In compliance with the
Canadian Privacy Legislation
some supporting forms
may have been removed from
this dissertation.

While these forms may be included
in the document page count,
their removal does not represent
any loss of content from the dissertation.

U-P2P: A Peer-to-peer System for Description and Discovery of Resource-sharing Communities

by Alope Tukul Mukherjee, B. Eng.

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

Faculty of Engineering

Department of Systems and Computer Engineering

Carleton University,

Ottawa, Ontario, Canada, K1S 5B6

September 2, 2003

© 2003, Alope Tukul Mukherjee

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitons et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-88978-5

Our file Notre référence

ISBN: 0-612-88978-5

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Canada

Abstract

Peer-to-peer file-sharing systems offer a simple way to store and access large amounts of data in a distributed and fault-tolerant manner. These systems enable users to form communities for sharing different types of files. One important feature is the ability to search, but many existing systems provide only limited ability to search using metadata. U-P2P is a novel application framework that exploits metadata to improve search in file-sharing communities. Additionally, U-P2P simplifies the creation and discovery of file-sharing communities. Each community allows the sharing of a particular type of structured document. Users are free to define their own structured formats. These facilitate metadata search by defining the available search criteria and validating published documents. Communities are themselves modeled as structured documents, thus enabling their sharing and discovery exactly like any other document. Users can easily create new formats and communities.

Acknowledgements

My two pieces of advice to anyone embarking on a Masters Thesis would be to read widely before hacking and to write regularly, summarizing one's readings and latest work. The former will enable you to clarify where your ideas stand in respect to the state of the art and will inform your hacking. The latter will greatly simplify the task of writing the final thesis. For these insights, and innumerable others, I must thank my advisor, Professor Babak Esfandiari. I also owe him an apology, since it took me far too long to appreciate most of them. His contribution to this thesis cannot be underestimated: U-P2P is a product of Professor Esfandiari's fertile mind and its transition from idea to reality could not have happened without his guidance throughout our many intriguing and inspiring conversations (and arguments!).

Neal Arthorne also made a substantial contribution to this thesis with his implementation and documentation of U-P2P 1.0. Neal is that most rare of commodities, the hacker who can write. It is no accident that he is the most cited author in the bibliography: his contributions can be found throughout this thesis.

Finally, I must thank my family for supporting me throughout and only occasionally worrying that I might never finish at all. Despite their hounding and cajoling, their action at a distance would have been for naught without the unflagging support of my wife, Tanya Roopnarine. I owe her a debt that I have no hope of ever repaying.

TABLE OF CONTENTS

ABSTRACT	III
ACKNOWLEDGEMENTS	IV
LIST OF FIGURES	XI
LIST OF TABLES.....	XV
CHAPTER 1 INTRODUCTION	1
1.1 What problems do file-sharing systems solve?.....	1
1.2 Weaknesses of File-sharing Systems	2
1.2.1 Inadequate Search	3
1.2.2 Limited to Popular Formats	3
1.2.3 Fragmented Communities	4
1.3 Goals.....	5
1.4 Approach	6
1.4.1 Improving Search.....	6
1.4.2 Simplifying Community Creation and Discovery	8
1.5 Contributions.....	10
1.5.1 Improving search	10

1.5.2	Simplifying Community Creation and Discovery	11
1.6	Thesis Overview	13
CHAPTER 2 STATE OF THE ART		14
2.1	Introduction.....	14
2.2	Search.....	15
2.2.1	Metadata.....	16
2.2.2	Query Routing.....	20
2.2.3	Communities	25
2.2.4	Search for Location-centric Systems	28
2.3	Creating and Discovering Communities.....	36
2.3.1	Community opaque.....	37
2.3.2	Community limited	38
2.3.3	Community implicit.....	39
2.3.4	Community unshared.....	41
2.3.5	Community partial	41
2.4	Other Parameters of File-sharing Systems.....	43
2.4.1	Other Classifications for Peer-to-peer systems.....	44
2.4.2	Network architecture.....	46
2.4.3	Structured Systems.....	49
2.4.4	Anonymity and Censorship.....	51

2.5	Conclusion	53
2.5.1	Weaknesses of Existing Search Methods	54
2.5.2	The Difficulty of Creating and Discovering Communities.....	57
2.5.3	Why Search and Communities Matter.....	60
CHAPTER 3	THE CONCEPT OF FILE-SHARING COMMUNITIES	63
3.1	What is a Community: Community Schema Design	63
3.2	Attributes of file-sharing communities	65
3.2.1	Name	65
3.2.2	Format	65
3.2.3	Protocol	66
3.2.4	Display	66
3.2.5	Other Parameters.....	67
3.3	Community as Class/Community as Object.....	68
3.4	Conclusion	72
CHAPTER 4	DESIGN OF U-P2P.....	74
4.1	Introduction.....	74
4.2	Overview	75
4.3	Schemas and the Choice of XML Schema	76
4.4	Improved Metadata	77

4.5	Web Adapter Design: Improved Search.....	80
4.6	Simplified Community Creation and Discovery	86
4.7	Repository Design	90
4.7.1	Interactions.....	93
4.7.2	XML Database	95
4.8	Network Adapter Design.....	97
4.8.1	Interactions.....	98
4.8.2	Protocol	101
4.9	Conclusion	104
CHAPTER 5 DETAILED DESIGN		106
5.1	Interface and Web Adapter	106
5.1.1	Generating pages with XSL.....	109
5.1.2	Major Software Components	113
5.2	Repository Detailed Design	117
5.2.1	XML Database	120
5.3	Network Adapter Detailed Design.....	123
5.3.1	Message Types.....	125
5.4	Conclusions.....	130
CHAPTER 6 EVALUATION AND VALIDATION OF U-P2P		131

6.1	Introduction.....	131
6.2	Challenges: Community Creation and Automation	133
6.3	Operations: Publish	136
6.4	Operations: Search	144
6.5	Community Effect.....	147
6.6	Multiple Central Servers.....	151
6.7	Conclusions.....	154
CHAPTER 7 CONCLUSIONS AND FUTURE WORK.....		158
7.1	Contributions.....	158
7.2	Future Work.....	160
REFERENCES		164
A	APPENDIX: XML SCHEMA PRIMER	177
A.1	Elements.....	178
A.2	Attributes	178
A.3	SimpleTypes.....	179
A.4	ComplexTypes.....	180

A.5	Deriving Types	181
------------	-----------------------------	------------

List of Figures

Figure 1: Internal Layout of an ID3v1 Tagged File [Nilson00]	17
Figure 2: JxtaSearch model.....	18
Figure 3 Example of a <i>text</i> queriespace: searching for documents about “U-P2P”	19
Figure 4: Hybrid Peer-to-peer Model	20
Figure 5: Gnutella (decentralized) model	21
Figure 6: Two approaches to query routing.....	24
Figure 7: Alpine model	26
Figure 8: Associative Peer-to-peer model.....	27
Figure 9: Search vs. Location	29
Figure 10: FASD query routing.....	34
Figure 11: File type support in Kazaa and Limewire	38
Figure 12: Superpeer model.....	48
Figure 13: Chord key lookup	50
Figure 14: General structure of the ObjVlisp model [Cointe87]	70
Figure 15: Relationship between Communities and Files	71
Figure 16: Simplified Schema for Communities	72
Figure 17: Communication between U-P2P clients.....	75
Figure 18: A schema for the "mp3-nap" community.....	77
Figure 19: An XML file described by the "mp3-nap" schema	78
Figure 20: U-P2P vs. existing user interaction model	81
Figure 21: A Simple Schema.....	82
Figure 22: A Simple Form.....	82

Figure 23: Role of XSLT in U-P2P	83
Figure 24: Generation of create interface from a XML schema.....	84
Figure 25: Generation of display interface from a XML instance.....	85
Figure 26: A Schema for Communities (community.xsd).....	87
Figure 27: The Root Community (community.xml).....	88
Figure 28: Community file for the “mp3-nap” community	89
Figure 29: Repository structure	91
Figure 30: Resource IDs in U-P2P.....	92
Figure 31: Interactions with repository.....	94
Figure 32: Interactions with Network Adapter	101
Figure 33: Publish message flow	102
Figure 34: Search message flow	103
Figure 35: Stamps.xsd - Schema for a Stamp Community.....	110
Figure 36: default-resource-create.xsl: Default Stylesheet for Create form.....	111
Figure 37: HTML output of <i>create.jsp</i> rendered in browser	112
Figure 38: Class diagram for the WebAdapter and its implementation [Arthorne02d] .	114
Figure 39: Role of Servlets and JSPs in U-P2P [Arthorne02d]	116
Figure 40: Internal data structure for resolving resource IDs	119
Figure 41: Internal data structures for locating files on central server	120
Figure 42: Role of DownloadServlet and DatabaseViewer components.....	122
Figure 43: Class diagram for Repository and its implementation. [Arthorne02d]	122
Figure 44: Class diagram for PeerNetworkAdapter and the generic implementation. [Arthorne02d]	124

Figure 45: Example SearchRequest message	126
Figure 46: Example SearchResponse message with results	127
Figure 47: Example SearchResponse with no results	127
Figure 48: Example RegisterRequest message (initial registration).....	128
Figure 49: Example RegisterRequest message (removing resources).....	128
Figure 50: Example RegisterRequest message (full registration).....	129
Figure 51: Example RegisterResponse message (unknown resources).....	129
Figure 52: Example RegisterResponse message (known resources).....	129
Figure 53: Example RegisterResponse message (removing/deregistering resources) ...	130
Figure 54: Time to Publish a File vs. Number of Files.....	137
Figure 55: Cumulative Publish Time.....	137
Figure 56: Operations involved in publishing a file	139
Figure 57: Time to Process Resource in Client vs. Number of Files.....	139
Figure 58: Time to Store Resource in Client DB vs. Number of Files.....	140
Figure 59: Time to Publish from Client to Server vs. Number of Files.....	141
Figure 60: Publish message flow	141
Figure 61: Time for Server Resource Lookups vs. Number of Files.....	142
Figure 62: Time to Store in Server Database vs. Number of Files.....	143
Figure 63: Comparison of Total Publish Time vs. Time for Server Operations.....	143
Figure 64: Search time with a Fixed Number of Files.....	145
Figure 65: Search time vs. Number of Files	146
Figure 67: Operations involved in searching for a file	146
Figure 68: Server DB Search Time vs. Total Search Time	147

Figure 69: Publish Times With Communities Present.....	148
Figure 70: Effect of Other Communities on Time to Add a File.....	149
Figure 71: Effect of Other Communities on Search Times	150
Figure 72: Multiple Server Deployment.....	151
Figure 73: Time to Publish Files With Three Servers	153
Figure 74: Publish Times on Different Servers	153
Figure 75: Time to Publish a File (Using eXist 0.8).....	155

List of Tables

Table 1: Summary of Existing Approaches to Communities	42
Table 2: Server Properties.....	152

Chapter 1 Introduction

1.1 What problems do file-sharing systems solve?

File-sharing systems solve a variety of problems that we didn't realize we had. For example how to make sure that files are accessible 24 hours a day -- or the problem of finding enough storage space for millions of files -- or reducing the load on servers that host popular content. The solution to all of these problems was taking advantage of the thousands of PCs that were traditionally considered the "edges of the internet".

The flow of information was supposed to be from powerful servers - computers with huge hard drives and powerful processors - towards less powerful client PCs. But the incessant march of Moore's law meant that these PCs on the edge of the internet had huge hard drives and powerful processors too. The popularity of broadband access to the internet also meant that many of these PCs had plenty of bandwidth too. This combination meant that edge PCs were well-equipped to become servers themselves.

Napster [Napster02] was the match that started the fire by turning each of these powerful PCs into servers. Napster connected all of these servers turning them into a gigantic always-on super-server with plenty of storage and bandwidth. Its popularity was inseparable from the popularity of the MP3 audio format. MP3s hit the sweet spot of size and quality: users could download a CD-quality song in the space of a few minutes to a few seconds depending on the quality of their connection.

Napster allowed users to connect with each other to great effect. The abundance of available hosts meant that intermittent usage patterns by individual hosts did not affect the general availability of files: there was usually someone on line with the file you wanted. In addition, it meant that network load was reduced since the best download speeds usually came from users who were close to you on the network. Rather than thousands of users bringing a single server to its knees, demand was balanced over the multiple nodes hosting replicas of the file. And each successful download added another host to that list.

Since the unity of Napster the file-sharing environment has fractured into a multitude of different applications. These arose in response to some of the weaknesses of Napster, most importantly the central point of failure represented by having a central server hosting the database of files and their locations. These systems have also expanded the repertoire of files to videos, documents and more. Today, despite the demise of Napster, the world of file-sharing is very much alive. It has moved beyond the initial problems of robustness, load balancing and storage to the new set of problems arising from its enormous success.

1.2 Weaknesses of File-sharing Systems

Napster exposed a variety of weaknesses with the centralized approach. Gnutella [Kan01] was a visceral response taking the opposite approach of completely eliminating any sort of centralized control. Other file-sharing applications occupy a middle ground where especially powerful PCs can become more equal than others forming a hierarchy of nodes. Freenet [Clarke00] takes a decentralized approach as well as focusing on issues

of anonymity and censorship resistance which is an area of weakness in other systems.

We discuss here some of the weaknesses of file-sharing systems that are most relevant to this thesis.

1.2.1 Inadequate Search

Napster relied on two simple fields, artist and title, as the main indicator about a file's content. [Dornfest01] These fields were then compared against filenames to find matches. Filenames, although simple to create and understand, are not the most effective source of metadata. Some major problems with filenames are the limited space and lack of structure.

The exception to this is the ability to search metadata for music and video formats in newer file-sharing systems such as Kazaa [Kazaa03] and Limewire [Limewire03]. These systems allow users to specify keywords for individual attributes of a file such as artist or title. Unfortunately, for less well-known formats, filename search is still the default. Even for popular types like MP3s, some important information is not searchable.

1.2.2 Limited to Popular Formats

Napster addressed the needs of a specific community: people sharing MP3 audio files. Since then, support for other formats has been added in successive applications: video, applications, pictures, etc. Unfortunately, application designers control this list of supported formats. It is not possible today to quickly create a community for sharing formats outside of this small list.

Applications like Napster and Kazaa that have explicit support for a few different formats cannot easily be extended to support new formats. The knowledge of those formats is embedded in the application and new formats can be added only by modifying the source code.

File-sharing programs like Limewire technically allow the sharing of any type of file. Limewire supports metadata search for a specific list of formats, but beyond that relies on filename search. This inability to extend metadata search abilities to new formats is also a barrier.

1.2.3 Fragmented Communities

Another weakness in the state of the art is the fragmentation of file-sharing communities. Each application is an island unto itself. There is no way to tell whether a community of users with common interests exists other than word of mouth or ad hoc search methods. This same problem can lead to users reinventing the wheel rather than taking advantage of communities that already exist. There is no standard mechanism for discovering communities. This would allow users to find communities of interest to them by specifying semantically what type of community they are looking for.

The proliferation of file-sharing applications shows that there is no single combination of design decisions that can satisfy all users. Each choice implies certain strengths and weaknesses. What is missing is a flexible method for creating and discovering new

communities that would allow these diverse communities to co-exist without being isolated. A more difficult goal is allowing decisions to be decoupled: for example allowing a certain format to be moved from one application to another.

1.3 Goals

This thesis presents a novel application framework called Universal Peer-to-Peer (U-P2P) [Mukherjee02, Arthorne03] that addresses the weaknesses in file-sharing systems that are outlined above. There are two major goals:

1. Improving the quality of search in peer-to-peer file-sharing systems.
2. Making it easier for new file-sharing communities to be created and discovered.

The use of structured metadata is the basis for both of these goals.

U-P2P specifies a metadata layer that can be implemented on top of existing peer-to-peer infrastructure. It facilitates the creation of metadata and ensures that metadata conforms to an expected structure which is an important enabler for metadata search. U-P2P improves search by exposing this metadata and handling rich metadata queries. The first goal of improving metadata search is a prerequisite for the second, simplifying the creation and sharing of communities. Improving the quality of search in a format-independent way lowers the barriers for creating communities.

U-P2P makes communities into concrete entities by defining their attributes. These attributes are expressed in a standard XML-based format that U-P2P can use to instantiate a file-sharing application that allows users to take part in this community. This satisfies one of the goals of U-P2P: allowing anyone to create their own community.

Communities by themselves are very useful but as with any networked application, their success depends on attracting users. Even today, with a handful of available file-sharing communities it can be difficult to find an application with the right combination of attributes. This problem will worsen if communities proliferate (which is U-P2P's goal). The solution to this problem hinges on solving the problem of community creation. Expressing the nebulous concept of community in the form of an actual concrete XML file, U-P2P enables the trading of communities. This means that the same techniques for search used within file-sharing communities can be brought to bear on the problem of finding the community itself. This is the other important goal of U-P2P: allowing users to easily find and join the communities of interest to them.

1.4 Approach

The goals of improving search and simplifying community creation and discovery are achieved through a few basic approaches.

1.4.1 Improving Search

All Files are XML Files

File-sharing applications are generally used to share binary formats which include embedded metadata. Each format requires a specialized algorithm for parsing this information from the file which must be integrated into each file-sharing application. In U-P2P, metadata for files of any format can be read using standard techniques. There is

no requirement to implement customized software to extract metadata from each new format. This approach enables rich metadata search for a wide array of file formats.

This is accomplished through the use of XML files. Instead of trading binary files, all files are "wrapped" in XML. Binary files are included as embedded links. The XML metadata is stored in a database and indexed to provide advanced search capabilities for all files in the community. From the perspective of existing applications, U-P2P can only trade in one extension ".xml". But this one format is enough to represent any format.

Use of Schemas

Schemas define the structure of all XML files shared in U-P2P. These schemas are specified using the standardized XML Schema language [W3C01]. Each schema specifies all the fields that are present in the XML file and their data types. This information is crucial for improving the quality of search.

Firstly, the schema can be used to automatically generate an interface for entering metadata for new files. This does not guarantee that metadata will be entered, but without it there is almost no chance that it will be. The schema can also be used to ensure that files are conformant reducing the possibility of malformed metadata. Finally, the schema can be used to automatically generate an interface for metadata search, allowing users to take advantage of the available metadata to target their search effectively.

The use of a standardized text-based format makes it easy for users to define their own formats. Customized software is not required for each format; instead the same standard algorithms can be used to handle all of them.

1.4.2 Simplifying Community Creation and Discovery

Communities as Tuples

One important attribute of file-sharing communities is protocol. In general the choice of a file-sharing application implies a choice about protocol. The question of what formats are supported by the community is secondary: you are expected to choose a protocol first and then make do with the formats supported.

We see file-sharing communities, in their simplest form, as a tuple of (format, protocol). By format we mean the structured metadata as well as the binary itself since to be effective, each file-sharing community should support metadata search. By protocol we mean the method in which queries and results are communicated between nodes. Format and protocol are orthogonal, and it should be possible to share the same format over other types of protocol. This contrasts with the existing approach which subordinates formats to protocol. It is not possible to compose unique communities from these monolithic applications. It is not possible to reuse the peer-to-peer infrastructure with some format not already embedded within the application. Expanding the tuple idea, the "protocol" aspect can be further decomposed into attributes such as network organization, anonymity, encryption and authentication.

The concept of communities as tuples of the form (format, protocol) which can be tailored to a particular application is crucial to simplifying the creation of file-sharing communities.

XML Format for Communities

The design decisions of the popular file-sharing applications are not exposed to users. Instead they are embedded within the application code. Allowing users to create their own communities requires exposing these decisions and allowing them to be specified. U-P2P proposes a schema that sets out what fields a community declaration should include. The schema for communities specifies a standard format (written in XML) for U-P2P community files. This method of specifying communities is much more open to the user -- allowing existing communities to be edited or new communities to be easily created. In addition, the decisions are factored out from the application - instead of hard coding them, the application must be able to interpret and instantiate these decisions.

Communities are Objects

The idea of community in existing applications is simply that: an idea. Creating a new community is a matter of software design. Finding a community involves talking to your friends or reading message boards about which service is best. The idea is made concrete by expressing the community as an XML file. A file can be manipulated much more easily by a computer than an idea can.

The idea that communities are objects is the key to U-P2P Community discovery. U-P2P communities are, after all, file-sharing applications with the ability to discover and download files. As an object, communities can benefit from this same infrastructure. Finding a community requires describing the community you're looking for and dispatching a search to the other nodes in the network. Joining a community is simply a matter of downloading the appropriate community object.

1.5 Contributions

The contributions of this thesis to the state of the art are an improvement in search in file-sharing peer-to-peer systems and simplifying the creation and discovery of file-sharing communities.

1.5.1 Improving search

U-P2P improves on the state of the art in file-sharing systems by bringing metadata search to a wider range of formats. In existing applications, it is generally difficult to provide this ability to file formats which are not well known. This presents a barrier to using peer-to-peer file-sharing for a wider array of applications. The key to providing improved search abilities without customized handling for each format is a standard metadata layer.

Standard Metadata Layer

The use of schemas to define formats is required by all U-P2P communities. From the schema, U-P2P can generate an application. Since the schema contains information about what data must be present in the file, it can be used for both creation and search. When creating a new file, U-P2P can prompt for the required fields to generate a new file, or use the schema to validate an existing file. When searching, U-P2P provides the searcher information about what fields are available for search. This is processed by a database capable of handling complex queries on structured metadata. All of this functionality can be generated for any schema written in the standard format.

This functionality constitutes a standard metadata layer because it is independent of the underlying peer-to-peer protocol. As discussed before, the list of supported formats (supported by metadata search) is often dictated by the choice of protocol. The U-P2P metadata layer can be implemented on top of a variety of protocols allowing these two parameters to be varied independently.

1.5.2 Simplifying Community Creation and Discovery

The improvements to search described above are prerequisites for the other important contribution of this thesis: simplified community creation and discovery. U-P2P communities are unique tuples of the form (format, protocol, ...). The format (i.e. schema) allows each of these communities to provide powerful services for creating and discovering files. These abilities are moot if users are not able to create and more importantly discover these diverse communities.

User-designed Communities

U-P2P is a framework that allows anyone to design a file-sharing community. The aspects that determine how a community operates have been decoupled and the configuration of the format and other aspects can be specified in a simple and standard XML file. This substantially simplifies the creation of communities and means that the benefits of peer-to-peer file-sharing can be extended beyond a narrow range of uses.

A Community for Communities

U-P2P communities are files. This opens the door to community sharing among users. Since all community definitions are created within the same framework, users can easily discover other user-created communities. U-P2P's structured approach greatly simplifies the task of finding a community. A standard schema describes the structure of each community. This allows communities to be compared and searched according to various criteria. This is simpler than the ad-hoc methods (word of mouth, web searches) used to discover communities today.

Performance and Scalability Gains

Search in peer-to-peer networks can be quite costly because of the number of objects and nodes which are involved. Centralized peer-to-peer models require a large database to maintain information about where objects are stored. Distributed models trade off database size against the costs incurred by having to query many nodes. Communities offer a way of partitioning these resources intelligently. Narrowing search to a given community reduces the load on resources outside the community without compromising

the quality of results. This results in better search performance and allows communities to scale to more resources and nodes than they otherwise would be able to.

1.6 Thesis Overview

The thesis is divided roughly into the following broad areas: introduction, state of the art, design and performance evaluation. This section concludes the introductory material.

The following sections detail existing research in the field of file-sharing systems.

Existing approaches are evaluated with respect to their support for search and the ability to create and discovering communities.

The design section is prefaced by a detailed discussion of the meaning of “community” in the U-P2P framework. The next sections describe the design of U-P2P at a high-level perspective and then in more detail. This includes a discussion of XML and why it was chosen for schemas and for other uses within U-P2P. The architecture and design of the major components of U-P2P are presented as well as the challenges arising from each one.

The final section is an evaluation of the performance of the U-P2P application. The need for certain key metrics is discussed. An explanation of the process used to evaluate these metrics is followed by a discussion of the results. A summary of conclusions and suggestions for future avenues of research concludes the thesis.

Chapter 2 State of the Art

2.1 Introduction

Research into peer-to-peer file-sharing has experienced a burst of activity in the past few years. There are a variety of important areas but we focus here on approaches to improving search and how existing applications support the creation and discovery of file-sharing communities.

One of the easiest ways to improve the quality of search results is to use metadata. Generally, peer-to-peer file-sharing applications are trending more towards the use of structured metadata. Metadata can also be used in decentralized systems like Gnutella to improve performance as well as the quality of results. This section also discusses communities as an important way of improving search. We discuss a few different approaches to designating which nodes belong to each community. Finally, we talk about a class of systems called “location-centric” systems in which search can be difficult. Despite that, there are some interesting proposals for providing search facilities in these systems that may have wider applications.

The problem of creating and discovering communities in existing applications is the next major topic in the state of the art. Metadata search is very important to the usability of file-sharing communities and inadequate search can be a barrier to creating effective new file-sharing communities. We propose a classification of existing approaches with

respect to their ability to create and discover communities with support for structured metadata search.

Finally, we move away from the focus on search and communities to discuss some other important ways in which file-sharing applications can vary. We present a brief discussion of differing network organizations, the difference between structured and unstructured systems and systems which emphasize anonymity and censorship-resistance. Each of these parameters involves some kind of engineering trade-off. Considering these trade-offs encourages thinking about peer-to-peer file-sharing systems in terms of a combination of various design decisions rather than the current paradigm of monolithic applications.

2.2 Search

The use of metadata is receiving increasing attention in peer-to-peer research. Metadata is simply information about information. This definition encompasses filenames, hash-based keys, embedded metadata (e.g. ID3 tags in MP3s), external metadata, link structure, etc. This thesis focuses on semantically meaningful metadata – human readable metadata that can also be used by machines to decide whether a document matches a given query.

All metadata used in peer-to-peer systems is “structured” in some way to allow computers to parse and compare it. The trend has been towards increasing this

information and making it even easier for computers to manipulate. We begin by discussing the varying levels to which metadata is used in existing applications.

2.2.1 Metadata

As discussed earlier, the original Napster design used a single piece of metadata – filename – to facilitate search. This simple approach proved effective since the important information about MP3s - artist and title – can easily fit in the space of the filename. This convention has carried over to many other file-sharing systems. Despite its effectiveness, this is not a scalable solution:

1. There is no convention even on how artist and title should be specified within the filename (e.g. artist-title, title-artist, etc.).
2. There is no convention about whether new fields can be added or how they would be expressed.
3. The filename must be of reasonable length limiting the number of fields that can be expressed.
4. There is no convention about how metadata for files other than MP3s should be expressed in the filename.

Kazaa addresses some of these concerns. This system exploits some well-known metadata conventions such as the MP3 ID3 tag. The MP3 ID3 tag is a binary metadata format that can be embedded in MP3 files [Nilsson00]. It specifies which fields are available and where they are found in the tag. Kazaa indexes these tags and allows users to perform searches on different fields. The limitation of the ID3 tag is that it is rigid – it

only describes one type of object: MP3s. There is no way to extend it to describe other formats.

Figure 1: Internal Layout of an ID3v1 Tagged File [Nilson00]

XML is a more general method of expressing metadata. The available fields and their layout are configurable. Despite this flexibility, there are enough conventions that software for parsing generic XML documents is widely available. These properties mean XML can be adapted to express metadata about a broader range of objects without losing the ability for machines to process that data. Limewire is an example of a peer-to-peer system that uses XML to express the metadata structure for the file types that it supports.

Limewire includes a few schemas written using an XML-based language called XML Schema [Thadani01]. According to the W3C, XML Schema provides a means for “defining the structure, content and semantics of XML documents” [W3C01]. Schemas

describe how metadata about various types of files – audio, video, documents, etc. – should be expressed. XML Schema allowed the creators of Limewire to specify structured machine-readable metadata for a wider variety of file formats than Napster. Unfortunately, schemas are embedded within the Limewire application making it difficult to extend support to new types of files.

JxtaSearch [Waterhouse01] uses a more extensible approach to metadata search. JxtaSearch is a distributed search system built on top of Sun's Jxta peer-to-peer programming environment [Gong01]. JxtaSearch's "queryspaces" are analogous to the schemas used in Limewire. Queryspaces allow "providers" to notify "hubs" that they are willing to answer certain types of requests.

Figure 2: JxtaSearch model

When queries arrive at a hub they are routed to providers that have registered to answer that type. Querspaces are specified in XML format and may be created by anyone (although to be useful there should be providers in the space).

```
<?xml version='1.0'?>
<request xmlns=http://search.jxta.org
  xmlns:t=http://search.jxta.org/text
  query-uuid=1C8DAC3036A811D584AEC2C23
  query-space=http://search.jxta.org/text>
  <t:query>
    <t:text>
      U-P2P
    </t:text>
  </t:query>
</request>
```

Figure 3 Example of a *text* querspace: searching for documents about “U-P2P”

JxtaSearch specifies the structure of querspaces, registration messages, etc. using custom schema rather than XML Schema.

Interestingly, Jxta by itself supports the idea of publishing metadata with the concept of “advertisements”. Although not explicitly used for file-sharing, advertisements are structured XML documents used to publish information about Jxta resources (peers, peer groups, services, etc.). A few major advertisements are defined and users can extend or override these using XML Schema, though the mechanism is not clear from the specifications. Advertisements can be discovered using the Peer Discovery Protocol (PDP).

2.2.2 Query Routing

Query routing is another important way in which metadata can be used to improve the quality of search. Research in this area is motivated by a problem specific to decentralized or “pure” peer-to-peer systems. Before discussing query routing in detail, we first contrast centralized and decentralized peer-to-peer file-sharing systems with respect to search.

The simple idea behind the now defunct Napster is using a centralized database to track the files hosted by each peer. Another early peer-to-peer application, the instant messaging system ICQ [ICQ03], is based on a similar idea. ICQ uses a centralized directory which stores information about each user’s location in the network and whether they are present.

Figure 4: Hybrid Peer-to-peer Model

In both these applications information about other nodes is stored in a central database. Searches are sent to a central server, which returns information about the remote peer where the file or user can be found. With this information, the client connects directly to the remote peer and initiates a download or a conversation. The reliance on a central server makes these systems centralized. These systems are also referred to as “hybrid” because of the combination of client-server and peer-to-peer interactions.

Systems that do not rely on any centralized authority are referred to as “pure” peer-to-peer systems. Two important examples are Gnutella and Freenet. In these systems, peers act as both servers and clients thus they are often referred to as “servents” [Kan01]. In Gnutella, search requests are sent to neighbouring nodes. The neighbours simultaneously return any matching results from their own databases and forward the request on to their neighbours.

Figure 5: Gnutella (decentralized) model

Queries ripple outwards from the initiating node in an attempt to reach as many nodes as possible. At each node the request's time-to-live (TTL) field is decremented. Requests with a time-to-live of zero are discarded to limit traffic. The finite TTL enforces a horizon on the propagation of the query. Search results return through the same path back to the originator. As in Napster, downloads are handled by a direct connection to the hosting peer.

Query routing is one of the most active areas of research for improving the scalability of Gnutella (and pure peer-to-peer systems in general). Clearly, flooding all the nodes within a given horizon is very costly in terms of bandwidth. The optimal approach would be to only forward queries to nodes which will return good matches. This technique improves search: for the same amount of network traffic, the quality of results is improved. In a hybrid peer-to-peer system, query routing is not required since the central server handles all search requests.

Although it is impossible to know a priori which nodes have matches, we can guess which neighbour is most likely to produce matches. This requires choosing some metric to indicate the best neighbour. Metadata is one useful source of data for calculating these metrics: metadata associated with files as well as in the query itself. We are most interested in this thesis in query routing methods which exploit metadata but it is worth pointing out that Yang has proposed a few metrics that don't rely on metadata [Yang02]. Her study found that directing queries to neighbours that had provided the most results in

the past resulted in a high probability of satisfaction with reduced search times and bandwidth usage.

Metadata routing underpins systems such as Limewire's query routing proposal [Rohrs02], Routing Indices [Crespo02], and Neurogrid [Joseph02]. Also referred to as "semantic routing" [Joseph02b], these systems use the query itself to inform their routing decisions. Internet routers are able to determine the interface that will take the packet closer to its destination. Query routers attempt to determine for a given *keyword* the neighbour that will take the query closer to matching documents. Closeness is defined as the similarity between the metadata specified in the query and the metadata of the document.

Limewire's query routing proposal resembles traditional routing approaches. Each node maintains a table that lists for each neighbour the number of hops required to reach a node with documents matching a specific keyword. Searches are directed to the neighbour advertising the lowest number of hops for the keyword. Nodes learn and update hop counts by trading routing tables with their neighbours. The Routing Indices approach uses information about the number of documents accessible via neighbouring nodes. When two nodes establish a connection they exchange information on the aggregate number of documents accessible through their existing neighbours. Queries are forwarded to nodes with a high proportion of documents on a given keyword – nodes with larger collections (independent of subject) will be favoured.

Figure 6: Two approaches to query routing

Limewire and RI take routing information from other nodes at face value. This can cause problems if some nodes advertise incorrect or misleading metadata. Neurogrid addresses this problem by relying on user feedback to rate the reliability of results - for example: "If a remote server is queried with a word such as 'automobile' and returns adverts for weight-loss products, the reliability of this server with respect to this query should be reduced." [Joseph02] Each neighbour is associated with different keywords with a strength that can be reinforced or weakened. Users can give feedback on the quality of matches by ignoring them, bookmarking them or marking them as spam. Queries are forwarded to the nodes deemed most reliable on the subject.

One important point to note is that Limewire, RI and Neurogrid use metadata as an implicit indicator as opposed to an explicit indicator. Implicit indicators mean the

assessment of how close a document is to a given query is left entirely to the routing mechanism. There is no way to explicitly target a query to a specific set of documents (i.e. “just .pdf files about peer-to-peer”). Queries and documents are not assumed to be structured. In the case of Limewire, which does use structured XML queries, they suggest the XML markup first be stripped out before analyzing whether a document matches the query. [Thadani01b]

2.2.3 Communities

Another approach to optimizing search is to explicitly define a subset of nodes most likely to return good matches and then restrict one’s search to those nodes only. These nodes form a community within the larger peer-to-peer network. By routing queries to nodes within the community, traffic is reduced without reducing the quality of results. JxtaSearch [Waterhouse01], Associative P2P [Cohen03] and Alpine [Alpine03] all fall into this category. One key differentiator between these systems is the method used to define a community.

JxtaSearch systems consist of consumers, providers and hubs. Consumers issue search requests which are serviced by providers. Queries are routed to providers by hubs. Peers may take on one or more of these roles. In JxtaSearch, communities of interest are defined by queryspaces. Queryspaces define the “space” of queries which a provider is qualified to answer and consist of a request type and one or more predicates. For example providers may advertise that they can handle requests of type “*conference paper*” where the subject matches “*peer-to-peer*”. Incoming queries matching this

template are directed by the hub to this community of providers. Jxta, the environment used to implement JxtaSearch, also includes the concept of communities or “peer groups” which are used to define a collection of peers providing a service rather than for routing searches.

Alpine and Associative Peer-to-peer communities are formed according to the searchers own choices. Providers do not choose their community as they do in JxtaSearch. In Alpine, users designate groups of peers that they consider authoritative on different

Each node connects directly to the peers they've grouped.

Figure 7: Alpine model

subjects. For example, a user can create their own music group and explicitly include nodes which have returned good results in the past. As with Neurogrid neighbour

rankings, these groups are a reflection of the user's feedback and each user's groupings will be unique. A notable difference of Alpine (versus Neurogrid) is the lack of query forwarding – each node connects directly to others in the community. This eliminates the possibility of taking advantage of the knowledge of other users to discover new search candidates.

Figure 8: Associative Peer-to-peer model

In Associative Peer-to-peer systems the user does not explicitly define which peers belong to each community. Instead associations are formed using concepts from collaborative filtering: nodes with similar interests are assumed to be the most likely

place to find results. Communities are composed of nodes sharing a common attribute or "guide-rule" indicating a common interest as well as acting as a guide to which nodes should be searched. One simple indicator is possession of a file ("possession-rule"): a community is defined as the subset of nodes that share a certain file. The tricky question for the searcher is choosing which file (and thus community) will lead to the optimal results. The authors propose two methods:

- Random Possession Rule – Files are chosen randomly from the local store and queries directed to the resulting community until a match is found.
- Greedy Approximation Strategy – Choose the files that are most likely to return matches based on their past performance.

This approach is also based on implicit indicators. Rather than providing explicit directives in the form of structured metadata, possession of a file acts as a proxy indicating common interests.

2.2.4 Search for Location-centric Systems

Another important area of research in peer-to-peer file-sharing is "location-centric" systems such as Chord [Stoica01, Dabek01, Dabek02] and Freenet [Clarke00]. These systems tackle the problem of efficiently locating a file in a purely decentralized system. The methods used make it difficult to search for files using keywords or more structured metadata. Before discussing techniques to improve search in these types of network, a brief discussion of location-centric systems is presented.

Figure 9: Search vs. Location

What is the difference between "search" and "location"? Existing file-sharing systems can be divided roughly into two groups based on this definition. In systems such as Napster or Gnutella, the user first performs a general search - usually a keyword search - and a set of results are returned. This is what we define as "search". "Location" is used here to refer to searches for an exact identifier - for example an URL that can be used to find where the file is hosted and transfer it back to the requester. In general, searching can be broken down into steps: a user searches for files matching certain criteria ("search") and then retrieves the desired result using its unique identifier ("location"). [Joseph02]

All peer-to-peer file-sharing systems provide a location service - without this ability it would be impossible to retrieve files from the system. However, there are many important file-sharing systems that do not provide a search service as outlined in the previous sections. These can be divided into roughly two groups: content-addressable systems which focus on providing efficient location of files in a massively distributed environment and privacy-conscious systems which do not provide search because it would make it more difficult to provide anonymity and censorship resistance.

Freenet is one example of a privacy-conscious system that does not provide search. The design goal of Freenet is providing anonymity for requesters and publishers. One decision that follows from this goal is using encryption to make it difficult to determine the content of documents. This makes keyword search using full-text search impossible. Filename search is also impossible. Unlike Gnutella files which are identified by human-readable names, Freenet files are identified by a key generated by hashing the file's contents (content-hash). A Freenet CHK (Content Hash Key) such as `CHK@dJxYMIGaCiFu3xaRohwhIE1~IQ0WAwI,QXF-c-ZUyU2JNRwyz39pVg` cannot be used for keyword matching. Files may also be identified by human-readable names such as `KSK@gpl.txt` but it is not possible to query a Freenet node to discover a list of these identifiers. Nodes only deal in CHKs – the client must hash the human readable name to discover the CHK of the actual file. This hashing operation means that in order to retrieve a file you must know its exact identifier.

The concept of the Content-Addressable Network (CAN) was first introduced by Ratnasamy et al. [Ratnasamy01] to refer to systems which implement the equivalent of a distributed hashtable where content can be efficiently located in the network given a corresponding key. These systems are also referred to as Distributed Hashtables (DHTs). In this thesis we will refer to these systems as DHTs to distinguish from the system proposed in Ratnasamy et al.'s paper which is also confusingly called CAN. Some other examples of DHTs are Chord [Stoica01] and Pastry [Rowstron01]. These improve on the key-based routing system of Freenet by specifying exactly where each key is stored in the network. Each node is given an identifier which corresponds to a point in the key space and takes over responsibility for hosting keys in its vicinity. As with Freenet, these systems rely on keys generated through content-hashing to identify and route files. These identifiers provide no clue as to the subject matter of a document. Additionally, files may be encrypted making it impossible to conduct full-text searches on the files stored at each node.

The keys used in Freenet and DHTs are a form of metadata, but not a human-readable form or even one that a machine can use to determine anything about a document besides its location. This doesn't mean it is impossible to search these systems. Additionally, their powerful location abilities make it a worthwhile task to investigate this topic. One heartening example is the World Wide Web which is itself a location-centric system: published documents have unique identifiers (uniform resource locators - URLs) that indicate where a file is hosted. The wide availability of search tools for the WWW proves that location-centric systems need not be un-searchable.

However, the WWW has properties that cannot be exploited in peer-to-peer systems:

1. link structure - documents contain references to each other which can be used to discover new documents
2. human-readable metadata - the names and content of documents on the WWW are generally unencrypted allowing humans and machines to infer their subject matter

Link structure is difficult to reproduce in peer-to-peer networks because links quickly become inaccurate as nodes join and leave the network. Additionally, stored documents are usually not hypertext. Some of the query routing methods described earlier attempt to use links between nodes as a proxy for link structure. This is possible if linkages between nodes are not completely random, meaning that a link between nodes expresses some information about the content of the other node.

Another avenue for creating P2P link structure is to create a WWW-like overlay which replaces URLs with unique identifiers. The underlying location-centric system then appears as a single storage device which handles the mechanics of locating the machine which hosts the file. Replication and caching ensures that commonly requested content should be persistent despite the transience of individual nodes. An example of this is the FreeWeb system [Freeweb01] which operates on top of Freenet by replacing location-specific HTTP URLs with generic Freenet identifiers. These links can then be used by spiders in the same way as links on the existing internet.

In the absence of link structure the only way to provide search in location-centric systems is to provide access to metadata. In general once metadata is made available the problem is reduced to that of search in general by breaking the process into steps. Search is conducted on the human-readable metadata using standard search techniques (indices, query routing, etc.). This returns unique identifiers that can be passed to the location layer for retrieval.

There are a few interesting proposals that utilize location-centric infrastructure for search. These include PIER [Harren02], FASD [Kronfol02] and distributed inverted indices. PIER is a system for enabling search in CAN. Documents in CAN are referenced by a hash as with other DHT systems, but they are stored in an unencrypted form at each node. In a PIER system search requests are routed using CAN and a query processor at each node conducts a search on the local store.

FASD (the acronym stands for Fault-tolerant, Adaptive, Scalable, Distributed) is a proposal to provide search for Freenet by generalizing Freenet's key-based routing system to apply to metadata. FASD proposes inserting automatically generated "metadata keys" which include a reference to the document's Freenet key. Metadata keys are tables of weightings for each word in the document generated using TFIDF (term frequency X inverse document frequency). Freenet attempts to route requests to nodes which store documents that are numerically close to the requested key. In FASD, metadata keys are treated as multidimensional vectors (each keyword is a component of the vector, and the weighting describes its length) and closeness is defined as the angle

between vectors. The cosine of this angle is used to measure closeness: a value of one indicates an exact match. This allows nodes to maintain Freenet-like routing tables to

Figure 10: FASD query routing

identify the next hop that will bring the query closer to a node specializing in that portion of the metadata keyspace. This approach inherits Freenet's properties - replication and caching moves popular metadata towards requesters, areas of the network should become specialized in part of the metadata keyspace and unrequested metadata will be dropped from the system.

The designers of Chord and PAST have both proposed using DHTs to store a distributed inverted index [Stoica01, Castro02]. An inverted index allows quick lookup of

documents matching a keyword and can be expressed as a hashtable that stores pairs of the form (keyword, matching documents). The keyword space is mapped to the node space by hashing the keyword to yield an identifier. The node whose identifier is closest to the hashed keyword is responsible for that entry in the inverted index.

Despite their relationship with location-centric systems, the methods outlined above are more generally applicable as search systems that can be used along with any location method. This indicates the more general point that search and location are independent operations which can be decoupled.

Most of the methods presented in this section do not rely on explicit metadata. PIER could certainly be used with explicit metadata since it does not specify exactly how searches are processed at each node. In this it is not much different from say Gnutella except that searches are routed using the underlying CAN infrastructure. FASD deduces the subject matter of documents by analysing term frequency rather than any explicit indicators such as structured XML metadata. Distributed inverted indices attempt to characterize documents using a single keyword which requires some machine driven way of discovering which keyword best characterizes each document. This could be accomplished using structured metadata but with significant loss of information (reducing the description down to a keyword).

2.3 Creating and Discovering Communities

We have now seen how metadata can be used to improve search. Allowing structured metadata to be specified in searches enables more exact requests. In decentralized systems, this additional information can be used to route the request to nodes that most likely have results. Also, structured metadata in the search request and in each file simplifies matching and improves the relevance of the results.

Communities are another way to improve search. Grouping nodes which contain related documents provides another piece of metadata which can be used to direct and service search requests. This section discusses the state of the art with respect to creating and discovering communities with an emphasis on the use of metadata. Communities for sharing MP3s or movies, with varying support for metadata search, already exist, but how difficult would it be to create a Gnutella-based community for sharing design patterns with full metadata search and publishing or a similar Jxta-based community for sharing peer-to-peer conference papers? This section discusses the more general question of whether it is possible, with existing applications, to create a peer-to-peer file-sharing community for a new file type, or discover an existing community.

Existing approaches can be classified as being one of:

- Community opaque: There is no way to differentiate communities for different formats.

- Community limited: There is limited support for communities but this is not extensible.
- Community implicit: Communities are formed based on implicit indicators.
- Community unshared: Communities are in the eye of the beholder and are not shared entities.
- Community partial: Providers explicitly join communities but can provide any kind of result.

These approaches are discussed in more detail below.

2.3.1 Community opaque

Early Gnutella clients and location-centric systems like Freenet and DHTs are opaque to communities. Search is done by filename matching or key matching but there is no mechanism for structured metadata search or publishing, therefore a community with support for metadata search is impossible.

Communities of users may come into existence in an ad hoc way but there is no formal support for them. Filename extensions, for example, may be used in Gnutella to define an ad hoc community. This relies on publishers following a convention for file extensions and searchers using that term in their searches. Even with these conventions lack of structured metadata in either the query or files increase the possibility of false positives.

In DHTs, the content-hash based identifiers and opacity of file contents obscure differences between files. The lack of any formal way to designate communities means that discovering communities is similarly ad hoc.

2.3.2 Community limited

Systems like Napster, Kazaa and Limewire support a limited number of communities.

Napster provided effective support for sharing MP3s. Kazaa and Limewire offer schema-based rich search for certain file formats: audio, video, books and software for example.

Figure 11: File type support in Kazaa and Limewire

Although there is no restriction to sharing new file types, beyond these specific formats there is no support for metadata search. Instead, they rely on simple filename matching, thus reducing to the "community opaque" class of systems. Since the schemas are embedded in the application, users can easily discover which communities are available from the application UI which lists the available choices. However, this binding between the application and the list of supported formats makes it difficult to expand the application to support new formats with effective search facilities.

2.3.3 Community implicit

Query Routing, Neurogrid and Associative P2P are approaches in which users or publishers do not explicitly designate their community. Yet they are not community opaque, or limited - they can support the existence of many communities. We refer to them as implicit community systems because they attempt to identify communities using implicit indicators of common interests.

Query Routing directs queries towards peers that are thought to be more likely to provide good matches for a given keyword. This is judged by indexing their content using techniques like TFxIDF [Kronfol02]. TFxIDF or "Term Frequency times Inverse Document Frequency" determines how relevant a document is to a particular keyword using a product of the frequency with which the keyword occurs in the document and the inverse of the frequency with which the term appears in the entire collection. In Neurogrid, user feedback is used to strengthen associations between keywords and neighbouring nodes [Joseph02]. With these approaches keywords are used as implicit

indicators of interest. In Associative P2P, queries are directed to nodes that possess a certain file [Cohen03]. In this case, the possession-rule acts as the implicit indicator of common interests.

The main drawback of using an implicit indicator is that it can lead to incorrect assumptions. In Associative P2P for example, possessing a file in common with another user does not necessarily indicate other shared interests. Keyword-based communities require common understanding of keyword semantics. For example, does the keyword "Java" refer to the Java computer language or the Indonesian island of Java?

Decisions of this type can be informed by feedback gathered in systems like Neurogrid, but as the query is routed through the network it may encounter a user with a different opinion resulting in false positives. This is another manifestation of the metadata problems covered earlier: lack of metadata makes it difficult to interpret the meaning of queries and documents.

The way that communities are formed makes it impossible to explicitly form or discover a community for a new format. All that one can do is share or search for a format and hope that the application's underlying algorithms will deduce that others sharing that format constitute a community. If structured metadata is gathered and allowed in searches, the information may be ignored or misinterpreted because of these algorithms.

2.3.4 Community unshared

Unlike the systems above, Alpine [Alpine03] users can explicitly choose nodes to form communities based on their own interests. The action of choosing is a form of feedback similar to Neurogrid's mechanism for evaluating results. The side effect of this is that each community is personalized and exists only in the eye of the beholder. There is no common understanding of community membership. Each user's communities and definitions of communities will be different.

The lack of shared definitions makes it difficult to discover communities. Each user must re-invent the wheel when creating a community. The lack of a shared definition also makes it difficult to bootstrap another user's "music" community since his tastes may differ from your own. Bootstrapping a user's "music" community because he has a few files of interest may also be a bad assumption. This is not done in Alpine: since there is no query routing in Alpine, there is no need to rely on other nodes to make these decisions.

2.3.5 Community partial

JxtaSearch [Waterhouse01] queryspaces define communities. A queryspace defines a structured metadata format for search requests. Providers register at hubs to service a specific queryspace. The set of providers registered for a particular queryspace constitutes a community since these nodes have advertised that they may have results matching that search format. Unfortunately, the mapping between file format and queryspace is blurred by the fact that search results need not conform to the same format.

Although this can allow useful features such as returning multiple different formats on a certain subject, it can also result in spurious matches. We refer to them as "partial" communities because although queries follow a structured format, providers may supply results in any format.

Table 1: Summary of Existing Approaches to Communities

Type	Description	Examples
Community opaque	No existing rich metadata search, no way to add it	Early Gnutella, Freenet, DHTs
Community limited	Rich metadata search for some formats but no way to support new formats	Limewire, Kazaa
Community implicit	Implicit indicators are used to identify communities, no way to specify explicitly	Query Routing (inc. RI, Keyword Routing, FASD), Neurogrid, Associative P2P
Community partial	Users can explicitly form groups but each grouping is in the eye of the beholder	Alpine, Neurogrid to some extent (each user's feedback is unique, but query forwarding is used)
Community unshared	Users can explicitly direct rich metadata queries to a community, but response format is not specified	JxtaSearch

There is no standard specified for discovering new communities in JxtaSearch. One possibility in JxtaSearch would be sharing the files describing the queryspace. One problem with this is that queryspaces could not be reused for different topics: for example a queryspace for conference papers could not be used for peer-to-peer conferences as well as networking. Requests for a queryspace are dispatched to all registered providers.

2.4 Other Parameters of File-sharing Systems

Our discussion to this point has focused on search using metadata and the ability to create and discover communities. But research in file-sharing is going in many other directions. In this section, we discuss some of these other directions and how they expand the definition of file-sharing communities.

Peer-to-peer applications can be thought of as community-formation machines. At one level each user has unique requirements and expectations of the system. These will often overlap with those of other users and communities are formed in this intersection of interests. Yet each application imposes a single community on all its users, defined by their use of that application. Separating search from location as suggested by Neurogrid and FASD illustrates one way in which these dimensions can be decoupled. If these parameters can be decoupled, can other parameters of file-sharing communities be similarly decoupled?

It is interesting to discuss these other parameters because they define a design space for file-sharing communities. Instead of envisioning a single system that is all things to all

people, a more practical approach is considering which methods are most appropriate to the application. Some possible applications of this approach would be combining rich metadata search with the serverless, persistent nature of Freenet or the efficient location of a DHT.

It is an open and intriguing question how other important features can be decoupled and combined. There is no mechanism by which users can easily take advantage of the available design options to create communities tailored specifically to their needs along all these axes. Although this subject is not addressed in this thesis, this discussion provides context and gives some background on what the possibilities might be.

2.4.1 Other Classifications for Peer-to-peer systems

In this section, we present some of the criteria that can be used to classify peer-to-peer systems besides their use of metadata for search and support for communities.

Peer-to-peer systems are a complex group of applications. The many attempts to give a definition to the phenomenon of “peer-to-peer” betray the nebulous nature of this label. My personal preference is for Clay Shirky’s definition of peer-to-peer applications as those that take “advantage of resources -- storage, cycles, content, human presence -- available at the edges of the Internet.” [Shirky01] Within this definition lies the kernel of taxonomy.

One way of evaluating peer-to-peer systems is by looking how well they satisfy various practical properties. Minar [Minar01] proposes the following criteria for evaluating peer-to-peer applications: manageability, coherence (how accurate is the information), extensibility (ability to grow), fault tolerance, security, resistance to legal intervention and scalability. These are important design factors for peer-to-peer applications.

A comprehensive paper by researchers at HP divides peer-to-peer applications into four groups [Milojicic02]: distributed computing (cycles), file-sharing (storage, content) and collaboration (human presence) and platforms (a generic platform for all three). Within each of these categories it is more difficult to divide applications into clear-cut categories. Here we are concerned mostly with file-sharing systems, but the classifications discussed below can be applied to each category of peer-to-peer systems.

One of the earliest of these classifications was drawn between hybrid and pure peer-to-peer systems. Napster characterized the hybrid approach: queries were handled at a central server although download was handled directly between peers. Gnutella and Freenet typify the pure approach. Gnutella queries are sent to neighbouring peers that relay them onwards through the network with no reliance on central authority of any sort.

More recently, work on DHTs has led researchers to highlight the distinction between structured and unstructured networks [Lv02]. Structured systems use deterministic methods to determine where files are stored, drastically reducing the number of hops required to locate an item. This is in contrast to unstructured networks such as Gnutella.

These systems cannot put an upper bound on search times because file locations are arbitrary. On the other hand, structured systems have limited abilities to support keyword search or metadata search. They rely on unique identifiers making it difficult to satisfy partial match queries. As we have seen, support for keyword and metadata search varies widely even among unstructured systems.

Another popular differentiating criterion is the level of anonymity a system can provide. Freenet, for example, guarantees anonymity to publishers and searchers: it is not possible to determine which node originated a search request originated or inserted a new file [Clarke00]. In addition files stored in the system are encrypted. Attention to security distinguishes Freenet from many other systems.

Each of the above is a valuable method for differentiating applications, but any one of these attributes is insufficient to completely characterize a peer-to-peer system.

File-sharing applications can be modelled as a combination of design decisions about each of these parameters. These criteria are examined in detail below and some of the existing research in each area is discussed.

2.4.2 Network architecture

Network architecture is a major differentiator between existing peer-to-peer file-sharing systems. We discuss the three major approaches: hybrid, pure and super-peer.

The intuitive and simple design of hybrid systems, with their reliance on a central server, means that traffic between peers is kept to a minimum. Another important property is their ability to answer with certainty the question of whether a file is present within the system. However, the centralized database presents a single point of failure that undermines the robustness and availability of the peer-to-peer network. Scalability is also an issue. Although the cost of storing files is spread among peers, the cost of storing the database and handling queries is concentrated squarely on a single central server. This cost increases linearly with number of peers.

Gnutella's fully decentralized approach means that all nodes are equally important. This removes any centralized point of failure. However, Gnutella's flooding approach has consequences for scalability. Instead of a single message – all that is required in a centralized system – Gnutella queries may cause many messages to be sent. For small communities this is not a problem but in larger networks query traffic may overwhelm certain nodes [Clip200].

Super peer systems are a combination of the pure and hybrid approaches. One popular example is the network architecture used by the application Kazaa. As with Gnutella, each peer can operate as both client and server. The difference is that some peers are more equal than others. Peers meeting certain criteria for bandwidth, latency and processing power are dynamically designated as super peers. [Yang03] Super peers act as Napster servers in miniature, aggregating information about files on many connected

peers. However, the super-peers themselves form a pure peer-to-peer system that is used to handle queries without reducing accuracy.

Figure 12: Superpeer model

Super peer systems remain highly decentralized. The costs of handling queries are spread across multiple super-peers and there is no central point of failure that can shut the entire system down. Additionally, network traffic is reduced since fewer nodes broadcast queries. Placing this burden on high-bandwidth peers prevents low-bandwidth peers from being overwhelmed by queries and the resulting disruption in network connectivity.

2.4.3 Structured Systems

Lv et al. propose dividing peer-to-peer systems into two classes: structured and unstructured. The defining characteristic of structured systems is that "the overlay topology is tightly controlled and files (or pointers to files) are placed at precisely specified locations." [Lv02] Central to the structured concept is the idea of a key space. Files (or pointers to files) are subjected to a hashing algorithm (e.g. SHA-1 [NIST95]) yielding a single key value. Each node in the system is responsible for a cluster of keys that are "close" to each other using some algorithmic definition of distance. This knowledge is used to efficiently route requests for certain keys to the appropriate node.

In Freenet, the division of the key space occurs in an ad hoc manner. As a consequence of its key-based routing algorithm Freenet nodes become specialized in a certain section of the key space. Chord and similar approaches such as CAN and Pastry are also referred to as Distributed Hashtable systems (DHTs). In a hash table, (key, value) pairs are stored in buckets by applying some function to the key to determine which hash bucket it will be included within. In a DHT, (key, value) pairs are stored at specific nodes by applying a function to the key to determine at which node it will be hosted. In Chord for example, the documents as well as IP addresses of nodes are hashed. Each node is responsible for the portion of the keyspace adjacent to its own hash value. DHTs can be used for document storage: the value is the document and the key is generated using a consistent hashing function applied to the document. Use of consistent hashing ensures that the number of documents stored by each node is roughly equal.

Figure 13: Chord key lookup

The tight coupling between overlay network and file placement in structured networks allows efficient location of files. Chord, for example, can locate files in a number of hops proportional to the logarithm of the total number nodes, allowing it to scale to very large numbers of nodes. DHTs also have the property of being able to establish with certainty whether a given key and its corresponding file are present.

Unfortunately, reliance on keys is also one of the weaknesses of structured systems. It is not difficult to find a given file provided one has a priori knowledge of that file's key value. This has important implications for searching which we have mentioned above.

2.4.4 Anonymity and Censorship

Anonymity and censorship resistance are desirable features for certain peer-to-peer communities. The justifications for anonymity and censorship resistance in peer-to-peer systems are a controversial topic but we believe the benefits outweigh the potential for misuse. One of the central criticisms of the hybrid model was vulnerability to censorship and lack of anonymity. The use of client-server architectures, whether on web servers or for Napster queries, compromises the anonymity of both publishers and requesters. A central server is an obvious point of attack for those wishing to shut down the system and prevent users from accessing any content on the system.

Gnutella was not explicitly designed to provide increased anonymity or censorship resistance, but its lack of central authority presents a partial solution to the problems inherent in more centralized systems. Queries are assigned identifiers that are used to route replies back to the requester. Information about the IP address of the requester is maintained only within the transient routing tables of the nodes in the reply path. This makes it difficult to discover which node originated a query. [Kan01] Freenet also uses this method for relaying queries, results and retrieving files providing anonymity for publishers and readers. In Gnutella, there is no anonymity for publishers or for readers since file transfers are done using direct client-to-client connections over HTTP. Gnutella's distributed nature gives it some protection from censorship since each publisher must be attacked directly rather than simply shutting down a central server.

Peer-to-peer systems such as Freenet, Free Haven [Dingledine00] and Publius [Waldman00] are specifically designed to provide anonymity and censorship resistance. By encrypting files, storing them with unintelligible names, spreading them over multiple hosts and obscuring locations, these systems make censorship difficult by making it difficult to discover where specific content resides. Freenet is somewhat of a contradiction in this matter: hosts cannot determine which files they are hosting since files are encrypted and named using a hash of the contents. But requesting a file causes it to be replicated on intermediate nodes including direct neighbours of the requesting node making it simple to determine at least one server which hosts the document. And yet this behaviour provides an interesting anti-censorship feature since it actually creates more replicas of the document.

Anonymity for publishers and readers can be provided through the use of anonymous channels. Free Haven and Publius rely on anonymous remailers. As Freenet messages traverse the network each node only knows the preceding and following nodes in the chain. This makes it difficult to find out where a message originated.

Publius and Free Haven also guarantee that files will be stored for a specific period of time. Freenet does not provide this guarantee: frequently requested content is widely replicated but this can push unpopular content (i.e. no one is requesting it) from the system.

Clearly each of these systems has important properties not available in some of the more popular peer-to-peer systems. But these are provided at the expense of efficiency and usability. In general it is more difficult to publish and retrieve documents from these systems. Their design also makes it difficult or impossible to search: without an external record of a document's key, the document may be irretrievable! The authors of Free Haven freely admit that they "emphasize a quality relatively few potential users care about at the expense of something nearly everyone cares about." [Dingledine01] Still, for certain communities this is an acceptable trade off.

2.5 Conclusion

This thesis tackles the challenge of improving search in existing peer-to-peer file-sharing systems. The second complementary goal is making it easier to create and discover communities.

What do we mean by "improving search"? The most fundamental improvement is providing a simple and standard way for users to explicitly and fully describe what it is they are looking for, and providing information about all types of resources to allow them to be quickly and accurately evaluated against such a search. Simplifying the creation and discovery of communities requires recognizing the concept of communities in existing file-sharing applications and making the concept concrete. Above and beyond this, we aim to lower the barriers to creating communities in order to disperse the benefits of file-sharing systems to as wide a variety of users as possible.

In concluding this section we justify the direction of our research with reference to the prior art. This is followed by a short discussion of what makes the challenge of improving search and simplifying community creation and discovery worthwhile in itself.

2.5.1 Weaknesses of Existing Search Methods

The biggest barrier to improving search is a lack of metadata. In order to move beyond the blunt instrument that is keyword search, file-sharing systems must provide more and better metadata. As we've seen, many systems rely on filenames. Aside from completely opaque identifiers like hashes (which are not meant to be human-readable anyway), filenames are pretty much the worst way one could imagine for this purpose. Placing the entire burden of expressing the metadata for a multitude of different formats on a single short unstructured string field is a recipe for bad search results.

This primitive form of metadata search has given way to richer methods. Kazaa and Limewire gave users of file-sharing applications their first chance at specifying more explicit searches. Simple keyword search was replaced with the ability to specify a context for that keyword: instead of simply looking for part of a song's title in the filename, they could now explicitly search only the title field. This is extremely important for improving search, and yet this ability is confined to a small group of selected formats. Outside of these formats, users are back to square one in terms of being able to explicitly and fully describe what they are looking for.

The flip side of giving users the ability to refine their searches is giving users the ability to make information about resources available. Without this, the extra information in the search will go to waste. This is another big problem. What is required is a shared, formal understanding of how these resources should be represented (ontology), but there are no shared ontologies for the majority of formats. This means it is not obvious what metadata should be made available or what fields are searchable. There is no way to extend existing applications to define these two sides of the coin. But without the ability to make these definitions, the quality of search for formats outside the mainstream will remain poor.

The lack of shared ontologies is part of a trend away from using structured metadata. Some examples are Limewire's query routing proposal, Routing Indices, Associative P2P, Neurogrid and FASD. Instead of using explicit metadata, these systems place more confidence in implicit indicators for directing searches and finding matching results. These systems rely on algorithms for determining how closely a given document matches some set of keywords ignoring the presence of perfectly good metadata.

Although these algorithms can be effective, we have seen that keyword search is a woefully blunt tool. Some of its weaknesses are distinguishing between different meanings of a term (Java the computer language vs. the island of Java), lack of precision (books about Winston Churchill vs. books by Winston Churchill or design patterns for schemas vs. a schema for expressing design patterns) or when one meaning of a term is obscured by another much more popular meaning. The need for explicit metadata search

becomes even more evident when searching for resources with a large body of associated metadata such as chemical molecules.

Another problem is introduced by systems which rely on the opinions of others to determine where searches should be routed or to define nodes with common interests. Examples are the routing of queries in Associative Peer-to-peer to nodes which possess common files or in Neurogrid based on user feedback. Possession of a file is one indication of community membership, but an explicit statement of that fact may be more reliable. Users with differing common interests may still share some files in common resulting in searches being processed by nodes which have little else of interest. Neurogrid's incorporation of feedback is another example of this: each user's opinions of which matches are relevant will differ slightly, yet these differing opinions are used to route each search. Without the ability to perform rich searches, there is no easy way to explicitly avoid whatever wrong turns may be introduced by the opinions of others.

To summarize, the major weaknesses in search which this thesis addresses are the general lack of structured metadata or lack of use of structured metadata. Clearly, what is required are shared ontologies for all types of files that might be shared and a simple way of extending file-sharing systems to incorporate such ontologies. These allow both publishers and searchers to know what structured metadata is expected for different formats and facilitates the ability to perform rich searches. Making explicit, structured and semantically meaningful metadata available enables users to explicitly and fully

describe what they are searching for and simplifies the task of quickly and accurately evaluating resources against such criteria.

2.5.2 The Difficulty of Creating and Discovering Communities

The fact of the matter is that the existing paradigm of systems with support for a few formats built in can not scale to the number of possible formats which users desire. How can a small central group of developers hope to keep up with demands for format support from the much larger group of users? One alternative, and the only currently viable one, is for those who want support for some new unpopular format to develop their own file-sharing application from scratch when all they really needed to add was the ability to search and publish metadata about the format.

Although there is nothing to stop people from sharing any format under the sun using a system like Kazaa or Gnutella, the lack of metadata support is a considerable barrier. So, as with the problem of search, we find that lack of metadata or lack of use of metadata is the fundamental problem with creating and discovering communities. Without the ability to search the metadata fields of new formats, there's no real way to differentiate the community from any other. These "community opaque" and "limited" systems provide no real way of defining a community.

A similar problem manifests itself in systems based on implicit indicators. For example, envision the task of forming a file-sharing community for peer-to-peer conference papers in a system based on implicit indicators. Individual users could make their conference

papers available, but there would be no way to direct your search explicitly to these documents. Users must enter some combination of keywords (e.g. if you were searching for conference papers on the topic of routing in P2P networks you might enter "p2p conference papers routing") and hope that the underlying search algorithm correctly infers its meaning. Although this may be effective in some cases, it would be much easier and less error prone to allow searches to be explicitly limited to the set of peer-to-peer conference papers. This requires that metadata be stored indicating which resources in the system are conference papers, and that this metadata be used when handling searches. This ability is missing in "community implicit" systems.

In community opaque, limited and implicit systems, the inability to differentiate communities translates into an inability to discover communities. If there is no way to define a community then how is a new user interested in that community supposed to find it? Although they may stumble across users with common interests, the process is ad hoc. There is no way to tell quickly and without ambiguity whether there is already a community dedicated to some new format. How do new users discover a new file-sharing system: right now the only real method is word of mouth.

A different challenge is presented by a system like Alpine in which users can explicitly group nodes into communities. Grouping nodes is only part of the solution though. This approach has some of the same drawbacks as opaque systems since each node can share a wide variety of different file types but there's no mechanism to narrow search to a specific type or to perform metadata searches. Additionally, there is no way to know

whether the nodes selected contain all files of a given type. This reduced search scope may reduce the quantity and quality of search results. Finally, this community exists only in the eye of the beholder. We refer to these systems as "unshared" systems since there is no way to join an existing community.

The system closest in nature to the problem we are trying to solve is JxtaSearch. Anyone can define a new community by creating a queryspace which defines what metadata fields are available in that community. For example, one could create a queryspace for peer-to-peer conference papers with fields like "conference name", "authoring institution", "year", etc. Users can then easily target their searches within the queryspace without resorting to keywords (e.g. peer-to-peer conference papers where authoring institution = Carleton University and year = 2002).

One of the major weaknesses of the JxtaSearch approach is the lack of a standard method for discovering queryspaces. Although one can easily imagine ways of finding queryspaces (e.g. a centralized queryspace index on the web), the lack of a well-defined method makes this process ad hoc and unreliable. The other weakness is that providers can supply results in any format, rather than using the same metadata structure as the query. This muddies the water about how searches are handled at each node. There is no clear cut requirement about what metadata should be indexed at each node, and the quality of results may vary from node to node.

The problem of providing a simple framework within which new communities can be created and discovered has not been solved by any of the above approaches. The major barrier to creating communities is a lack of metadata: to make each community effective there needs to be a way to search metadata and publish metadata. Simple lack of metadata, or reliance on implicit indicators, does not adequately meet this goal. The other side of the coin is allowing communities to be easily discovered. Without the ability to define a community, there is no discovery except in an ad hoc and inaccurate way. Unshared communities also undermine the ability to discover communities. The current approaches foster an environment of fragmented and isolated communities.

2.5.3 Why Search and Communities Matter

The utility of file-sharing systems is directly related to the ability to find what you are looking for. The crucial problem is figuring out ways for machines to understand what we mean when we search for something. The flip side is matching the meaning of our searches with available files in order to find good matches. As we have shown, existing file-sharing systems are ineffective at meeting this challenge for a wide variety of file formats. Providing a framework which solves this problem is a worthwhile problem because it will provide improved search for existing formats and enable applications that are simply not possible with the existing infrastructure. Some examples that we mention in this thesis are communities for sharing peer-to-peer conference papers, design patterns, chemical molecules or genetic annotations. These are all communities which would be difficult to search using existing applications.

Creating the new communities discussed above is difficult within existing applications because of inadequate search facilities, the use of implicit indicators, the lack of shared ontologies, etc. The only alternative may be creating customized programs to implement each community. This will often present an insurmountable barrier for those without the time or resources to spare on such a project which may lie outside their domain of expertise. This is why simplifying the creation of communities is so important: without the ability to easily create communities and provide good metadata search support, these applications might not come into existence.

There is no standard way of finding communities. There are only ad hoc methods like word of mouth. This landscape of fragmented and isolated communities perpetuates itself by making it more likely that users will reinvent the wheel and recreate already existing communities. Like any other network, file-sharing communities benefit from a network effect: if people can easily find a community which matches their interests it benefits everyone by enriching the variety of content and increasing the network's robustness. This is why a standard mechanism for discovering communities is so important.

Discovering communities goes hand in hand with describing communities. In order to discover communities we must decide what information defines a community. This information serves two purposes: firstly it gives users a basis for comparing different communities. Secondly, U-P2P uses this information to instantiate a working file-sharing

community. The following section explores the central concept of communities in more depth.

Chapter 3 The Concept of File-sharing Communities

3.1 *What is a Community: Community Schema Design*

Anywhere that people congregate you will witness the formation of communities. The internet is of course no exception. It is filled with groups of people who gather to share information based on common interest. Some examples of communities on the internet include mailing lists, newsgroups and websites such as Slashdot and Metafilter.

Similarly, users of peer-to-peer file-sharing programs can be thought of as communities united by the use of some application and a common interest in sharing files.

This thesis is concerned with *peer-to-peer file-sharing communities*. To simplify the discussion we will use the term *community* to refer to this specific type of community. Peer-to-peer file-sharing communities can be distinguished by a variety of criteria. In this section, we discuss these criteria and develop a more formal method of describing these communities in the form of an XML schema.

Some peer-to-peer file-sharing systems rely on centralized or hierarchical organization whereas others are completely decentralized; some systems guarantee anonymity while others provide no guarantees, and so on. Many of these criteria are determined by the protocol that is used - this governs the way members of the community (nodes) communicate. This choice may have consequences for anonymity, persistence, security and so on. In part then, a community is a group of users who have agreed on a standard method of communication.

As discussed earlier, some systems don't distinguish between file formats. Other systems offer the ability to choose a format and search its fields. When we say a community "supports" a certain format, this is what we mean: that the unique fields of that format can be searched individually or together. Traditionally, formats have been binary, but this is restrictive and assumes that all people sharing MP3 files (for example) are satisfied by the metadata available within an MP3 file. To encompass the variety of approaches and allow new ones, we can think of format as independent of the binary format. Among other applications, this would allow classical music MP3s to include an additional composer field. The same metadata definition can also be used for multiple communities: the classical music format could be used in separate Mahler and Mozart communities. Another part of the definition of a community then, is a group of users sharing and searching for files of a certain format for a certain purpose.

Combining the two aspects, a community is a tuple of different attributes. In its simplest form, a file-sharing community is a tuple of (format, protocol). The first describes what kind of files will be shared in a community and the second, how its members will communicate. Existing file-sharing applications bundle all of these aspects together. This makes them unnecessarily restrictive and makes it impossible to form an uncommon community (e.g. a community dedicated to the compositions of Gustav Mahler) using an arbitrary protocol. These aspects should be orthogonal.

This separation of concerns leads to more effective file-sharing communities. It allows protocol designers to focus on protocols and metadata designers to focus on metadata.

Most of all, it allows users of the system to make their own choices about how the aspects should work together. Users need a vocabulary to describe the community that suits their needs. The discussion of what constitutes a community is the first step towards this. We now present a tentative set of attributes - design criteria - for file-sharing systems.

3.2 Attributes of file-sharing communities

3.2.1 Name

In order to identify each unique community tuple it must have an identifier. Like a filename, a community's name supplies descriptive metadata that can be helpful for identifying a community's purpose. It is also used as a way to differentiate communities that are identical in other aspects: the combination of name and the other attributes of a community should be unique. This allows popular combinations of attributes to be reused for different purposes.

In the existing paradigm, this attribute can be thought of as the name of the application itself, e.g. "Gnutella", "Kazaa", etc. Like the other attributes we will discuss, it is not possible to specify "Name" independently with existing systems.

3.2.2 Format

To a certain extent any format can be accommodated in a system that provides simple keyword search (e.g. Gnutella). However, search by filename places a limit on the amount of information that can be encoded and searched. Another approach is allowing

only a small list of popular formats (e.g. Kazaa). FASD and Neurogrid are responses to this limitation and show that metadata search is valuable. These approaches approximate full metadata search using implicit indicators. JxtaSearch includes queryspaces which define the format of search requests, but not search results.

In contrast to these approaches, U-P2P requires that each community include a schema which describes the attributes of files that are shared. These schemas take the role of formats in other applications - instead of searching for files with a certain extension, users search for files conforming to a schema. This is a fundamental attribute of communities: a detailed description of the structure of the files shared by that community.

3.2.3 Protocol

The Protocol attribute determines how users of the community communicate. This includes not only aspects such as message format, but also how queries and search results are routed. This decision will have implications for network organization, structure, etc. Some examples of protocols are Gnutella or Freenet.

3.2.4 Display

The Display attribute determines the look and feel of each community. Look and feel encompasses the interface for publishing and searching as well as a way to view files and metadata. As with the other attributes, this is usually inseparable from the application itself. There is no real way for users to control how each community is displayed. In the

following section, we propose a simple standardized way for specifying this in a way that is independent of other attributes.

3.2.5 Other Parameters

Although the parameters Name, Format, Protocol and Display are the ones which we have implemented, the earlier discussion on other parameters in file-sharing systems suggests a few other possible parameters to be used for defining file-sharing communities.

Security

This determines whether communications between nodes are encrypted or not. Rather than choosing a specific algorithm, this is a qualitative attribute: users specify what level of security they require.

Anonymity

Anonymous systems make it difficult for observers to trace who is requesting a file or from where the file originated. This may or may not be a required property depending on the application so it should be configurable.

Deniability

Freenet illustrates the need for deniability in file-sharing communities. In Freenet, content is distributed throughout the network without any explicit control over its placement. This may result in a user hosting content which she is uncomfortable with, or

for which possession may have consequences. Rather than imposing restrictions on what content may be shared, files are encrypted making it difficult for operators or anyone for that matter to discover what information they are hosting. This allows operators to plausibly deny knowledge of hosted files.

Authentication

This refers to how access to the community is regulated. Some systems may choose to have no form of authentication allowing any node to participate. Others may use this attribute to specify that nodes must authenticate themselves against a database: this can be used to implement communities with restricted membership.

Security, anonymity, deniability and authentication are often attributes that are determined by the choice of protocol. The protocol designers have already made choices for these, restricting users to a specific configuration. Future protocol implementations could allow these aspects to be configurable or decoupled completely.

3.3 Community as Class/Community as Object

In this section, we develop an analogy between file-sharing communities as developed above and the concepts of Metaclass, Class and Object from the world of Object Oriented Programming (OOP). As we will see, this analogy is the basis of a simple method for community creation and discovery.

OO programming languages such as Smalltalk [Goldberg83] and ObjVlisp [Cointe87] are built on the concept of classes and objects. A class expresses methods and properties that are common to a set of objects. A class instantiates an object with these methods and properties. Instantiated objects are distinguished by the values of their properties. For example, an "Employee" class may have properties such as "Name" or "Employee ID" and methods such as "getManager" or "raiseSalary". Each individual Employee object implements these methods and is distinguished by Name or Employee ID.

Communities include a format. This expresses the properties that are common to the set of files which will be shared within that community. A community is similar to a class because it describes a class of files with common properties. The format can be used to instantiate files which are like objects. For example, an MP3 community may specify the properties "Artist" and "Title". Each individual MP3 within the community will have unique values for these properties and will be distinguished by them.

We now focus on the concept of classes: what exactly are they and what describes their structure? Some OO languages introduce the concept of metaclass which describes the commonalities of classes themselves. The handling of metaclasses varies. Cointe, in his paper "Metaclasses are First Class: the ObjVlisp Model" [Cointe87] notes: "In most common class-oriented languages, despite Krasner's uniformity principle [referring to the assertion that uniformity is an important part of Smalltalk's philosophy], a class is not a REAL object." For example, the Java "Class" object is not like other objects: "Class has no public constructor. Instead Class objects are constructed automatically by the Java

Virtual Machine" [Sun02]. Cointe notes that metaclasses in Smalltalk-80 are not true classes since they "cannot be defined explicitly by users".

In ObjVlisp, Cointe proposes the following relationship between classes and objects:

Figure 14: General structure of the ObjVlisp model [Cointe87]

Point is a class in the above diagram which is instantiated by Class. Similarly, the generic class Object, from which Point inherits, is also instantiated by Class. The self-referring arrow of Class indicates that it too is instantiated by Class. This expresses Cointe's goal of unification: "To suppress the gap between class and object, we propose a unification of the metaclass, class and object concepts. We claim that a class must be an object defined by a real class allowing greater clarity and expressive power."

Applying the same unification to file-sharing communities: a community must be a file defined by a real community.

Figure 15: Relationship between Communities and Files

In this model, there is no concept of inheritance so there is no analogy to the "Point" class in the above diagram. We also remove the inheritance arrows. The idea of inheritance for file-sharing communities is a topic for future consideration and is discussed in the conclusion.

What are the consequences of this unification? It means that communities, like classes, are defined by another community. This community is unique because it also describes itself. As Cointe points out: "we have to guarantee that the instance variables specified by Class match the corresponding values also held by Class, as its own instance." We call this self-referential community "The Root Community" (it can also be called, confusingly, the Community Community). This suggests the other important corollary of "classes are objects": communities are files. A community is meant for sharing files and

the Root Community is not different in this respect. The Root Community is a community for sharing communities.

This unification of the concept of communities and files leads to a simple and elegant method of discovering communities. We have already described some of the attributes that define file-sharing communities. These attributes describe the format of communities. This definition is both part of the Root Community and a description of the Root Community (and all communities). Communities are not an exception to the primary application of file discovery and file-sharing. *Community discovery reduces to the problem of file discovery.*

3.4 Conclusion

In this section we have developed the concept of a community as a concrete object with a defined set of attributes. This can be formalized as an XML Schema. XML Schema is a simple and standardized way to list the attributes required to fully describe a file-sharing community. A discussion of XML Schema is presented in Appendix A. Here is a simplified schema for communities:

```
<schema>
  <element name="community">
    <complexType>
      <sequence>
        <element name="name" type="xsd:string"/>
        <element name="protocol" type="protocolTypes"/>
        <element name="schema" type="xsd:anyURI"/>
        <element name="display" type="xsd:anyURI"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

Figure 16: Simplified Schema for Communities

Note that the “format” attribute is expressed here as “schema”. This is a logical extension of using XML Schema to express the structure of communities. Each community must include an XML Schema which describes the expected structure of the files being shared in the community.

As we’ve seen, we can unify the way in which communities and files are treated, in the same way that classes and objects are unified in Cointe’s elegant ObjVlisp implementation. This means that communities themselves can be shared as regular files in a “Root Community”. The Root Community must include a schema: the XML Schema shown above. This schema describes the structure of all communities as well as the Root Community itself.

In the following sections we show how the concept of communities can be made concrete in the form of the Universal Peer-to-peer Framework.

Chapter 4 Design of U-P2P

4.1 Introduction

In the preceding chapter, we developed the concept of a community as a concrete object with a defined set of attributes. These attributes include name, protocol, format, etc. The format attribute is a schema describing the structure of files in the community. By analogy we also reached the conclusion that communities themselves can be treated like files.

The important benefits of this approach are:

- Every community includes metadata which is available for search.
- A standard format for communities allows anyone to specify their own community and the structure of the files it will share.
- Communities are shared in a *real community* that anyone can use to publish their own community and to find new communities. Searchers can search metadata for communities like any other file.

Universal Peer-to-Peer (U-P2P) is a novel application framework that implements these ideas. U-P2P exploits metadata to improve search and to simplify the creation and discovery of peer-to-peer file-sharing communities. This chapter discusses the design of U-P2P and how it achieves these goals.

4.2 Overview

Figure 17: Communication between U-P2P clients

U-P2P is an interactive application that allows them to create, join and participate in a variety of peer-to-peer file-sharing communities. To facilitate this, U-P2P has a user interface which allows users to search for communities and for files within communities. In order to participate in the underlying peer-to-peer network an interface is required to convert requests to the appropriate format. In addition to these two components, each U-P2P peer maintains a searchable database of local files to accommodate queries from other peers.

The different components are referred to as Web Adapter, Network Adapter and Repository in further discussion. The interaction between these components is illustrated above. These form U-P2P's skeleton, but a community is required to breathe life and

personality into this framework. We discuss the practical details of communities in the next sections along with discussions of each component.

4.3 Schemas and the Choice of XML Schema

A schema is a structured description of the attributes an object is expected to have. This is a prerequisite for metadata search since it defines what metadata is available for objects in a community. There are various formats for expressing schemas such as DTD (Document Type Definition) [W3C00], XML Schema [W3C01] and RDF (Resource Description Framework) [W3C03]. U-P2P uses XML Schema for a variety of reasons. XML Schema was preferred over RDF because of its simpler syntax and better support for data types. RDF is especially good at expressing information about relations between objects, but its syntax adds unnecessary complexity to the generally simple property lists that U-P2P uses.

The structure of XML documents has traditionally been specified using Document Type Definitions (DTD). XML Schema was chosen over DTD because DTD is not an XML format. DTD documents are less intuitive and readable than XML Schema. The fact that XML Schema documents are written in XML is appealing because there is a large community of XML developers and accordingly tools and source code for working with XML are widely available. The availability of transformation tools enables the automated creation of search forms and the like from any XML Schema document. U-P2P schema documents are written using standard ASCII text. This is useful because it makes it easy for users to read them or create new schema. Simple XML Schema

documents are not difficult for someone familiar with HTML or XML to decipher. This lowers the barriers to users who wish to repurpose existing schemas or create new ones. Taking advantage of the more advanced concepts in XML Schema may pose a challenge to many users. To address this we have also put some effort into devising a tool for creating schemas. This web application, called SchemaTool [Arthorne02b], presents a “wizard” interface in which users are guided in a step-by-step fashion through the creation of an XML Schema document. With this tool, even novice users can quickly create schemas for use with U-P2P. The importance of schemas in U-P2P is explained in the following sections. An overview of the XML Schema language can be found in Appendix A.

4.4 Improved Metadata

U-P2P requires metadata for each and every file which improves the quality of search for all types of files. Schemas are used to represent the “Format” attribute of U-P2P communities which describe what metadata is required. A community for sharing MP3s might have the following simple schema:

```
<xsd:schema>
  <xsd:element name="mp3">
    <xsd:complexType>
      <xsd:all>
        <xsd:element name="file" minOccurs="0" type="xsd:anyURI"/>
        <xsd:element name="title" minOccurs="0" type="xsd:string"/>
        <xsd:element name="artist" minOccurs="0" type="xsd:string"/>
      </xsd:all>
      <xsd:attribute name="title" type="xsd:string"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Figure 18: A schema for the "mp3-nap" community

And here is an example of an XML file that follows this schema:

```
<mp3>
  <file>http://127.0.0.1/mp3s/billie-jean.mp3</file>
  <title>Billie Jean</title>
  <artist>Michael Jackson</artist>
</mp3>
```

Figure 19: An XML file described by the "mp3-nap" schema

The tags in the XML file must correspond to those defined in the schema. This highlights one important difference of U-P2P communities with respect to communities in existing applications: the files being shared are always XML files. Rather than trading different binary formats (mp3, avi, etc.), U-P2P users trade "wrappers" which contain metadata defined by the community as well as embedded links to binary files.

The use of XML wrappers has many advantages over the sharing of binary formats. XML provides a standard way of representing the file type and the "embedded" metadata. This means that one piece of standard code (an XML parser) can handle any format. There are no "second class" formats with only support for filename matching. From one perspective, U-P2P supports only one format - XML files. But from another perspective this is all that is required since XML files are a mould into which any format can be cast. This is clearly an improvement on most existing systems which rely on implicit metadata or fail to provide metadata at all.

Embedded links are designated in the schema by the use of a specific data type. This allows U-P2P to detect and download them automatically - in general both the embedded

links and the metadata are downloaded together. The embedded link can be a binary file such as the MP3 embedded in the MP3 metadata example above. But really it could be any type of file, binary or otherwise. There is also no limit on the number of embedded links. For example, we could extend the MP3 schema above to include the title of the album the song came from and an embedded link to the album's cover art.

On the other side of the coin, the same binary file could be embedded in many different metadata descriptions. Rather than a one size fits all "mp3 community" with a rigid set of metadata, U-P2P allows many different descriptions of the same object to co-exist. For example, a community of music collectors might want to add more fields, such as format and date of release, to the simple mp3 schema above.

Keeping track of metadata externally to the binary also amounts to a separation of concerns. The design of a binary format is application specific (e.g. better compression for MP3s) and may be needlessly complicated by the requirement for a separate but embedded metadata specification. Factoring out metadata from the binary allows the metadata to be flexibly extended or altered independently of changes in the binary. The standardized metadata format also means that the file-sharing application need not be rewritten to handle changes in formats or entirely new formats. New metadata requirements can be accommodated by dropping in a new schema.

4.5 Web Adapter Design: Improved Search

The two sides of the metadata coin are creating and searching. Without a mechanism for creating metadata, the existence of a metadata specification is moot since there won't be any. On the other side of the coin, making the best use of the available metadata requires the ability to search using the available fields. Peer-to-peer applications must provide interfaces for creating and searching that are aware of the structure of a variety of different file formats. Traditionally though, only search interfaces have been provided. Additionally, rich search interfaces (i.e. ones which allow each field to be specified) are limited to the popular formats. Other formats are generally supported with basic filename matching and there is no simple way to add new formats to the system. This section illustrates how the metadata available within U-P2P is used to improve search for all formats. These functions are encapsulated by U-P2P's WebAdapter component.

The use of XML wrappers removes the requirement of building in knowledge of how to parse metadata from each format. Similarly, the use of XML schema removes the requirement of designing a custom interface for each format. The technique used to accomplish this is described below.

The U-P2P client is a web application. The traditional model for P2P applications is for users to interact with a single application which offers a user interface to the P2P network. The model used by U-P2P is to use a standard web browser as the user interface. Users interact with their U-P2P client through their browsers. Beyond this point the model is identical.

Figure 20: U-P2P vs. existing user interaction model

The decision to use the browser as a user-interface flows directly from the choice of XML and XML Schema. XML and XML Schema are easily transformed into HTML since HTML is an XML-based language itself. There are many tools available for working with XML documents. One of the most important of these is XSLT (Extensible Stylesheet Language Transformations), which is expressly designed for transforming

XML documents into other XML documents. [W3C99] XSLT is the most important element in providing generic interface support for new communities.

XML Schemas describe the expected metadata fields and their data types. To the human eye it is not difficult to create a simple form for searching or creating based on an XML schema.

```
<xsd:schema>
  <xsd:element name="mp3">
    <xsd:complexType>
      <xsd:all>
        <xsd:element name="file" minOccurs="0" type="xsd:anyURI"/>
        <xsd:element name="title" minOccurs="0" type="xsd:string"/>
        <xsd:element name="artist" minOccurs="0" type="xsd:string"/>
      </xsd:all>
      <xsd:attribute name="title" type="xsd:string"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Figure 21: A Simple Schema

Maps quite obviously into something like:

File:	<input type="text"/>
Title:	<input type="text"/>
Artist:	<input type="text"/>
<input type="button" value="Search"/>	

Figure 22: A Simple Form

This can easily be represented using an HTML form. HTML is an XML format and XSLT can be used to transform an XML Schema into HTML. The operation of XSLT is discussed in the detailed design section below. Briefly, it requires a stylesheet written in XSL (Extensible Stylesheet Language). A stylesheet is the static representation of how the tags of one XML document can be reformatted into another XML format. For example a stylesheet directive might map a schema element into a form element.

The schema and the HTML form encode the same information, but the latter can be used as part of a web application. It is conceivable that the same XML Schema could be transformed into a GUI or into a text-based interface (although the transformation to HTML is much simpler). However, the important point is that generating an interface that is *aware* of the fields of each custom type does not require special handling for each type. The interface can be generated in a consistent and programmatic way from the XML schema.

Figure 23: Role of XSLT in U-P2P

The diagram above illustrates the role of XSLT in U-P2P. Aside from its role in creating the search and create interfaces, XSLT is also used to display the shared files. The XML file can be transformed into an HTML representation and then rendered in the browser.

Figure 24: Generation of create interface from a XML schema

Figure 25: Generation of display interface from a XML instance

Default stylesheets are provided to accomplish these translations. But U-P2P allows users to specify their own stylesheets for each of the operations: create, search and view. Stylesheets are an important component of U-P2P communities since they determine each community's look and feel. This explains the "display" attribute mentioned earlier

in the enumeration of the attributes of file-sharing communities. These are embedded links to the stylesheets used for each function. When a user downloads a community, the stylesheets are also downloaded. If they are not specified, U-P2P will use the default stylesheets.

The interface design presented above is an example of how U-P2P exploits metadata to improve search in file-sharing communities. Each file-sharing community is associated with a very important piece of metadata: the schema. When used along with a default stylesheet, the schema contains all the information required to render a user interface for creating and searching within the community. The additional effort of writing a custom stylesheet allows the user even more control over the look and feel of the interface. Rather than hacking an existing peer-to-peer application, or trying to get a new format accepted, all that is required is a schema.

4.6 Simplified Community Creation and Discovery

Solving the community creation and discovery problem requires a flexible method to create new communities, with full metadata support, as well as a way to discover them. We have introduced the concept of a community as a tuple of attributes which can be expressed in a structured form. One of these attributes, format, is a schema describing the structure of the files that are traded by the community. The MP3 schema and MP3 XML file shown above illustrate this relationship. We have already seen how the use of a schema improves search. But how are community tuples defined?

As discussed earlier in the analogy with Cointe's unification of classes and objects, the concept of communities and files can be unified. In OOP, it has been shown that treating classes as objects provides increased flexibility and expressive power. Applying the same concept to communities yields a simple and elegant method for discovering communities. Unifying the concept of communities and files implies that communities must also be described by a schema. Communities are described by the schema of the Root Community. The Root Community is unique because its schema also describes itself. A first attempt at such a schema was presented in the preceding chapter.

Here is the expanded schema for communities used in the Beta 1 version of U-P2P. It differs from the schema presented at the conclusion of the previous section in that the "display" attribute has been expanded (the three "location" attributes used to render different aspects of the user interface) and a few extra fields have been added for descriptive purposes (description, keywords, category).

```
<schema>
  <element name="community">
    <complexType>
      <sequence>
        <element name="name" type="xsd:string"/>
        <element name="description" type="xsd:string"/>
        <element name="keywords" type="xsd:string"/>
        <element name="category" type="xsd:string"/>
        <element name="protocol" type="protocolTypes"/>
        <element name="schema" type="xsd:anyURI"/>
        <element name="displayLocation" type="xsd:anyURI"/>
        <element name="createLocation" type="xsd:anyURI"/>
        <element name="searchLocation" type="xsd:anyURI"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

Figure 26: A Schema for Communities (community.xsd)

This corresponds with the fields we saw earlier in the description of the MP3 community. As with the MP3 community it is a mix of metadata and embedded links. The embedded links point to the community's schema (format) and stylesheets (display) used to create a user interface. Fields like name and description provide human-meaningful information making the community easier to find. Protocol determines how peers will communicate.

Armed with a schema for communities it is a small step to expressing the Root Community. Here is the XML file describing the root community:

```
<community title="Root Community">
  <name>community</name>
  <category>U-P2P</category>
  <keywords>U-P2P, root community</keywords>
  <description>The root community of U-P2P</description>
  <schemaLocation>community.xsd</schemaLocation>
  <displayLocation>community-display.xsl</displayLocation>
  <searchLocation>community-search.html</searchLocation>
  <createLocation>community-create.html</createLocation>
  <protocol>central-db</protocol>
</community>
```

Figure 27: The Root Community (community.xml)

To reiterate, the role of the schema is twofold here:

1. The Root Community (community.xml) is described by the community schema (community.xsd). In this sense it is the same as other communities since they are all described by the community schema.
2. The Root Community is the one community that also includes the community schema. The schema of a community describes the structure of the files shared within it. The Root Community is used for sharing communities and thus includes a pointer to the community schema.

U-P2P allows communities to be concretely expressed in the form of XML files. These files, whose structure is dictated by the community schema, must include a pointer to the schema for the community. Each community's schema describes the metadata associated with the files in that community. This addresses the weaknesses inherent in community opaque, limited, implicit and unshared systems since shared communities with rich, structured metadata support can be created by simply editing an XML file. Since searches and results must conform to the same schema, the weakness of community partial systems is also addressed.

Here is an example of how a MP3-sharing community using a Napster-like architecture might be expressed:

```
<community>
  <name>mp3-nap</name>
  <description>
    This is an mp3 sharing community using a napster-like protocol.
  </description>
  <keywords>mp3 napster music</keywords>
  <schema>mp3-nap.xsd</schema>
  <displayLocation>mp3-nap-display.xsl</stylesheet>
  <createLocation>mp3-nap-create.xsl</stylesheet>
  <searchLocation>mp3-nap-display.xsl</stylesheet>
  <protocol>napster</protocol>
</community>
```

Figure 28: Community file for the “mp3-nap” community

The first step in creating a U-P2P community is to create such a file. This file, like any other file, is then published to a community for others to discover. The difference is that it will be published to the Root Community, the one community for which U-P2P includes a description and the community used for creating and discovering other communities.

U-P2P's schema for communities and the existence of a well-known Root Community simplify the creation and discovery of communities. Creating a new community, with metadata support, requires specifying a few parameters in an XML file along with a schema. Discovering a new community requires entering a few search parameters into the Root Community's stylesheet-generated search interface.

The next section leaves behind the discussion of schemas and stylesheets to return to the discussion of the core U-P2P software components.

4.7 Repository Design

Each U-P2P client includes a repository for storing objects that have been created or downloaded from elsewhere. The repository presents a standard interface for retrieving objects and embedded links. It handles requests from the user via the interface as well as from other peers via the adapter. Users interact with the repository by creating new objects or by browsing the local collection. When a user downloads a copy of an object from another peer it is stored in the repository. On the other side of the transaction, the remote repository must accommodate requests to search and to download objects.

At a more general level, the repository consists of data structures representing communities. The objects shared within each community are referred to as "resources". Resources can contain embedded links referred to as attachments. These characteristics lend themselves to a hierarchical design. The repository uses the concept of collections to represent communities. Collections contain entries for each local resource that belongs

to the community. These entries are indexed by a resource ID. The terms collection and resource are borrowed from the underlying database implementation discussed in detail below.

Figure 29: Repository structure

Resource IDs have a few specific functions. They are globally unique IDs generated by applying the MD5 hashing algorithm [Rivest92] to the resource's XML representation. The chances of two documents hashing to the same identifier are small. One application of this is identifying multiple registrations of the same resource that may have different filenames. This can be used to identify multiple sources advertising the same resource, increasing the possibility that it will be successfully retrieved. Another application is validating the contents of a downloaded resource by comparing the advertised MD5 signature to a computed signature.

Figure 30: Resource IDs in U-P2P

Resource IDs also act as indirect file references, much as a C pointer provides an indirect reference to the contents of some memory location. Resources are stored at each peer in the form of an XML file (in addition to the repository). Existing applications force users to explicitly designate a root directory or directories beneath which all files and directories are exposed to other peers. U-P2P allows files to be located anywhere on the peer's hard drive and only explicitly published resources are visible to other peers. Resources are identified by a URL which refers to the resource ID, hiding information about the user's hard drive. The repository maintains the mapping between resource identifiers and local files and their attachments.

4.7.1 Interactions

The other components of U-P2P (web adapter and network adapter) interact with the repository in the following ways:

- **Upload/Publish:** When users create new resources, the web adapter must interact with the repository to enter the resource into the community's collection. If the new resource is a community, a new collection must be created.
- **Downloading:** From the perspective of the repository, this operation is much like uploading: a new resource or community is created. The difference is that the remote peer supplies information about the new resource via the network adapter.
- **Browsing:** Users can browse their own collection of resources by community. This is just a matter of querying the repository to list the contents of a given collection. The returned list is then post-processed into HTML for the user to browse.

The user initiates downloads which are an interaction with a remote peer. Search is an example of a repository interaction initiated by a remote peer.. Registration is an interaction that occurs in a central server deployment between a remote peer and the repository of the server. In a distributed deployment, this step would not be required since each peer keeps track of its own files.

- **Registration:** The central server operates as a registry in which peers advertise the resources they have available. This requires a mechanism to associate each resource with the peers that host it. After publishing a resource to its own repository, the peer must also register its information with the central server. The resource's metadata is also published on the central server to facilitate search. The embedded links remain on the peer. When peers leave the network they should also unregister their resources.
- **Search:** In a completely decentralized implementation each peer must support this operation. In a hybrid peer-to-peer deployment, searches are conducted on the central server. Peers connect to the repository on the central server. The query will return a set of resources matching the query, each accompanied by a list of peers who have registered the resource.

Figure 31: Interactions with repository

The repository interface is standardized allowing different underlying implementations to be plugged in easily. This allows various possible implementations such as a file-based repository, a RDBMS or an XML database. The current implementation relies on a combination of data structures and an XML database. Another advantage of this standardized interface is the ability to swap in newer and faster XML database implementations in the future.

4.7.2 XML Database

The need to search through metadata motivated the decision to use some kind of database in U-P2P. One possibility of course would have been full-text indexing, but this technique does not preserve the structure of XML documents. Another possibility that we considered was simply using the file system for storing resources and using some other tool to perform queries on the collection of files. This introduces the additional complexity of managing the document collection. Unlike existing database implementations this approach is neither standardized nor fully featured.

The Magenta software package [Magenta01] served as a database in the initial implementation of U-P2P. Magenta preserves the structure of U-P2P objects and allows queries of resources using the concept of filters. Each individual resource is instantiated as an object with properties corresponding to its metadata. Queries are written in a modified form of CMIP (Common Management Information Protocol) - originally meant for network management applications. This is a somewhat standardized interface but not one usually associated with peer-to-peer applications.

The Magenta approach had various weaknesses that led to us abandoning it in the second iteration of U-P2P. Using software intended for agent and network management applications as a database was not the best fit, and a lack of experience in database development meant that it was not a robust implementation for that purpose.

Instantiating an object for every resource in the system was not scalable and the mapping of XML to objects and attributes was imperfect - for example there was no support for multiple instances of the same attribute or nested attributes. Using the CMIP protocol for querying XML documents was also an imperfect fit - other query languages exist that are tailored specifically to XML.

These weaknesses led us to the decision to use a standardized database component. We favoured an XML database - one especially designed for storing and querying XML documents - over an RDBMS because U-P2P is based on XML documents. Storing XML in an RDBMS requires the fields and structure of an XML document to be mapped to tables and columns in an RDBMS. Unfortunately, coming up with such a mapping may be quite complex, difficult to automate and result in the loss of structural metadata. A traditional RDBMS requires highly structured data and accommodating the variability found in XML documents is the subject of much research (e.g. [Florescu99] [Shanmugasundaram99]).

The repository is central to the design of U-P2P and indeed, all peer-to-peer file-sharing must deal with the problem of storing and searching for resources. An XML database allows simple storage of XML resources and the ability to handle powerful queries over

all of the available metadata. This gives U-P2P the flexibility to handle a multitude of different resource types. This is an important ingredient that simplifies the design, use and sharing of new communities. Support for rich queries allows searchers within communities to take advantage of the structured metadata to accurately describe what they are looking for. This facility also helps to simplify the discovery of new communities: as far as the repository is concerned this is yet another query.

4.8 Network Adapter Design

Existing peer-to-peer systems come in a wide variety of network deployments from hybrid to complete decentralization. Although U-P2P currently only supports a hybrid deployment, the aim is to eventually allow communities to be designed with a choice of protocol. This is reflected by the presence of a protocol field in the proposed schema for communities. Allowing a diverse set of deployments motivates the existence of the Network Adapter component.

The Network Adapter component provides an abstract interface to an underlying peer-to-peer network. The network adapter encapsulates the functionality of communicating with other nodes (i.e. other peers or a central server in a hybrid P2P model). This functionality includes:

- routing search requests and handling the results
- handling incoming search requests
- retrieving files from remote peers

- handling incoming retrieve requests
- other protocol-specific communication (e.g. query forwarding, peer discovery, etc.)

The other components of U-P2P, the web adapter and repository, interact with the network adapter using a standard API. This is the foundation of U-P2P's protocol independence. By encapsulating this functionality and providing a standard API, it is possible to plug in different network adapter implementations without affecting other components. This means that the ability to create, share and discover new communities is portable to a variety of different peer-to-peer architectures.

The current implementation of U-P2P includes a network adapter for a hybrid peer-to-peer deployment. Another important deployment would be a purely decentralized network adapter that allowed U-P2P to interact with Gnutella nodes, but this was not implemented. Network adapters could also be envisioned for other deployments such as Freenet or a DHT implementation. The choice of network adapter should be driven by the community description, but currently, since there is only one network adapter implementation, the choice defaults to the hybrid peer-to-peer model.

4.8.1 Interactions

User actions like searching, sharing and downloading correspond directly to operations on the underlying P2P network. Interactions with the network adapter are generally invoked by the interface. The main interactions are:

- Publish - After a user has created a resource and it has been entered into the local repository it is also published to the P2P network. Many P2P deployments require this ability although some do not: for example in Gnutella networks, peers only publish resources locally. This method is required for a hybrid P2P deployment in which a record of the new resource and its location must be entered into the central database. Other P2P architectures require a publish operation since resources may not be stored at the node which created them (e.g. Freenet or DHT implementations).
- Remove - This operation is the complement of publishing. Users can choose to remove resources from the P2P network so that they are no longer shared with others. In the hybrid P2P model this notifies the central server to remove the peer from the list of locations for the unshared resource.
- Retrieve - Downloading a resource from a remote peer is also the responsibility of the network adapter. After the user has received search results and selected a resource to download, the web adapter invokes the network adapter to retrieve the resource from the peer network. In our hybrid implementation, the network adapter selects a download location and then retrieves the resource using standard HTTP. This aspect of the network adapter is a useful abstraction barrier since retrieval may not always be HTTP-based. For example, files are retrieved from Freenet according to their key rather than their location.

- Download - This adapter function is invoked from a remote peer. It is somewhat transparent to the remote peer: downloads look like simple HTTP gets. There is no message-based communication, instead a peer simply issues a request for a URL such as `http://myhost/up2p/community/stamps/resource-id`. Requests for URLs are redirected to a component which uses the mappings defined in the repository to resolve the actual XML file corresponding to the resource-id. The document is what is returned to the remote peer.
- Search - After a user has entered a query using the web adapter, the parameters are passed to the network adapter's search interface. The network adapter is responsible for any formatting and protocol details required by the P2P network, for sending the query to the appropriate node and receiving the results to pass back to the interface. In our implementation this requires dispatching the parameters to the server formatted according to U-P2P's search request message format.

As discussed earlier in the repository section, the ability to service remote searches is currently only implemented on the central server. In a completely decentralized system, each node would support this operation.

Figure 32: Interactions with Network Adapter

4.8.2 Protocol

U-P2P defines APIs for all of the major interactions described above. Although the XML database implementation does have support for remote configuration via the XML-RPC protocol this level of API was too database specific. Instead a protocol was developed at the level of U-P2P primitives: publish request and response and search request and response. The format of the protocol messages is described in more detail in the next chapter, but the basic protocol is described here.

4.8.2.1.1 Publishing a resource

The network adapter's publish method sends a RegisterRequest message to the central server containing the resource ID and the location of the resource. The central server is assumed to be at a well-known address that can be configured by editing a configuration file. The location passed in the RegisterRequest is an indirect URL as described in the

Download interaction above (e.g. `http://peer/up2p/community/<community-name>/<resource-id>`) that can be used to download the resource, but does not give any other information about where the resource might be located on the peer's file system.

The central server will reply with a `RegisterResponse`. There are two possible responses. One response type indicates that this is the first time that the resource has been registered. In this case, the publishing peer must reply to the server with the resource's metadata. This metadata is entered into the central repository and the peer's message is acknowledged. Registration of the resource's full metadata is important because it allows the repository to handle searches from other clients without talking to the original publisher each time. The other response type indicates that the repository already has a record for this resource and that a new location for it has been added. This reduces the messaging overhead for already published resources.

Figure 33: Publish message flow

4.8.2.1.2 Searching for a resource

The network adapter's search method sends a `SearchRequest` message to the central server. The query is encoded as a set of parameter and value pairs - for example an MP3 community query might include pairs like (artist, Michael Jackson) which would match songs where the artist field included the term "Michael Jackson". A list of matching resources along with their known locations is returned to the peer within a `SearchResponse` message. This response message is decoded by the web adapter and then used to display results to the user. The peer caches the locations of each resource. If the user decides to retrieve one of the matching resources, the location cache is used to connect directly to another peer hosting the resource. If a retrieve attempt from the first location fails, attempts to download are made in sequential order from the list of locations.

Figure 34: Search message flow

The use of a standard API for interacting with the peer-to-peer “layer” of the application makes it possible for U-P2P to easily support different deployments and protocols at that level. Existing file-sharing communities support a single such protocol, each of which come with their own set of pros and cons. Although the application may have support for a format a designer is interested in, they must still contend with the possibility that the P2P architecture does not meet their needs. The interface and P2P layer are coupled by residing together within the same monolithic application. U-P2P is a framework for creating applications in which the P2P layer can be chosen independently. This greatly simplifies the task of creating and using new file-sharing communities.

4.9 Conclusion

Creating and sharing new communities with support for metadata search requires metadata. The first requirement is a community description. The structure of this description is dictated by the standard schema for communities contained in the “Root Community”. The fields of this schema have been discussed above and describe the parameters of each community “tuple” – e.g. name, format, protocol, etc. The second important requirement relates to the “format” element of the tuple: a schema which describes the structure of the format. U-P2P uses XML wrappers (which may contain embedded links) and the structure of these wrappers is dictated by a community’s schema. Some examples of these important requirements have been presented in this section.

Once specified, this information sits atop a metadata aware infrastructure capable of them into actual functioning communities. There are three main elements of this infrastructure: Web Adapter, Repository and Network Adapter. The Web Adapter component applies transformations to this metadata to render the user interface which is presented to the user via a standard web browser. The Repository is a flexible database capable of organizing and storing a wide variety of different formats. This is why an XML database was chosen for this component: it is able to accommodate XML documents which are less rigid than the definitions used by a standard RDBMS. Finally, the Network Adapter mediates access to the network of peers. This acts as an abstraction barrier allowing U-P2P to adapt to different peer network implementations.

These components are part of the U-P2P framework which improves search and simplifies community creation and discovery. Search is improved since all communities include searchable metadata. Creation and discovery of communities is simplified by expressing communities in a standard XML format that can be shared in a community like any other file.

The next section discusses these three components in more detail and the variety of technologies used to implement each one.

Chapter 5 Detailed Design

I must acknowledge Neal Arthorne for his substantial contribution to this chapter.

Although I am responsible for the first implementation of U-P2P (version 0.2), the discussion below is based on Neal's U-P2P version 1.0 implementation. This is a re-implementation of the ideas in a more modular and maintainable fashion. Additionally, functions which Magenta served in version 0.2 are now served by more robust components: an XML Database and an HTTP based protocol for inter-peer communication. Diagrams from the U-P2P Developer's Guide have been used with Neal's permission.

5.1 Interface and Web Adapter

Many P2P applications are standalone software applications that combine a custom user interface together with P2P functionality. Users start up a binary executable which offers an interface for communicating with other peers. U-P2P is a web application. Installing the U-P2P client requires installation of a webserver along with the U-P2P code which runs within the context of the webserver. Users interact with the U-P2P client through their browsers. U-P2P code running within the context of the webserver takes care of the P2P functionality.

As described earlier this was initially motivated by the ease with which XML documents can be transformed into HTML. There are four major functions available to users via the interface:

- create new resources or communities
- search for resources or communities
- view resources
- browse resources in the local repository

Of these, the first three rely on the ability to transform XML documents into HTML.

U-P2P is a web application. Web applications run within the context of a web server. U-P2P is a combination of Java Servlets and Java Server Pages (JSP) [Sun03, Sun03b]. These technologies must run within a Servlet container. The current implementation of U-P2P runs within the Tomcat 4 servlet container [Apache03]. A full explanation of these technologies is beyond the scope of this thesis, but a capsule summary is presented below.

Java Servlets are Java programs which run within the context of a web server. Users interact with servlets using standard HTTP methods.

Java Server Pages are a hybrid of Java and HTML. Rather than writing a Java program to output HTML, JSPs allow Java functionality to be embedded within HTML. These are compiled into servlets and executed on the web server. This is in contrast to JavaScript which is a similar mix of HTML and Java, but executes in the context of the web browser.

A Servlet container is a web server especially designed to run servlets. It implements standard HTTP server functionality. HTTP requests are decoded and made available via a standardized Java API to servlets running within the container.

Tomcat proved to be a good fit for U-P2P for two reasons:

- Tomcat is a Java application and can run on multiple platforms.
- Tomcat is an open-source application in keeping with the philosophy of the U-P2P project

Additionally, choosing Java meant we could take advantage of several powerful Java packages. Most importantly:

- Xerces: XML parsers [Apache02]
- Xalan: XSLT stylesheet processors [Apache03b]
- eXist: XML database [Meier03]
- log4j: powerful logging tool [Apache03c]
- JUnit: unit test framework [ObjectMentor02]

There are some drawbacks with using Tomcat. It requires the pre-installation of the Java Runtime Environment on the client computer. Over time this may become more ubiquitous. Limewire is one application which has this dependency and its popularity is a promising sign. Another issue may be a reluctance to run a web server, although, under

the covers, popular file-sharing systems like Gnutella and Kazaa already run web servers for serving download requests.

Jxta was also considered as a possible development environment for U-P2P. But due to the complexity of the package and the difficulty of integration we decided against it.

5.1.1 Generating pages with XSL

Major user functions map to different Java Server Pages. These pages include a mix of HTML and Java code to handle dynamic content. As mentioned previously three of the major user functions rely on transformation of XML documents into HTML. This is accomplished using XSL stylesheets. XSL relies on the interpretation of XML documents as trees of elements. Here is a brief description of how XSL operates on XML documents from the W3C website:

"A transformation expressed in XSLT describes rules for transforming a source tree into a result tree. The transformation is achieved by associating patterns with templates. A pattern is matched against elements in the source tree. A template is instantiated to create part of the result tree." [W3C99] Patterns are specified as XPath expressions [W3C99b]. XPath is based on the tree-like nature of XML documents. It provides a systematic method for identifying branches and leaves of that tree using the path notation similar to that used for navigating file systems. These are associated with templates that can contain any type of data (for example HTML formatting tags) mixed in with XSL

variables. When a transformation is performed, this template is applied to each matching pattern to produce the "result tree". Here is an example of such a transformation.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="stamps">
    <xsd:complexType>
      <xsd:all>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="description" type="xsd:string"/>
        <xsd:element name="country" type="xsd:string"/>
        <xsd:element name="dateOfIssue" type="xsd:date"/>
        <xsd:element minOccurs="0" name="lastDayOfSale" type="xsd:date"/>
        <xsd:element minOccurs="0" name="denomination" type="xsd:string"/>
        <xsd:element minOccurs="0" name="design" type="xsd:string"/>
        <xsd:element minOccurs="0" name="dimensions" type="xsd:string"/>
        <xsd:element minOccurs="0" name="gumType" type="xsd:string"/>
        <xsd:element minOccurs="0" name="illustration" type="xsd:string"/>
        <xsd:element minOccurs="0" name="paperType" type="xsd:string"/>
        <xsd:element minOccurs="0" name="perforations" type="xsd:string"/>
        <xsd:element minOccurs="0" name="photography" type="xsd:string"/>
        <xsd:element minOccurs="0" name="printer" type="xsd:string"/>
        <xsd:element minOccurs="0" name="printingProcess" type="xsd:string"/>
        <xsd:element minOccurs="0" name="quantity" type="xsd:string"/>
        <xsd:element minOccurs="0" name="tagging" type="xsd:string"/>
        <xsd:element minOccurs="0" name="layout" type="xsd:string"/>
        <xsd:element minOccurs="0" name="picture" type="xsd:anyURI"/>
      </xsd:all>
      <xsd:attribute name="title" type="xsd:string"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Figure 35: Stamps.xsd - Schema for a Stamp Community

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xml:lang="en"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html"/>

  <!-- Parameter passed to this stylesheet -->
  <xsl:param name="communityName"/>

  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template name="SchemaTemplate" match="*[local-name()='schema']">
    <h3>Create a Resource</h3>
    <form action="create" method="post">
      <table border="1" cellpadding="5" cellspacing="0">
        <tr><th style="border-bottom: thin solid black;">Property</th><th style="border-bottom:
        thin solid black;">Value</th></tr>
        <tr><td>Filename</td><td><input type="text" name="up2p:filename"/> Filename for the
        created object.</td></tr>
        <tr><td>Title</td><td><input type="text" name="up2p:title"/> The title as seen in U-
        P2P.</td></tr>
        <xsl:for-each select="descendant::*[local-name()='element' and count(./child:*) = 0]">
          <xsl:call-template name="ElementTemplate"/>
        </xsl:for-each>
      </table>
    </form>
  </xsl:template>
```

```

</table>
<p><input type="hidden" name="up2p:community"><xsl:attribute name="value"><xsl:value-of
select="$communityName"/></xsl:attribute></input>
<input type="submit" value="Submit"/></p>
</form>
</xsl:template>

<xsl:template name="ElementTemplate">
<tr><td><xsl:value-of select="attribute::name"/></td><td>
  <xsl:choose>
    <!-- string -->
    <xsl:when test="(contains(@type, ':') and substring-after(@type, ':') = 'string') or
@type = 'string'"><input type="text" size="20"><xsl:attribute name="name"><xsl:value-of
select="concat(..../@name, '/', @name)"/></xsl:attribute></input></xsl:when>

    <!-- date -->
    <xsl:when test="(contains(@type, ':') and substring-after(@type, ':') = 'date') or
@type = 'date'"><input type="text" size="20"><xsl:attribute name="name"><xsl:value-of
select="concat(..../@name, '/', @name)"/></xsl:attribute></input> Date in YYYY-MM-DD
format.</xsl:when>
    <!-- anyURI is for file upload -->
    <xsl:when test="(contains(@type, ':') and substring-after(@type, ':') = 'anyURI') or
@type = 'anyURI'"><input type="file" size="20"><xsl:attribute name="name"><xsl:value-of
select="concat(..../@name, '/', @name)"/></xsl:attribute></input></xsl:when>

    <!-- default input is a text box -->
    <xsl:otherwise><input type="text" size="20"><xsl:attribute name="name"><xsl:value-of
select="concat(..../@name, '/', @name)"/></xsl:attribute></input></xsl:otherwise>
  </xsl:choose></td></tr>
</xsl:template>

</xsl:stylesheet>

```

Figure 36: default-resource-create.xsl: Default Stylesheet for Create form

This transformation must be performed by software. Recent versions of Microsoft's Internet Explorer browser can be used to perform this transformation on the client side by embedding a link to the stylesheet within the XML document to be transformed. U-P2P performs the transformation on the server side using the Xalan Java package. This functionality is encapsulated within a U-P2P function and is called from within the JSP page "create.jsp".

Create a Resource

Property	Value
Filename	<input type="text"/> Filename for the created object.
Title	<input type="text"/> The title as seen in U-P2P
name	<input type="text"/>
description	<input type="text"/>
country	<input type="text"/>
dateOfIssue	<input type="text"/> Date in YYYY-MM-DD format
lastDayOfSale	<input type="text"/> Date in YYYY-MM-DD format
denomination	<input type="text"/>
design	<input type="text"/>
dimensions	<input type="text"/>
gumType	<input type="text"/>
Illustration	<input type="text"/>
paperType	<input type="text"/>

Figure 37: HTML output of *create.jsp* rendered in browser

If you have U-P2P installed you can see the results of the transformation at:

<http://localhost:8080/up2p/create.jsp>. This JSP parses the parameters from the URL, performs the HTML formatting and inserts the result of the XSL transformation. This is all executed within the servlet container. The result is pure HTML which is returned to

the browser. The output of create.jsp for the stamps community is shown below as rendered by the browser:

The same process is used to render the Search and View interfaces as well. In the case of search (search.jsp), the search stylesheet is applied to the community's schema. For viewing (view.jsp), a display stylesheet is applied to the resource's XML. The fourth function of the U-P2P UI, browsing local files within a community, is also handled by view.jsp.

5.1.2 Major Software Components

Aside from the JSPs, the interface is constructed of the WebAdapter, CreateServlet, UploadServlet and SearchServlet components.

WebAdapter

This is one of the three major components of U-P2P. Its purpose is to provide an abstraction barrier between the JSPs and servlets implementing the interface and the other two components (Repository and NetworkAdapter). It provides interfaces which correspond to the major U-P2P operations (create, search, view, browse). It also includes much of the initialization code for U-P2P.

Figure 38: Class diagram for the WebAdapter and its implementation [Arthorne02d]

U-P2P also relies on a number of servlets to encapsulate common tasks that are too unwieldy to include within JSPs. There are three which belong logically to the Web Adapter component: `CreateServlet`, `UploadServlet` and `SearchServlet`. The JSP page takes care of gathering the parameters and these are then passed to the appropriate servlet which then takes care of interacting with the Repository and Network Adapter to get the task accomplished.

CreateServlet

This component contains the code which reads the fields entered via the rendered create form and constructs an XML resource. Creating an XML file via an HTML form requires that each input value be labelled as an XPath expression. For example, an HTML form for submitting a resource to a community for sharing stamps called "stamps" might have an entry like the following:

```
<input type="text" name="stamps/dateOfIssue">
```

Which tells U-P2P that the value entered in this field corresponds to a specific part of the XML document. This conversion of XPath expression and value pairs into an XML document is accomplished by the CreateServlet.

UploadServlet

This component contains the code which formally shares the new resource. The use of the word "upload" comes from the fact that in a hybrid P2P deployment, the resource metadata must be uploaded to the central server in order for others to discover it.

New resources can be shared in two ways

1. Construct the resource using the form interface
2. Register the resource as an XML file ("Upload")

In implementation, the first technique relies on the second. Once the XML file has been created by the CreateServlet it is published by the UploadServlet. The UploadServlet

allows users to create an XML resource using some other technique (e.g. in their favourite editor) and skip the form interface. The UploadServlet validates the XML, calculates the resource ID and records the new resource in the repository. In the hybrid P2P model, it also uploads the resource metadata to the central server. This latter concept would not apply to all P2P architectures (e.g. Gnutella which uses local publishing).

Figure 39: Role of Servlets and JSPs in U-P2P [Arthorne02d]

SearchServlet

Like the CreateServlet, the SearchServlet must also deal with (XPath expression, value) pairs submitted from the search form. Instead of creating an XML file, the SearchServlet is responsible for creating a query which is then submitted to the WebAdapter's

"searchNetwork" method. The WebAdapter then submits the query to the Network Adapter. The results are cached within the servlet's context and the user's browser is redirected to another JSP (displayResults.jsp). This JSP uses the information stored in the servlet context to render the search results so that the user can choose an appropriate match to download.

5.2 Repository Detailed Design

Internally, the repository consists of an XML database and data structures used to maintain the association between resource IDs, resources and attachments. The XML DB is used primarily for storing and querying metadata. The task of associating resource ids to actual files and their attachments is handled by a set of linked tables.

A standard URL format is used to identify U-P2P resources. These take the following form:

`http://<host:port>/up2p/community/<community-name>/<resource-id>`

For example:

`http://localhost:8080/up2p/community/community/d702a046555cd5d964
2dd6d180a02c06`

This URL identifies a resource that resides within the "community" collection (i.e. this resource is actually a community definition). This URL is used to retrieve a resource

from a remote peer. A URL relative to the resource URL is used to identify attachments - for example:

```
http://24.102.27.174:3300/up2p/community/community/d702a046555cd5  
d9642dd6d180a02c06/community.xsd
```

All URLs underneath `http://host:port/up2p/community` are handled by the `DownloadServlet` component. This component belongs logically to the network adapter since it responds to remote download requests, but its operation relies on the repository. `DownloadServlet` consults the repository to resolve the reference and returns the requested document from wherever it might be located on the peer's hard drive.

In order to perform this service the repository maintains lookup tables for each community mapping IDs to actual files. The repository must also maintain the association between resources and attachments: each resource ID is also associated with a table of attachments.

There is a `resourceMap` for each community that lists the locally stored resources belonging to that community. This `resourceMap` can be looked up by the community's name in the `communityMap`. For each resource a few pieces of information must be tracked: the resource ID, the location of the corresponding XML file on the local hard drive and a pointer to the table of attachments.

Figure 40: Internal data structure for resolving resource IDs

This implementation differs slightly when a peer is operating as a central server (all peers can act as clients and servers simultaneously). The repository on the central server is only concerned with resources and their locations. This removes the need for an **attachmentMap**. Since individual peers only store information about local resources, there is only one location for each resource. In the hybrid peer-to-peer model, the central server must keep track of multiple locations for each resource. Instead of an attachment map, each resource is associated with a list of locations that can be used by searchers to retrieve the resource directly from another peer.

Figure 41: Internal data structures for locating files on central server

5.2.1 XML Database

eXist [Meier03] was chosen for the XML database implementation. eXist was ideal for U-P2P for the following reasons:

- open source project - free to incorporate it within U-P2P
- written in Java - simple to integrate with the rest of U-P2P
- compliant with the XMLdb API standard - standard API with good feature set

The XMLdb API is a project of the XML Database API working group of the XML:DB initiative. The goals of the API are described on its website: "This API should be vendor neutral to support use with the largest array of databases possible. The API is also intended to be implementable in multiple languages though it does assume the implementation language is object oriented." [XMLDB03] The API currently includes specifications for managing collections, querying and updating resources. The first of these APIs is used for creating or removing per-community collections and the querying API is required to support the ability to perform rich metadata queries.

The query service relies on XML Path Language (XPath) expressions. As noted in the discussion of XSLT, XPath expressions are a precise way to identify specific branches and leaves of an XML document tree. Filtering is accomplished by the use of predicates: expressions that can be applied to select the desired results from the result set. XPath was the obvious choice because it is especially designed for filtering XML documents and because it is the expected query format for XML:DB compliant databases. More information about XPath can be found in [Harold02].

A simple XPath query to check whether a particular stamp had "Canada" in its name would be: `contains(stamps/name, "Canada")`.

In general, the user does not see XPath queries when he performs a search (although there is a mechanism for submitting raw XPath queries). The output of the search form is transformed into an XPath query. The details of how the query and responses are

communicated between peers and the central server are discussed in the Network Adapter detailed design.

Figure 42: Role of DownloadServlet and DatabaseViewer components

Figure 43: Class diagram for Repository and its implementation. [Arthorne02d]

The DatabaseViewer servlet is another useful component of the repository. This servlet services requests for the URL: `http://host:port/up2p/database.xml` and returns a listing of the resources stored on the local peer.

5.3 Network Adapter Detailed Design

The NetworkAdapter is a standard interface that can accommodate a variety of network deployments. It includes the following methods that can be accessed by the WebAdapter class (i.e. the interface):

- `publish(resource, communityName, resourceFile)`
- `remove(communityName, resourceId)`
- `retrieve(searchResponse)`
- `searchNetwork(community, searchQuery, minResults, maxResults, maxTimeout)`

The NetworkAdapter also includes a handle to access the local repository in order to service incoming queries. Implementing adapters for different networks requires implementing this interface. U-P2P currently includes one such adapter referred to as the "Generic Network Adapter". This implements the hybrid P2P (Napster) model.

In a hybrid P2P model, the methods above map to the following tasks:

- `publish` - Notify the central server of the new resource including the associated metadata.

- remove - Notify the central server that the current peer is no longer a valid location for a given resource ID.
- retrieve - Connect to a remote peer and download one of the search results from the most recent query.
- searchNetwork - Issue query to central server and receive the resource IDs and locations of matching resources.

Figure 44: Class diagram for PeerNetworkAdapter and the generic implementation. [Arthorne02d]

The hybrid P2P model requires one peer to also act as a central server. The server runs simultaneously with the client within the Tomcat servlet container. The central server leverages much of the same repository code as the client. It does not however include the other client components. No user interface is required except for debugging purposes since users interact with the server via the client interface. There is no network adapter component per se since the server does not initiate transactions on other peers. Incoming transactions are serviced by servlets. Central server functions are invoked via the HTTP post method.

A simple protocol is used to exchange information. Request messages are sent via HTTP post method to the central server. The host and port of the server are defined in the peer's configuration. The request is serviced by the appropriate servlet and the server returns the result in the HTTP response. Request messages are terminated on the central server by two servlets: RegisterServlet and SearchServlet. The latter is the counterpart of the SearchServlet running on the peer and only takes care of handling incoming queries. The RegisterServlet takes care of the publish and remove functions. In order to process incoming queries, these servlets include references to the central server repository and location table (described in the repository section).

5.3.1 Message Types

There are four message types in the example NetworkAdapter implementation. SearchRequest, SearchResponse, RegisterRequest and RegisterResponse. Each request is given an identifier which is used in the corresponding response message. The flow of

search messages is straightforward: the query is sent in the SearchRequest and matching resource IDs and locations are returned in the response. Register messages follow a slightly more complicated flow in order to achieve a reduction in overhead.

Registration requires the transmission of the entire XML metadata of the resource (but not the attachments) to the central server. This could be a large amount of data. But there is no need to retransmit the information if a record of the resource already exists on the central server. To avoid this, peers initially register the resource ID. The server response indicates whether the resource ID already exists in the central repository. If not, the client must then initiate a full registration including the complete resource metadata. Register request and response messages also accommodate de-registration: in this case only the resource ID is passed along with an additional attribute indicating that the location should be removed from the central server's tables.

The format of each message type is shown below [Arthorne02c]

5.3.1.1.1 Search Request

```
<up2pMessage>
  <searchRequest id="xyz" community="community" maxResults="10">
    <searchQuery>
      <term path="community/keywords" value="stamp"/>
      <term path="community/description" value="stamp"/>
    </searchQuery>
  </searchRequest>
</up2pMessage>
```

Figure 45: Example SearchRequest message

Rather than sending a single XPath query, a collection of XPath expression and value pairs is sent. This is composed into a single XPath query on the central server and issued to the XML database.

5.3.1.1.2 Search Response

Two matching resources, each with two possible download locations:

```
<up2pMessage>
  <searchResponse id="xyz" community="community" resultSetSize="2">
    <searchResult title="Canadian Stamps Community"
      id="494db2fc050198e895380d17a04beebd">
      <resource
location="http://192.168.1.100:8080/up2p/community/community/494db2fc050198e895380d17a04b
eebd"/>
      <resource
location="http://192.168.1.101:8080/up2p/community/community/494db2fc050198e895380d17a04b
eebd"/>
    </searchResult>
    <searchResult title="Global Stamp Community"
      id="d09414f0f0d56f96e0c41c9333b36eea">
      <resource
location="http://192.168.1.100:8080/up2p/community/community/d09414f0f0d56f96e0c41c9333b3
6eea"/>
      <resource
location="http://192.168.1.101:8080/up2p/community/community/d09414f0f0d56f96e0c41c9333b3
6eea"/>
    </searchResult>
  </searchResponse>
</up2pMessage>
```

Figure 46: Example SearchResponse message with results

Example for no results found:

```
<up2pMessage>
  <searchResponse id="xyz" community="community" resultSetSize="0"/>
</up2pMessage>
```

Figure 47: Example SearchResponse with no results

5.3.1.1.3 Register Request

```
<up2pMessage>
  <registerRequest id="xyz">
    <resource community="community" id="494db2fc050198e895380d17a04beebd"
location="http://192.168.1.100:8080/up2p/community/community/494db2fc050198e895380d17a04b
eebd"/>
    <resource community="community" id="d09414f0f0d56f96e0c41c9333b36eea"
location="http://192.168.1.100:8080/up2p/community/community/d09414f0f0d56f96e0c41c9333b3
6eea"/>
  </registerRequest>
</up2pMessage>
```

Figure 48: Example RegisterRequest message (initial registration)

When a peer starts up, it must register all of its existing resources with the central server. All resources can be registered in a single message (example shown above). These resources must also be de-registered when the peer is shutdown. Here is an example of a request to remove two resources:

```
<up2pMessage>
  <registerRequest id="xyz" remove="true">
    <resource community="community" id="494db2fc050198e895380d17a04beebd"
location="http://192.168.1.100:8080/up2p/community/community/494db2fc050198e895380d17a04b
eebd"/>
    <resource community="community" id="d09414f0f0d56f96e0c41c9333b36eea"
location="http://192.168.1.100:8080/up2p/community/community/d09414f0f0d56f96e0c41c9333b3
6eea"/>
  </registerRequest>
</up2pMessage>
```

Figure 49: Example RegisterRequest message (removing resources)

The central server will request a full registration for new resource IDs. Here is an example of full registration of one resource:

```

<up2pMessage>
  <registerRequest id="xyz">
    <resource community="community" id="494db2fc050198e895380d17a04beebd"
location="http://192.168.1.100:8080/up2p/community/community/494db2fc050198e895380d17a04b
eebd"/>
    <community title="Canadian Stamps Community">
      <name>canStamps</name>
      <keywords>stamps, Canada</keywords>
      <description>Canadian stamps and related materials.</description>
      <displayLocation>file://test/stamps/stamps-display.xsl</displayLocation>
      <searchLocation>file://test/stamps/stamps-search.html</searchLocation>
      <createLocation>file://test/stamps/stamps-create.html</createLocation>
      <schemaLocation>file://test/stamps/stamps.xsd</schemaLocation>
    </community>
  </resource>
</registerRequest>
</up2pMessage>

```

Figure 50: Example RegisterRequest message (full registration)

5.3.1.1.4 Register Response

The register response indicates the number of resources that were registered as well as resources that must be fully registered. Here is an example of the response expected for two unknown resources:

```

<up2pMessage>
  <registerResponse id="xyz" registered="0">
    <resource community="community" id="494db2fc050198e895380d17a04beebd"
location="http://192.168.1.100:8080/up2p/community/community/494db2fc050198e895380d17a04b
eebd"/>
    <resource community="community" id="d09414f0f0d56f96e0c41c9333b366ea"
location="http://192.168.1.100:8080/up2p/community/community/d09414f0f0d56f96e0c41c9333b3
66ea"/>
  </registerResponse>
</up2pMessage>

```

Figure 51: Example RegisterResponse message (unknown resources)

Example of response for two known (previously published by another peer) resources:

```

<up2pMessage>
  <registerResponse id="xyz" registered="2"/>
</up2pMessage>

```

Figure 52: Example RegisterResponse message (known resources)

De-register responses include a registered attribute of zero, but include no <resource> elements:

```
<up2pMessage>  
  <registerResponse id="xyz" registered="0"/>  
</up2pMessage>
```

Figure 53: Example RegisterResponse message (removing/deregistering resources)

5.4 Conclusions

U-P2P is built using a variety of existing packages. XML, XSLT and XML Schema are important, but these are just text files without support from underlying software such as Java Servlets, JSPs, the eXist XML database and the code developed for U-P2P to coordinate all of these. This section described some of these choices and gave an overview of the structure of the U-P2P software. The following section presents our results regarding the performance of this software and its ability to handle large communities.

Chapter 6 Evaluation and Validation of U-P2P

6.1 Introduction

Evaluating the performance of U-P2P is an important part of understanding how useful it can be in the real world. As with all P2P applications, the first question is whether U-P2P is scalable. There are many dimensions in which scalability can be evaluated: one important metric is the time it takes to search and publish files. Ideally it should be quick to perform both of these operations even with many users and files. How do these times vary as the number of files in the system increases? A variation of the same question for U-P2P is how do these times vary as the number of communities increase?

The choice of network deployment affects performance as well as what needs to be measured. Although we only evaluate performance in a hybrid deployment, it is interesting to contrast how hybrid and decentralized deployments handle searching and publishing. In a hybrid system, such as the current implementation of U-P2P, where searches are serviced at the central server, search times are essentially a measure of the time it takes to query the central database. In a fully decentralized system, the time to search is a function of the number of nodes, the search algorithm employed, as well as the size of each node's collection. Searches may take a long time to return if many nodes must be queried. In a decentralized system, publishing a file is a relatively quick operation since it requires only an update of the peer's local database. In the centralized model, where all files are indexed centrally, the time to add a new record will be affected

by the central database size. These considerations motivate the first set of performance experiments set out below.

The publish and search operations are actually composed of multiple sub-tasks such as querying the database on the central server and communicating the result back to the client via the browser. The total time for search is the sum of these sub-tasks, however each sub-task may vary differently with collection size. Decomposing these operations and examining how each sub-task scales can reveal bottlenecks and hint at areas of improvement.

Finally, U-P2P communities raise some interesting performance issues. We perform experiments to demonstrate how the presence of multiple communities affects performance. When all communities have a common central server, is the performance within each community independent or is it influenced by the existence of other communities? As we will see, communities influence performance within a single server as well as the case when each community resides on its own server. These experiments illustrate how communities can improve the scalability of the overall system. Although each community/server will have fixed overhead costs, partitioning the set of available files increases the overall scalability of the system without reducing the ability to discover files.

6.2 Challenges: Community Creation and Automation

The experiments were performed using a client and server running on a single computer. Running on the same node removes the effect of network latency and emphasizes the delays incurred by the software and the XML database. There were two major challenges to measuring U-P2P's performance characteristics. The first challenge was having multiple communities each with a large collection of objects. The second was automating the process of publishing and searching for objects.

Community Creation

Creating a community and populating it with files manually was too time-consuming so we sought out archives containing large numbers of XML objects. One example of such a repository can be found at the Berkeley Drosophila Genome Project (BDGP).

[Rubin96] This project maintains a database of annotations of different parts of the Drosophila (fruit fly) genome. The project makes these annotations available in different forms including as a large XML file. This file was split such that each annotation was stored in an individual file. Another pre-requisite for U-P2P communities is a valid XML schema. As with many existing XML repositories, the BDGP repository is described using the older DTD technology.

Transferring a large collection of previously defined XML documents that are defined by a DTD is a tedious process. The experience may be of value to those who might want to import such existing collections into U-P2P. It is much easier if the collection already validates with an existing XML Schema.

Translation of DTDs to XML Schema as required by U-P2P was accomplished using the DTD2XS [Lumrix03] package. DTD2XS is a Java program from the Liebig University Institute of Medical Informatics. It runs as a Java applet and is accessed via a browser on the local machine. The user provides a path to an existing DTD and DTD2XS outputs the corresponding XML Schema to the browser. As described above, it was used during the performance evaluation to allow large collections of XML documents described only by a DTD, to be imported into U-P2P. U-P2P does not support the DTD format since it is not specified in XML and because XML Schema documents are more easily manipulated and transformed.

Since DTDs and XML Schemas are both valid descriptions of XML document structure, most of the structures in one language can be mapped to the other. The translations done by DTD2XS were workable but often required some small changes after the fact. This was not due to problems with DTD2XS however but to the original DTD and XML files. U-P2P requires that a document validate against the community's schema before it can be published. There were three main problems with the DTD and resulting schema that blocked validation:

- namespaces - the U-P2P validator does not currently handle namespaces, this means tags which depend on namespaces must be flattened (i.e. go:term should become term) and external element types need to be added to the schema

- missing elements - some of the collections of XML documents included tags not defined in the DTD. This required adding a definition for the element into the schema.
- restrictions - some of the XML documents did not include instances of required elements. This required relaxing the restriction on the number of occurrences of an element in the schema.

With these changes the bulk of the XML documents could be validated and published to U-P2P.

Although the BDGP collection was not intended for a peer-to-peer environment, it does present one possible peer-to-peer application. The BDGP is part of FlyBase [Flybase03], a distributed project involving scientists at many different institutions. Each annotation contains a collection of information that can be enriched as information is gathered. With U-P2P each institution (or even each scientist) could maintain their own local database of annotations. By searching on a specific gene's name these multiple annotations could be discovered, viewed and dynamically reconciled, rather than requiring curation of each annotation at a central repository.

In addition to the fruit fly information, three other collections of XML documents were created in a similar fashion:

- DBLP [Ley03] – references to Computer Science related papers, theses and proceedings
- EDGAR [MediaOrg01] – filings with the US Securities and Exchange Commission (SEC)
- Gene Ontology [GeneOntology00] – database of gene-related concepts

These are all pure XML communities – objects do not contain embedded objects. These communities were used in some of the following performance measurements.

Automation

Manually publishing and searching for large numbers of files was not practical, so these operations were automated. An XML-RPC [Winer99] interface was designed for the publish and search functions. XML-RPC is a simple way to allow remote procedure calls of methods within web applications. Command-line XML-RPC clients were written to take advantage of these capabilities. These allow operations to be performed more rapidly than would be possible using the web-based interface. These command-line tools can also be used to script more complex experiments.

6.3 Operations: Publish

The time to publish was measured as repository size increases. One thousand files were published to the Fruitfly Community from a single client. Measurements are shown in Figure 54. This graph shows that publishing time increases in a linear relation with the

number of files created. Figure 55 displays the cumulative time vs. number of files and shows that the relation is not completely linear.

Figure 54: Time to Publish a File vs. Number of Files

Figure 55: Cumulative Publish Time

A polynomial of degree four can be fit closely to this curve, but the two high order terms are small, confirming the linear relationship between the time to add a new file and the existing number of files up to a thousand files. Publishing a thousand files takes about eight minutes but as the curve is extrapolated the high order terms become more dominant: publishing two thousand files is estimated to take about 35 minutes with an identical setup.

The factors which contribute to this growth can be further explored by breaking down the publish operation into steps and profiling how the time taken by each of these steps increases as the number of files increases.

The publish operation can be broken down into the following steps from the client's point of view:

1. Process the resource file in the client. This includes the time taken to parse the XML file and updating the client's data structures.
2. Store resource in client database. The resource is entered into the client's XML database.
3. Publish to server. Information about the resource is relayed to the central server which registers it in the central database.

Figure 56: Operations involved in publishing a file

The following graphs illustrate how the timing of these steps varies as the number of files published increases.

Figure 57: Time to Process Resource in Client vs. Number of Files

Step 1, processing the resource in client, seems to converge to a constant value. Over the span of one thousand files, the trend is negative acting to reduce the total publish time as the number of files increases. In any case, this step does not contribute to a linear increase in publish time.

Steps 2 and 3 (Figure 58 and Figure 59) account for most of the trend in the total publishing time. The characteristic for storing the resource in the client database is a property of the database implementation. Interestingly, the time to add a file to the client database has a slope about one third that of the time required to publish the file to the central server. What accounts for the difference in the trend between updating the client database and updating the server database?

Figure 58: Time to Store Resource in Client DB vs. Number of Files

Figure 59: Time to Publish from Client to Server vs. Number of Files

In Chapter 4, we discussed the message flow involved in publishing. The diagram is reproduced here:

Figure 60: Publish message flow

The initial register request requires a lookup in the central server database to verify that the resource id has not previously been published. If the id is not found, a response is dispatched to the client requesting a full registration. The second register request invokes another database lookup to verify the resource id and then the resource is entered into the database. The net effect is that a publish operation consists of two resource lookups and one resource store. The trend for these operations versus the number of files previously published is shown below.

Figure 61: Time for Server Resource Lookups vs. Number of Files

These operations display a similar linear trend to that found in the client. Together these operations account for the variable portion of the time required to publish from the client to the server.

Figure 62: Time to Store in Server Database vs. Number of Files

Figure 63: Comparison of Total Publish Time vs. Time for Server Operations

The constant factor represented by messaging overhead and other server operations can be seen in Figure 63. The times for the server operations are combined and compared against the time to publish to the central server as seen by the client. The two trend lines are almost parallel indicating that the other operations involved are independent of database size and essentially constant.

Profiling indicates that database operations are the most costly aspect of the publish operation. They dominate timing and account for the linear increase. The main components are the resource store on the client and server plus the two resource lookups on the server. Reducing the number of database operations or optimizing them will yield the most performance gain and increase the system's scalability.

6.4 Operations: Search

With a constant number of files, search times remain constant. This is illustrated by Figure 64 which displays the time taken by a set of one thousand searches within the Fruitfly community containing one thousand files. The fluctuation in values around an average of about 450 ms may be a function of differing size of query terms, number of results and background computation on the computer.

A more interesting question is whether search times vary with the number of files stored at a server. As we've seen, the time required to publish a file is dependent on the number of files which have previously been published. The time to publish a file is made up of the times required to perform its sub-operations: lookups and stores. The search

operation also requires a lookup so it makes sense to assume that it too will increase with the number of files.

Figure 64: Search time with a Fixed Number of Files

To test this assumption, a set of fifty-eight queries was executed after one hundred files were added to the community. The same queries were repeated every one hundred files, up to twelve hundred files. This yields the curve shown in Figure 65.

As with publishing, the curve seems roughly linear. This characteristic is a property of the underlying eXist database implementation.

Figure 65: Search time vs. Number of Files

The search operation (Figure 66), unlike publishing does not involve a database access on the client. A search request message is dispatched to the central server which executes the query on the database. This is the only database access required. This search yields a set of resource ids. Download sites for these resources are stored in server data structures. The information about resource ids and download sites is then returned to the client.

Figure 66: Operations involved in searching for a file

As shown in Figure 67 the only time consuming operation is the database access. This one step accounts for most of the variability in search time. The lookups in the server's data structures take a negligible amount of time even for a large database.

Figure 67: Server DB Search Time vs. Total Search Time

The performance and scalability of search, as with publishing is largely a function of the underlying database implementation. Improvements must therefore come from a change in that layer.

6.5 Community Effect

Up till now we've focussed on performance as a single community grows in size. Does the presence of other communities have an effect on performance? At each node resources are stored in a common repository, regardless of community, so we might

expect there to be some coupling in performance. On the other hand, if each community's resources are maintained in separate tables then the size of one community should not affect the performance of another. Two experiments were conducted to assess whether there is a "community effect" on U-P2P performance.

In Figure 68 we plot the time required to publish 250 files to a Fruitfly community in the presence of an increasing number of other communities. The first set of 250 values is the time to publish to a community when it is the first and only existing community. Each subsequent set of 250 values is measured in the presence of one more community than the previous up to a maximum of three other communities. In each case the existing communities are made up of approximately 250 files. We utilize the example communities mentioned earlier (GeneOntology, EDGAR, DBLP).

Figure 68: Publish Times With Communities Present

Figure 69: Effect of Other Communities on Time to Add a File

Figure 69 displays the data for each set of values superimposed. Both graphs illustrate clearly that there is little or no coupling between communities stored on the same node. The publishing performance is essentially identical regardless of the presence of resources published in other communities. Publishing time is a function of the number of files published *in each community*. As seen earlier, publish timing is a sum of the times taken to perform database operations on both client and server. These times will vary independently since the client database and the server database will differ in size. The experiments above simplify matters because we assume that the server and client are initially empty.

Figure 70 displays a similar experiment designed to measure how search performance varies in the presence of existing communities.

Figure 70: Effect of Other Communities on Search Times

As we saw before, search time is a function of the number of resources in a community. To measure the effect of communities, we conduct searches as the number of resources in the community increases, while increasing the number of existing communities from zero to three. To produce the series shown here we measure the time for each of ten searches, increasing the community size by 50 files between each set of ten, up to a total of 250. Each set of measurements is conducted in the presence of different numbers of communities, each containing roughly 250 files. Search times, like publish times, vary

with the size of the community but are independent of the number and size of other communities.

6.6 Multiple Central Servers

Figure 71: Multiple Server Deployment

The U-P2P deployment discussed to this point allows only a single central server which contains the Root Community and all other communities. It is a small change to allow each community to reside on its own "central server" (for example, a community creator could host its server on his or her own machine). Two fields must be added to the community description, host and port, which allow the client to contact the community's central server. Publish and search operations look at this information rather than a system-wide field in order to contact the community's central server. In this scenario, the Root Community resides on one server and contains all information about existing

communities. The client retrieves the community info including the location of its central server. The client then contacts the community's central server for publishing and searching. We constructed a small three server network consisting of computers with the following specs:

Table 2: Server Properties

Server	Processor	Processor Speed	Operating System
1	Pentium 4	1.8 GHz	Windows 2000
2	Pentium II	250 MHz	Linux (Redhat 7)
3	Celeron	1 GHz	Windows XP

Server 1 is the machine which was used for all the single server experiments and is the most powerful. Server 2 is the weakest in terms of CPU power. Each server acts as the central server for a single community. Server 1 also hosts the Root Community.

Figure 72 shows the results of publishing 750 files, with 250 being published into each community in the order Server 1, Server 2, Server 3. Figure 73 presents for comparison the performance numbers as if there were a single community and all 750 files were published to one of the servers.

Figure 72: Time to Publish Files With Three Servers

Figure 73: Publish Times on Different Servers

The graphs show that partitioning the set of files into communities provides average performance better than Server 2 and Server 3. The superior performance of Server 1 of course dominates the results, but even if it were to act as a single central server with one community, the average performance is only slightly better.

These characteristics show that multiple central servers for multiple communities allow the system as a whole to host more resources while maintaining a constant level of performance.

6.7 Conclusions

In general these preliminary performance results indicate that there are improvements to be made. Assuming that users will not tolerate publishing times greater than 5 seconds or so, a Pentium 4 server could host a community containing a couple of thousand files. The ability to host multiple communities per server and multiple servers increases this capacity, but we aim to eventually support communities much larger than this. The results above suggest a number of performance improvements.

The cost of database operations suggests removing one of the lookups involved in publishing. The initial registration phase requires a lookup which should be cached if a full registration is required. This would eliminate the need to do a second lookup during full registration. Another option is allowing publish operations to be batched and performed in the background. Batching would only provide a constant improvement because each resource must still be entered into the database individually incurring the

associated costs. Performing operations in the background allows the user to tolerate higher waiting times.

The database implementation is an obvious target for performance improvement since most of the performance characteristics of U-P2P are determined by database operations. During the course of the thesis, the database implementation was upgraded from eXist 0.8 to eXist 0.9. Figure 74 shows publish times versus number of files using eXist 0.8. The eXist 0.9 implementation was used for all of the previous results included in this section. Comparing with Figure 54 we see an improvement of four times (250 ms with eXist 0.9 vs. 1000 ms for eXist 0.8), and multiple orders of magnitude for the initial publish operation (300 ms with eXist 0.9 vs. 100 *seconds* with eXist 0.8).

Figure 74: Time to Publish a File (Using eXist 0.8)

Future improvements in eXist may yield similar performance improvements for U-P2P. Since eXist is an open source project there is also the potential for eXist to be improved based on our results. Another more drastic option would be to use the system's file system instead of an XML database. This has the drawback of making rich search more difficult.

The results above highlight one of the important contributions of U-P2P. **Communities improve the scalability of peer-to-peer systems.**

In a central server deployment, each community's publish and search times scale independently as if they were the sole community operating on the node. This allows each community to scale to the same size as a single monolithic community without degrading performance. The total capacity of the server for a given performance level is thus multiplied by the number of communities.

Allowing multiple servers acts as another multiplier for the peer-to-peer system's capacity. Each "central server" continues to support multiple communities. The total capacity of the U-P2P system is thus multiplied by the number of servers without performance degradation. In practice of course, the differing capabilities and bandwidth available to each server may act to reduce this multiplier.

Although there are performance improvements to be made, the U-P2P system is usable today for small deployments. Some obvious improvements can be made immediately which are discussed above. Allowing a multiple central server deployment is one important example. The ability to form communities and to use communities to partition the files among multiple servers illustrates that U-P2P can be used to improve even the most basic peer-to-peer deployment. We believe that U-P2P can provide similar scalability and performance gains to other peer-to-peer deployments. The application of communities in a fully decentralized (Gnutella-style) deployment is explored below in the discussion of future work.

Chapter 7 Conclusions and Future Work

7.1 Contributions

U-P2P is a novel application framework that exploits metadata to simplify the design, use, sharing and discovery of peer-to-peer file-sharing communities.

It is an application framework because it provides users with tools to tailor file-sharing to their own applications. Some examples which we have seen are customized communities for sharing MP3s, fruit fly genes or stamps. Additionally, massive communities can be adapted to focus on smaller niches: for example a file-sharing application dedicated to sharing a particular genre of music.

Although U-P2P's use of metadata is not novel, metadata is at the heart of what makes U-P2P an advancement in the state of the art. Existing systems either do not exploit metadata at all, or do not allow it to be used in a flexible manner. These often present barriers to customizing file-sharing applications to diverse applications.

The major contributions of U-P2P which improve search and simplify the design, use, sharing and discovery of peer-to-peer file-sharing communities are:

Standard Metadata Layer

All U-P2P communities include metadata describing important properties such as format and protocol. The metadata fields are defined by a U-P2P community schema. Each

community includes a schema which defines the format of the metadata associated with files in that community. The framework is generic enough to handle diverse communities yet provide powerful services based on that metadata.

User-designed Communities

All of the previous points underpin U-P2P's basic goal: making it simple to create file-sharing communities. Users design communities by writing community XML documents and choosing an existing schema or creating a new one for their format. This enables metadata search and publishing within the community. Publishing the new community allows it to be discovered by others.

A Community for Communities

U-P2P includes a bootstrap community, the root community, which allows U-P2P users to discover other file-sharing communities. The root community provides the infrastructure which allows users to publish new communities. The metadata associated with each community makes it simple to find the relevant community.

Performance and Scalability Gains

U-P2P communities offer a logical way to partition resources such as database size and network size. Working within these partitions reduces the load on system-wide resources without compromising the quality of results. The resulting performance gain makes it possible for each community to scale to greater sizes than would be possible without partitioning.

These abilities contrast with some existing file-sharing systems which fall on a spectrum between no use of metadata to limited use of metadata. These systems are not flexible and there is no standard way to extend them to support new data formats. Another class of applications attempts to use implicit metadata to classify documents and improve search, but these implicit indicators may lead to incorrect assumptions of common interests or common meanings.

The applications closest in nature to U-P2P are Alpine and JxtaSearch. Alpine communities are defined per user so there is no concept of joining an existing community and there is no shared benefit. JxtaSearch queryspaces allow providers to advertise in a structured way the types of queries they can serve, but there is no guarantee that they will respond with results that satisfy the same schema. Indeed, the original vision of JxtaSearch was to allow different providers to respond with whatever information they thought relevant regardless of the intent of the searcher.

U-P2P's ability to perform metadata publishing and search is an improvement over the basic abilities found in most popular systems. U-P2P improves on all of these systems in the ease with which it can be extended to different communities. Additionally, none of these systems provides any mechanism for community discovery.

7.2 Future Work

One important area for improvement is performance. Some of the options for improving performance were discussed in the evaluation of U-P2P and include: improvements in the

database implementation (eXist or possibly a new implementation) and optimizing operations to reduce database lookups.

Another major direction for future work is in demonstrating the protocol independence of U-P2P. Developing Network Adapters to interface to existing networks such as Freenet or Gnutella, would prove that U-P2P can be generic meta-data layer for different deployments.

For Gnutella especially the implementation of per-community neighbour lists may provide performance improvements. In traditional implementations, requests are flooded to all neighbours. We can think of nodes belonging to a community as an overlay network. By maintaining a unique list of neighbours for each community which lie within the overlay load is reduced. Only nodes which can plausibly service the query will receive it. Accordingly, a request with the same TTL will traverse many more relevant nodes than it would if flooded in the traditional manner.

U-P2P communities may also be extended beyond the schema format to include definition of parameters such as security and authentication. These various parameters define a design space for peer-to-peer systems. Instead of envisioning a single system that is all things to all people, a more practical approach is considering which methods are most appropriate to the application. It is an open and intriguing question how important features such as security or authentication can be decoupled and mixed and matched.

Some challenges associated with communities must also be addressed. If everybody can create a community there goes the neighbourhood: an enterprise once limited to open source hackers and venture capitalists becomes viable for a much wider audience. What methods can be used to distinguish between 11 000 communities dedicated to sharing MP3s? The incorporation of trust metrics is one possible answer and might result in a process of natural selection where communities with the ideal characteristics triumph over those using inferior schemas or protocols. Trust metrics are also applicable within communities to address the problem of untrustworthy metadata.

The analogy with natural selection also suggests more practical concerns. How can communities evolve? One approach would be to support some sort of format versioning to allow files from the old community to co-exist with those from the new.

Accommodating changes in other properties such as protocol is more challenging: if a file cannot even be accessed using the new protocol how can it be made visible to the new community?

The analogy between classes and communities also suggests the question of how the concept of multiple inheritance can be applied to U-P2P. Communities could be created by composing the properties of multiple parent communities. Another application of community composition would be to allow files to exist in multiple communities arranged in a hierarchy. Sharing a file in one of the more specific communities would also allow it to be found when searching in a more general community.

The current experimental implementation of the Universal Peer-to-peer concept is not universal in its support for protocols and other dimensions involved in file-sharing. We hope future work in this area will eventually achieve this goal. But in one important way, U-P2P is universal: users can easily create new communities using existing formats or write their own without limitations. We hope that this powerful capability will enable people in diverse fields and with diverse interests to easily take advantage of the power of peer-to-peer file-sharing.

References

[Alpine03] Alpine Network website, <http://www.cubicmetercrystal.com/alpine/>, 2003.

[Apache02] Apache Software Foundation, “*Xerces2 Java Parser Readme*”, 2002. See <http://xml.apache.org/xerces2-j/index.html>.

[Apache03] Apache Software Foundation, “*Apache Tomcat*”, 2003. See <http://jakarta.apache.org/tomcat/>.

[Apache03b] Apache Software Foundation, “*Xalan-Java version 2.5.1*”, 2003. See <http://xml.apache.org/xalan-j/>.

[Apache03c] Apache Software Foundation, “*Log4j*”, 2003. See <http://jakarta.apache.org/log4j/docs/index.html>.

[Arthorne02] Neal Arthorne, “*XML Schema for Patterns*”, February 2002. See <http://chat.carleton.ca/~narthorn/project/patterns/pattern.xsd>.

[Arthorne02b] Neal Arthorne, “*XML Tools for Resource Sharing*”, April 2002. See [http://chat.carleton.ca/~narthorn/project/finalReport/XML Tools for Resource Sharing.pdf](http://chat.carleton.ca/~narthorn/project/finalReport/XML%20Tools%20for%20Resource%20Sharing.pdf).

[Arthorne02c] Neal Arthorne, "*U-P2P Generic Peer Messages*", November 2002. See <http://u-p2p.sourceforge.net/release/up2p-1.0-B1/GenericPeerMessages.html>.

[Arthorne02d] Neal Arthorne, "*Universal Peer-to-peer (U-P2P) Developer's Guide*", November 2002. See <http://u-p2p.sourceforge.net/release/up2p-1.0-B1/DevGuide.pdf>.

[Arthorne03] Neal Arthorne, Babak Esfandiari and Alope Mukherjee, "*U-P2P: A Peer-to-peer Framework for Universal Resource Sharing and Discovery*", Proceedings of Freenix track of Usenix 2003, 29-38, June 2003.

[Berners-Lee01] T. Berners-Lee, J. Hendler and O. Lassila, "*The Semantic Web*", Scientific American, 284(5)34-43, May 2001.

[Bitzi03] The Bitzi Corporation Homepage, <http://bitzi.com>, 2003.

[Clarke00] I. Clarke, O. Sandberg, B. Wiley, T. Hong, "*Freenet: a distributed anonymous information storage and retrieval system*", In Proc. Workshop on Design Issues in Anonymity and Unobservability, 311-320, July 2000.

[Castro02] M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron, "*One ring to rule them all: Service discovery and binding in structured peer-to-peer overlay networks*", SIGOPS European Workshop, September, 2002.

[Clip200] Clip 2 Corporation, "*Gnutella: To the Bandwidth Barrier and Beyond*", November 6, 2000. See

<http://web.archive.org/web/20010606114842/http://www.clip2.com/gnutella.html>.

[Cohen03] E. Cohen, A. Fiat and H. Kaplan, "*Associative Search in Peer to Peer Networks: Harnessing Latent Semantics*", In Proceedings of the IEEE INFOCOM'03 Conference, 2003.

[Cointe87] P. Cointe, "*Metaclasses are First Class: the ObjVlisp Model*", In Proceedings OOPSLA '87, 156-167, October 4, 1987.

[Crespo02] A. Crespo and H. Garcia-Molina, "*Routing Indices for peer-to-peer systems*", In Proceedings International Conference on Distributed Computing Systems, 23-34, July 2002.

[Dabek01] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica, "*Wide-area cooperative storage with CFS*", In Proc. 18th ACM Symposium on Operating Systems Principles (SOSP '01), Banff, Alberta, 202-215 October 2001.

[Dabek02] F. Dabek, E. Brunskill, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica, H. Balakrishnan, "*Building peer-to-peer systems with Chord, a distributed lookup service*", In Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), 81-86, 2001.

[Deugo99] Dwight Deugo, Darrell Ferguson, "*Carleton Pattern Repository*", 1999. See <http://muffin.nexus.carleton.ca/~darrell/repo/> (archived version: <http://web.archive.org/web/20010611121845/muffin.nexus.carleton.ca/~darrell/repo/>)

[Dingledine00] R. Dingledine, M. J. Freedman, D. Molnar. "*The Free Haven Project: Distributed Anonymous Storage Service*", In Proceedings of the Workshop on Design Issues in Anonymity and Unobservability, 67-95, July 2000.

[Dingledine01] R. Dingledine, M. J. Freedman, D. Molnar, "Chapter 12: Free Haven" in *Peer-to-Peer: Harnessing the Benefits of Disruptive Technologies* ed. A. Oram, O'Reilly & Associates, 159-190, 2001.

[Dornfest01] R. Dornfest, "Chapter 13: Metadata" in *Peer-to-Peer: Harnessing the Benefits of Disruptive Technologies* ed. A. Oram, O'Reilly & Associates, 191-202, 2001.

[Druschel01] P. Druschel and A. Rowstron, "*PAST: a large-scale, persistent peer-to-peer storage utility*", In Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), 75-80, 2001.

[Esfandiari96] B. Esfandiari, G. Deflandre, J. Quinqueton and C. Dony, "*Agent-Oriented Techniques for Network Supervision*", Annals of Telecommunications, Tome 51, No. 9-10, Sept 1996, pp 521-529.

[Ferguson00] Darrell Ferguson. “*Updates to the Pattern Repository*”, February 2000.
See <http://muffin.nexus.carleton.ca/~darrell/papers/UpdateReport.pdf> (archived version:
<http://web.archive.org/web/20010615071832/http://muffin.nexus.carleton.ca/~darrell/papers/UpdateReport.pdf>)

[Florescu99] Daniela Florescu and Donald Kossman, “*A performance evaluation of alternative mapping schemes for storing XML data in a relational database*”, Technical Report 3684, INRIA, March 1999

[Flybase03] The FlyBase Consortium (2003). “The FlyBase database of the *Drosophila* genome projects and community literature”, *Nucleic Acids Research* 31:172-175., 2003.
See <http://flybase.org>.

[Freeweb01] Freeweb Project Website, <http://freeweb.sourceforge.net>, 2001.

[GeneOntology00] The Gene Ontology Consortium, “*Gene Ontology: tool for the unification of biology*”, *Nature Genetics* 25:25-29, 2000.

[Goldberg83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[Gong01] L. Gong, "*JXTA: A Network Programming Environment*", IEEE Internet Computing, 5(3)88-95, May/June 2001

[Harold02] E. R. Harold, W. S. Merens, "Chapter 9: XPath" in *XML in a Nutshell 2nd Edition*, O'Reilly & Associates, 154-174, 2002. See <http://www.oreilly.com/catalog/xmlnut/chapter/ch09.html>.

[Harren02] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, I. Stoica, "Complex queries in DHT-based peer-to-peer networks", In Proc. 1st International Workshop on Peer-to-peer Systems (IPTPS '02), Cambridge, MA, 242-259, March 2002.

[ICQ03] ICQ Inc. Homepage, <http://web.icq.com>, 2003.

[Joseph02] S. Joseph, "*NeuroGrid: Semantically Routing Queries in Peer-to-Peer Networks*", In Proceedings of the International Workshop on Peer-to-Peer Computing, Pisa, Italy, 202-214, May 2002.

[Joseph02b] S. Joseph, "*Decentralized Meta-Data Strategies*", 2002. See http://www.neurogrid.net/Decentralized_Meta-Data_Strategies-neat.html.

[Kan01] G. Kan, "Chapter 4: Gnutella" in *Peer-to-Peer: Harnessing the Benefits of Disruptive Technologies* ed. A. Oram, O'Reilly & Associates, 94-122, 2001.

[Kazaa03] Kazaa file-sharing network site, <http://www.kazaa.com>, 2003.

[Keleher02] P. Keleher, B. Bhattacharjee, B. Silaghi, “*Are virtualized overlay networks too much of a good thing?*”, In Proc. 1st International Workshop on Peer-to-peer Systems (IPTPS '02), Cambridge, MA, 225-231, March 2002.

[Kronfol02] A. Z. Kronfol, “*FASD: A fault-tolerant, Adaptive Scalable, Distributed Search Engine*”, Princeton University Masters Thesis, 2002. See http://freenetproject.org/twiki/Main/FASD/kronfol_final_thesis.pdf.

[Ley03] Michael Ley, “*DBLP: Computer Science Bibliography*”, 2003. See <http://dblp.uni-trier.de>.

[Limewire03] Limewire homepage, <http://www.limewire.com>, 2003.

[Lumrix03] Lumrix Website, <http://puvogel.informatik.med.uni-giessen.de/lumrix/>, 2003.

[Lv02] Q. Lv, S. Ratnasamy, S. Shenker, “*Can heterogeneity make Gnutella scalable?*”, In Proc. of IPTPS 2002, Cambridge, MA, 94-103, March 2002.

[Magenta01] The Magenta project website, <http://magenta.sourceforge.net>, December 2001.

[MediaOrg01] Media.org archive of rescued raw EDGAR data,
<http://bulk.resource.org/edgar>, 2001.

[Meier03] W. Meier, “eXist: Open Source XML Database”, 2003. See <http://exist-db.org>.

[Milojicic02] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu, “Peer-to-peer computing”, Technical Report HPL-2002-57, HP Lab, 2002. See <http://www.hpl.hp.com/techreports/2002/HPL-2002-57.html>.

[Minar01] Nelson Minar, “Peer-to-Peer is Not Always Decentralized...when Centralization is Good”, Presentation at O’Reilly Peer-to-peer and Web Services Conference, December 14, 2001. See <http://www.openp2p.com/p2p/2001/12/14/examples/nelson-centralization.ppt>.

[Mukherjee02] A. Mukherjee, B. Esfandiari, N. Arthorne, “U-P2P: A Peer-to-peer System for Description and Discovery of Resource-sharing Communities”, ICDCS Workshops 2002: 701-705, July 2002.

[Murray-Rust99] Peter Murray-Rust and Henry S. Rzepa. “Chemical Markup, XML, and the Worldwide Web. 1. Basic Principles”, J. Chem. Inf. Comput. Sci., 39(6): 928-942, May 26, 1999.

[Napster02] Napster Inc. Website, <http://www.napster.com>, November 2002. (Napster Inc. closed down in November of 2002 – past versions of its website can be found at http://web.archive.org/web/*/http://www.napster.com.)

[Nejdl01] W. Nejdl, et al., "*EDUTELLA: A P2P Networking Infrastructure Based on RDF*", November 14, 2001. See <http://edutella.jxta.org/reports/edutella-whitepaper.pdf>.

[Nilsson00] M. Nilsson, "*ID3 tag version 2.4.0 - Main Structure*", November 1, 2000. See <http://www.id3.org/id3v2.4.0-structure.txt>.

[NIST95] NIST, "*Secure Hash Standard (SHS)*", Federal Information Processing Standards Publication 186, April 1995.

[ObjectMentor02] Object Mentor Inc., "*JUnit: Testing Resources for Extreme Programming*", 2002. See <http://junit.org>.

[Ratnasamy01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, "*A scalable content-addressable network*", In Proc. ACM SIGCOMM '01, San Diego, CA, 161-172, Aug. 2001.

[Ratnasamy02] S. Ratnasamy, S. Shenker, I. Stoica, "*Routing algorithms for DHTs: some open questions*", In Proc. IPTPS 2002, Cambridge, MA, 45-52, March 2002.

[Rivest92] R. Rivest, “*The MD5 message-digest algorithm*”, IETF Network Working Group, RFC 1321, April 1992.

[Rohrs02] C. Rohrs, “*Query Routing for the Gnutella Network*”, May 16, 2002. See http://www.limewire.com/developer/query_routing/keyword%20routing.htm.

[Rowstron01] A. Rowstron and P. Druschel, “*Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems*”, In Proc. 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001). Heidelberg, Germany, 329-350, November 2001.

[Rubin96] G. M. Rubin, “*Around the Genomes: The Drosophila Genome Project*”, Genome Research 6:71-79, 1996. See <http://www.fruitfly.org/about/pubs/rubin96.html>.

[Shanmugasundaram99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, Jeffrey F. Naughton, “*Relational Databases for Querying XML Documents: Limitations and Opportunities*”, In VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, Edinburgh, Scotland, 302-304, 1999.

[Shirky01] C. Shirky, “Chapter 2: Listening to Napster” in *Peer-to-Peer: Harnessing the Benefits of Disruptive Technologies* ed. A. Oram, O'Reilly & Associates, 21-37, 2001.

[Stoica01] I. Stoica, R. Morris, D. Karker, M. F. Kaashoek, H. Balakrishnan, "*Chord: a scalable peer-to-peer lookup service for internet applications*", In Proc. ACM SIGCOMM '01, San Diego, CA, 149-160, Aug. 2001.

[Sun02] Sun Microsystems Inc., "*Java 2 Platform, Standard Edition, v 1.4.1 API Specification*", 2002. See <http://java.sun.com/j2se/1.4.1/docs/api/>.

[Sun03] Sun Microsystems Inc., "*Java Servlet Technology*", 2003. See <http://java.sun.com/products/servlet/>.

[Sun03b] Sun Microsystems Inc., "*JavaServer Pages*", 2003. See <http://java.sun.com/products/jsp/>.

[Thadani01] S. Thadani, "*Meta Information Searches on the Gnutella Network*", 2001. See http://www.limewire.com/index.jsp/metainfo_searches.

[Thadani01b] S. Thadani, "*Meta Data searches on the Gnutella Network (addendum)*", 2001. See <http://www.limewire.com/developer/MetaProposal2.htm>.

[Waldman00] M. Waldman, A.D. Rubin, and L.F. Cranor, "*Publius: A robust, tamper-evident, censorship-resistant and source-anonymous web publishing system*", In Proceedings of the Ninth USENIX Security Symposium, 59-72, August 2000.

[Waterhouse01] S. Waterhouse, "*JXTASearch: Distributed Search for Distributed Networks*", May 2001. See <http://search.jxta.org/JXTASearch.pdf>.

[Winer99] Dave Winer, "*XML-RPC Specification*", June 1999. See <http://www.xmlrpc.com/spec>.

[W3C99] W3C, "*XSL Transformation (XSLT)*", W3C Recommendation, November 16, 1999. See <http://www.w3.org/TR/xslt>.

[W3C99b] W3C, "*XML Path Language (XPath)*", W3C Recommendation, November 16, 1999. See <http://www.w3.org/TR/xpath>.

[W3C00] W3C, "*Extensible Markup Language (XML) 1.0 (Second Edition)*", W3C Recommendation, October 6, 2000. See <http://www.w3.org/TR/REC-xml>

[W3C01] W3C, "*XML Schema Part 0: Primer*", W3C Recommendation, May 2, 2001. See <http://www.w3.org/TR/xmlschema-0/>.

[W3C03] W3C, "*RDF Primer*", W3C Working Draft, January 23, 2003. See <http://www.w3.org/RDF/>.

[XMLDB03] The XML:DB Initiative, "*XML:DB Application Programming Interface for XML Databases*", 2003. See <http://www.xmldb.org/xapi/>.

[Yang02] B. Yang, H. Garcia-Molina, "*Improving Search in Peer-to-Peer Systems*", In Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria, 5-14, July 2002

[Yang03] B. Yang, H. Garcia-Molina, "*Designing a Super-peer Network*", In Proceedings of the 19th International Conference on Data Engineering (ICDE), Bangalore, India, March 2003

A Appendix: XML Schema Primer

This appendix is primarily due to Neal Arthorne and is used with his permission.

U-P2P relies heavily on XML Schema. This section presents a brief description of the structure of XML Schema documents.

XML Schema is a Recommendation (2 May 2001) by the World Wide Consortium for a schema language for XML. It constitutes a set of markup tags themselves written in XML, that are used to describe both the structure of an XML document and the datatypes of individual elements and attributes. XML Schema uses the XML Namespace Recommendation that allows authors to prefix their own tags with a specific namespace. It also allows importing types from other schemas both in the current namespace and from other namespaces.

XML Schema can be used to validate XML documents using a schema processor. XML documents themselves can link to their schema using a `schemaLocation` tag. However this is only a suggested method for specifying a schema for the document and they may use other types of schemas such as DTDs or not require validation at all.

In XML Schema, there are four basic units that can be used to describe parts of an XML document: elements, attributes, `complexType`s and `simpleTypes`.

A.1 Elements

Elements are the tags used in XML to label a piece of data. They have a datatype associated with them that is either defined at the same time as the element (anonymously) or is a `complexType` or `simpleType`. Elements can contain sub-elements, attributes and can also be part of an XML Namespace. The structure of an element can be controlled in XML Schema using sequences, choices and the same restrictions used to derive simple and complex types. For example an author can specify that an element called 'name' should consist of an exact sequence of the two elements 'firstname' and 'lastname'.

An example of a simple element declaration is:

```
<xs:element name="title" type="xs:string" />
```

Example of XML conforming to this schema: `<title>Romeo and Juliet</title>`

A.2 Attributes

Attributes are properties that belong to a parent element. They can only use `simpleTypes` and they cannot contain any elements or attributes. Attributes can be optional or mandatory within an element. For example:

```
<xs:element name="term">
  <xs:complexType>
    ...
    <xs:attribute name="title" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>
```

This would define the XML tag: `<term title="some string value">`

A.3 SimpleTypes

These types are either built-in data types specified in the XML Schema Recommendation or simpleTypes derived from the built-in or existing types (XML Schema defines some derived types such as date and time). The Recommendation includes many useful data types such as string and decimal that can be easily manipulated into more specialized simpleTypes. For example, you could derive a simpleType that is a list of months in the year by restricting the string type and providing an enumeration for the month strings:

```
<xsd:simpleType name="monthsOfYear">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="January"/>
    <xsd:enumeration value="February"/>
    ....
  </xsd:restriction>
</xsd:simpleType>
```

Example XML conforming to this schema would be

```
<monthsOfYear>January</monthsOfYear>.
```

In the above example, enumeration is called a 'facet' and is used to restrict an existing simpleType. Other facets include minLength, maxLength, pattern and other restrictions.

A.4 ComplexTypes

These types are a sequence or choice between a set of elements and/or attributes. They are reusable (when defined on a global level) and can be used in any part of the current schema. Once a complexType is defined an element can use it by specifying the type attribute equal to the name of the complexType. An example of a complex type follows:

```
<xsd:complexType name="CanadaAddress">
  <xsd:sequence>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
    <xsd:element name="postalCode">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:pattern="\w\d\w\s\d\w\d"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element name="province" type="xsd:string"/>
    <xsd:element name="country" type="xsd:string" fixed="Canada"/>
  </xsd:sequence>
</xsd:complexType>
```

Example XML conforming to this schema:

```
<CanadaAddress>
  <street>1125 Colonel By Drive</street>
  <city>Ottawa</city>
  <postalCode>K1S 5B6</postalCode>
  <province>Ontario</province>
  <country>Canada</country>
</CanadaAddress>
```

Having defined the above example, the type can then be used in an element anywhere in the schema or even referenced from other schema. Note how the simpleType postalCode was defined anonymously within the structure. This means that postalCode cannot be used on its own and has no meaning outside of the scope of CanadaAddress.

A.5 Deriving Types

There is a basic hierarchy for deriving types:

simpleTypes -> derived simpleTypes -> complexTypes -> derived complexTypes

Both simple and complex types can be derived either by extension (add values, elements or attributes) or by restriction (constrict lexical values in simpleTypes or elements and attributes in complexTypes).