being routed.

Payload 0x00 - Ping Message

0x01 - Pong Message

0x40 - Push Message

0x80 - Query Message

0x81 - QueryHit Message

TTL The amount of times the descriptor will be forwarded to other

servents currently connected to the Gnutella network. This value is known because every time the descriptor is passed the TTL field is decremented.

Hops The number of times the descriptor has been forwarded within the network.

Payload Length The length of the descriptor following this header.

Ping 0x00 Messages:

The PING Messages do not have a payload assigned to them and are therefore of length zero. In simple terms this means that any Descriptor that has a size of 0x00 is perceived as being a PING message. As mentioned previously, the PING messages are actively sent out by servents wishing to explore the network for other connected hosts.

If a remote host receives a ping message then it is assumed that the host will respond with a pong message.

Pong 0x01 Messages:

As the size of the payload of a Pong message is not zero, the Pong message itself is made up of separate components. Pong messages are however only sent in response to Ping messages received from other hosts.

This message descriptor is comprised as follows: -

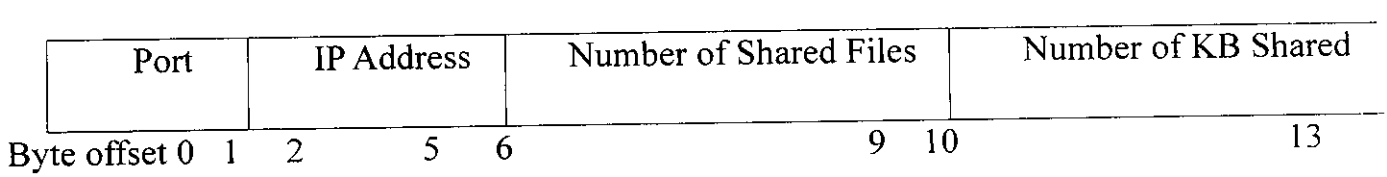


Table 3.4 - Pong 0x00 Message

Port Port accepting incoming connection.

IP Address IP Address of the responding host.

Number of The number of files the remote host is sharing.

Files Shared

Number of KB The number of Kilobytes of data that the remote host Shared

Query 0x80 Messages:

Again as the size of the query message is not zero the headers are made up of separate parts. How these messages are comprised is shown.

	Minimum Speed	Search Criteria
Byte Offset	0 1	2 ...

Table 3.5 - Query 0x80 Message

Minimum Speed The minimum speed in KB/sec of servents that are expected to respond to a particular Search Query.

Search Criteria The search criteria specified by the user for a particular file that they might wish to download.

QueryHit 0x81 Messages:

Again as the size of the query message is not zero the headers are made up of separate parts. How these messages are comprised is shown below.

	Number of Hits	Port	IP Address	Speed	Result Set	Servent Identifier
Byte Offset	0 1	2 3	6 7	10 11	-- x	x+16

Table 3.6 - QueryHit 0x81 Message

Number of Hits Number of query hits that are in the Result Set.

Port The responding hosts incoming connections port

IP Address The IP Address of the responding host.

Speed The speed in KB/sec of the responding host.

Result Set Responses to a given Query. This part of the header is itself split into three sub-sections of which all appertain to the file and are

the File Index, the File Size and the File Name.

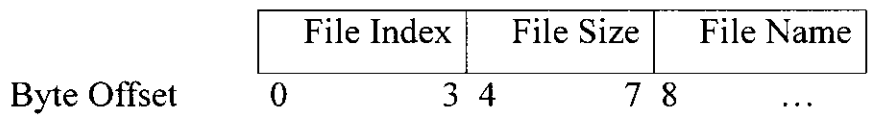


Table 3.7 - QueryHit 0x81 Message Result Set

File Index	Assigned by the remote host
File Size	Size of the file in bytes
File Name	The name of the file
ServentIdentifier	16-byte String that identifies the responding servent on the network.

Push 0x40 Messages:

Again as the size of the push message is not zero the headers are made up of separate parts. How these messages are comprised is shown below.

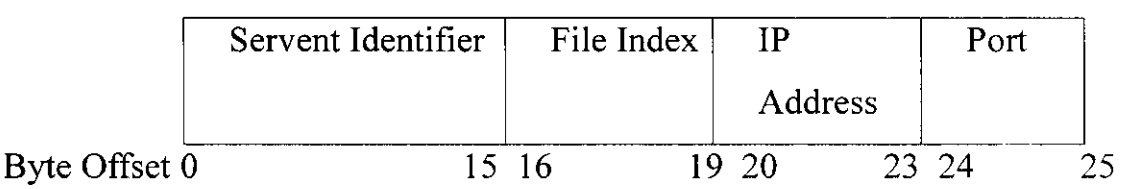


Table 3.8 - Push 0x40 Message

Servent Identifier	16-byte String that identifies the responding servent on the network.
File Index	The index of the file to be received from the remote host.
IP Address	The IP Address of the host that the file should be sent to.
Port	The port of the host that the file should be sent to.

3.1.3 File Downloads

If any particular servent receives a QueryHit Message then this basically means that a search that they have previously conducted has been matched somewhere within the Gnutella network. Although these searches have been conducted using the Gnutella network, downloading of files does not use it. Instead of this, a direct connection between the two servents is made and the servent that is requesting the file becomes the client and the host of the required file is the server. This nicely displays the concept that every servent is a client and a server.

As no file data transfer takes place across the Gnutella network, to be able to receive the required file, the requesting servent sends a HTTP GET "Request String" as follows to the remote servent. The variables<FileIndex> and <File Name> are the same as found within the QueryHit message that the servent has previously received and so the servent does not need to calculate the relevant parts.

GET /get/<*File Index*>/<*File Name*>/HTTP/1.0\r\n
Connection: Keep-Alive\r\n
Range: bytes=0-X\r\n
User-Agent: Ishare 0.8\r\n
\r\n

The remote Servent that receives the request String then responds with a Query Response message with the appropriate HTTP headers included.

HTTP 200 OK\r\n
Server: Gnutella\r\n
Content-type: application/binary\r\n
Content-length: *<range of bytes requested>*\r\n

\r\n

The requested file then gets transmitted and sent to the requesting servent from the remote host who is serving the file.

3.2 The Current Gnutella Protocol – 0.6

3.2.1 Ultrapeers And Leaf Nodes

This system was developed to improve the structure of the Gnutella network from that of the 0.4 version of the protocol.

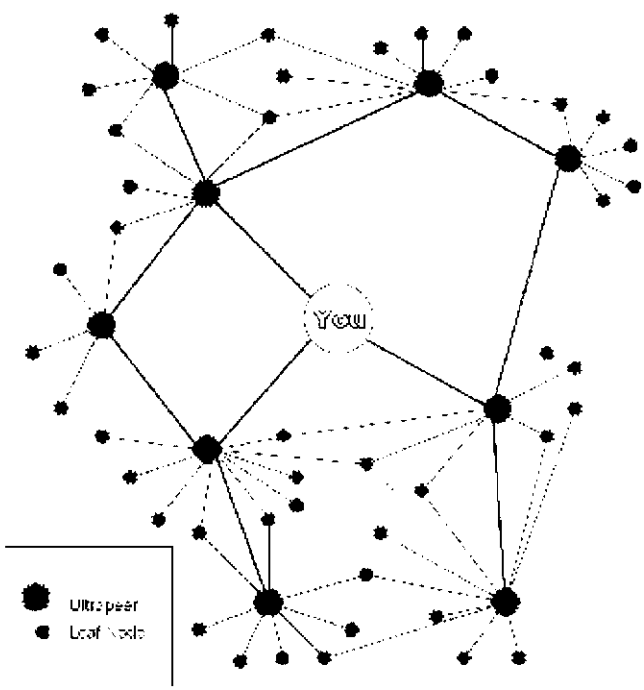


Figure 3.1: A representation of the Gnutella 0.6 network

Gnutella 0.6 specifies two different types of servent that can connect to the Gnutella network: ultrapeers and leaf nodes. Ultrapeers are servents that allow incoming connections, while leaf nodes do not. The analogy is to a tree like structure. However, the Gnutella network is a cross between a tree and a mesh. No des can specify that they are ultrapeers or leaves. A leaf node can only make outgoing connections to ultrapeers, and an ultrapeer can connect to anything using

the network. When a leaf requests a connection to an ultrapeer the ultrapeer accepts or rejects the connection based on how many incoming connections it already has and how many is its limit. Outgoing connections from ultrapeers always connect to other ultrapeers.

It is recommended that only hosts on broadband connections are set up to be ultrapeers. Ultrapeers identify themselves with the 'X-Ultrapeer' header, set to true on either the first or second part of the 3 way handshake, depending on whether it is the ultrapeer attempting the connection or being connected to. Lower speed hosts should be set to be leaf nodes, by either setting the header to false or not listing it at all. Low speed hosts thus avoid the problem of using all their bandwidth to route messages between hosts, as they only have outgoing connections to ultrapeers.

They do still receive queries and such, as the ultrapeer forwards all messages it receives with Time To Live (TTL)> 0 to any connected leaf nodes. This also means that hosts will not receive messages that they sent, as the ultrapeer will not forward a message to a host which is the same as one it received from said host. Ultrapeers connect to each other as well. Obviously if every host on the network was an ultrapeer then the same problem would arise as if the ultrapeer system was not in effect, so the 'X-Ultrapeer-Needed' header is used.

If an ultrapeer attempts a connection with another ultrapeer that already has its full quota of ultrapeer type connections, but can still accept leaf node connections, then the 200 OK response will be sent back with the additional X-Ultrapeer-Needed header, set to false. The ultrapeer attempting the connection can then do one of two things. If it has any connected leaf nodes, then there is no real choice but to search for another ultrapeer that will accept its connection. If it has no connected leaf nodes, however, it can change to being a leaf type node itself, by

sending its own 200 OK message, with the additional header of 'X- Ultrapeer', once again set to false. So a connection is established but the connecting ultrapeer now functions as a leaf node and can no longer accept incoming connections. In this way the number of ultrapeers in any given part of the Gnutella network is kept to approximately the correct value.

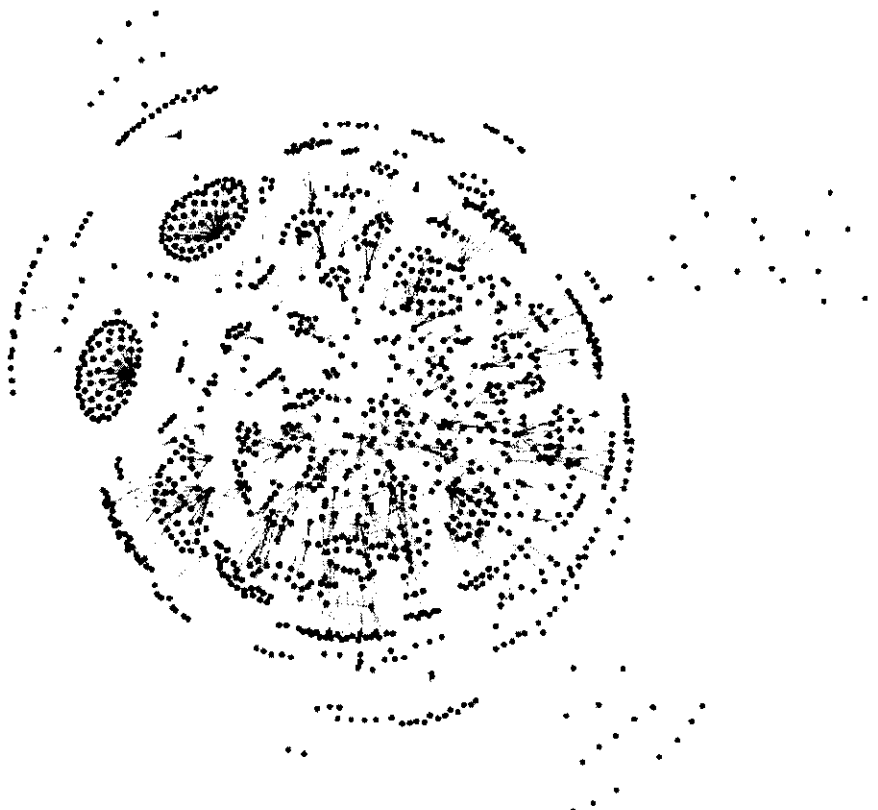


Figure 3.2 Ultra Peer Network Structure

3.2.2 Bootstrapping

Before a Gnutella servent can connect to the Gnutella network it must know the location of other hosts to attempt a connection with. On a local network this could be achieved using broadcast messages. However this would quickly slow the network to a crawl if many servents all tried to connect at once, and over the Internet it is simply not possible or desirable to broadcast.

The method of bootstrapping with Gnutella0.4 involved well known, semi-permanent Gnutella hosts with limited functionality. All they were capable of was sending out a list of current hosts to any new hosts which attempted to connect. Most well known servent developers hosted several of their own HostCaches, which their servent would attempt to connect to when it started. Most servents also allowed users to enter other HostCaches to attempt a connection to, in case the hardcoded ones were down. In Gnutella 0.6 this was changed for a totally different system using GWebCaches.

As commented above the way in which servents connect to the Gnutella network has changed since version 0.4 of the specification. Host-Caches are no longer used since the release of version 0.6 of the protocol. Instead of this a new concept known as a GWebCache was introduced.Currently for any particular host to connect to the Gnutella network, it needs to be capable of finding and storing remote hosts' IP Addresses. There are currently four ways by which a servent may obtain hosts' addresses, and these are:

- Referencing a known Gnutella GWebCache.
- Storing hosts' addresses which have been read from X-Try and X-Try-Ultrapeer headers, (these are received as an addition to the new protocol during connection handshakes)
- Storing hosts' addresses in pong messages which have been received from other hosts during connection attempts. (Again this is an addition to the headers for version 0.6 of the specification)
- Storing hosts' addresses that have been read from QueryHit messages received from Querying the network

Simply explained, GWebCaches are merely scripts that run on web servers. Most GWebCaches are written in Perl or PHP. They are similar in their functionality because they provide basically the same service as a Gnutella specification 0.4 HostCache. The only difference is that because they are basically a web server, they are accessed on port 80 rather that the Gnutella conventional port 6346. As these GWebCaches are often simple PHP scripts, then they are much more lightweight than running a different adaptation of a servent. Therefore the web server that they are running on does not have as much work to do.

As before, a GWebCache still accepts the standard HTTP connections which are used in version 0.4 of the specification, but have the added functionality of containing the request data in the URL. To illustrate this, if a client wishes to receive a list of Gnutella nodes then it would send out one of the following requests depending on what the nature of the request is:

URL?hostfile=1

A client should send a hostfile request whenever it needs information on hosts to connect to or a client should send a urlfile request to build its internal list of caches i.e. when initially connecting to the network.

URL?urlfile=1

As mentioned above, if a **hostfile** is requested successfully, then a list of approximately 20 currently active connected Gnutella hosts is returned to the servent. Naturally, as each GWebCache implementation is different, such as each Gnutella servent implementation is different, the specific number of returned results is specific to each individual GWebCache. The reason that the accepted number of results to return is 20, is due to the fact that this is what is suggested.

The format of these results is also specified and is as follows:

ip:port<CR>

where the "ip" must be in the form of xxx.xxx.xxx.xxx, and the "port" is that of the machine listening for incoming connections.

If however a urlfile request is issued, then this again returns a list of approximately 20 URL's of other known GWebCaches. This list is only useful if the servent wishes to update its locally stored list of GWebCaches that it knows about. The reason that servents might wish to do this is, that sometimes depending on the age of these GWebCaches, they might be stale or even have been taken offline. Naturally the servent does not know this unless it tries to connect to them and fails if this is found to be the case then another GWebCache updates the servent by saying the particular GWebCache is no longer available to use. This is not as uncommon as it might seem, some GWebCaches are taken offline frequently and so it is important to update the locally stored list of GWebCaches.

In response to these various requests, the GWebCache may send back a response of either: -

- a list of actively connected Gnutella nodes in the format "ip:port", or
- a redirect, i.e. HTTP code 3xx response which indicates that the client needs to resend another HTTP GET request for the file, or finally the string "ERROR", which is often preceded by more specific error information regarding the specific request.

It is also possible that the Servent will request GWebCache updates. In this scenario, the servent connects to the GWebCache and issues it a request in the

structure of:

URL?ip=xxx.xxx.xxx.xxx:PORT

Again, where xxx.xxx.xxx.xxx is replaced by the IP Address of the particular servent and PORT is replaced with the port number of the machine that the servent is listening for incoming connections on.

The connection attempt does not however have to be successful and this is not always the case. If the connection is successful, the GWebCache will send a response of "OK", but if not then a response of either "ERROR" or "ERROR: *message*" will be sent back to the servent. (Where *message* is the message sent by the GWebCache to say what went wrong during the connection)

Other operations such as a standard Ping message can be sent to the GWebCache to see if it actually online or not, but these are not important or rudimentary for the operation of connecting to the GWebCache, and so have not been detailed.

3.2.3 Handshaking

Once a list of hosts has been retrieved, connection attempts must be made on each host. Servents have an internal value relating to the number of incoming and outgoing connections to make. In some implementations this may change based on the connection speed of the servent. In Ishare it is fixed at a maximum of 3 incoming and 3 outgoing connections, regardless of connection speed. For a servent to send and receive queries it must have at least one outgoing connection to the network, but does not necessarily require any incoming ones.

Connections are made by handshaking with the host the servent is attempting

a connection to. Gnutella handshaking requests are HTTP like, as can be seen in the later examples, but do not conform to HTTP header types in the data that they actually send and receive.

3-Way Connection Handshaking:

Whenever a list of potential Gnutella connected hosts is received from a GWebCache, then connection attempts can be made on these given hosts. Depending on the value that the individual servent has with regard to active incoming and outgoing connection slots, then the connections are established. By this, one means that if the value is 5, then a maximum of 5 incoming and 5 outgoing connections can be established. Once the servent reaches this value, no further connections can be established. It is often the case that this number of connections is based on Host Speed. Therefore a host with a larger speed is likely to accept more connections than a smaller host due to upstream and downstream bandwidth limitations. Despite this value, any given servent must have at least one active outgoing connection to another host within the gnutella network. The number of incoming connections is inconsequential, as the servent is only dependant on outgoing connections to other hosts on the network.

Connections to and from the Gnutella network in version 0.6 of the Gnutella specification are done via handshaking. This was also the case with version 0.4 however there have been some developments in the way that the connections are conducted. The connection headers are still in the HTTP format but now they do not conform to the HTTP specification as they did in Gnutella 0.4.

The new handshaking of connections that is done is 3-way and no longer simple 2-way. Now, if any particular servent which supports Gnutella 0.6 wants to connect to another Gnutella host then it issues the following connection String

followed by other headers specific to the individual servent:

GNUTELLA CONNECT/0.6<CR><LF>

As can be seen, the connection String is similar to the 0.4 connection String, except the protocol has changed to 0.6 within the String and as within HTTP, each line of the connection request String is separated by a <CR><LF>.

In addition to this the end of each connection request consists of another <CR><LF>. This is so that the end of the connection String can be identified by the different servents when they parse this request. Therefore a complete connection request could look like:

GNUTELLA CONNECT/0.6<CR><LF>
***Servent Specific Headers* <CR><LF>**
<CR><LF>

As can be seen the initial connection request String is sent, followed <CR><LF> and then any servent specific headers followed by <CR><LF> and then another <CR><LF>.

The Servent Specific Headers comprise of Gnutella 0.6 Protocol specified headers. These are a made up of Mandatory, Recommended and Optional Headers, all of which are supported by all servents who follow the 0.6 Specification. Having said this, if a servent reads a connection header that it does not recognise (and therefore does not support it) it should simple ignore this header.

When received by the servent these connection response headers are displayed in the format:

GNUTELLA/0.6 200 OK<CR><LF>

or

GNUTELLA/0.6 503 Busy<CR><LF>

As mentioned above, these return codes are general web-based codes and mean the same.

200(OK) Code 200 is returned when a request is successful and information is returned. This is definitely the most common code returned on the web.

503(Service Unavailable) Code 503 is returned when the server cannot respond due to temporary overloading or maintenance. Examples of these errors could be that some hosts/servents have limited accounts which can only handle so many requests per day or bytes send per period of time. More specifically to Gnutella, it is likely that the servent has reached its maximum connection slot value.

Handshake Headers Explained:

There are many different headers which are supported by the Gnutella 0.6 specification. There are however many generic ones which feature amongst many commonly used servents. These are: User-Agent, Remote-IP, X-Try and Pong-Caching. The other connection headers are used but one is not as likely to find them as these more commonly used headers.

User-Agent:

As mentioned above the User-Agent connection header is used to describe the

particular servent implementation. There is no set definition for the user-agent but the assumed form is of Servent Name and the Version Number. This being true, the following describes the Servent Name as being Ishare and the Version Number as 0.8. As with the other headers the line delimiters are again, <CR><LF>.

User-Agent: Ishare/0.8<CR><LF>

Remote-IP:

The Remote-IP header is necessary for those servents who are sat behind a firewall or NAT scenario. It follows the standard xxx.xxx.xxx.xxx IP dotted-quad format. This header is necessary so that the connecting servent does actually send the request to the correct IP Address and not for example a LAN address. This is important because if the request is sent to the wrong place then the request will be dropped and lost, therefore resulting in no reply from the remote host.

Remote-IP: 10.36.152.194<CR><LF>

X-Try:

X-Try headers provide the servent with information about other known servents so that they can attempt connection on them. This function is very useful for all servents because it does not require a successful connection to obtain the information about other hosts. By this, on means that even if a servent receives a 503 return message then this information about other hosts is contained within the 503 message. This therefore means the unavailable host has given the servent another means of connecting to the network even though itself cannot provide a connection. An example of this header is in the format of X-Try-Ultrapeer and is shown below:

X-Try-Ultrapeers: 127.0.0.1, 10.36.152.193, 10.36.152.197<CR><LF>

Pong-Caching:

The notion of Pong-Caching is whether or not a particular servent supports Pong-Caching. Simply put, Pong-Caching is where a servent receives a Pong

message form a remote host in response to a Ping message that it sent. If the remote host that replied with the Pong is not currently listed in the Host Cache then it is added. The format of the Pong-Caching header is as follows:

Pong-Caching: 0.1<CR><LF>

3.3 The Gnutella Protocol

Gnutella is a protocol which governs the needed operations for distributed search and file distribution. Despite the fact that the Gnutella protocol has support for the traditional client/server search concept, Gnutella distinguishes itself from other P2P models in that it operates with the concept of a decentralized server. As mentioned previously, this means that every client is a server and vice versa meaning every Gnutella client implementation is referred to as a Servent.

This theoretically means that they each have the capability and functionality to provide both the client-side and server-side operations. The client-side functionality is required so that users can send queries to the network and the server-side so search results can be received in response. It is also possible that these implementations have been designed in such a way that they can also accept queries from other servents, decode these requests and check the message content against their locally stored files, and respond with appropriate search response messages (QueryHit).

As a result of the network being decentralized from any standalone server implementations, the Gnutella protocol and network is very highly fault-tolerant. This therefore means that even if a random selection of servents go offline then others will not be halted due to the network having a distributed architecture. This is what makes Gnutella a more popular and desirable P2P network over others. In

addition to this, Gnutella provides great opportunities into various areas of distributed P2P architecture development with its ease of creation of development tools. This therefore is the main reason why it is such a popular and currently developing area of computing.

Despite this, every Servent that wishes to connect to the Gnutella network still needs a way of finding out about currently connected remote hosts which it can connect to. There are central points at which servents can connect to, in order to find out about currently connected Gnutella nodes. These central points from which the servent can connect to enable them to access network connection information are known as Host Caches.

Naturally, it is therefore feasible to say that a Gnutella servent can only connect to the Gnutella network if it knows the global location (IP Address) of another remote host. One example of finding this required information may be that, on a local network a broadcast message could be sent out across the network to discover other actively connected hosts. However, depending upon the great volume of servents attempting this broadcast message, then the network will be slowed and potentially halted due to the capacity of broadcast messages being flooded around the network. Therefore this is not a good approach.

This way of connecting to the network, known as the "bootstrapping" method was used in the Gnutella 0.4 specification and was successful. To be able to connect to the network in this version of the protocol there were "dumb" Gnutella hosts which had very limited functionality. These dumb hosts could be connected to so as to gain access to other connected Gnutella hosts and were known as Host Caches. Being "dumb" meant that all the Host Caches were capable of doing was sending a list of currently connected remote host IP Addresses to any new servent which

attempts to connect to the network.

These Host Caches are in the main hosted and maintained by Gnutella developers who can often be contacted through the Gnutella GDF. These developers are either simply Gnutella enthusiasts or more often developers who work for companies professionally producing Servents.

When a particular servent bootstraps to any given Host Cache, it is provided with known currently connected hosts and then a connection attempt can be made on them. This information is sent in a PING message. It is then used to actively discover any remote hosts which are on the Gnutella network. The servent which receives this PING message is actively expected to respond to the host that the ping came from with at least one but possibly more PONG messages. These PONG messages include the address of the connected Gnutella Servent and other information regarding the amount of data it is willing to host to the network etc.

Once any particular connection to a remote host has been established, the user has the option to send Search requests known as QUERY Messages for files which they would like to download. If a remote servent makes a match to one of these search requests, the remote servent sends back a Search Response Message (known in the Gnutella Specification as a QueryHit Message) to the querying servent, using the same channel which the search request was sent on. When information is received by the searching servent, it can download a file directly from any particular remote host who has a required file. This is possible because all relevant information needed about the remote host for a file to be requested is known.

The only exception to this rule of thumb is if a particular servent is located behind a NAT Gateway or firewall. In this situation the Gnutella Protocol has the

provision of a Push Message. The reason for this message, is that if a particular host is shielded by a firewall or similar and so does not have a globally visible IP Address which can be seen by other servents, then the servent can request that the shielded servent actually connects to back to it, instead of the usual method of the requesting servent connecting to the remote servent.(i.e. a direct connection to the remote host cannot be made if it is behind a firewall).

This concept is seen as very successful because it enables connections to many hosts who do not allow a direct-connection and which otherwise would not be available to be connected to. This is important because many college and university students use and make Gnutella the popular service that it is and so are often located on the campus or workplace network which itself is guaranteed to have a firewall and NAT in place.

3.3.1 Network Connection

GNUTellaConnection:

The GNUTellaConnection class forms the basis of connecting to the Gnutella Network using ishare. There are two distinct methods which are used in this class to actually connect to and disconnect from the network. These are the **start()** method and the **stop()** method. Whenever an instance of this class is created the start() method is what actually calls and instantiates all the other required classes which are associated with connecting. In contrast the stop() method calls the shutdown() method on all the classes that were created by the start() method. Consequently the connection that has been established with the Gnutella Network is dropped.

Connection:

The Connection class is used to instantiate the startIncomingConnection() and startOutgoingConnection() methods. These methods have the functionality which

enables them to undertake all operations associated with the Connection "Handshaking". The only distinction between the two methods is that the startIncomingConnection() method is concerned with the operations of a remote host requesting to connect to this servent and the startOutgoingConnection() method (which is more frequently used) is concerned with the initial outgoing connection attempt to another host. This class "**extends**" the Java native "**Thread**" class and implements the "**Runnable**" interface.

As Isare was developed when Gnutella 0.4 was in operation it has functionality built in to accommodate HostCache connections. These are carried out in exactly the same way as normal (using standard handshaking operations) connections to other remote hosts. Therefore to enable Ishare to distinguish between these different types of connection there are classes that support both connections to Host Caches (HostCacheConnection) and normal remote host connections (NodeConnection).

NodeConnection:

As mentioned above, this class is concerned with connecting to a single remote host and unless there is specific purpose for a single remote connection, it is no longer used due to the development of the GWebCache in Gnutella 0.6. In spite of this the class can still be used when a connection attempt is being made. Naturally as this class is concerned with connecting it "**extends**" the Connection class and this is why the handshaking is conducted in the same way. Once handshaking has completed the NodeConnection class sends a "**PING**" message to the remote host and awaits its "**PONG**". Once the ping message is received the class simply accepts any messages that are sent by the remote host.

HostCacheConnection:

This is the other class that is concerned with connecting and again this class was developed to function within the Gnutella 0.4 specification. This class is no longer used in the manner that it was designed for. The HostCacheConnection was used to actually connect to the remote HostCache dumb servents. Any results that are returned are added to the HostCache class so that connection attempts can be made on them.

HostCache:

The HostCache contains a list of currently connected Gnutella hosts which connection attempts can be made on. This class gains its information from the HostCacheConnection class and every time this class has new data it automatically updates the HostCache with this new information.

ConnectionData:

This class contains all the information about the servent which is needed and is referenced when attempting to connect to a remote host. Information within this class includes, the number of incoming/outgoing connections, the number of files that the servent has to share with other hosts, the vendor code of the servent, the size of the servents shared files, the gateway IP Address of the servent, the IP Address of the Servent, etc. This information is passed to the Connection class as an argument when needed.

ConnectionList:

This class is used to store a list of the remote hosts that the servent is currently connected to. As and when new connections are established the ConnectionList class is updated. It also contains the functionality to remove old connections that have failed or have been dropped for some reason.

ConnectionManager:

This is an "**abstract**" class which "**extends**" the Java native Thread class. It is however unique because even though it extends Thread it does not contain a run() method. The only job that this class has is to provide functionality used by both the IncomingConnectionManager class and the OutgoingConnectionManager class.

IncomingConnectionManager and OutgoingConnectionManager:

As commented above these classes extend the ConnectionManager abstract base class and are basically the same. The only real difference between these classes is that the IncomingConnectionManager is concerned with connections from a remote host and the OutgoingConnectionManager is concerned with connections to a remote host. These classes contain the information regarding maximum number of incoming connections to the servent and outgoing connections to other remote hosts. The classes make sure that these values are not exceeded and also that the meet minimum criterion such as becoming too low. If minimum criterion are not met then connection attempts are conducted on other Gnutella hosts until the criterion are filled once more.

Router:

The Router class is used to basically forward messages and data that have been received from the Gnutella Network. This data may consist of messages such as, PING, PONG and QUERY etc and this class validates the messages and then sends the data to other connected hosts if required. This operates in the same way as other remote hosts passing messages to this servent. Again this class extends the Thread class so that multiple execution of the class can be performed.

KeepAliveThread:

This is a simple class that basically just keeps established connections active. To do this the class simply sends a PING message to a connected remote host if

there has been no activity between the two servents in a given time period.

GWebCache:

This class has basically made the HostCacheConnection class redundant. Instead of having to initially connect to a dumb servent (HostCache) which returns a list of currently connected hosts in Gnutella 0.4, the process is now to connect to a GWebCache to obtain the same information. This class simple returns a list of default GWebCaches which in turn are attempted to be connected to.

GetHostsFromCache:

This process consists of the servent making a connection to one of the GWebCaches returned from the GWebCache class and also sending a specifically formatted string. If a connection is established between the servent and the cache, then the GWebCache sends back data to the servent who must then parse it and add the relevant information to the Ishare HostCache. The whole process was introduced to try to reduce the number of broadcast messages that were congesting the network. As the new information is received the HostCache becomes more populated and therefore has more hosts that it could connect to. This class extends the Thread class to enable it to run alongside other classes within the Ishare implementation. This is so that if the class is needed to be executed, it can be immediately without having to wait, thus making the implementation operate in a more concurrent manner.

3.3.2 Making a Search

The classes that are concerned with making a search to the Gnutella Network and how they contribute to a search are described below.

SearchSession:

The SearchSession class is undoubtedly the most significant class with regard to searching the Gnutella Network for files using Ishare. For every instance of this

class that is instantiated, it relates to specific search which has been sent to the Gnutella Network. When created the SearchSession is passed a class which implements the MessageReceiver interface. As this class implements the MessageReceiver interface it will contain the receiveSearchReply() method which enables it to receive the search reply messages (QueryHit) in response to the search message that was sent out.

This is basically the main class that needs to be understood with regard to searching using Ishare. However there are other classes which are needed in order for successful searches to be carried out. Therefore it is possibly more beneficial to describe how the SearchSession class actually conducts a search rather than repeating descriptions of classes already mentioned which are used during a connection.

When a SearchSession is conducted the run() method calls the Connection.sendAndReceive() method and passes it the Message that is to be sent and the Message Receiver as required arguments. This Connection.send AndReceive() method is conducted on all currently connected remote hosts. This Connection.sendAndReceive() method then calls the Router.routeBack() method and this gets passed both the Message that is to be sent and the MessageReceiver as required arguments.

The MessageReceiver is then added to the Originate Table (so that the connection information can be reused for example if the file is requested from the particular host) using the GUID as the unique identifier for the host. The Message is then simply sent using the Connection.prioritySend() method. As with connection attempts the Router receives the messages relevant to any searches which may have been conducted and forwards them in the relevant direction. If the Router determines

that the message received is a Search ReplyMessage (QueryHit) then a check is done on the servent's GUID to see if it is the same as that in the message and therefore to see if this message is a response to a search that this servent conducted. If this is the case then the Router forwards the message to the MessageReceiver and is then dealt with appropriately.

SearchResponseReceiver:

As Ishare now includes a GUI interface, the Message Receiver which is used is the Search Response Receiver class (this "**extends**" the Message Receiver Adapter abstract class). The Search Response Receiver class was designed so that it would include the functionality which was capable of populating the TableTree within the Search Tab (located in the GUI) with relevant search results that have been received as a response to a query. This class simply receives the relevant information about search results and populates them within the GUI. If there is more than one file returned which has the same SHA1 Hash code then all these items will be grouped together. If all search response items are different then a new item will be created for each.

3.3.3 Downloading

Again it is impractical to discuss all classes which are involved with the downloading of a file in detail and so only the main areas of control will be outlined.

Download Descriptor:

The DownloadDescriptor class is concerned with describing a particular instance of a SearchReplyMessage and an instance of a FileRecord (which is associated with the SearchReplyMessage). Together these contain all the information needed to initiate a download. The class also contains details of a connection to the specified host, but only if that host has been used in the Download already.

Therefore the DownloadDescriptor is important as it contains all the information that is needed to conduct a download on a specific host.

MultiSource Downloader:

The MultiSourceDownloader class is concerned with initiating a download of a particular file from multiple sources on the Gnutella Network. This class has the functionality needed to swarm the network for files which are to be downloaded. As files can potentially be downloaded in several pieces, the MultiSourceDownloader is responsible for reassembling them when the download is completed.

To download these files the MultiSourceDownloader object creates a Section object which in turn acquires a DownloadDescriptor appertaining to the file that is to be downloaded. These objects are then passed to a new Downloader object which monitors the downloading of the particular section.

Downloader:

The Downloader class is a representation of a connection to a remote host on the network which has acknowledged that it has the required file to be downloaded. The Downloader object does not have any functionality for actually carrying out the download, it merely requests a specific section of the file and then streams any downloaded data received from the remote host to a file on disk. The Downloader monitors each individual section download and if for some reason there is an error or the download fails it is dealt with appropriately. Whenever a section is completed the Downloader object informs the MultiSourceDownloader and if the file is completed all the pieces get reassembled.

Section:

The Section class simply describes a specific section of a file that is linked to a single Downloader instance. This class contains the appropriate functionality to

hold the current download status, (e.g. Downloading, Waiting etc) the size of the section, whether the section is actively downloading and the number of bytes downloading etc. As commented previously the Section object is used as an argument in the Downloader class to download a specific section of a file.

DESIGN

CHAPTER - 4

DESIGN

This section shows an overview of the System Design. Displayed below are control diagrams of the connecting, searching, downloading and uploading processes within Ishare.

4.1 Connection

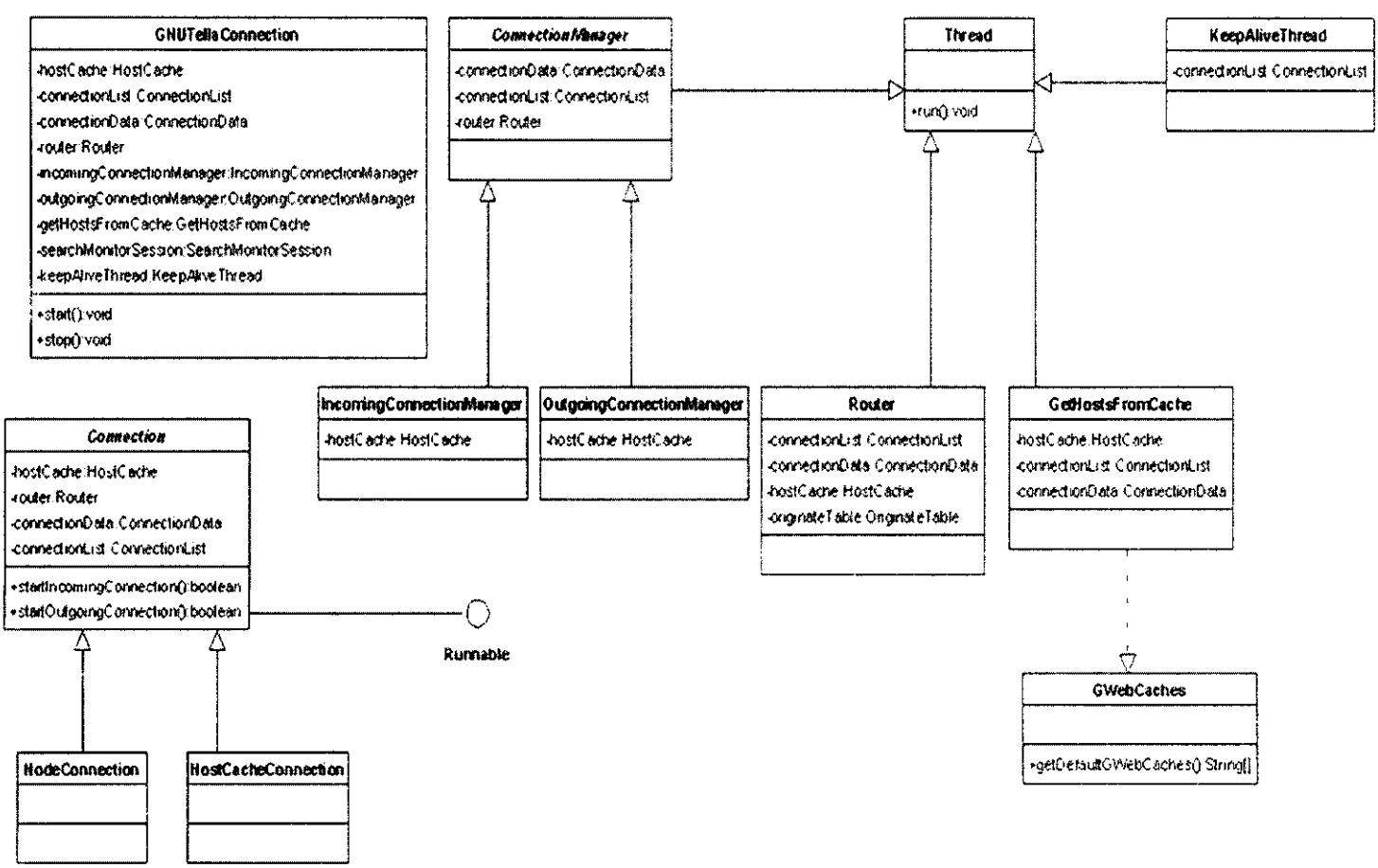


Figure 4.1 Connection System Design

4.2 Searching

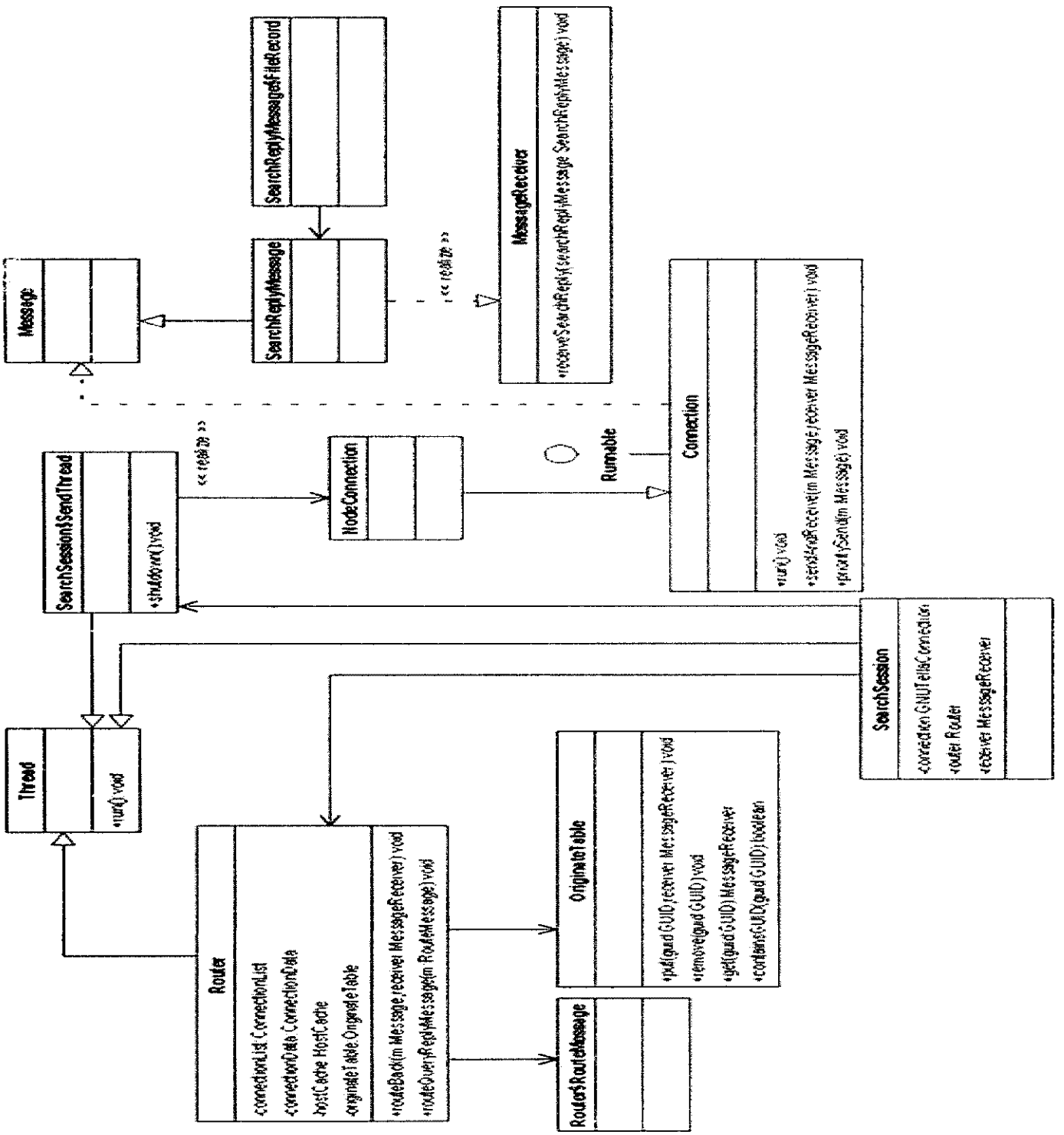


Figure 4.2 Searching System Design

4.3 Downloading

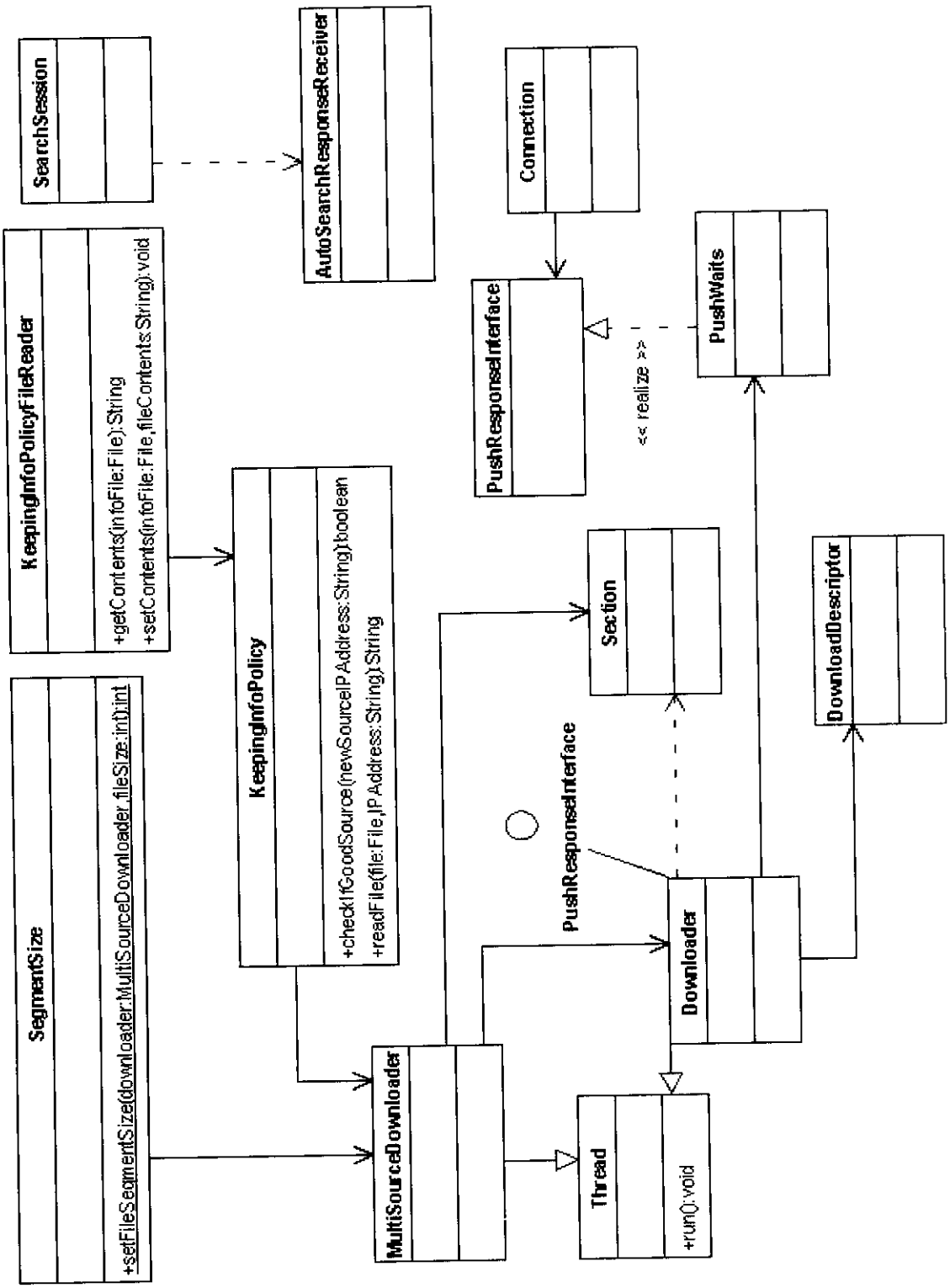


Figure 4.3 Downloading System Design

4.4 Uploading

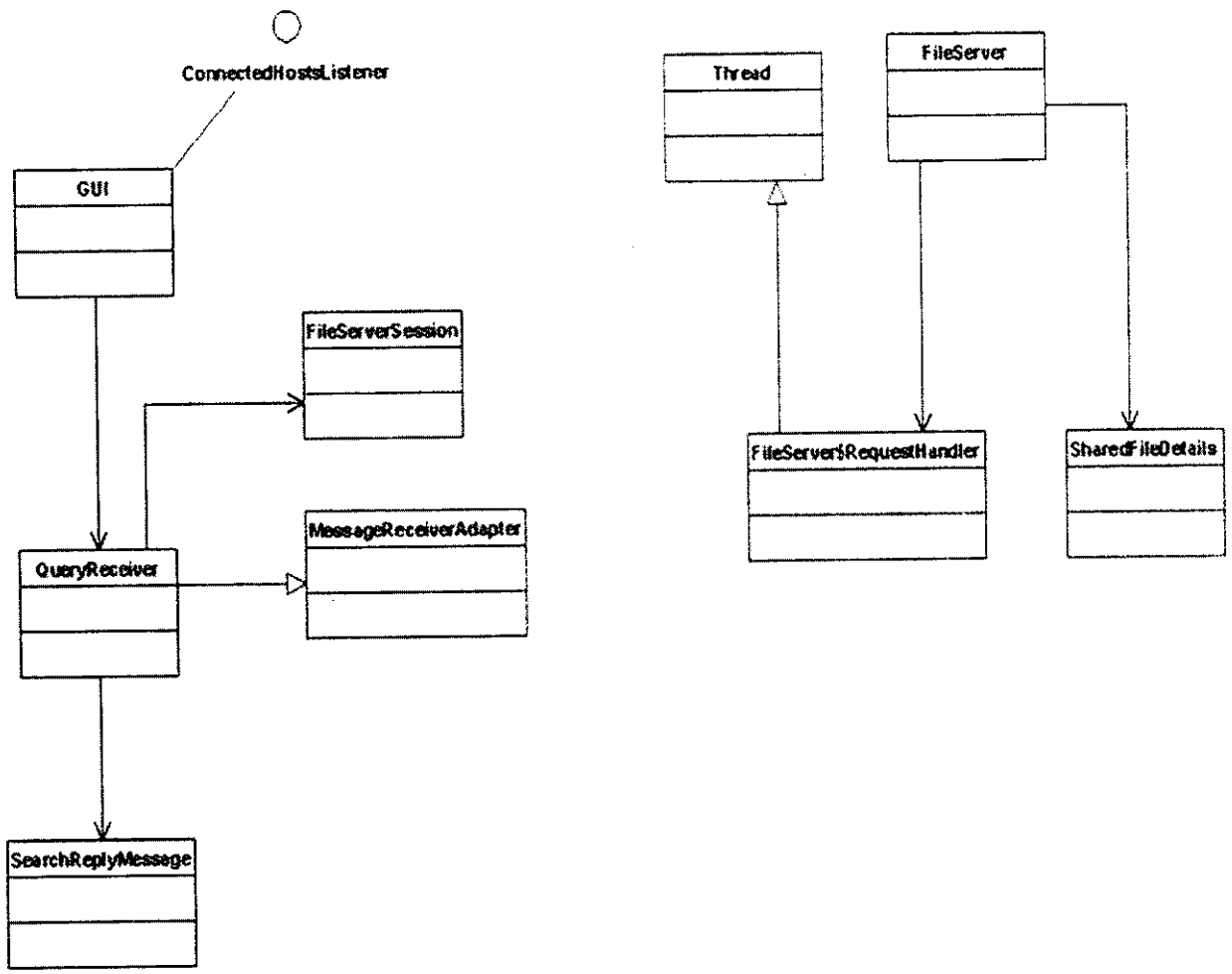


Figure 4.4 Uploading System Design

IMPLEMENTATION

CHAPTER - 5

IMPLEMENTATION

5.1 Implementation Overview

As mentioned above there are many new functions that have needed to be implemented so that the strategies/policies of download can be undertaken. In this section, the underlying functionality is explained and how these functions contribute towards the way in which downloads are undertaken. There have been many significant changes to the way that the Ishare implementation actually carries out downloads and these are also described below.

5.2 Shared File Data

As outlined previously, to-date there has been no support for file serving within the Ishare libraries. This situation has been rectified by adding file serving capabilities to the core set of Ishare classes. This therefore means that there are now 4 key sets of classes which make up the Ishare libraries; connecting, searching, downloading and uploading.

Two situations have been overcome with the adding of this server-side functionality. These being that there are far too many "freeloading" servents using the Gnutella network (Ishare being one of them) and also so that local controlled tests are able to be carried out if required. As there is now server-side file serving functionality within the Ishare libraries, future developers can obtain results from local controlled testing which may be conducted. These local simulated tests are likely to provide more meaningful results to the user as they are in control of what is

actually happening where as with the Gnutella network they have no "real" control.

The implementation includes a Shared File class which is an object that contains all the information of each of the shared files that the servent has. There is also the functionality in place now so that the servent receives search requests, checks to see if the request matches any of the servents shared files, and then sends a QueryHit to the querying servent. The incoming Query messages are received by the servent and are passed to the Router class so that the query criteria within them can be checked against any shared files that the servent has and routed as necessary. If there is a match then a new QueryHit is created and sent along the original connection to the remote host. The remote host should then receive the QueryHit message and decode it appropriately. If there is then an "HTTP GET" request sent to the servent for the required file, then the File Server acknowledges this request and acts upon it.

5.3 FileServer

The FileServer has been designed to handle the Server-Side operations of the Ishare Servent which have never been supported previously. The FileServer is designed in such a way that it accepts incoming "HTTP GET" requests which have been sent by other remote servents. The FileServer listens to Gnutella's default port 6346 or whichever port that the servent is connected on. The FileServer itself is designed to run on port 8080 (as it is a web service) but can be used on any port because this is not important. The important part of the FileServer is to make sure that it is listening to the correct Gnutella Port otherwise no requests will be received or acknowledged.

When the FileServer receives GET requests for particular files it decodes them

and uses the information contained within them to decipher whether or not the servent has the particular file which is required. If it does then the system acts upon the request as appropriate. If the file is present a check is done against the SharedFileDetails class to check the fileSize, SHA1 Hash Code etc. The information returned is then used and the appropriate amount of data is sent to the remote host which has requested it. However, if the file is not present the connection to the remote host is simply dropped and no further action is taken.

To enable remote hosts to communicate with the FileServer itself (not that they know they are actually doing this, they assume it is the servent itself which they are communication with) it includes the functionality to send a response message back to the remote servent acknowledging that it has received the GET request and is acting upon it. It therefore needs to recognize Gnutella HTTP 1.1 200 OK return codes to say that the file is either OK to download, or has support for incoming/outgoing connections to and from other remote servents(and also file streams in place to support these file transfers).

5.3.1 QueryReceiver

The QueryReceiver class simply monitors any Query Messages which are sent to the servent. A check is done to see if the file details of the Query message match any of the files in the servents shared directory. If a match is found with the file that has been requested then a QueryHit (SearchReplyMessage) is routed to the relevant remote host who sent the Query.

It is possible that this functionality could be used for certain other research purposes for monitoring the search criterion that the servent receives over a given time. Studies could be done to show which types of data are frequently requested

and what the variety of search criteria is. Ethical and social issues could be addressed by the monitoring of these searches and conclusions drawn.

5.3.2 SharedFileDetails

The SharedFileDetails class simply contains the relevant information for each of the files that are contained within the servent's shared directory. The class has the functionality to calculate the SHA1 Hash code of each of the shared files using the SHA1 facility which is built into Java's API Security Package. The class also has the functionality to get the filename according to the file's hash value. This is useful for comparison checks of individual files due to the fact that two files could have the same name but may not necessarily be the same file.

5.4 Default Download Times

Whenever a segment of a file is downloaded, the defaultTime ToDownload Section () method is called from within the Downloader class on the MultiSource Downloader object that it is listening to. This method takes the speed of the remote host and the size of the segment and uses these to find the default time that any particular section should take to download. This calculation includes an offset value to compensate for network traffic and congestion on the line. The algorithm used for this method is as follows: -

speed * 1024; //Speed in bits per second

fileSize * 8; //file size in bits

Default time (secs) = **(fileSize/speed) + offset;**

Naturally, a number of permutations of the time format are possible to create but the raw data that the function returns is the default time to download a section in seconds.

5.5 Segment Download Times

Whenever a segment completes downloading successfully then the check SegmentDownloadTime () (which is located within the MultiSourceDownloader class) method is called. This is to check what the actual time of the download was. This method calls and stores the return time of the defaultTimeToDownloadSection() method. The default time for a download and the actual time are then compared. The number of active servents that the file is being downloaded from is also used to make sure that a file is not dropped if it is the only source downloading.

If while conducting these checks, a file is found have downloaded faster than its default time then this host is perceived as being a potentially good source. The algorithm that this method makes use of is as follows: -

```
if(the number of active downloading servents == 1)
{
    Do not abort the file
    if(segment download time <= default download time)
    {
        A potentially good host has been found
        Add this Host's IP Address to the Info file as a potentially good source
    }
    else if(segment download time > default download time)
    {
        Do not abort the file as only one source
        Re-search the network for other files of same search criteria
        Re-start a new download on the new hosts with replicas of the file
    }
```

```
}
else if((segment download time > default download time) and (the number of active
downloading!=1))
{
        Abort selected segment
        Re-search for the same file using SHA1-Hash value
        Re-start a new download on the new hosts with replicas of the   file
}
else if(segment download time <= default download time)
{
        A potentially good host has been found
        Add this Host's IP Address to the Info file as a potentially good source
}
```

5.6 Keeping Info

This mechanism of the system is used when there has been a re-search for a file executed. The KeepingInfoPolicy class is used to check whether or not a new Host who has sent back a QueryHit (in response to the search sent out) is a potentially good source to use. To do this, the object reads the Info file (which stores all the IP Addresses of Hosts that have previously had files downloaded from and that have downloaded faster than the default times). If the remote host IP is contained within the file then this source is attempted over other hosts which have also returned QueryHit messages.

In addition, a further check is done to see if any of the hosts that have returned QueryHit descriptors are actually globally positioned in the same place as the servent. If any IP Addresses match the servent IP Address, then it is assumed that the

remote Host is on the same network as the servent. Therefore it is each of the policies jobs to attempt to download from this host before others.

To read the file the KeepingInfoPolicy class uses the KeepingInfoPolicy FileReader class to actually read the file. This class simply reads the contents or sets the contents by adding IP Addresses if a potentially good Host is found. As mentioned above the checkSegmentDownloadTime () method uses this policy to check which Host should be attempted first. If the policy is found to not return any local globally positioned sources or no sources which have previously proved to be good sources, then downloads are conducted as normal.

The algorithm that this policy uses to check if a source is potentially good source is:

```
Check if the source is good()
{
    read the Info file that contains the IP Addresses of good sources check if the
    IP Addresses of Hosts which sent back search results are in the Info file, if any
    of them are then return that the source is good
}
```

The algorithm that this policy uses to check if the servent is globally positioned near or on the same network as a remote Host is: -

```
Check if IP Address is the same (serventIPAddress, remoteIPAddress)
{
}
```

5.7 Set File Segment Size

The SegmentSize class is used to set the number of individual segments that a requested file will be split into. Naturally, the size of the segments can be altered if

necessary. The class sets the number of segments according to the size of the file.

It was felt that the number of segments that any particular file should be split into should be proportional to the files size. This way the servent will not be doing any extra work, by say for requesting 20 or so segments for a file that is only 1 MB in size. Also, granted that it is not feasible just to request a 700MB file from 1 source. There needs to be a balance which will hopefully be shown with testing of this functionality. If this were the case then potentially more requests for the file could be being sent rather than the actual downloading of the file. Therefore the purpose of the different number of file segments is to increase file download efficiency.

The initial suggestion for the number of segments before any testing to prove if these values are relevant can be seen below.

Size Of File	Number of Segments
< 2 MB	5
< 6 MB	10
< 50 MB	15
> 50 MB	20

Table 5.1 - Default Segment Sizes

The algorithm that this object uses is simply to check if the size of the file is above or below a certain value, as shown below:-

```
if(file Size < X)
{
    segments = A
}
```

```
else if(file Size < Y)
{
        segments = B
}
else if(file Size < Z)
{
        segments = C
}
else
{
        segments = D
}
```

Where X, Y and Z are the size of the file in bytes and A, B, C and D are the number of segments the file will be split into. Consequently due to the way that the system is designed, the number of different hosts/sources that the file will be requested from is also the number of segments that the file will be split into.The number of segments was not simply a random number; careful consideration was given to choosing the number of sources that the file was to be downloaded from. The theory behind the decision of the number of segments is explained below: -

File Size < 2 MB

Most files such as image files which people may wish to search for and download such as jpg or gif files are smaller than 2 MB and so splitting the file into 5 segments was seen as a suitable decision for this size of file.

File Size < 6 MB

It could be argued that the most common type of file that is requested for download from the Gnutella Network is the mp3 format Music file. As most mp3

files are approximately 5.5 MB, a segment size of 10 was deemed suitable for this sized file.

File Size < 50 MB

Many small movie clips which are available to download from peers within the Gnutella Network are less than 50 MB and so a suggested segment size of 15 seems plausible.

File Size > 50 MB

Most other files such as movie files are naturally above these other previously discussed file sizes. Therefore it is assumed that if the file size is above 50 MB then a suitable number of file segments is 20.

Despite these above arguments, when testing the system, it may be proven that these suggested values do not return the optimum output and may need changing. However, it is thought that there is sufficient reasoning to start with these suggested values.

5.8 Download Constants

This class now has support to recognise connections of type: Modem, ISDN, Dual-ISDN, 256K Cable, 350K Cable, 512K Cable, 768K Cable, T1, T3, 10MB LAN and 100MB LAN. The speed values of these connection types are the standard recognised values (e.g. 56K) but there can be many variations of these speeds within the Gnutella network and so it was felt that a general Connection Type would identify the speed values into separate areas of connection. These connection types are however not 100% accurate but are merely only an estimate of the Remote Hosts Connection Type based on its set Speed value.

5.9 Automatic Download Restart

If for some reason a particular download is started and fails before transmitting any pieces of the required file to the servent, then it was felt that the download should restart and try all the possible given sources of the file once more. If however the download fails a second time then it should be dropped and the file re-searched for. Application of this theory has shown that in more cases than not this theory was well worth implementing because most downloads do actually start when this restart functionality is used.

PROCESS DESCRIPTION

CHAPTER 6

PROCESS DESCRIPTION

This chapter will explain the main internal processes and interactions involved with connecting to the Gnutella network and then finding and downloading a file. The aim here is to give the reader an idea of what interactions are actually going on in the system while these operations are being carried out. Not all options are covered, but the operation of the system in the event that no problems are encountered should be clear after reading this chapter. Each section has an associated Figure. See the Figure for a simple diagram of the thread of control in each instance. Numbers in brackets in the text refer to the same numbered arrow in the associated diagram.

6.1 Connecting To The Gnutella Network

On clicking the 'Connect' button in the first tab of the GUI, a GnutellaConnection is created (1). It is passed a ConnectionData instance as an argument (2), which contains many of the options set in the 'Options' tab of the GUI. The GnutellaConnection start() method is then called, which in turn instantiates and starts up all the other objects and threads required for connecting to the Gnutella network (3). These include a GetHostsFromCache object (4), which immediately attempts to connect to one of the GWebCaches in the GWebCache class (5) and obtain a list of hosts for initial connection attempts. These hosts are then placed in a HostCache object (6).Once a list of hosts has been obtained, the OutgoingConnectionManager comes in to play. It starts a NodeConnection object

for each of the hosts in the Host cache in turn 7), and calls the startIncomingConnection() method of that NodeConnection.

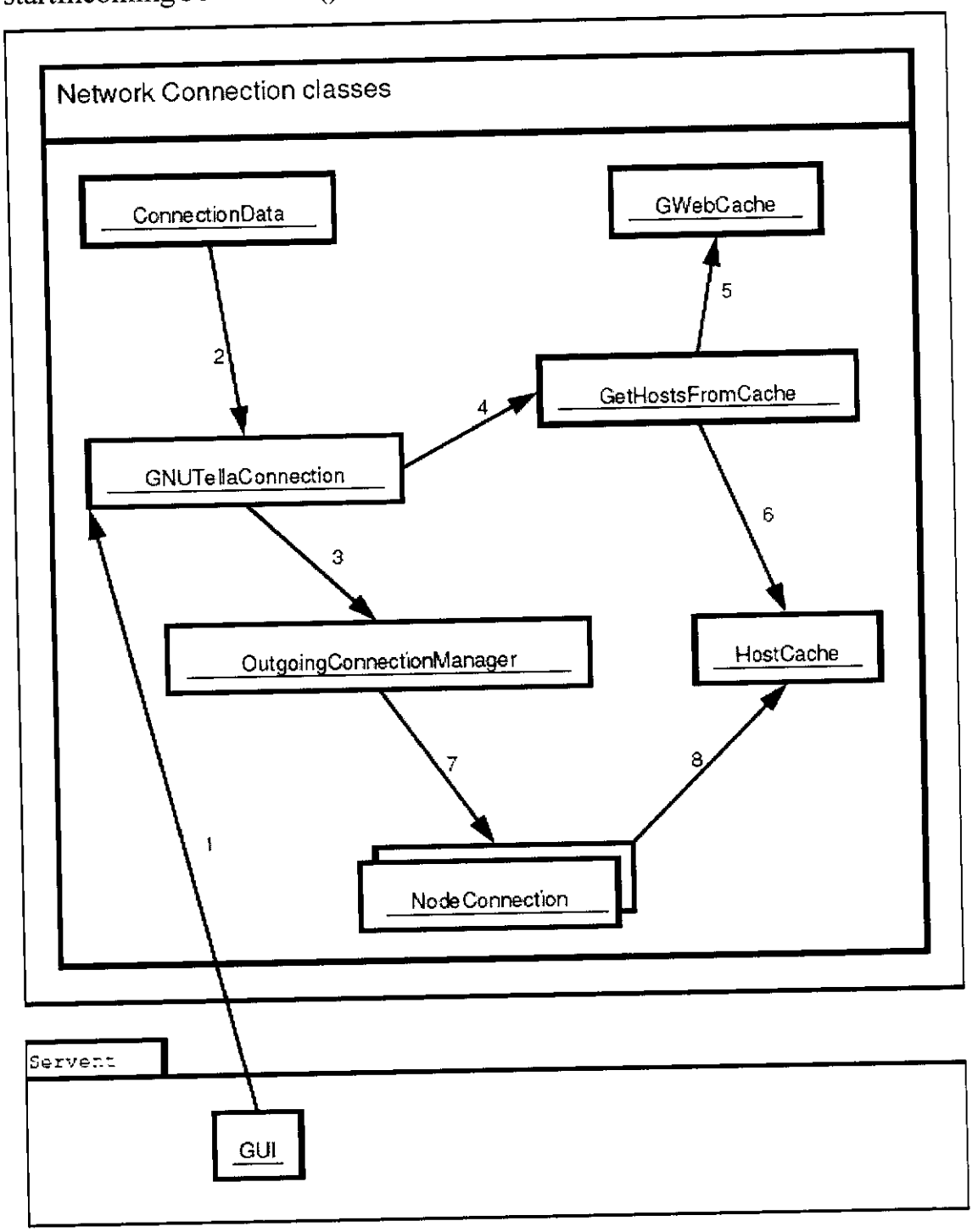


Figure 6.1 Thread of control - Connection

This method then attempts to handshake awith the remote host. If a '200 OK' reply is received during the handshake then a new connection is listed as having been created to that remote host, and any listeners waiting on this event are notified. If any other reply is gained then the connection is dropped. In either case the response is also searched for the 'X-Try-Ultrapeers' header, and if the HostCache is shorter than its maximum length then hosts are added from the 'X-Try-Ultrapeers' header to the HostCache (8). This ensures that the HostCache never empties itself. This process of attempting connections to remote hosts repeats until the required number of connections (as specified in the ConnectionData instance) has been reached.

6.2 Searching For A File

When the 'Search' button in tab two is pressed, a new SearchSession is created (1, 2). This is always the same object, which is nullified and then recreated every time a new search is made. The two most important arguments which are passed to the SearchSession on creation are the text to search for and a MessageReceiver, in this case of type SearchResponseReceiver, which will deal with replies. Firstly, the SearchSession starts up a SendThread (3), when initially sends the search message out to every remote host urrently connected to the servent (i.e. every NodeConnection instance) (4). The SendThread then waits for any additional Connections which may appear during the lifetime of the search, and sends the message to those newly connected hosts as well. The SearchSession then calls the routeBack() method of Router (5), which places an identifier into the OriginateTable.

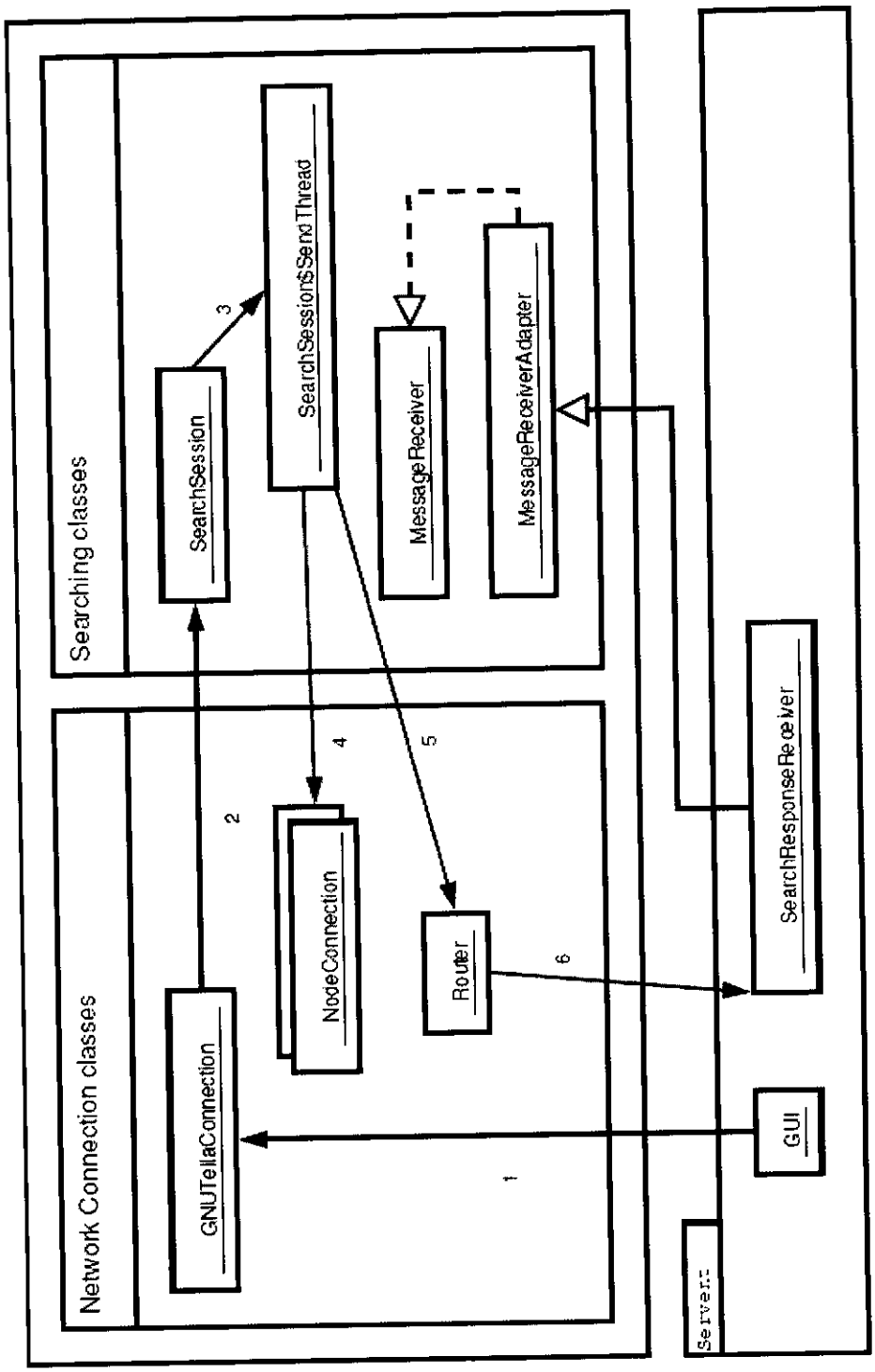


Figure 6.2 Thread of Control - Searching

This identifier allows the Router to check if an incoming message is a response to a message that the servent originally sent out or not. If it is, and if the

incoming message is a query hit, then the message is passed on to the SearchResponseReceiver that the SearchSession initially took as an argument (6). In particular, the receiveSearchReply() method is called. This method then adds the details of the message to the TableTree on the second GUI tab.

6.3 Downloading A File

When a file is double clicked on in the search responses TableTree it, and all other files with the same sha1 hash that have been received as search responses, are passed in a Vector as an argument to a MultiSource Downloader(1). The MultiSourceDownloader then loops through the list of file sources, starting a Downloader and creating associated objects for each one up to a limit specified in the code (2). Once this limit has been reached the MultiSourceDownloader waits for one of the current downloads to finish. The alternative is that the MultiSource Downloader empties the list before it reaches the specified limit. In this case, it first starts a new search for files with an identical sha1 hash to the requested one (3, 4), and then waits for a response to the search (5, 6) before attempting to reach the limit again. Each Downloader is passed a Section as one of its arguments, which specifies which part of the file it is to request for download. attempts to connect to its specified remote host, either by using an already open connection, connecting directly, or sending a Gnutella push message to the host in question.

Once a connection has been established the Downloader requests the specified section of the required file.It then streams the data it receives from the network into a file, buffering it in chunks to prevent disk access from crippling the machine.The Downloader finishes once all requested bytes have been received or if some error occurs, such as the socket becoming disconnected. The variables in the associated Section are set, and then the MultiSourceDownloader is notified of the Downloader

finishing.Once all bytes of the file have been downloaded, the MultiSource Downloader reassembles all the parts of the file into a single final file, and then finishes.

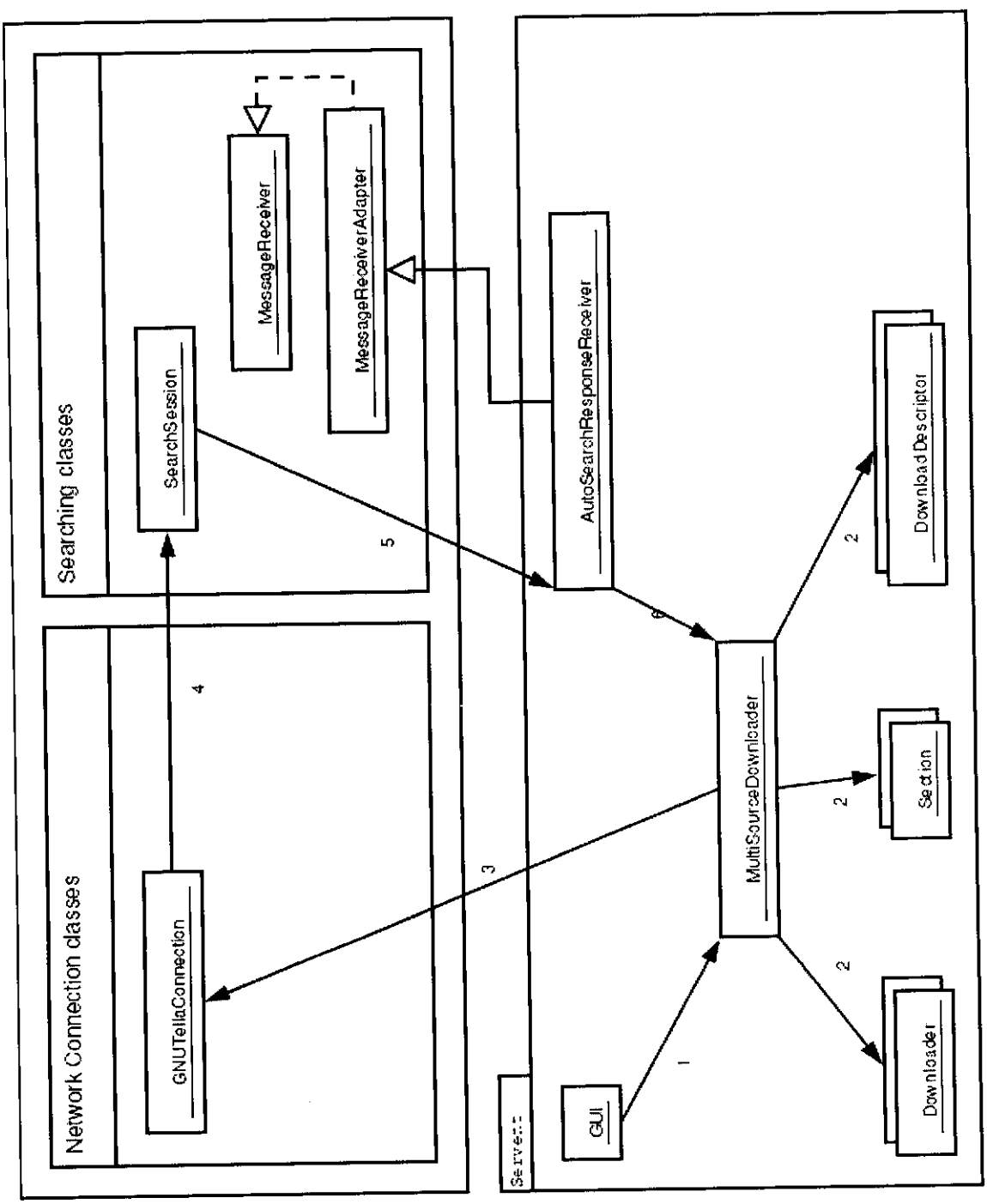


Figure 6.3 Thread of Control- Downloading

TESTING

CHAPTER 7

TESTING

Testing of this project servent will have two main sections. Firstly, testing of various values for the maximum number of hosts to download a file will be undertaken to see if multi-sourced downloads provide some advantage over single sourced ones with the system used in the project servent and also to see if there is some optimum value for the number of hosts to upload from. Secondly, the project servent will be tested against other well known Gnutella servents to see how it measures up. Results and conclusions from all these tests can be found below.

7.1 Download Speed

	Average Time			Average Speed		
File	1 Host	5 Hosts	10 Hosts	1 Host	5 Hosts	10 Hosts
1	2m14s	1m58s	1m48s	22.09KB/sec	25.45KB/sec	28.84KB/sec
2	4m34s	2m55s	3m00s	15.44KB/sec	25.42KB/sec	21.31KB/sec
3	3m23s	3m56s	2m51s	29.87KB/sec	24.85KB/sec	30.44KB/sec
4	7m32s	3m56s	3m06s	18.54KB/sec	30.48KB/sec	31.86KB/sec
5	6m33s	3m13s	4m28s	22.65KB/sec	39.19KB/sec	29.94KB/sec

Table 7.1 : Average Downloads For Different Max Hosts Compared

Initial tests involved varying the number of hosts allowed for the project system to download a file from at any given time. Values of 1 (single-sourced downloading), 5 and 10 for the number of allowed download sources were tested, with some interesting results. 5 files were downloaded, 5 times each. The files were chosen to try and take a range of download sizes, as of course the size of the download affects the size of each section, which may affect the overall download speed. It would have been preferable to also attempt downloads of much larger files. Unfortunately, not enough download sources which were available for any file of a much larger size could be found to make it a viable option. Each file was downloaded 5 times to try and remove the possibility of a statistical fluke where a very fast or very slow host overly affected the results.

The table shows only the averages of each download, to allow comparisons to be seen more clearly. For the most part, multi sourcing seems to provide an increase in download speeds. However, this is not always the case and further increasing the number of hosts able to be downloaded from does not seem to significantly increase the average download speed in enough cases for it to be considered viable.

CONCLUSION & FUTURE WORK

CHAPTER – 8

CONCLUSION & FUTURE WORK

8.1 Conclusion

Overall it is fair to say that there has been vast knowledge gained from implementing this project. There has been a greater understanding of the Java programming language with regard to Object Orientated Design. Furthermore, for the project to even begin there had to be an interest within the area of Peer-2-Peer Distributed architecture. This knowledge has grown with respect to the Gnutella Protocol from its initial introduction to current day specification.

Through testing of the project, it has shown that all the new functionality that has been added to the servent implementation has been worth while. This is because by using the strategies of download, the actual download times for files have been significantly reduced. However, it is fair to say that there could be much more done within this area to reduce download times further.

A run through of the strategies using the Gnutella network provided meaningful results that showed that all of the strategies are working and do actually reduce download times. When adding further functionality to the system, blocking I/O calls have to be compensated for and also concurrent thread access etc.

In conclusion, the main objectives of the project have been met. Finally, the project has provided some meaningful results which should be considered by anyone who is intending designing a multi-source downloader for the Gnutella network.

These are: -

- For the servent to avoid being dependant on a slow host (due to the hosts file segment downloading slowly) there needs to be some functionality in place that can check a download and if it is performing slowly then it can be dropped and another host attempted. This is imperative to keep the servent downloading at a higher speed.
- There needs to be some way of checking if a remote host is actually geographically situated on the same network as the servent. This is so that the servent is not attempting to download from other remote hosts on the other side of the world if there is another servent on the same network.

- A mechanism for checking the download times of each segment of a file as they finish downloading is certainly needed to accompany the dropping of a download. This is so that if a potentially good host is found (i.e. one that has downloaded a segment of a file quicker than the default time) then this should be used again before attempting other hosts.

- There could also be a mechanism in place to check how many sources of a file are available. If for example the file is well populated and there are 50 sources of the file, then an attempt could be made to download all 50 simultaneously. This is most likely to give a better download rate than say using 5 source of the file. (Bandwidth limitations must however be taken into consideration)

8.2 Future Work

There is much scope for further work within this area of which there is too much to describe. There are still many sections of the Gnutella Specification that does not support and naturally these could be included. However, some key areas

that are seen to be important are outlined below and arguably the most important being that the implementation should be made to include server-side file serving capabilities. This is an important issue which needs to be resolved.

8.2.1 Active Queuing System

This Active Queuing System allows the provision of servents actively waiting in a queue to download a particular file from a given host which is currently uploading many files and so consequently has no free uploading-slots available. It is not hard to see that this functionality is far better than regularly probing the remote servent to see if it has any slots free. With this system of operation, there is not the risk factor that another servent could request a free slot whilst another is waiting. All servents form a queue and await a free slot in a more orderly manner.

8.2.2 X-Gnutella- Location Header

This header is included in the search response message which is sent by servents and contains a list of known remote hosts who have the identical file which was requested. At the start of the project it was the aim to include support for this header. This is because it was seen as the obvious solution to finding sources of files from other locations. However, after research was done into this area it was found that this was an optional Gnutella header and that not all servents used it. It was therefore seen that a substantial re-write of the DownloadDescriptor class to support this was not in the scope of this project.

8.2.3 Static Download State

When downloads are conducted all information about them is stored in the computers memory. This solution is not a problem if the particular download completes without any error. Many downloads however do not complete and so their state is lost. This is a useful phenomenon because downloads could be restarted from where they last finished without having to be restarted again.

REFERENCES

REFERENCES

Books:

1. Andrew Oram "Peer-To-Peer : Harnessing the Power of Disruptive Technologies" O'Reilly & Associates, 1st edition March 2001, ISBN: 059600110X.

2. David Barkai "Peer-To-Peer Computing: Technologies for Sharing and Collaborating on the Net, Intel Press, 1st edition, 2002, ISBN: 0970284675.

3. Elliotte Rusty Harold "Java network programming", O'Reilly Media, Inc., 3rd edition 2004, ISBN: 0596007213.

4. Herbert Schildt, Patrick Naughton, "Java 2 Complete Reference", Osborne Publishing, 3rd edition , 1999 ISBN: 0072119764.

5. Merlin Hughes "Java network programming - A Complete Guide to Networking, Streams and Distributed Computing" Manning Publications, 2nd edition,1999,ISBN:188477749X.

Web:

http://en.wikipedia.org/wiki/Peer-to-peer(Visited during the month of jan 2006).

http://www.the-gdf.org/index.php?title=Main_Page(Visited during the month of jan 2006).

http://java.sun.com (Visited during the month of dec 2005).

http://eclipse.org (Visited during last week of march 2006).

APPENDICES

APPENDICES

A.1 Testing Results

This section contains the full results of tests done on the various servents. It should be noted that the average download time or speed listed for each section is the average of the values of time or speed for the five downloads attempts, and as such may not match up. To explain by example, a single download with a much higher than normal download speed would have a fairly large effect on the average download speed for the file. However, due to the file sizes used in testing being relatively small, this might only result in a relatively small drop in the download time. As such, the average of the download speed value would be lower than expected compared to the average of the download time value.

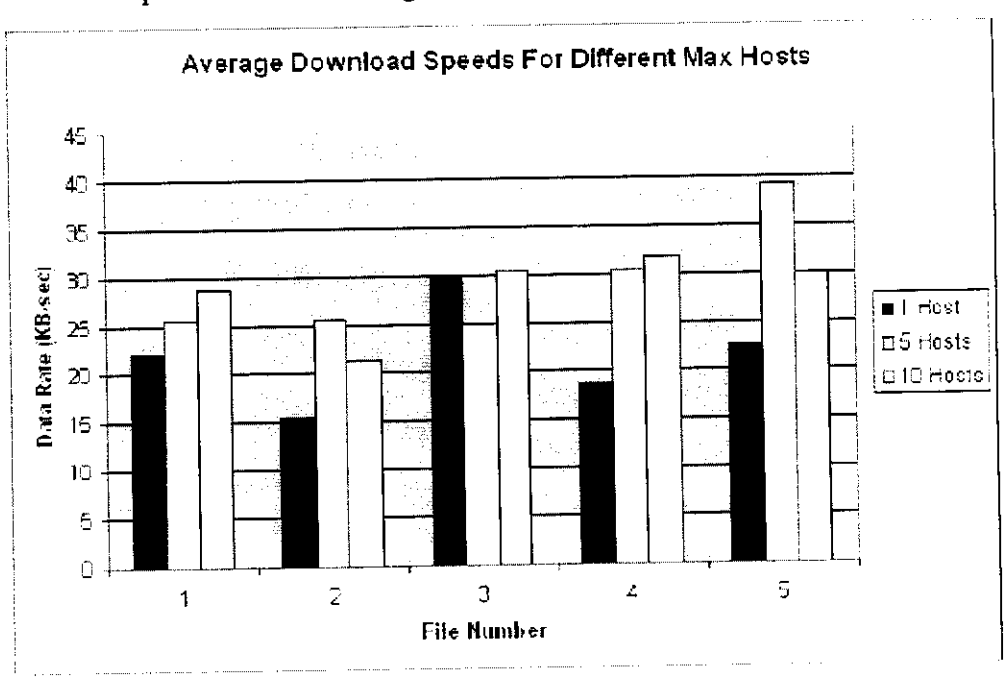


Figure A.1 Download Speeds for Different max hosts

File		Download		
Num	Size	Attempt	Time	Speed
1	2.58MB	1	1m47s	24.73KB/sec
		2	3m50s	11.50KB/sec
		3	1m26s	30.76KB/sec
		4	1m51s	23.84KB/sec
		5	2m15s	19.60KB/sec
		average	2m14s	22.09KB/sec
2	3.61MB	1	6m13s	9.90KB/sec
		2	2m29s	24.78KB/sec
		3	4m16s	14.42KB/sec
		4	3m16s	18.84KB/sec
		5	6m38s	9.28KB/sec
		average	4m34s	15.44KB/sec
3	4.43MB	1	3m53s	19.48KB/sec
		2	2m13s	34.13KB/sec
		3	1m44s	43.64KB/sec
		4	7m14s	10.46KB/sec
		5	1m49s	41.64KB/sec
		average	3m23s	29.87KB/sec
4	5.39MB	1	12m30s	7.36KB/sec
		2	9m33s	9.63KB/sec
		3	6m30s	14.15KB/sec
		4	7m13s	12.74KB/sec
		5	1m53s	48.84KB/sec
		average	7m32s	18.54KB/sec
5	7.35MB	1	6m18s	19.91KB/sec
		2	9m37s	13.05KB/sec
		3	3m01s	41.59KB/sec
		4	8m37s	14.56KB/sec
		5	5m12s	24.13KB/sec
		average	6m33s	22.65KB/sec

Table A.1 Single sourced downloads

File		Download		
Num	Size	Attempt	Time	Speed
1	2.58MB	1	2m27s	18.00KB/sec
		2	2m11s	20.20KB/sec
		3	1m11s	37.26KB/sec
		4	1m14s	35.75KB/sec
		5	2m45s	16.03KB/sec
		average	1m58s	25.45KB/sec
2	3.61MB	1	3m45s	16.34KB/sec
		2	2m54s	21.22KB/sec
		3	1m37s	38.07KB/sec
		4	1m36s	38.46MB/sec
		5	4m44s	13.00KB/sec
		average	2m55s	25.42KB/sec
3	4.43MB	1	2m15s	33.62KB/sec
		2	6m52s	11.02KB/sec
		3	2m09s	35.19KB/sec
		4	6m04s	12.47KB/sec
		5	2m22s	31.96KB/sec
		average	3m56s	24.85KB/sec
4	5.39MB	1	3m10s	29.04KB/sec
		2	2m21s	39.14KB/sec
		3	2m46s	33.24KB/sec
		4	3m06s	29.67KB/sec
		5	4m19s	21.31KB/sec
		average	3m56s	30.48KB/sec
5	7.35MB	1	3m18s	38.02KB/sec
		2	3m04s	40.91KB/sec
		3	3m22s	37.26KB/sec
		4	3m21s	37.45KB.sec
		5	2m58s	42.29KB/sec
		average	3m13s	39.19KB/sec

Table A.2 Five sources Downloads

File		Download		
Num	Size	Attempt	Time	Speed
1	2.58MB	1	1m12s	36.75KB/sec
		2	1m10s	37.80KB/sec
		3	1m11s	37.26KB/sec
		4	2m54s	15.21KB/sec
		5	2m34s	17.18KB/sec
		average	1m48s	28.84KB/sec
2	3.61MB	1	2m39s	23.22KB/sec
		2	3m21s	18.37KB/sec
		3	2m26s	25.29KB/sec
		4	2m30s	24.62KB/sec
		5	4m05s	15.07KB/sec
		average	3m00s	21.31KB/sec
3	4.43MB	1	2m04s	36.60KB/sec
		2	2m01s	37.51KB/sec
		3	2m26s	31.09KB/sec
		4	2m17s	33.13KB/sec
		5	5m27s	13.88KB/sec
		average	2m51s	30.44KB/sec
4	5.39MB	1	2m15s	40.88KB/sec
		2	2m14s	41.18KB/sec
		3	2m58s	31.00KB/sec
		4	4m30s	20.44KB/sec
		5	3m34s	25.79KB/sec
		average	3m06s	31.86KB/sec
5	7.35MB	1	5m52s	21.38KB/sec
		2	5m23s	23.30KB/sec
		3	3m08s	40.04KB/sec
		4	4m41s	26.79KB/sec
		5	3m17s	38.21KB/sec
		average	4m28s	29.94KB/sec

Table A.3 Ten sources Downloads

A.2 Sample outputs

I share – connector tab

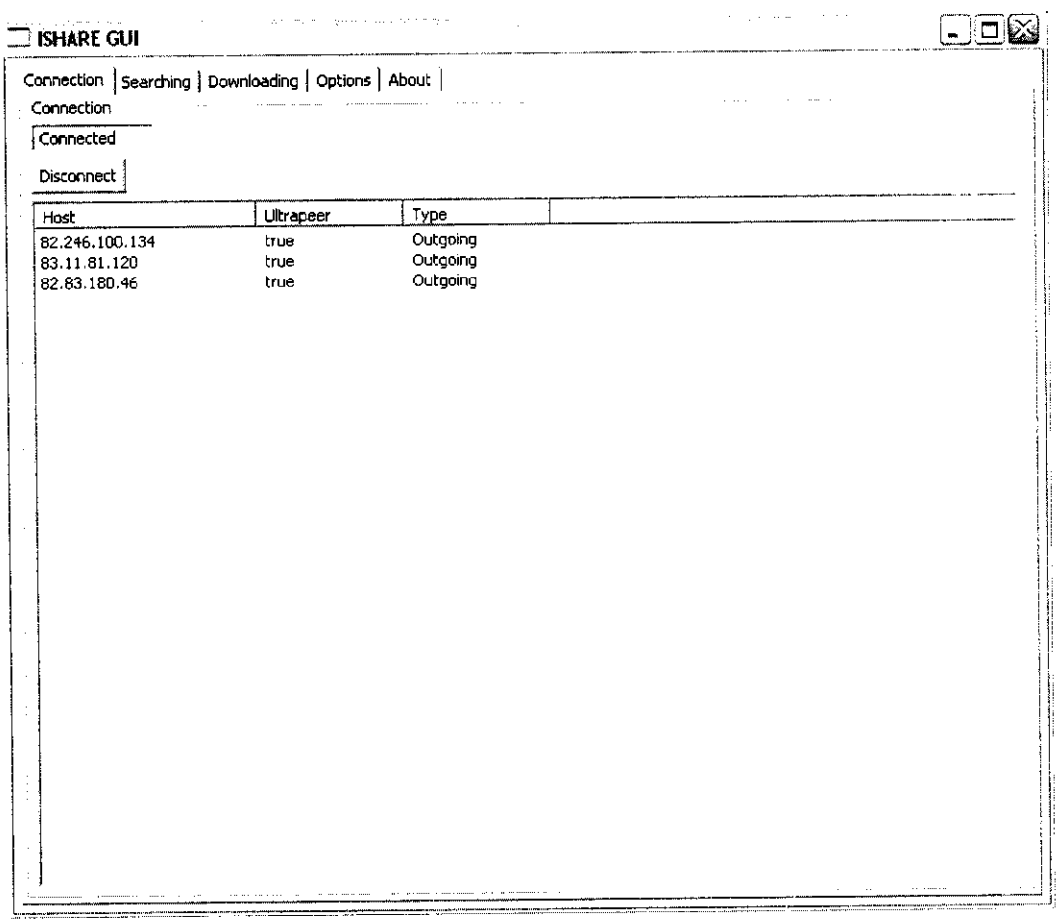


Figure A.2 - Connector GUI

I share –Searcher tab

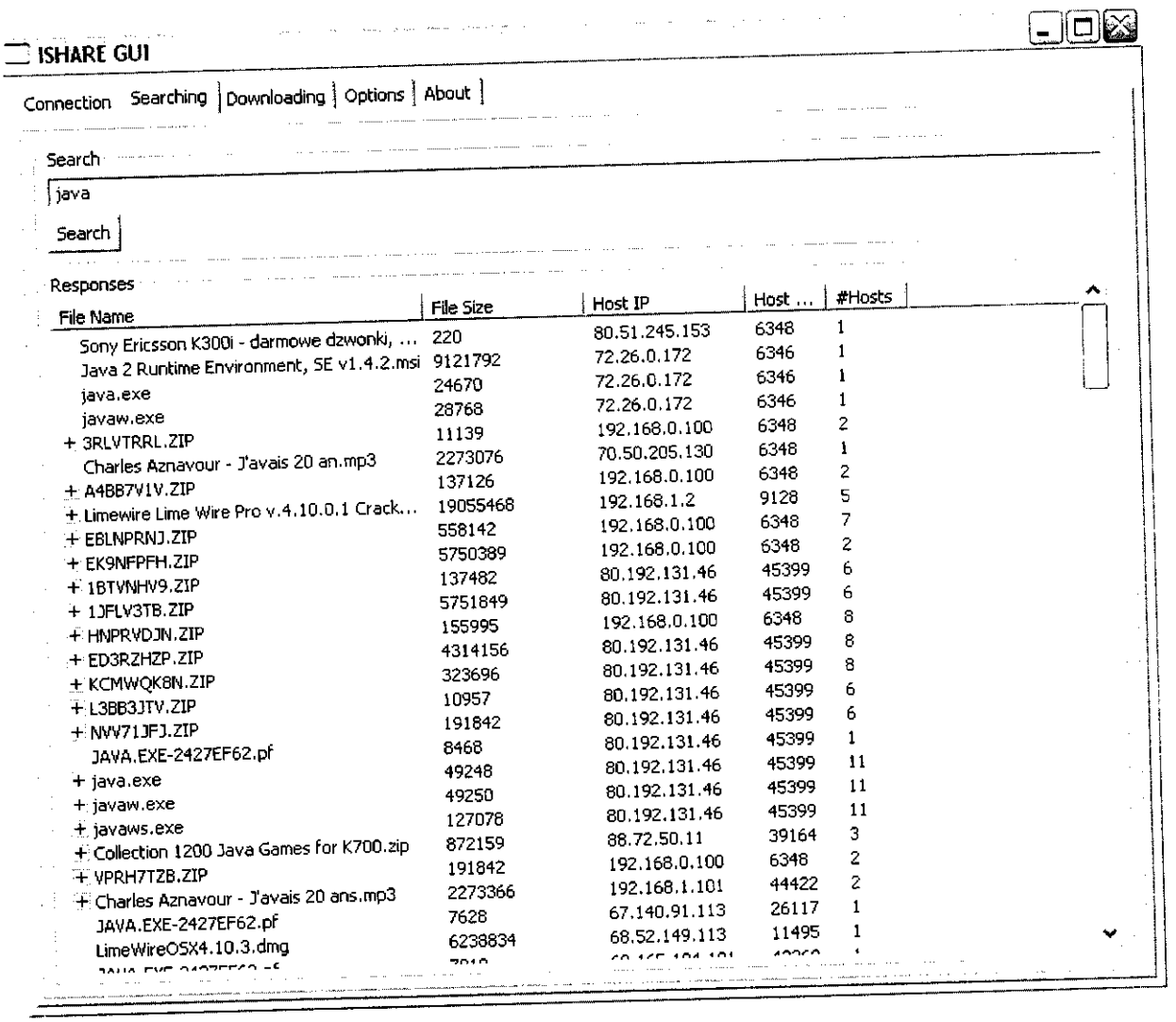


Figure A.3 - Searcher GUI

I share – Downloader tab

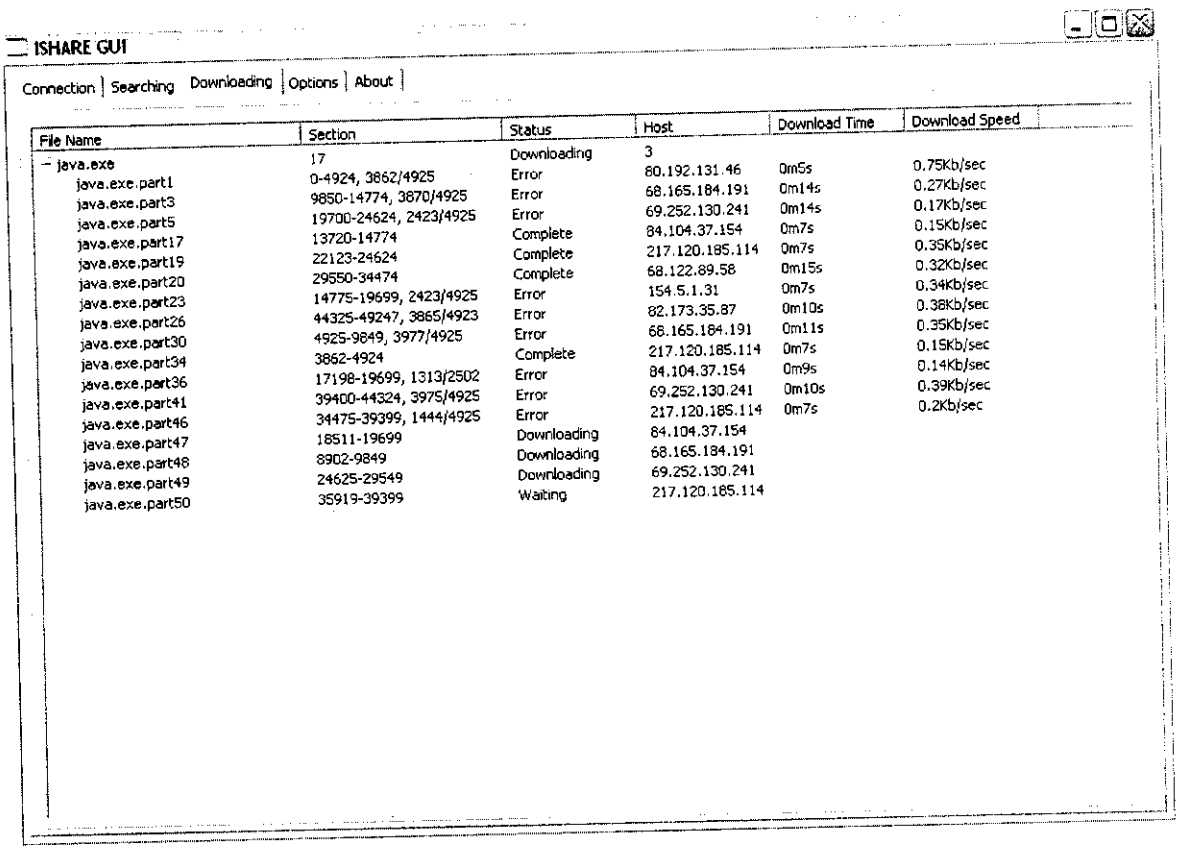


Figure A.4 - Downloader GUI