Umeå University
Department of Computing Science
Master Thesis

Improving performance of modern Peer-to-Peer services

Marcus Bergner

10th June 2003

Abstract

Peer-to-peer networking is not a new concept, but has been resurrected by services such as Napster and file sharing applications using Gnutella. The network infrastructure of todays networks are based on the assumption that users are simple clients making small requests and receiving, possibly large, replies. This assumption does not hold for many peer-to-peer services and hence the network is often used inefficiently.

This report investigates the reasons behind this waste of resources and looks at various ways to deal with these issues. Focusing mainly on file sharing various attempts are presented and techniques and suggestions on how improvements can be made are presented.

The report also looks at the implementation of peer-to-peer services and what kind of problems need to be solved when developing a peer-to-peer application. The design and implementation of a peer-to-peer framework that solves common problems is the most important contribution in this area.

Looking at other peer-to-peer services conclusions regarding peer-to-peer in general can be made. The thesis ends by summarizing the solutions to some of the flaws in peer-to-peer services and provides guidelines on how peer-to-peer services can be constructed to work more efficiently.

Contents

List of Figures

List of Tables

Preface

This preface contains information related to the master thesis that is not directly related to the topic itself. Information about the author and supervisors are presented here. An informal introduction to how and why this report was created is also present. Some details regarding the software used to complete the thesis is also presented.

About this report

The initial ideas for this thesis, and hence this report, appeared during the summer the year 2002. Companies in Umeå at this time did not have much need for students participating in their work, by doing a master thesis at their company. Only a few years earlier this situation was the completely opposite, but times had changed.

After looking around at various companies for an interesting thesis proposal I, the author, decided to take control of the situation. Finding a suitable topic was the next necessary step and peer-to-peer networking was an appealing area. After looking at various sources of information on peer-to-peer the most interesting aspects of the area appeared to be the performance of peer-to-peer services. Since most users view of peer-to-peer is synonymous with file sharing applications it felt important to present other available services using similar techniques as well, although file sharing would make up a substantial part of the thesis.

Identifying flaws in the usage of network resources and possible solutions to these flaws would be the major part of the thesis. This included investigating existing protocols, such as Gnutella, in theory and practice. Investigating alternative solutions for addressing and querying with the purpose to minimize wasting network resources was to be an important aspect of this thesis.

The practical aspects of the thesis were not really that well defined in the beginning, but became more obvious as the work progressed.

About the author

If you are interested in contacting the author of this report the information in table 1 on page ii should prove to be useful.

At the moment of writing, I am completing my final year as a student in the Master of Science Programme in Computing Science at Umeå University. This thesis concludes five years of successful studies and results in a Masters degree.

Name	Marcus Bergner
Personalnr.	790426–8591
Address	Gnejsvägen 47–108
Postal	907 40 Umeå
E-mail	`bergner@cs.umu.se`
Phone	+46–(0)70–270 18 40

Table 1: Author contact information

Over the last three years studies have been combined with working as a teaching assistant at the Department of Computing Science. During this time I have also had the privilege to hold numerous lectures and met a lot of nice people through my work.

My knowledge of computing science has over the past years increased from knowing a few programming languages and some elementary theory to knowing a wide variety of languages, tools, techniques and technologies

Supervisors

The primary supervisor for this thesis was Jerry Eriksson, Ph. D. and the formal examinator was Per Lindström. Both are working at the Department of Computing Science at Umeå University. To contact either of these supervisors table 2 should provide sufficient information.

Name	Jerry Eriksson	Per Lindström
E-mail	`jerry@cs.umu.se`	`perl@cs.umu.se`
Phone	+46–(0)90–786 76 68	+46–(0)90–786 61 24

Table 2: Supervisor contact information

Typographical conventions

Throughout this thesis a number of typographical conventions are used.

Roman regular
: Used for plain text. Denotes nothing special at all.

Roman italic
: Used to define new terms. The term will appear in the index and glossary. Is also used for emphasizing important words or phrases.

Roman bold
: Used for description and table headings.

`Constant width`
: Used for code and protocol related constructs, file names and URLs.

Constant width slanted
: Used for code and protocol related constructs that should be substituted for some real value or contents.

Constant width bold
: Used to highlight important parts of protocol related constructs.

SMALL CAPITALS
: Used for abbreviations. The abbreviation will appear in the index and glossary.

Sans-serif regular
: Used for chapter descriptions and table and figure captions. Also used for function declarations in pseudo code.

Sans-serif bold
: Used for names of programs and utilities. The name will appear in the index and glossary.

Software used

The programming performed during this thesis has mostly been performed on a Sun UltraSparc 10 workstation running the Solaris 8 operating system from Sun Microsystems and Linux based systems running the Debian GNU/Linux distribution, Woody. Some simulations have also been performed using a maching running the Microsoft Windows XP operating system.

The Java Development Kit (JDK) version 1.4.1 was used for most of the programming, although C was used to implement some simulations, with the assistance of the GNU C compiler **gcc** 2.95.2.

Other tools used throughout the work with this thesis are the network analyzer **Ethereal** for both Windows and Linux, the Gnutella client **Limewire** versions 2.8.6 and 2.9.10 along with a Linux installation of giFT/OpenFT version 0.10.0 with the giFTcurs frontend.

Most of the code and this report were written using either the **nedit** or **vim** editors. The LaTeX macro package for the TeX compiler were used to assist in the creation of this report. The bibliography was managed using BibTeX and the index was created using the **makeindex** utility with the assistance of a **Perl** script. Most of the figures were developed using **xfig**.

The Portable Document Format (PDF) version of this report was created using **pdflatex** with any PostScript images converted using **epstopdf**. The *PostScript* version was generated using **dvips**, and the web based HTML (Hypertext Markup Language) version was generated using **latex2html**. All of these versions, along with some additional documentation, is available at the following website.

`http://www.cs.umu.se/~bergner/thesis/`

Acknowledgments

Several people deserve an acknowledgment for contributing with ideas, motivation or any other important aspect of life. My supervisor, Jerry Eriksson

deserves one for his patience with this thesis, which hopefully turns out satisfactory.

My gratitudes are also given to my office mates, Per Nordlinder and Jon Hollström, for keeping the spirit up in our office over the years working as teaching assistants. Further on, I would like to thank all the other people that I consider to be my friends and family, since without them my life would be far too empty.

Without the extraordinary efforts of Donald Knuth (author of TEX), Leslie Lamport (author of LATEX) and Oren Patashnik (author of BIBTEX) this thesis would not look as appealing as it does, for which I am eternally grateful.

Marcus Bergner

Umeå University
June, 2003

bergner@cs.umu.se
http://www.cs.umu.se/~bergner

Chapter 1

Requirements

This chapter specifies the details concerning this thesis. All requirements are specified in sufficient detail.

1.1 Summary of requirements

The goal of this thesis is to present a thorough study of peer-to-peer services. The study should present the flaws in some of todays services and suggest improvements. File sharing, the most well known service, will be the service receiving the most attention. The requirements of the thesis can be summarized as follows.

- Investigate existing peer-to-peer services and analyze their performance, especially file sharing systems should be considered.
- Identify the key components in a generic peer-to-peer system, their responsibilities and interaction.
- Design a framework from which efficient peer-to-peer systems can be built.
- Investigate file sharing and discuss protocol performance related topics such as:
  - caching methods
  - hierarchical aggregation
  - specialized directories
- Look at other applications and frameworks for peer-to-peer services and see how they can benefit from the earlier work.

1.2 Investigate existing services

To understand what peer-to-peer is all about it is important to provide a survey of existing services. Since most of the thesis will focus on file sharing, it is the service that will be given the most attention. Several file sharing services and their protocols will be presented. The study of these services will

also result in an understanding on what is needed to construct a working file sharing system that uses peer-to-peer technology.

Three protocols have been chosen. Gnutella is a well-known protocol that has received a lot of press and it will also receive attention in this thesis. P-Grid is another protocol that can be used to locate files in a peer-to-peer environment. It uses a distributed search tree and provides some interesting ideas on how protocols can be designed.

Finally FastTrack is a another file sharing protocol that is widely used. It cannot be studied directly since no specification or reasonable documentation is available. Instead a project called giFT that strives to provide a similar protocol known as OpenFT will be investigated along with the giFT project itself.

1.3 Analyzing file sharing protocols

The Gnutella protocol has become the de-facto standard protocol for peer-to-peer based file sharing. Since this protocol is widely used an in depth analysis is in order. The flaws of Gnutella will be explored and possible solutions will be discussed.

P-Grid and OpenFT will also be analyzed in a similar fashion and their flaws will be discussed, along with suggestions on how they can be improved.

1.4 Constructing a peer-to-peer framework

Based on the studies of the file sharing protocols and related projects a more efficient protocol is presented. In addition to the protocol proposal a design of a peer-to-peer framework is outlined. The framework should be general enough to work with most existing services, but also be easy to use and extend.

The framework is then implemented and, if there is enough time, a suitable protocol plug-in is implemented for testing purposes. Suitable choices would be the efficient protocol presented earlier and a Gnutella plug-in. The Gnutella plug-in would make it a lot easier to verify how well the framework actually works in an environment of thousands of peers.

1.5 Other applications and frameworks

File sharing is the most commonly used application, but to make the thesis complete a look at other applications is important. Identifying if other services have similar problems and if they can be solved in a similar manner makes up the final part of this thesis. This should include for example Grid computing and Instant Messaging.

Some attempts to generalize peer-to-peer services have been attempted, for example the JXTA project by Sun Microsystems. Such frameworks for peer-to-peer services should be investigated to see how well they solve the problems encountered. It is also useful to compare these attempts with the framework designed earlier.

Chapter 2

Introduction

This chapter introduces the necessary terminology and historical ideas that provide the foundation for peer-to-peer networking. Common problems encountered by peer-to-peer services are identified. Some aspects on todays network architecture that contributes to these problems are also presented.

2.1 History

The history of the *Internet* began in the United States in the late 1960's. Originally named *ARPANET*, the Internet was primarily used to share computing resources between university campuses. These campuses were at the time already independent computing sites and the purpose of the ARPANET was to connect these sites using an architecture where all hosts were equals. With the terminology used today this network arrangement, where all hosts are considered to be equals, is called *peer-to-peer* [Ora01].

In the early days the Internet was much more open than it is today. Security was not a big concern and connections could be made directly to any host connected to the Internet. The increasing security concerns has, besides better protection, led to difficulties for efficient use of certain services. For more details see section 2.3 on page 5.

2.1.1 Early services

Although the initial network architecture was peer-to-peer based the earliest applications used extensively, FTP (File Transfer Protocol, [FTP85]) and *Telnet* ([Tel83]), were based on the concepts of *client* and *server*. A server is a host providing a certain service and a client is a host using this service.

These applications were client/server applications, but the idea was to allow all hosts to work both as clients and servers. Any host accepted FTP and Telnet connections and were also able to connect to any other host. This made the Internet as a whole work as a peer-to-peer network.

Client/server protocols are usually based on the client sending a *request* to the server. The server manages the request and sends a *response* (sometimes called a *reply*) back to the client. Telnet, FTP and web browser clients work in this fashion. There are many more client/server based applications available and an exhaustive list would occupy most of this report.

2.2 Peer-to-peer related services

Most people feel that peer-to-peer networking is something new, but it is not. As mentioned earlier, the early Internet was a peer-to-peer network where all hosts were equals. There has also been several applications that have worked in a manner resembling todays peer-to-peer systems, although they have not been treated as such by most people.

These services are not as strictly distributed peer-to-peer systems as for example Gnutella, but instead they use a hierarchical architecture leading to better performance.

2.2.1 Usenet

News has always been one of the most important network services although other more popular services have arrived over the years. Since the late 1970's, *Usenet* news has been an important part of the Internet and is referred to as "the grandfather of today's peer-to-peer applications" [Ora01].

Originally Usenet used UUCP (Unix-to-Unix copy protocol). This protocol allowed one Unix machine to connect to another, exchange files and disconnect. In this fashion Usenet used UUCP to exchange messages within a set of topics [TGO92].

The growth of the networks, the number of topics and the extensive growth of TCP/IP has led to that the Usenet uses NNTP (Network News Transfer Protocol, [NNT86]) instead of UUCP. This new protocol allows Usenet machines to discover new newsgroups and exchange messages in each group.

Usenet uses a *semi-distributed* architecture, where a local system administrator controls what news channels should be stored at the local news server. Each server is connected to a number of other servers. When a user sends a news posting it arrives at the local news server. The local news server then does the following for each of the servers to which it is connected:

1. Examine the posting and compare it to the preferences set by the server. If the server is not interested in this type of news, proceed with the next server.

2. Ask the server if it wishes to receive the posting.

3. If the server accepted the posting, send it to the server.

The first step makes it possible to discard postings that are considered unsuitable at an early stage. A news server administrator perhaps considers adult content and binary content (such as compressed archives and applications) to be unsuitable. It is also possible to set limits on how old the posting could be and how large it may be.

The Usenet uses *push* technology to transfer news postings. This is more efficient than having servers monitor each other for new postings. Most Usenet servers accept postings to more groups than they actually store. These postings are only accepted to be forwarded further, which is necessary if all servers that want to store the posting should receive it.

There is no guarantee that a posting will reach all servers that want it. If for example the local server is connected to a set of servers that all deny the posting it will never leave the local server.

2.2.2 Domain Name System

In the early days of the Internet all hosts stored a file named `hosts.txt` where all existing domain names were mapped to the corresponding IP *addresses*. A *domain name* is a more human friendly representation of an IP address, such as `www.ietf.org` representing the web server of the Internet Engineering Task Force (IETF) organization. At the time of writing this corresponded to the IP address 4.17.168.6, which most likely is harder to remember. Another benefit of using human friendly names is that the IP address corresponding to a certain name could be changed without affecting the users, as long as the human friendly name remains the same [AL01].

Having all computers on the Internet store information about all other hosts was not feasible in the long run, and in 1983 the DNS (*Domain Name System*, [DNS87]) was created. The DNS uses a hierarchical system of names, which has allowed it to withstand the immense growth of the Internet without seriously reduced performance.

The use of hierarchical distribution of knowledge turns out to be a clever way to maintain good performance despite constantly increasing demand. For this reason hierarchical structures are worth investigating for peer-to-peer services used today. More details on how DNS works will be presented later when hierarchical methods are discussed in chapters 4 on page 23 and 6 on page 39.

2.3 Common problems

With the original appearance of the Internet, where most hosts supported the same services, peer-to-peer applications could be developed fairly easy. At that time network performance was limited, basically eliminating the use of such applications.

2.3.1 Firewalls

When the networks grew larger administrators became more concerned with security and started to protect their networks. At this point many administrators chose to setup a *firewall* protecting a local area network from the surrounding Internet. This causes problems, since the hosts within a protected local area network are not equal to hosts that are not protected by firewalls. It is usually impossible to connect to a host behind a firewall using some arbitrary port number. This problem is henceforth simply referred to as the *firewall problem*.

There are also people who install firewall software on their machines. Some of these firewalls are hard to configure in a way that makes them work with certain peer-to-peer services, even for outgoing connections.

It is hard to provide statistics on how common this problem is, but [LIM03a] maintains some statistics on the number of hosts currently connected to the Gnutella network. These statistics show that only about 20% of the total number of nodes are accepting connections. This is not due to firewalls alone, but they are certainly a contributor to this low number of hosts that accept connections.

2.3.2 Port abuse and pushing

Since many firewalls allow connections to hosts on the local area network if that connection is made to a certain port (usually port 80, the port used by HTTP) many peer-to-peer applications (and other network applications as well) use port 80 for many other things than it was originally intended to be. This is sometimes referred to as the *abusing port 80 problem*.

The most common technique to bypass the firewalls though is *push* techniques. If a host behind a firewall has opened a connection to some host outside the firewall, the host outside the firewall can work as a relay. If we ask the relay to forward a push message to the host behind the firewall, the host behind the firewall probably connects to us when it receives the message. The only problem is that we first have to find the host which already has an open connection to the host behind the firewall.

2.3.3 Dynamic host addresses

As the networks grew even further the address space of IPv4, (Internet Protocol version 4, [IP81]) started to become exhausted. Since many ISPs (Internet Service Providers) were assuming that their customers spent their time on the Internet downloading content, and not uploading, permanent IP addresses were not necessary. This caused the DHCP (Dynamic Host Configuration Protocol, [DHC93]) to appear. This protocol automatically distributes IP addresses to the connected hosts. This allowed ISPs to reuse addresses more easily, and only maintain a pool of addresses instead of assigning a unique address to each host on the network. This problem is from now on referred to as the *dynamic* IP *address problem*.

2.3.4 Network Address Translation

Another invention to prevent the addresses in IPv4 to run out was NAT (Network Address Translation, [NAT00]). NAT allows a router to act as an address translation proxy for an entire network. This means that the hosts inside the network could use addresses used by some other network internally, but when one of these hosts wants to access an external resource the NAT router replaces the internal address with an external address. When a response arrives at the NAT router it translates the external address back to the internal address and sends the response to the appropriate host on the network. This problem is referred to as the NAT *problem*.

2.4 Examples on different services

To illustrate the differences between various services three different categories are presented. The client/server example shows how a web page containing embedded images is downloaded by a web browser. The centralized peer-to-peer example illustrates how Napster was used to locate and download an MP3 music file. Finally a decentralized system is used to illustrate how a file can be found and downloaded using the Gnutella protocol. A study of different categories of peer-to-peer systems is also presented in [BWDD02]. For a more complete view on the peer-to-peer field [Ora01] or [MKL+02] provides surveys over several aspects of the peer-to-peer area.

2.4.1 Client/server: browsing the web

Figure 2.1 shows a typical session for downloading a web page containing some (in this case two) embedded images. Some details concerning the establishment of TCP connections are left out. Some details concerning HTTP (Hypertext Transfer Protocol, [HTT99]) are mentioned. The following steps occur:

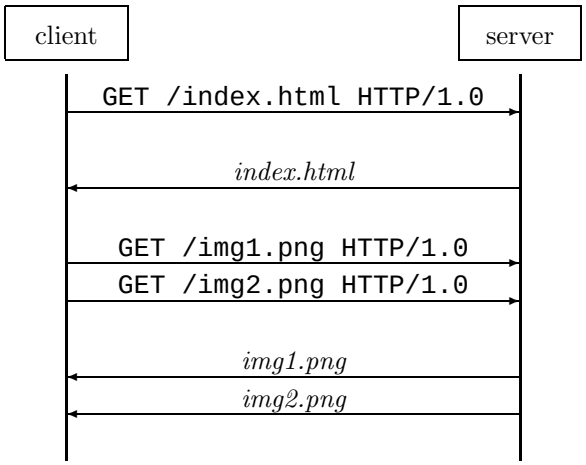

Figure 2.1: Client requesting a document from a web server

1. The client (in this case the web browser) tries to *connect* to the web server, usually using port number 80.
2. The server *accepts* the connection.
3. The client *requests* a web page (in this case `index.html`). This is performed by using the `GET` command of HTTP.
4. The server *responds* by sending the requested web page. The server also sends (as a part of the reply) an indication that the operation was successful.
5. The client parses the response and detects that there are some embedded images on this web page.
6. The client issues *requests* for these images to the server.
7. The server *responds* by sending the images to the client.
8. The client *closes* the connection to the server.

The older version of HTTP, version 1.0 ([HTT96]), stated that the client closes the connection between each request and hence needed to reestablish the connection with the server to request the embedded images. This made the protocol very simple, but somewhat inefficient. The later version of HTTP, version 1.1 ([HTT99]), allowed clients to maintain a session with the server, making it possible to request the embedded images using the same connection.

2.4.2 Centralized peer-to-peer: Napster

Figure 2.2 shows a Napster client locating and downloading a music file from another host on the Internet. The following steps occur:

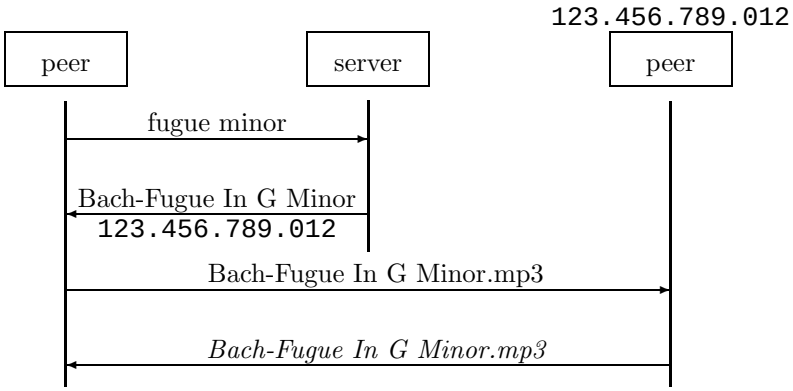

Figure 2.2: Napster client querying directory server and downloading file

1. The client tries to *connect* to the Napster directory server.
2. The directory server *accepts* the connection.
3. The client sends a *query* to the directory server describing the music file wanted. This could for example include the name of the song, the artist performing it or the name of the album where it resides.
4. The directory server processes the query and sends a *response*, containing the hosts that have music files matching the query, back to the client.
5. The client *disconnects* from the directory server.
6. The client *connects* to one or more of the hosts that have the music file.
7. Hopefully at least one of the hosts *accepts* the connection.
8. The client *requests* the music file.
9. The other host (or possibly hosts) *respond* by sending the music file.
10. When the client has downloaded the file it *disconnects*.

The most important part of the Napster system is the directory server. It contains a large database with available music files. A host connecting to the Napster directory server adds descriptions for all its' music files to the database. Queries to the directory server can then be processed efficiently. To make good use of bandwidth the actual download takes place directly between the client and the host storing the file.

2.4.3 Decentralized peer-to-peer: Gnutella

The performance of Napster is superior to that of Gnutella because a central directory server is available. Since legal matters have basically stopped the use of Napster another approach was needed. Both Napster and Gnutella have been used to distribute multimedia illegally over the Internet, but since there is no central point of connectivity in Gnutella it is much harder to shut down.

A host in a Gnutella network is commonly referred to as a *servent* (note the spelling). This name comes from combining the words server and client. Figure 2.3 shows when a servent wishes to locate a file and download it. The figure is somewhat simplified since it does not show the actual download. Three hosts reply to the query and the host issuing the query can contact either one, or all of them if it supports *swarm downloading*. The following steps occur:

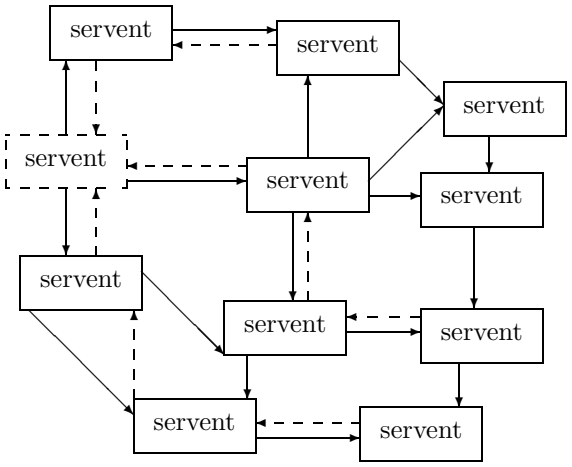

Figure 2.3: Gnutella servent performing a query

1. The servent *contacts its neighbours* in the Gnutella network and sends the *query* to these hosts.
2. The neighbours *process* the query to see if they have anything that matches the query. If they have, they send a *response*.
3. The neighbours send the query to their *other neighbours*.
4. These neighbours-of-neighbours repeat the same steps performed by the neighbours.
5. After a certain number of steps the query stops propagating through the network, due to the expiration of a TTL (time to live) counter.
6. If the initial servent receives responses it can *connect* to the host storing the file and request the file using a HTTP `GET` command.
7. The host storing the file *accepts* the connection and *responds* with the requested file.

Chapter 3

Gnutella

Gnutella, one of the most common protocols for fully distributed file sharing, is presented in this chapter. All the necessary details of the protocol are presented.

3.1 Introduction

Gnutella is a *file sharing protocol*. Using a Gnutella client files shared by a certain host can be located and downloaded by another Gnutella client. The protocol has a simple structure and uses broadcasts to locate files. The material in this chapter is based on the Gnutella Protocol Specification 0.4 ([Gnu00]) and the 0.6 draft ([Gnu02]) unless specified otherwise.

There are many applications available that use Gnutella. Some of the most widely used Gnutella clients are the following.

BearShare is a Gnutella client for Windows being updated frequently and containing the latest features of Gnutella clients [BEA03].

Limewire is a Java based, open source Gnutella client. Contains the latest features in the Gnutella field and runs on most platforms [LIM03b].

GTK Gnutella is a client written in C using GTK, the GIMP Toolkit. It is an open source client, that lacks some of the newest features, but still is useful [GG03].

These and some other Gnutella clients can be found through the website `http://www.gnutelliums.com`, where clients for several platforms are listed.

3.2 Protocol messages

The Gnutella protocol is based on maintaining TCP connections to a number of other Gnutella hosts. These hosts are from now on simply referred to as *neighbours*. In order to find new neighbours a Gnutella servent broadcasts a `Ping` message. If a Gnutella servent receives a `Ping` message it responds with a `Pong` message. A host on the Gnutella network is called a *servent*.

A servent issues a query by sending a `Query` message to all its neighbours. The neighbours pass this message on to their other neighbours and so on, for

a certain number of steps defined by the TTL (Time To Live) field in the message header.

A servent receiving a query checks if it has something that matches this query. If it does, it responds with a `QueryHit` message, but still passes the `Query` message on to its other neighbours.

If a servent receives a `QueryHit` message from a host that does not support incoming connections a `Push` message is sent to that host. This message causes the host holding the file in question to open the connection. This usually makes it possible to bypass firewalls.

The Gnutella protocol, and many other protocols for that matter, exploit what is known as the "small world" phenomenon. This basically says that whereever you want to go you do not have to go very far. In more networking related terms this means that number the of hops you need to take to find what you are looking for is not as many as one might think. More details on this can be found in [Ora01] or [JAB01].

The remainder of this chapter describes the components of the protocol more in detail. The descriptions are based on both the Gnutella protocol version 0.4 and the Gnutella protocol draft version 0.6. If no version number is given a certain description applies to both versions.

3.2.1 Establishing connections

When a servent C has found another Gnutella servent S and wishes to open a connection the following steps occur

Gnutella 0.4

1. C sends the string `GNUTELLA CONNECT/0.4\n\n` where `0.4` is the protocol version and `\n` denotes the newline character (ASCII character 10)
2. S responds, if it chooses to accept the connection, with `GNUTELLA OK\n\n`
3. Connection is established and C and S can exchange messages

Gnutella 0.6

1. C sends the string `GNUTELLA CONNECT/0.6\r\n` where `0.6` is the protocol version and `\r` denotes the carriage return character (ASCII character 13)
2. C sends capability headers (not including vendor specific headers), each terminated by `\r\n`, with an extra `\r\n` at the end
3. S responds with the string `GNUTELLA/0.6 200` *`string`*`\r\n`. The *`string`* should be `OK`, but clients are advised to only check the `200` code
4. S sends all its headers in the same format as in step 2
5. C responds with the string `GNUTELLA/0.6 200 OK\r\n`, as in step 3 if the client, after parsing all headers sent by S, still wants to connect.

Otherwise a reply containing an error code is sent and the connection is closed

6. C sends any vendor specific headers to S in the same format as in step 2

7. Connection is established and C and S can exchange messages

3.2.2 Message header

Each Gnutella message has a header. This header contains information describing what kind of message it is, how much further it should be sent and a unique identifier for the message. All fields in the message header are in *network byte order* (big endian). Table 3.1 shows the structure of the message header.

Bytes	Description
0–15	Message ID (globally unique)
16	Payload type
17	TTL (Time To Live)
18	Hops
19–22	Payload length

Table 3.1: Gnutella message header fields

The fields in the Gnutella message header have the following meaning and semantics.

Message ID is a 16-byte string that is globally unique and identifies a message on the network. Gnutella version 0.6 states that byte 8 (if bytes are numbered 0–15) should be all 1's to indicate that the id belongs to a modern servent. Gnutella version 0.6 also states that byte 15 should be zero, since it is reserved for future use.

Payload type denotes the type of message. It should have one of the values shown in table 3.2.

Value	Message type
0x00	`Ping` message
0x01	`Pong` message
0x02	`Bye` message (only Gnutella 0.6)
0x40	`Push` message
0x80	`Query` message
0x81	`QueryHit` message

Table 3.2: Gnutella message header payload types

Time to live holds the number of times the message will be forwarded by Gnutella servents before is is removed from the network. Each servent decrements the TTL before sending the message to another servent. When the TTL reaches 0, the message will not be forwarded further.

Hops contains the number of hops the message has been forwarded before reaching the current servent. As a message is passed from servent to servent the TTL and Hops fields satisfy the following condition.

$$\text{TTL}_0 = \text{TTL}_n + \text{Hops}_n$$

In this equation TTL_0 denotes the initial TTL (which usually is 7) and TTL_n and Hops_n is the value of TTL and Hops after n hops.

Payload length contains the length of the message following the message header. No padding is used. Messages should not be larger than 4 kB.

3.2.3 `Ping` message

`Ping` messages do not have any payload. The message header contains a payload type of 0x00 and a payload length of 0x00000000. Gnutella version 0.6 allows `Ping` messages to have an extension block defined using GGEP (Gnutella Generic Extension Protocol). Servents are recommended to implement GGEP.

`Ping` messages are broadcasted to all reachable hosts. The TTL field limits the *horizon* of the broadcast. The horizon is the edge where messages are discarded due to expiration of the TTL. No TTL should exceed 7 in order to limit network overhead.

3.2.4 `Pong` message

`Pong` messages contain information about a Gnutella host and are sent as response to an incoming `Ping` message. The message payload type is 0x01 and the message has fields shown in table 3.3. All fields are in network byte order (big endian), except the IP address in Gnutella version 0.4 which is in little endian byte order.

Bytes	Description
0–1	Port number
2–5	IP address
6–9	Number of shared files
10–13	Number of kilobytes shared
14–	Optional GGEP extension block

Table 3.3: Gnutella `Pong` message fields

3.2.5 `Query` message

`Query` messages, with a payload type of 0x80, are broadcasted to all neighbour nodes in the Gnutella network. Messages larger than 256 bytes may be discarded to lower the load on the network. A `Query` message has the fields shown in table 3.4 on the next page.

The fields in the `Query` message have the following meaning and semantics.

Bytes	Description
0–1	Minimum speed, kb/s
2–	Null terminated search criteria
Rest	Optional extensions

Table 3.4: Gnutella `Query` message fields

Minimum speed is the lower bound of connection speed a servent accepts. A servent receiving a `Query` message should only respond to the querying servent with a `QueryHit` if it is able to communicate at least at this speed.

Search criteria is a string of keywords. Servents should only respond with files matching *all* keywords. Matching should be case insensitive. GGEP extensions may be used to specify an alternate matching procedure, such as interpreting the search criteria as a regular expression, but servents can never be sure that other servents understand these GGEP directives.

A `Query` message with a TTL field of 1, a hops field of 0 and a search criteria of exactly *four spaces* is used to index all files a host is sharing. If necessary several `QueryHit` messages may be sent. Servents should reply with all its shared files, unless it considers it harmful for privacy or bandwidth.

Optional extensions is only available in Gnutella version 0.6. Allowed extension types are HUGE (Hash/URN Gnutella Extensions), XML or GGEP. For more information on these extensions see [Gnu02].

3.2.6 `QueryHit` message

The `QueryHit` message, with a payload type of 0x81, is sent in response to a `Query` message when its search criteria matches one or more files shared by the servent. The fields of a `QueryHit` message are shown in table 3.5. All fields are in network byte order (big endian).

Bytes	Description
0	Number of hits (elements in result set)
1–2	Port number of responding host
3–6	IP address of responding host
7–10	Speed of responding host, kb/s
11–	Result set

Table 3.5: Gnutella `QueryHit` message fields

Each element in the result set of a `QueryHit` message has the fields shown in table 3.6 on the next page. It has quite an odd structure with fixed length fields at the beginning and end, with variable length fields in the middle.

The extension blocks, EQHD block and private vendor specific fields are part to the Gnutella 0.6 draft. The EQHD block contains some vendor information structured in the fields shown in table 3.7 on the following page. For additional details see [Gnu02].

Bytes	Description
0–3	File index (unique)
4–7	File size in bytes
8–	File name, null terminated
x–	Extension blocks (HUGE, GGEP, plain)
y–	Recommended EQHD block
z–	Private vendor specific data
Last 16	Responding servent identifier

Table 3.6: Gnutella `QueryHit` result set fields

Bytes	Description
0–3	Vendor code (four case insensitive characters)
4	Open data size
5–	Open data. Flags describing servent capabilities

Table 3.7: Gnutella `QueryHit` EQHD block

3.2.7 `Push` message

The `Push` message is used to make it possible to download data stored on a host residing behind a firewall. Instead a servent receiving a `QueryHit` from a servent that does not accept connections, issues a `Push` message. When a `Push` message is received a servent should act if and only if the servent identifier field contains the receiving servants identifier. `Push` messages are routed back the same way the `QueryHit` message traveled, but in the opposite direction.

The fields in a `Push` message are shown in table 3.8. For more detailed descriptions see either [Gnu00] or [Gnu02].

Bytes	Description
0–15	Servent identifier of host having file
16–19	Index of file to push
20–23	IP address of host wanting file
24–25	Port number to push to
26–	Optional GGEP extension block

Table 3.8: Gnutella `Push` message fields

3.2.8 `Bye` message, version 0.6

The `Bye` message was introduced in the Gnutella 0.6 draft and is used to inform the servent, or servents, a node is connected to that the node is closing the connection. Since implementing this message is optional a special header must be sent during the connection handshake.

```
Bye-Packet:  0.1
```

Servents must not send `Bye` messages to hosts that has not indicated that they support this message. The TTL field in a `Bye` message must be set to 1 to avoid accidental propagation.

Upon receiving a `Bye` message a servent closes the connection in question immediately. The servent sending the message must wait a few seconds for the remote servent to close the connection before closing it. No data may be sent after the `Bye` message. The `Bye` message has the fields shown in table 3.9.

Bytes	Description
0–1	Code (as classified by SMTP)
2–	Description string, null terminated

Table 3.9: Gnutella `Bye` message fields

The code stored in the first two bytes correspond to return codes specified by SMTP (Simple Mail Transfer Protocol, [SMT01]). For example 200 means that everything is ok and 502 means that the send queue became full. For details on the semantics of individual codes see [Gnu02].

3.3 `Ping` and `Pong` optimizations

Since the original protocol specification ([Gnu00]) used network resources very inefficiently [Gnu02] states some optimizations for `Ping` and `Pong` messages. To avoid unnecessary propagation of `Ping` broadcasts caching of `Pong` messages is deployed.

If a Gnutella host conforms to the following requirements it is considered `Pong` caching compatible and must provide the following header when establishing a connection. Possibly a higher number can be given if the caching scheme has a higher version number.

```
Pong-Caching:  0.1
```

1. All `Pong` messages sent should have a high probability of referring to connectable hosts and provide a good distribution of hosts across the network.

2. The bandwidth used by `Ping` and `Pong` messages should be minimized.

3. If a `Ping` message is received, with TTL greater than 1 and it was at least one second since another `Ping` was received on that connection, a servent must (if possible) respond with a number of `Pong` messages. The number of messages sent in response may vary but 10 is a reasonable number according to [Gnu02].

4. An incoming `Ping` message with a TTL field of 1 and a Hops field of 0 or 1 must always be replied to with a `Pong` message, since the `Ping` is used to probe the remote host of a connection.

5. An incoming `Ping` message with a TTL field of 2 and a Hops field of 0 is called a "Crawler `Ping`" and is used to scan the network. `Pong`

messages should be sent in reply listing the host receiving the `Ping` and all its' neighbours.

3.3.1 `Pong` caching schemes

Some ideas on how to implement the `Pong` caching in practice has been proposed. One caching scheme is presented in [Gnu02] and another scheme is presented in [RF01]. The [Gnu02] draft, [RF01] and some other suggested schemes are available at [RFG03]. This section describes the most important parts of these proposed caching schemes.

All caching schemes are based on the observation that the initial implementations of the Gnutella protocol consumed a lot of bandwidth. Closer examination revealed that connection management with `Ping` and `Pong` messages were using more than 50% of the bandwidth. The main reason is of course the broadcasting of `Ping` messages that propagate through the network consuming lots of bandwidth. Similar results have been presented in [RIF02] and [Rip01].

To reduce the effects of a `Ping` broadcast `Pong` caching is deployed. Each Gnutella host stores all `Pong` messages received in a local cache. When receiving a `Ping` message a number of messages from the cache are chosen and returned. When returning a cached `Pong` message the message identifier is of course changed to correspond to the incoming `Ping` message. A cached `Pong` message is considered valid for a couple of seconds. Suggestions between 3 and 15 seconds have been proposed.

The `Pong` caching schemes only apply to `Ping` messages with a TTL greater than 2 and a hop count greater than 0. Otherwise the incoming `Ping` message is managed as described in section 3.3 on the page before.

The pseudo code shown in figure 3.1 on the following page is a slightly modified version of one of the schemes presented in the archive of Gnutella protocol proposals at [RFG03].

Incoming `Ping` messages are answered directly if the local cache contains enough items. A certain number of items are selected from the cache and returned. If the cache does not contain enough items to answer the `Ping` directly, a regular broadcast is performed.

Incoming `Pong` messages are stored in the cache. The messages stored in the cache are removed after a certain amount of time, keeping the entries in the cache fresh, and hence more reliable.

3.3.2 `Ping` multiplexing

Another optimization that can be performed is `Ping` multiplexing. If a host receives a `Ping` message, broadcasts it further and, before receiving enough `Pong` messages to fill the cache, a `Ping` message from another host is received. Instead of issuing another `Ping` broadcast the host awaits the arrival of the expected `Pong` messages, stores them in the cache, and responds to both of the hosts that sent `Ping` messages.

```
AddPongEntry(pongmsg) {
    timestamp = CurrentTime()
    CacheInsert(pongmsg, timestamp)
}

RemoveExpiredPongs() {
    foreach pong in CACHE
        if (CurrentTime() - pong.timestamp > TIMEOUT)
            RemovePong(pong)
}

HandlePing(socket) {
    pingmsg = ReadPing(socket)
    if (pingmsg.ttl < 2)
        ...
    else
        RemoveExpiredPongs()
        if (CACHE.size() >= THRESHOLD)
            pongs = SelectPongsFromCache(THRESHOLD)
            foreach pong in pongs
                SendPong(pong, socket)
        else
            BroadCastPing(pingmsg)
}

HandlePong(socket) {
    pongmsg = ReadPong(socket)
    ...
    if (pongmsg.hops > 0)
        AddPongEntry(pongmsg)
}
```

Figure 3.1: Pseudo code for a Pong caching scheme

3.4 Ultrapeers and routing

An attempt to reduce the network overhead caused by the initial version of the Gnutella protocol is the introduction of higher level nodes called called ultrapeers. An *ultrapeer* is a powerful node that maintains many connection to non-ultrapeer nodes, from now on called *leaf nodes*, and a small number of connections to other ultrapeers.

Ultrapeers shield leaf nodes from almost all `Ping` and `Query` traffic. Mainly two suggested approaches of managing ultrapeers and leaf nodes have been proposed.

Indexing means that ultrapeers periodically send an indexing query to each leaf node and the leaf nodes respond with a message naming all shared files. The ultrapeer uses these to build an index of the leaf nodes, which is then checked when a query arrives.

Bit vector matching means that each leaf node constructs an array based hash table from words that causes matches for a certain shared resource. Stored in the hash table is simply a flag (or bit) indicating "present".

The bit vector is then sent to the ultrapeer. The ultrapeer can then check the words of an incoming query, by running them through the same hash function, and forward the query if the bit vector indicates that matching resources are present in the leaf node. This method is commonly referred to as DHT-based routing, of which two variants is discussed in [HHH+02] and [Pin01].

Some of the benefits of bit vector matching is that the bit vector can be compressed and has better support for incremental updates. It is easier to implement the hash check then to efficiently check an index.

Ultrapeers communicate with other ultrapeers using the broadcast scheme initially presented by Gnutella. It is also possible to have ultrapeers perform additional levels of bit vector passing, creating several layers of ultrapeers, but this has not been implemented.

A problem with ultrapeers is that the risk of receiving false answers to a query increases. Since an ultrapeer U answers a query by saying that servent S has a certain list of hits the node issuing the query relies on the behaviour of U and that S has not disconnected or changed its set of shared files without notifying U.

3.5 Transferring files

Upon receiving a `QueryHit` message a servent may initiate a HTTP connection with the host holding the file. The `GET` command issued by the servent contains the *index* of the requested file, as specified in the *result set* of the `QueryHit` message. Following the index is the name of the file. Next follows a small example.

Assume that a `QueryHit` message contained at least the entry shown in table 3.10 on the next page.

A download request for the file `gnutella_0.6_draft.txt` would be initiated as follows:

Index	Size	Name
1342	78835	gnutella_0.6_draft.txt

Table 3.10: Example of a QueryHit result

```
GET /get/1342/gnutella_0.6_draft.txt HTTP/1.1\r\n
User-Agent: Gnutella\r\n
Host: 123.456.789.012:6346\r\n
Connection: Keep-Alive\r\n
Range: bytes=0-\r\n
\r\n
```

The servent receiving this request responds with HTTP compliant headers such as:

```
HTTP/1.1 200 OK\r\n
Server: Gnutella\r\n
Content-type: text/plain\r\n
Content-length: 78835\r\n
\r\n
```

The Range header can be used to request a file segment, if this header contains a range other than the whole file. This allows a Gnutella client to download different parts of the same file from different hosts, which can reduce the download time substantially. This technique is called *swarm downloading*. If a segment is requested the HTTP response headers could look as follows:

```
HTTP/1.1 206 Partial Content\r\n
Server: Gnutella\r\n
Content-Type: text/plain\r\n
Content-Length: 24761\r\n
Content-Range: bytes 32104-56864/78835\r\n
\r\n
```

3.6 Metadata and rich queries

To make it possible to perform detailed searches *metadata* that classifies the shared files are necessary. For example, a book has an author, a publisher, an edition number, a publishing year and an ISBN number. A music file is made by some artist, has a title, resides on some album, was recorded in a certain studio, released a certain year on some record label and so on.

Metadata is sometimes available within the file itself. For example, MP3 music files often contain an ID3 tag at the end. This tag provides metadata for that file, such as title, artist, album, year and genre. Since all file types do not have this ability an external metadata scheme has been proposed, and implemented in modern Gnutella clients.

The proposed scheme uses XML to encode a rich query, but is still able to maintain compatibility with older clients. A Query message containing a rich query contains an XML block after the null terminated query string. This block is also null terminated, and will be discarded by older clients. The format of a rich Query message is outlined in table 3.11 on the following page.

Bytes	Description
0–1	Minimum speed, kb/s
2–	Null terminated search criteria
x–	Null terminated rich XML metadata search criteria
Rest	Optional extensions

Table 3.11: Rich XML metadata `Query` message

A rich `QueryHit` message contains embedded XML in the EQHD block and private vendor data fields of the `QueryHit` message. The EQHD block and private vendor data has the format shown in table 3.12 for rich replies. See also table 3.5 on page 15 for a description of the `QueryHit` message and table 3.7 on page 16 for a description of the original EQHD block.

Bytes	Description
0–3	Vendor code
4	Public area size (set to 4)
5–6	Public area
7–8	Size of XML payload + 1, m
9–	Private area
x–	Null terminated XML payload, m bytes

Table 3.12: Contents of EQHD block in rich `QueryReply` messages

The size of the private area is calculated by subtracting the size of the XML payload from the total size of the EQHD block.

Chapter 4

P-Grid

P-Grid is a decentralized fully distributed search tree used for query routing. This chapter also contains additional discussions on hierarchical protocols, since P-Grid uses a hierarchical architecture.

4.1 Introduction

The P-Grid peer-to-peer system uses a virtual binary search tree to guide key based searches. It uses completely decentralized algorithms and has received quite a lot of press. The official web site, [PGR03], contains lots of information on the P-Grid architecture and a list of publications.

This chapter describes hierarchical and tree based algorithms that are useful in peer-to-peer contexts and describes the algorithms that P-Grid uses to achieve its result. Some analysis results of P-Grid are presented later in chapter 6 on page 39.

4.2 Hierarchical protocols

Hierarchical protocols is nothing new, but provides an interesting approach to the balance between scalability and performance. The most well known service in use today that uses a hierarchical protocol is DNS. The purpose of DNS is to translate a human friendly domain name, such as `www.ietf.org`, to its corresponding IP address (in this case `4.17.168.6`). The DNS architecture consists of the following:

- Root name servers
- Other name servers
- Clients

The other name servers can also be classified as *authorative* name servers for some domains. More on this later on. As described earlier the early Internet forced all hosts to maintain a copy of a file named `hosts.txt` which contained all necessary translations. As the network grew the size and frequent changes of the file became unfeasible. The introduction of DNS remedied this problem and has worked successfully since then.

4.2.1 An example of a DNS lookup

Assume a host is located in the domain `sourceforge.net`. The following scenario shows what a DNS lookup could look like in practice.

1. If a user on the aforementioned host, in the `sourceforge.net` domain, directs his web browser to `http://www.ietf.org` the web browser issues a DNS lookup for the name `www.ietf.org`.
2. The request is sent to the *local name server* of the `sourceforge.net` domain.
3. The name server at `sourceforge.net` is not able to answer the question directly, but it knows the addresses of the *root name servers* and contacts one of them.
4. There are 12 root name servers (9 in the US, 1 in the UK, 1 in Sweden and 1 in Japan). The root name server knows the address of a name server for the `org` domain. This address is sent in response to the question from the local name server at `sourceforge.net`.
5. The name server at `sourceforge.net` asks the name server of the `org` domain, but it does not have the answer either, but the name server of the `org` domain knows the name and address of the *authorative name server* for the `ietf.org` domain.
6. The name server at `sourceforge.net` contacts the name server at `ietf.org` and once again asks for the address of `www.ietf.org`. This time an answer is found and the IP address `4.17.168.6` is returned.
7. The web browser can continue its work by opening a connection to the correct host.

Note that a question sent to a name server can be either *recursive* or *iterative*. A recursive question causes the name server to continue asking other name servers until it receives an answer, which could be that the name does not exist. An iterative query returns an answer to the host asking the question immediately. If a definite answer cannot be given, suggestions on which servers to ask instead are given.

4.2.2 Caching in DNS

Caching plays an important part in DNS. In the example above the local name server will cache the addresses obtained for the name server of the `org` domain and the `ietf.org` domain as well as the final answer, the address of `www.ietf.org`. This causes subsequent translations of `www.ietf.org` to be answered directly by the local name server, and translations of other hosts in the domain `ietf.org` can bypass the root name server and the `org` server. The translation of an address such as `www.gnu.org` bypasses the root name server and asks the name server for the `org` domain directly.

4.2.3 Redundancy and fault tolerance in DNS

To make DNS fault tolerant any name server can hold a set of entries as the answer to a single question. A name server can answer a question such as "What is the address of `www.gnu.org`" with something like figure 4.1, which provides the names of name servers for the `gnu.org` domain. The results were obtained using the **dig** utility available on most Unix systems. In reality the response is much more compact.

```
;; ANSWER SECTION:
gnu.org.                86385   IN      NS      nic.cent.net.
gnu.org.                86385   IN      NS      ns1.gnu.org.
gnu.org.                86385   IN      NS      ns2.gnu.org.
gnu.org.                86385   IN      NS      ns2.cent.net.
gnu.org.                86385   IN      NS      ns3.gnu.org.

;; ADDITIONAL SECTION:
nic.cent.net.           79574   IN      A       140.186.1.4
ns1.gnu.org.            86373   IN      A       199.232.76.162
ns2.gnu.org.            86385   IN      A       195.68.21.199
ns2.cent.net.           79574   IN      A       140.186.1.14
```

Figure 4.1: Sample response from a DNS query

The example shows that the `gnu.org` domain appears to have five name servers (`NS`), of which four of their addresses are known to us. The question of the address of `www.gnu.org` can be sent to anyone of the four servers. This means that we can receive an answer to our question as long as at least one of the name servers is reachable.

4.3 P-Grid architecture

P-Grid provides a tree-like structure where key based searching can be performed. In terms of bandwidth usage searching scales very well since no broadcasting or other bandwidth consuming activities takes place during searches. Since all searches are key based there are two possibilities:

- Let each host implement the same translation algorithm, that translates a sequence of keywords to a binary key.
- Let another service provide the binary key. This service accepts keyword based queries and can respond with the corresponding key.

As discussed later in chapter 7 on page 61 the second approach is more precise. It is also possible to use a more centralized implementation for such a service. For more details on the possibilities and requirements of such a service see chapter 7 on page 61. From now on we assume that the key is available. The papers and articles published on P-Grid describe an algorithm for the first case. See for example [APHS02].

4.3.1 Peers in the P-Grid network

The peers in the P-Grid network uses a few elements to guide searches. If peer i, henceforth denoted p_i, is a member of the P-Grid network it maintains

the following:

- A k bit binary key $k_i = b_1 \dots b_k$, where $k \leq n$ for some bounded constant n which is the same for all p_i. This key denotes which subset of the key space that the peer is *responsible* for.

  Responsibility means that a p_i should be able to answer queries for keys that begin with the bit pattern k_i.

- An array (indexed from 1) with k elements, each containing a set of addresses to other peers. Index j in this array, $j \leq k$, contains the addresses of peers p_x that had the key $k_x = b_1 \dots b_{j-1} \neg b_j$ the last time peers p_i and p_x met.

  This array is called the *reference table* of the peer and the expression REFS(l) denotes the set of addresses at index l in the array. In P-Grid terminology this is called the *references at level* l.

- A set of data items D that match the key for the peer, and possibly an additional set of data items D' that do not match. The second of these sets are for redundancy purposes, but is not strictly required. The descriptions of the P-Grid system actually vary on the description of data sets. Some descriptions does not even mention that the data set should contain elements matching the key. These differences are discussed further in section 6.3 on page 52.

4.3.2 Key distribution

The distribution of responsibilities in the P-Grid network is critical. Without a well performing algorithm for key and data distribution the efficient search algorithm described later becomes useless. Each peer in the P-Grid network is responsible for a specific *key prefix*. The length of this prefix grows as the peer communicates with other peers in the network.

Initially all peers are responsible for everything, or at least they think they are. If figure 4.2 describes a set of hosts that are all initially responsible for everything and the order of communication is determined by the numbered arrows. Table 4.1 on the following page then shows how the prefixes and reference table that each host is responsible for could change during a sequence of exchanges.

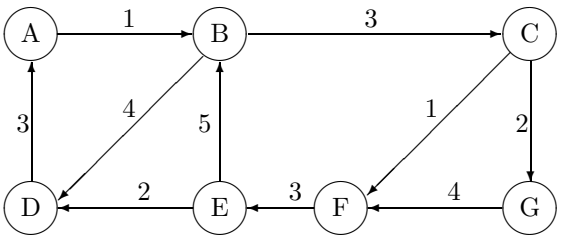

Figure 4.2: P-Grid network connection establishment order

1. During the first step an exchange between A and B, as well as C and F takes place. Both of these exchanges correspond to case 1 in 4.3 on

p	Step 1		Step 2		Step 3	
A	0	1→B	0	1→B	01	1→BF 00→C
B	1	0→A	1	0→A	11	0→AE 10→D
C	0	1→F	0	1→F	00	1→FB 01→A
D			1	0→E	10	0→EAC 11→B
E			0	1→D	0	1→D
F	1	0→C	1	0→C	11	0→CEA 10→D
G			1	0→C	10	0→CEA 11→F
p	Step 4		Step 5			
A	01	1→BF 00→C	01	1→BFD 00→C		
B	11	0→AEC 10→D	11	0→AEC 10→D		
C	00	1→FB 01→A	00	1→FB 01→A		
D	10	0→EAC 11→B	10	0→EAC 11→B		
E	0	1→D	00	1→DBF 01→A		
F	11	0→CEA 10→D	11	0→CEA 10→D		
G	10	0→CEA 11→F	10	0→CEA 11→F		

Table 4.1: Key responsibility distribution in P-Grid

page 28. The `randombit()` function is assumed to return 0 on both occasions.

2. In step 2 E and D performs an exchange, along with C and G that also perform an exchange. The first of these correspond to case 1 and the second correspond to case 2 in figure 4.3 on the next page.

3. During step 3 D and A perform an exchange, which also is performed by B and C, along with F and E. The first of these correspond to case 3, where A directs D to B. When D performs an exchange with B their keys are extended with different bits, which is case 1. If we assume that the exchange between B and C happens a bit later, B will forward C to A since it is case 3. When C and A perform an exchange their keys will be extended due to case 1. The exchange between F and E causes F to issue an exchange with D. This exchange causes F to extend its key (case 2), since D has just extended its key in an exchange with B.

4. In step 4 G and F do an exchange, which is also performed by B and D. In the exchange between G and F their keys are extended since it is case 1. The exchange between B and D does not change anything since no forwarding in case 3 is possible. This is not unexpected since B and D were involved in an exchange in the previous step.

5. In the final step of the example E starts an exchange with B. This is also case 3 and E is forwarded to A. In the exchange between E and A, E extends its key due to case 2.

The `exchange` algorithm used to distribute responsibilities between two peers p_1 and p_2 is shown in figure 4.3 on the following page. It is quite complicated, but simulations have shown that it reaches a stable state in about the same time as the Freenet information sharing system, as presented

in [ACMD+02] and [AHP03]. See section 10.3 on page 113 or [FNP03] and [CMH+02] for more details on Freenet.

The algorithm presented here assumes that p_1 initiates the exchange. The algorithm presented here is a reworked version of the algorithm that closer matches the way it would be implemented (in most cases). For details on the original formulations of the algorithm see [Abe02], [Abe01], [APHS02] or [ACMD+02]. All of these sources present slightly different versions of the algorithm, which makes it a little unclear exactly how the algorithm should be formulated.

```
exchange(p1, p2) {
    master = p1
    slave = p2
    // Make sure p2 has the longest key
    if (length(p1.key) > length(p2.key))
        swap(p1, p2)
    common = length(common_prefix(p1.key, p2.key))
    l1 = length(p1.key) - common
    l2 = length(p2.key) - common
    // Exchange references at common level
    if (common > 0)
        xchg_refs(p1, p2, common)
    if (l1 == 0 && l2 == 0)
        // Case 1: Extend both keys
        if (MYSELF == master)
            b = randombit()
            master.key += b
            sendbit(slave, 1 - b)
            REFS(common + 1) = {slave}
        else
            slave.key += recvbit(master)
            slave.refs(common + 1) = {master}
    else if (l1 == 0 && l2 > 0)
        // Case 2: Extend p1.key, which is shorter
        if (MYSELF == p1)
            p1.key += 1 - bit(p2.key, common + 1)
            REFS(common + 1) = {p2}
    else if (l1 > 0 && l2 > 0)
        // Case 3: Exchange data and tell p2 to do another exchange
        xchg_data(p1, p2)
        if (MYSELF == p1)
            p3 = recvpeer(p2)
            connect(p3)
            exchange(p1, p3)
        else
            sendpeer(p1, REFS(common + 1))
}
```

Figure 4.3: Pseudo-code for responsibility exchange in P-Grid

The algorithm, as it is described in figure 4.3, does not converge to a stable state rapidly enough. By adding a threshold on the minimum number of elements stored in order to extend the key by one bit and a threshold on the number of recursive calls allowed the algorithm reaches a stable state faster. For further performance related details see chapter 6 on page 39.

4.3.3 Searching

Assume that a peer performs a search for key k. To perform the search a connection to a peer p in the P-Grid network is established and the call `pgrid_search(`p`, `k`, 0)` is performed. The function `pgrid_search` is shown in figure 4.4. To understand what is actually happening an example is useful.

```
pgrid_search(p, k, i) {
    remain = substring(p.key, i + 1, length(p.key))
    common = length(common_prefix(k, remain))
    if (p is responsible for this k)
        return p
    if (someone else is responsible)
        k_next = substring(k, common + 1, length(k))
        foreach p_next in REFS(i + common + 1)
            if (online(p_next))
                host = send_search(p_next, k_next, i + common)
                if (host)
                    return host
    return null
}
```

Figure 4.4: Pseudo-code for P-Grid searches

Assume that the network is in the same state as right after the five rounds of exchanges shown in table 4.1 on page 27. If node A issues a search for a key with binary representation 1101 the following steps occur:

1. A detects that the the request needs to be forwarded, and chooses to send the query to node B.
2. B notices that only the first bit matches the two bit prefix that B is responsible for. B forwards the query to node D.
3. D matches its entire prefix 11 and knows it is responsible for answering this query.
4. D checks its depository to see if an item with key 1101 can be found. The answer is sent to B.
5. B forwards the answer to A.
6. If D reported a matching, A can initiate a connection with the host in question and request the item.
7. Otherwise, A could try F instead. There is no point in trying D since it already reported that it did not have a matching element.

For performance related discussions on P-Grid see chapter 6 on page 39.

4.4 Related protocols

There are many possible ways to use key based searching in distributed environments. In [RWE+01] a key based routing and searching architecture

named OceanStore is presented. It uses an architecture named Tapestry to actually implement the searching. The query routing technology in Tapestry is very much similar to P-Grid.

A more exotic approach is presented in [TXM02], where a vector based scheme is proposed. Each document is classified according to a vector and routing is based on the angular distance between two vectors. This is a proposal for the more mathematically inclined, but offers interesting ideas. Among other things mentioned in that paper is the transformation of vectors using matrix decompositions in order to minimize synonymy and noise in document vectors. This would improve the accuracy of searching.

There are also a number of systems that use distributed hash trees, where routing is performed based on running the query through a hashing function and usually performing routing based on a bit vector.

Chapter 5

FastTrack

This chapter discusses the FastTrack protocol used by the KaZaA family of file sharing applications. Since the protocol is a well kept secret most discussions will relate to the giFT project, an open source implementation attempting to provide similar capabilities.

5.1 Introduction

FastTrack is another distributed file sharing protocol used by a number of clients. Unfortunately the protocol is a well kept company secret and details are not available to the public. This chapter presents some of the properties of the FastTrack protocol. Some of the applications using the FastTrack protocol is the following.

KaZaA is the original FastTrack client for Windows [KAZ03].

KaZaA Lite is simply a light weight KaZaA client with no advertisements [KZL03].

Grokster is another similar FastTrack client [GRO03].

iMesh is yet another Windows client using the FastTrack protocol [IM03].

The other part of this chapter is dedicated to the giFT project, an open source project for file sharing. The project also involves the creation of a file sharing protocol similar to FastTrack named OpenFT.

5.2 FastTrack architecture overview

FastTrack uses a *semi-distributed* and *hierarchical* architecture to achieve performance greater than that of Gnutella. Even though Gnutella introduced ultrapeers, FastTrack based clients perform better on most occasions.

Since the protocol is a company secret, where a license needs to be obtained through the company Sherman Networks, only minor details have been available, meaning that the architecture described could have been changed recently. Nevertheless this section contains a description of the architecture

used by FastTrack clients, even though it might be outdated, since it provides a useful insight on how protocols can be designed.

The FastTrack protocol classifies some nodes as *super nodes*. These nodes act as directory servers for other clients and are elected without centralized control. It is certainly possible that more roles exist. There is probably some kind of aggregation between the super nodes as well, but this has not been proven. Attempts at cracking the FastTrack protocol have been made but has failed to break the encryption.

Some FastTrack clients also uses a *reputation* system which encourages users to share files and allow uploads. For example, KaZaA Lite users reputation are reflected by their *participation level*, which is a number that is well encapsulated by encryption. A user starts at participation level 10 and can get a participation level between 0 and 1000. A high participation level means that the client has been connected for long periods of time and allowed many users to benefit from it. User with higher participation level are favored in queuing policies and should receive better *quality of service* (QoS).

5.2.1 Protocol details

FastTrack runs on top of both UDP and TCP. Clients receive fewer packets per minute compared to Gnutella clients. FastTrack does not maintain TCP connections for longer periods of time, unless it is a download or upload. A basic study of the traffic caused by FastTrack clients is presented in chapter 6 on page 39. FastTrack uses a simplified version of HTTP to perform the actual downloads. This makes it possible for users to bypass the regulations set by the client on the maximum number of simultaneous downloads. Earlier versions of FastTrack clients even allowed a user to download files from itself using a web browser and thereby fooling the reputation system to believe that the client had contributed a lot to the network.

5.3 The giFT project

The giFT project was initially a project devised to break FastTrack. The meaning of the abbreviation giFT at that time was Generic Interface to FastTrack or giFT isn't FastTrack. When the giFT project came too close at cracking the protocol, FastTrack changed its encryption making it unfeasible to reverse engineer.

The giFT project changed direction, and meaning of its abbreviation to giFT Internet File Transfer. The aim of this project is to provide a system that can connect to many different networks. The project also involves the development of a new network protocol named OpenFT, superior to FastTrack. At the moment the OpenFT network is the only network that giFT clients are able to connect to.

The giFT framework, often referred to as the giFT *bridge*, is implemented as a separate daemon process to which protocol plug-ins can be added. The bridge provides a generic interface that can be used by client front ends to access any of the networks currently installed. See figure 5.1 for an overview of the giFT bridge. As mentioned earlier, OpenFT is the only available plug-in at the moment.

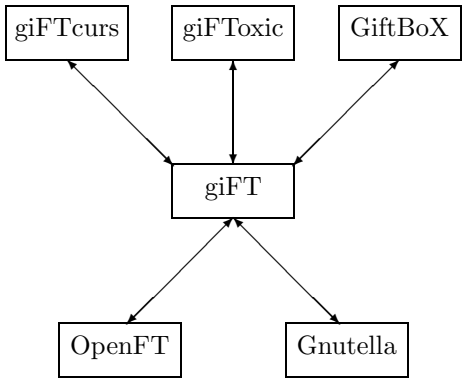

Figure 5.1: The giFT bridge with front- and backends

The OpenFT protocol causes nodes to be classified as one of the following roles. A node is elected as a candidate for a certain role depending on the speed of its network connection, its availability, processing power and available memory. See figure 5.2 on the following page for an example on how the network is structured. At the moment no distributed algorithm appears to be used for role selection. The role is set explicitly when configuring the OpenFT plug-in.

Index nodes are the most reliable hosts, that are available at almost all times. They maintain *indices* of available *search nodes*, collect statistics and monitor the structure of the network.

Search nodes are the servers of the OpenFT network. They maintain *indices* of *files* shared by its' *user nodes*. The default configuration allows a search node to manage information about files stored at 500 user nodes. By default each user node selects three search nodes to maintain their shared file information.

User nodes are the remaining hosts. The client nodes maintains connections to a large set of search nodes. Information regarding the files shared by the client are sent to a subset of the search nodes for incorporation into their file index.

It is possible for a node to be both an index node and a search node at the same time. User nodes maintain connections to a few search nodes (3 by default) as their *parent* nodes. If these parent nodes accept the user node as a *child* node the user node sends their list of shared files to the search nodes. All nodes maintain a registry of available search nodes. When queries are issued the query is sent to the nodes found in the registry. This means that a query is sent to many more nodes than the information regarding a user nodes' shared files. Index nodes are expected to maintain the largest registries of search nodes.

The protocol also includes the exchange of node information, which is used to update the registry of search nodes.

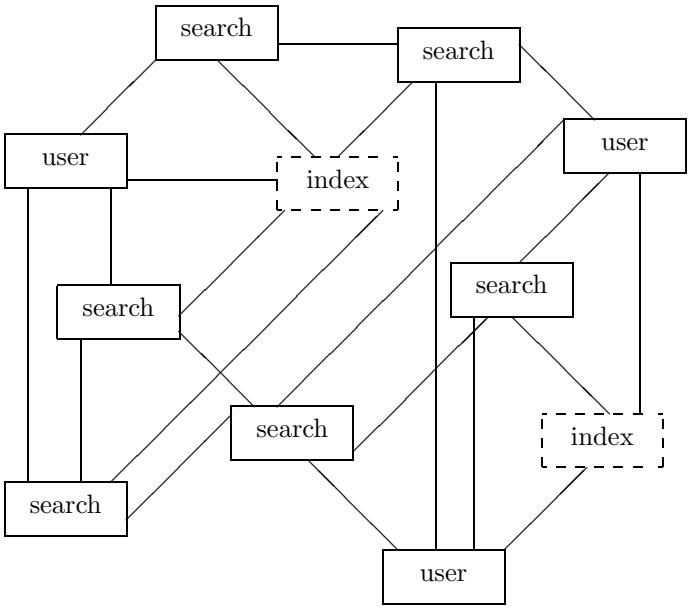

Figure 5.2: OpenFT network structure

5.4 giFT interface protocol

The giFT interface protocol is a more complex protocol than the Gnutella protocol. This comes as no surprise considering the intentions of the giFT project. The protocol describes the communication between clients and the giFT bridge. The bridge than converts the commands to a format appropriate for the network being accessed. The description in this section is a summary of the official protocol specification found at [GIF03].

5.4.1 Protocol format

The protocol is a text based protocol. Messages are called *commands* and the commands have the grammar shown in table 5.1.

cmd	$\longrightarrow$	*name modifier arg keys subcmds ;*
keys	$\longrightarrow$	*key* $\mid \varepsilon$
key	$\longrightarrow$	*name modifier arg*
subcmds	$\longrightarrow$	*subcmd* $\mid \varepsilon$
subcmd	$\longrightarrow$	*name modifier arg* {*key keys subcmds* }
modifier	$\longrightarrow$	[*charseq*] $\mid \varepsilon$
arg	$\longrightarrow$	(*charseq*) $\mid \varepsilon$
name	$\longrightarrow$	an identifier, [A-Za-z_][A-Za-z0-9_]*
charseq	$\longrightarrow$	any characters, but `()[]{};\` needs \ escape

Table 5.1: Grammar for the giFT interface protocol

Whitespace is arbitrary and any sequence of whitespace is equivalent to a

single space. Whitespace is considered a token separator.

The order of keys is not important in a command. An entire command is parsed and the presence of the keys is the important thing, not their order. The *modifier* and *arg* parts can be interchanged and for *subcmd* it is even possible to place the part surrounded by braces before the *modifier* and/or *arg*.

5.4.2 Command summary

To get a quick view of the kind of commands used by the protocol table 5.2 provides a quick summary. It shows what messages should be sent by clients, and what messages are sent from the bridge to the clients. The column named **C** denotes transmissions by the client and **B** denotes transmissions made by the giFT bridge.

Command	C	B	Description
ATTACH	•	•	Register session and reply
DETACH	•		Detach from bridge
QUIT	•		Stop all transfers and detach
SEARCH	•		Issue a search query
BROWSE	•		Gets entire file list from a host
LOCATE	•		Find additional data sources
ITEM		•	Response to the three above
ADDSOURCE	•		Initiate transfer
TRANSFER	•		Cancel or pause transfer
DELSOURCE	•		Delete data source
DOWNLOAD_ADD		•	Initiate download
UPLOAD_ADD		•	Initiate upload
DOWNLOAD_CHANGE		•	Download status report
UPLOAD_CHANGE		•	Upload status report
DOWNLOAD_DEL		•	Finished download
UPLOAD_DEL		•	Finished upload
SHARE	•	•	Manipulate share index
STATS	•	•	Request statistics

Table 5.2: Summary of giFT protocol commands

For more detailed descriptions on the individual commands, their syntax and usage see [GIF03].

5.5 OpenFT network protocol

The protocol used by the OpenFT network plug-in for giFT is a binary protocol. It has a simple packet structure making it easy to parse. At the moment of writing the protocol consists of 28 packet types. There are 14 packet categories and each category contains one request and one response packet. Each packet has the header shown in table 5.3 on the following page.

The implementation actually uses the bits 16–31 of the header for both flags and packet type, but currently only one bit is used to denote a flag. The

Bits	Description
0–15	Length of packet excluding header
16	Set to 1 if the packet belongs to a stream
17–31	Packet type

Table 5.3: Header in OpenFT packets

highest packet type number was at the moment of writing 301. This means that bits 17–23 remains unused and can be used for additional packet types, or additional flags.

As described earlier the OpenFT network divides the nodes in the network into different roles. The user chooses the role it wants, making the assignment static. It can be changed by the user, but there is no distributed algorithm that performs the role assignments. When giFT and OpenFT matures such an algorithm could prove extremely useful to help the system scale better and improve the overall performance. All nodes in the network support the same protocol, and uses the same callback functions when network packets arrive, but the configuration determine their behaviour.

5.5.1 Connection establishment

The OpenFT protocol uses a set of messages to establish a connection. The connection establishment is divided into four stages. Words in parenthesis denotes the packets sent during each phase:

1. Version exchange (`version`)
2. Version acceptance and node information exchange (`nodecap`, `nodelist`, `nodeinfo`)
3. Port test and session request (`session`)
4. Session acceptance (`session` response)

At the moment the connection and session establishment is not considered fully completed. The implementation makes some assumptions regarding the behaviour of the hosts, that needs to be fixed. For well behaving hosts the connection establishment works without any problems though.

5.5.2 Protocol summary

Table 5.4 on the next page gives a summary of the packets used by the OpenFT protocol. As mentioned earlier all packet types have a request and reply packet. The column named **Direction** describe how the message is sent. An arrow to the right denotes a request and an arrow to the left denotes a response. The letters u, s and i denotes user, search and index nodes respectively.

All packets can be classified into a couple of categories, namely:

- Session establishment: `version`, `nodeinfo`, `nodelist`, `nodecap`, `session`

Packet	Direction	Description
`version`	$usi \rightleftarrows usi$	Communicate protocol version and check if it is outdated
`nodeinfo`	$usi \rightleftarrows usi$	Communicate host information such as IP address, port numbers and OpenFT host category
`nodelist`	$usi \rightleftarrows usi$	Communicate information about other nodes
`nodecap`	$usi \rightleftarrows usi$	Communicate host capabilities
`ping`	$usi \rightleftarrows usi$	Used to keep connections alive
`session`	$usi \rightleftarrows usi$	Establish session
`child`	$us \rightleftarrows s$	Request to become child of a search node
`addshare`	$u \rightleftarrows s$	Adds information about a file stored in a user node to the registry of a search node
`remshare`	$u \rightleftarrows s$	Removes information about a file stored in a user node from the registry of a search node
`modshare`	$u \rightleftarrows s$	Modifies information about a file stored in a user node from the registry of a search node
`stats`	$usi \rightleftarrows i$	Request network statistics from index nodes
`search`	$usi \rightleftarrows s$	Perform a query for files
`browse`	$usi \rightleftarrows usi$	Browse all shared files of a host
`push`	$usi \rightleftarrows usi$	Perform HTTP PUSH through fire walls

Table 5.4: Summary of OpenFT messages

- Connection maintenance: `ping`
- File sharing management: `child`, `addshare`, `remshare`, `modshare`
- Searching for files: `search`, `browse`
- Miscellaneous: `stats`, `push`

Since the protocol has not matured to a level comparable to Gnutella additional details are intentionally left out. Unfortunately there is not any thorough documentation on OpenFT. To know the details a checkout of the project source code from their CVS is necessary. The source code is fairly easy to read and is (unlike some other open source projects) not completely uncommented. Instructions on how to obtain the source code are found in [GIF03].

Some additional details concerning the performance of OpenFT is presented in chapter 6 on page 39.

5.5.3 Query management

Queries are the most important part of a file sharing protocol. When a client issues a query it sends a `SEARCH` command to the giFT bridge. The giFT bridge will translate the call to the corresponding call in any registered protocols. If we assume that OpenFT is the only registered protocol the steps shown below occur.

1. The giFT handler for `SEARCH` commands calls `handle_search` specifying that a search command has been issued.

2. Create a new search by calling `if_search_new`.

3. For each registered protocol (in this case OpenFT), call its `search` function, in the case of OpenFT `openft_search`.

4. If the current node is a search node, search the local database by calling `ft_search`. The local database contains information about files stored at child nodes and at the node itself.

5. Forward the search to the search nodes of the network to which the node maintains a connection.

When a query arrives at a search node no further propagation of queries appears to be performed. This lowers bandwidth, but increases the number of messages sent by user nodes. The node responsible for the query contacts all the search nodes it maintains connections with, and no other nodes will be contacted. For this to work well the exchange of node lists (using the `nodelist` command) becomes more important. All nodes needs a good view on where search nodes can be found. To lower bandwidth consumption some criterias for choosing which search nodes to query would also be a good idea, but this remains unimplemented. When the network grows larger the exchange of node lists could waste quite a lot of network bandwidth, since many nodes also means an increased number of changes in the network per time unit.

When responses arrive the callback `ft_search_response` will be called. This callback calls `ft_search_reply`, which later on issues a call to the `search_result` function, that is responsible for forwarding OpenFT search responses to the giFT bridge.

5.5.4 File transfer

OpenFT uses HTTP for file transfer. This is the same method used by Gnutella and FastTrack clients. For a more thorough description on how this could work in practice see section 3.5 on page 20.

5.6 Evaluation of giFT usability

Even though giFT does not have the amount of users that Gnutella has and certainly not the amount that FastTrack has, it still works fairly well. If this is because the percentage of generous users is much higher, or if the way the OpenFT protocol works provides good results for queries is not known. At the moment, FastTrack outperforms giFT, but Gnutella does not provide considerably much better results, even though they have about one hundred times as many users.

If you have a Linux machine without an X server giFTcurs is the client of choice for file sharing. The same goes for those people who prefer curses and terminal based clients over those that use graphical widgets. Believe it or not, there are such people. This client is easy to install and provides an easy-to-use and clean interface. The only problem is that frequent updates are needed to the giFT libraries, as well as those used by OpenFT. Using CVS this is not a very time consuming task.

Chapter 6

Analysis

This chapter analyzes the previously discussed projects and examines the network usage of the corresponding protocols. Existing research on protocol analysis is also summarized.

6.1 Introduction

To determine what parts are needed for an efficient peer-to-peer system existing approaches need to be analyzed from an efficiency perspective. A lot of work has been performed on Gnutella and the results obtained are presented here. A similar analysis is performed on P-Grid and giFT/OpenFT to see what parts of these systems needs to be changed or discarded in order to achieve better utilization of network resources.

6.2 Gnutella network usage

According to [RIF02] earlier versions of Gnutella used only 36% of the total bandwidth consumption for `Query` messages. The remaining bandwidth was consumed by `Ping` and `Pong` messages (55%), `Push` message (8%) and additional erroneous messages (1%). A big concern is the possibly big difference with the structure of the Gnutella network and the underlying network topology. It is possible that a single Gnutella message passes a single link several times, which seems unnecessary.

Newer versions of Gnutella that use ultrapeers and better caching techniques reach 91% `Query` messages according to [RIF02], which is good. The problem of messages crossing a link several times still remains though since the protocol does not consider geographical locations of its' peers.

6.2.1 Test configuration and scenario

The Gnutella client **Limewire**, version 2.8.6, was installed on a computer with the configuration shown in table 6.1 on the following page. The files shared included several books/manuals in PDF format and classic computer games (in compressed archives). No video or music files were shared. The shared documents were related to computers and programming, or in some cases mathematics.

Component	Configuration
CPU	AMD Athlon XP 2000+
Memory	512 MB DDR
Network card	D-Link DFE 530TX 10/100
Network	10 Mbps, shared by 30 hosts
OS	Windows XP Professional
Shared files	562 MB, 59 files

Table 6.1: Configuration of Gnutella simulation environment

Each of the test sessions were performed in the manner described below.

1. Start the Gnutella client and make sure it gets connections to 8 other hosts on the network.
2. Enable statistical logging for incoming and outgoing messages as well as routing errors.
3. Let the application work for five minutes to become reasonably well connected.
4. Make a search using the keyword `divx` and note the number of hits.
5. Try to start downloading a random sample of the hits. If download starts it is considered a successful hit, if not an invalid hit.
6. After about 30 minutes make an audio search using the artist keyword `Madonna` and note the number of hits.
7. Try to download a random sample in the same way as described earlier.
8. After about 45 minutes make a document search using the keyword `nutshell` and note the number of hits.
9. Try downloading a random sample in the same fashion as before.
10. Let the system run for a few more minutes before shutting down.

6.2.2 Test results

Several test sessions have been conducted and the results are presented in this section. The conclusions from these sessions are presented later in this chapter.

Session 1: March 13th, 2003

This test session was performed at lunch time, around 12:00, local time (GMT +01:00). The results are presented in table 6.2 on the next page.

Session 2: March 17th, 2003

This test session was performed in the evening, around 19:00, local time (GMT +01:00). The results are presented in table 6.3 on the following page.

Query	Hits	Samples	Ok	Failed	Queued
divx	1445	35	9	24	2
Madonna	1339	31	19	10	2
nutshell	18	9	3	2	4

Table 6.2: Results from Gnutella test session 1

Query	Hits	Samples	Ok	Failed	Queued
divx	1121	39	15	22	2
Madonna	1139	35	17	15	3
nutshell	27	15	2	11	2

Table 6.3: Results from Gnutella test session 2

Session 3: April 15th, 2003

This session was conducted to see if there were any noticeable differences when using a client that is not brand new. The session was performed around 19:00 local time (GMT +01:00). The results are shown in table 6.4.

Query	Hits	Samples	Ok	Failed	Queued
divx	1449	45	10	33	2
Madonna	1443	39	20	17	2
nutshell	12	11	3	7	1

Table 6.4: Results from Gnutella test session 3

No remarkable difference in the results. The only thing worth noting is that the most successful samples on the divx test where found early. After about 25 samples almost all 10 successful items had been encountered. This shows that the hosts on the Gnutella network are often short lived and unreliable, since the hosts that were contacted later on seemed to have disconnected.

6.2.3 Logged statistics

The statistics logged throughout the sessions have been analyzed and the results are presented in this section. It turned out to be rather difficult to analyze the statistics since the logging is asynchronous and hence the time between each entry is unknown. The only thing known is the total number of entries and an approximate knowledge of the length of the entire session. This still gives good accuracy since the estimation ends up at slightly above one second per entry. The source code of the Limewire client has been examined without finding anything pointing to a constant time difference between log entries, but the entries are ordered correctly since the actual output to the log files are synchronized.

Since the logging of individual statistic types is not started at the same time, some statistical values obtained early on have been discarded. This means that each statistical type has the same number of entries. The number

of entries used for each session is shown in table 6.5.

Session	Entries
1	2678
2	2708
3	3119

Table 6.5: Entries per statistical category for Gnutella sessions

To reduce the number of points presented in diagrams each 50 consecutive statistical samples have been added together. Such a sample seems to be a reasonable approximation for the traffic flow during the last minute on the system used.

Statistical gathering

The bandwidth used during each session almost solely consists of protocol messages. If a download sample finds one or more hosts to download from, the download is terminated at once. Although a small overhead exists when performing downloads as well since several hosts are contacted in order to perform *swarm downloading*. Swarm downloading means that a file is downloaded from several hosts at the same time. Each participating host can then send a different part of the file, which decreases the download time substantially if the network to which the file is sent has higher capacity than that of the fastest host storing the file. The interesting things to examine is the following:

- How many messages are arriving over a certain period of time?
- How much network bandwidth does these messages consume?
- How many messages are sent over a certain period of time?
- How much network bandwidth does the transmitted messages consume?
- How many erroneous messages arrive over a certain period of time?

Incoming messages

Figures 6.1 on the next page shows the number of incoming messages and figure 6.2 on page 44 shows the corresponding number of bytes during test session 1. For each arriving message an additional 40 bytes should be added to include the TCP and IP headers used during the transmission. In reality some additional data is used on the network link layer, but these are not included in the discussions.

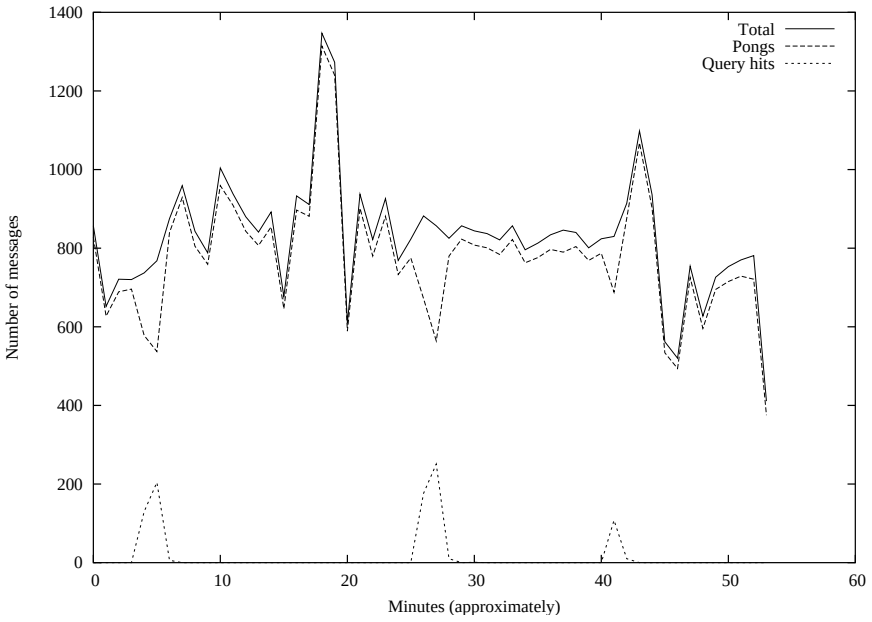

Figure 6.1: Messages received throughout Gnutella session 1

The corresponding results of test session 2 is shown in figures 6.3 on the next page and 6.4 on page 45.

Finally the third session resulted in figures 6.5 on page 45 and 6.6 on page 46.

Outgoing messages

The number of transmitted messages and the corresponding number of bytes used for session 1 is shown in figures 6.7 on page 46 and 6.8 on page 47.

During session 2 the statistics obtained for sent messages resulted in figures 6.9 on page 47 and 6.10 on page 48.

The third session recorded the statistics shown in figures 6.11 on page 48 and 6.12 on page 49 for outgoing messages.

Routing errors

Routing errors proved to be an interesting thing to look at. The Limewire client is able to log any `Pong`, `QueryHit` or `Push` message that has been erroneously routed. Only `Pong` messages showed up in this category and hence those messages are the only ones shown in figures 6.13 on page 49, 6.14 on page 50 and 6.15 on page 50 corresponding to the three sessions.

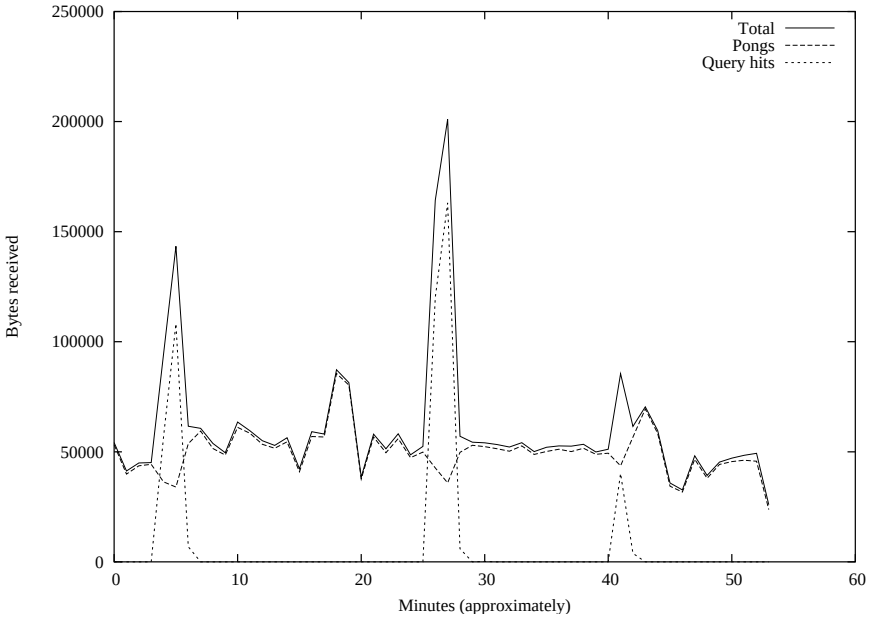

Figure 6.2: Bytes received throughout Gnutella session 1

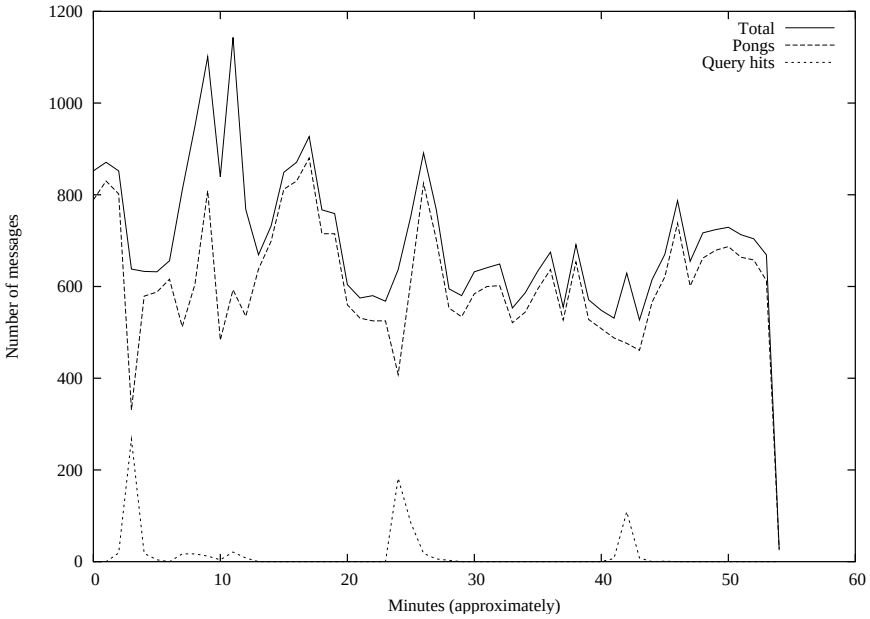

Figure 6.3: Messages received throughout Gnutella session 2

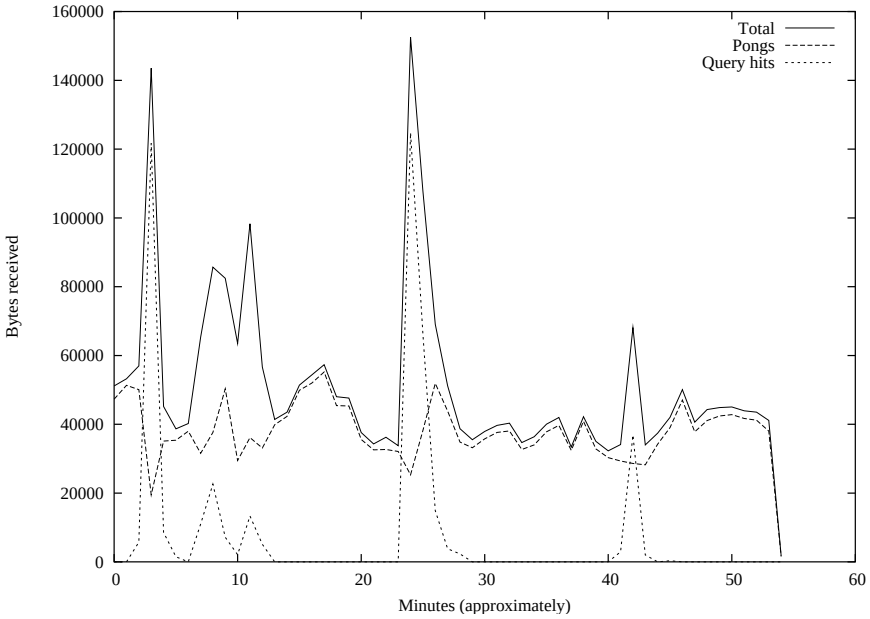

Figure 6.4: Bytes received throughout Gnutella session 2

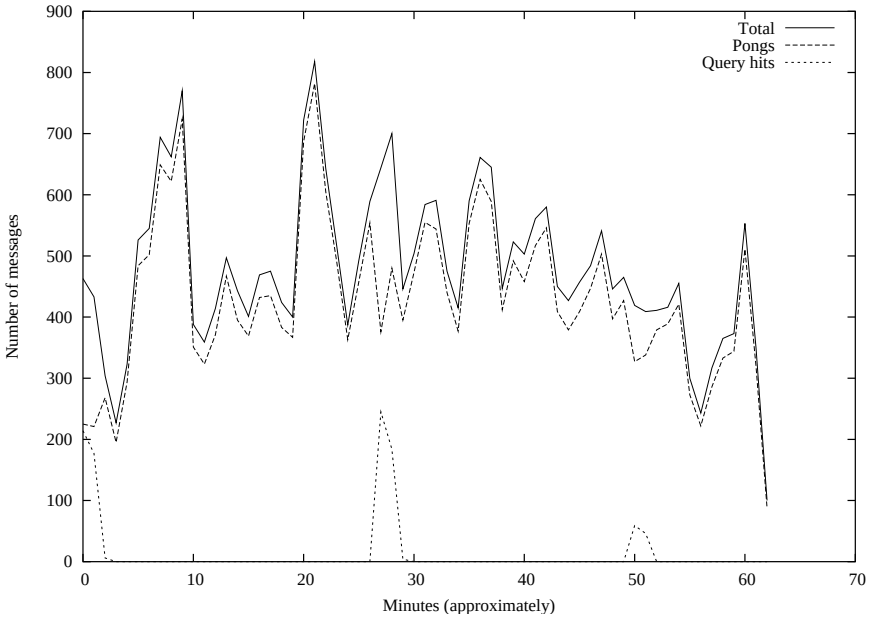

Figure 6.5: Messages received throughout Gnutella session 3

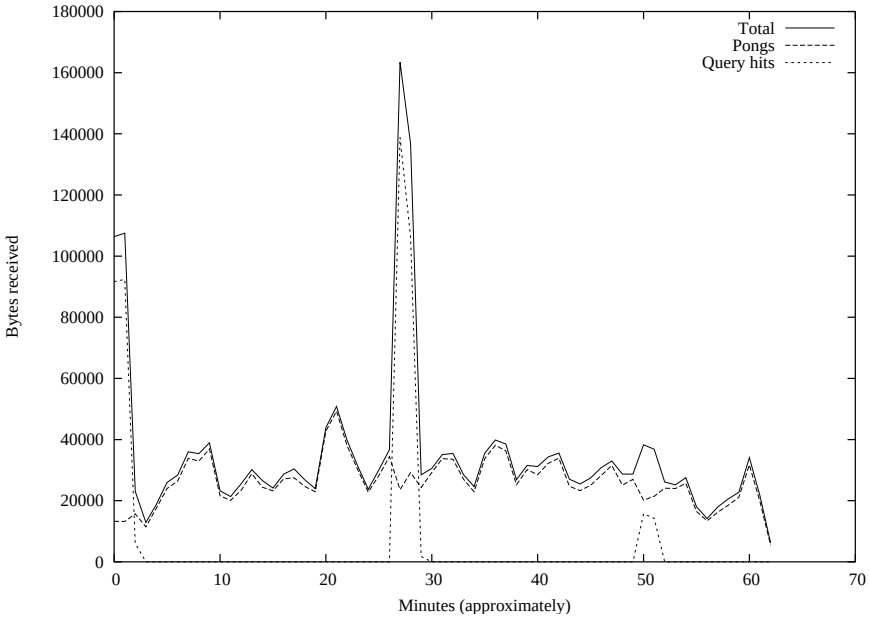

Figure 6.6: Bytes received throughout Gnutella session 3

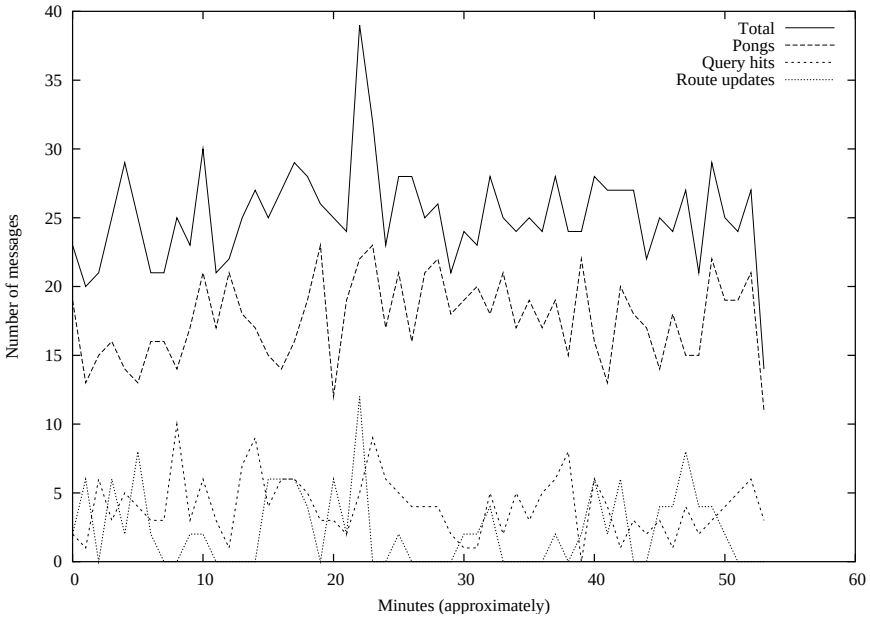

Figure 6.7: Messages sent throughout Gnutella session 1

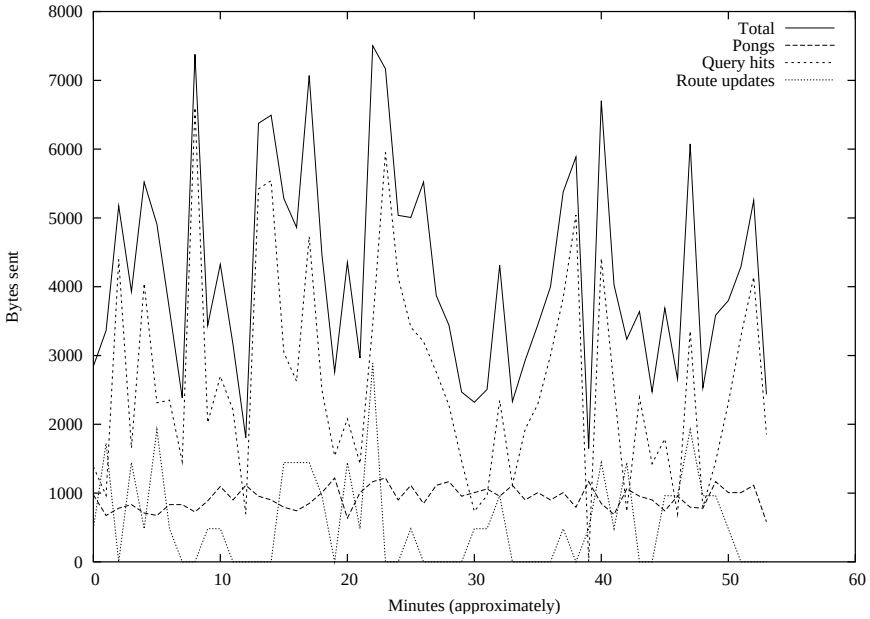

Figure 6.8: Bytes sent throughout Gnutella session 1

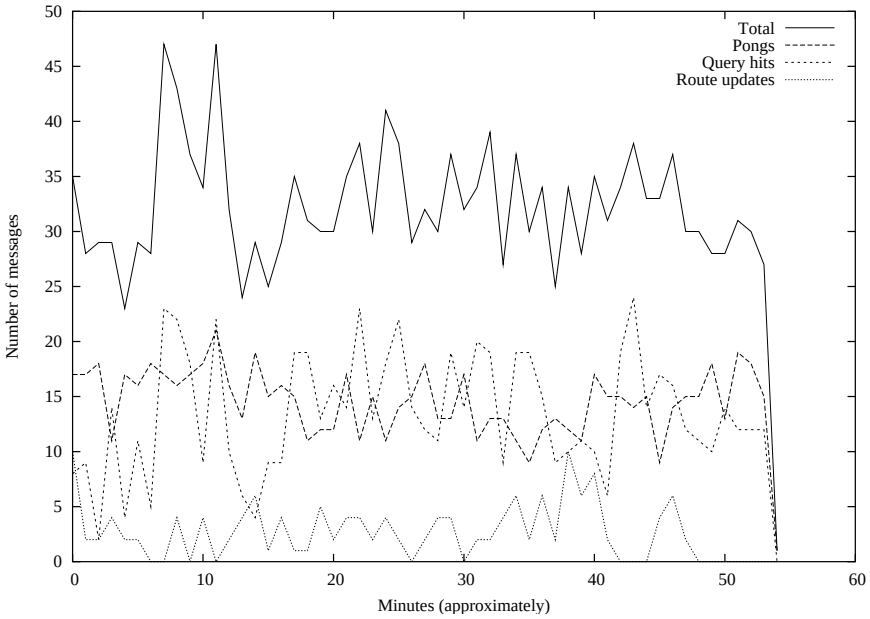

Figure 6.9: Messages sent throughout Gnutella session 2

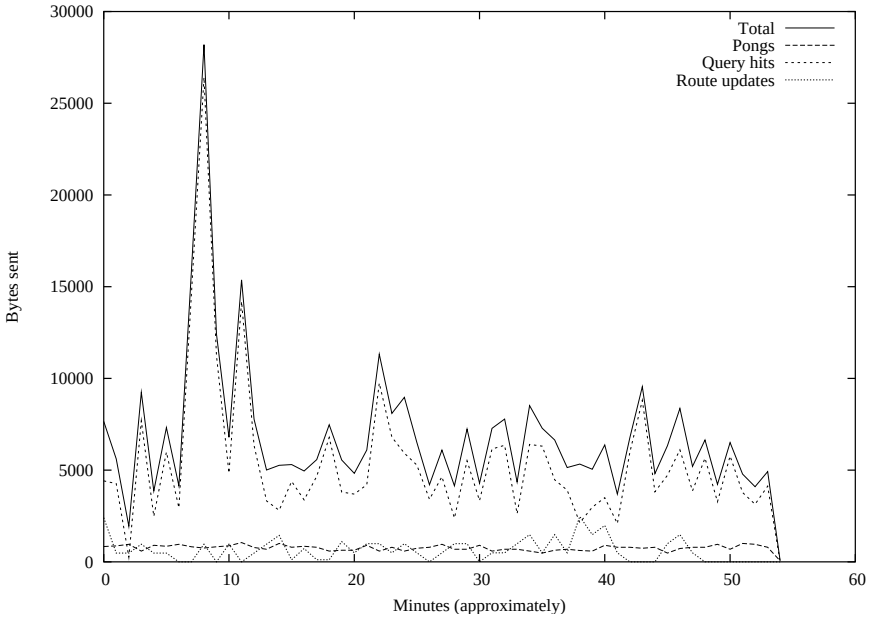

Figure 6.10: Bytes sent throughout Gnutella session 2

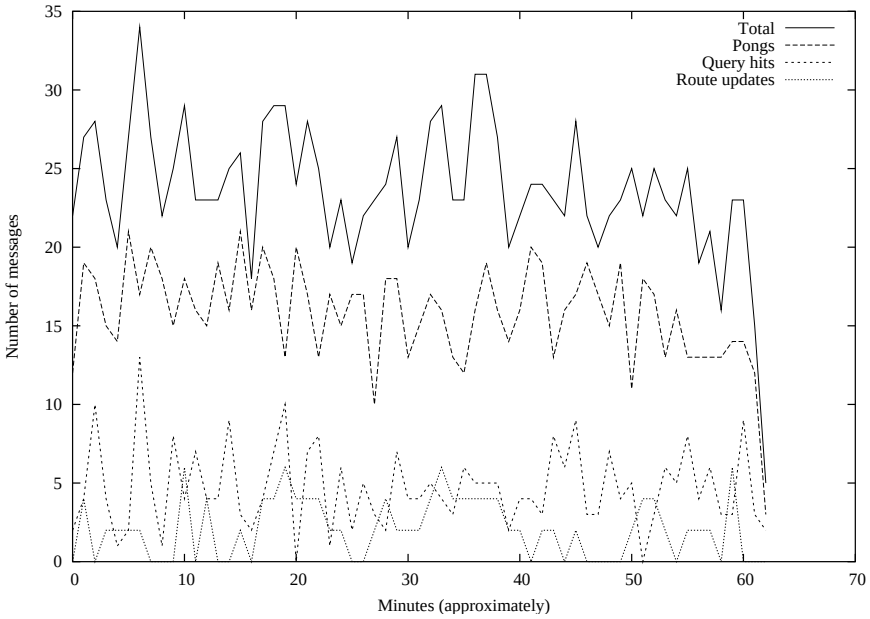

Figure 6.11: Messages sent throughout Gnutella session 3

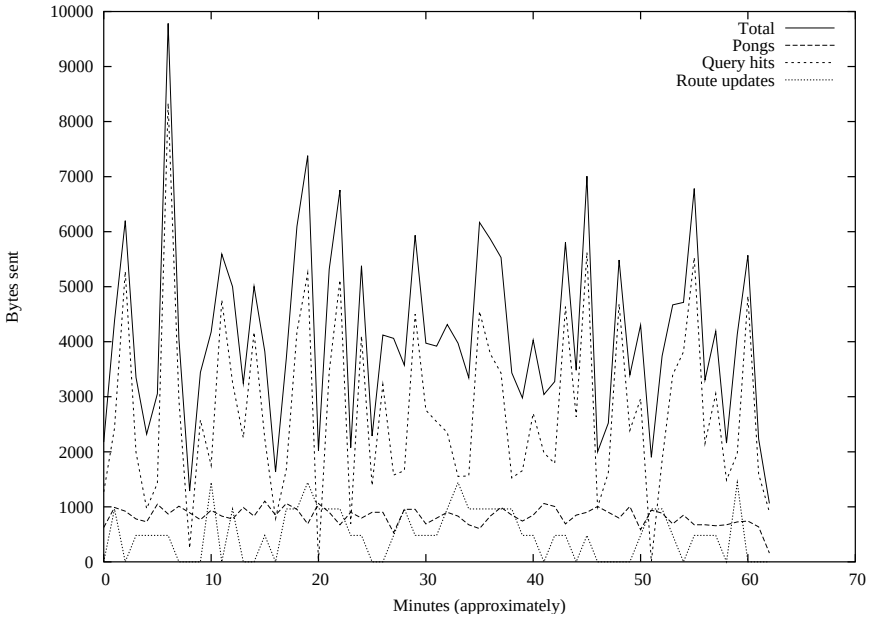

Figure 6.12: Bytes sent throughout Gnutella session 3

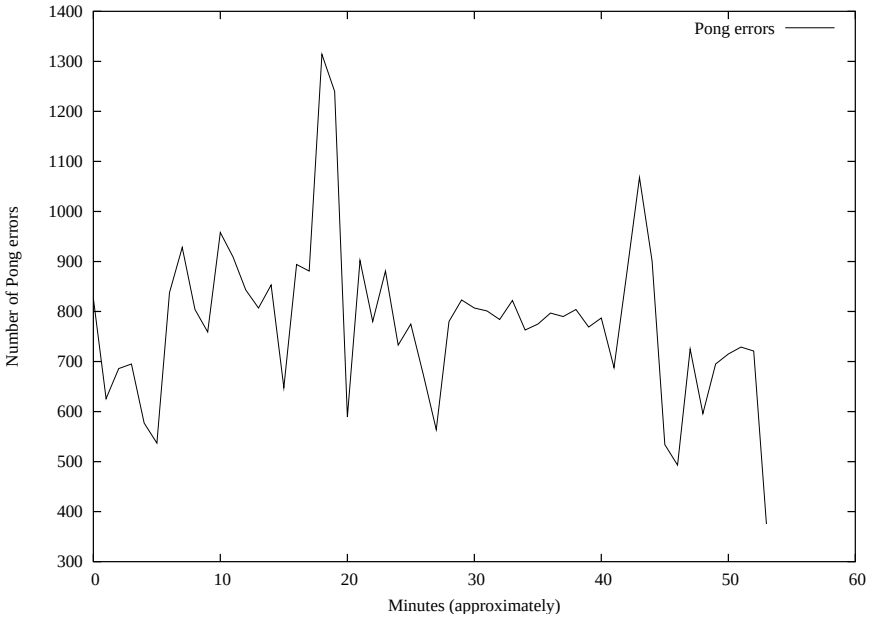

Figure 6.13: Number of erroneous Pong messages received in session 1

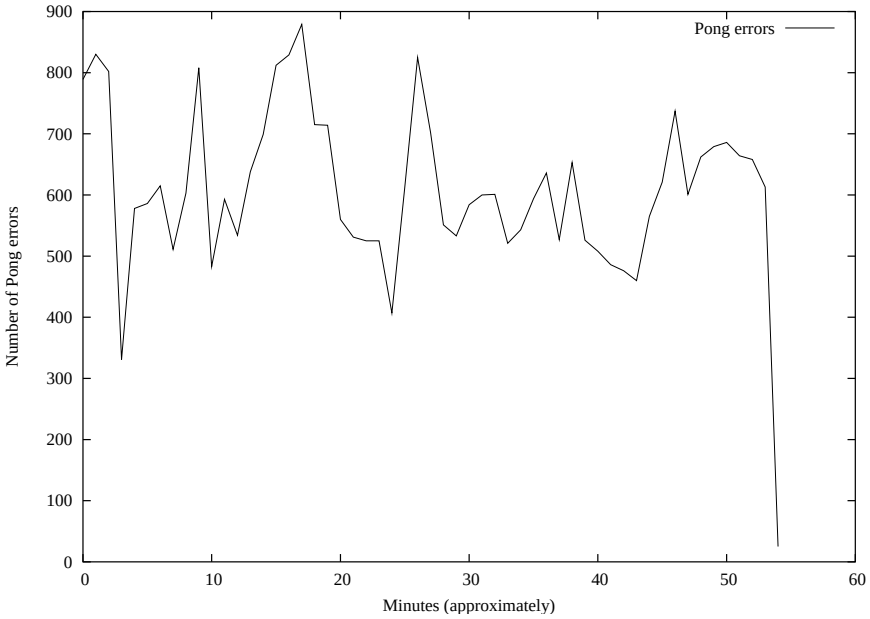

Figure 6.14: Number of erroneous Pong messages received in session 2

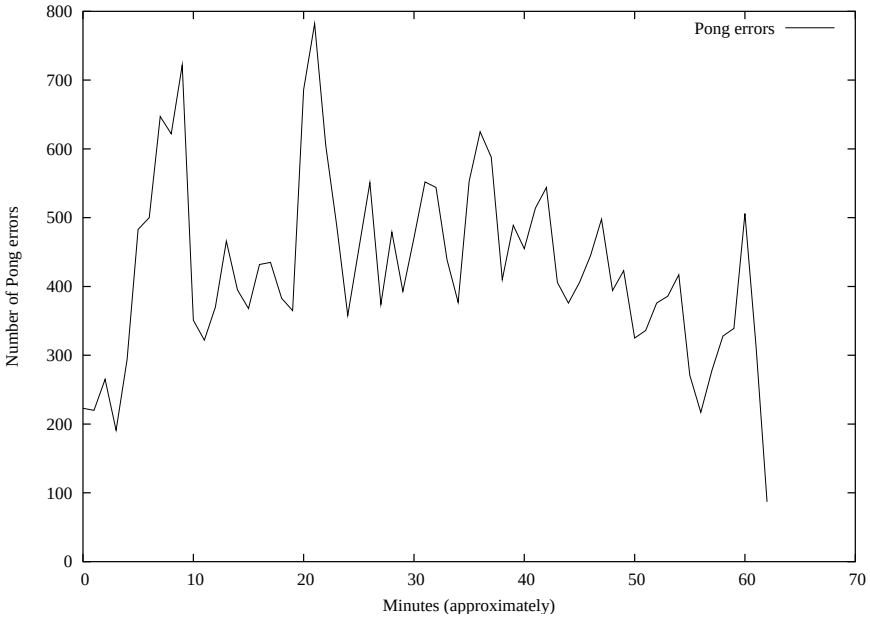

Figure 6.15: Number of erroneous Pong messages received in session 3

6.2.4 Summary and conclusions

To summarize what has been obtained during the three test sessions a number of tables are presented in this section. Each table contains a summary of a certain aspect of the statistics. Calculating the average amount of network consumption is the most important part, but several other aspects are also considered. A number of conclusions are also drawn from these summaries.

The average amount of incoming traffic for each of the three sessions is presented in table 6.6 and the corresponding data for outgoing traffic is shown in table 6.7.

Session	**Avg**	$\mathbf{Q_0}$	$\mathbf{Q_1}$	$\mathbf{Q_2}$	$\mathbf{Q_3}$	$\mathbf{Q_4}$
First	60609	25947	48526	53304	59316	201019
Second	51928	32266	37748	43527	53797	152548
Third	35666	12910	24907	28687	35428	163427
Total	48754	12910	32473	41384	53931	201019

Table 6.6: Gnutella session incoming bytes/minute

Session	**Avg**	$\mathbf{Q_0}$	$\mathbf{Q_1}$	$\mathbf{Q_2}$	$\mathbf{Q_3}$	$\mathbf{Q_4}$
First	4179	1647	2885	3871	5215	7503
Second	6975	1939	4874	6097	7564	28197
Third	4120	1062	3052	3974	5332	9784
Total	5040	1062	3436	4365	6093	28197

Table 6.7: Gnutella sessions outgoing bytes/minute

It is interesting to note that the bandwidth used by incoming messages is decreasing for each session conducted. The difference is due to a large decrease in incoming `Pong` messages. The difference between the first two does not appear to so large, but that is because during the second session a lot more `Query` messages were forwarded to us than during the other two sessions. This makes the difference between sessions two and three even larger.

Another very interesting aspect of the Limewire Gnutella client is that the routing of `Pong` messages appears to be flawed. During session 1, 41937 of 41956 incoming `Pong` messages were considered erroneous. For sessions two and three these numbers are 33346 of 33375 and 26912 of 27048 respectively. In all cases 99.5% or more of all incoming `Pong` messages were discarded. If this thing happens all over the Gnutella network it will result in a large waste of bandwidth.

The large number of bytes sent during session 2 is due to the larger number of incoming queries during that session. These queries led to the transmission of about twice as many bytes in `QueryHit` messages during session 2 than in the other sessions.

If all nodes of the Gnutella network are considered to have a similar amount of traffic it is easy to make some calculations on how much network bandwidth is actually used by the Gnutella protocol (not including any bandwidth used for downloading files). This assumption should not overestimate the total amount of traffic since ultrapeers use more bandwidth than

leaves of the network. Also assume that the third session, which consumed the least bandwidth, is reasonably representative. The total bandwidth used per minute for each Gnutella host can then be calculated from the following:

$$\begin{aligned} H &\approx 500 \text{ TCP/IP headers/minute} \\ h &= 40 \text{ bytes/header} \\ i &\approx 35000 \text{ incoming bytes/minute} \\ o &\approx 4000 \text{ outgoing bytes/minute} \end{aligned}$$

According to [LIM03a], which maintains statistics for the Gnutella network, a reasonable approximation on the number of hosts connected to the network at any given time appears to be $n = 95000 \pm 15000$. If this number is reasonably accurate the total network usage each minute by the Gnutella protocol is approximately:

$$T = n \times (h \times H + (i + o))$$

With real numbers this becomes quite a lot, but to see what it could mean in scalability terms lets extend the result to a 24 hour period instead of just one minute. It then becomes:

$$\begin{aligned} T &= 95000 \times (40 \times 500 + (35000 + 4000)) \times 60 \times 24 \\ T &\approx 8 \times 10^{12} \text{ bytes/day} \end{aligned}$$

Even if all TCP/IP headers are discarded it still accumulates to 5.3×10^{12} bytes/day. In reality the number of such headers could be less than 500 if several Gnutella messages are sent in a single packet, but this depends on the frequency of transmissions, the network library implementation and its configuration in the operating system. Nevertheless, the order of magnitude remains at a couple of terabytes/day of network bandwidth being used by the Gnutella protocol.

The problem is fairly evident. There are still, even after the introduction of ultrapeers and caching schemes, problems with the Gnutella protocol. It consumes lots of bandwidth just for maintaining its structure and relaying queries. There is also the possibility that messages cross a single network link several times as it is relayed between peers, as described in [RIF02]. It is also worth noting that the headers used by TCP/IP consume a lot of bandwidth since many small messages are transmitted, making the overhead rather large. A good protocol should avoid polling the connection with small messages as much as possible and instead use more meaningful operations to, besides their normal purpose, maintain connectivity.

6.3 P-Grid network usage

Since there is no complete client available that uses P-Grid the analysis will be more theoretical. Some simulations on the exchange algorithm have been performed though and the results of those simulations will be presented in this section as well.

6.3.1 Problems with P-Grid

The papers written on P-Grid present results on the performance of the exchange and search algorithms. These results are not based on bandwidth consumption but rather on more theoretical aspects, such as number of exchanges performed before reaching a stable state.

It is not specified if a node that has been offline for a period of time needs to start from scratch with the exchange algorithm, or uses exactly the same key and reference table as before it went offline. The second approach is preferable, but if a node is away for long periods of time the reference table could become outdated, which requires exchanges to be updated.

Some of the descriptions of the exchange algorithm involves the exchange of data, see [Abe02]. Such an implementation would probably not be very useful in practice since it could lead to that the time to perform a single exchange is several hours, if lots of data is to be sent. This also puts additional requirements on the availability of nodes, since an incomplete exchange is less useful than a completed one, although an incomplete exchange is not necessarily useless. The exchange of data in the exchange algorithm is not mentioned in either of [Abe01], [APHS02] and [AHP03].

6.3.2 Building the distributed search tree

The most important part of P-Grid is the construction of the distributed search tree using the exchange algorithm. The main question is, how much work is required to maintain the distributed search tree? The construction of the P-Grid search tree initially leads to dramatic changes, since exchanges partition the key space between peers. After a while peers have initialized their reference tables with some entries and they can start forwarding each other to other peers.

Three things should be investigated in order to verify the efficiency claimed by the authors (see [Abe01]), namely the following.

- Given an uninitialized network with N nodes, how many exchanges are needed to reach a stable state?
- Given an initialized network with N nodes, how many exchanges are needed to incorporate a new node n into the network and reaching a stable state?
- If a subset of the initialized nodes disconnect, how many exchanges are needed to reestablish the stable state?

The results depend a lot on the parameters of the P-Grid. For example, the number of references stored by each peer, the expected availability of nodes and the number of expected alternative references to a data item all affect the results. The number of recursion levels allowed when forwarding peers to other peers also has a large impact on the resulting performance of the algorithm.

The first of the items above is investigated thoroughly in [Abe01], which shows that the network converges to a stable state no matter how many nodes are in the network. One of the examples studied in [Abe01] shows that a network of 20000 peers with an online probability of 30%, a maximum key length

of 10 bits and a maximum number of references per level of 20, converged to a stable state after each node had participated in approximately 62 exchanges. Simulations have for the most part verified those results, although some differences has been detected. The differences are minor and probably depend on minor implementation differences.

If the network already is in a stable state and a new node arrives it quickly receives a lot of information about the network and is able to extend its key very fast. It simply initiates exchanges with some random nodes on the network, which leads to that the reference table of the new node quickly receives entries and is able to extend its key. Simulations have confirmed this behaviour.

If a subset of the nodes disconnect all the other nodes that refer to a disconnected node could be affected. If the number of alternative references is large this is not a problem, but using lots of alternative references slows down the algorithm and it takes longer time to reach a stable state. Overall, this is a problem that has been discussed by the developers of P-Grid, but has not received as much attention as it deserves. Simulations have shown that quite a large amount of exchanges could be necessary to reach a stable state again. It does however appear to be the case that the usability of the network is not seriously affected.

6.3.3 Performing queries

Assuming that a P-Grid tree has been constructed in some way it is interesting to see how efficiently queries are expected to perform. If there are N peers in the network, each peer can store d data items and that each leaf node stores r references to data items. Then the key length k can be expressed as follows:

$$k \geq \log \frac{N \times d}{r}$$

If a alternative references should be present for a single data item we have that the number of peers that needs to be online N' is bounded by the following expression:

$$N' \geq \frac{N \times d}{r} \times a$$

If a peer is online with probability p then the probability of a search returning at least one successful answer is given by the expression below. It simply says that the success ratio S is equal to the probability that at least one peer is available for each level in the tree on the way from the root to the leaf.

$$S = (1 - (1 - p)^a)^k$$

A short example

If we assume that there are $N \times d = 10^7$ data items in a file sharing system and that the nodes at the leaf level store $r = 10^4$ references to data items then, assuming that for example $a = 20$ the number of nodes required to be online is given by:

$$N' \geq \frac{10^7}{10^4} \times 20 = 20000$$

These numbers require that $k \geq 10$. Depending on the number of alternative references per data item, a, and the probability of a node being online, p, we get the success ratios shown in table 6.8 for some chosen values.

	p						
a	0.1	0.2	0.25	0.3	0.35	0.4	0.5
10	0.014	0.321	0.560	0.751	0.873	0.941	0.990
15	0.100	0.699	0.874	0.954	0.984	0.995	1.000
20	0.273	0.891	0.969	0.992	0.998	1.000	1.000
25	0.475	0.963	0.992	0.999	1.000	1.000	1.000
30	0.648	0.988	0.998	1.000	1.000	1.000	1.000
35	0.776	0.996	1.000	1.000	1.000	1.000	1.000

Table 6.8: Success ratios for queries in a P-Grid network

This shows that it is possible to achieve very high probabilities for success. Assuming that a peer is online only 20% of the time gives us 96% success ratio if 25 alternative references are maintained. If the availability of nodes go up to 25% the same amount of alternative references gives a success ratio of 99%.

If we for example changed r from 10^4 to 10^3 then k would change from 10 to 14. It would still give a success ratio of 0.990 for $a = 25$ and $p = 0.25$. For $p = 0.2$ the success ratio becomes 0.948.

6.4 OpenFT and giFT

The tests of OpenFT has been performed using the giFTcurs client for giFT. Since this client does not provide any decent way to log statistics a separate network monitor has been used. This makes it easier to see how much bandwidth is consumed by the protocol per unit of time, but makes it much harder to categorize the protocol traffic according to packet types.

6.4.1 Test environment

The test system used is shown in table 6.9 on the next page. The bandwidth analysis will be based on a session where no file sharing systems at all have started, to see how much bandwidth is used normally. This includes ICMP messages, ARP broadcasts and other messages that could arrive without actually running a file sharing application. The same files were shared as in the Guntella simulations.

Once an estimation of the network bandwidth used by other messages has been obtained, giFT and giFTcurs were started. The bandwidth consumption was measured using **Ethereal** and the number of active connections was monitored using **netstat**.

Component	Configuration
CPU	AMD Athlon XP 2000+
Memory	512 MB DDR
Network card	D-Link DFE 530TX 10/100
Network	10 Mbps, shared by 30 hosts
OS	Debian GNU/Linux, Woody
Shared files	562 MB, 59 files

Table 6.9: Configuration of OpenFT simulation environment

6.4.2 Performed tests

A couple of different tests have been performed to see how well the OpenFT protocol is working. First the system is started and left running for 10 minutes. During these minutes no queries are issued. After these 10 minutes **Ethereal** is stopped and the result is saved for later analysis.

A new network capture session is then started to see how the protocol behaves when queries are issued. During a 10 minute period the following five queries are issued, and the number of unique hits is noted.

- `divx` in the *everything* realm
- `Madonna` in the *audio* realm
- `nutshell` in the *text documents* realm
- `James Bond` in the *video* realm
- `Games` in the *software* realm

Before each query is performed **netstat** is used to find the number of currently open connections. Connections that are initializing are also noted, which gives an idea on how of much the set of established connections is changing.

After the 10 minutes has expired, **Ethereal** will be stopped once again and the captured information will be saved. All the queries will then be repeated, without monitoring the network consumption, and attempts to download some files will be made. If a download starts it is considered successful, otherwise a failure. For the queries that generate a lot of hits, only a subset of the files will be chosen for download tests. These download tests will show how well the classification of hits works. A hit is classified as either good, bad or somewhere in between. The number of hosts that stores the file is also presented in the user interface of giFTcurs depending on the answers received from the search nodes.

6.4.3 Test results

When no file sharing system was running, two 5 minute sessions were conducted. These two led to the following results.

- The first session found almost 62% of the traffic to be caused by ARP. The remaining traffic was mostly caused by NetBIOS running over UDP,

but several protocols contributed with about 2–3% of the traffic. This included ICMP and IGMP.

An average of 120 bytes per second of traffic was recorded throughout this session. This is a very small amount, as the next session illustrates.

- The second session used about twice as much bandwidth as the first one. During this session various TCP connection attempts were coming in and, for some reason, a few outgoing connections were established as well.

  About 29% of the traffic was caused by ARP, 39% by NetBIOS over UDP and 27% by TCP. Once again about 2–3% each were caused by ICMP and IGMP. The average traffic amount during this session was 245 bytes per second.

Since there was quite a large difference between the two tests with no file sharing application enabled, the worst one will be chosen as a point of reference. This means that on average about 250 bytes per second are generated by traffic not related to the file sharing service.

Running giFT without issuing queries

When running giFT, and hence OpenFT, without issuing any queries the following traffic was measured during three separate sessions.

- Session 1 consumed 878 bytes per second of which 81% was TCP traffic.
- Session 2 consumed 880 bytes per second of which 90% was TCP traffic.
- Session 3 consumed 1742 bytes per second of which 93% was TCP traffic. The increased traffic appears to have been caused by new transmissions of share information to parent nodes, since TCP data dramatically increased for this session. Examining the captured data closer confirms the theory of share indexing.

It is hard to provide a good approximation on the traffic amount. If the giFT daemon is running for longer periods of time the first two sessions are more appropriate. If the daemon is started only when the user wants to run a client, such as giFTcurs, additional traffic depending on how many files the user is sharing will be caused. For the further discussions an average value of 900 bytes per second will be assumed, unless explicitly specified otherwise. This leads to 650 bytes per second caused purely by OpenFT, without any queries being issued.

Issuing queries

When issuing queries the traffic amount increases. Session 1 above was followed by a 10 minute session where the aforementioned queries were performed. The results of the queries are shown in table 6.10 on the following page. The numbers in the quality column give the number of good, medium and bad quality hits, as classified by the giFTcurs client. Good hits require at least three different sources for the file. Bad hits are those that the search

node answering the query believes belong to a node, or nodes, that are currently offline. The last column denotes the number of connections in the `ESTABLISHED` state, according to **netstat**, before each query was issued, and the number in parenthesis denotes the number of connections being in the `SYN_SENT` state.

Query	Hits	Hit quality			Connections	
`divx`	350	8	111	231	92	(19)
`Madonna`	289	4	175	110	84	(11)
`nutshell`	3	0	2	1	115	(13)
`James Bond`	30	0	8	22	114	(6)
`Games`	21	0	13	8	80	(6)

Table 6.10: Query results using OpenFT

The number of hits presented is the number of *unique* hits. This means that the total number of hits at least as large. For example, all good quality hits correspond to at least three hits of the same file, but is only counted once. Gnutella and FastTrack clients use a different method, where each reply is counted, not each file.

It is evident that the number of connections used is very large, with an average of about 100. This corresponds to almost 10% of the total number of nodes online at any given time. The set of connections also appears to be changing rapidly as the number of initializing connections is about 10 at any given time. This causes quite a lot of bandwidth to be used for connection management. The total bandwidth used was 1444 bytes per second, with 88% being TCP traffic.

6.5 Comparison test of FastTrack and Gnutella

In order to have a reasonable way to compare the efficiency of file sharing protocols they must undergo the same analysis procedure. For this purpose the same tests as for OpenFT will also be performed with Gnutella and FastTrack. The setup is the same as for the Gnutella test sessions described earlier, and is shown in table 6.1 on page 40. The **Limewire** client, version 2.8.6 is used for Gnutella tests and the **KaZaA Lite** client, version 2.0.2 is used for FastTrack tests. The **Ethereal** network monitor is also used in these tests, although it is the version intended for Microsoft Windows users. This version also requires the **Win Pcap** packet capture driver to be installed. These tests used version 3.0 of this driver.

6.5.1 Results using Gnutella

Two sessions, each one being 10 minutes long, were conducted without issuing any queries. The following results were obtained using the **Limewire** Gnutella client.

- Session 1 used 1150 bytes per second of bandwidth, of which 89% were TCP traffic.

- Session 2 used 1641 bytes of bandwidth per second, with 92% caused by TCP traffic.

The reason for this 42% increase in bandwidth usage is due to an increase in the number of directly nodes, from 3 to 8. If 250 bytes per second are considered to be non-Gnutella traffic, this corresponds to 300 bytes/second per neighbouring node in the first session and 174 bytes/second per neighbouring node in the second session.

Finally, a 10 minute session was performed where the same set of queries were issued. Since Gnutella has a much larger user base, the number of replies should be much higher, and the total bandwidth consumption inevitably higher than that of giFT and OpenFT. The results are shown in table 6.11. The queries were performed using 8 neighbouring Gnutella nodes.

Query	Hits
`divx`	1266
`Madonna`	2444
`nutshell`	11
`James Bond`	19
`Games`	296

Table 6.11: Comparison test query results using Gnutella

It is worth noting that the number of hits for the second query is larger than the results obtained for the same query when analyzing Gnutella earlier on. One possible explanation for this is that a new album has newly been released and had some time to spread throughout the Gnutella network. The bandwidth usage was 2860 bytes per second on average, of which 91% was TCP traffic. Almost 22% of the total traffic was `QueryHit` messages.

6.5.2 Results using FastTrack

Since FastTrack uses both TCP and UDP as its transport protocol, both of them needs to be considered. A session where no queries were issued resulted in an average bandwidth usage of 200 bytes per second, of which 41% was TCP traffic and 29% was UDP traffic. This is *much* less than the bandwidth used by both Gnutella and OpenFT.

When issuing queries, FastTrack appears to limit its answers to about at most 500. This is a good thing, since it encourages users to issue more precise queries, which automatically decreases the number of replies sent compared to more general queries.

Since no connections are maintained for longer periods of time in FastTrack, table 6.12 on the next page only shows the number of hits received for the different queries.

When queries are issued the bandwidth usage increases a lot. The session conducted resulted in an average bandwidth usage of 1015 bytes per second, which still is comparable to Gnutella when no queries are issued at all. The traffic consisted of 82% TCP and 16% UDP traffic.

Query	Hits
divx	430
Madonna	457
nutshell	29
James Bond	121
Games	495

Table 6.12: Comparison test query results using FastTrack

6.6 Summary

OpenFT and Gnutella used a similar amount of bandwidth. The largest difference is that the traffic measured in Gnutella is only a small sample of the total traffic, since queries are propagated through the network. This could lead to an even larger amount of traffic somewhere else in the network, whereas in OpenFT the largest traffic amount is at the origin of the query, since queries are not forwarded between search nodes. There is of course the risk that a single search node receives a lot of queries in OpenFT, but the situation is the same in Gnutella, and most likely in FastTrack as well. In other words, issuing a lot of queries causes lots of traffic in both the Gnutella and the OpenFT network, but being passive could use more bandwidth in the Gnutella network. The introduction of ultrapeers removes some of these problems, but as simulations have shown, quite a lot of traffic are reaching leaf nodes as well.

In terms of network consumption, P-Grid is very efficient when it manages queries, but the exchanges performed to maintain the tree structure is the thing that potentially uses a lot of network resources. Since no exact numbers have been obtained it is difficult to compare P-Grid with for example Gnutella. It is however possible to conclude that P-Grid is superior to both Gnutella and OpenFT if the availability of the nodes in the network core is reasonably high. The results indicate that a more structured network, such as that of P-Grid, is preferable in terms of network usage compared to a more ad-hoc structure, such as Gnutella. An important issue however is that if nodes go offline they need to be quickly incorporated into the structure when they go online again. This procedure should not use too much bandwidth and, in particular, use the knowledge obtained in early sessions to lower the load on the network when it is reincorporated into the network structure.

Chapter 7

Protocol design

This chapter presents the design of an efficient peer-to-peer file sharing protocol. This design is based on the analysis performed earlier and contains protocol discussions and considerations such as partial centralization.

7.1 Introduction

Designing an efficient peer-to-peer file sharing system protocol requires careful structuring and aggregation of data. Efficient queries can be implemented only if the metadata being queried has been structured appropriately. Since peer-to-peer is distributed any algorithms maintaining any kind of structure must be fully distributed if there is a risk of facing legal charges or other problems.

It is also very important to structure the hosts appropriately. More powerful hosts can contribute more, without noticeably reducing their performance. This is especially important when discussing network bandwidth and the rules regulating its use.

Completely decentralized protocols are difficult to make efficient in terms of network usage. Using some kind of centralization introduces other problems. A single point of failure and the risk of network congestion at that point are problems that could be solved by traditional techniques such as mirroring and caching.

Another problem with centralization faced by file sharing systems is those of copyright. Many users of such systems are considered criminals, most notably by the RIAA (Recording Industry Association of America), since they are sharing music files. Video files and entire movies are becoming more and more common on todays file sharing systems and the movie industry are also trying to stop them.

The problem of balancing centralization and decentralization is one of the most commonly discussed issues in the peer-to-peer community. This section discusses a set of protocols that together would provide a file sharing system that is more efficient than for example Gnutella.

7.2 Metadata query protocols

A metadata query protocol is used to locate resources based on metadata describing this resource. Of special interest in this case is protocols that make it easy to go from a metadata based search to a key based search, since these can be made more efficient in distributed environments. For a file sharing system it is reasonable that the key used for searching is a hash value of the actual file. The SHA hash is used by several systems to identify files. It uses 160 bits and implementations are easily available.

In [GHI+01], databases for metadata queries in a peer-to-peer context are examined. Another interesting article is [JBBS01] which uses SQL queries for distributed data storages to query XML based metadata. A similar protocol, although simplified in that sense that it does not use SQL, is outlined later. Before going into the details of such a protocol an example of an existing distributed metadata service is discussed, namely the Internet Nomenclator Project.

7.2.1 Internet Nomenclator Project

The Internet Nomenclator Project (INP) integrates many CCSO name servers [INP98]. The abbreviation CCSO comes from the place where this server type is distributed from, namely University of Illinois Computing and Communications Services Organization.

Each CCSO server contains meta-information about things of interest to a particular site, such as a university campus or company. The main use of these servers seems to be information about people. If this information is not classified as secret the administrator of a CCSO server could send a description of how data is structured in the server to the Internet Nomenclator Project. The INP directory server creates the necessary indices and mappings to support the database schemas used by the particular CCSO server.

A meta-query sent to the INP server is processed and a set of matching CCSO servers is found. Queries to these servers, using their specialized format, are sent by the INP server. The responses are then translated back to standard form before they are forwarded to the host initially issuing the query.

The CCSO servers can contain almost any kind of information. The official web site of the project [INP97] contains the following paragraph when describing the system, which summarizes what the purpose of the project is.

> Nomenclator is a scalable, extensible information system for the Internet. It supports descriptive (i.e. relational) queries. Users locate information about people, organizations, hosts, services, publications, and other objects by describing their attributes. Nomenclator achieves fast descriptive query processing through an active catalog, and extensive metadata and data caching.

Caching plays an important role in a metadata service. Local caching increases the overall performance and lowers the load on the INP server. For maximum performance INP mirrors could be created. If a company or university campus are using INP queries quite a lot it could also be worthwhile setting up their own INP query resolver. The resolver is a local cache for INP queries and data responses. This enables a user to benefit from answers to

queries made by other people of the company or campus, which increases the overall performance and lowers the burden on the INP server.

The INP uses a query protocol named Simple Nomenclator Query Protocol (SNQP) [SNQ98]. This protocol uses SQL query statements to perform searching. The protocol is text based and a quick summary of the available commands is shown in table 7.1.

Command	Description
`advice`	Provide advice on query costs without executing query
`attributes`	List the attributes for a relation
`compare`	Set type of comparison operation
`help`	Explain the SNQP commands
`imagui`	Format replies for a graphical user interface
`next`	Stop processing current query, continue with next query in block
`noadvice`	Provide responses to queries. Do not advise on costs
`noimagui`	Format replies for people
`query`	Submit a block of one or more SQL query statements
`relations`	List the relations (names of database tables) available through the SNQP server
`stop`	End processing of current query, and cancel any queries remaining in block
`quit`	Terminate the query session

Table 7.1: Summary of SNQP commands

7.2.2 Metadata directory service

The INP provides an interesting example on a service that efficiently implements metadata based queries. This section outlines a metadata directory service intended for use in file sharing systems. It is important to note that the service is not intended for file sharing systems exclusively. It is intended to provide metadata searches in a more general context. For example a car manufacturer could register metadata about their cars in the system. One metadata property could for example be an URL where details of the car in question are presented. Since metadata is legal there is no problem with using a centralized architecture for better utilization of network resources. Mirroring and caching can improve network utilization even further.

It is important to note that a file sharing system cannot use an URL property describing which host stores the file. It could exist to denote the web site of the original manufacturer, but if such a property is used in other ways the system will be shut down, since it helps users find places to perform illegal downloads.

A file could be registered using appropriate metadata. Besides keywords or phrases describing the file, its size and SHA hash value are suitable properties to be registered. The system could also perform classification based on file

types or MIME types. If files are given too much details it could be argued that these details endorse illegal searches and distribution. Storing hash values provides a way to check the validity of a file download from the Internet, which is a service many people and companies could benefit from.

The registrations of files could be performed using an XML encoded message which also makes it easy for an application to perform automatic registrations. Since such a simple scheme is an easy target for a malicious user, some kind of authentication is probably necessary. If the registrations are transmitted using HTTP a possible improvement would be to use HTTPS (HTTP over SSL, Secure Socket Layer) to establish a secure connection that requires authentication using certificates. This should lower the number of people that tries to harm the system. The best thing from a security standpoint is to require each user to register herself before receiving the necessary certificates.

7.3 Metadata protocol specification

This section describes a simple metadata protocol that can be used to perform arbitrary metadata based searches. The protocol is based on XML and initially uses a centralized architecture, since storage of metadata is legal, if it is performed in a wide enough area. Simply having a metadata service for music files would be shutdown, but a more arbitrary service has uses in many more (legal) areas. For better performance a set of server mirrors can be used instead.

7.3.1 Definitions

Each entity stored in the metadata database has a corresponding XML DTD (Document Type Definition). It describes a single entity in the database, such as the metadata description of a music file. Since the database is not intended for file sharing alone, the level of detail for each file type is limited. Instead all files can be represented by the following fields:

Hash value of the entire file, using the SHA-1 algorithm. This is considered to be the key item in the database table for files and is also used when searching for nodes that host this file.

Size is an attribute that certainly all files have and hence it is useful.

Type contains the MIME type of the file.

Description is a short textual description of the entry. This is the field that will receive the most attention when performing searches.

Since the topic of discussion is file sharing suggestions on how other tables could look are omitted. For comparisons a set of comparison methods is needed. The following methods provide a place to start, but could be extended to include additional comparison methods:

Equality simply means to match a field in the query with the corresponding field in the database table and see if they are equal. This is the default comparison method.

Greater is used to check if a numeric field is greater than the value given in the query.

Less is the similar to the one above, except that fields less than the field in the query matches.

Contains performs word based matches. If all words given in the query match words found in the corresponding field in the database the comparison was successful. Words in this context means characters separated by whitespace. If the query contains a longer phrase that should be matched the spaces are translated to pluses '+' as in HTTP. Other special characters are encoded using the $\%xy$ construct also found in HTTP. Here x and y denote hexadecimal characters.

7.3.2 Performing a query

Figure 7.1 shows a suggested DTD for queries to the metadata server. Figure 7.2 shows how such a query looks in practice when it adheres to the DTD. The `!DOCTYPE` has been excluded from the example, but should refer to the previously mentioned DTD.

```
<!ELEMENT query          (compare+)>
<!ATTLIST query
          realm CDATA #REQUIRED
          id    CDATA #REQUIRED>
<!ELEMENT compare        (hash|size|type|desc)>
<!ATTLIST compare
          method (equality|greater|less|contains) "equality">
<!ELEMENT hash           (#PCDATA)>
<!ELEMENT size           (#PCDATA)>
<!ELEMENT type           (#PCDATA)>
<!ELEMENT desc           (#PCDATA)>
```

Figure 7.1: Possible DTD for metadata queries

Figure 7.2 will perform a query in the `file` *realm*. A realm denotes the context in which the query should be processed, and hence which database tables the server should use. The query will match any files with a MIME type of `audio/mpeg3`, whos size is at least 2.5MB and has a description containing the word Mozart and the two word phrase Betulia Liberata.

```
<query realm="file" id="78A5C7230E1F732AF0731D74902DAE65">
    <compare>
        <type>audio/mpeg3</type>
    </compare>
    <compare method="greater">
        <size>2500000</size>
    </compare>
    <compare method="contains">
        <desc>Mozart Betulia+Liberata</desc>
    </compare>
</query>
```

Figure 7.2: Example of a meta-deta query

If all goes well, and the query is successful, a response is sent to the client. In the previous example the database entries for all registered MP3s with Mozarts La Betulia Liberata from 1771 will be sent as a response. If the number of hits is large the server could choose to send only a random sample of the hits. A possible extension to the query itself would be to specify the largest number of hits to return.

7.3.3 Response to a query

The response has a similar format as the query. It also uses XML and has a similar DTD. A possible DTD for reponses could be the one shown in figure 7.3.

```
<!ELEMENT response (entry*)>
<!ATTLIST response
          realm CDATA #REQUIRED
          id    CDATA #REQUIRED>
<!ELEMENT entry    (hash|size|type|desc)>
<!ELEMENT hash     (#PCDATA)>
<!ELEMENT size     (#PCDATA)>
<!ELEMENT type     (#PCDATA)>
<!ELEMENT desc     (#PCDATA)>
```

Figure 7.3: Response DTD for metadata queries

The `realm` and `id` attributes correspond to the same attributes in the query. Any identifier can be used according to the DTD, but the semantics of the various fields are straightforward. Figure 7.4 shows a sample response to the previously shown query.

```
<response realm="file" id="78A5C7230E1F732AF0731D74902DAE65">
    <entry>
        <hash>Encoded SHA hash value</hash>
        <size>2786402</size>
        <type>audio/mpeg3</type>
        <desc>W.A. Mozart, La Betulia Liberata</desc>
    </entry>
    <entry>
        <hash>Encoded SHA hash value</hash>
        <size>4179600</size>
        <type>audio/mpeg3</type>
        <desc>
            Mozart 1771 La Betulia Liberata
            The Liberation of Bethulia
        </desc>
    </entry>
</response>
```

Figure 7.4: Example of a metadata query response

To save bandwidth it would be easy to use compression on the message payload if it is very large. If the service runs on top of HTTP a special header would be used to indicate that compression, and which compression algorithm, has been used.

7.4 An efficient peer-to-peer protocol

This section describes an efficient peer-to-peer protocol, Amorphous Distributed Tree Protocol ADTP, intended for file sharing. It can be used in other areas as well since the protocol mainly is about maintaining network structure and locating the resource we want. This suggested protocol is influenced by P-Grid, DNS, Gnutella and OpenFT which have been discussed in the earlier chapters. This is only a proposed protocol design, which has not been implemented and tested. Anyone who wishes to do so may implement it, but if it is intended to be used for commercial purposes the final protocol specification used must be publicly available. Any implementation of the protocol is humbly requested to give credit where such is due. Furthermore, the protocol is not allowed to be patented in any way.

7.4.1 Network nodes

All nodes in the network are equal. They all maintain the same kind of references. The only difference between nodes is the amount of reference information stored. Each node maintains the following:

Data repository is fundamental in a file sharing system. Any files that should be shared are added to the repository. In other applications this can be any kind of resource whos' elements should be located by searching the network.

Responsibility key is the binary key which the node is responsible for. This key can be extended if the *workload* becomes too heavy. It can also be decreased if the workload becomes too light. Is similar to the key prefixes used in P-Grid.

Set of buddies or mirrors. This set maintains the addresses of other known hosts that have the same responsibility as you do. If your responsibility is changed your buddies are notified. Responsibility in this case refers to your responsibility key.

Set of parents in the network. Contains the nodes in the network that are responsible for a prefix of your key. Could perhaps include parents several levels away for efficiency purposes, but this is an extension.

Set of children directly connected to you. Each child has a key with your key as a prefix. If you would extend your key, some of the children will become your buddies, other children your siblings and your current buddies will be your directly connected parents.

Set of siblings in the network. This set contains the nodes in the network you are aware of that have the same parent nodes as you do. Whenever a node extends its key, the children with keys different from the new key can place the node in their sibling set. This set is not necessary, but can be used to bypass parent nodes, which lowers their network traffic. Local caching is a simple and useful strategy, and hence it is used. It is also useful for monitoring what is happening in other branches of the parent nodes, without having to request a list of children from the parents. No node should depend on the correctness of the elements in

this set. Timestamps are probably needed if this set should be useful, since it is usually not updated very often.

7.4.2 Protocol overview

The proposed protocol builds a distributed search tree that fans out more than the binary tree used by P-Grid. This reduces the number of nodes involved in routing a query, which increases network utilization. All queries in the protocol are assumed to be key-based. This means that a separate metadata service could be needed, if high-quality metadata searches are required. It is possible to implement the step from metadata to key-based queries using a local conversion algorithm, but such algorithms usually produce worse results.

Peers connect to each other and locate a suitable place to join. By joining the network at a special place the node becomes a mirror at that location. This causes the node to have a special responsibility key, that denotes the set of search keys the node is responsible for. The responsibility key can grow or shrink, which means that the position of the node in the network changes, when the worload is deemed to high or too low.

Queries are routed based on key prefixes, similar to P-Grid. Keys in a file sharing system are expected to be 128 or 160 bits. The protocol is independent on the actual key length used though.

7.4.3 Protocol limitations

One big drawback with the protocol described here is that it does not provide the ability to route through firewalls and NAT routers. To provide such an extension without significant change to the protocol the following things are needed:

- The node with which the host behind the firewall maintains a connection must take responsibility over the files shared of the host behind the firewall. This does not mean that it has to store the files, but the host outside the firewall changes the address in the share registration messages to its own address.
- The node outside the firewall must maintain a registry of all registered files that does not belong to itself, but that it has registered as its own.
- If the node behind the firewall disconnects the node outside the firewall should attempt to cancel the registrations made, if the node behind the firewall has not already done it. If the node behind the firewall requests to cancel one or more shared files the node outside the firewall performs the corresponding address translations, if and only if, it has a matching entry in its registry. Any cancel requests that does not match are dropped.
- The node behind the firewall must be precise in which parent nodes it talks to, since parent node A cannot cancel a file that was initially registered with parent node B. There are two solutions. The first makes sure that the same parent node is used all the time, which relies on the reliability of a single host. The second approach is to use several parent nodes and send registrations (and unregistrations) to all (or a selected

subset) of the parent nodes. Another approach is to have a single parent node forward the registrations (and unregistrations) to its' mirrors.

- When a query for a file stored behind a firewall arrives at the parent node outside the firewall it determines, by checking its registry, which node stores the file. If the connection is still open, which it should be since there was an entry in the registry, the node outside the firewall tells the node behind the firewall to issue a push message to the host requesting the file.

This behaviour causes one or more hosts outside the firewall, or NAT router, to act as a proxy for the host that cannot be reached by new connections.

7.4.4 Key management

The most important part of the protocol is the management of keys. Initially the key is empty, which means that the node is responsible for everything. The key of a peer grows during interactions with other peers. A peer may grow its key if there are too few children with the corresponding key. A peer may also shorten its key if it wants to do so. The purpose of this freedom is to allow peers to find a place in the network that has a suitable workload corresponding to a reasonable fraction of the peers capacity.

A key may grow when a peer considers its workload too high. To allow bursts in workload this decision should be based on a longer time span. The peer examines the keys of its children to see which key has the least amount of mirroring. The peer then extends its key in the corresponding way and performs the necessary notifications of the change. Peers could also decide to extend their key based on the observation that the mirroring at the current level is sufficiently large, that is they have a lot of buddies.

A key may shrink when a peer feels that it can handle more work than it is currently performing. This decision should also be based on a longer period of time to avoid frequent key changes. If the number of parents is shrinking one or more child peers could decide to shrink their keys in order to maintain a good mirroring level.

If the mirroring level in the local area is very good, a node could try to find a place that is in more need of assistance. This is performed by requesting node lists from surrounding nodes. These node lists contains nodes further away, and helps a node detect if a place in the nearby network is in need of assistance. This causes a node to work in approximately in the same surrounding all the time. To maintain a reasonable balance in the network the initial location of a node, when it has never connected to the network before should be determined by a randomized algorithm. From this initial location the node is able to grow or shrink its key depending on its performance and workload.

Whenever a key is changed the buddies, children and parents needs to be notified. This could potentially use a lot of bandwidth if key changes are frequent, which is something they should not be. To avoid lots of communication when a new node is trying to find an appropriate place in the network, the other nodes in the network with which the new node communicates does not share the knowledge of the new node with any other nodes. It is the responsibility of the new node to notify its surrounding peers when it has found

a place that appears suitable. For the purpose of finding a suitable place for a new node all nodes needs to monitor their own work. By telling new nodes what kind of workload they are experiencing the new node can decide if it is worth trying to work at this place in the network, or move on to another place without trying. With the expression movement in the network we mean the extension or shrinking of the key a peer is responsible for.

7.4.5 Multi-bit keys

To lower the number of hops necessary to route a query through the network multi-bit keys can be used. This means that a child can have more than a single bit longer key than its parent. For example, a peer with key k could have 16 children where there are two children responsible for each of the keys $k000$, $k001$, $k010$, $k011$, $k100$, $k101$, $k110$ and $k111$. This allows a peer to use several bits of the key for a single hop, which reduces the total amount of hops. This is useful if keys are expected to be long, for example 128 bits.

If varying length keys are used then the parent is responsible for managing a key that is too short for any of its children. If k consists of 63 bits and the children use an additional three bits, for a total of 66 bits, then the node with key k is responsible for 64 bit keys with k as a prefix.

7.4.6 Peer classification

No real classification of peers exist in the protocol, but when examining the behaviour of nodes in the network three roles can be found. Since all nodes in practice has all three roles it is simply a matter of how much work is performed in the various roles.

Router peers are the peers with fairly short keys that are located at the "top" of the network tree. They maintain large sets of children and spend a lot of time forwarding messages queries. These nodes should have a reasonably high reliability and a high performance network connection, since they are often involved in query routing.

Server peers are the leaves of the network tree and store references to the resources being queried. Upon receiving a query the depository of references is searched for matches. Any references that match the queried key are returned. These nodes contribute with hard disk space for storing references and needs a reasonable network connection, but does not need as much network capacity as the router peers.

Client peers are the peers that issue queries in the network. They can be router peers or server peers, since it makes no difference in their behaviour when issuing queries.

7.4.7 Establishing connections

When a peer A connects to another peer B the scenario shown in table 7.2 on the following page takes place. The connection is made against a well known port number. The recommended number to use is 1232. Any headers could be used, but the headers shown below should be supported.

Peer A	**Peer** B
`ADTP/0.1 CONNECT\r\n` headers terminated by blank a line	
	`ADTP/0.1 200 OK\r\n` headers and header replies terminated by a blank line
`ADTP/0.1 200 OK\r\n` header replies terminated by a blank line	

Table 7.2: Connection initialization in ADTP

`Accept-encoding` gives any specific encoding types that the node is capable of accepting. Allows the protocol to support compression and encryption, although the protocol does not provide any means for safe distribution of encryption keys. This header is borrowed from Gnutella 0.6. The only suggested encodings are XML and `deflate`. The second of these denots zlib deflate/inflate compression.

`Content-encoding` acknowledges a `Accept-encoding` header, by listing the encodings that the responding node can use. If for example several compression alternatives are listed *one* is chosen, since the protocol only supports one compression algorithm at a time between two peers. It is possible to use compressed and non-compressed messages interchangeably though. By replying to an encoding a peer is expected to be able to handle both incoming and outgoing data encoded in the specified encoding.

If one of the peers wishes to discard the connection some other status code than 200 is sent as a reply during the connection initialization. The codes are standardized by SMTP, which is documented in [SMT01].

7.4.8 Protocol header

Each message exchanged after a connection has been initialized has the header shown in table 7.3. It is very simple, but provides the necessary flexibility to implement an advanced protocol. All fields are in network byte order.

Bytes	**Description**
0–1	Message type and message flags
2–3	Payload size

Table 7.3: Message header in ADTP

Message type and message flags share 16 bits. The uppermost bits contains flags and the lowermost bits contains the message type. The flags that could be used are the following, but additional flags can be added. A bit of 1 means enabled and 0 means disabled for flags, unless specified otherwise.

- Bit 31, payload compression. Indicates that the payload has been compressed using algorithm agreed upon during the connection initialization.
- Bit 30, XML payload that requires parsing.
- Bit 29–28, payload encoding. Specifies the text encoding used by the payload. Values are 00=ASCII, 01=ISO-8859-1, 10=UTF-8 and 11=other. If other is used the payload is expected to contain a description of the encoding if it is necessary for enterpreting the data. For normal binary messages, ASCII should be used.
- Bit 27, payload length multiplied by 8. The payload is padded by between 1 and 8 additional bytes. All extra bytes have the same value, but the value is arbitrary. It should be chosen so that it is different from the last real byte in the payload. This feature should rarely be needed, but could be useful if large sets of data needs to be transmitted.
- Bit 26, no reply needed. If a request message with this bit set is received no reply message is needed. This can be used to distribute information to other peers and prevent waste of bandwidth due to superfluous replies.
- Bit 25, host addresses use IPv6. All IP addresses are sixteen bytes instead of four. If XML encoding is used the standard a:b:c:d:e:f:g:h notation is used for addresses, instead of $a.b.c.d$.

The available message types are presented later.

Payload size holds the number of bytes in the payload or, if the size multiplier option is enabled, one eighth of the total size. In that case this number is multiplied by eight. Note that this requires the number to be stored in a 32-bit variable, since using a 16-bit variable there is a risk of numerical overflow when performing the multiplication. This value is unsigned.

7.4.9 Message types

The protocol defines 12 message types. 4 of these are request and response message pairs and finally there are 4 one-way messages. It would be possible to extend the protocol with additional response messages, which could simplify the management of firewalls and NAT routers. The corresponding message type values have been classified as reserved for future use. The following messages are defined. The message type value is shown in hexadecimal notation in parenthesis. A convention is used such that response messages have the least significant bit set.

Peerinfo request (0x0000) requests information about a peer in the network. Contains the senders key, number of mirrors, children and parents, the session uptime (in seconds), bandwidth capacity and an approximation on the fraction of bandwidth in use.

Peerinfo response (0x0001) returns information about ourselves. Contains our key, number of mirrors, children and parents, our session uptime,

our bandwidth capacity and an estimation on the fraction of bandwidth used.

List request (0x0002) fetches host lists from a peer. Possible host lists include mirrors, children and parents. There is room for extension of additional lists.

List response (0x0003) contains address/port/key triples from the lists requested. There is room for extensions with additional lists.

Join request (0x0004) ask a peer if we can join as a mirror using a given key. This message is routed through the network to hosts with the corresponding key.

Join response (0x0005) provides feedback on our attempt to join as a mirror.

Changekey request (0x0006) tells another peer that we are changing our key and what change we are doing.

Changekey response (0x0007) reserved for future use.

Register request (0x0008) registers the set of shared keys with the network. This is usually performed once a connection has been established.

Register response (0x0009) reserved for future use.

Unregister request (0x000A) unregisters the set of shared keys with the network. This is usually performed just before connections are closed.

Unregister response (0x000B) reserved for future use.

Leave request (0x000C) tells a peer we are disconnecting from it. It does not necessarily mean we are shutting down, but it means that we are not making a minor change to our key. We are performing something that the peer should interpret as leaving them.

Leave response (0x000D) reserved for future use.

7.4.10 Using XML encoded messages

All message types can be either binary or XML encoded, if the connection initialization contained the exchange of `Accept-encoding` headers listing XML as a supported encoding. If XML is supported the XML bit in the message header can be used to indicate an XML encoded message. The character encoding should also be specified when using XML encoded data. If possible, avoid using the *other* character encoding in the header.

The structure of the XML payload is dependent on the message type. All DTDs are given in an appendix. The names of all tags are in lower case and uses the same names as the corresponding fields in the PDUs (Protocol Data Units) presented later. Any field described later containing # should have a tag containing the letters `num` instead of the #. Figure 7.5 on the following page shows an example of a `peerinfo` request message encoded as XML and figure 7.6 on the next page shows an example of a `list` response. The `peerinfo` request is handled by sending a `peerinfo` response. This response contains similar information.

```
<peerinfo>
    <key>Base64 encoded key</key>
    <numbits>19</numbits>
    <nummirrors>2</nummirrors>
    <numchildren>7</numchildren>
    <numparents>3</numparents>
    <uptime>Number of seconds uptime in this session</uptime>
    <bandwidth>Total bandwidth in bytes/second</bandwidth>
    <bandwidthused>Bandwidth percentage in use 0--100</bandwidthused>
</peerinfo>
```

Figure 7.5: Example of a `peerinfo` XML request

```
<listresponse>
    <list>
        <id>1</id>
        <item>
            <address>IP address</address>
            <port>1232</port>
            <numbits>Number of additional key bits, e.g. 2</numbits>
            <key>Plain text key suffix, e.g. 3 means binary 11</key>
        </item>
        <item>
            <address>IP address</address>
            <port>9726</port>
            <numbits>Number of additional key bits</numbits>
            <key>Plain text key suffix</key>
        </item>
    </list>
    <list>
        <id>2</id>
        <item>
            <address>IP address</address>
            <port>1764</port>
            <numbits>0</numbits>
            <key/>
        </item>
    </list>
</listresponse>
```

Figure 7.6: Example of a `list` XML response

7.5 Payload details

This section describes the details of the payload in the different message types. It uses a notation where the name of the corresponding field in an XML representation is shown in constant width font. The # character should be replaced with `num` in XML documents though.

7.5.1 Exchanging information with `peerinfo`

To know your neighbouring nodes you can request information from them. When requesting information you automatically provide them with information about yourself. This should lower the amounts of `peerinfo` messages needed. The structure of the message is shown in table 7.4. The message types for `peerinfo` messages is shown in section 7.4.9 on page 72. All fields are in network byte order. The key is also in network byte order and right aligned in its field. This means that if any bits should be discarded it is the most significant ones. The most important fields in the `peerinfo` message have the following semantics.

Bytes	Description
0–1	`#bits` Number of key bits
2–3	`#mirrors` Number of mirrors
4–5	`#children` Number of children
6–7	`#parents` Number of parents
8–11	`uptime` Session uptime
12–15	`bandwidth` Total bandwidth
16	`bandwidthused` Percentage (0–100) bandwidth used
17–	`key` key in binary format

Table 7.4: Message PDU for ADTP `peerinfo`

Bits denotes the number of bits in the key at the end of the message. This field is used to calculate the number of bytes, B the key occupies in the message, with the formula below, but the length can also be calculated with the payload length in the message header, $B = (\text{bits} + 7)/8$ or $B = \text{payload length} - 17$. An ambitious implementation checks that these values are equal.

Uptime holds the number of seconds the node has been available without any interruption.

Bandwidth holds the bandwidth capacity of the peer, when there is no load, given in bytes/second.

Bandwidth used is an integer in the intervall 0–100 denoting the percentage of the above bandwidth that is currently in use. The peer is recommended to use an average value over a recent period of time.

Key is the responsibility key of the peer. Any extra padding bits are located at the most significant bits of the first byte.

If the node sending the `peerinfo` message is not a member of any of the three sets (mirrors, children, parents) the key is examined carefully. If the key is equal to the key used by the parents the node is added to the parent set. A node is added to the child set if there exists at least one child with the same key. This behaviour is relevant when a node has joined as a mirror and wishes to notify its surrounding nodes.

A possible extension would be to aggregate the number of known children as the message is passed up the tree. This could allow a parent node to detect that there is an uneven balance between different branches and issue some kind of request that some nodes in the branch with many nodes move over to the other branch, in order to maintain good mirroring levels. This would probably require an additional message to be added to the protocol, and also requires very careful planning to avoid that parent nodes issue such messages too often.

7.5.2 Exchanging host lists with `list`

The `list` request message is used to request information about the peers to which another peer is connected. The information concerning the keys of the hosts returned is relative to the key of peer sending the response. The `list` request message has the format shown in table 7.5 and the `list` response message is shown in table 7.6. All fields are in network byte order. Some notes on the semantics of some of the fields are needed.

Bytes	**Description**
0	`lists` Lists to return. Each bit represents a list
1	`options` Options, reserved for future use.
2–3	`maxanswers` Maximum number of answers per requested list

Table 7.5: Message PDU for ADTP `list` request

Bytes	**Description**
0	Number of returned lists
1–	`list` Zero or more lists
Each list has the following structure	
0	`id` List identifier
1	`options` Options used
2–3	Number of elements
4–	`item` One or more list items
Each list item has the following structure	
0–3	`address` IP address
4–5	`port` Port number where connections are accepted
6	`#bits` Number of additional bits. This number is *signed*
7	`key` Additional bits, if number of bits> 0

Table 7.6: Message PDU for ADTP `list` response

Lists is a set of bits that indicate which lists to request (or which lists are

returned). Predefined values are 1=mirrors, 2=parents, 4=children. Values can be combined using bit-wise or.

Options is intended for future extensions that involve some kind of filtering to be performed on the lists before they are transmitted. The field in the response message should keep all bits set from the incoming options field that corresponded to options used for the list in question. One possible extension would be to request nodes with a key suffix matching a few bits. This could for example be useful if we intend to extend our key. No recommendations on what options should be supported are given.

Address is the IP address used by the host. Support for IPv6 is included as a future extension. If such addresses are used the IPv6 flag must be set in the header. This means that all list items in all lists must use IPv6.

Number of bits is a signed integer that describes the length of the key relative to the key of the peer sending the response. Mirrors have the same key so this number is always 0 for them. Parents have a negative number, depending on how many bits shorter their key is. Children have a positive number, since they have a longer key. Note that this value cannot be applied to an entire list since members of the child set are not required to have keys of the same length. This means that children can have different length keys, but still have the same parent.

Key contains the additional key suffix for children. This field is empty for both mirrors and parents. The number of bytes B in this field is calculated with $B = (\text{bits} + 7)/8$.

7.5.3 Becoming a mirror with `join`

To become a mirror of a certain node the `join` message is used. This simply causes the two nodes to add each other to their buddy sets. It does not affect the other buddies. A node wishing to become a mirror needs to fetch the mirror list from a node and issue `join` messages to all these nodes.

Once a node has become a mirror at some place in the network it can grow or shrink its key, depending on the experienced workload. Changing keys should also consider how it affects mirroring at the new and old levels of the network tree. The format of the join request message is shown in table 7.7 and the response message is shown in table 7.8 on the next page.

Bytes	Description
0–3	`address` IP address of the host
4–5	`port` Port number where connections are accepted
6–7	`#bits` Number of bits in key
8–	`key` The current responsibility key

Table 7.7: Message PDU for ADTP `join` request

If the responsibility key in the `join` request does not match the key of the host receiving the message it should be routed towards nodes where it would

match. If the node wants to become a mirror of the node with whom it is talking the number of bits is set to the maximum value, 65535, and the key field is left empty. This should rarely be done, except the first time the node is ever connecting to the network. The same behaviour can be achieved by first issuing a `peerinfo` message and then a `join` request. The difference is that some bandwidth could be saved if the node is sure it wants to join and does not care what the key for the node other is.

If the host sending the `join` message is a member of the parent or child sets of the receiver, that entry is removed. The purpose of this behaviour is to simplify key changes. See section 7.5.5 on the following page for more details.

Bytes	Description
0–3	`address` IP address of responding host
4–5	`port` Port number where connections are accepted
6–7	`#bits` Number of bits in key
8–	`key` The responsibility key

Table 7.8: Message PDU for ADTP `join` response

If for some reason the node refuses to accept the new node as a mirror the number of bits in the response is set to the *maximum value* 65535 (or -1 if a two byte two's-complement signed integer is used to hold the value) and the key field remains empty. This should rarely happen. Note that the response message contains the key of the responding host. This key is not necessarily the same key that was contained in the request message, but rather the closest key that could be found when routing the message. The key in the response message should be the new responsibility key, unless the response indicated an error, i.e. number of bits is set to 65535.

Upon receiving a `join` response it is common to send a `peerinfo` request if a previously unknown host responded to the join message. The node issuing the `join` message is responsible for contacting other mirrors, along with parents and children. This is usually performed by first issuing a `list` message to obtain the addresses of the hosts in question and then send `join` messages with the no-reply bit set in the header to the mirrors, and `peerinfo` messages to notify the children and parents. If the node already has sufficient knowledge about the parents and children the no-reply bit can be set on the `peerinfo` messages to save bandwidth.

Since nodes that are not directly connected to us could respond to a `join` request we should expect that a new connection attempt could arrive, from the responding host. This causes problems if we are behind a firewall. An alternative is to add a unique identifier to the `join` request message and place the same identifier in the `join` response message. Nodes that nodes routing `join` requests are then forced to save the identifier along with a reference to the socket from which it arrived. The `join` response can then be routed back the same way that the request came, but it makes the message routing a little more complicated since identifier registries are involved. Another possible solution is the one discussed in section 7.4.3 on page 68.

7.5.4 Disconnecting from nodes with `leave`

To leave a node, perhaps to disconnect or to perform more advanced changes to the key, the `leave` message is used. It is the opposite of `join`, except that it is not routed. When receiving a `leave` message the host sending the message should be removed from any of the maintained sets. The structure of the `leave` message is shown in table 7.9.

Bytes	Description
0–1	`#bits` Number of bits in key
2–	`key` The current responsibility key

Table 7.9: Message PDU for ADTP `leave` message

7.5.5 Changing keys with `changekey`

The `changekey` message is used to tell other nodes that we are changing our key, and how we are changing it. There are two operations, *grow* and *shrink*. When a key grows the additional bits are specified in the message. The format of the `changekey` request is shown in table 7.10.

Bytes	Description
0	`#bits` Number of bits changed. This number is signed
1–	`key` New added key suffix if it is an extension

Table 7.10: Message PDU for ADTP `changekey` request

If we intend to change our key it should be sent to all our buddies, which means that they move us from their buddy set to either the parent or the child set. This means that a `changekey` message needs to be followed by a number of `join` messages to nodes at the new level. To save bandwidth the no-reply flag could be set in the header, if a response is not needed. These `join` messages causes the nodes at the new level to update their parent or child sets if the node changing its key was previously a member of one of those sets.

The `changekey` message can also be used to bypass parents. First our parents must be notified that we are leaving, with `leave`. Then a `join` message is sent to the grandparents, using their key or a key length of 65535 if we do not know their key. Then we send a `changekey` message to the grandparents, restoring our previous longer key.

A possible extension involving a `changekey` response message would be to allow a node to deny key changes. If a key change is denied another node that could grant the change must be returned. This could be used for multi-bit extensions in heavily loaded conditions. A node with heavy workload would prefer to delegate the work of a multi-bit extension to a child with a shorter key extension.

7.5.6 Registering shares with `register` and `unregister`

When connecting to the network your shares must be registered and the keys distributed among the hosts responsible for those particular keys. Registration requires a node to have a valid responsibility key, which can be obtained by joining (that is sending a `join` request message to become a mirror of some node).

Furthermore, you need to be able to route messages, which means that you need to know your buddies, parents and children. These is easily obtained using the `list` message. Once your node is initialized with the necessary knowledge to route queries `register` messages can be issued. Each register message contains information about the host storing the files and a list of keys. The format of the `register` message is shown in table 7.11. The `unregister` message is identical, except for the different message type value in the header.

Bytes	**Description**
0–3	`address` IP address of node issuing the registration
4–5	`port` Port number where *down-/upload* requests are accepted
6–7	`#shares` Number of shares
8–	`share` One or more shares
Each share has the following structure	
0–1	`#bits` Number of bits in key
2–	`key` Share key (probably MD5/SHA hash)

Table 7.11: Message PDU for ADTP `register` and `unregister`

The routing of `register` and `unregister` messages is a little special, somewhat similar to multicast routing. If node n is responsible for key $k = 0100110$, has both children and parents and wishes to register shares with key $k_1 = 010011010110$ and $k_2 = 010100101101$ the following steps occur:

1. Since k is a prefix of k_1, n is responsible for routing this key downwards (to its children). Since k is not a prefix of k_2 n looks up the most suitable parent(s) it can find in its registry.

2. Two `register` messages are issued. The one containing k_1 is sent to the child whos' key has the longest prefix of k_1. The one containing k_2 is sent to a parent. If possible a parent node whos' key is a prefix of k_2 is chosen. This could involve looking further up the tree, if n happens to have knowledge on nodes located there. If not, the node is forwarded to an arbitrary parent.

3. The messages containing k_1 and k_2 are forwarded until they reach nodes that are responsible for the keys. This either means that the nodes do not have any children with longer prefixes, or that a node had a prefix equal to the key.

4. Information about the registrations are stored at the nodes. A node can forward registrations to its buddies for better fault tolerance.

If several keys are sent in a message the keys are separated into different messages when reaching a point in the network where the keys do not share a common prefix of the required length when routed downwards.

7.5.7 Performing searches with `query`

Query requests are very much similar to `register` and `unregister` messages. They are infact so similar that the only difference is the message type in the header. See 7.12 for a description of the fields in a `query` request message. The routing of `query` requests is identical to the routing of `register` and `unregister` messages.

Bytes	Description
0–3	`address` IP address of node issuing the registration
4–5	`port` Port number where connections are accepted
6–7	`#shares` Number of shares
8–	`share` One or more shares
Each share has the following structure	
0–1	`#bits` Number of bits in key
2–	`key` Share key (probably MD5/SHA hash)

Table 7.12: Message PDU for ADTP `query` request and response

Nodes that receive `query` requests, and have items matching the key of the query, should open a connection with the host specified in the `query` request. This also causes problems with firewalls. Possible solutions to this problem is discussed in sections 7.4.3 on page 68 and 7.5.3 on page 77.

The `query` response message also has the same structure as `query` requests. The only difference is that the port number specifies the port from which download and uploads are accepted. The download can be performed using a suitable protocol, such as HTTP.

7.6 Practical examples

Since the protocol has not been implemented and tested in practice here are some examples on how it is intended to work. If it requires a programming wizard to make it work this way is not known, but a well educated guess is that it should not be impossible to implement for a decent programmer.

7.6.1 Bootstrapping the network

If there is no network one needs to be constructed automatically by the nodes, using the proposed protocol. This example consists of a number of steps that occurrs in the network, but does not fully show the parallelizm. The example starts of from nothing at all.

1. Given a few nodes that know nothing the network can start to be constructed. In this step the few nodes connect to each other and become buddies. Since the number of nodes is very low there is only one level of nodes.

2. As more nodes arrive, the workload increases. One peer decides that it is time to do something about it and extends its key by one bit, assume it is 0. Other nodes soon follow and extend their keys with 0 or 1, trying to keep a reasonably even balance.

3. When there are enough nodes responsible for the key prefixes 0 and 1, the top-level nodes feel that they do not have to maintain mirrored data from the level below. They simply trust that they will be able to find someone else to do the job for them. This lowers the workload on the top-level servers.

4. As more and more nodes join the network the tree grows downwards with additional levels. As the workload increases more nodes extend their keys. This could lead to that a level in the middle of the tree becomes exhausted, since there are too few nodes who wants to work at that level. This causes the nodes on the lower level to issue a `changekey` with their grandparents, maintaining their current key. If nodes are powerful and have a lot of available resources other nodes could choose to connect to the node and extend by more than one bit since the parent node will maintain lots of child nodes. Both of these causes the tree to fan out.

5. As the network grows mature it consists of clusters of buddies responsible for the same key and those clusters are ordered by their key in a tree that could be heavily fanned out.

It is important to mention that once a suitable location in the network has been found a node should only rarely have to change its key. Since mirroring and local caching of addresses should be large it is likely that at least one desired host is availble, without having to probe the network constantly to maintain connectivity. If a node fails to connect to some other nodes it issues a new `list` request to obtain addresses of additional peers.

7.6.2 New host joins the network

When the network has been initialized a new host joins the network. This example shows the steps performed in order to find a suitable place to join.

1. The host connects to an arbitrary node and reads the information about the node. If the node appears to be suitable buddy it is joined as a mirror. If we consider ourselves stronger than this node in terms of network capacity, and want to contribute more to the network the parent list of the node is requested. From the parent list new nodes can be connected and the operations are repeated. The node could also choose to request the child list of a node, if it feels that it does not have enough capacity.

2. Finally a suitable node is found, and the node probably has knowledge of quite a few other nodes already. The new node issues a `join` request using a key length of 65535 and an empty key field. Alternatively could the key obtained in a recent `peerinfo` message be used. Given the semantics of the `peerinfo` message, it could also be used by specifying the key previously obtained. If this is performed the number of

mirrors, children and parents should be set to 0. Otherwise there is a potential danger that the other node wants to receive the lists of all those new hosts that this new node knows about. This would only waste bandwidth.

If a node has been connected before and found its place in the network it is easy to reaquire the same responsibility, as this example shows.

1. The node simply connects to an arbitrary node in the network, although a node with a closer key is better.
2. When issuing a `join` message with the previously used key of the peer it is forwarded through the tree to the node with the closest matching key.
3. The peer with the matching key responds, and tells the node where it can connect.
4. The node can then obtain fresh lists from the part of the tree it is working in, and hence quickly get in touch with the necessary nodes.

Chapter 8

Framework design

In this chapter the design of a peer-to-peer framework is presented. The purpose of this chapter is to show the kinds of problems encountered by peer-to-peer applications and present a way to solve them.

8.1 Introduction

The design presented in this chapter is influenced by object oriented technology. It has been implemented in Java and has undergone some basic testing. It has not been used in any large scale peer-to-peer projects yet, though. The goals of the framework can briefly be summarized as follows:

- The framework should be easy to understand for developers with knowledge in object oriented languages and basic network programming.
- The framework should be modular and reasonably easy to extend and customize.
- The framework should be general enough to cover a broad range of possible applications.
- The framework should provide the application programmer with a reasonably simple interface and make it easy to add the details needed to form a complete application.

The goals are quite ambitious, but without ambitions humanity would not have come this far. To provide the necessary levels of abstraction the framework is divided in a set of core components that are described in the coming section. The chapter also contains some ideas on how to provide additional abstractions for specific a application.

8.2 Backend components

The tasks of the system not directly related to the application, such as managing a GUI (Graphical User Interface) or protocol specific details, could be described as follows:

- Monitor a set of server sockets for incoming connections. Each protocol used has at least one well-known port number where connections are accepted.
- Initialize connections if the protocol requires it. This includes exchanging capability headers, negotiations regarding encryption and authentication and other things that are specific to a certain protocol when initializing a new connection.
- Monitor initialized connections for incoming messages and handle them appropriately.
- Make it possible to add and remove protocols from being used. It could possibly be useful to be able to provide the ability to temporarily disable a protocol from being used and later be able to enable it easily. This would be especially important if the work required to add and initialize a protocol module is significant.
- Allow the application to receive notifications when important events occur. For example, a Gnutella file sharing system needs to present the responses for a query to the user, and hence a notification of an incoming `QueryHit` message is necessary.
- Allow the application to easily access the underlying network. For example, a file sharing system needs to send queries easily.

There are many designs that could meet the demands of these tasks, but here follows one set of components that together provide a solution. Throughout the following sections the names of the components are abbreviated as follows:

- `pcm` means Peer Connection Monitor
- `pm` means Peer Monitor, or could possibly also be interpreted as Protocol Monitor
- `pr` means Peer Reader, or Protocol Reader
- `pp` means Peer Protocol
- `pbe` means Peer Back End

8.2.1 Peer Connection Monitor

The `pcm` is responsible for monitoring a set of server sockets, where connections can be accepted. It consists of one thread that performs the following tasks until it is explicitly terminated:

1. Wait until connections arrive
2. Tell the `pm` for the appropriate protocol to add the connection

The `pcm` is also responsible for maintaining registrations on what port numbers to monitor and what `pm` to use for each port number.

8.2.2 Peer Monitor or Protocol Monitor

The pm is responsible for monitoring a set of client connections for a specific protocol. Each protocol used has its own pm and each pm consists of one thread that performs the following tasks until it is explicitly told to terminate:

1. Wait until data arrives at one of the client connections
2. Use the corresponding pr to read data into a buffer. The pr detects when a complete message has arrived and extracts it
3. If a complete message has arrived, create an appropriate job to handle the incoming message and, tell the pbe to schedule the job for execution

The pm also registers itself with the pbe as the callback for connection initialization jobs for its protocol. This means that the pm will be notified whenever a connection initialization job for protocol used by the pm is completed. When this occurs the pm takes the following actions:

1. Assert that the connection is valid and should be accepted
2. If validation was ok, register the connection for monitoring. This means that the connection is now fully connected and initialized
3. Notify the pm thread that the set of monitored connections has been modified

The pm is also used by the pcm to add new client connections. When such a request is made by the pcm the following steps occur:

1. Verify with the pbe that a new connection can be accepted, and discard the connection if it failed
2. Register the accepted connection as being in its initialization phase
3. Create an initialization job for the connection
4. Tell the pbe to schedule the initialization job for execution. When this job is completed the pbe will notify the pm as described above

This approach places all protocol specific details in the pm and the jobs it is responsible for creating when messages arrive.

8.2.3 Peer Reader, or Protocol Reader

When data arrives over the sockets monitored by the pm this component is responsible for doing the actual reading. Data is stored in a buffer and, when a complete message has arrived, the complete message is returned to the pm.

The pr is preferably implemented using non-blocking I/O routines. Such an implementation allows the number of connections to be large and still keep the system responsive.

8.2.4 Peer Protocol

Both the `pm` and the `pr` are protocol specific. Many aspects of their functionality is protocol independent, but additional details for each protocol needs to be added. To encapsulate a protocol the `pp` is used. It contains the corresponding `pm`, `pr` and well-known port number.

8.2.5 Peer Back End

The `pbe` is responsible for managing all protocols and dispatching notifications of events. Is also responsible for coordinating the number of connections between the different `pm`s, making sure that the number of simultaneous connections does not exceed a certain threshold. The following services are provided to other components:

- Add/remove a protocol (that is `pp` components)
- Enable/disable a previously added protocol
- Register a callback for the completion of a specific job type
- Withdraw a previously made callback registration

It is also used by other components to schedule tasks for future execution. The scheduling and execution is performed using that a thread pool with a priority based scheduling policy. The `pbe` also registers itself as the callback for the thread pool. This means that the `pbe` is notified whenever a job is completed in the thread pool. When this occurs the thread performing the job in the thread pool notifies the `pbe` and performs the following:

1. Add the job to the queue of completed jobs
2. Notify the `pbe` thread that a job has completed

The `pbe` also contains a thread that is responsible for dispatching calls to registered callbacks. It runs until it is explicitly told to terminate and performs the following tasks:

1. Wait for a job to be completed
2. Look if a callback has been registered for the job type that just completed
3. If a callback is found, call it

8.3 Frontend components

To construct a useful application additional components are required. Besides providing a reasonable user interface an additional level of abstraction with the following properties would simplify the construction of a useful application:

- Hide protocol differences. A file sharing system could for example provide the application with the possibility to make a query. The query would then be relayed to all the available protocols

- Provide another level of event dispatching. The benefit of this would be that the parameters of the actual event handlers can be of arbitrary types. This also helps to hide protocol differences. For example, a file sharing application sending a query as mentioned in the previous item will receive replies from different protocols. The frontend component could then translate all replies to a common form and dispatch them to the *same* event handler in the application regardless of what protocol generated the response.

The frontend components are usually application specific. Besides a main frontend component that has the above mentioned properties a set of job components needs to be constructed. When the user performs some action that requires the underlying framework to be used an appropriate job component is created. This job is then scheduled for execution in the backend. Such a job would usually work with the interface provided by the frontend, to benefit from protocol abstractions etc.

Any complete application requires some kind of user interface. Since the underlying peer framework is based on threads and event dispatching it works well with window based environments. Most window based environments available work in similar fashion, with a large set of events and event handlers and a thread that dispatches events to the appropriate handler. A separate thread that waits for user input is often used and when the user performs some action, such as clicking the mouse, an event is created and added to a queue. The event dispatcher continuously examines the queue and dispatches events.

8.4 Support components

The components described earlier in this chapter requires the help of several additional components to work in practice. This section describes the most important support components and presents suggestions on how they can be implemented.

8.4.1 Thread pool

A thread pool is a component that maintains a set of *worker threads* and a queue of jobs to be performed. Other components add jobs to the queue and the worker threads pick up the jobs and executes them. This means that synchronization of the queue is important.

The queue does not necessarily have to be a FIFO queue. Any component to which elements can be added and later removed from could be used. If a priority queue is used the jobs can have different priority levels.

A thread pool can be implemented by using a linked list to simulate the queue of jobs. The set of worker threads is represented by an array of worker thread components. A possible extension to a thread pool could be to set a limit on how long time a single job may take. If a job exceeds this time it can be discarded. For more advanced job scheduling a priority queue can be used.

8.4.2 Priority queues

A priority queue is a well known abstract data type (ADT). It usually supports the following operations, although the names of the operations can vary. All objects stored in the priority queue needs to be comparable. There must exist an ordering of all the elements. If a comparison considers two elements to be equal their internal ordering in the priority queue is unspecified. The comparison between the objects can be based on a separate priority value, or some other property of the object itself.

`add(object)`
: Adds `object` at the right place in the priority queue.

`removeNext() → object`
: Removes and returns the next object from the priority queue.

`hasNext() → boolean`
: Checks if there is at least one element available in the priority queue.

Additional operations are possible, but the ones mentioned above are the most important. There are many ways to implement a priority queue. An efficient implementation uses a *binary heap*. A binary heap allows for logarithmic time insertions and removals. A binary heap is a balanced binary tree possessing the *heap order property*:

> In a heap where keys are ordered in increasing order, for every node n with parent node p then $\text{key}(p) \leq \text{key}(n)$.

A binary heap can be implemented using a linked structure, or using a growable array. If the growable array is indexed from 0 the following is one of the easiest ways to represent a binary heap:

- Position 0 is reserved for internal use in the heap. It is used by the percolation algorithms that are responsible for maintaining the heap order property.
- If position i contains an element its parent is at position $i \div 2$ and its two children at positions $i \times 2$ and $i \times 2 + 1$.

A good example in C++ of a binary heap implemented using a growable array used in the fashion mentioned above is presented in [Wei00]. Such an implementation has very efficient insertions, which on average requires only 2.6 comparisons. Most books on data structures contain thorough discussions on priority queues.

8.4.3 Dictionaries for registrations

Dictionaries, sometimes called maps or associative arrays, store (key, value) pairs. The three most important operations of a dictionary are the following, although different names can be used:

`add(key, value)`
: Adds the pair (`key`, `value`) to the dictionary.

find(`key`) → `value`
: Finds and returns the `value` whos' key is equal to `key`.

remove(`key`) → `value`
: Removes the pair whos' key is equal to `key` and returns its `value`.

There are lots of possible implementations of dictionaries. Anything from a simple list to a balanced binary search tree or hash table is possible. Programming languages such as C++ and Java already provide good implementation of dictionaries, which makes it a waste of time to implement your own, although it has some educational benefits to do so. Here is a small list of possible implementations, but for more details see a good book on data structures.

- A simple array can be used if the number of items is relatively fixed. If the usage pattern is that the dictionary is first initialized with a set of items, then a large set of lookups is performed and finally the dictionary is destroyed it is probably worthwhile to sort the array using the keys. This allows the find operation to use a binary search algorithm, reducing its running time from linear to logarithmic.
- A linked list can be used instead of an array if the number of items is relatively small and the operations are rather random.
- A binary search tree can be used to increase the performance compared to the linked list, although a simple binary search tree has the same theoretical worst case performance.
- A balanced binary search tree, such as an AVL tree or a red-black tree can be used to maintain high performance even in the worst case.
- A hash table can be used if maximum performance for lookups is desired. Using a good hash function is vital to its performance, but this is often a good choice.

8.4.4 Sets

Sets are collections of objects with no duplicates. A set is usually implemented using similar data structures as a dictionary. For a very simple implementation an array or a linked list can be used, but for better performance a balanced tree or a hash table are more suitable choices. The tree alternative requires that the elements of the set can be ordered though. The most important operations on a set is the following:

add(`object`)
: Adds `object` to the set.

contains(`object`) → `boolean`
: Checks if the set contains the given `object`.

remove(`object`)
: Removes an `object` from the set.

As with dictionaries sets are often available in some library ready to be used. For implementation details see a data structures book.

8.4.5 Input multiplexing

When constructing network based services it is common that several connections must be managed at the same time. This does not include peer-to-peer services alone, but also client-server applications. If a server maintains several simultaneous connections to clients there are two ways that the connections can be monitored:

1. Use polling and check connections, in a round-robin fashion, using non-blocking calls.
2. If the underlying system, or programming language, supports it a blocking multiplexing approach can be used. This involves setting up a set of inputs that should be monitored. After this a single blocking call is made, which monitors all inputs at the same time. When data arrives at one of the inputs the call terminates and it is possible to determine which inputs have new data waiting to be processed.

The second approach is naturally preferable. Unix operating systems provide the functions `select(2)` and `poll(2)` that both allow the programmer to use input multiplexing. Java as of version 1.4 provides similar abilities using its `nio` package.

Chapter 9

Implementation

This chapter presents the implementation details of an efficient peer-to-peer framework. Focusing on components and their interaction this chapter gives an understanding of how peer-to-peer systems could be implemented.

9.1 Introduction

Peer-to-peer systems are quite different from traditional client/server based network application when it comes down to the actual implementation. If you want to implement a peer-to-peer system you need to use more sophisticated techniques to make the system run smoothly. Developing peer-to-peer applications from scratch can be done, but many applications solves the same problems. Some of these are:

- Manage a changing set of connections to hosts that could disconnect at any time that have varying bandwidth and delay.
- Manage new incoming connections.
- Perform protocol specific tasks, e.g. managing a certain type of incoming messages.
- Notify the application of important events.
- Allow the application to add user generated tasks, such as a task that performs a query in a file sharing system.

In chapter 8 on page 85 a set of framework components were presented. The main part of this chapter contains the details on how these components can be implemented in Java using the Java Development Kit (JDK) 1.4.

9.2 Overview

This rest of this chapter assumes that you have knowledge on basic Java programming. More specifically you should know about the following things before you proceed:

- Basic Java application development.

- Programming with basic TCP/IP in Java.
- Programming with threads and basic synchronization in Java.
- Knowledge of the utility classes of Java, most notably the container classes.
- Familiarity with the concepts of events and event notifications.

No knowledge on using the `java.nio` package is assumed. The necessary details on this package are presented later.

9.2.1 Core classes

There are eight classes that either provide important functionality to the framework or are especially important when the framework is used to construct a peer-to-peer application. In alphabetical order these classes are shown below. Any names to the right of the class name denotes super classes and directly implemented interfaces.

HeapPriorityQueue — `OrderedQueue`

Implements a binary heap using a `Vector`. It contains instances of `ThreadJob` and is used by `ThreadPool` to schedule jobs for execution.

PeerBackEnd — `Thread`, `Callback`, `Terminatable`

Is the main backend component. It maintains protocol and event callback registrations. It also manages the `ThreadPool` and the dispatching of events. This component also controls the number of simultaneous connections that can be maintained and manages the `PeerConnectionMonitor`.

PeerConnectionMonitor — `Thread`, `Terminatable`

Monitors a set of server socket channels for incoming connections using a `Selector`. Each server socket channel is bound to a specific port number. When a new connection arrives the `PeerMonitor` is responsible for managing the connection.

PeerMonitor — `Thread`, `Callback`, `Terminatable`

Is the main protocol specific component. It monitors a set of socket channels, that use the same protocol, for incoming data. It uses the `PeerReader` to read incoming data, and when a complete message has arrived a job is created to handle the message. This class is also responsible for handling the connection scenario for the protocol the monitor implements. This class is subclassed for each protocol that should be supported.

PeerProtocol

Is a holder for components that provide a protocol implementation. These are `PeerMonitor`, `PeerReader` and the port number from which new connections are accepted. `PeerProtocol` objects are registered with the `PeerBackEnd` using a string identifier.

PeerReader

Is responsible for reading incoming data in a specific protocol and storing the data in buffers. When a complete message has arrived the `PeerMonitor` is notified. This class uses non-blocking operations to read data into a `ByteBuffer`. `PeerReader` also maintains registrations between channels and buffers. It is subclassed for each protocol that should be supported.

ThreadJob `Runnable`, `Prioritized`

Is a simple abstract class that can be executed by threads and maintains a priority level. The priority level is used in the scheduling policy in the `ThreadPool`. This class is created in many different subclasses since each one implements a certain scenario, such as the management of an incoming message of type A.

ThreadPool `Terminatable`

Implements a pool of threads that execute tasks assigned to it, all implementing the `Runnable` interface. Allows another component to register itself as responsible for event notification. In the framework `PeerBackEnd` takes on this role. Whenever a job is completed the `PeerBackEnd` is notified, which in turn is responsible for more detailed dispatching.

9.2.2 The Java `nio` package

The `nio` package is new in JDK 1.4 and provides two important additions to the I/O system:

- I/O multiplexing
- Non-blocking I/O

The most important class for I/O multiplexing is the `Selector` class. Other components, known `nio` terminology as *channels*, can be registered with a `Selector` for one or more of the constants shown in table 9.1, found as `static` members of the `SelectionKey` class. The constants can be combined using the *bit-wise or operator*. Each registration yields a `SelectionKey` that identifies the channel and the registered operations.

Constant	Description
`OP_ACCEPT`	Accepting socket connections operation
`OP_CONNECT`	Connect socket operations
`OP_READ`	Read data operation
`OP_WRITE`	Write data operation

Table 9.1: Defined operations for Java `nio` multiplexing registrations

Instances of `Selector` are created using the `SelectorProvider` class. The `SelectorProvider` class is also used to create channel instances, such as instances of `SocketChannel`.

When registrations have been performed the `Selector` object calls one of its `select` methods. This method waits for any of the registered operations

to occur on the corresponding channels and returns the number of *keys* that are ready. The set of keys can then be retrieved from the `Selector` and each key can be used to obtain the corresponding channel.

Non-blocking I/O can be performed by using special *buffers* that are a part of the `nio` package. Data can be read or written to or from such buffers in non-blocking fashion. The buffers are responsible for keeping track of the number of elements read or written so far, which makes them easier to use than simple arrays. It is also easy to reuse a buffer once an operation has been completed, which reduces the time spent on allocating and releasing memory.

9.2.3 System initialization

To initialize the framework several objects needs to be created and several calls needs to be made. The order in which these operations are performed is fixed, since later calls usually require objects created earlier to be passed as arguments. To start the backend the following steps are performed:

1. Create a `PeerBackEnd` instance with a specified number of threads for the thread pool and a threshold of concurrent connections. This also creates a `ThreadPool` instance and a `PeerConnectionMonitor` instance.

   ```
   pbe = new PeerBackEnd(numThreads, maxConnections);
   ```

2. For each protocol *`Proto`* to use, create its `PeerMonitor` and `PeerReader`. Start the `PeerMonitor` thread, and create a `PeerProtocol`. Register the `PeerProtocol` with the backend and enable it (if you want to use it).

   ```
   pm = new ProtoMonitor(pbe);
   pr = new ProtoReader();
   pm.start();
   pp = new PeerProtocol(port, pm, pr);
   pbe.addProtocol(pp, name);
   pbe.enableProtocol(name);
   ```

3. Start the threads in the backend. This also causes the threads in the thread pool and the connection monitor thread to start their execution.

   ```
   pbe.start();
   ```

Before issuing the `pbe.start()` call it is probably necessary to make a number of calls to `pbe.addCallback(Callback, String)` to register callbacks for events. These callbacks are called whenever a corresponding job is completed in the thread pool and is dispatched by the `PeerBackEnd` thread. The `String` argument of the call must match the string returned by the method `getID()` in the corresponding job.

9.2.4 System shutdown

Shutting down the system is easier than starting it. Only the following steps are necessary to shutdown all threads.

1. For each installed protocol name, `name`, uninstall the protocol and tell its `PeerMonitor` thread to terminate.

```
pp = pbe.removeProtocol(name);
pm = pp.getMonitor();
pm.terminate();
```

2. Tell the backend to terminate. This causes the connection monitor thread and all threads in the thread pool to terminate.

```
pbe.terminate();
```

9.3 Protocol modules

The possibility for an application to use several protocols and making the rest of the framework independent of the protocols used is a very important aspect. The framework assumes that all protocols use TCP as the underlying protocol, which to some extent limits the flexibility of the framework. On the other hand, this decision makes it possible to put more intelligence into the framework and lower the burden on the application programmer.

A possible extension to the framework would be to implement an abstraction over the actual underlying protocol. This could involve some problems with connectionless protocols, such as UDP, but it is probably possible to implement such an abstraction without rewriting to much of the framework code. One possibility would be to extend the `PeerProtocol` class in such a way that it specifies if it should be passed to the `PeerConnectionMonitor`, or if the protocols `PeerMonitor` is responsible for the requests from new hosts.

9.3.1 The `PeerProtocol` class

The `PeerProtocol` class is a placeholder for protocol specific components. It is registered with the `PeerBackEnd` when it is intended to be used. This component does not provide any significant functionality except encapsulating the protocol specific components. The most important methods of this component are:

getMonitor`()` → `PeerMonitor`
: Returns the monitor specific for this protocol.

getReader`()` → `PeerReader`
: Returns the message reader specific for this protocol.

9.3.2 The `PeerReader` class

This abstract class is the superclass of all protocol specific reader classes. It maintains registrations of readable channels and their corresponding message buffer. When data arrives over a particular channel the reader class is responsible for reading data into the corresponding buffer. When a complete message has arrived its is returned to the calling object. The most important methods are:

register`(ReadableByteChannel rbc, int capacity)` → `boolean`
: Registers channel `rbc` with a buffer with the specified capacity. Returns true if the registration was successful, false otherwise.

`read(ReadableByteChannel rbc) → ByteBuffer`

Reads bytes from channel `rbc` and stores them in the corresponding buffer. If a complete message has arrived that message is moved to a separate buffer, which is returned. If no complete message has arrived null is returned. This method uses non-blocking I/O. The decision of determining if a complete message has arrived is delegated to subclasses.

If end of file is reached when reading from `rbc` the channel is closed, unregistered from its buffer and null is returned. The calling object (in this case an instance of `PeerMonitor`) is responsible for detecting that the channel has been closed, if any further actions should be taken.

9.3.3 The `PeerMonitor` class

The `PeerMonitor` abstract class is the most complicated class in a protocol plug-in. Each protocol subclasses this class and adds the protocol specific details. Depending on the protocol used this can range from a simple to a very big task. Most of the work however, when implementing a protocol plug-in consists of constructing many smaller job classes that perform a protocol specific job. This could for example include managing an incoming message of a certain type.

The `PeerMonitor` abstract class has several responsibilities. It is responsible for connection management for the protocol it implements, although protocol specific details are delegated to subclasses. This responsibility includes registering the channel for a new connection as *pending* while it is being initialized. When the initialization completes, and it was successful the channel is registered as *connected* and it is added for monitoring.

All channels that have been added for monitoring are monitored using a `Selector` that waits for incoming data on all connected channels. The `PeerReader` is used to read messages and the work of creating the appropriate jobs for managing messages are delegated to subclasses.

The following methods are the most important ones in the `PeerMonitor` abstract class:

`addConnection(SocketChannel sc, boolean incoming) → int`

Registers a new socket channel for monitoring. The channel is registered as pending and a job for initializing the connection is created. Return 0 on success, 1 if the `PeerBackEnd` did not allow any more concurrent connections or -1 if the channel had already been registered. A call to the `createConnectionJob(SocketChannel, boolean)` method is performed to create the job that initializes the connection.

`removeConnection(SocketChannel sc) → int`

Removes a socket channel from the monitor. If the channel has been successfully registered it is removed at once. If the channel is pending, and hence being initialized, it is scheduled for removal as soon as the initialization completes. Returns 0 on success, 1 if the channel was pending and will be removed later and -1 if the operation failed.

`callback(Object job) → Object`

Called automatically by the `PeerBackEnd` when a connection has been initialized, which means that a initialization job has completed. The

`job` is examined to see if the initialization was successful, which is a task that is usually delegated to subclasses. The current implementation of `validateConnectionJob(Object)` only checks to see if the job is an instance of `ConnectionThreadJob`. If the initialization was successful the channel is registered as connected and the `Selector` is notified that changes to the set of monitored channels have been performed. This method always returns null.

`run()`
: The main execution method of the `PeerMonitor` thread. Uses a `Selector` to wait for incoming data on the connected channels. Uses the `PeerReader` to read messages and when a complete message has arrived a job is instantiated to manage the message. Which job to instantiate is delegated to subclasses. The job is then executed in the `PeerBackEnd`.

9.3.4 Example scenario: Managing an incoming message

Assume that a protocol plug-in named `Proto` has been implemented, initialized and registered and, that a connection with some other peer using this protocol has been established. If the other peer sends starts to send a message to us the following things happen.

1. The `ProtoPeerMonitor` is waiting for incoming data using a blocking call to a `Selector`. This takes place in the `run()` method of the base class, `PeerMonitor`.
2. It detects that new data is coming in on at least one channel.
3. The particular channel on which data is arriving is found through its `SelectionKey`.
4. The reader for this protocol, `ProtoPeerReader`, is called using its `read(ReadableByteChannel)` method. The implementation of this method is found in the abstract base class `PeerReader`.
5. The `read(ReadableByteChannel)` method in `PeerReader` reads all data that has arrived into the `ByteBuffer` registered with the channel.
6. It then calls the `isCompleted(ByteBuffer)` method, which is implemented in `ProtoPeerReader`.
7. The `isCompleted(ByteBuffer)` method checks if a complete message is available, and (assuming that it is) returns the number of bytes in that message.
8. Back again in the `read(ReadableByteChannel)` method in the `PeerReader` class, the complete message is moved to a separate `ByteBuffer` and returned.
9. Back in the `run()` method of the `PeerMonitor` class a call to `createJob(ByteBuffer, SocketChannel)` is performed to create a job that handles the incoming message.

10. The `ProtoPeerMonitor` class implements the `createJob(ByteBuffer, SocketChannel)` method any way it sees appropriate and returns the job that handles the message, or null if the message should be discarded.

9.4 Backend implementation

This section outlines the design of the core components in a peer backend.

9.4.1 The `PeerConnectionMonitor` class

The `PeerConnectionMonitor` is a separate thread that monitors all ports that the registered (and enabled) protocols use to accept new incoming connections. The implementation uses a `Selector` to perform the monitoring. It also manages two dictionaries, one mapping a port number to a `SelectionKey`, and the other mapping the same `SelectionKey` to the corresponding `PeerMonitor`.

When a new connection arrives the `SelectionKey` is extracted from the `Selector` and the corresponding `PeerMonitor` is looked up. The `PeerMonitor` is then responsible for setting up the connection properly. The following methods are the most important in the `PeerConnectionMonitor` class:

`add(int port, PeerMonitor monitor) → boolean`
: Adds a new port for connection monitoring and associates it with the given monitor. Returns true if the registration succeeded, false otherwise.

`remove(int port) → PeerMonitor`
: Removes the registrations for the given port number. Returns the monitor associated with the port number, or null if an error occurred.

`get(int port) → PeerMonitor`
: Returns the monitor associated with a given port number, or null if no monitor was found.

`run()`
: Main method of execution for the `PeerConnectionMonitor` thread. Uses a `Selector` to wait for incoming connections. The `PeerMonitor` for the corresponding port is looked up when a connection arrives and the monitor is then responsible for initializing the connection.

9.4.2 The `ThreadPool` class

The `ThreadPool` class is the place where most of the actual execution takes place. Tasks that needs to be performed by the various protocols and the application are added to the thread pool, and are scheduled for execution in one of its' threads.

No jobs that could block indefinitely should be added to the thread pool, since it does not provide any means to discard a job. A job is allowed to block, but should avoid doing so for longer periods of time if it is possible.

If all threads are blocked the system would appear to be unresponsive, or at least slow to respond.

If all threads are busy when a task is added it is stored in a queue. When a thread has completed a job it fetches the next job from the queue and starts to execute it. The threads that perform the work are instances of the class `WorkerThread`, which is a `private` class inside the `ThreadPool` class. The only method worth mentioning in the `WorkerThread` class is:

`run()`
: Fetches the next job from the queue, or waits until one arrives if there are no available jobs. Execute the job and, when the job is completed, notify the registered callback dispatcher. This is usually the `PeerBackEnd`, and hence its `callback(Object)` method is called.

The `ThreadPool` class uses a `Queue` internally to which jobs are added. When the `ThreadPool` is instantiated, the number of threads, a `Callback` and a `Queue` are specified. The `Callback` denotes the object that should be notified whenever a job is completed and the `Queue` specifies the queuing policy.

In the implementation of this framework the `PeerBackEnd` is registered as the `Callback` and a `HeapPriorityQueue` is used as the `Queue`. The implementation used by the framework assumes that the jobs added to the thread pool are subclasses of the `ThreadJob` class, and not simply implement the `Runnable` interface, which is the requirement set by the thread pool. This is because the comparator used in the `HeapPriorityQueue` imposes the additional requirements to the jobs.

The most important methods of the `ThreadPool` class are the following:

`addJob(Runnable job)`
: Adds a new job to the thread pool.

`nextJob() → Runnable`
: Fetches the next job from the queue of jobs, or waits until one is added.

9.4.3 The `HeapPriorityQueue` class

The `HeapPriorityQueue` class implements a standard binary heap priority queue. It uses a `Vector` internally and the implementation is based on the C++ implementation of heaps presented in [Wei00]. The class extends the `OrderedQueue` abstract which is a class that claims to implement the `Queue` interface. It does not do that however, since the particular implementation does this. The `OrderedQueue` only provides a `Comparator` for comparison between elements in the queue. The `Queue` interface has the following methods, where the purpose of each method is self-explanatory.

```
add(Object o)
next() → Object
removeNext() → Object
hasNext() → boolean
size() → int
clear()
rebuild(Collection coll)
```

The `HeapPriorityQueue` uses the `Comparator` provided by the base class, `OrderedQueue`, to structure the heap. An element that is ordered before another element by the comparator is considered to have higher priority.

The framework uses a `PriorityComparator` which orders objects that implement the `Prioritized` interface. The thread pool requires that the jobs implement the `Runnable` interface and the priority queue requires that the elements implement the `Prioritized` interface. The `ThreadJob` class fulfills both of these requirements and is the class that should be used to work with the thread pool and its instance of the priority queue.

9.4.4 The `PeerBackEnd` class

The `PeerBackEnd` class brings the other parts of the framework backend together making them easier for application programmers to use. The class is responsible for callback registrations, which is implemented using a dictionary mapping a string identifier to a `Callback` object. The back end class is the central point of authority that determines if more connections can be accepted and it maintains a set of all currently active connections.

This class is also responsible for the thread pool, the registration of callbacks and the dispatching of calls to these callbacks. All completed jobs are added to a first-in-first-out (FIFO) queue, from which jobs are extracted by another thread and dispatched.

The `PeerBackEnd` also manages the `PeerConnectionMonitor` and the registration of protocol plug-ins. The registration information is stored in a dictionary mapping a protocol name to the corresponding `PeerProtocol`. There are many important methods in the `PeerBackEnd` class:

addJob`(ThreadJob job)`
: Adds a job to the thread pool for later execution.

addCallback`(Callback cb, String id)` → `Callback`
: Registers a new callback for the given identifier. Returns the previously registered callback with the same identifier, or null if no previous registrations have been made.

findCallback`(String id)` → `Callback`
: Locates the registered callback for the given identifier. Returns the callback object, or null if no matching identifier was found.

removeCallback`(String id)` → `Callback`
: Removes a registered callback. Returns the callback object, or returns null if no registration with the given identifier has been performed.

addProtocol`(PeerProtocol p, String id)` → `PeerProtocol`
: Registers a new protocol with a given name. Returns the previously registered protocol with the same name, or null if no previous registration has been made.

findProtocol`(String id)` → `PeerProtocol`
: Locates the protocol with the given name. Returns the protocol, or null if no protocol with the given name was found.

removeProtocol(`String id`) → `PeerProtocol`
: Removes the registration of the protocol with the given name. Returns the protocol, or null if no matching registration was found.

enableProtocol(`String id`) → `int`
: Enables a previously registered protocol with the given name. This means that new connections can be accepted. Returns 0 on success, -1 if a protocol with the given name has not been registered, -2 if the port already is in use and 1 if protocol for some reason could not be added to the connection monitor.

disableProtocol(`String id`) → `int`
: Stops accepting new connections for the protocol with the given name. Returns 0 on success, -1 if the protocol has not been registered and 1 if the connection monitor has a mismatching peer monitor registered.

addConnection(`SocketChannel sc`) → `boolean`
: Attempts to add a new connection. Returns true if the connection has not already been added and the number of currently open connections is less than the maximum. If true is returned the number of currently open connections is increased by 1. If these conditions are not met, false is returned.

removeConnection(`SocketChannel sc`) → `boolean`
: Removes the connection from the set of currently open connections. Returns true and decreases the number of open connections if the connection was previously added successfully. Otherwise returns false.

hasConnection(`SocketChannel sc`) → `boolean`
: Checks if the backend already has registered the given channel as an open connection. Returns true if a connection already has been added, false otherwise.

callback(`Object job`) → `Object`
: Called automatically by the threads in the thread pool when a job completes. Adds the job to a FIFO queue for later dispatching. Always returns null.

run()
: Main execution method of the backend thread. Extracts completed jobs from the FIFO queue, checks if there is a corresponding callback and, if there is one, calls it.

9.4.5 Example scenario: New connection

Whenever a new incoming connection attempt is made by another peer the following steps are taken, for a protocol named `Proto`.

1. The connection monitor `PeerConnectionMonitor` is waiting for incoming connections on the port number registered for `Proto`. The monitoring is performed in a blocking call on a `Selector`.
2. When another peer initiats a new connection, the `PeerConnectionMonitor` detects this and returns the `SelectionKey` of the corresponding server socket.

3. The `ProtoPeerMonitor` is fetched from the internal registry by using the same `SelectionKey`.

4. The `SocketChannel` is retrieved and accepted using the `SelectionKey`.

5. The channel is added as an incoming connection of `ProtoPeerMonitor` with a call to `addConnection(SocketChannel, boolean)`.

6. The `addConnection(SocketChannel, boolean)` method, implemented in `PeerMonitor`, verifies that the same connection has not been added before.

7. If not, the `PeerBackEnd` is asked to grant the connection with a call to `allowConnection(SocketChannel)`.

8. If the `PeerBackEnd` allows the connection, a `ConnectionThreadJob` is created with a call to the `createConnectionJob(SocketChannel, boolean)` found in the `ProtoPeerMonitor` class. An appropriate class, perhaps named `ProtoConnectionThreadJob`, is instantiated and returned.

9. The job is scheduled for execution in the `ThreadPool`, by performing a call to the `addJob(ThreadJob)` in the `PeerBackEnd`.

10. When the `ProtoConnectionThreadJob` completes the `PeerBackEnd` will call the `callback(Object)` method of the `ProtoPeerMonitor`, passing the job as the parameter. This method is actually implemented in the base class, `PeerMonitor`.

11. The connection is validated with a call to `validateConnectionJob(Object)`, returning the corresponding socket channel if it was successfully validated. The following steps assume that it validated successfully. Otherwise they are simply discarded. It is the responsibility of the `ProtoConnectionThreadJob` to close the channel if the connection initialization fails.

12. The socket channel is added to the `PeerBackEnd`.

13. The socket channel is configured as non-blocking and registered with the `ProtoPeerReader`, which assigns a buffer to the channel.

14. The channel is registered with the `Selector` of the `ProtoPeerMonitor` and it is notified that a change has been performed.

15. The connection has been setup appropriately!

To initiate a new outgoing connection the same steps are taken, except the first four. Creating a `SocketChannel` that is connected to the peer in question is necessary. When this has been performed a call to the `addConnection(SocketChannel, boolean)` in the `PeerMonitor` class takes care of the rest. For outgoing connections the second parameter should be false.

9.4.6 Additional documentation

Additional documentation on the framework and its API (Application Programming Interface) can be found on the web at the following address:

`http://www.cs.umu.se/~bergner/thesis/api/`

9.5 Constructing an application

This section describes how the framework is extended to build a real peer-to-peer application. The necessary extensions needed are described in detail.

9.5.1 Extending the `PeerReader` class

The subclasses of the `PeerReader` class usually need a constructor without arguments, that simply calls the constructor of the base class. Furthermore, the subclass must provide an implementation of the following method, which has `protected` visibility:

isCompleted`(ByteBuffer buf)` → `int`

> Examines `buf` to see if it contains a complete protocol message. The check should always start from the beginning of the buffer since it is the place where the currently incoming message starts. If a complete message is contained in `buf` the method must return the number of bytes in this completed message. If no complete message has arrived the method should return 0.
>
> If a complete message has been found the calling `PeerReader` extracts the message from the buffer and moves any remaining bytes to the beginning of the buffer. This means that this method *always* should assume that data starts at the beginning of the buffer.
>
> The current position of `buf` must not be changed by this method, since the rest of the `PeerReader` class depends on it. If the position is changed inside this method it must be restored before returning from the method.

9.5.2 Extending the `PeerMonitor` class

The subclasses of `PeerMonitor` provide the additional details that makes the `PeerMonitor` class work. The constructor of the subclass needs at least a reference to the `PeerBackEnd`, which is required by the `PeerMonitor` constructor. The implementation of a monitor can use any additional arguments it wants. The methods that the subclass must, or is most likely to override, is the methods shown below. All of them have `protected` visibility.

createJob`(ByteBuffer buf, SocketChannel sc)` → `ThreadJob`

> Creates an appropriate job for managing the message stored in `buf` that arrived over the channel `sc`. This method probably involves the creation of additional objects, such as objects corresponding to a certain message type as well. Returns the created `ThreadJob`, or null if the message should be discarded.

createConnectionJob(SocketChannel sc, boolean incoming) → ConnectionThreadJob
: Creates a job that is responsible for initializing the connection for channel `sc`. Returns the connection job to use.

getConnectionJobID() → String
: The identifier used by the connection jobs for this protocol returned by `createConnectionJob(SocketChannel)`. This identifier is used by the monitor to register itself as the callback for connection jobs. When a connection job completes the `callback(Object)` method is called.

validateConnectionJob(Object job) → SocketChannel
: Checks if the connection job `job` was successfully initialized. If it was, the corresponding socket channel is returned. Otherwise null is returned. The default implementation simply checks if the job is an instance of `ConnectionThreadJob`. Usually some kind of status information from the job is needed in the implementation of this method.

Additional methods for protocol specific purposes are most likely needed as well. In an implementation of a file sharing protocol this class would be a good place to put methods for issuing queries, and possibly other methods for user initiated network access.

9.5.3 Constructing `ThreadJob` classes

To implement a scenario a job class is needed. All job classes, except connection job classes, inherit from the `ThreadJob` class. Connection jobs inherit from `ConnectionThreadJob`, which in turn is a subclass of `ThreadJob`.

The constructor of a job class needs to call the constructor in the `ThreadJob` class and specify the *priority* of the job and its *name*. The priority is used by the scheduling policy and the name is used for event dispatching when the job is completed.

Other than that any initialization is possible, which gives the programmer lots of freedom. The only other method the programmer must implement is the `run()` method. It contains the execution of the scenario and may use any components whatsoever. This also gives the programmer plenty of freedom.

9.5.4 Example scenario: Job execution and callback dispatch

Whenever a job is completed it can be dispatched to a callback. The thread that executes the job also executes the registered event handler. Here is how it works.

1. A callback is registered using the `addCallback(Callback, String` method of the `PeerBackEnd`.
2. A job is scheduled for execution using a call to `addJob(ThreadJob)` in the `PeerBackEnd`.
3. A `WorkerThread` in the `ThreadPool` is assigned to the job and executes.

4. When the job is completed the thread calls the `callback(Object)` method of the `PeerBackEnd`, since it has been registered as the callback for completed jobs in the `ThreadPool`.

5. Inside the `callback(Object)` method of the `PeerBackEnd` the registry is searched for a registered callback.

6. If a callback was found, call it and pass the just completed job as the parameter.

9.5.5 Constructing a facade, `PeerFrontEnd`

If a large scale peer-to-peer system should be constructed it is useful to add an additional layer of abstraction. This layer provides complete protocol transparency to the application, which is useful when several protocols are used simultaneously, and improves the event dispatching.

The main component of such an abstraction layer is the `PeerFrontEnd`. This class, which does not exist as a part of the framework, should be registered with the `PeerBackEnd` as the callback for all events that the application could be interested in. The `PeerFrontEnd` allows the application to register callbacks for application events. An application event in this context is not related to a specific protocol, but rather an event that could occur from one or more of the protocols.

In a file sharing system the `PeerFrontEnd` could provide the application with the ability to register for incoming query results that are destined for us. With one registration all query results are sent to the same callback, regardless of the protocol used to send them. The callback then has a more convenient parameter list, such as the file name, a text description, its size and a protocol specific *key*.

If a user of the file sharing system wants to download a file the corresponding protocol specific key is passed to the `PeerFrontEnd` requesting it to download the file. The key is used to determine which underlying component could be used to perform the actual download and it is contacted by the `PeerFrontEnd`.

In addition to the protocol transparency the `PeerFrontEnd` specifies which operations are possible to perform by the application. A file sharing system could contain operations for searching, browsing, downloading and finding more hosts that share a certain file. The application calls the corresponding methods in the `PeerFrontEnd` and they are responsible for performing the requested operation using the currently running set of protocols.

9.6 Implementing a file sharing system

This section presents an example on how a simple file sharing system could be implemented using the framework.

9.6.1 Choosing a protocol

First of all a decision concerning the protocol, or protocols, to use is necessary. If more than one protocol should be used it is important to think of protocol transparency aspects. Implementing a `PeerFrontEnd` could be the answer.

In this example we use the Gnutella 0.6 protocol, as specified in [Gnu02]. A simplified version of the protocol is described, but still compatible with modern clients. Things that are not directly related to message routing is discarded, such as actually downloading files and finding an initial set of hosts to connect to.

9.6.2 Implementing basic protocol support

To begin with connections must be established and incoming messages must be read. To do this three classes needs to be implemented. As before the name to the right of the class name denotes base classes or directly implemented interfaces.

GnutellaReader — `PeerReader`

This class extends the `PeerReader` class and provides an implementation of the `isCompleted(ByteBuffer)` method. The method first checks if a complete Gnutella header has arrived. If it has, the payload length is extracted from the header. If the buffer contains at least the header and the number of bytes specified by the payload length, the sum of those two numbers is returned. Otherwise 0 is returned.

GnutellaConnectionJob — `ConnectionThreadJob`

This class provides the basic functionality for initializing a Gnutella connection. If it is an incoming connection the following steps occur:

1. Verify that the first line is `GNUTELLA CONNECT/0.6`.
2. Read lines (containing headers) until a blank line is encountered.
3. Send `GNUTELLA/0.6 200 OK` and then an empty line. In reality the empty line can be preceded by a set of headers.
4. Verify that the next line is `GNUTELLA/0.6 200 OK`.
5. Read vendor specific headers. This means that lines are read until an empty line is encountered.
6. The connection has been established.

GnutellaMonitor — `PeerMonitor`

This class is the heart of the Gnutella protocol plug-in. It contains all protocol registries where message identifiers are stored, and later looked up. It is responsible for creating jobs for incoming messages and it provides the application with an easy-to-use method for performing queries.

This class could also be responsible for maintaining the shared archive, although this is probably performed by another component that this class only has a reference to. The class could also maintain some kind of timer to know when to issue `Ping` messages and when to perform some clean-up duties in the protocol registries.

9.6.3 Implementing protocol routing

As described in chapter 3 on page 11, message routing is specified to a high detail in Gnutella. The `GnutellaMonitor` is responsible for creating jobs

that handle incoming messages. Given a `ByteBuffer` that, according to the `GnutellaReader` class, should contain a complete message and the corresponding `SocketChannel` a job is created to handle the message. This makes it suitable to create small job classes for each message type, which gives us five job classes for messages:

GnutellaPingJob `ThreadJob`
: Handles incoming or outgoing `Ping` messages. If it is an incoming message a `Pong` is sent as a response.

GnutellaPongJob `ThreadJob`
: Determines if a `Pong` message is destined for us. If it is not, it is forwarded to the appropriate host.

GnutellaQueryJob `ThreadJob`
: Handles incoming or outgoing `Query` message. If it is an incoming message a lookup in the repository could result in the transmission of `QueryHit` messages. The query is forwarded to other neighbouring nodes.

GnutellaQueryHitJob `ThreadJob`
: Determines if a `QueryHit` message is destined for us. If it is not, it is forwarded to the appropriate host.

GnutellaPushJob `ThreadJob`
: Determines if a `Push` message is destined for us. If it is not, it is forwarded to the appropriate host.

Since each job is based on a single message type it is possible to take two approaches. First, we could implement small classes that encapsulate the contents in a message. This involves creating an abstract base class `GnutellaMessage` and then implementing subclasses such as `GnutellaPing` and `GnutellaQueryHit`. The `GnutellaMonitor` instantiates the appropriate class depending on the contents of the header, the message class parses the payload if necessary and the message object is passed to the constructor of the job. This approach causes the work performed by each job to be reduced, but it increases the amount of work performed in the `GnutellaMonitor`.

The second approach is the opposite of the first one. Instead of having the `GnutellaMonitor` create message objects that parse the contents of a `ByteBuffer`, the `ByteBuffer` could be passed directly to the job. The job is then responsible for parsing the message and it could eliminate the need for message classes entirely. The problem with this approach is that erroneous messages could cause the `PeerBackEnd` to dispatch events. An adjustment to the `PeerBackEnd` that checks if the job completed successfully would remove this problem. This also requires a minor adjustment to the `ThreadJob` class.

9.6.4 Implementing protocol user access

As mentioned earlier the `GnutellaMonitor` class provides methods that the application could use for user initiated network accesses, most notably queries. The application implements a callback for query hits and registers it with the

`PeerBackEnd`. Alternatively, the `GnutellaMonitor` could forward such a registration to the `PeerBackEnd`, which completely eliminates the need to interact with the `PeerBackEnd` directly for the application.

9.6.5 Managing events

All events are managed using the `Callback` interface. A Gnutella application could probably get away with only one callback, namely for query hits. All other messages could be handled automatically internally.

The callback is registered with the `PeerBackEnd` and checks if the query hit message was destined for us. If it was, it can present it to the user. Otherwise the message is simply ignored, since it has already been forwarded by the job that managed the message.

9.6.6 Putting the final touch to the application

Now most of the routing details should be working. Most problems probably occur with the implementation of the `GnutellaMonitor` and the tracking of messages and message identifiers. The finer details of the Gnutella protocol could be implemented, such as support for headers. At this point the most important part of the system would be to smoothly handle the changing environment, where hosts suddenly disconnect. This requires the system to cache information on other hosts and attempt to connect to other hosts when a connection fails. This also turns out to be the responsibility of the `GnutellaMonitor`. The `addConnection(SocketChannel, boolean)` method of the `PeerMonitor` should be used to initiate a new outgoing connection, since it already implements the necessary registrations needed for a connection.

Once the system remains stable despite an unstable environment it is time to put the icing on the cake and create an appealing user interface. But that is another story...

Chapter 10

Other applications

There is more to peer-to-peer than sharing files. This chapter discusses other applications of peer-to-peer and how they would benefit from the earlier work performed in this thesis.

10.1 Grid computing

Grid computing is a name that have appeared relatively recently. Grid computing is an extension of distributed computing with the purpose of using large sets of computers, not necessarily located in geographical proximity, to be used as a single virtual computer. Such a computer would have an immense computing power. The middleware providing Grid capabilities share many of the problems described in this thesis. The implementations of middleware also share many similarities with the implementations of common peer-to-peer services.

The utopia that Grid enthusiasts are striving for is something resembling a power grid where people can plug in their computer to harness the power of thousands of other computers. Before it becomes this simple, there are many problems that have to be solved.

Since Grid computing is, for the most part, about cooperating computers performing some kind of computation a Grid environment may seem vastly different from file sharing. To some extent this is true, but for one of the most fundamental aspects there are similarities. File sharing to a large extent is about locating one or more nodes that store a certain file. Grid computing is comparable in that sense that it also strives to find useful nodes, but this time for making use of their idle CPU cycles or some other useful resource. This means that similar protocols could be perhaps be used, although there are differences. The most noteworthy differences that comes to mind is:

- Grid computing does not have the legal issues that file sharing systems have. Such problems could develop if it, sometime in the future, appears that the Grid is used by people that monitor network traffic from governments, break encryptions of classified material, or something else that a majority of politicians would see as a serious problem. This means that the Grid could use more centralized techniques to locate nodes.

- A Grid is not an application. It is a vision. The concept of a Grid is the vision of successfully combining distributed resources, such as CPU power and disk space, in a way that makes it easily available. It can be argued that a Grid is a higher level concept then peer-to-peer services, but the underlying intelligence of a Grid have many things in common with peer-to-peer services.

Grid computing is a very broad topic. The Grid can be used to share disk space, certain special applications and even access to special hardware. A Grid can range from a small set of homogeneous computers to a world wide set of heterogeneous systems with different capabilities and operating systems. Some of the most interesting problems with Grid computing include the following.

- Given an application with a certain set of constraints regarding its execution environment, how can we find a suitable set of nodes that fulfill these, sometimes complex, requirements?
- Given an application, how can it be distributed in the Grid to achieving good, usually meaning fast and accurate, results?
- Given an application that was not originally intended for a distributed environment, could the Grid be used anyway to improve performance?
- Given an application not originally intended for a distributed environment, can the application be "modified" in some, hopefully automated, way to be able to use the benefits of the Grid?

This list of question actually goes on and on. A Grid presents an interesting idea on a virtual computer composed of many others, but the resemblance to peer-to-peer lies in the underlying middleware. For Grid middleware, any information and knowledge concerning distributed environments is useful, since it is such a complex platform. This also includes material on more traditional peer-to-peer services, which is the topic of this thesis. For a more detailed introduction to Grid Computing see for example [Ber02].

10.2 Instant messaging

Instant messaging is a service where users exchange messages, and nowadays various files as well. The number of users of ICQ, AIM and, more recently, Jabber powered applications have reached a very large audience. Although instant messaging has on various occasions been related to other peer-to-peer services, it is more centralized than many peer-to-peer services.

Jabber uses a client-server-server-client architecture, similar to e-mail. When a message is sent from the client it arrives at the local server. The local server examines the message destination and contacts the server at that location, and sends the message. The server at the remote destination can then deliver the message to the designated client.

Other instant messaging systems, such as ICQ, use a more peer-to-peer related model. Messages are exchanged using direct connections between peers, but there is still a server that maintains status information for the users. The most visible part, that is managed by the server, for the users

of these systems is the presence information. When a user starts his client it contacts the server and authenticates itself, using its user id. It tells the server which users are considered interesting, or more commonly referred to as friends or buddys. The server is able to provide information on the status of these users, which is shown directly in the user interface of the client.

If ICQ would replace its set of servers, for which I have a hard time to think of a single good reason, and have the clients share the responsibilities of the server things get a lot more interesting. In such a case instant messaging would without question be a full fledged peer-to-peer system. In that case most of the work in this thesis are relevant, but for now instant messaging actually resembles client/server systems as much as they resemble peer-to-peer systems. To make a long story short, ICQ is a centralized peer-to-peer system where most of the problems are solved on a client/server basis, which also was the case for Napster.

10.3 Freenet

Freenet is a system where users share diskspace. This diskspace is used to host files published by anonymous authors. Files are found by using key based searches and both searching and downloading content is anonymous. Anonymity comes at a prize though since the actual files are routed back through all the hosts that relayed the query. This could potentially cause a file to be routed three laps around the world until it finally arrives at your neighbour. For small files this is not an issue worth arguing about, but for larger files it takes a lot more time and help congest network in an abundance of places. Anonymity is a very important aspect of Freenet and should certainly not be neglected, but since this thesis has contained quite a lot of performance related material it had to be mentioned.

Freenet can be used for a variety of applications. The official website, [FNP03] mentions the following:

- Publishing websites or "freesites"
- Communicating via message boards
- Playing simple turn-based games like Chess
- Content distribution

Freenet orders nodes in a way that makes searching quite efficient, which means that queries use the network efficiently. The structure is somewhat comparable to P-Grid, since both of them promote an ordered architecture. The main difference is that Freenet evolves more freely depending on the queries issued in the network. Over time nodes tend to specialize in handling keys that lie close to each other, which does *not* mean that the content of the corresponding files are related. Queries are not routed as efficiently as in P-Grid, but still achieves good query routing performance on most occasions.

Since Freenet is a similar service, although it shares disk space instead of files, most of the thesis is relevant. Freenet uses a text based protocol, although all files are encrypted. It should be possible to implement the Freenet protocol using the framework proposed in this thesis. Another similar service

to Freenet is OceanStore and its corrsponding routing architecture Tapestry. See [RWE+01] for a description of OceanStore and Tapestry.

10.4 JXTA

Project JXTA is a project started by Sun Microsystems. This is not by itself anything even resembling a service or application, but rather a set of open peer-to-peer protocols that allow nodes to communicate and collaborate. Many experts in various areas of computing science have been, or are, involved in the project. Details concerning the project can be found in [JXT03].

The *core* of JXTA specifies the basic elements of the family of protocols that is JXTA. This includes identifiers, advertisements and the most important protocols. The core is necessary for every JXTA node to support.

In addition a set of *services* are defined by JXTA. These services define additional protocols that together make up the parts of a powerful and generic peer-to-peer framework, on which applications can be built. The complete specification of the JXTA protocols and services can be found in [JXP03]. A discussion on searching in the JXTA network is presented in [WDKF02].

The set of protocols is too advanced for most needs, which makes it too time consuming to actually implement for most peer-to-peer projects. Fortunately for most developers, there are some ambitious people in the JXTA community. There are programming language implementations of the JXTA core and the most common services. The Java implementation is the most stable one, and as it appears the C implementation also appears reasonably stable. There are development going on with languages such as Perl, Python, Ruby and Smalltalk.

The closest thing to project JXTA in this thesis is without question the peer-to-peer framework. Besides that JXTA in reality is a protocol specification it is possible to compare the Java implementation of JXTA with the framework presented in this thesis.

- To relate to another part of computing science, namely programming languages, the comparison between the framework in this thesis and project JXTA would be like comparing a simplified version of Pascal with the ADA programming language. There is no question which one is the most powerful, flexible and the one to choose if an advanced system should be constructed. On the other hand, being small and easy to learn and use are very appealing arguments under many circumstances. To summarize, JXTA has everything that the framework in this thesis has, and a whole lot more. The main difference is the complexity.

- The framework in this thesis implements application protocols on top of TCP/IP. Project JXTA uses any arbitrary underlying transport, which in the most extreme case could mean that pigeons are used to carry messages. On this arbitrary protocol resides the three platform core protocols, and on top of them, the four standard service protocols.

- Project JXTA uses XML to encode the messages, while the proposed framework makes no assumptions regarding the encoding of messages. There is also an alternative binary representation specified.

Chapter 11

Summary and conclusions

This chapter summarizes the entire thesis. It also presents the most important results and conclusions. It also contains general reflections of the work with this thesis. The chapter ends with a discussion on what could be done in the future to follow up this work.

11.1 Introduction

To obtain a quick understanding on what this master thesis contains and what the most important results of the work are, this chapter should provide the necessary information. By reading this chapter you should have a reasonable understanding on what the rest of the thesis will discuss. This chapter will also attempt to answer a few important questions regarding this master thesis and the work of completing it. Here are some examples.

- How useful are the results presented in this thesis? From a scientific point of view? From a more directly practical point of view?
- Considering the efforts put in, are the results satisfactory, or is something missing?
- From an educational point of view, has the work with this thesis led to an increasing understanding of the area, and perhaps other related areas?
- How much of the original goals of the thesis have been accomplished?
- How much additional contents have found their way into the thesis along the way?

11.2 Contents of the thesis

The thesis presents a thorough study of peer-to-peer file sharing systems. The protocols are presented and discussed. The protocols are then analyzed and their performance and network utilization is measured, or estimated. Using the knowledge obtained during the study of the file sharing systems and the protocol analysis two separate issues are covered.

- First, a peer-to-peer protocol that could achieve high efficiency, compared to some of the protocols analyzed, is designed. A detailed proposal is presented, although a rigorous analysis and implementation of the protocol is not performed.
- Second, a framework for building peer-to-peer applications is designed and implemented in Java. The chapters devoted to this framework describes the design and implementation in high detail, and also describes how the framework can be extended to build an application.

At the end of the thesis some other application areas of peer-to-peer networking are discussed, and the possible benefits that these applications could have from the work presented in the thesis.

11.3 Results obtained

There are several important results presented throughout this thesis. Those that especially deserve to be mentioned are the following.

- Todays network infrastructure contains many challenges that peer-to-peer systems must circumvent. Important solutions include push technology and proxy based techniques.
- The Gnutella file sharing protocol still uses a lot of bandwidth, even though caching mechanisms and ultrapeers have been introduced.
- The P-Grid distributed search tree requires very finely tuned parameters and simulations have had a hard time achieving results similar to the developers of P-Grid. The results received indicate that the developers of P-Grid have been correct in their simulations.
- The P-Grid developers has not given much attention to convergence speeds when a previously initialized network changes. Simulations have shown that this is a potential problem with P-Grid, although no thorough theoretical basis for this argument exists.
- The OpenFT network protocol is at the moment comparable to Gnutella in terms of network performance. This protocol offers better control of the network traffic than Gnutella, which makes it possible for the protocol to improve its network utilization once the protocol matures.
- To use a network efficiently nodes must be structured appropriately. The thesis proposes that binary keys are used to structure peers. These keys are used for locating content. Such an approach could make it necessary to use a separate metadata service to handle metadata. For better utilization of network resources such a service should be constructed using more traditional methods, such as client/server, mirroring and caching.
- Most peer-to-peer applications solve the same problems, regarding connection management and handling asynchronous event dispatching. The thesis identifies the most important common aspects and outlines a design for the backend of a peer-to-peer application.

- The proposed design is implemented in Java and well documented. Support for multiple protocols plug-ins (which are fairly easy to implement) and a way to manage connections and events that makes it easy for application programmers in most cases. The framework has not been tested extensively, but a simplified Gnutella plug-in has been developed successfully, from which message routing and queries can be performed.

11.4 General remarks from the author

The study of protocols and their performance is an interesting topic, but sometimes hard. To study a protocol there must be some useful documentation, which is *not* the case with FastTrack. The study of OpenFT was also very difficult, since the protocol is not so mature and not well documented. Many hours have been spent reading source code for the giFT project, and OpenFT in particular.

The statistical measurements of OpenFT were not as detailed and thorough as was intended, but the parsing of statistical data was too time consuming to fully complete. The analysis and simulations of P-Grid have caused a lot of problems, since the descriptions of the algorithms used are different in most papers. The low level of detail on the simulations performed by the authors of P-Grid have also made it difficult to achieve similar results. If the implementation is completely unknown it is hard to estimate if the difference is due to implementation differences, or actually problems with the algorithm.

The focus of the thesis has shifted somewhat throughout the work. Initially the thesis was a more analysis oriented and aimed at more peer-to-peer services. After a while the focus shifted more towards file sharing, since otherwise it would have been hard to achieve any real depth. After months of studying file sharing protocols and performing some protocol analysis the idea of a framework was born. The studies of existing projects provided a good foundation on the construction of peer-to-peer systems. The initial design of the framework was not too far away from the final design presented in this thesis.

Overall, the results obtained are certainly satisfactory and I know for a fact that the effort put in is more than necessary. When the work started I had never heard of P-Grid or the giFT project, but they became important pieces along the way. The same goes for the framework which, to some extent, found its way into the thesis due to my wishes to write some code. In the end the work has been very rewarding in terms of obtained knowledge. The knowledge I regard as the most rewarding is the understanding of how peer-to-peer systems can be built, but knowledge in protocol design is also a very worthy candidate for the top spot.

11.4.1 The proposed protocol

The protocol proposed could be perceived as it lacks scientific foundation. Partially this is true since the protocol was designed from scratch. The design is based on extensive studies of peer-to-peer protocols though and most ideas in the protocol proposal are influenced by other protocols. The lack of references to other protocols in the protocol proposal is due to the fact that

the number of sources is almost infinite, since it involves protocol ideas from decades of computer networking.

Concerning the quality of the protocol there is no definite answer at this point, since this thesis is as far as the ideas reached. A test implementation would certainly lead to improvements in the design. Personally, I believe that the protocol, perhaps with some minor modifications, has a lot of potential. I have tried to add notes about where extensions could be made, and for what purpose.

Compared to protocols and ideas presented at various conferences the potential of the proposed protocol exceeds several of them. From a scientific point of view, a more mature version of the protocol could probably be accepted to be presented at some conference on peer-to-peer networking or distributed computing. More solid results and a test implementation are of course needed first, but compared to other protocols that appears at various conferences it is not necessarily a lost cause.

11.4.2 The peer-to-peer framework

The peer-to-peer framework was designed after extensive study of other peer-to-peer projects, their design and implementation. Since the thesis has only brushed the surface of services other than file sharing it could require some adjustments for other application areas.

The nicest thing about the framework is, in my own opinion, its usability. It is possible to use the framework and write a protocol plug-in, and hopefully later a complete application with almost no knowledge about peer-to-peer networking and its special requirements. The framework allows the developer to work in a more traditional "network programming" way and many of the details are effectively hidden. The introduction of a thread pool and to make much of the intelligence located in small job classes turned out to be a very flexible choice, which allowed the framework to manage many of the trickiest parts and letting the developer fill in the protocol specific blanks.

The framework only consists of 2500 lines of commented Java code, which is quite small. The simplified Gnutella plug-in consists of about 1600 lines of code. If we assume that it would require ten(!) times as much to implement the remaining parts of the protocol we would end up with $2500 + 10 \times 1600 = 18500$ lines of code. When comparing this to the 26000 lines of code in the `com.limegroup.gnutella` package of the **Limewire** client we could use many thousand lines more, without having to be ashamed. This becomes even more evident if we count the subpackages, which include 50000 lines of code, for a total of 76000 lines. And this is only files from the `core` project of **Limewire**. The `gui` project contains almost 50000 lines of code in its `com.limegroup.gnutella.gui` package and its subpackages. To me this seems that the framework is well on the way to something really useful.

Compared to JXTA is does not really stand a chance for really large scale projects, but for smaller projects and educational purposes the framework is more suitable, since it is easier to understand. The JXTA core has of course gone through much heavier testing, which should make it more reliable, but if you do not mind fixing a bug or two the framework is an excellent choice!

11.5 Future work

The proposed protocol design needs to be reviewed more thoroughly. A test implementation is necessary to reveal the flaws in the design. A large amount of testing to verify the performance of the protocol is also necessary. Any detected flaws needs to be removed, in order to obtain a protocol that is at the front of todays file sharing protocols in terms of performance. Further studies of how other services could use the proposed protocol is also needed, since this thesis has only looked at this at a glance.

There is always a need for further depth in terms of security aspects. Since this is a thesis about performance security has been, more or less, left out. If the proposed protocol is to be implemented and used in practice a more thorough study from a security perspective is needed.

Needs of applications other than file sharing must be identified, and if necessary incorporated in the framework. The purpose of the framework is to simplify development of almost any peer-to-peer service and if important areas of peer-to-peer computing work in a way that makes the framework difficult to use, the framework probably needs to be redesigned to some extent.

It would be nice if the framework could control the connectivity more automatically than at the moment. Now each protocol plug-in is responsible for maintaining connectivity. Perhaps a separate connectivity module could be a part of every protocol plug-in and used automatically by the framework.

Appendix A

List of abbreviations

This appendix explains all abbreviations used throughout this thesis. Most abbreviations are explained only with their meaning, but some are described in a more informal manner.

ADT	Abstract Data Type
ADTP	Amorphous Distributed Tree Protocol
AIM	America Online Instant Messenger
API	Application Programming Interface
ARP	Address Resolution Protocol
ASCII	American Standard Code for Information Interchange
AVL	Adel'son-Vel'skii-Landis (AVL-tree)
CCSO	Computing and Communications Services Organization
CPU	Central Processing Unit
CVS	Concurrent Versioning System
DHCP	Dynamic Host Configuration Protocol
DHT	Dynamic Hash Tree
DNS	Domain Name System
DTD	Document Type Definition
EQHD	Extended QueryHit Descriptor
FIFO	First-In-First-Out
FTP	File Transfer Protocol
GGEP	Gnutella Generic Extension Protocol
giFT	giFT Internet File Transfer or giFT isn't FastTrack
GMT	Greenwich Mean Time
GNU	GNU's Not Unix
GTK	GIMP Toolkit
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol over SSL
HUGE	Hash/URN Gnutella Extension
ICMP	Internet Control Message Protocol
ICQ	I Seek You
ID3	Metadata tag in MP3 files
IETF	Internet Engineering Task Force

IGMP	Internet Group Management Protocol
INP	Internet Nomenclator Project
IP	Internet Protocol
IPv4	Internet Protovol version 4
IPv6	Internet Protocol version 6
ISBN	International Standard Book Number
ISO-8859-1	A common character encoding
ISP	Internet Service Provider
JDK	Java Development Kit
JXTA	The JXTA Project, short for juxtaposition
MD5	Message Digest 5, a hash algorithm
MIME	Multipurpose Internet Mail Extension
MP3	Mpeg Layer 3, a common format for music files
NAT	Network Address Translation
NIO	New I/O, a Java package
NNTP	Network News Transfer Protocol
OS	Operating System
PDF	Portable Document Format
PDU	Protocol Data Unit
QoS	Quality of Service
RIAA	Recording Intustry Association of America
SHA	Secure Hash Algorithm
SMTP	Simple Mail Transfer Protocol
SNQP	Simple Nomenclator Query Protocol
SQL	Structured Query Language
SSL	Secure Socket Layer
TCP	Transmission Control Protocol
TTL	Time To Live
UDP	User Datagram Protocol
UK	United Kingdom
URL	Uniform Resource Locator
URN	Uniform Resource Name
US	United States
UTF-8	A common character encoding
UUCP	Unix-to-Unix Copy Protocol
XML	Extensible Markup Language

Appendix B

DTDs for ADTP

This appendix contains the DTDs for the messages in the Amorphous Distributed Tree Protocol.

B.1 `peerinfo` request

```
<!ELEMENT peerinforequest (key, numbits, nummirrors, numchildren,
                           numparents, uptime, bandwidth,
                           bandwidthused)>
<!ELEMENT key             (#PCDATA)>
<!ELEMENT numbits         (#PCDATA)>
<!ELEMENT nummirrors      (#PCDATA)>
<!ELEMENT numchildren     (#PCDATA)>
<!ELEMENT numparents      (#PCDATA)>
<!ELEMENT uptime          (#PCDATA)>
<!ELEMENT bandwidth       (#PCDATA)>
<!ELEMENT bandwidthused   (#PCDATA)>
```

B.2 `peerinfo` response

```
<!ELEMENT peerinforesponse (key, numbits, nummirrors, numchildren,
                            numparents, uptime, bandwidth,
                            bandwidthused)>
<!ELEMENT key              (#PCDATA)>
<!ELEMENT numbits          (#PCDATA)>
<!ELEMENT nummirrors       (#PCDATA)>
<!ELEMENT numchildren      (#PCDATA)>
<!ELEMENT numparents       (#PCDATA)>
<!ELEMENT uptime           (#PCDATA)>
<!ELEMENT bandwidth        (#PCDATA)>
<!ELEMENT bandwidthused    (#PCDATA)>
```

B.3 `list` request

```
<!ELEMENT listrequest (lists, options, maxanswers)>
<!ELEMENT lists       (#PCDATA)>
<!ELEMENT options     (#PCDATA)>
<!ELEMENT maxanswers  (#PCDATA)>
```

B.4 `list` response

```
<!ELEMENT listresponse (list*)>
<!ELEMENT list         (id, options?, item+)>
<!ELEMENT id           (#PCDATA)>
<!ELEMENT options      (#PCDATA)>
<!ELEMENT item         (address, port, numbits, key)>
<!ELEMENT address      (#PCDATA)>
<!ELEMENT port         (#PCDATA)>
<!ELEMENT numbits      (#PCDATA)>
<!ELEMENT key          (#PCDATA)>
```

B.5 `join` request

```
<!ELEMENT joinrequest (address, port, numbits, key)>
<!ELEMENT address     (#PCDATA)>
<!ELEMENT port        (#PCDATA)>
<!ELEMENT numbits     (#PCDATA)>
<!ELEMENT key         (#PCDATA)>
```

B.6 `join` response

```
<!ELEMENT joinresponse (address, port, numbits, key)>
<!ELEMENT address      (#PCDATA)>
<!ELEMENT port         (#PCDATA)>
<!ELEMENT numbits      (#PCDATA)>
<!ELEMENT key          (#PCDATA)>
```

B.7 `changekey` request

```
<!ELEMENT changekeyrequest (numbits, key)>
<!ELEMENT numbits          (#PCDATA)>
<!ELEMENT key              (#PCDATA)>
```

B.8 `register` request

```
<!ELEMENT registerrequest (address, port, share+)>
<!ELEMENT address         (#PCDATA)>
<!ELEMENT port            (#PCDATA)>
<!ELEMENT share           (numbits, key)>
<!ELEMENT numbits         (#PCDATA)>
<!ELEMENT key             (#PCDATA)>
```

B.9 `unregister` request

```
<!ELEMENT unregisterrequest (address, port, share+)>
<!ELEMENT address           (#PCDATA)>
<!ELEMENT port              (#PCDATA)>
<!ELEMENT share             (numbits, key)>
<!ELEMENT numbits           (#PCDATA)>
<!ELEMENT key               (#PCDATA)>
```

B.10 `leave` request

```
<!ELEMENT leaverequest (numbits, key)>
<!ELEMENT numbits      (#PCDATA)>
<!ELEMENT key          (#PCDATA)>
```

Bibliography

[Abe01] Karl Aberer. *P-Grid: A Self-Organizing Access Structure for P2P Information Systems*. In *Sixth International Conference on Cooperative Information Systems*, September 2001.

[Abe02] Karl Aberer. *Scalable Data Access in P2P Systems Using Unbalanced Search Trees*. In *Workshop on Distributed Data and Structures*, June 2002.

[ACMD+02] Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, Roman Schmidt, and Jie Wu. *Advanced Peer-to-Peer Networking: The P-Grid System and its Applications*. Technical Report 73, Distributed Information Systems Laboratory, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2002.

[AHP03] Karl Aberer, Manfred Hauswirth, and Magdalena Punceva. *Self-organized construction of distributed access structures: A comparative evaluation of P-Grid and FreeNet*. In *Workshop on Distributed Data and Structures*, June 2003.

[AL01] Paul Albitz and Cricket Liu. *DNS and Bind*. O'Reilly & Associates, Sebastopol, California, fourth edition, 2001.

[APHS02] Karl Aberer, Magdalena Punceva, Manfred Hauswirth, and Roman Schmidt. *Improving Data Access in P2P Systems*. *IEEE Internet Computing*, 6(1):58–67, January 2002.

[BEA03] *BearShare*. Official web site, 2003, *http://www.bearshare.com/*. Windows Gnutella client.

[Ber02] Viktors Berstis. *Fundamentals of Grid Computing*. Technical report, IBM Corporation, 2002.

[BWDD02] Peter Backx, Tim Wauters, Bart Dhoedt, and Piet Demeester. *A comparison of peer-to-peer architectures*. In *EURESCOM Summit*, Heidelberg, Germany, October 2002.

[CMH+02] Ian Clarke, Scott G. Miller, Theodore W. Hong, Oskar Sandberg, and Brandon Wiley. *Protecting Free Expression Online with Freenet*. *IEEE Internet Computing*, 6(1):40–49, January 2002.

[DHC93] Internet Engineering Task Force. *Dynamic Host Configuration Protocol*, October 1993. RFC 1541.

[DNS87] Internet Engineering Task Force. *Domain Names*, November 1987. RFC 1034 (Concepts and Facilities), 1035 (Implementation and Specification).

[FNP03] *Freenet: The Free Network Project*. Official web site, 2003, *http://freenetproject.org*. Information sharing system supporting anonymous publishing.

[FTP85] Internet Engineering Task Force. *File Transfer Protocol*, October 1985. RFC 959.

[GG03] GTK *Gnutella*. Official web site, 2003, *http://gtk-gnutella.sourceforge.net/*. Open source Gnutella client.

[GHI+01] Steven Gribble, Alon Halevy, Zachary Ives, Maya Rodrig, and Dan Suciu. *What Can Databases Do for Peer-to-Peer?* In *Proceedings of the Fourth International Workshop on the Web and Databases*, 2001.

[GIF03] *giFT: Internet File Transfer project*. Official web site, 2003, *http://gift.sourceforge.net*. FastTrack based open source project.

[Gnu00] Clip2 Distributed Search Services. *The Gnutella Protocol Specification v0.4*, 2000.

[Gnu02] *Gnutella 0.6 Protocol Draft*, June 2002, *http://rfc-gnutella.sourceforge.net/draft.txt*.

[GRO03] *Grokster*. Official web site, 2003, *http://www.grokster.com/*. A FastTrack client.

[HHH+02] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon Thau Loo, Scott Shenker, and Ion Stoica. *Complex Queries in DHT-based Peer-to-Peer Networks*. In *International Workshop on Peer-to-Peer Systems*, March 2002.

[HTT96] Internet Engineering Task Force. *Hypertext Transfer Protocol — HTTP/1.0*, May 1996. RFC 1945.

[HTT99] Internet Engineering Task Force. *Hypertext Transfer Protocol — HTTP/1.1*, June 1999. RFC 2616.

[IM03] *iMesh*. Official web site, 2003, *http://www.imesh.com/*. Commonly used FastTrack client.

[INP97] *Internet Nomenclator Project*. Official web site, 1997, *http://cm.bell-labs.com/cm/cs/what/nomenclator/*. Meta-data query system.

[INP98] Internet Engineering Taskforce. *Internet Nomenclator Project*, January 1998. RFC 2258.

[IP81] Internet Engineering Task Force. *Internet Protocol*, September 1981. RFC 791.

[JAB01] Mihajlo A. Jovanović, Fred S. Annexstein, and Kenneth A. Berman. *Modeling Peer-to-Peer Network Topologies through "Small-World" Models and Power Laws*. In *Telecommunications Forum*, Belgrade, Yugoslavia, November 2001.

[JBBS01] Matthew B. Jones, Chad Berkeley, Jivka Bojilova, and Mark Schildhauer. *Managing Scientific Metadata*. *IEEE Internet Computing*, 5(5):59–68, September 2001.

[JXP03] *JXTA v2.0 Protocols Specification*. Technical report, Sun Microsystems Inc., 2003.

[JXT03] *Project* jxta. Official web site, 2003, *http://www.jxta.org/*. Sun's peer-to-peer framework.

[KAZ03] *KaZaA*. Official web site, 2003, *http://www.kazaa.com/*. Standard FastTrack client.

[KZL03] *KaZaA Lite*. Official web site, 2003, *http://www.k-lite.tk/*. Light-weight FastTrack client.

[LIM03a] *Limewire*. Official web site, 2003, *http://www.limewire.com/*. Java based Gnutella client.

[LIM03b] *Limewire*. Official project web site, 2003, *http://www.limewire.org/*. Java open source Gnutella client.

[MKL+02] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. *Peer-to-Peer Computing*. Technical Report 57, Hewlett-Packard Laboratories, Palo Alto, California, March 2002.

[NAT00] Internet Engineering Task Force. *Network Address Translation*, February 2000. RFC 2766.

[NNT86] Internet Engineering Task Force. *Network News Transfer Protocol*, February 1986. RFC 977.

[Ora01] Andy Oram, editor. *Peer-to-Peer — Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates, Sebastopol, California, first edition, 2001.

[PGR03] *P-Grid*. Official web site, 2003, *http://www.p-grid.org/*. The P-Grid consortium.

[Pin01] Michael T. Pinkey. *An Efficient Scheme for Query Processing on Peer-to-Peer Networks*. Technical report, Aeolus Research, Inc., 2001.

[RF01] Christopher Rohrs and Vincent Falco. *Ping/Pong scheme*. WWW Page, 2001, *http://www.limewire.com/-index.jsp/pingpong*. Lime Peer Technologies.

[RFG03] *RFC Gnutella*. Official web site, 2003, *http://rfc-gnutella.sourceforge.net/*. Depository of Gnutella specifications.

[RIF02] Matei Ripeanu, Adriana Iamnitchi, and Ian Foster. *Mapping the Gnutella Network*. *IEEE Internet Computing*, 6(1):50–57, January 2002.

[Rip01] Matei Ripeanu. *Peer-to-Peer Architecture Case Study: Gnutella Network*. Technical Report 26, University of Chicago, July 2001.

[RWE+01] Sean Rhea, Chris Wells, Patrick Eaton, Dennis Geels, Ben Zhao, Hakim Weatherspoon, and John Kubiatowicz. *Maintenance-Free Global Data Storage*. *IEEE Internet Computing*, 5(5):40–49, September 2001.

[SMT01] Internet Engineering Task Force. *Simple Mail Transfer Protocol*, April 2001. RFC 2821.

[SNQ98] Internet Engineering Taskforce. *Simple Nomencaltor Query Protocol*, January 1998. RFC 2259.

[Tel83] Internet Engineering Task Force. *Telnet Protocol Specification*, May 1983. RFC 854.

[TGO92] Grace Todino-Gonguet and Tim O'Reilly. *Managing UUCP and Usenet*. O'Reilly & Associates, Sebastopol, California, tenth edition, 1992.

[TXM02] Chunqiang Tang, Zhichen Xu, and Mallik Mahalingam. *PeerSearch: Efficient Information Retrieval in Peer-to-Peer Networks*. Technical Report 198, Hewlett-Packard Laboratories, Palo Alto, California, July 2002.

[WDKF02] Steve Waterhouse, David M. Doolin, Gene Kan, and Yaroslav Faybishenko. *Distributed Search in P2P Networks*. *IEEE Internet Computing*, 6(1):68–72, January 2002.

[Wei00] Mark Allen Weiss. *Data Structures and Problem Solving Using C++*. Addison Wesley, Reading, Massachusetts, second edition, 2000.

Index

This index contains all terms used and defined throughout this thesis. Some conventions are used to make it easier to find what you want. If a term occurs on several occasions, the most important one is indicated by showing the page number in **boldface**. The words themselves have various appearances classified as follows:

Roman regular
: Denotes important terms or keywords in the text.

Sans-serif regular
: Denotes names of programs and utilities.

Small capitals
: Denotes the primary definition of abbreviations.

`Constant width`
: Denotes protocol or implementation specific elements.

R

S

T

U

V

W

X