Technological University Dublin
ARROW@TU Dublin

Masters | Engineering

2004-10-01

Peer-to-peer Searching and Sharing of Electronic Documents

Paul Stacey
Technological University Dublin, paul.stacey@tudublin.ie

Follow this and additional works at: https://arrow.tudublin.ie/engmas

Part of the Computer Engineering Commons

Recommended Citation

Stacey, P. (2004). *Peer-to-peer searching and sharing of electronic documents*. Masters dissertation. Technological University Dublin. doi:10.21427/D7TK75

This Theses, Masters is brought to you for free and open access by the Engineering at ARROW@TU Dublin. It has been accepted for inclusion in Masters by an authorized administrator of ARROW@TU Dublin. For more information, please contact yvonne.desmond@tudublin.ie, arrow.admin@tudublin.ie, brian.widdis@tudublin.ie.

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 License

Peer-to-Peer Searching and Sharing of Electronic Documents

By

Paul Stacey

Master of Philosophy (Mphil)
In
Computer Engineering

School of Control Systems and Electrical Engineering
Dublin Institute of Technology

Supervised by:
Mr. Damon Berry
Dr. Eugene Coyle

October 2004

I certify that this thesis which I now submit for examination for the award of Master of Philosophy in Computer Engineering, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

This thesis was prepared according to the regulations for postgraduate study by research of the Dublin Institute of Technology and has not been submitted in whole or in part for an award in any other Institute or University.

The Institute has permission to keep, to lend or to copy this thesis in whole or in part, on condition that any such use of the material of the thesis be duly acknowledged.

Signature Paul Stacey

Date 17th Nov 2005

Acknowledgements

It is with great pleasure that I would like to thank the people who made writing this thesis possible. I would like to gratefully acknowledge the enthusiastic supervision of Mr. Damon Berry. Throughout my thesis-writing period and research, Damon provided much encouragement and sound advice. I would like to thank Dr. Eugene Coyle for his support and guidance throughout this research. I am also grateful to everyone in room 36C for providing an enjoyable and relaxed working environment. I would also like to thank Elaine for all her support and patience throughout the writing of this thesis. Thanks!

Abstract

Peer-to-peer systems have existed since the first incarnation of the Internet. In recent times the Internet has taken on a more hierarchical form, power has been taken away from the individual and placed in the hands of operators of large servers. However, with the re-emergence of p2p the individual is gaining more freedom. End users attached to the Internet now have the power to host and publish content through the use of p2p technologies. Peer-to-peer systems present significant design challenges and have opened up a floodgate of research in an effort to overcome some of the fundamental problems.

This thesis details a conceptual peer-to-peer design solution to some of the fundamental problems facing p2p systems today. A prototype is developed as a proof-of-concept. The prototype demonstrates how by enabling users to join communities of other like-minded users in order to exchange files, the p2p environment becomes more structured. It is argued that this structuring of the network into communities leads the way for a more scalable p2p system. Utilising the routing capabilities of the latest p2p algorithms, the prototype includes an indexing service that facilitates the creation of virtual rendezvous points for users with similar interests (manifested by shared keywords). Users are described by the content they store. By using state of the art Information Retrieval techniques users can be grouped together to form content sensitive communities. It is intended that this system be used as the basis of a distributed archive of research papers. The concept presented seeks to improve the efficiency of file searches associated with p2p file sharing applications. This can be achieved by incorporating a novel content community service layer in order to organise nodes based on the content they store. This has the potential to improve on existing systems such as Gnutella where nodes are organised in a somewhat random fashion. Initial findings suggest that the system has better scalability properties giving the user better search results that reflect the true content available within the system. Search results are more comprehensive; as search queries are based on semantic meaning rather than literal matching.

Table of Contents

1 Introduction 4
- 1.1 Peer-to-Peer Networking ... 4
- 1.2 Hybrid Versus Pure P2P ... 5
- 1.3 Hybrid Peer-to-Peer Systems ... 6
  - 1.3.1 Napster ... 6
- 1.4 Pure Peer-to-Peer Systems ... 7
  - 1.4.1 Gnutella ... 8
- 1.5 Present Day P2P ... 9
- 1.6 Motivation for Research ... 11
- 1.7 Thesis Outline ... 12

2 Distributed Hash Tables 13
- 2.1 Hash-Tables ... 13
  - 2.1.1 Distributing the Hash Table ... 14
- 2.2 A New Beginning for P2P ... 16
- 2.3 Developing a Suitable Routing Substrate for P2P ... 20
  - 2.3.1 Pastry ... 21
  - 2.3.2 Tapestry ... 22
  - 2.3.3 CAN ... 22
  - 2.3.4 Chord ... 23
- 2.4 Discussion of Distributed Hash Table Implementations ... 24
- 2.5 Projects Using Distributed Hash Tables ... 24
  - 2.5.1 PAST ... 24
- 2.6 DHTs Versus the "Napsters" of P2P ... 25
  - 2.6.1 Limitations of DHTs ... 26

3 Information Retrieval (IR) 27
- 3.1 An Introduction to IR ... 28
  - 3.1.1 Current IR Systems ... 29
  - 3.1.2 IR in P2P Systems ... 30
  - 3.1.3 State-of-the-art IR Techniques ... 31
- 3.2 Vector Space Approaches ... 31
  - 3.2.1 Generating Document Representatives ... 32
  - 3.2.2 Vector Space Modelling of Documents ... 33
  - 3.2.3 Document Pre-processing ... 35
- 3.3 Classification of Documents ... 36
  - 3.3.1 Centroid Based Classification ... 36
- 3.4 Clustering of Data ... 37
- 3.5 Information Retrieval and P2P ... 39

4 Design Concepts and Solutions 40
- 4.1 The Substrate Layer ... 41
  - 4.1.1 Building on Pastry ... 42
- 4.2 Potential Solutions ... 42
  - 4.2.1 Fuzzy Domains ... 42
  - 4.2.2 Algorithms and Avalanches ... 44

4.2.3 Zip Technology........ 45
4.2.4 Vector Space Modelling........ 46
4.3 The Final Solution........ 46
4.3.1 Building a Decentralised indexing service........ 46
4.3.2 Creating indexing nodes........ 47
4.3.3 Building Content Sensitive Communities........ 48
4.3.4 Comparing Nodes Based on Content Stored........ 49
4.3.5 Organising Network into Communities........ 50
4.3.6 Two Layer Network........ 51
4.4 Searching for Documents........ 52
4.5 System Overview........ 53

5 Software Design and Implementation **58**
5.1 The Pastry Application Programming Interface (API)........ 60
5.2 System Classes and packages........ 62
5.2.1 The Vector Space Modelling Package........ 63
5.2.2 The Community Package........ 65
5.2.3 The Register Package........ 67
5.2.4 The Message Package........ 69
5.2.5 The Node Package........ 69
5.3 Imported API's........ 70
5.3.1 JAMA: A Java Matrix Package........ 70
5.3.2 PJX........ 71

6 Evaluation and Discussion **73**
6.1 Evaluation and Simulation........ 74
6.1.1 Evaluation of Vector Space Modelling Algorithm........ 74
6.1.2 Simulation of Peer-to-Peer Networks........ 77
6.1.3 Simulating a Peer-to-Peer File-Sharing Network........ 77
6.1.4 Modelling Content Distribution........ 78
6.1.5 Implementation of the Simulator........ 80
6.1.6 Network Simulators........ 80
6.1.7 Simulating with J-Sim........ 84
6.1.8 Simulations and Results........ 89
6.2 Discussion and Future Work........ 93
6.2.1 Possible Optimisations........ 93
6.2.2 Future Investigation........ 94
6.2.3 Possible Extensions of the Work........ 95

7 Conclusion 97

Bibliography 99

Glossary 108

Appendix A 116

Appendix B 139

Appendix C 143

Appendix D 156

Chapter 1

"Blocking Napster is like standing before hundreds of hungry jackals and shouting "Shoo!" to keep them from 400 pounds of raw hamburger." (Chronicle, 9/21/00)

Introduction

Community: a body of people having common rights, privileges, or interests [1]. Applying this notion of community to the Internet, it seems that the Internet as a whole lacks a sense of community. The World Wide Web was originally envisaged in 1986 by Tim Berners-Lee as a way for academics to share knowledge [2]. The web has become a victim of its own success in relation to that goal. While there is a huge quantity of information on the web and it is easy to find text about almost any topic, quality results are often hidden in a huge set of results. The client-server paradigm may be partly to blame for this. In recent times, systems such as Napster [3] and Gnutella [4] have gained huge popularity. These systems have initiated a surge of interest and research into the peer-to-peer (p2p) framework. These systems are restructuring the Internet away from the client server model to one where a client is also a server, giving individuals more freedom and control. Users can now choose to be connected to more specific stores of data.

1.1 Peer-to-Peer Networking

Peer-to-peer networking has recently been made famous by the music industry. The arrival of Napster in May of 1999 ignited serious debate and legal proceedings all based around the issue of copyright and the rights of people to share music. But in reality p2p is the oldest architecture in the world of communications. The Internet was originally conceived in the late 1960s as a p2p system. This original incarnation of the Internet (called ARPANET [5]) was built to share computing resources around the U.S., within the network each peer was an equal player. In subsequent years the Internet has become more restrictive and the client/server paradigm has become the main architecture of the Internet. This means that control has been taken away from the individual and placed in the hands of operators of large servers. This, some argue, leaves these servers and the content they serve open to abuse and censorship. But as has

been observed in recent years, p2p applications are becoming common again, giving the end user back more power and control.

Until Napster, the sharing of music over the Internet or web did not prove much of a threat towards the music industry, but the use of p2p technology is proving to be a major headache for such organisations as the Recording Industry Association of America (RIAA). The RIAA has sought to prosecute the users and implementers of Napster and has even succeeded in shutting the original system down. This success for the RIAA was short lived as a new breed of p2p systems were emerging; due to their technical design these systems have managed to escape any copyright infringements and thus have managed to stay operational. As a result it could be said the move to a completely decentralised approach such as Kazaa [6] may have been more legally than technically motivated.

The discussion begins by looking at the architectures of Napster and Gnutella. These two systems are chosen to highlight some fundamental technical issues, although these applications are becoming a somewhat historical starting point for any p2p discussion.

1.2 Hybrid versus Pure P2P

The most distinctive difference between p2p architectures and client/sever architectures is "the concept of an entity acting as a *Servent*" [7]. A servent describes a node that acts both as a server and client. P2P systems can again then be broken down into two other categories, that of *"pure"* p2p and *"hybrid"* p2p systems. The key distinction of hybrid peer-to-peer compared to pure peer-to-peer is the fact that a hybrid p2p network always includes a central entity, which is forbidden by the definition in pure p2p.

1.3 Hybrid Peer-to-Peer Systems

1.3.1 Napster

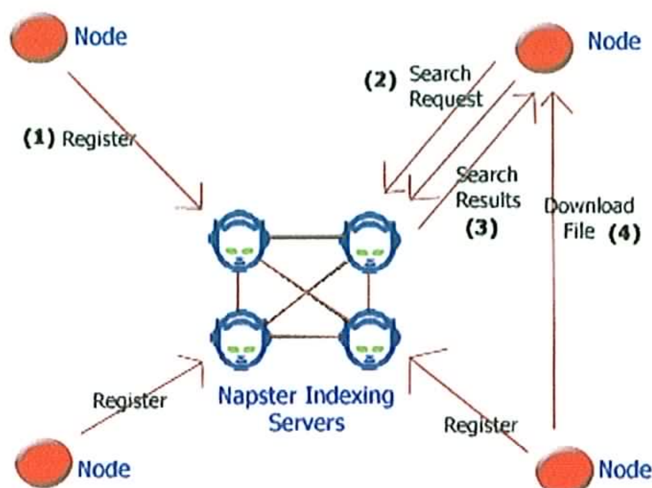

Figure. 1.1. Hybrid Peer-to-Peer system Napster, Peers register their files on a centralised indexing Server (1). Searches are directed to the indexing server (2), which returns the node where the file is located(3) Files can then be downloaded from the node hosting the file (4).

To describe Napster as a peer-to-peer network is not necessarily accurate. Napster's architecture is designed to rely on a centralised indexing service. It is this centralised indexing service that puts Napster in the category of a hybrid peer-to-peer system. There are advantages and disadvantages to this type of system. Other than the legal ramifications of a centralised indexing service, searching of files becomes much easier. The reason for this is that all files within the system are registered on this central server, therefore searches are localised. In hybrid systems it is only the files that are distributed across the peers that comprise the network. The index containing the locations of files is centralised (see figure 1.1) nodes registering their files with the centralised server populate the index. All search requests from nodes are sent to the central server. When a search on the central server returns a hit for a requested file, the location of the file is returned to the peer issuing the search request. The searching application may then connect directly to the peer hosting the file and download directly from this site [3].

The use of a centralised indexing server makes searching easier but introduces problems to the system. One big issue with this type of architecture is robustness, if the

server fails; the whole system becomes inoperable. The server also introduces scalability issues to the system. As more users and files enter the system a bigger server is required [8].

1.4 Pure Peer-to-Peer Systems

Complete decentralisation i.e. no centralised control is what is known as a pure p2p network. This type of architecture solves many of the problems presented by the hybrid model. A pure p2p network boasts such attributes as robustness; if one node fails, the network still works. This is because the network is made up of many connections and nodes that carry traffic and serve data. If one node fails, several links may be lost. However, in most cases there will exist several paths between two nodes. Pure p2p algorithms systems are designed to route messages around these link or path failures and hence provide access to the data still remaining within the network. With severe node loss networks may fragment, this is where groups of nodes become detached and communication between these now autonomous p2p networks becomes temporarily impossible. How p2p systems handle situations like this differ from system to system but in any case content is still available within the separate fragmented groups of nodes but at a reduced quality of service. This may seem like a flaw in design but if we consider the situation that the centralised server in the hybrid model failing the whole system becomes inoperable.

Pure p2p architectures are also more scalable, the reason for this is that they are not reliant on the size of the centralised indexing service. Size has two important meanings here. Physical memory size can be expensive and a limiting factor to the amount of content the network may serve when using a centralised approach. Size also describes the limitations of the server to accept connections and answer requests or for data. In a centralised architecture the central server introduces potential bottleneck to the system. These limitations are over come by a pure decentralised approach. However, complete decentralisation introduces some serious challenges when designing a p2p system. The main problem introduced involves the ability of a peer to find the data or file requested, even if the data is live on the network. A scalable searching mechanism is one of the biggest issues with pure p2p networks. It is useful to examine more closely an implementation of such a system, Gnutella.

1.4.1 Gnutella

Gnutella first appeared in 2000, released as an experiment by Justin Frankel and Tom Pepper the developers of Winamp [9]. It was originally offered to AOL [10] who rejected the idea. The open source community subsequently took on board the project, lead by Gene Kan. Gnutella allows each node within the network to connect to a small group of other nodes that in turn connect to other nodes and so on. In order to search for a particular file, a node issues a search request to the group of nodes that it is connected to. The other nodes receiving this request search their local hard drives for the requested file and also forward the request to the nodes that they are connected to. This type of searching is called *flood searching*, the search request is also given a *time-to-live* (TTL) stamp which determines the number of hops the request will do before 'dying' (see figure 1.2).

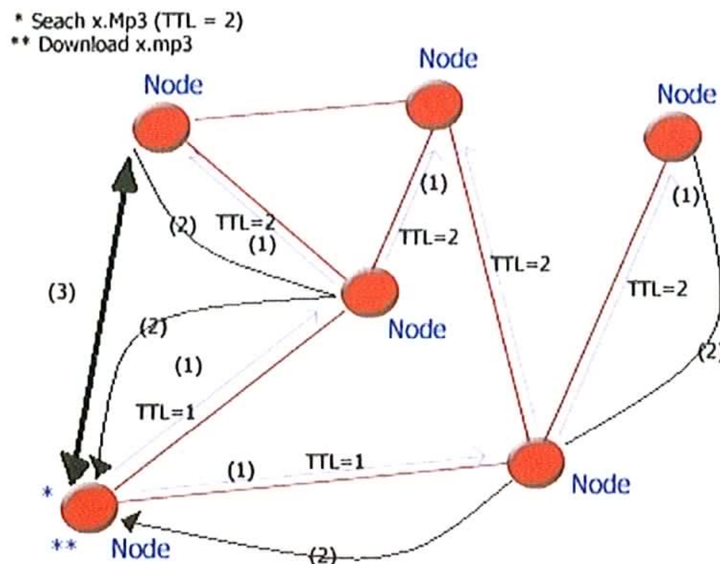

Figure 1.2. A pure p2p network, Gnutella. Search requests are forwarded to nodes that are directly connected who in turn forward the request to other nodes (1). Query hits are sent back to the requesting node (2). The file may be downloaded directly from the peer hosting the file (3).

The problem presented with this type of searching method is that it has scalability issues. As the number of nodes within the network becomes larger, the TTL needed to search every node becomes prohibitively expensive. Search requests eat up bandwidth trying to access every node.

The trade off here is keeping the TTL low and thus leaving parts of the network unreachable. As a result search requests do not return a true representation of the files available within the network. Another interesting point about Gnutella's search

techniques is that Gnutella is built from many different "flavours" of the software. When a node receives a search request, the software acting as the servent may interpret the request differently to other incarnations of the software. Some incarnations look inside the files that are being shared while others interpret multiword queries as conjunctions. Search requests may also return different results. These attributes of Gnutella mean that Gnutella is truly an open non-restrictive system and is truly real-time reflecting what is available at the time of the request. This has advantages for certain situations and disadvantages for others.

Gnutella suffers from some scalability issues. However, it has been shown that Gnutella's flood search technique scales well to 10,000 nodes [11]. Even though it is possible to form private Gnutella networks to cut down on the number of nodes that need to be searched. Private networks have a scaling factor that when exceeded means results of search requests do not reflect all the files that are available at the time of the request.

In the past number of years with the popularisation of p2p through Gnutella and Napster there has been a massive increase in the number of p2p applications running across the Internet. Studies have found that p2p traffic can account for up to 70% of all Internet traffic [12] [13]. The next section takes a brief look at some of the recent developments that are currently in operation creating this massive amount of traffic. The discussion will then lead to what kind of systems can be achieved in the near future and what lessons from current systems can be incorporated into new designs.

1.5 Present day P2P

The application of p2p systems is ever increasing. P2P has been used in such contexts as distributed computation, for the search for extra-terrestrial intelligence with SETI@home [14]. Distributed search engines such as InfraSearch [15] and OpenCOLA [16] and file sharing in a distributed environment through systems such as Gnutella [4], Kazaa [6], BitTorrent [17], Freenet [18] and Napster [3]. All of these systems have been deployed to varying degrees of success. SETI@home, which at the time of writing is investigating its first possible success in its search for evidence of extra terrestrial intelligence. On the other hand the creators of Napster have spent most of their time at the forefront of a legal battle to remain operational. Each system has been successful for different reasons. Their deployment has highlighted some very fundamental design rules

for p2p algorithms. The attributes that make them successful can be reused in other systems and the attributes that limit their success can be redesigned. For example systems such as Morpheus [19] and Kazaa have implemented a kind of hierarchical network where nodes with sufficient bandwidth become super peers, this design aids searching for files, and is a rework of the original Gnutella implementation. Most of the current file sharing systems mentioned above are different incarnations of the same ideas with slightly different methods of searching and organising nodes. In order to find more of the innovative and unique ideas that are being developed one has to look to the research community. Some of the more promising p2p protocols are still only in their early stages. Behind the publicity about courtrooms and aliens there has been a flurry of research into improving p2p systems from lessons learned from the above systems. One interesting project is under development by Sun Microsystems. JXTA [20] is Sun's attempt to get involved with the increasingly popular peer-to-peer computing revolution. JXTA attempts to define a framework for peer-to-peer applications by establishing a *virtual network* overlay on top of the existing physical network infrastructure. It provides mechanisms to advertise and find peers, peer groups, services and content information. It also provides primitives for communication between peers. The trend towards defining an overlay or framework for operational p2p systems is not confined to JXTA. One of the most interesting recent developments comes in the form of distributed hash-tables. Distributed hash-tables (DHT) have been shown to be both scalable and robust, two very much sought after qualities of p2p systems.

Arising from the seminal work of Plaxton et al. [21] several independent projects appeared in 2001. Pastry [22], Tapestry [23], CAN [24] and Chord [25] attempt to achieve the constant access time performance achieved with a traditional hash table and employ this idea in a distributed p2p system. JXTA is a higher-level implementation than other substrate overlay networks such as DHT systems. This is where JXTA presents problems. In Sun's attempts to define such a framework, JXTA has become for some applications, too constraining. It has been argued that because p2p networking is such an open research topic and that p2p networks represent a shift from the normal client-server paradigm to a more "free" and non-regulated environment. Sun's efforts of defining such a framework may in fact go against the principles of "free" p2p systems. Distributed hash tables are a lower-level implementation of a p2p framework and give the designer much more freedom of design. There are a number of projects using DHT systems as frameworks for p2p systems, PAST [26], Skipnet [27].

PAST is a global storage system built on top of Pastry, a similar system is Oceanstore [28], which is built on top of Tapestry. Distributed hash tables are discussed in detail in Chapter two.

The above discussion has given the reader a general look at the area of peer-to-peer networking and discusses some of the first incarnations of p2p systems to where the area is today. The rest of this chapter describes the motivation behind this research and thesis, and concludes by giving a brief overview of the layout of the rest of the thesis.

1.6 Motivation for Research

The above discussion gives an overview of peer-to-peer networking and identifies some important aspects of p2p design and some of the pitfalls that arise. P2P systems have opened a door to new interesting and exciting possibilities for building networks for end-users and has given public Internet users the ability to serve files and data of their own choosing. The work presented in this thesis is intended to tackle the problem of providing easy access to quality texts from more prestigious sources that may become hidden in a barrage of search results performed on the web. Users will be placed into communities of other "like-minded" users to share files. An interesting point to note is that often texts and work especially within the research community although never making it to a high profile status within its area are more than worthy of some form of publication. P2P gives the individual power to publish with ease. Also, it is often the case that research is done in parallel and never has the opportunity to cross paths, two seemingly distinct areas are in fact quite similar and would be of benefit to both parties. The system developed as part of this work is designed to provide users with a framework whereby researchers can meet and join together into communities to share content that would be of interest to the community. This type of framework creates an interesting situation for identifying parallels between research, which through the web or other means may never have been discovered. This framework is realised in the form of a peer-to-peer network. This is not a new idea but one that hasn't been around since the first incarnation of the Internet.

Utilising the routing capabilities of some of the newest routing algorithms to emerge from the recent wave of p2p research, the proposed system includes an indexing service, which will facilitate the creation of virtual *rendezvous* points. These rendezvous

points serve as points of contact for users with similar interests (manifested by shared keywords). Users will be described by the content they store. Using Information Retrieval and clustering techniques users can be grouped together to form communities that are content sensitive in nature because a community only exists because of the type of files that are in the system. Documents are placed loosely within the system, positioned near other documents that are deemed similar instead of being grouped together under one heading. This placement of texts facilitates the discovery of new parallels between subjects and users. The system is built to serve as a distributed archive of research papers. The system is also designed to improve search quality of service (QoS) compared to current p2p file sharing applications. Search requests result in a comprehensive set of relevant documents being returned as searching will be based on semantic meaning rather than literal matching.

1.7 Thesis Outline

Before continuing a brief outline of the thesis is provided for the reader. **Chapter two** introduces a new breed of p2p routing algorithms that are both scalable and robust in nature. **Chapter three** then introduces the subject of Information Retrieval and the techniques used in information retrieval that may be adapted to a p2p system to discover inter-document relationships. **Chapter four** discusses the design evolution of the system and serves as the recipe for the various ingredients of the system that are introduced to the reader in chapters two and three, Chapter four also introduces the novel idea of *node-vectors*. Node-vectors are user identifiers that are rich in semantics and used to assess a measure of "like-mindedness" between peers. **Chapters five** and **six** present the software design process, implementation and a results and evaluation discussion of the research idea in comparison to some of the systems introduced at the beginning of this Chapter. They also discuss what advantages are gained through employing the ingredients used to realise the system in contrast to the other p2p systems surveyed as part of this work. Finally **Chapter seven** contains concluding remarks.

Chapter 2

Distributed Hash Tables

This chapter discusses the emergence of a new generation of peer-to-peer routing systems based on the distributed hash-table (DHT) concept. These recent additions to p2p have generated a lot of research interest in the new technology. Distributed hash-tables act much in the way the name suggests. It is worth first looking at what a hash-table [29] is in order to be able to appreciate the concept of distributed hash-tables.

2.1 Hash –Tables

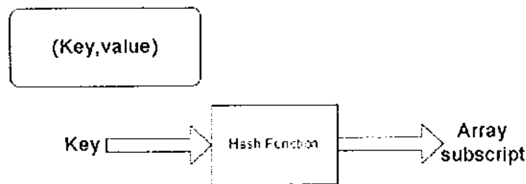

Figure 2.1A. Given a key-value pair the key is applied to a hash function to produce an array subscript. This subscript is used to map the value to an array position as in figure 2.1.B.

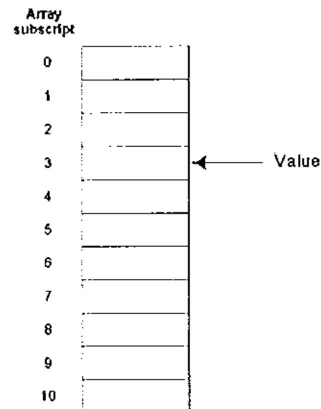

Figure 2.1.B. Given a hash function output of 3 the value will be inserted into that position within the array

A hash-table is a particular way in which data may be structured and stored. Hash-tables achieve the constant insert/search time performance of arrays even without suitable array subscripts. Constant time performance relates to the number of steps needed to retrieve an object from the table. Although this is not strictly true for worst-case scenarios such as when multiple collisions occur (discussed below) the probability of this happening is very small and so because the best and average cases are O(1) ($O(1)$ = constant function using Big-O notation) it is said that hash tables do in general achieve constant time performance. In order to store a value, the value must be associated with a key. The idea is to map a key and an array subscript using some function. Hash-tables use a *hash function* that takes keys as arguments and returns subscripts of some appropriately sized array as values (see figure 2.1.A). When inserting an item into a hash-table, the hash function is applied to the key and then the key and the value are recorded at the position according to the array subscripts that the hash function generates (see figure 2.1.B); to recover the object again, the key is again applied to the hash function and the value will be situated in the array at the location indicated by the result. Hash-tables can be viewed as a dictionary in which keys are mapped to array positions by a hash function.

Given an array of fixed size and a very large number of possible keys there cannot be a one-to-one assignment. Inevitably the assignment of the different keys to the same array subscript will occur, this is known as a *collision*. Hash-tables must be built to deal with this inevitability and be able to recover from it [30]. Obviously this requires extra computational time. This fact would therefore lead to the conclusion that in reducing the number of collisions one is improving the hash-table performance. "Randomisation" of keys is an approach that is often used to cut down on collisions [31]. "Randomisation" means that the values that the hash function produces do not conform to any pattern that may characterise the keys. This property of hash functions has been exploited to achieve load balancing in distributed hash-table systems; this will be discussed later in this chapter.

2.1.1 Distributing the Hash Table

Some useful attributes of hash-tables have been noted in the above discussion. One very important attribute associated with hash-tables is that with appropriate care, access to an arbitrary element within a hash-table can be provided in roughly constant time i.e. hash-tables have approximately, an insert time $= O(1)$ and a search time =

$O(1)$. This means that hash-tables have constant insert and search times. Switching the focus to the problem of locating stored objects in a distributed environment. Building a distributed system that exhibits qualities such as constant insert times and search times of hash-tables is a very worthwhile venture. This point leads us to the discussion of distributed hash tables.

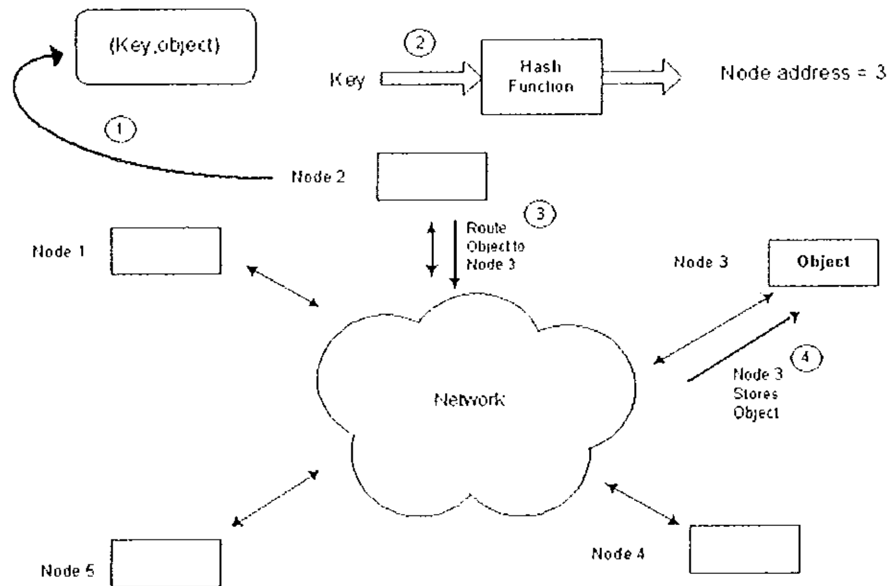

Figure 2.2. Operation of a Distributed Hash-Table. Nodes correspond to computers connected to a network. Node 2 wishes to store an object within the system. An Object and a key are provided (1). The key is "hashed" and an address for the object to be stored is produced (2). The object is then forwarded to the node corresponding to the address (3) When the object arrives at the destination node it is stored (4).

Within a distributed hash-table system, a network is formed that consists of many nodes that are organised in a p2p configuration. These nodes act as storage locations for the objects that are placed in the "hash-table" and may be viewed as the various memory locations in the array of a traditional hash-table. DHT systems provide a "put" and a "get" function. In a distributed hash-table objects are assigned keys and then mapped to nodes or storage locations using a hash function (see fig 2.2). Objects are placed into the network by generating a cryptographic hash of, for example their textual name. The hashing algorithm generates an output, typically a 160-bit digit number that is associated with the object. Nodes are assigned Identifiers from the same numbering space; these node identifiers are analogous to the array subscripts in a locally

stored hash-table. This technique also finds application in message authentication, *bucket hashing* is one method used to optimise the technique for this application [32].

Only the number of unique identifiers available limits the number of nodes that may join the network. Given the order of this number this limitation is unlikely to ever be reached. Once each node within the network has an identifier associated with it; the newly introduced object can be forwarded to and then stored on the node whose identifier is related to the mapping from the key of the object being stored (which was produced from the hashing function). This is the "put" functionality. Storage capacity within the network is determined by the amount of memory each node allocates and the number of nodes connected to the network. To retrieve the object again one must know the key of the object they wish to retrieve. The key is again "hashed" to determine the location of the object. This is the "get" functionality. Removal of objects from the network is very much implementation dependant. DHT layers mainly provide the put and get function. Higher layers such as the application layer built on top of the DHT substrate provide mechanisms for the removal of objects introduced to the system. What the DHT layer does provide is a means of accessing or referencing the object. As this topic is very much implementation specific and this is a more general discussion the reader is referred to section 2.5.1 for an example application.

There are a number of implementations of these types of p2p systems. These implementations have many similarities but enough differences to warrant a closer examination. The rest of this chapter is devoted to the discussion of these systems as DHTs form a major part of this work and so these other distributed hashing solutions provide a useful reference.

2.2 A New Beginning for P2P

In a 1997 paper by Plaxton et al [21] a simple randomised algorithm for accessing shared objects was described. Plaxton's algorithm satisfies each access request with a nearby copy. The algorithm was based on a novel mechanism to maintain and distribute information about object locations.

Plaxton's technique allows a node to have multiple roles. A node can be a server (store objects), a message router (assist in the forwarding of messages to their destination), and a client (requesting a lookup for a certain object). A term that has become commonplace in p2p for this type of node is a *servent*. Servents and objects

within the network are assigned identifiers that are independent of location or semantic properties of either servent or object. This, as we shall see, achieves load balancing within the network. The identifiers are in the form of random fixed-length bit-sequences with a common base. The entry location of an object or node into the system is determined by the numeric value of its identifier. The determination of these identifiers can be achieved by using the output of *hashing algorithms* such as SHA-1 [33]. A hashing algorithm such as SHA-1 is nonreversible, collision-resistant, and has a good *avalanche effect*. The avalanche effect of hashing algorithms means that given two very similar strings, two very different and non-numerically close hash codes will be produced. This is the randomisation property of hash-tables that was discussed previously. It is this property that is used to achieve load balancing, because nodes and objects get placed randomly within the network.

Plaxton's work involved designing a simple randomised *access scheme*. The term access scheme refers to a set of algorithms for read, insert and delete operations. The goal was to produce an access scheme that exploited locality and distributed control information [21]. This distribution of control information achieves a low overhead in memory because nodes only need to know about a fraction of the other nodes within the network making it unnecessary to store large index tables containing network information.

Routing

To get a better feel for Plaxton's access scheme it is useful to examine its operation in more detail. Examination of the read operation gives a good understanding of how messages are routed throughout the network; the reason for focusing on the read operation is that this operation will be the key element of the system that has been developed. Before fully examining the read operation it is necessary to define a few terms. The auxiliary memory of each node is partitioned into two parts, the *neighbour table* and the *pointer list*.

Neighbour table:

Each node stores a neighbour table. This table is populated with the IP-addresses [34] of neighbouring nodes within the network. The term neighbouring nodes refers to nodes whose identifiers are numerically close to the identifier of the node storing the neighbour table [21].

Pointer List:

Each node maintains a pointer list. This pointer list contains pointers or the locations of copies of the objects within the network. The pointer list of x may only be updated as a result of insert and delete operations [21]

Read

Consider a node x attempting to read an object A. Object A has associated with it a unique alphanumeric object identifier. Objects inserted into the network are forwarded to a node whose identifier is numerically closest to that of the objects. The node storing the object then becomes the root node of that object A. Now looking at what happens when a read request is generated at node x. Node x first checks local memory for the object. Should the object not be found the read request is forwarded to a neighbouring node whose address is contained within neighbour table of node x. The choice of neighbour to which the read request will be forwarded is determined by correcting the first few digits of the objects identifier. In this way the read request operation is forwarded to the node within the neighbour table whose identifier is numerically closest to the identifier of object A. In effect the read request re-traces the steps taken to insert the object into the network so that the read request will end up at the root node of the object being read.

For the purpose of illustration of the basic principles of the read operation let us assume that neighbour lists are consistent. In this case the routing method of a read operation guarantees that any object A existing within the network will be found within at most $Log_b N$ logical hops, in a system with an N size Id space and Ids using base b [21].

Plaxtons routing system provides several desirable properties. Due to the fact that routing requires that nodes only match a certain number of digits before sending the message, link breaks or server failures can be routed around by using different route paths made up of connected nodes. This is possible because a node can choose a different route path for the message by matching a different node in its routing table with a similar identifier to forward the message to. This means that the routing is robust to node failure along the path to the root node (the root node is the node hosting the desired object or the destination of the read message), see figure 2.3.

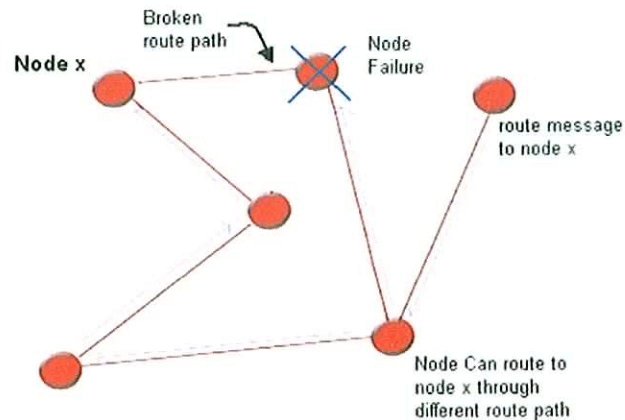

Figure 2.3. Plaxton et al's routing scheme is robust because nodes can route around node failures by choosing a different route path

However the use of a root node for an object does leave the access scheme with a single point of failure. This means that should the root node for an object fail the object will become inaccessible. The access scheme is scalable due to the fact that all routing is done using locally available data i.e. the neighbour lists. There is no centralised control so the only place bottlenecking can occur is at the root node should the object it's hosting become popular. The routing of a message is achieved quickly throughout the network. This is because the number of route hops drop geometrically with each additional hop.

The valuable and interesting part of Plaxton et al's work on p2p systems is that the Plaxton data structure allows messages to locate objects and route to them across an arbitrarily sized network, while using a small fixed sized routing map at each hop. This means that the algorithm is scalable. Scalability is a very desirable property in p2p, the lack of scalability of many p2p systems is the cause of many problems within p2p networks, and this especially comes into play as a limiting factor in the provision of good "quality of service" for search requests. As we have seen above the algorithm is also robust to routing around node failures. The major problem with Plaxton's algorithm is that it assumes a static node membership. Applying this algorithm to a situation where nodes are constantly joining and leaving presents significant problems. In order to achieve a unique mapping between document identifiers and root nodes the Plaxton scheme requires global knowledge at the time of creation. This global knowledge greatly complicates the process of nodes joining and leaving the network. The Plaxton scheme also incorporates a single point of failure. The use of a root node

for an object means that this root node becomes the node that every other node relies on to provide the object's location information. As a result, Plaxton's work does not directly translate to a dynamic p2p system; the algorithm is in need of modification to be able to be used in a p2p sense.

2.3 Developing a Suitable Routing Substrate for P2P

Described as "a new generation of p2p systems", as relative new comers to the world of p2p, DHTs have generated much enthusiasm and some hope of solving some of the principle problems that face p2p systems. As has been noted in chapter one, peer-to-peer networks are one of the fastest growing technologies in computing. In developing these systems two major problems have been noted, scalability and robustness. Scalability problems have plagued systems such as Napster [3] and Gnutella [4]; a system's ability to scale well can be the determining factor leading to the success of a system or its failure. Robustness is an essential ingredient for all practical p2p systems; the concept of p2p almost implies it, yet a closer look at Napster reveals a central point of failure. In developing a p2p system one has to take these and many more issues into account. Distributed hash-tables are one attempt to conquer these fundamental issues in p2p.

DHT overlay network implementations such as Pastry [22] and Tapestry [23] have produced implementations that adhere to the principles of p2p, decentralisation, robustness and scalability. These systems may be used as the basis for functional p2p systems. They provide a routing substrate; a mechanism that efficiently locates objects within a certain number of routing hops. DHTs in themselves do not present a fully scalable robust p2p system but they do provide a mechanism for such systems to be built. All of the systems described below take a key as an input. In response to this key the routing layer forwards a message using the key as the route destination for the message. When a node receives a message with a key that it is not responsible for, that node then forwards the message to a node that makes most progress in resolving the final destination of the message. Determining which node makes most progress is a fundamental design of DHT systems and is discussed in detail below. Also known as overlay networks or structured p2p overlay networks, these systems bring order to p2p. This order means that lookups can be efficiently directed to their destination, thus cutting down on Internet and network traffic

The following sections look at the various projects mentioned above. These systems have many similarities and all derive their initial inspiration from the access scheme developed by Plaxton et al. Each system has adapted the Plaxton access scheme to work in a dynamic environment where nodes constantly join and leave the network.

2.3.1 Pastry

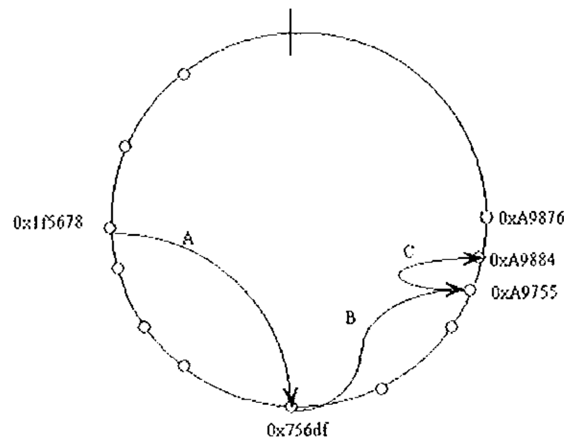

Figure 2.4. Pastry routes messages to nodes whose nodeIds are progressively closer to the message key.

Pastry nodes are organised around a circular id space. Each node within the Pastry network is assigned a 128-bit unique identifier that is generated typically from the cryptographic hash of, for example, its IP address and users name. For example, using the SHA-1 hashing algorithm a string such as "*computer science*" will produce the hash code "*0bbb843c75b8cb93ceb9d5594e208668484448ee*". Pastry has the ability to route messages between nodes when given a message key (see figure 2.4). The message is routed to the node whose *nodeId* is numerically closest to the message's key. In order to route efficiently each Pastry node maintains a routing table. A node's routing table is organized into $128/2^b$ rows and 2^b columns, b is a configuration parameter. The 2^b entries in row r of the routing table contain the IP addresses of nodes whose nodeIds share the first r digits with the current node's nodeId. A routing table entry is left empty if no node with the appropriate nodeId prefix is known. Each node also maintains a neighbor set (called a "leaf set"). The leaf set is the set of l nodes with nodeIds that are numerically closest to the present node's nodeId, with $l/2$ larger and $l/2$ smaller nodeIds than the current nodes id. The value of l is constant for all nodes in the

overlay, with a typical value of approximately $[8 * \log_{2^b} N]$, where N is the number of expected nodes in the overlay. The leaf set ensures reliable message delivery and is used to store replicas of application objects.

2.3.2 Tapestry

Tapestry also takes its roots from Plaxton et al's work. Again it is an augmented framework that tackles some of the failings of the Plaxton scheme mentioned above. Tapestry has also adapted the access scheme to work in a dynamic environment where nodes are constantly joining and leaving the network.

At Tapestry's core is a set of routing algorithms similar to the ones developed by Plaxton. Tapestry bears much similarity to Pastry.

Tapestry implements a routing algorithm called *surrogate routing*. This algorithm is deterministic; meaning that when routing to a node that is the root node of an object (i.e. the node storing the object because it has the closest numerical nodeId to the objects Id) it is possible to reach the same point or node from any other point within the network. Within surrogate routing the root node for an object is found by matching the objects Id to a node Id. It is likely that a node exactly matching the Object Id will not in fact exist within the network. This is due to the large Id space and the fact that it is not necessary to find an exact match. Tapestry attempts to route the object to a node that in most cases is non-existent. The object will be assigned to a node within the routing process whereby the last node reached would in fact be the nearest existing neighbour node to the non existing root node. This existing node now becomes the root node.

Surrogate routing requires a few additional hops compared to systems based on Plaxton's algorithm. It was shown in [35] however that adaptable routing algorithm with Tapestry has minimal routing overhead compared to the static global routing algorithm of Plaxton.

2.3.3 CAN

CAN uses a d-dimensional Cartesian coordinate space (for some fixed d) to implement a distributed hash table that maps keys onto values. Each node maintains $O(d)$ state, and the lookup cost is $O(dN_{1=d})$. In the CAN model, nodes are mapped onto the d-dimensional coordinate space on top of TCP/IP [36] in a way analogous to the

assignment of IDs in Tapestry and Pastry. The space is divided up into d-dimensional blocks based on each servent's density and load information, where each block keeps information on its immediate neighbours. Because addresses are points inside the coordinate space, each node simply routes to the neighbour whose coordinates are the closest towards the destination coordinate. Location of objects is achieved by the object server pushing copies of location information back in the direction of incoming queries. There are several key differences between CAN and Tapestry and Pastry. Tapestry's hierarchical overlay structure and high fan out at each node results in paths from different sources to a single destination that converge quickly. Consequently, compared to CAN, queries for local objects converge much faster to cached location information. CAN assumes objects are *immutable*, and must be reinserted once they change their values. CAN, like Chord, does not attempt to approximate real network distances in their topology construction unlike Tapestry and Pastry. As a result, logical distances in CAN routing can be arbitrarily expensive, and a hop between neighbours can involve long trips in the underlying IP network. The main advantage a CAN has is that because of the simplicity of the node addition algorithm, it can better adapt to dynamically changing environments.

CAN focuses on providing distributed hash-table functionality where as Pastry and Tapestry are more geared towards routing and locating.

2.3.4 Chord

Chord is an efficient distributed lookup system based on consistent hashing [25]. It provides, like all the other DHT systems discussed, a flexible high performance look up scheme upon which the main features of a peer-to-peer system can be built. Like Pastry, Chord also uses a one-dimensional circular key space. For Chord, in a network consisting of n nodes each node maintains information about only $O(\log n)$ other nodes, lookups require $O(\log 2n)$ messages.

The focus is on providing hash-table-like functionality of resolving key-value pairs. For a namespace defined as a sequence of m bits, a node keeps at most m pointers to nodes which follow it in the namespace by 2^1, 2^2, and so on, up to 2^{m-1}, modulo 2^m. The *ith* entry in node n s routing table contains the first node that succeeds n by at least 2^{i-1}, in the namespace. Each key is stored on the first node whose identifier is equal to or immediately follows it in the namespace. Chord provides similar

logarithmic storage and logarithmic logical hop limits as Tapestry, but provides weaker guarantees about worst-case performance. The main distinction worthy of note is that there is no natural correlation between overlay namespace (i.e. the numerical space that Ids are assigned from) distance and network distance in the underlying network, opening the possibility of extremely long physical routes for every close logical hop. This problem is partially alleviated by the use of heuristics.

2.4 Discussion of Distributed Hash Table Implementations

It has been noted that a common theme of discussion regarding these systems is to compare them and contrast them in a "which is better" way. However in [37] it is suggested that a more useful way of thinking would be to look at their similarities and their strengths rather than their failings in regard to each other. It would then be possible to suggest some improvements upon which one could build upon the strengths of all systems.

In 2003 the designers of the DHT systems discussed produced a paper [38] detailing the beginning of their work to define a common API for structured peer-to-peer overlays. Their work attempts to define the fundamental abstractions provided by the system and combine them together so to enable designers of p2p systems to be able to interchange the substrates to evaluate them and thus not tying their system into one substrate. The difference and similarities of the above DHT systems are discussed in more detail in chapter four.

2.5 Projects Using Distributed Hash Tables

Structured p2p substrates have proved useful in building such systems as global storage facilities, including PAST [26], CFS [39] and Oceanstore [28]. In particular PAST, as we shall see has a lot in common with the system designed in the work for this thesis. PAST is built on top of Pastry and uses the power of Pastry to route files entered into the system to a particular point, given a file key.

2.5.1 PAST

PAST is a large scale Internet based global storage facility. PASTs use of DHTs to store files at nodes with similar nodeIds to the files Id is very similar to the method in which our system builds an indexing service (see section 4.3.1). Due to this similarity it is worth considering PAST in more detail. PAST is built upon Pastry's location and

routing substrate and is comprised of nodes that contribute to the routing of client messages to insert and retrieve files. A node may also contribute to the storage space of the system by allowing files within the system to be stored on part of its hard drive. Files are replicated among nodes of similar Ids and due to Pastry's random placing of nodes this means that files will be replicated in diverse geographical locations.

At the time of insertion, files are assigned a unique file identifier that is taken from the same name space as the Pastry node Ids. This fileId is also created in much the same way, using a hashing algorithm and the file name plus for instance the name of the papers author as the input into the hashing function. This means that files within the system are immutable because files cannot be inserted multiple times with the same fileId. Each file that is inserted in to the PAST system is assigned a 160-bit fileId corresponding to the cryptographic hash of the file's textual name, the owner's public key and a random salt. A file certificate is then produced, this certificate contains the fileId and the replication factor *k*; the value of *k* determines on how many nodes the file must be stored upon. The certificate also contains a cryptographic hash of the file's contents. Upon the actual insertion of the file into the system, the Pastry layer routes the file to the k nodes whose identifiers are numerically closest to the 128 most significant bits of the fileId. Each of these nodes will store a copy of the file. In order to "lookup" this file, the fileId is firstly required. The lookup request message is routed throughout the Pastry ring using the fileId as the message key to the live node again whose nodeId is numerically closest to the message key or fileId. Presuming that any of the k nodes storing the file related to the fileId are live on the network the file will be located within the network and may be retrieved.

PAST does not support a *delete* operation. Instead, the owner of a file may *reclaim* the storage associated with a file, which does not guarantee that the file is no longer available.

2.6 DHTs versus the "Napsters" of P2P

The subject of keyword searching is of particular importance if DHT systems are to stand alongside more mainstream p2p systems such as Gnutella or Kazaa [40]. However, DHTs currently provide only put and get functions. Introducing keyword-searching capabilities into these systems is likely to render them a more powerful tool for file sharing than is currently available.

2.6.1 Limitations of DHTs

DHTs provide scalable exact match lookups, these lookups lack any kind of "intelligent" matching such as is available through the use of modern query languages. In [41] the authors have pointed out two serious limitations of Peer-to-peer networks. Poor Scaling and impoverished query languages. Napster and Gnutella are two networks that have poor scaling properties; they also provide a basic query language format [42]. DHTs largely solve the first problem of poor scaling presented in [41] as they provide a set of lookup algorithms that are extremely scalable. However because DHTs only support exact lookups they fall down on the second problem presented.

DHTs are a substrate to provide a system with scalable and efficient lookup and routing algorithms. They do not constitute a fully operational peer-to-peer system. These systems may however be used to form the foundation for functional p2p systems. They provide a routing substrate; a mechanism that efficiently locates objects within a certain number of routing hops. A more advantageous approach would be to build a system that tackles the second problem presented in [41] by providing not just keyword searching but a system that supports rich search results by finding files that have semantic meaning rather than just literal matching. Achieving this would place these systems well above the popular file sharing systems that currently exist. One promising solution is for the p2p community to look at more advanced search techniques. Information Retrieval is one area of study that deals with this type of work. Information Retrieval seeks to find relevant information from a large corpus, the results of this area of study has obvious advantages for distributed search techniques. In an effort to tackle the problem of weak p2p search techniques a review of Information Retrieval and some techniques that have been developed was undertaken. This is the focus of the next chapter.

Chapter 3

Information Retrieval (IR)

Information Retrieval (IR) is concerned with retrieving relevant information from a large collection of documents or data. Peer-to-peer file sharing systems attempt to allow the user to retrieve files from a large distributed group of storage locations. The challenges of searching for relevant files within this environment have already been presented in the last two chapters. The answers to many p2p systems poor search techniques may lie with the information retrieval community.

IR has traditionally been used in centralised sets of data [43]. In recent years with the increasing use of distributed systems and of course with the increased popularity of p2p systems, distributed IR is becoming not just a more common area of study but also a necessary one [43]. On the World Wide Web designers of search engines have also recognised the need for and the advantages of more intelligent IR techniques [43]. Within p2p environments, search algorithms are somewhat lagging behind other information systems; searches are generally basic and provide poor quality search results to the user [44]. Search engines use web crawlers to search the "deep web" (the deep web is made up of publicly accessible pages that are not indexed by search engines) and with a move to more sophisticated IR techniques are becoming more effective. However, there is still a huge resource of information that is stored on users hard-drives of computers connected to the Internet that is inaccessible to search engines and web crawlers. If the users wish, p2p networks can be used to publish this information to an interested audience; the problem however exists of searching these hard drives. A p2p text file-sharing system built on top of a DHT has many qualities but lacks the IR techniques that are available on many centralised texts resources. The trend towards distributed information systems has also sparked the need for similar text retrieval systems. P2P file-sharing techniques require text retrieval systems that can work in a distributed environment. There is work underway in this area. A good example of this is InfraSearch [15]. Information Retrieval provides a way of finding

inter-document relationships that are more accurate than through matching search words with titles of texts alone.

Before tackling the issue of searching data stores shared over a p2p system, it is necessary to consider the more general problem of locating the best storage locations to search in this distributed environment. The work in the area of distributed IR is closely related to the work on multi-agent systems [45]. In a pure p2p system where the clients themselves act as servers and so as the storage locations, the problem is effectively locating the 'best' peers or clients to search. Information retrieval can be used in identifying peers that are most likely to be hosting the relevant data.

A good entry point for this discussion is to firstly introduce the idea of a *Content Network*. A content network is an overlay IP network that supports *content routing*. Content routing means that messages are routed based on their content rather than their IP-address. It is intended in this research to use the idea of content routing for the forwarding of search queries. Forwarding of queries based on the content it is created to find, as will be shown gives a much stronger guarantee of reaching an appropriate site containing desired texts. This can be achieved by creating a semantically rich identifier for the search query. Also, this type of search technique when deployed over a network organised based on communities of nodes hosting similar content cuts down on arbitrary hops from node to node that occur with Gnutella. This type of system can be categorised as a p2p content network. In recent years many types of content networks have been developed, including content p2p networks. PAST, which has already been discussed in section 2.5.1 is one such development that can be viewed as a content network. Content networks can be classified into many different types. [46] This gives taxonomy to the different types of content networks. Further discussions on the design of semantically rich identifiers and organising a network into content sensitive communities are given in the next chapter. For now it is necessary to introduce to the reader the subject of Information Retrieval and IR methods that are used to accomplish this task.

3.1 An Introduction to IR

Like Peer-to-peer systems, IR techniques continue to attract an increasing amount of attention [47]. IR dates back to well before the advent of computers. Computers were seen as a tool for facilitating information retrieval rather than the other

way around. IR techniques are more important than ever now. There has been an enormous increase in the number of text databases available on-line, it is also worth noting that a study in 1989 revealed that approximately 9,600 different periodicals are published in the United States each year, the amount of information available doubles every five years and the amount of books in a library doubles every fourteen years [48]. A more recent survey also shows similar trends of activity on the web with every 24 hours 4.3 million new web pages being created [49]. There are many consequences resulting from this level of publishing activity; one is a need for better techniques to access the vast quantities of information. Consequently there has been a strong resurgence of interest in the research done in the area of IR. Again, this bears similarity to the situation in peer-to-peer networking where the concept is relatively old but its importance is only really being realised by a wider community in recent years. Both areas of study have been placed into the "mainstream" research hot-topics category in the hope that they can be used to solve some of the problems faced by modern computing. In a world where the quantity of information available can simply be described as vast, the need for a way to retrieve the right information from a huge source highlights the importance of information retrieval techniques. It is interesting that even though Information Retrieval has been around for some time the problem of effective retrieval remains largely unsolved and is very much an active area of research within a centralised environment.

3.1.1 Current IR Systems

Numerous IR systems are deployed in various contexts around the world at present. The World Wide Web is one information source that heavily relies on IR techniques to locate information. IR and the Internet were previously called Advanced Information Access. The Internet and the World Wide Web rely on search engines to allow users to find web sites and documents contained in the "deep web". Some of the better-known search engines are Google [50] and Yahoo [51]. Search engines like these use various techniques to match content or web pages to search queries. Web pages mostly contain semi-structured and dynamic information. The key problem is how to embed knowledge into information mining algorithms. This will be discussed in more detail later on. There are several advanced methods for Web information mining such as Syntax analysis [52] Metadata-based search using RDF [53] (Resource Description Framework) and KPS: Keyword, Pattern, Sample search techniques [54]. However, the

most common and basic technique is called *literal matching* or *Boolean searches*. Literal matching takes words entered in a search query and matches them to titles of documents, document keywords or Meta data associated with documents files and sites available on the Internet. Results for searches are then ranked in order of importance relating to the match with the search query. Literal matching is the technique that is used by many p2p systems as a search solution and as we shall see suffers from many inaccuracies. A goal of this research has been the improvement of these inaccuracies through the use of more advanced IR techniques.

3.1.2 IR in P2P Systems

Many p2p systems such as Gnutella use literal matching to find relevant files within their systems. This literal matching suffers from numerous inaccuracies. In the case where a title is used to match words entered into a search query the quality of the results can be impaired due to synonyms of words used in titles and by misspelling of titles. There are a number of factors that add to the ability of a p2p system to retrieve data. Gnutella is described as a *loose guarantee* system. A p2p system that is described as a loose guarantee system means that search queries or the quality of service (QoS) to search requests to a particular query are not guaranteed to return a true representation of the actual files within the system. The QoS of a system such as Gnutella as discussed in section 1.4.1 gets worse as the number of nodes in the system increases. Systems that employ DHTs (see chapter two) and JXTA [20] are *strong guarantee systems*. DHTs assign a unique identifier to each file and the file may be retrieved by any user who knows the file's Id. DHTs do not however provide a mechanism for partial-match lookup capability, so although loose guarantee systems provide a lower QoS to search requests they are in general more open to loosely stated queries where the user is not fully sure of the particular files they are looking for. It is important to note the weaknesses of these two types of systems so as to suggest a solution to the problems. As has been stated, keyword or Boolean methods are the most popular way to search loose guarantee systems. These types of searches add only more inaccuracies to the set of all ready loosely guaranteed search results presented to users. There are however much more accurate methods for the comparison of documents to search queries, these state of the art IR techniques seek to improve on the inaccuracies found with literal matches. Noting that the two main pitfalls with Gnutella are its lack of organisation and its

impoverished query language it is now useful to look at some methods to over come these problems.

3.1.3 State-of-the-art IR Techniques

Several methods for matching queries to files have been mentioned with literal matching being the most popular. One set of promising alternatives to these approaches are *vector-space* approaches. Vector space approaches to IR allow the user to search for concepts rather than specific words and rank the results of the search according to their relative similarity to the query. This type of search is becoming more and more important as large, heterogeneous collections will become more and more difficult to search due to the sheer volume of unranked documents returned in response to a query. Vector-space approaches avoid the problem of synonyms and avoid categorising data under one topic because the method for identifying data represents all of the data and not just meta-data or keywords. The rest of this chapter focuses on vector space approaches to information retrieval as it is proposed in this chapter to use these techniques to solve the problem of organising peers and documents into communities and the retrieval of these documents from these communities.

3.2 Vector Space Approaches

As discussed above literal matching and textual names or keywords for the retrieval of documents has many inaccuracies and problems associated with it. A more accurate approach to representing documents by headings or keywords is to develop an identifier for a document that has *semantic meaning*. An identifier with semantic meaning means that the value of the id tells something about the subject of the document it represents. Documents and their identifiers can be compared to a search query or other documents based on this semantically rich identifier. A method of developing a semantically rich identifier is through *Vector Space Modelling*. There is an effort to add semantics to the web and create what has been called the *semantic web*. "*The semantic web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation*". -- [55]. This effort stems from similar problems faced in this work; the semantic web is based on the Resource Description Framework [53]. Here it is proposed to use vector space modelling to achieve a semantic p2p system. Vector-space models were developed to eliminate many of the problems associated with exact, lexical matching

techniques. In particular, since words often have multiple meanings (polysemy), it is difficult for a lexical matching technique to differentiate between two documents that share a given word, but use it differently, without understanding the context in which the word was used. Also, since there are many ways to describe a given concept (synonomy), related documents may not use the same terminology to describe their shared concepts. One area where this type of misinterpretation is of particularly importance is for medical diagnosis, in an effort to tackle this problem giant complex lexical systems have been developed over the last 20 years e.g. SNOMED [56], GALEN [57], ICD10 [58], CEN [59].

The effects of polysemy and synonomy mean that a query using the terminology of one document will not retrieve other related documents. In order to achieve a semantically rich identifier that avoids this problem the first task when using vector space modelling techniques is to generate a document identifier or representative that captures the semantics of the data being represented. This is the subject of the next section.

3.2.1 Generating Document Representations

An obvious but important observation to make about documents is that a document is 'about' a topic(s). The evidence to support whether a document is in fact about a topic or not lies in the words of a document. Hans Peter Luhn is known as the father of Information Retrieval and in one of his early papers [60] stated, *"It is proposed here that the frequency of word occurrence in an article furnishes a useful measurement of word significance"*. This statement implies the same observation made above, that words give evidence to a documents topic(s). Luhn however, goes further to recognise that it is in fact the word frequencies that give true meaning, Luhn's observation gave birth to the technique, which has evolved into a concrete method of generating document representatives. Basically Luhn's assumption was that frequency data could be used to extract words and sentences to represent a document. The next sections describe in greater detail how Luhn's observations can be practically implemented.

3.2.2 Vector Space Modelling of Documents

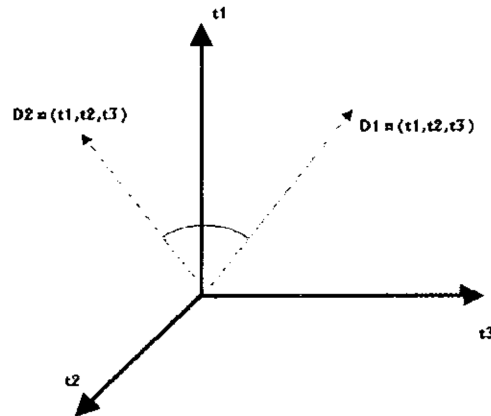

Figure 3.1. Documents can be compared by getting the cosine of the angle between their vector representations

Modelling of a document as a vector is called "document vector space modelling"[61]. In this model each document is considered to be a vector in the *term-space*. A weighted value associated with the number occurrences of a term or word becomes a dimension of the vector. In its simplest form, each document is represented by the *term-frequency* (TF) vector $d_{tf} = (tf_1, tf_2, ..., tf_n)$, where *tfi* is the frequency of the *i* th term in the document. For example, a three-dimensional term space is shown in figure 3.1. There are two documents that are represented by the vectors D1 and D2; each document is plotted within the three-dimensional term space based on the frequency of the three terms occurring within each document. A similarity measure between each document can then be calculated by, for example getting the cosine of the angle between each document. If the angle is zero then the documents are the same and as the cosine of the angle approaches 1 the similarity of two documents decreases. A widely used refinement in this model is to weight each term based on its *inverse document frequency* (IDF) in the document collection. This is commonly done by multiplying the frequency of each term *i* by $\log(\frac{N}{df_i})$, where N is the total number of documents in the collection, and df_i is the number of documents that contain the *i* th term (i.e. document frequency). This leads to the *tf-idf* representation of the document in Eqn 3.1.

$$d_{tfidf} = (tf_1 \log(\frac{N}{df_1}), f2 \log(\frac{N}{df_2}), \ldots\ldots\ldots, f_n \log(\frac{N}{df_n})). \qquad \textit{Eqn 3.1}$$

In [62] it was shown experimentally, that any measure used should be normalised by the length of the document vectors. In order to account for documents of different lengths, the length of each document vector is normalised so that it is of unit length, i.e. $\| d_{tfidf} \|_2 = 1$, where 2 signifies l^2-norm. There are two major similarity metrics for the comparison of documents using vector space modeling [63] [64]. One of them is the angle-based metric that uses for example the cosine function. This is achieved by calculating the cosine of the angle between the vectors as mentioned above. The cosine function is given in Eqn. 3.2. Documents may now be compared and a similarity index can be established between two documents represented by the documents d_i and d_j using this method.

$$\cos(d_i, d_j) = \frac{d_i \bullet d_j}{\| d_i \|_2 * \| d_j \|_2}, \qquad \textit{Eqn 3.2}$$

Where "•" denotes the "dot product" of two vectors. Since the document vectors are of unit length, the above formula simplifies to Eqn 3.3.

$$\cos(d_i, d_j) = d_i \bullet d_j. \qquad \textit{Eqn 3.3}$$

The ability to represent and compare documents using vector space modelling is very useful, enabling inter-document relationships to be determined more accurately than through the use of keyword matching.

The above discussion describes a very useful tool for representing and comparing documents to accurately determine inter-document relationships. This technique takes its fundamental ideas from Luhn's original assumption discussed above, i.e. concerning the significance of word frequency.

3.2.3 Document Preprocessing

There are many efficiency improvements that can be applied to the above model; some of them take their foundation from some of Luhn's other ideas. Luhn also produced two cut off points, an upper and a lower cut off point. Using *Zipf's law* (which states that: "the product of the frequency of use of words and the rank order is approximately constant.") as a null hypothesis Luhn developed a way of excluding non-significant words. This technique is the removal of high frequency words that give no semantic meaning to the document. These words are called *stop words*.

Stop Words

An example of stop words would be, "the", "if", "was". These words give no additional evidence to establish the topic or semantics of the document. There are several stop lists available. One of the most commonly used is the Van Rijsbergen stop list [43]. Words contained in the stop list are removed from the document before the task of representing the document as a vector commences.

Stemming

The stemming of words is a process of removing "commoner morphological and inflexional endings from words in English" [65]. This means that the suffixes of words are removed or that words are reduced to their roots, often the roots of words convey topicality better.

Example of stemming:

Information -> inform

Presidency -> presid

Presiding -> presid

Happiness -> happi

Happily -> happi

Discouragement -> discourag

Battles -> battl

This section has presented a method for generating document representatives that give a document semantic meaning. The following sections discuss how by using these identifiers documents may be grouped or classed together. This method as will be shown can be used in a similar way to group users with compatible research interests together into communities within a p2p system. The reason for doing this is to provide a system whereby searches may be directed at a specific area within the network. This is discussed in more detail later on; first the background to the method is presented.

3.3 Classification of Documents

It has been discussed how a document can be associated with an identifier that has semantic meaning. Using these identifiers, documents can be searched for and compared together using the similarity metric discussed above. This technique has proved to be much more accurate than traditional literal matching approaches [66]. The use of such advanced IR techniques can add greater search capabilities to p2p systems. Documents can be represented by semantically rich identifiers to assess their similarity; using this technique (as will be shown later) it is also possible to represent a user with a semantically rich identifier. It is then possible to assess two users "like-mindedness" to enable the formation of communities using a similarity metric. It is then possible to determine the 'best' placement of users within the network. This idea is discussed in greater detail in chapter four. First however, it is necessary to give background to the idea. The classification of users into communities within the system begins with a method of classifying documents.

During classification, a document is assigned to pre-specified topics. There are many automatic categorisation techniques for texts. Text categorization is a difficult task due to the large number of attributes that are present within a data set. The following discussion deals with a centroid based classification technique; the reason for this is it leads to a way of assessing users "like mindedness"

3.3.1 Centroid Based Classification

As discussed in section 3.2.2 a document may be represented as a vector. In [67], a method called a centroid-based classification is presented. This technique has been shown to out perform other classification techniques such as Naive Bayesian [68], K-nearest-neighbors [69] and C4.5 [70]. The centroid-based scheme allows it to classify a new document based on how closely its behavior matches the behavior of the documents belonging to different topics. The technique calculates a centroid vector that is used to represent the documents of each topic. First of all the documents are represented as a vector using vector space modeling. Given a set of S documents and their calculated vector representations, a centroid vector C can be defined, which is the average vector of the set of document vectors Eqn 3.4.

$$C = \frac{1}{|S|}\sum_{d \in S} d \qquad \textit{Eqn 3.4}$$

The idea behind the centroid-based classification algorithm is extremely simple. For each set of documents belonging to the same topic, the centroid vector of all the vectors representing the documents in the set is computed. The topic of a new document is determined by generating its vector space representation and then computing the similarity between the new document and all centroids this is done using the cosine measure. Finally, based on these similarities, the new document is assigned to the topic corresponding to the most similar centroid. In chapter four it is shown how this technique is used to represent a node and compare nodes based on the content it stores, the point of this is to be able to use a similarity metric to determine if two users are 'similar'.

In addition to representing a node based on the content it stores, a method for grouping similar nodes is needed. One method that could be used is clustering. It will be seen in the next section that clustering techniques provide a method of grouping objects together based on their identifiers. Clustering is introduced in the next section.

3.4 Clustering of Data

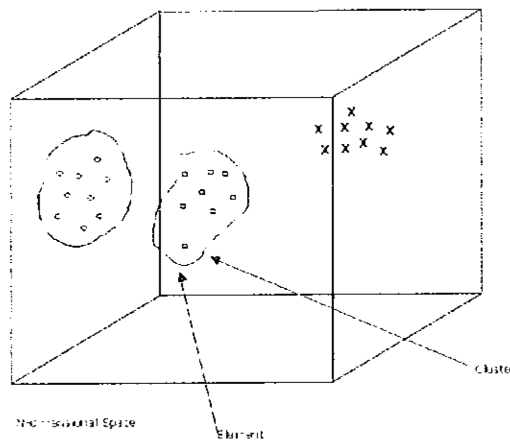

Figure 3.3. Objects may be clustered together based on distance within an N-dimensional space.

Clustering is the unsupervised classification of patterns (observations, data items, or feature vectors) into groups (clusters) [71]. Document clustering was originally investigated as a means of improving the performance of search engines. Since then document clustering or cluster-based techniques have been used in domain identification in such areas as radio news [72] and imaging [73]. Cluster analysis allows the identification of groups, or clusters, of similar objects in multi-dimensional space (see figure 3.3). *Hierarchic clustering* has been put forward for its efficiency and

effectiveness in Information retrieval. There are numerous document-clustering algorithms. *Agglomerative Hierarchical Clustering* (AHC)[74] algorithms appear to be the most commonly used. However these algorithms have proved to be slow when applied to large document sets [75]. *Linear time clustering* algorithms have been suggested as the best candidates for large document sets. These clustering algorithms include the K-means algorithm [76] and the single pass method [77].

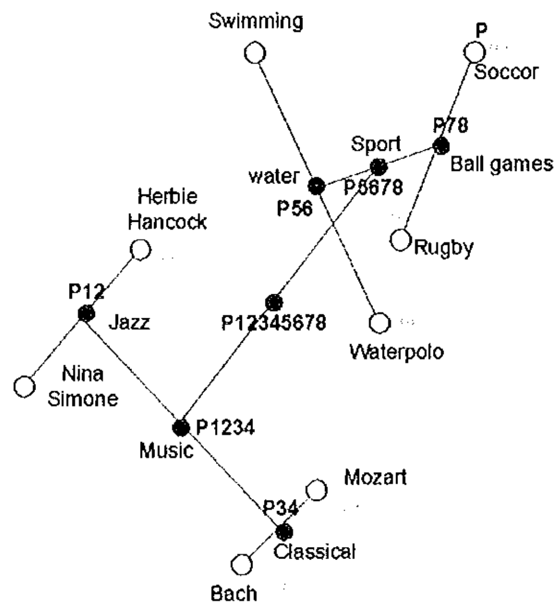

Figure 3.4 Example of a clustering tree.

Given a set of feature vectors such as document vectors discussed above, a clustering tree can be constructed. If an identifier, such as a vector represents a node, a cluster tree containing various nodes can also be constructed. Nodes that are deemed similar, using a similarity index such as the cosine function are placed near each other on the tree and those less similar are positioned farther away (see figure 3.4). There are many methods for constructing clustering trees. [71] Provides a good overview and discusses the pros and cons of each method.

Clustering trees provide a good way of grouping objects such as nodes together based on semantically rich identifiers. Sections of the clustering tree can be considered as communities of similar of "like-minded" users given that the clustering process has been done using a semantically rich identifier. Documents stored by nodes are

somewhat grouped into a topic defined by different sections in the cluster tree. This method is discussed in more detail in chapter four.

3.5 Information Retrieval and P2P

As discussed in section 2.6.1 it was stated that the authors of [41] have pointed out two serious limitations of peer-to-peer networks. These are, inability to scale well to an increasing number of nodes and impoverished query languages such as Boolean searches. It has already been discussed how DHT systems largely solve the first problem. This chapter has presented some ingredients that can be used to tackle the second one. How this is achieved is dealt within the following chapter. It will be shown how some of the techniques described in this chapter may be combined with a DHT (discussed in chapter two) system to create a fully operational peer-to-peer system and thus develop a p2p system that can form content sensitive communities. It is then possible to perform intelligent and efficient searches and thus improving on some of the searching and scalability issues observed in systems like Gnutella.

Chapter 4

Design Concepts and Solutions

The two previous chapters have introduced several "ingredients" and suggested them as tools to solve the problems of scalability and poor searching presented in Chapter one. This chapter gives a full description of how these tools may be used and combined into a fully operational p2p system that tackles the issues of scalability and search QoS. The end of the chapter gives an application overview of how the ingredients and overall final design satisfies the application proposed in Chapter one as well as incorporating design lessons learned from researching other p2p systems.

A full description of the potential design solutions considered is provided for the reader. The initial stages of the project as presented in Chapters one, two and three involved investigating the many p2p systems in operation. The survey of these systems has provided some interesting design challenges to be tackled as part of this research. As was seen in sections 1.3.1 and 1.4.1, Napster and Gnutella both suffer from some fundamental problems. Both applications suffer from poor scaling capabilities. Also, because Napster uses a centralised indexing approach the robustness of the system is compromised.

Chapter two presented an overview of a new generation of p2p algorithms, DHT systems. These systems solve some of the problems faced by many p2p systems. However, despite their evident superiority to Napster and Gnutella they still only provide a routing substrate layer and for instance do not provide any mechanism for keyword searching. DHT systems merely provide *put* and *get* functionality, which places them in a different category to some of the mainstream network file sharing applications. This chapter presents the design concept of a p2p application that utilises the robust and scalable routing algorithm of a DHT system. The application provides a mechanism to perform searches using loose search parameters and thus bringing DHT systems in to direct competition with the mainstream file-sharing application. The discussion shows how using the latest information retrieval techniques discussed in

Chapter three along with state of the art p2p routing algorithms; scalable and robust p2p systems can be built.

4.1 The Substrate Layer

Chapter two introduced the idea of distributed hash-tables (DHTs). Deciding on which DHT substrate to base a system on can be difficult due to the similarities of the various overlays available. However, even though they are similar, some fundamental differences exist. Each system has a different geometry, Tapestry and Pastry exhibit a geometry that is a hybrid of a tree and a ring, Chord is based on a ring and CAN on a hypercube. A ring topology has been shown to be more resilient to node failures and failures of static routes than the other topologies [78]. It has also been shown to be better for routing performance. Tapestry and Pastry out perform Chord. Chord opens up the possibility for long physical distances between hops because it does not attempt to approximate real network distances [79]. Subsequently for this research Pastry was chosen over Tapestry. This is because the node organisation within Pastry better facilitated some of the design concepts such as fuzzy domains that were developed here. This point will become clearer in the coming sections where the design ideas that make use of Pastry's node structure are discussed.

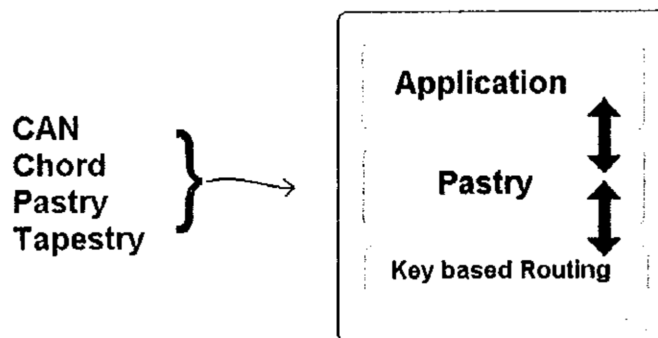

Figure 4.1. A common DHT API allows for the interchange of routing substrates

It was mentioned in the last chapter that there is an ongoing effort to design a global or common API for these DHT systems. In line with this, a central aim of this work is to provide a way in which developers of p2p systems can develop independent of the substrate they are working with. This feature of the system will make it possible for future researchers to interchange substrates to evaluate and compare their performance in a real working system and decide which is the most appropriate for a particular situation. All DHT systems export a key-based routing mechanism that an

application can use to route messages to various nodes. The common API uses this similarity to be able to interchange substrates (see figure 4.1).

4.1.1 Building on Pastry

Pastry is a scalable distributed object location and routing substrate for peer-to-peer applications that are deployed over a wide-area network such as the Internet. It is a self-organising overlay network of nodes that through the assignment of nodeIds, positions nodes within the network. Each node involved in the Pastry network is given a unique identifier known as a nodeId. A Pastry node has the ability upon receiving a message with an appropriate numeric key to efficiently route the message to a node with the nodeId that is numerically closest to the key. Each node in the network keeps a table in which it stores the nodeId of its neighbours, it notifies applications of new node arrivals, node failures and recoveries (see section 2.3.1) [22].

4.2 Potential Solutions

4.2.1 Fuzzy Domains

It was desired to build a system that allowed researchers to share knowledge in a p2p environment. A problem faced by pure peer-to-peer systems (i.e. completely decentralised system) is that of distributed searches. One solution to the searching problem would be to organise the network in communities. This would lead to a situation similar to one encountered in a library where searching for a book becomes easier because each book is categorised and placed within the library in a certain section. Searching is made easier because it is now possible to go directly to a section of the network where the sought object is most likely to be. In a p2p environment this translates into a much more scalable searching mechanism because searches can now be focused on a subset of nodes rather than hopping blindly around the network consuming valuable bandwidth. Gnutella is an example of an application that incorporates a flood-based search technique. Gnutella applications form an unorganised network where nodes are placed at random points within the network. The documents that are stored in the network are also unorganised, as nodes containing objects of a certain 'type' will most likely be connected to or linked with a node containing objects bearing no similarities. It has been observed in [80] that peers in a p2p network mainly search for files similar to files that they are sharing or storing. It therefore makes sense to organise

or group peers that are sharing similar content together. This would mean that the majority of searches would only need to be performed on the community a node has been placed in and thus a flood-search based technique could be more effectively used in a p2p network.

Another point to note is that documents obviously span many different subject categories. One could then argue that a library does not represent true book or document relationships, as to categorise a book under one heading by placing the book in a certain point on a shelf is not entirely accurate. Documents are in their nature fuzzy and sometimes resist categorisation. Library indexing systems and taxonomies attempt to solve this problem by categorising documents within a fixed hierarchy. This again causes problems when two different taxonomies need to be combined as they have two different fixed hierarchies. The search for a solution to this problem is currently very active within the p2p community; one attempt is the Dublin core [81]. The Dublin core initiative attempts to define metadata standards for combining different taxonomies. Given this, it would also be exciting to build a system that has the ability to re-categorise a document or object, as at the time of entry into the system the full subject span of an object may not be known or may not have even been forged. A flexible and dynamic system to incorporate such object relations would certainly be an interesting feature. The system is designed to facilitate the construction and the development of a type of "fuzzy" community where objects fade into various categories and are not constrained into just one.

P2P applications provide an ideal environment to do this due to the dynamic nature of the networks. Furthermore, a DHT implementation brings scalability and robustness to the system. Pastry was chosen for its geometry, which facilitated various design concepts, (see below). In an attempt to discover inter-object relationships in order to loosely categorise them, IR techniques can be employed. The following discussion describes the various ideas that were investigated in order to build such a system using Pastry as the substrate layer, the discussion describes in more detail, specific IR methods that can be used with Pastry in an attempt to achieve the intended system.

4.2.2 Algorithms and Avalanches

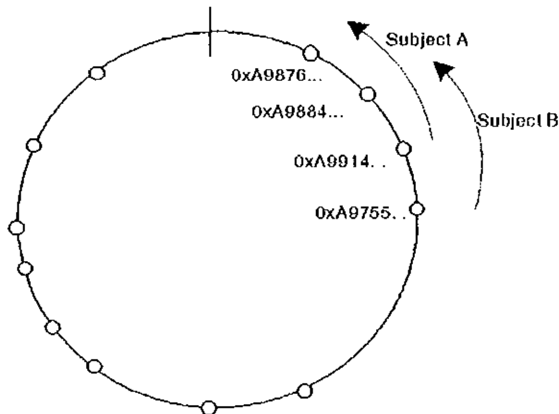

Figure 4.2 Fuzzy domains. Subjects fade into each other around the ring as nodes sharing similar content are placed close together.

Having decided on the routing substrate to be used (Pastry), the first step in implementing fuzzy domains was to investigate the routing substrate fully. The use of hashing algorithms is a central facility within DHT systems. The output of the chosen hashing algorithm with all DHT systems determines where nodes are placed within in the system. The same techniques are used to place objects within systems built on top of DHT systems such as PAST see section 2.5.1. The first task was to investigate hashing algorithms and to examine how they operate. The reason for this was that, if the output of the hashing algorithm could be manipulated to represent something about the input given to the algorithm; the placement of nodes could be controlled in this way.

Hashing algorithms have a feature known as the avalanche effect (see section 2.2). This means that given two similar inputs, the two hash codes produced are numerically different. This provides security to the hashing algorithm as the outputs are random and no pattern may be observed, and so it is difficult to reverse the hash-function and work out the input. The original idea for this part of the work was to find a hashing algorithm with a "bad" avalanche effect. It was considered that this hashing algorithm, when given two similar input strings would produce two numerically close output numbers. Recall that in Pastry as we have seen nodes are organised around a ring. The placement of a node on this ring is determined by its nodeId, which is derived from a hashing algorithm. So nodes that are close together on the ring have numerically close nodeIds. So, if it were possible to manipulate the hash function to produce a number that would change relative to the input string it would then be possible to

produce an identifier with semantic meaning. Nodes with similar interests could associate themselves with a string or sentence that describes their interest areas. These strings would then be hashed along with their IP-address to form their node Ids. Nodes sharing similar content will now have numerically 'close' nodeIds and Pastry's joining mechanism will position them close together on the ring. This is one possible way to form "fuzzy domains" using the Pastry substrate that was developed. In order to accomplish this idea, various Hashing algorithms such as MD5 [82], SHA256 [83] and SHA_1[33] were investigated in order to find an algorithm with a 'bad' avalanche effect. It was found that the ADLER [84] algorithm had the worst avalanche effect but was still not 'bad enough' for the purpose. To continue with this idea a specific hashing algorithm would have to be designed, given the amount of work involved in designing such an algorithm it was decided to investigate other means of forming communities on top of the Pastry substrate. Nevertheless, in the author's opinion, this remains an interesting and worthwhile approach to clustering documents holding similar content.

The basic problem is to find an identifier to represent a document and node that tells something about the content contained in a document or the content the node stores. Another possible solution is to look into other work done in comparing texts of files. One interesting technique to identify whether files are similar is through the use of compression algorithms such as zip technology. This is discussed next.

4.2.3 Zip Technology

Vitanyi et al. have worked on a novel way to identify and categorise music. In their paper [85], zip compression algorithms are used to identify pieces of music with no known composer. This is accomplished by 'zipping' several pieces from a particular composer. The resulting compressed file is then saved. The same particular pieces are again put together and the unknown piece is added to the files to be compressed. The group of files are again compressed. The size of the resulting compressed file is compared with the size of the original compressed file. The new unknown file can be said to be similar if there is little difference in the size of the two compressed files. The zip compression algorithm works by identifying repeated binary sequences. These binary sequences are indexed and replaced by one smaller identifier. This results in a smaller file. When the file is uncompressed the binary sequences are reinserted at every occurrence of the identifier. This attribute of zip compression can be used to assess whether two files are similar. If an unknown file is similar as in the case of two musical

pieces written by the same composer, the resulting compressed file will be small due to the fact that both files contain the same repeated binary sequences. If the music piece is completely different the compressed file will be bigger due to the fact there is less repetition of information.

This technique could also be used for identifying similarities in the content of a particular node compared to another and thus form a link if the similarities satisfy a certain threshold.

4.2.4 Vector Space Modelling

As was seen in chapter three, documents may be represented as vectors and inter document relationships may be determined by using a similarity metric such as the cosine of the angle between document vectors. This method of representing and comparing documents provided a much easier way of representing nodes, this is because it is possible to create centroids representing average content. Using the zip method there was no obvious way to do this. For that reason it was decided to use vector space modelling techniques to group nodes together into communities. The following section describes how this can be done.

4.3 The Final Solution

The final solution for realising the system considered incorporates such ingredients as Pastry and Vector Space Modelling; the following sections explain how these ingredients can be used together to create a p2p system that supports the formation of decentralised, scalable and robust content sensitive communities.

4.3.1 Building a Decentralised indexing service

One of the key elements of the proposed system is the *Pastry routing scheme*. The routing scheme is Pastry's algorithm for forwarding messages around the Pastry ring (described in section 2.3.1). Pastry will provide the system with a scalable, robust routing algorithm that is able to route to any node in $[\log_{2^b} N]$ steps on average, where N is the number of nodes and b is a configuration parameter. As stated previously, Pastry routes messages within the Pastry network to nodes whose nodeIds are closest to the key of the message. Within the system proposed here, it is intended to build a decentralised indexing service where users who are 'interested' in a certain topic may join communities of nodes belonging to like-minded users. In order to facilitate this, a

rendezvous point where nodes can discover others that have similar interests needs to be created. It will be shown in the following section that Pastry provides a routing algorithm that can be used to build such a system without any central control.

4.3.2 Creating indexing nodes

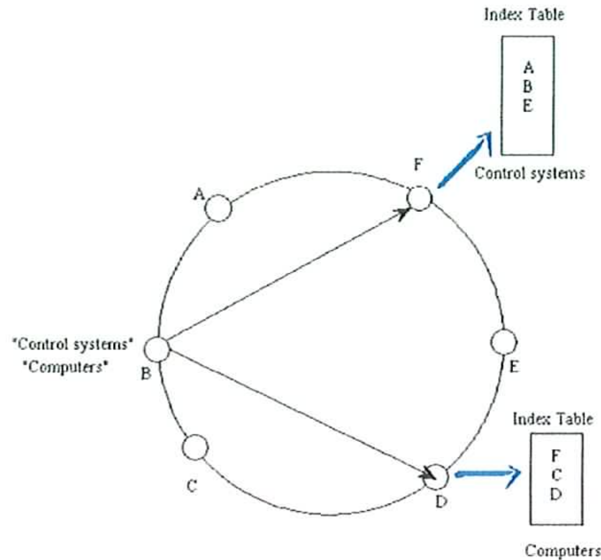

Figure 4.3. Node B routes a register message with 2 message keys that are the cryptographic hash of the keywords "control systems" and "computers" throughout the Pastry ring. Index tables registering all nodes sharing the same keywords are constructed.

Given a string of characters, a hashing algorithm such as the SHA_1 hashing algorithm will produce a 160-bit hash code representing the string. This property forms the basis for the indexing service. Once a node has calculated its unique nodeld it may join the Pastry network. The joining mechanism is provided as a service by Pastry. A node joining the network will have a set of keywords associated with it that best describes the nodes 'topics of interest'. These keywords will serve as the basis for discovering those that share similar content. Each of the keywords provided by the joining node is hashed to get a hash code for each word. These hash codes will be used as message keys so that Pastry can route them around the Pastry ring to the live node whose nodeld is numerically closest to the 128 most significant bits of the 160-bit key. A *registry-message* is constructed; a registry message is an extension of the Pastry message and contains the node details such as its IP-address. The same registry-message is routed several times throughout the network for each keyword. Each registry-message uses the 160-bit codes generated from the hashing of the keywords as keys. When the

registry messages have arrived at the destination nodes, each destination node is required to register the new node (see figure 4.3). This means that every other node stating "controls systems" as their keyword will end up registering at node F. If an index does not exist at node F, one will be created. The node whose nodeId corresponds to the 128 most significant bits of the hashed keyword will now serve as the rendezvous point or indexing node for all other nodes using the same keyword. Nodes will register at the same point for two reasons:

- A hashing algorithm given the same input string will always produce the same output key. Therefore a registry-message will always get the same key for the same keyword.
- Pastry's routing algorithm routes messages to the live node whose nodeId is numerically closest to the message key. Therefore two nodes will always end up registering at the same point once they share a keyword.

The above only holds true of course when the indexing node remains live on the network. To deal with node failures and hence loss of indices, it is proposed to replicate the index table among the indexing nodes *k* nearest neighbours. There are a number of other systems (e.g. PAST) that use the properties of DHT systems for similar purposes. When a file is inserted into PAST, Pastry routes the file to the *k* (*k* is a configuration parameter that determines the number of neighbouring nodes where the index will be replicated) nodes whose node identifiers are numerically closest to the 128 most significant bits of the file identifier (fileId). These nodes then store the file. Other global storage systems that are built on top of DHTs include OceanStore [28] that is built on top of Tapestry, and CFS [39], which uses Chord.

4.3.3 Building Content Sensitive Communities

Now that it has been shown how nodes with similar interests can discover each other in a pure p2p environment, it is necessary to show how the construction of content sensitive communities is achieved. This section describes how nodes are compared based on stored content, the organising of the p2p network into communities. It also shows how this can be viewed as a two-layer network, which can be used as the basis for content sensitive communities.

4.3.4 Comparing Nodes Based on Content Stored

Consider a node storing a set of documents that share the same subject content. It can be said that a node's set of documents can be classified under one general heading. This heading will have a relation to the subject content of each of the documents stored. Another way to look at this general heading would be to describe it as the "average subject" of the documents. Consider again the situation where each document has an associated vector representation, derived from the *td-idf* representational model. It is now possible to generate an *average vector* of these document-vectors. This average vector is representative of the average subject content of a particular node's document set. It therefore reveals something about the subject content the node "is interested in". A way of deriving such a vector comes from Centroid based classification [67]. Given a set S of documents and their vector representations, a centroid vector C can be defined, which is the average vector of the vectors d representing the set of documents. This is given by Eqn 4.1.

$$C = \frac{1}{|S|}\sum_{d \in S} d. \qquad \textit{Eqn 4.1}$$

Average vectors are called *node-vectors*. Nodes may now be succinctly represented based on the content they store. It is also possible to compare a pair of node-vectors to assess a similarity index between the corresponding pair of nodes by using the cosine measure as described in section 3.2.2. These tools provide a means of comparing and hence grouping together nodes that are similar within the network. This process is described further in the next section.

4.3.5 Organising Network into Communities

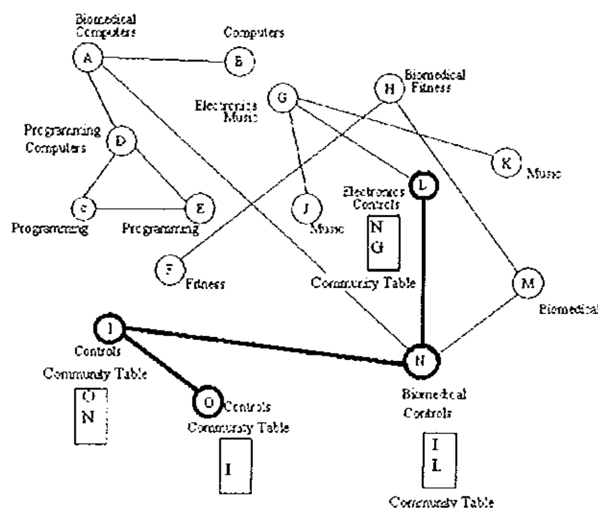

Figure 4.4. The effect of the community layer formed by community tables

Taking the simplest case that the nodes discussed in the previous section have a single topic of interest then all documents stored at the node will be related in some way to this topic. In the previous section it was seen that node-vectors represent the average of a set of document vectors stored by the node. It is therefore possible to say that in general a node-vector captures the average topic that a node is interested in. A node-vector can therefore be used to compare nodes that store similar content. If each node stores a community table with a list of nodes that are similar (based on the similarity index from the vector space model described above), document collections of a similar content will then be implicitly linked or grouped together (Figure 4.4). This could be considered as organising the network into domains whose boundaries are not strictly defined but have a "fading" effect as we jump from one community table to the next. Consider the example in figure 4.4. By moving along the path *O→I→N→L*, the document collection moves from "*Controls*" to "*Biomedical*" and then to "*Electronics*". Node *A* can be seen as the node within the network that stores documents relating both to "*Biomedical*" and "*Computers*" and thus is the point within the network where the two domains overlap. Any linked group of nodes can be seen as a domain. Links are formed based on comparing whether two nodevectors are similar with a certain threshold. Looking at figure 4.4 in this particular case it can be seen that although node A and M store similar information based on the similarity metric nodes A and N are in fact deemed more similar and therefore form a connection. Each domain can be categorised based on the nodes that have linked with each

other. As they all store similar content, these "communities" of nodes are *content sensitive* in nature because only nodes that store similar content to the community will become part of it.

4.3.6 Two Layer Network

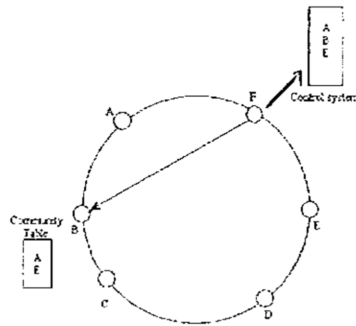

Figure 4.5.A. The new node B populates its community table with A and E as their node-vectors are most similar to its own.

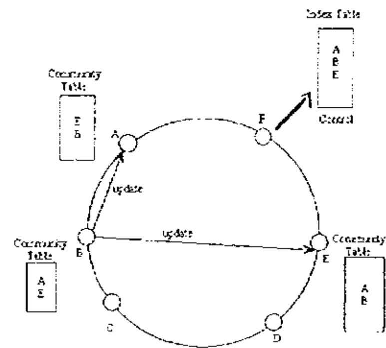

Figure 4.5.B.Creating Community Links. Node B then informs nodes A and E that they must add B to their community table.

The network is maintained and organised by employing two layers, the Pastry Layer and the community layer. Pastry maintains routing tables and *leaf sets*. Leafsets are tables containing neighbouring nodes within a certain number of hops that each node 'knows' about [86]. The Pastry layer provides a means for nodes to find indexing nodes of certain subject areas. Nodes are organised based on the assignment of nodeIds. NodeIds are generated by computing the cryptographic hash of the nodes IP address and name of the subscriber to the network. This technique ensures that with high probability nodes with adjacent nodeIds are not related in interests or for example geography. This is a result of the type of hashing algorithm employed to create their unique nodeId and helps to achieve load balancing within the network. The second layer, the *community layer*, is more organised. The ability of the second layer to organise itself is a direct result of the indexing service that is built on top of Pastry. When a node is added to an indexing node's index table, the indexing node is provided with the new node's node-vector and IP-address. Comparing its node vector to that of the already registered nodes, the indexing node places the details of the registering node in the appropriate place within a clustering tree (see section 3.4 for a discussion on clustering trees).

The new node then uses similar nodes within the cluster tree to populate its community table (see figure 4.5.B). All nodes added to the community table are then contacted and asked to add the new node to their community table. This step is taken for

each keyword. After this procedure has been carried out, the new node has links to other nodes that have similar node-vectors and hence have a good probability of 'being interested' and by extension sharing similar content. This means that nodes that "have similar interests" know about each other and can share content directly. This type of organisation makes searching within p2p networks much more efficient as all the content is grouped together into communities and so searches can be directed to a specific area of the network instead of being flooded blindly through the network.

4.4 Searching for Documents

Pure flood-based searches have become an essential operation in unstructured p2p networks such as Gnutella [4] and LimeWire [87]. These systems rely on flooding of search messages throughout the network in order to locate files stored by nodes. Within these systems, pure flood search requests are given a time to live stamp (TTL), this TTL sets the number of hops a search message is allowed to execute before "dying". Pure flood-based search methods have proved inefficient and non-scalable and result in bottle necking within the Internet. However, Pure flooding has been shown to scale well within the Gnutella network up to 10,000 nodes [11].

Within a more structured overlay network, flood-based searches can be used while maintaining scalability and cutting down on unnecessary query messages. This is achieved by performing a focused flood search where the search is focused on a specific area of the network. It is possible to target a particular part of the system and perform an exhaustive search on those areas that are more likely to contain the type of files being requested. This is achieved by the proposed system as follows. First of all a list of keywords are entered. A *search-vector* is then produced. In order to direct the search to an area of the network storing files relating to the keywords, the search-vector is compared locally to nodes within the community table. The search request is then forwarded to those nodes whose node-vectors are the most similar to the search vector. The contacted node performs a flood-search on its community table using the original keywords as the search parameters. The proxy searcher then compiles a list of "hits" and returns them directly to the requesting node. The requesting node may choose to download directly any of the files found or perform another search on a different part of the network.

Having given a detailed description of the final solution it is now useful to give a brief overview of the system. This is presented in the next section.

4.5 System Overview

As was stated in Chapter one the purpose of the work presented in this thesis is to develop a peer-to-peer system in order to allow users to join communities of other like-minded users to facilitate the sharing of documents. The concepts presented in section 4.3 describe the design solution of the system. This section gives an application overview before discussing the implementation in the next chapter. Chapter one gave an introduction to p2p computing and outlined the desired application. Grouping together users into communities of other like-minded users is the central aim of the system. One of the more challenging aspects of creating the desired system was finding a way to assess users like-mindedness and led to the various ideas presented throughout this chapter. Within the application "like-mindedness" is assessed based on the files stored by a user. This is achieved by taking the average subject content of all the document topics (see figure 4.6). Each document is first of all represented as a vector. This vector representation allows document semantics to be compared using a similarity measure. In the same way, users may be represented by the average vector of all the document vectors they store. This is a core design element of the system.

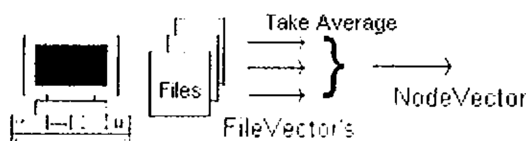

Figure 4.6. A user stores documents and their associated filevectors; a node is then described by the average vector of the document set, known as a NodeVector.

In-order to become a part of the network, each user will run an instance of the application and store a number of files. The nodevector representing the average content of the stored files is calculated locally on the users machine as in figure 4.6. A nodeId is also calculated locally by hashing the users IP-address and name. Once these initial steps have been completed the application then automatically joins the user to the pastry layer of the network using the nodes nodeId. The next step is for the application to populate its empty community tables, thus linking the user with other like-minded users. Using the nodes nodevector it is possible to assess different users "like-mindedness". It is also necessary for the user to be able to discover potentially similar users without

having to search the whole network. In a decentralised environment this can be achieved through the use of rendezvous points, rendezvous points are nodes or users machines that act as meeting points for potentially similar users by storing an index table of users. This means the any user running the application could potentially become a rendezvous point. Rendezvous points are established using a similar method to that of the global file storage system PAST [26] for establishing file storage locations. As with PAST, messages are given a message key and routed to the node whose nodeId is numerically closest to the message key. In the design solution presented here, registry-messages are routed towards destination nodes. Registry messages are used to register a user at an appropriate rendezvous point. When an application is being intialised, a user must supply one or more keywords that best describe his/her interests. The keywords are then hashed to become a key for a registry message (see figure 4.7).

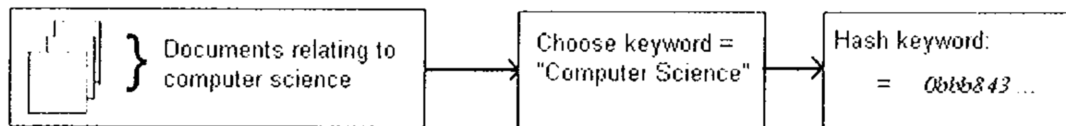

Figure 4.7. A user stores a certain number of documents, a keyword or keyphrase that best describes the documents or in fact the users interests is chosen, the keyphrase is then hashed using the same hashing algorithm used to produce nodeIds. The resulting hash code becomes a key for a registry message.

The registry message is then routed through out the Pastry layer and will end up arriving at a node or users machine whose nodeId is numerically similar to the registry message key. The application running on this machine is then required to register the user that initiated the registry message and become a rendezvous point by establishing an index table. All users stating the same keywords will also end up registering at the same point (see figure 4.8).

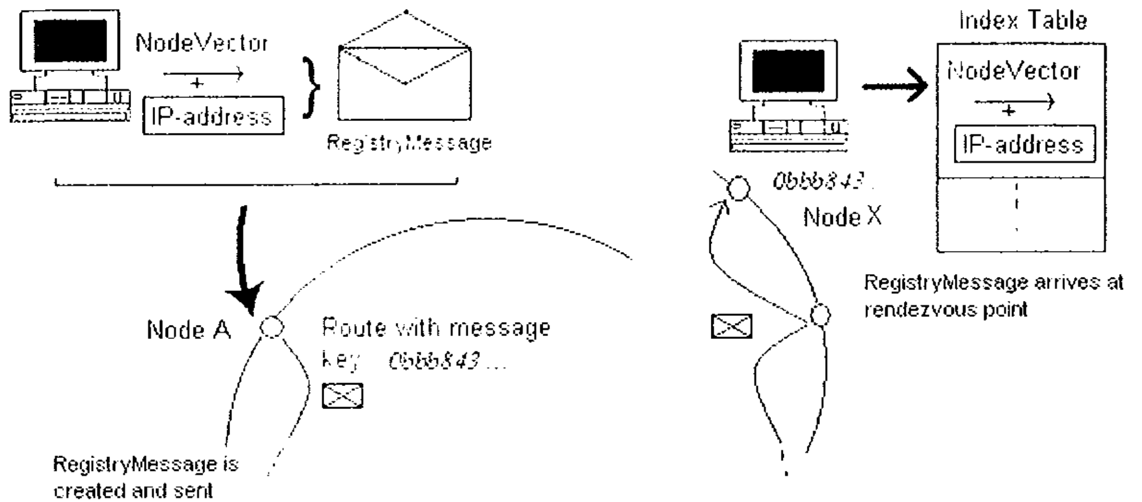

Figure 4.8. Node A registers itself in the appropriate index tables (at Node X) by creating registry messages and using the hash of keywords that describe the nodes interests as message keys.

Nodes register their IP address and nodevector within their index table. All users nodevectors are compared by the application acting as the rendezvous point using a similarity metric to assess similarity. If the nodevectors of two users are found to have a similarity that is within the value represented by a similarity cut-off point (see section 6.1.7), they are deemed to have similar interests and a connection is established between these users. Connections are maintained by the application at each node through the use community tables (see figure 4.9).

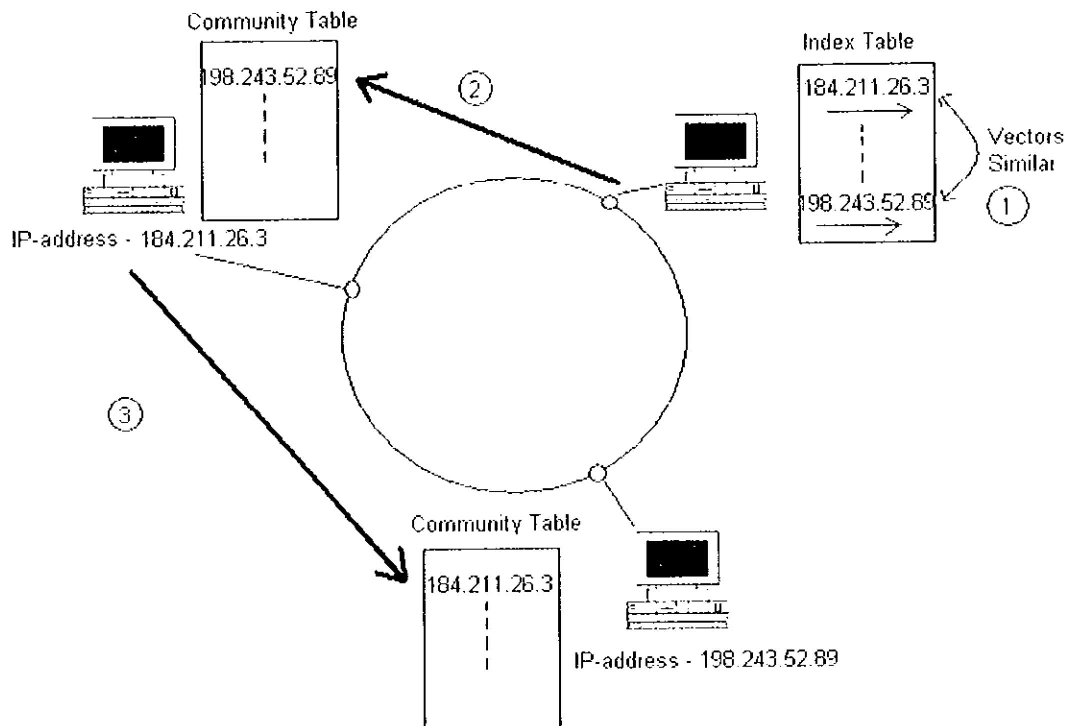

Figure 4.9. Joining a Community.

1 –the application nominated as a rendezvous point compares nodevectors to establish user "like-mindedness".

2 – the rendezvous application contacts the registering application and returns the results indicating that node 198.143.52.89 satisfies the similarity cut-off point. The setting of this cut off point is discussed in section 6.1.8.

3- the application running at 184.211.26.3 then contacts the application running at 198.143.52.89 to establish a community link.

As the network becomes more populated, so do the community tables and a community-based layer is formed. The purpose of this layer is to provide a mechanism where it is not necessary to search through the entire network in order to find relevant files but to direct a search over community links. As the community links are formed with similar nodes sharing similar content and the fact that the majority of nodes search for files similar to the ones they store themselves network traffic is reduced, as is search time. Users now have direct connections to other like-minded users. Flood searching is used to retrieve objects but because the network is now structured it is possible to perform a focused flood as is described in section 4.4 above. Users enter keywords into a search dialog box on the application. A searchvector is then produced and forwarded

throughout the systems to the relevant nodes. The users are then presented with a list of files whose file vectors are deemed most similar to the searchvector based on the similarity metric. Searching is also improved with the design as documents are searched for using search vectors, which makes it possible to return more semantically relevant documents that through traditional Boolean searches would have been overlooked.

Having given a description of the final design solution the next chapter describes the implementation of the application that realises the solution.

Chapter 5

Software Design and Implementation

Having given a description of the various potential solutions and the final design concept detailed in the previous chapter, this chapter details the software design and implementation of the system. The software design presented here is at a reasonably mature state and is the result of a number of iterations of the software. The full software design process and external APIs used to realise the application are discussed. The system is split into various packages, each one containing classes all related to each other. The system uses the idea of managers to handle the interactions between packages. Managers act as interfaces between the various functionalities needed to realise the system (see figure 5.1, Appendix D contains an explanation of the UML version used).

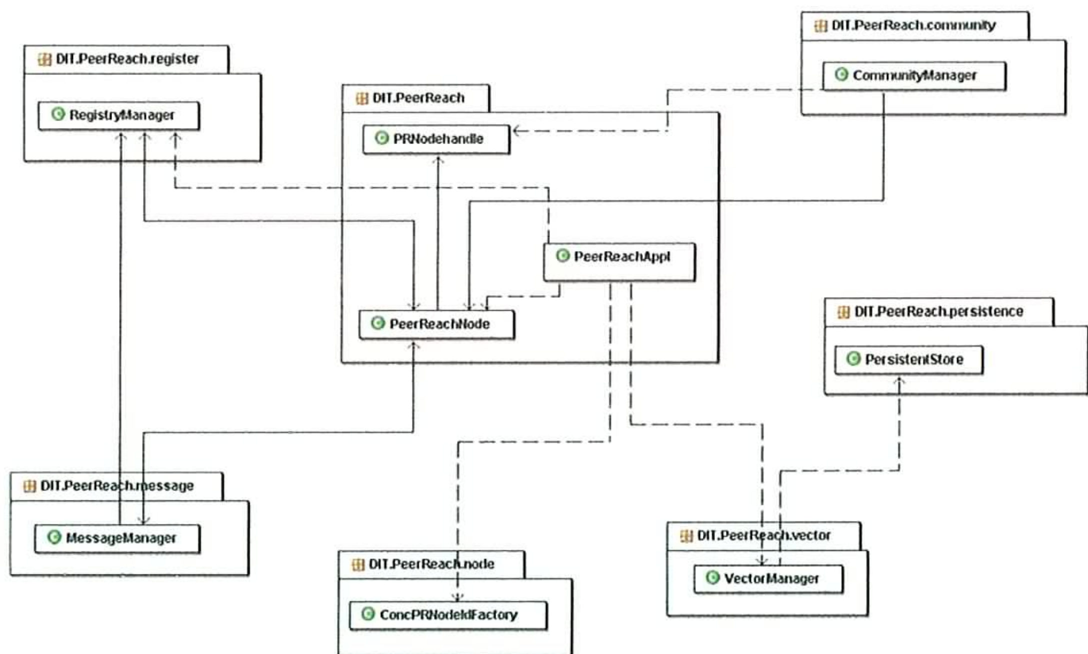

Figure 5.1 Classes are grouped into packages; the application uses the idea of managers to handle interactions between some of the more complex packages. The package diagram has been simplified to only show the interactions between the managers or interfaces for each package.

Using manager classes as interfaces to each package separates the different functionalities of the system and allows for different implementations to be plugged or unplugged from the application. This approach makes implementation more manageable and allows for the testing of each piece of functionality separately. Functional block testing of code makes it easier to pinpoint design or implementation problems. This design methodology also achieves abstraction of classes and helps the code to become more readable and maintainable. Maintainable code is essential for future iterations of the system. The main packages of interest are the *Community* and *Vector* packages. These packages will be discussed primarily within this chapter. The *Community* package contains classes to maintain and handle connections between peers that have ended up at the same rendezvous point as described in section 4.3.2. The *Vector* Package contains classes that provide the implementation of the vector space modelling methods described in section 3.2.2. The application is also composed of a number of other packages as can be seen in Figure 5.1. These packages will be discussed later in the chapter. The system also extends various classes from some imported APIs.

The most notable external API is the Pastry API. Pastry routes messages to the appropriate nodes given the appropriate key. This routing functionality is used to form a distributed indexing service and creates various rendezvous points so nodes sharing similar files can discover each other. The system is developed in Java, this is due to Java's suitability to writing applications that operate over network connections or that are deployed over the Internet. The Pastry API is the first point of discussion. The reason for this is that it is Pastry's design and organisation of nodes that has inspired the design of the community layer and provided a way in which the distributed indexing service can be realised. In the initial stages of this research it was intended that Pastry would play a more significant role in routing all messages throughout the system. However, throughout the development process it was found that the development of a second layer, the community layer would better serve connections and message passing within the application. Pastry's role is now to provide the basic services and initialisation. Despite this, Pastry's API provides the core functionalities for the system and so will be discussed first.

5.1 The Pastry Application Programming Interface (API)

When building applications on top of Pastry, developers are given the freedom to use Pastry in many different ways; this is due to Pastry's extensibility and its low level design. As has been discussed in section 2.3.1, Pastry provides a mechanism for sending messages around the ring in which it has organised the various nodes. To achieve this, any applications using Pastry must export the following operations [22]:

`deliver(msg, key)` called by Pastry when a message is received and the local node's nodeId is numerically closest to key among all live nodes, or when a message is received that was transmitted via `send()`, the application typically responds using the IP address of the local node.

`forward(msg, key, nextId)` called by Pastry just before a message is forwarded to the node with nodeId = nextId. The application may change the contents of the message or the value of nextId. Setting the nextId to *null* will terminate the message at the local node.

The Pastry API exports the following operations:

`NodeId = pastryInit(Credentials)` when called by the application causes the local node to join an existing Pastry network (or start a new one) and initialise all relevant states; the method returns the local node's nodeId. The credentials are provided by the application and contain information needed to authenticate the local node and to securely join the Pastry network.

`routeMsg(msg, key)` causes Pastry to route the given message to the node with a nodeId numerically closest to the key provided, among all live Pastry nodes.

`send(msg, IP-addr)` causes Pastry to send the given message to the node with the specified IP address, if that node is live. That node then receives the message through the deliver method. `Send` differs from `routeMsg` as `routeMsg` forwards messages based on a messageId. `routeMsg` attempts to match a nodeId of a node that is known about locally to the messageId of the message in order to determine the best node to forward the message to.

Applications that wish to use the native Pastry API must extend the class `rice.pastry.client.PastryAppl`. This class implements the Pastry API. An application may also extend `rice.pastry.client.CommonAPI` to incorporate the common API discussed in section 2.4. Each application that uses Pastry should consist minimally of an application class that extends `PastryAppl` or `CommonAPI`, and a driver

class that implements `main()`, creates and initialises one or more nodes. The application for ease of implementation has been given the working title PeerReach (peer reviewed archive). Within PeerReach `dit.peerreach.PeerReachNode` extends `rice.pastry.client.CommonAPI` and `dit.peerreach.PeerReachAppl` acts as the driver class that contains the `main()` method (see figure 5.2).

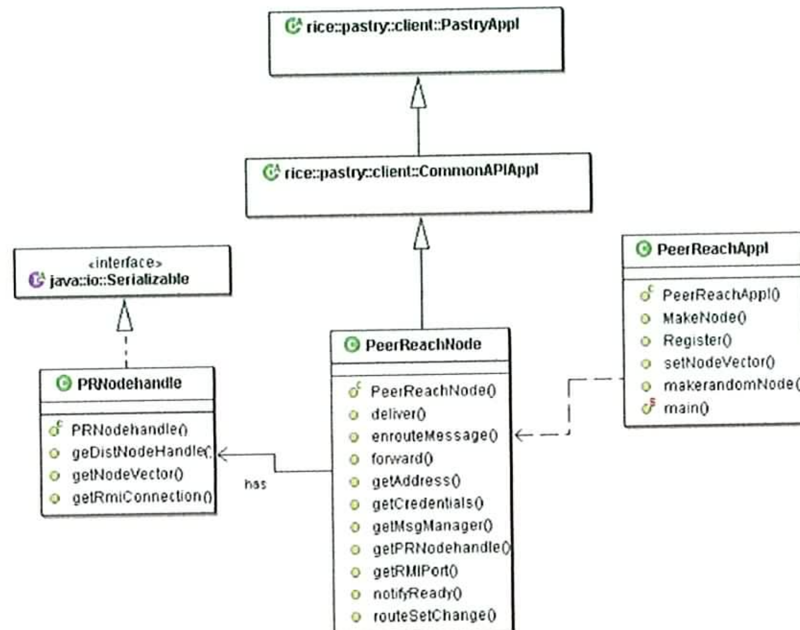

Figure 5.2. PeerReachNode extends the Common API class and PeerReachAppl acts the Driver for the application

Pastry provides the application developed here with a method for creating a distributed indexing system. It is Pastry's routing algorithm that is used to accomplish this. `PRMessage` extends `rice.pastry.messaging.Message`, and it is this message that is used to register nodes at particular rendezvous points around the ring, (see section 4.3.2). When keywords inputted by the user are hashed, a registry message key is produced. This registry message is then passed down to the Pastry layer using `routeMsg()` which is an inherited method from the PastryAppl class (see figure 5.2). `routeMsg()`, routes the given message to the live node whose NodeId is numerically closest to the message key. When the message arrives at the appropriate node Pastry calls the `forward()` method which has been implemented by `PeerReachNode()`, this is Pastry passing the message back to the application layer (see figure 5.3).

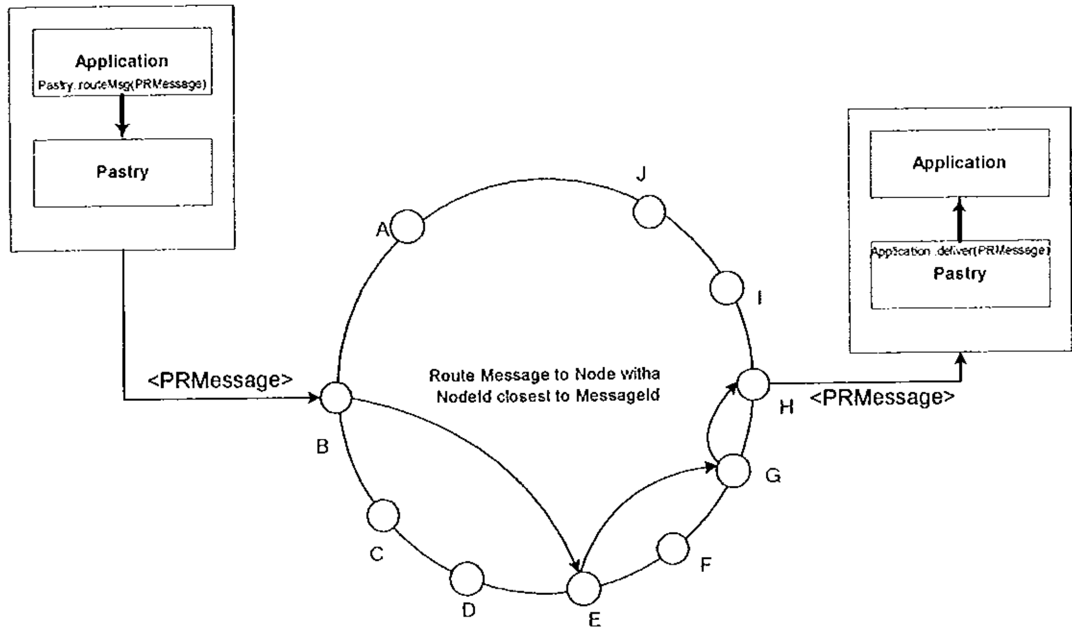

Figure5.3. Delivering a Message, used to register a nodes interest to the node with a nodeId closest to the registry message key

`PeerReachNode` provides an implementation for deliver (see figure 5.2); once a message has arrived the message is passed on to a message reader that determines what is done with it next, this will be discussed later. It is this attribute of Pastry that is used to construct the distributed indexing service and provide an environment where nodes with similar interests may discover each other.

5.2 System Classes and Packages

The classes implemented to create the application are grouped into various packages. These classes are developed in order to satisfy the design ideas presented in the previous chapter (see Appendix A for a code listing). Each package contains classes that together perform certain tasks. `DIT.PeerReach` contains the classes that implement the necessary classes to layer the application over Pastry, as discussed above.

Two of the more important packages are that of the *Vector* and *Community* packages. These two packages will be discussed before moving on to the other packages.

5.2.1 The Vector Space Modelling Package

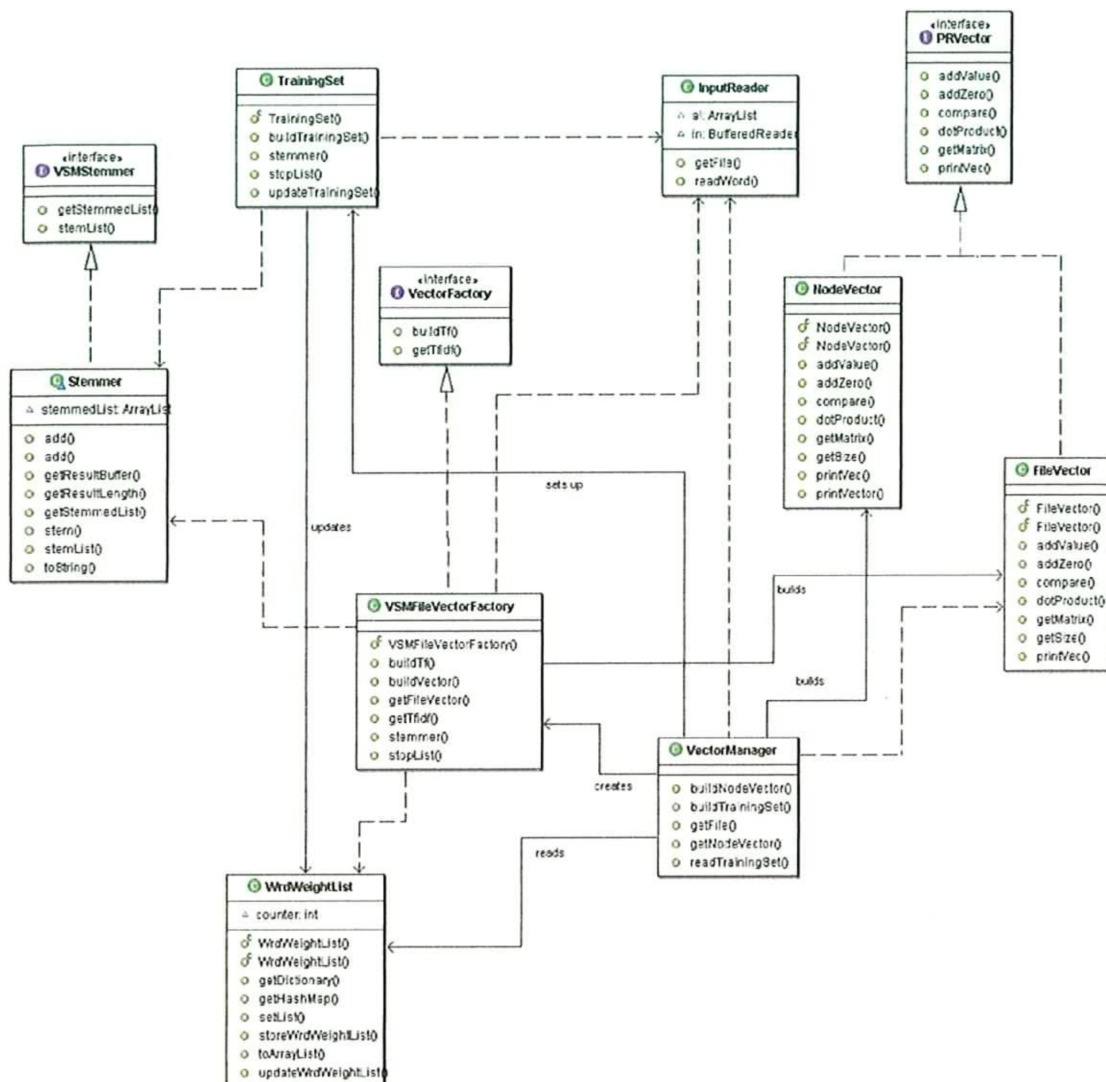

Figure 5.4 Vector Space Modelling Package UML diagram.

The Vector Space Modelling package, `DIT.PeerReach.vector`; groups together classes that implement the algorithms used to represent documents as vectors. This is one of the two core packages within the application. The package provides 3 interfaces, PRVector, VectorFactory, and PRStemmer for extensibility. The package has been designed using the Java pattern, Factory [88].

PRVector

PRVector is an interface to any class that wraps an instantiation of the class `Jama.Matrix`; Jama [89] is an external API that provides matrix algebra functionality. Jama implements a Matrix class, a vector is a one dimensional Matrix, thus the Matrix

class provided by Jama can be used to represent a vector representation of files of nodes etc. Jama is discussed in more detail below. `FileVector` and `NodeVector` currently implement `PRVector`. `FileVector` is an abstract representation of the term frequency vector discussed in section 3.2.2. All implementers of `PRVector` must provide methods of comparing two vectors

```
- public double compare(FileVector fv);
```

The returned value of type double is the similarity metric that is used to determine whether the two files represented by their `FileVector` are similar. The class `NodeVector` that extends `PRVector` is an abstract representation of a centroid or node-vector (see section 4.3.4), which represents the average interest of a node. Again the compare method is used to assess whether two nodes are similar enough to add each other to their community tables.

VectorFactory

`VectorFactory` is an Interface to a class that implements an algorithm to construct a vector of the form `PRVector`. This class is currently implemented by `FileVectorFactory`, which is used to construct `FileVector` representations of text files stored by a node. The reason for using an interface here is to allow other vector factories to be developed. This allows the application to be extended to share other file formats such as images or audio.

PRStemmer

`PRStemmer` is an interface to any class implementing a stemming algorithm, stemming is discussed in section 3.2.3. All classes implementing this interface must implement the two methods.

```
- public void stemList(ArrayList al);
- public ArrayList getStemmedList();
```

These two methods take a list of words contained in an ArrayList stem the appropriate ends and return the list of roots. Currently the interface is extended by the porter-stemming algorithm [90] (see section 3.2.3), which is implemented in the class Stemmer. `PRStemmer` allows the use of other stemming algorithms.

TrainingSet

`TrainingSet` represents an abstract implementation of the global dictionary or word frequency dictionary.

VectorManager

`VectorManager` is a class that handles all interactions between the *Vector* package and the rest of the system. This class also controls the actions within the package.

5.2.2 The Community Package

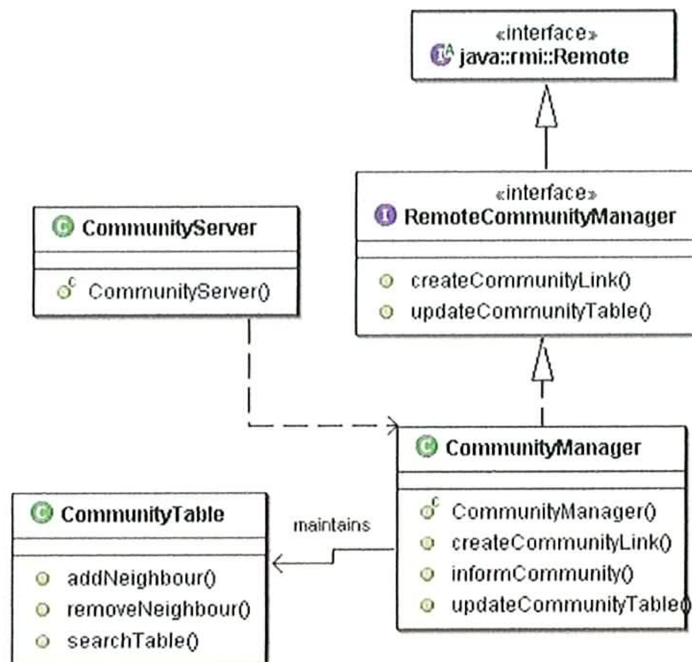

Figure 5.5 Community package class UML diagram

The *Community* package contains classes for setting up and managing community links (see figure 5.5). Community links are made through an RMI connection. Community tables are kept up to date through the exported RMI methods:

```
- createCommunityLink(PRNodeHandle);
```

Called by the `CommunityManager`, and:

```
- updateCommunityTable (Vector nodeSet);
```

Called by the `RegistryManager`

`CommunityServer` acts as the server and `CommunityManager` acts as the client, in a peer-to-peer environment each node is an RMI server and an RMI Client (see figure 5.6).

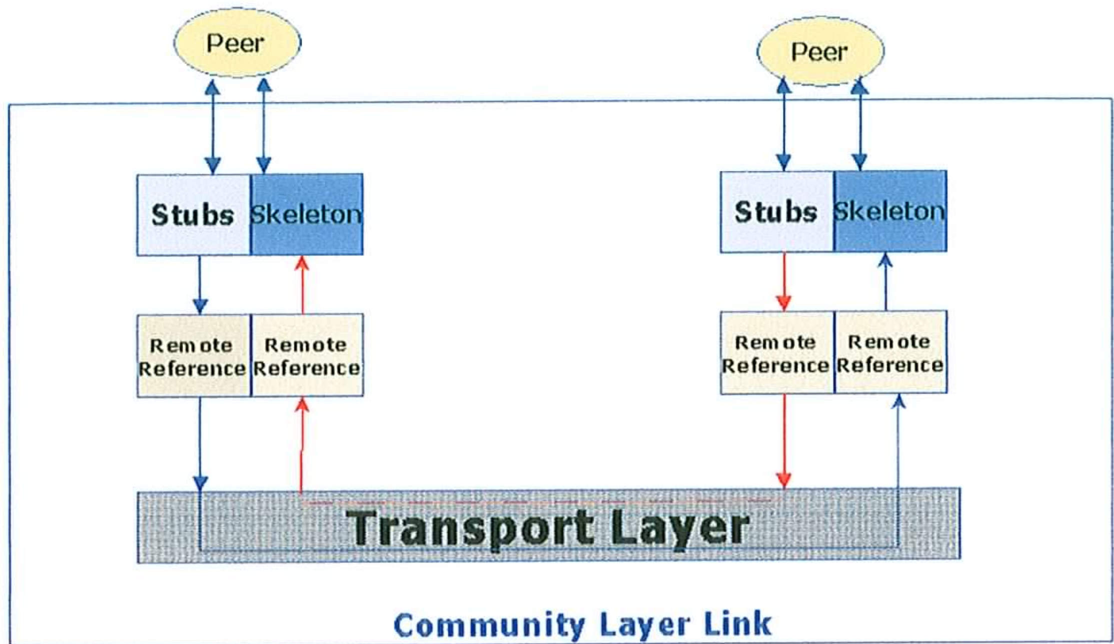

Figure5.6 RMI links between two peers, each one act as a server and a client

5.2.3 The Register Package

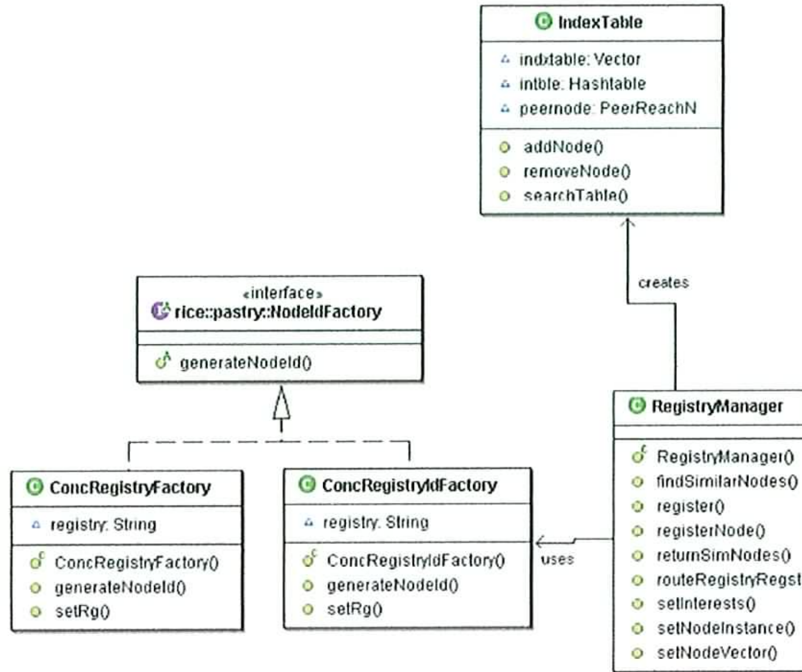

Figure 5.7 UML class diagram of the classes contained in the register package

The register package contains classes that are used for registering a nodes interests and building index tables when a node receives a registry message. The package contains one interface `RegistryIdFactory` to facilitate methods of creating registry messageIds. The RegistryId class is an abstract implementation of a `RegistryId` and is created by hashing keywords entered by the user. `ConcRegistryIdFactory` implements a method to create the `RegistryId`. This is done using a hashing algorithm (see section 4.3.6), currently the SHA_1 hashing algorithm is used. `IndexTable` is used to register any other nodes that are to be registered at this node. If a registry message arrives at `deliver()` in `PeerReachNode` it is first sent to the *MessageManager* to find out the type of message. The *MessageManager* then passes it to the *RegistryManager* and it is added to the index table.

Below is a sequence diagram and accompanying description that illustrates the process of registering a node at a rendezvous point. The registering node then returns similar nodes to the registered node. The registered node then contacts the similar nodes and sets up community links.

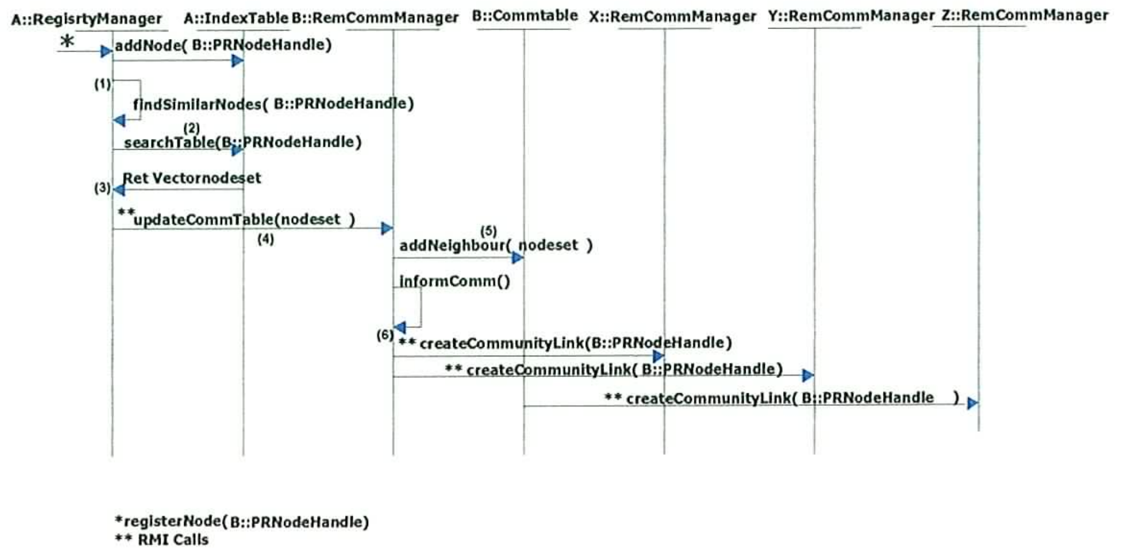

Figure 5.8 Sequence diagram detailing the sequence of events to add a node to a nodes community table

Explanation of sequence diagram in Figure 5.8

(1) `addNode()` is called in the *RegistryManager* and a `PRNodeHandle` object is passed in that contains the node to be registered details. This method adds the node to the register using findSimilarNodes().

(2) The registry Manager then attempts to find other similar nodes by searching the index table using `searchTable()`.

(3) The search returns a nodeSet containing similar nodes using `updateCommTable()`.

(4) The Registry Manager then returns the nodeSet via an RMI link to the newly registered nodes Community Manager using `addNeighbour()`.

(5)The registering node then adds the nodeSet to its community table using `informComm(),createCommunityLink()`.

(6)The community manager then contacts the community nodes to set up community links

5.2.4 The Message Package

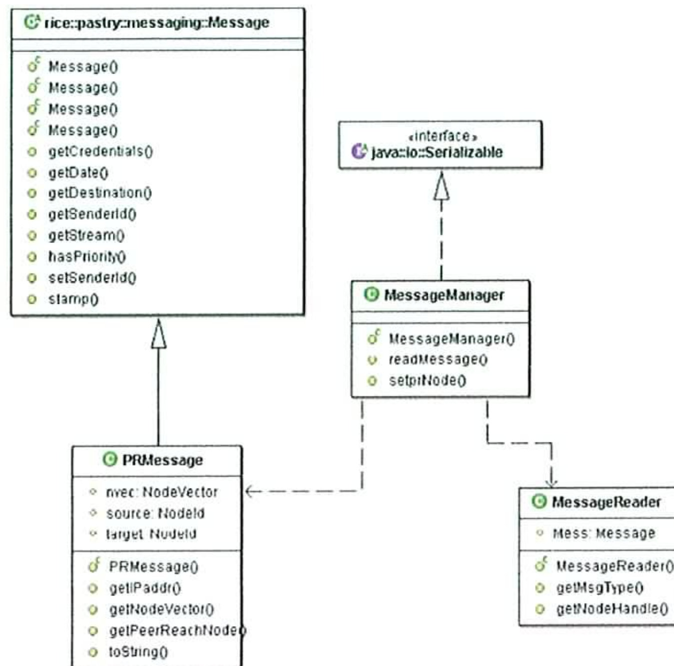

Figure 5.9. UML Diagram of the Messaging Package

The Messaging package contains classes that represent messages and can deal with messages that are sent between peers.

PRMessage extends the class `rice.pastry.messaging.Message.PRMessage` `and is` used to register nodes at the various rendezvous points.

5.2.5 The Node Package

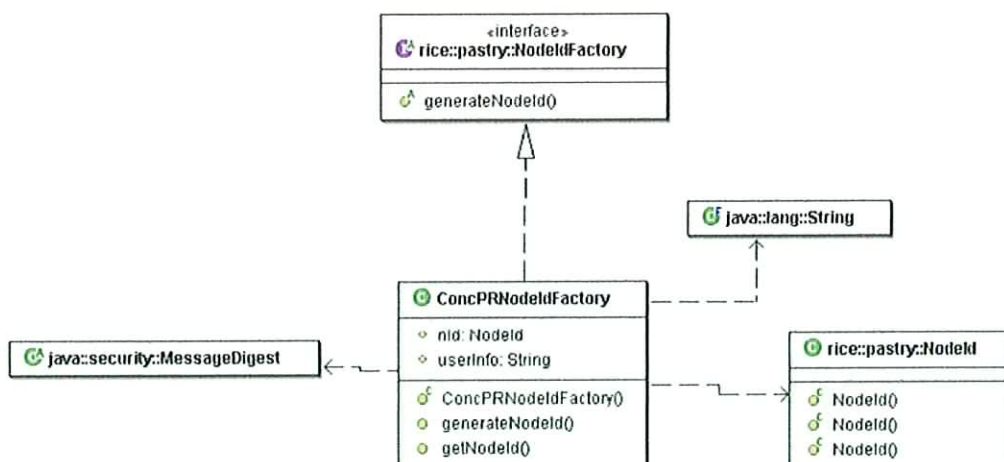

Figure 5.10. UML Diagram of the Node Package

The Node package contains classes for constructing nodeIds. Currently the Node package contains `ConcPRNodeIdFactory`, which has methods to construct and return nodeIds based on user information.

5.3 Imported APIs

Aside from the native Java API imports, a number of other important APIs are used to realise the system which require more discussion. Pastry is one of the main APIs imported and has already been discussed above. In this section JAMA a matrix algebra API is discussed as well as the PJX API [91] that is used to read text from PDF files.

5.3.1 JAMA: A Java Matrix Package

JAMA is a basic linear algebra package for Java. It provides user-level classes for constructing and manipulating real, dense matrices. The Matrix class is the main class provided by JAMA and it provides the fundamental operations of numerical linear algebra. Various constructors create Matrices from two-dimensional arrays of double precision floating point numbers. Various *gets* and *sets* provide access to sub-matrices and matrix elements. The basic arithmetic operations include matrix addition and multiplication, matrix norms and selected element-by-element array operations.

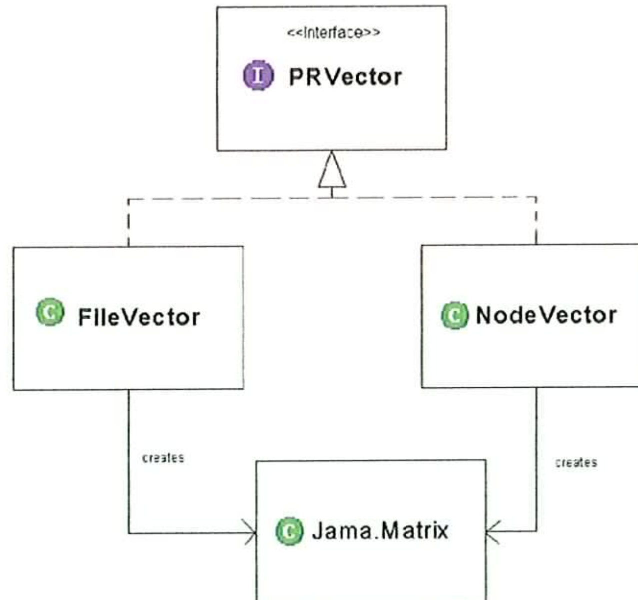

Figure 5.1.1. UML Diagram of the Jama.Matirx interactions Package

JAMA is used in this system within the `dit.peerreach.vector` package. `Dit.peerreach.vector.FileVector` and `dit.peerreach.vector.NodeVector` act as wrapper classes for `Jama.Matrix`, which is used to represent a vector. There are a number of useful methods that are implemented by the Jama.Matrix class. `norm2()` may be used to normalise the vector as was described in section3.2.2. `FileVector` and `NodeVector` use the `norm2()` method so as to compare vectors and obtain a similarity metric.

5.3.2 PJX

PJX [91] is a Java class library for PDF software development. Initially this application has been developed to deal with conference papers. As the majority of papers come in PDF format it is necessary to provide a mechanism for automatically extracting text from a PDF document in order to be able to represent the document as a vector within a term space. The PdfReader class within the PJX API provides low-level access to an existing PDF document. Once the PdfReader instance has been constructed, a PdfManager instance can be associated with it. The text from a PDF may then be extracted and saved into a text file for further processing.

This chapter has described the software design and implementation of a prototype that implements some of the main functionalities needed to realise the proposed solution outlined in Chapter 4. Code for the major classes is contained in Appendix A. This implementation does not constitute a fully operational system but has served to develop a working prototype to demonstrate and evaluate the ability of the system to satisfy the basic requirements of the proposed system. It was desired to develop a p2p system that demonstrates an improved performance in the areas of scalability and robustness compared to other systems detailed in Chapter 1. This has been done through the development of a software prototype that allows users to form communities using the vector space modelling of stored documents and the creation of node vectors. The prototype also realises search vectors. Using these functionalities the system is able to search for objects based on semantics rather than keywords in a more organised fashion. The implementation also provides the functionality needed for the routing and receiving of messages from the Pastry substrate.

Before developing a second version it is first of all necessary to test the current code. Testing of the code will evaluate its performance in order to assess how well it realises the system requirements and show what other considerations need to be taken

into account at this stage of the development process. The next chapter discusses the simulation and evaluation of some of the core assumptions and functionalities implemented within the discussed prototype and draws conclusions and suggestions for the next stage of development.

Chapter 6

Evaluation and Discussion

The previous chapter detailed the prototype implementation of the p2p system that was described in Chapter four. It is necessary to evaluate the core functionalities of the developed prototype to assess how well it might perform when deployed over a large number of nodes in comparison to a less structured p2p node configuration. In addition, the system has a number of configurable settings and attributes. One example of a configurable setting is the cut off value of a similarity metric that determines whether two documents are similar. The values that are chosen for these attributes influence the efficiency, speed and scalability of the application. In order to optimise and choose appropriate settings and values, some simulation and evaluation was required. It was also necessary before developing the system, to assess the vector space modelling algorithms that were discussed in Section 3.2.2. It was also important to investigate how a system with a community layer performs in comparison to a non-structured p2p system. This chapter provides a discussion of the evaluation and simulations performed. Specifically, the simulations were intended to answer the following questions. Can the vector space modelling of documents and the idea of a centroid vector be used to accurately identify a document that is similar to a group of documents? When using the implemented Java vector package, what is an approximate threshold or similarity metric cut off point to determine whether two documents are similar? Does the addition of a community layer for the organisation of nodes into content sensitive communities improve searching compared with a non-structured node organisation? The chapter is split into two major sections. The first part of the chapter presents the experiments and simulations that were carried out to answer the above questions. The second part of the chapter includes a discussion of the overall application, observations made from the work and points to possible future related work.

6.1 Evaluation and Simulation

This section describes the initial exercise of becoming familiar with the method of representing and comparing documents as vectors. At the time, this task was seen more as a personal exploratory exercise than a formal research task. The intended outcome was to gain experience and to evaluate the *Vector Space Modelling* methods described in section 3.2.2. Despite this, it provides a useful means of describing in further detail the task of deriving vector representations and so it is included here.

6.1.1 Evaluation of Vector Space Modelling Algorithm

In order to assess the ability of a centroid vector (see section 3.3.1) to represent the average subject content, a group of documents and the corresponding node vectors (see section 4.3.4) ability to represent the average content of a node, an experiment was undertaken using Matlab (see Appendix B for code listing).

Matlab [92] is a tool for performing numerical computations with matrices and vectors. Desired tasks in Matlab can be described in the form of an m-file, which is a file containing a series of Matlab commands that are executed in sequence. The vector space algorithm described in section 3.2.2 was implemented as a Matlab m-file. This type of evaluation or simulation environment is a more sensible choice in performing a task of this type. Matlab's extensive support for matrix mathematics through its toolboxes increases implementation time compared with coding in other languages. The first step was to gather 15 research papers in PDF format. These 15 papers were selected from three main topics. 7 of the papers detailed research on peer-to-peer networking, 3 contained work done in the area of control engineering and 5 papers dealing with RF standards. The text was stripped from all PDF files and a plain text file was created for each one. The m-file was implemented to read in each text file individually. After reading each word a space in an array was created for each new word. For each re-occurrence of the word, a frequency counter was incremented and stored in the array. The number of documents that the word had occurred in was also recorded. A list of the thousand most common words in the English language was obtained and compared against the constructed array; any word occurring within the 1000 most common English words was removed from the array, so only words that contained semantic meaning remained. This is the removal of *stop words* described in section 3.2.3. A second m-file was then developed in order to calculate a vector representation of the

frequency of occurrence of words within each text file. This vector is known as the term-frequency (TF) vector as discussed in section 3.2.2.

The array of words constructed from the 15 documents contained 8,556 individually occurring words that did not fall into the 1,000 most common words category. This number of words is quite large, although not implemented here there are a number of other optimisations to reduce this size discussed in section 3.2.3. This meant that each vector representation would contain an 8,556-term space. Each term was then weighted as described in section 3.2.2. At this point, the array contained the number of documents a word occurred in, denoted N, and the frequency or the number of times the word occurred within the document set, denoted df_i. This now gives the *tf-idf* representation of each document as represented by Equation 3.1.

A centroid vector for each set of documents belonging to the same class was then constructed. This centroid vector is computed from the average of the vectors representing each of the files in the document set. A third m-file was developed to perform this task. Given a set of documents S and their corresponding vector representations, as previously calculated, the centroid vector C can be calculated using Equation 3.4.

To test whether the centroid-based classifying method could correctly determine document similarity to a centroid vector, 3 new documents were introduced to the experiment. Each of the three documents was similar to one of the three topics of the original sets of documents. In order to identify their subject, a vector representation for each document was found using the array of terms/words found from the 15 documents. Each new document was taken and the similarity between each document and the three constructed centroid vectors was found using the cosine measure given by Eqn 3.3.

As can be seen from figure 6.1. A centroid vector representing the average content of each of the three sets of documents was produced. A documents similarity to the centroid vector is measured between 0 and 1 (1 completely the same and 0 having no similarity). As can be seen in each case, the document vector representing the three new documents correctly matched the documents to the centroid vectors.

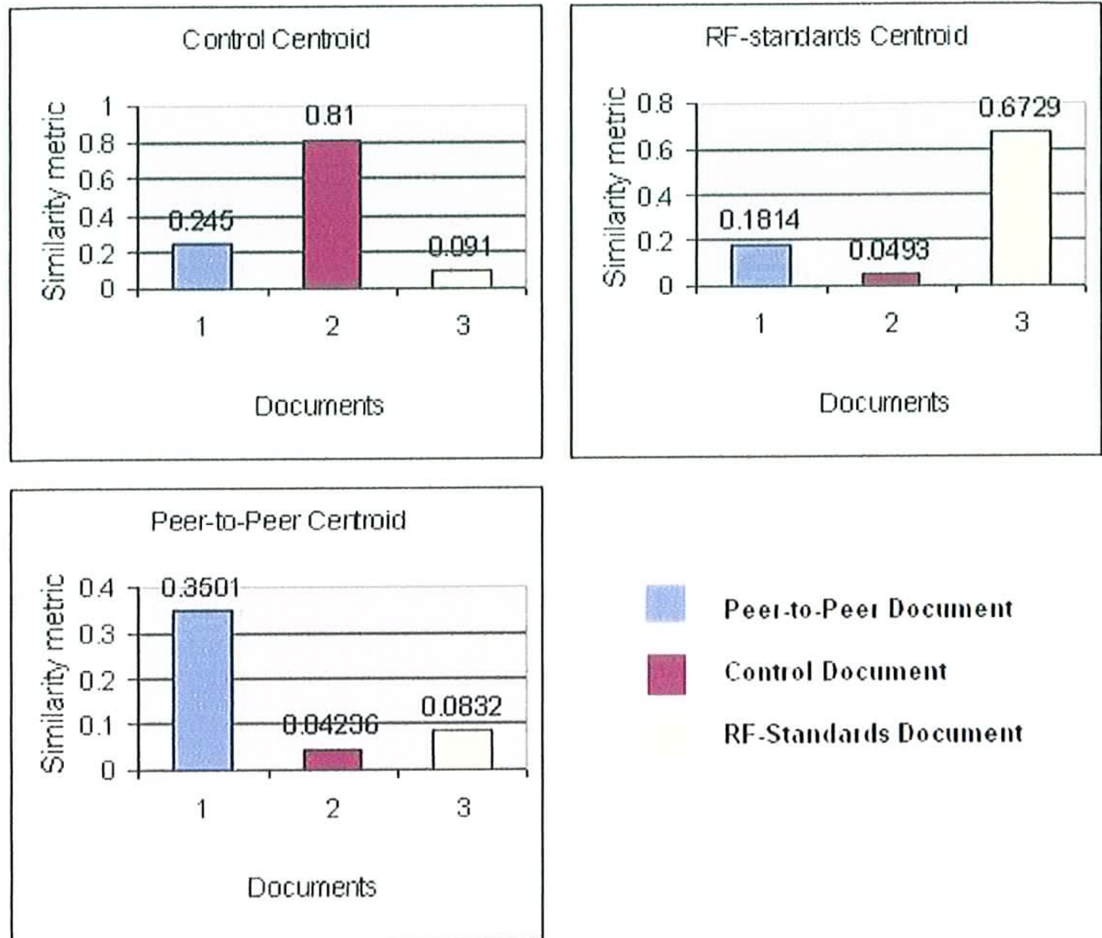

Figure 6.1.Results from Matlab vector space implementation.

This exercise provided a good way of understanding the idea of the vector space representation of documents. It is important to note that the results that were observed although very promising the experiment cannot be used as a full proof of concept regarding the vector space modelling and classification of documents. There have been numerous more exhaustive performance evaluations of the method that have proved its ability to identify semantics in natural language based texts, one example can be found in [93]. Given this, the experiment serves mainly as a preliminary implementation evaluation. For this reason only small number of documents were necessary to perform the exercise. This helped to keep the processing necessary to a minimum yet still gaining the required outcomes from the exercise. The algorithm was used in its basic and most crude form without any optimisations but made the transition to implementing in Java much smoother. The success of this experiment led to the integration of the vector space modelling algorithm into the application, this was implemented as the *Vector* package, see section 5.2.1. As Matlab and Java support different levels of coding

support the transition to Java was not as straight forward as translating the code line for line. In order to reduce coding time, JAMA [89] an external API that supports matrix mathematical functions much in the same way the Matlab toolbox does was imported. This significantly reduced the complexity of the transition. Java's support for object oriented design techniques also made implementing the functionality cleaner than the more structured coding techniques needed to write m-files.

6.1.2 Simulation of Peer-to-Peer Networks

As was stated above, it was required to simulate the system to determine appropriate configuration parameters. It is also necessary to get an accurate idea of how a peer-to-peer system will perform in the real world. Deploying p2p systems on the Internet for the purpose of testing is impractical and prohibitively expensive.

In order to produce realistic results, it is necessary to develop simulations that resemble real-world p2p networks. P2P algorithms and protocols need to be tested by simulation under network models that attempt to mimic typical node interactions, traffic patterns etc. Decentralised p2p systems are more sensitive to performance issues than in comparison to many other centralised systems and thus require such simulation. The reason for this is that decentralised p2p systems are highly dependent on communication between peers; as a result the system becomes highly dependant on the network. In the case where node connection speeds are low, bottlenecks can occur. As a result of this bottlenecking, it also takes more effort to search through a p2p system as nodes are not responding within a given interval of time and messages, are forwarded many times to an ever-increasing number of nodes. This in turn adds to bandwidth load and affects the time needed to respond to a query. Since algorithms and protocols are often sensitive to traffic and network behaviour, there is a clear need for accurate p2p network models. At this point it is useful to consider some p2p modelling concepts and systems that may help in predicting the performance of the system that has been described in this work.

6.1.3 Simulating a Peer-to-Peer File–Sharing Network

The simulation of p2p file sharing networks has mainly been carried out for specific algorithms and systems [80]. Although it is clearly important to model the underlying network is also important to model the behaviour and interests of the various users. This is especially important in the case of the system discussed in this thesis, as

connections from a node are reliant on content stored of that node. In order to simulate and model some of the important interactions between nodes, it is very important to model the content distribution and the interests of peers. Schlossler and Kamvar in [80] classify these parameters into two types, content distribution parameters, and peer behaviour parameters.

Content Distribution – It is necessary to accurately model the type of content each peer carries. P2P networks are far from heterogeneous in terms of type and volume of data shared; hence a model reflecting real world p2p networks is required.

Peer Behaviour – It is also necessary to accurately model peer behaviour, i.e. what kind of content is a peer most likely to search for, as this will affect the queries being generated.

One observation that can give hints on how to model the behaviour of users who search and store content is presented in [94]. It is observed in [94] that many storage systems such as WWW follow a Zipf distribution. A Zipf distribution implies that small occurrences are extremely common, whereas large instances are extremely rare. Within a p2p file-sharing network a Zipf distribution would mean that a large number of users search for a small but popular selection of the documents stored. Observations of Zipf distributions have strong implications for the design and function of the Internet. To model the community layer it is proposed to take the above observations on content distribution and Zipf's law along with Schlossler and Kamvar's method in [94] to develop a simulation model. It is first necessary to discuss the model in more detail.

6.1.4 Modelling Content Distribution

It has been shown in section 4.3.6 how the community layer of the application creates and manages connections to other peers that are based on the interests of the users within the system. Within this model, n categories are assigned as follows (see Eqn 6.1). The set of categories C that are available within the simulated community is defined as.

$$c = \{c_1, c_2..c_i...c_n\} \qquad \textit{Eqn 6.1}$$

Where c_i is an integer between 0 and n that represents a category from the set of n categories. Each peer i within the model is assigned a selection of content categories from the set C. This selection happens according to the Zipf distribution. This models the observations discussed above about the distribution of content on the web.

Equation 6.2 [94] is a Zipf distribution model, where $p(c_i)$ is the probability of a peer being interested in the category c_i.

$$p(c_i) = \frac{\frac{1}{c_i}}{\sum_{i=1}^{n} \frac{1}{i}} \qquad \text{Eqn 6.2}$$

The content categories for each node are assigned using equation 6.2. Consider the case where a peer has only two interest categories, the two categories with the highest probability for a particular peer will be chosen. After assigning content categories to each peer, a peer's "interest level" is assigned for each of the assigned categories c_i. The interest level determines what number of files from each assigned category a node will "store". The reason for this is that not all files from the assigned categories will be stored by the peer. A peer is assigned files by randomly choosing a number of files from the categories that have been assigned to satisfy the nodes calculated interest. In other words if a peer has the capacity to store 10 files, first of all a number of categories are assigned to the peer as in equation 6.2, it is then necessary to decide how many files from the categories chosen will be stored within the storage capacity of the peer. The calculated interest level in c_i determines the percentage of the storage capacity that will be taken up with files from c_i. The interest level is calculated by a uniform random distribution. Each file may be uniquely identified by a file number and the category it belongs to i.e. $\{c_i, r_i\}$, where c_i is the category the file belongs to and r_i is the files identifier within that category. Therefore a file may be identified using f_{c_i, r_i}.

In order to model the types of queries that will be generated by each peer it has been observed in [80] that peers search for files that they are interested in, as in the same category as the files they store. Within the proposed model, *FileVectors* (see section 4.3.4) stored by peers are used as search queries. This means that peers will search for content similar to that of the content that they already store.

Having described the simulation model, it is now necessary to describe the implementation of the simulator and the steps taken to realise the model.

6.1.5 Implementation of the Simulator

It is interesting to note that simulations of p2p systems and algorithms are mainly constructed for specific p2p algorithms and cannot be easily adapted to other implementations. More general simulators that meet the needs of p2p designers do not currently exist. As a consequence p2p designers are forced to design their own simulators, which is quite a large undertaking. There are however network simulators that aid in the development of network models. These simulators can be adapted to give the low-level network implementation upon which a p2p simulator can be built. It is then possible to model the desired p2p system to gain insight about how it may behave on the Internet and to determine appropriate configuration parameters for the system.

As was mentioned above, it was required to create a simulation environment that could assess the performance of the community layer compared to other methods of creating node links. The first task in implementing an appropriate simulator was to review the available low-level network simulators. The first prerequisite in choosing a simulator for this work was that it must be compatible with Java. Using a simulator that is compatible with Java allows the reuse of much of the code already written for the implementation described in chapter five. The following section gives a review of some of the more interesting and potentially useful simulators that were encountered.

6.1.6 Network Simulators

The following section gives a brief overview of some of the simulators that were evaluated. Although the simulators evaluated were not designed to model p2p networks specifically, their design was flexible enough to allow for them to be used in this way. From the literature it was found that the NS simulator and the later version ns-2 appeared to be the most popular. For that reason the NS simulator was evaluated first.

Network Simulator (NS)

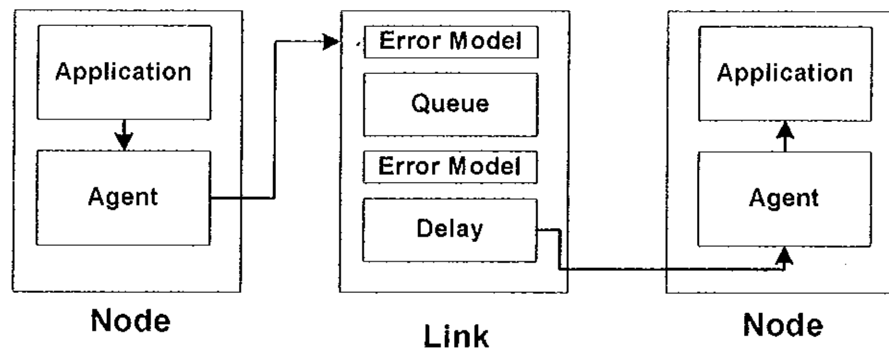

Figure 6.2. Block Diagram showing the NS Network Model.

The Network Simulator (NS) [95] was developed at U.C. Berkeley; the simulator is a discrete event simulator. This means that the system is considered only at selected moments in time (the *observation points*). NS is mainly targeted at networking research. It gives substantial support for the simulation of TCP [36], routing and multicast protocols. NS is written in C++ and uses a scripting language known as Tcl [96] (tool command language) as a command and configuration interface. The simulator is event driven and runs in a non-real-time fashion. NS also includes single or multiple traffic generators and statistical generators it also allows such functionality such as FTP [97] and Telnet [98] to be attached to any node. Network transport protocol behaviour is simulated by attaching agents to nodes (see figure 6.2). Links between nodes are than modelled as simplex or duplex links with the option of defining capacity, delay etc. NS also provides a way of outputting trace data. NAM (the Network Animator) is a Tcl based animation tool that can be used for viewing NS trace files.

The NS simulator appears to be a very popular simulator and appears in many texts and papers [99] [100] [101], this is why it was the first simulator to be investigated. However, although NS and consequently the ns-2 simulator provided very desirable qualities, it wasn't Java compatible. It was also recommended that NS should not be run in Windows, which is the current operating system used in the development of this work. For this reason NS and ns-2 was not investigated any further.

SSFNet

SSFNet [102] is another well-documented simulation environment. SSFNet provides a collection of Java SSF-based components that can be used for modelling and

simulation of Internet protocols and networks at and above the IP packet level. SSFNet models can be written and configured in DML [103] (Domain Modelling Language) a scripting language that is used in the same way Tcl is used by NS.

Having evaluated a number of models in SSFNet, it was found that the code was somewhat difficult to wrap around the existing PeerReach implementation. A significant rework of the PeerReach code would be needed to perform the necessary simulations. So, this simulator was also rejected.

Java Network Simulator

The Java Network Simulator (JNS) [104] is a Java implementation of ns-2. JNS is platform independent unlike ns-2. Again, this simulator allows developers of protocols to simulate and test their protocols in a controlled environment. JNS is purely Java based and does not provide any scripting configuration. The simulator also provides trace outputs that may be viewed in Javis [105]. This simulator satisfies the minimum set of desired attributes but as we shall see below, it was not chosen, as a more suitable simulator exists.

JiST

One of the main motivations for the development of JiST [106] is the observation that most published ad-hoc network results are based on simulations of few nodes (usually less than 500 nodes), for a short duration and over a limited field. Larger simulations usually compromise on simulation detail or duration, reduce density or restrict node mobility. JiST has complied with several self-inflicted constraints from the outset [106]:

- Do not invent a simulation language, since new languages, and especially domain-specific languages are rarely adopted by the broader community
- Do not create a simulation library, since libraries often require developers to clutter their code with simulation specific library calls and impose unnatural program structure to achieve performance
- Do not develop a new system Kernel or language runtime for simulation, since custom kernels or language run-times are rarely as optimised, reliable, featured or portable as their generic counterparts.

JiST transparently introduces simulation time execution semantics to simulation programs written in Java and they are executed over an unmodified Java virtual machine. JiST converts a virtual machine into a simulation system that is both flexible and surprisingly efficient and scalable.

One can write JiST simulation programs in plain, unmodified Java and compile them to bytecode using a regular Java language compiler. These compiled classes are then modified, via a bytecode level rewriter, to run over a simulation Kernel and to support the simulation time semantics.

The main reason to choose this simulator would be that it greatly minimises the learning curve and facilitates the reuse from building simulations. However as we shall see a more suitable simulator was found.

JavaSIM/J-Sim

J-sim [107] was formerly known as JavaSIM and is a component based simulation environment. This means that the basic entities in J-Sim are components, but unlike other component-based software packages/standards such as CORBA or JavaBeans, components in J-Sim are autonomous and are a software realisation of integrated circuits (ICs) found in electronic hardware design. The behaviour of J-Sim components is defined in terms of contracts. A system can be composed of individual components in much the same way that a hardware module is composed of IC chips. Also, components can be plugged into a software system, even during execution. J-sim has been completely developed in Java, which makes it platform independent and extensible.

J-Sim is fully integrated with a Java implementation of the Tcl interpreter (with the Tcl/Java extension), called Jacl. J-Sim was chosen as the appropriate simulation environment. The reasons for this were [107]:

- J-Sim is written in Java, this makes it platform independent and thus may run on Windows, it also means that it can be integrated with classes already developed for the p2p application. This also works in reverse; the development of the new classes required for simulations could be useful for future implementation. Development of simulator and the implementation become a single development process.

- J-sim provides Tcl scripting support that makes the integration of projects even easier as well as enabling online dynamic testing.
- J-Sim developers have also provided clean and easy to read documentation and tutorials on their website [107].
- The idea of components within J-Sim creates aide the development of simulations, it was found that this was a much more natural way to visualise the simulator class interactions than was found while evaluating the other simulators such as SSFNet, JNS and JiST.
- J-Sim has also been recommended over some of the other simulators by the well-respected NS research team as a Java alternative to NS or ns-2.

6.1.7 Simulating with J-Sim

As has been mentioned above J-Sim is a component-based simulator. This means that each element within the system is a component. When writing classes to represent network elements on top of J-Sim each element class should extend `drcl.comp.Component.` Components are connected together through ports.

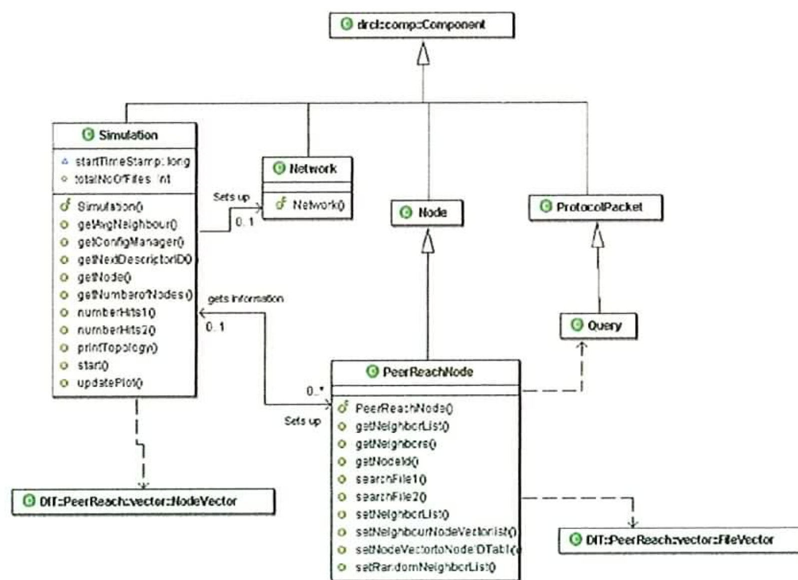

Figure 6.3. UML diagram of PeerReachNode class and simulation elements extending J-Sim's Component class.

Within the developed simulator (see Appendix C for code listing), `PeerReachNode` was written to represent a node within the system model.

`PeerReachNode` extends `drcl.comp.Component`, see (figure 6.3 above). `PeerReachNode` has one port attached to it. This port is called the "commandPort". `PeerReachNode` can be connected to other elements within the simulation through this port (see figure 6.4).

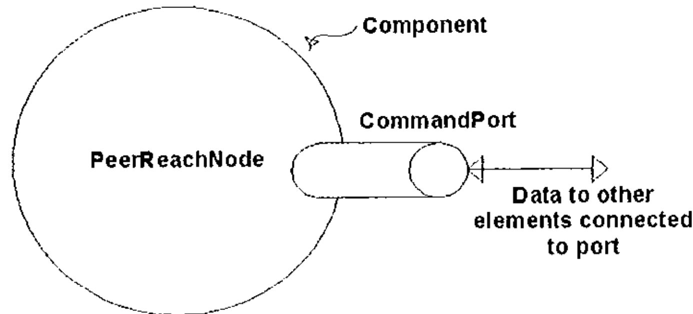

Figure 6.4. Components within J-Sim are connected together through ports.

When a class extends `component` it may override the method, `process(Port p, Object data)`. This method is called by J-Sim when data has arrived at the specific port. The implementation of `process()` within the extended class defines the components behaviour and acts as the components 'contract'. PeerReachNode is a representation of a node, within PeerReachNode `process()`. This method implements the proposed p2p algorithm developed as part of this work. When data arrives at PeerReachNode's port, it is processed by this method. The data that has arrived may be in the form of a query. A query is processed by first checking the files assigned to the specific node to ascertain if there are any query matches. This is done using the similarity metric. A file is considered to be similar if it exceeds a similarity metric value cut-off point. The similarity metric between two files x and y is calculated from the `x.compare(y)` method from the `FileVector` class. If there is a query match /hit, the node will return a query hit to the issuing node. The query may also be forwarded to the most similar nodes within the node's community table. Similar nodes are determined again by using the `compare()` method but in this case the searchvector is compared to nodevectors. Search queries will only be forwarded if the TTL of the query has not reached zero.

The simulation environment can be seen as a wrapper around the existing application code where the application outputs and inputs are stubbed and the functionality already designed is "exercised" by the simulation code.

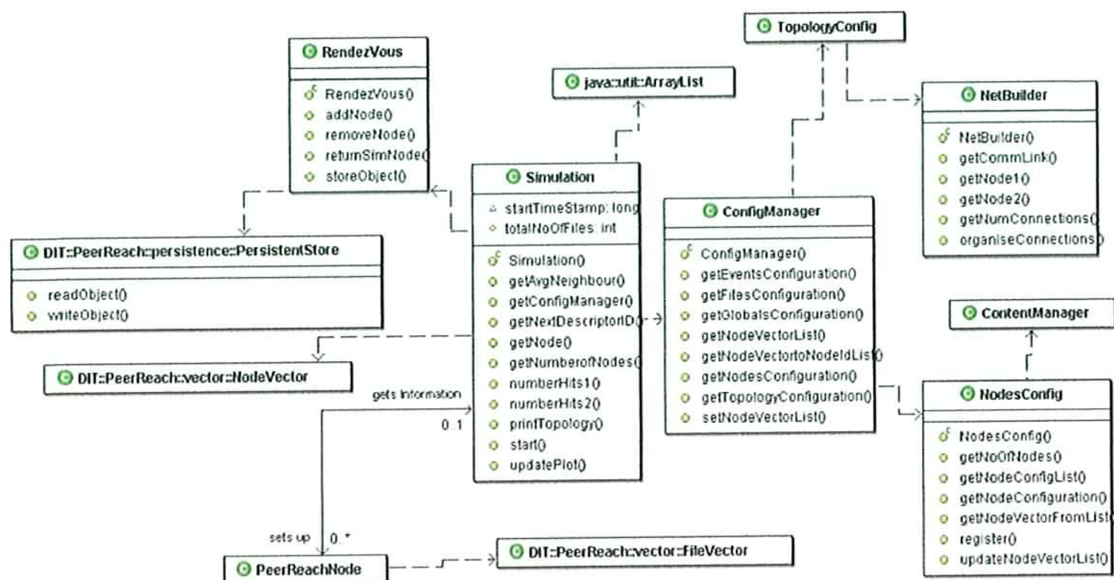

Figure 6.5. UML diagram showing the interactions of the main Simulation class with the configuration classes.

The first step in setting up a simulation involves configuring the nodes. The simulator is not required to model the Pastry routing layer as analysis of Pastry is outside the scope of this work. In order to provide rendezvous points for nodes to discover each other, the `RendezVous` class sets up *N* `ArrayLists` (see figure 6.5 above for class interactions), one for each category being modelled as in equation 6.1. The `ContentManager` then assigns content categories to each node within the simulation. The content categories are assigned by way of the content distribution model that was described in section 6.2.4. When the content categories have been assigned, the nodes register themselves in the `ArrayLists` corresponding to the category they have been assigned. Nodes also store a number of files. Each category has a number of files associated with it. A node is also assigned a number of files from the appropriate category. From these files and the file's associated filevectors, a node has the information required to calculate its nodevector. All Nodes and their associated nodeIds and nodevectors are stored and managed by a configuration manager (see `ConfigManager` in figure 6.5).

```
topology:
  node: 0
    neighbors: 6 11 13 17
    bandwidth(in/out): 520/520
  node: 1
    neighbors: 2 5 14 15
    bandwidth(in/out): 520/520
  node: 2
    neighbors: 1 5 14 15
    bandwidth(in/out): 520/520
  node: 3
    neighbors: 1 8 13
    bandwidth(in/out): 520/520
  node: 4
    neighbors: 6 7 18
    bandwidth(in/out): 520/520
  node: 5
    neighbors: 1 2 14 15
    bandwidth(in/out): 520/520
  node: 6
    neighbors: 4 7 18 0
    bandwidth(in/out): 520/520
  node: 7
    neighbors: 4 6 18 0
    bandwidth(in/out): 520/520
  node: 8
    neighbors: 1 3 13 0
    bandwidth(in/out): 520/520
  node: 9
    neighbors: 12 17 18 19
    bandwidth(in/out): 520/520
  node: 10
    neighbors: 2 9 14 15
    bandwidth(in/out): 520/520
  node: 11
    neighbors: 0 6 13 17
    bandwidth(in/out): 520/520
  node: 12
    neighbors: 9 17 18 19
    bandwidth(in/out): 520/520
  node: 13
    neighbors: 0 6 11 17
    bandwidth(in/out): 520/520
  node: 14
    neighbors: 1 2 5 15
    bandwidth(in/out): 520/520
  node: 15
    neighbors: 1 2 5 14
    bandwidth(in/out): 520/520
  node: 16
    neighbors: 1 2 5 14
    bandwidth(in/out): 520/520
  node: 17
    neighbors: 0 6 11 13
    bandwidth(in/out): 520/520
  node: 18
    neighbors: 4 6 7 9
    bandwidth(in/out): 520/520
  node: 19
    neighbors: 9 12 17 18
    bandwidth(in/out): 520/520
the average neigbour size is3.0
```

Figure 6.6. A dos print output of the topology of a simulated 20 node network; nodes discover neighbours through rendezvous points.

The next step is to populate each of the node's community tables. This happens through the classes `TopologyConfig` and `NetBuilder`. `TopologyConfig` and `NetBuilder` read in each `ArrayList` or category list and return a list of neighbours or a community list for each node. These lists are then stored within the PeerReachNode class (see figure 6.6).

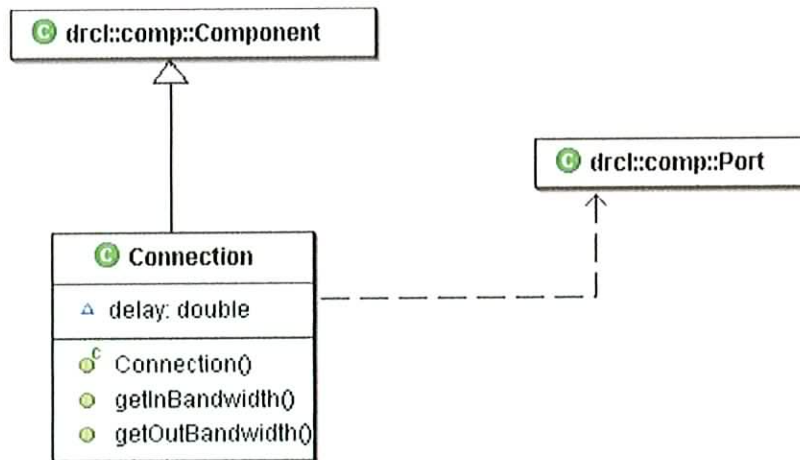

Figure 6.7 UML diagram of the connection class used to represent network connections from nodes.

Once the nodes have been configured, the next step is to configure the simulation network and the simulated physical connections for each node. Connections

are simple TCP/IP connections and are modelled by the `Connection` class. Each node has a bandwidth associated with it that must be set. Each node is connected to the network via a "connection" (see figure 6.8). The `Connection` class is also a component with associated ports for sending and receiving data.

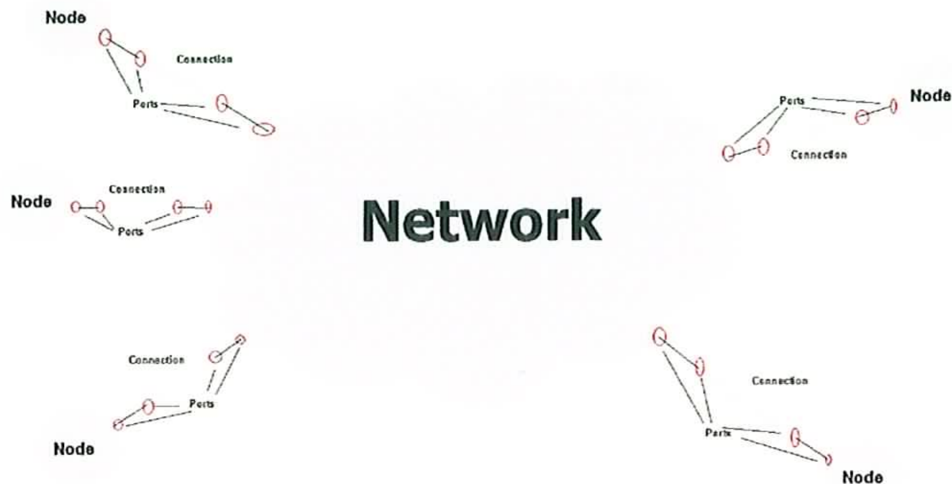

Figure 6.8.A representation of the network simulation. Each element within the network simulation is a component. Simulated nodes are connected to the simulated network via connections; simulation data is passed between elements through 'port's.

J-Sim Plotter

Having described the simulator it is also necessary to discuss the J-Sim Plotter. Using the `drcl.comp.tool.Plotter` class, it is possible to plot *xy* data gathered from running simulations. In-order to add plot functionality; the plotter must be connected to the simulation. The plotter is implemented as a component (i.e. extends the `Component` class) and must be treated as such within the simulation. As stated in the previous section, all data is passed between components through ports and, the plotter is no exception. Any *xy* data to be plotted must be passed to the plotter via its port. Within the simulator developed here, the plotter is attached directly to the `Simulation` class. The `Simulation` class then gathers the desired data during simulation and passes it to the plotter. The plotter then displays the data on a *xy* axis display.

6.1.8 Simulations and Results

The last sections described the implementation of the simulator merged with PeerReach implementation code. This section describes the simulations that were carried out using the above simulator and presents the results and the implications of these simulations.

Determining an Appropriate Similarity Metric

Determining an appropriate similarity metric cut-off point for assessing whether a search query has returned a search hit or not is difficult. It is unrealistic to suggest that it is possible to determine an accurate cut-of point for all situations and users needs. However, for the purpose of simulation and to give an approximate value for the implementation of vector space modelling developed, the calculation of an approximate value was investigated.

From one of the subject categories chosen for simulation, one PDF file and its associated vector representation was selected. The file's vector was then compared to all other vector representations of files within the same category. Note that the files contained within the category were compiled from a search on the IEEE database of conference papers [108]. It was thus thought that all similarity metrics would be quite high (i.e. close to 1). This was not the case. One file when compared, gave a similarity metric of 0.08. This was quite a surprising result, as on first glance the two papers appeared to contain similar content. However, upon closer examination of the topics of the two documents it was found that, although the documents did contain similar keywords and titles, the subject of the content was very different. This is interesting, as comparing all titles of the documents using a Boolean search techniques would have deemed them all to be 'very similar' documents. Table 6.1 gives a list of the similarity metrics calculated. After examining each document in detail and considering the similarity metric returned in each case it is the author's opinion that a value of 0.3 represents a reasonable cut-off point when using the vector package of classes developed as part of this work. It is important to note that 0.3 seems like an arbitrary value but it should be taken into account that it is impossible to quantify an appropriate cut-off point that will satisfy the requests of every query. This is the human factor. To deal with this, search applications must allow the human user to configure this parameter based on their experiences with the system. One can see a similar situation

with such search engines as Google [50] or in many every day search algorithms where on occasion the user may need to perform several searches each time varying the search parameters to find the desired object. This is an acceptable aspect of keyword searches and could be described as allowing for a margin of error from one, the search algorithm and two, the user not fully knowing what exact document is desired.

	Similarity metric compared to Doc 1
Doc 1	0.999
Doc 2	0.086
Doc 3	0.287
Doc 4	0.254
Doc 5	0.284
Doc 6	0.441
Doc 7	0.652

Table 6.1. Similarity metric of Document 1 when compared to other document within the same category.

It was also noted that while running the simulator, the value of similarity metrics produced from comparing searchvectors to nodevectors to determine what nodes a query should be forwarded to were generally of the order $x.0 * 10^{-5}$. This would suggest that it is important not to lose any precision with the value of the similarity metric or that an additional way of highlighting the differences between nodevectors is needed.

Community Based Searches Vs Non-Structured Searches

It was required by way of simulation to evaluate how well the use of community links performs when compared to a network with random connections. This can be measured by analysing how many hops and consequently the time it takes for a search query to return a certain number of results. In order to do this it was necessary to compare the duration of a query request on an unstructured network to that of a community-based network. To investigate this effect the p2p simulator was configured to set up a network of 20 nodes, each node had two different tables containing node links for forwarding queries. The first table was constructed by way of the community approach described in this thesis and the other table was chosen randomly. Two queries were generated with the same search criteria but one was forwarded over community links and the other over the random links. The number of hops 'travelled' by each query

versus the time taken was plotted until 3 hits were returned for the search query. 3 hits in such a small document represents a good margin of error should the first hit not be the desired document. Hits were determined by the file similarity metric determined above (a value of 0.3). The simulation was run a number of times to ensure accuracy and an average plot was taken. As can be seen from figure 6.9, the dashed line plot representing the query forwarded over community links consistently took less time and required a significantly smaller number of hops to return 3 query hits than the search using the random links.

These results suggest that a more structured approach to forming links within a p2p environment can reduce the number of hops a query must be forwarded over and the amount of time it takes to return a query hit. This simulation validates the assumptions made in the initial chapters that suggested a network such as Gnutella would benefit from more structured community based node links. Note: the sharp increments seen on the plots are a result of queries forwarded from one node simultaneously. The different intervals between sharp increments are the result of different nodes taking different times to search their stored files before forwarding queries. Nodes contain a different number of files due to the way in which content is assigned to nodes. Also the diagonal lines observed on the plot do not imply any useful information, and are the way in which the JavaSIM Plotter has represented the data. Point A is the point at which the search over community links returned 3 hits (it took 8 hops and a time of 0.24 simulation seconds) and point B is the point when the search over random links returned 3 hits (it took 15 hops and a time of 0.33 simulation seconds). The two plots begin at different times because they are issued by the same node, the lag represents the time taken for the node to determine the appropriate forward routes and arrive at the first hop.

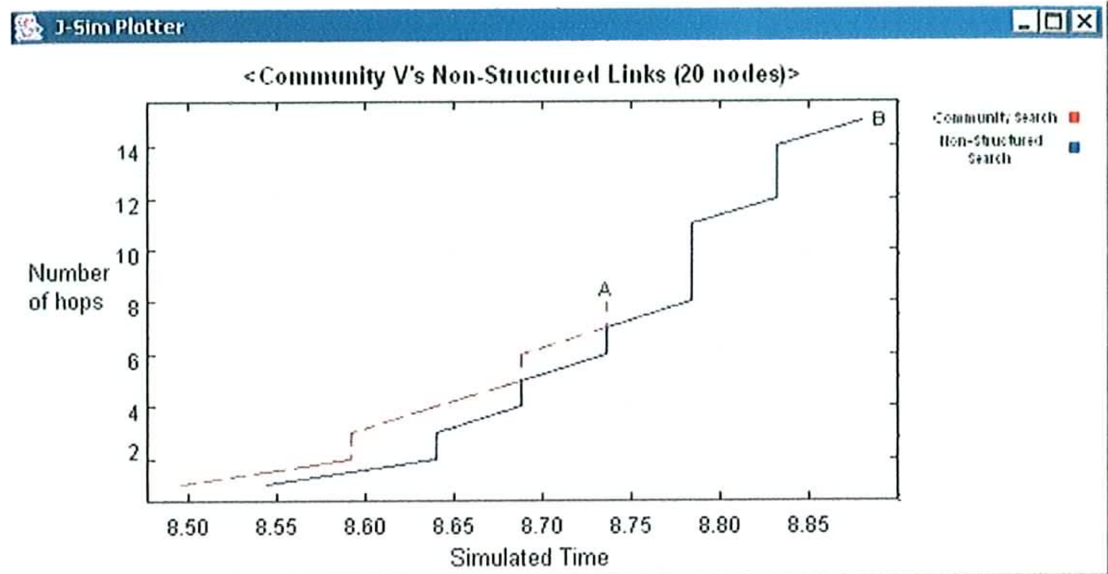

Figure 6.9. Simulation results showing that queries forwarded over community based links return query hits in significantly less time and over a lesser number of hops.

The same simulation was then run with an increased number of nodes of 100. It is both interesting and important to note how the two searches performed with an increased number of nodes. It can be seen looking at the plot in figure 6.10 that the search over the unstructured links increased by an even greater number of hops to return the required search results. It would seem to suggest that adding an equal number of extra nodes to the unstructured network and the community based network the unstructured network performance gets progressively worse in comparison to the community based network. This gives initial evidence to the original assumption that a community based approach would prove to be more scalable

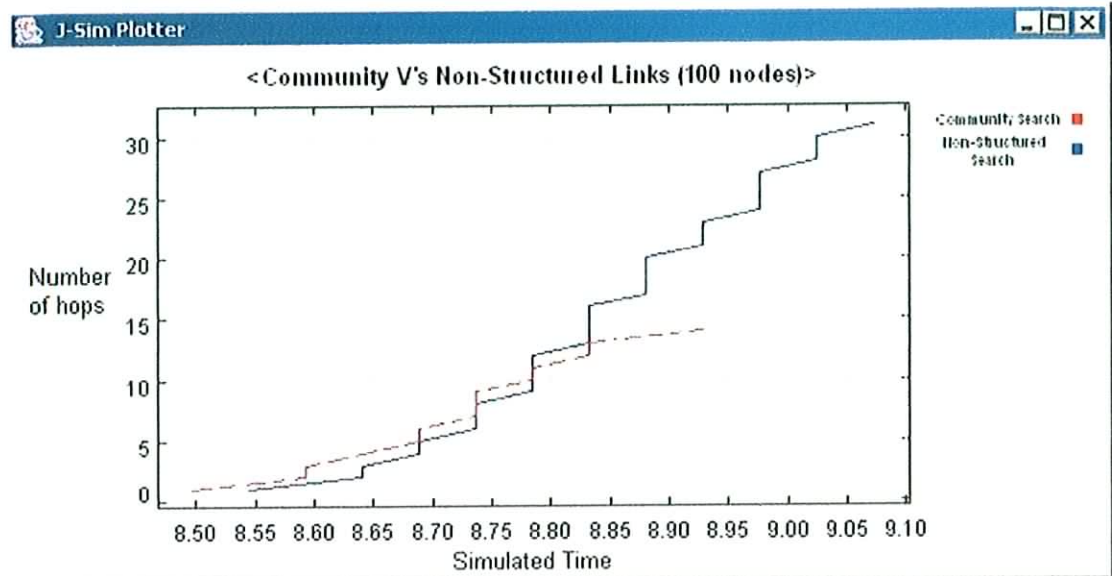

Figure 6.10. Plot comparing searches over community links and un-structured links over a 100-node p2p network.

6.2 Discussion and Future Work

Because of the increase in scale and complexity from the original idea, the work and application presented in this thesis forms the basis of an ongoing research question. The research has produced the groundwork for a fully functional system. However, continuations of this work will need to investigate a number of other aspects of the system. As discussed in section 4.3.6, cluster trees could be used as a possible method of optimising the community table. Currently the system uses a simple `ArrayList` object to index nodes. The system could also be extended to deal with objects other than research papers. These and other points are discussed below.

6.2.1 Possible Optimisations

As the number of nodes within the network increases, index tables at rendezvous points become larger. For a node to populate its community table, a subset of similar nodes from an index table needs to be selected. The reason for this is that creating links between all nodes found at a rendezvous point would be too expensive in terms of network bandwidth. Currently the index table is a flat `ArrayList`. In order to optimise this, it may be beneficial to organise nodes in a clustering tree using NodeVectors. This idea has been introduced in section 3.4.

Another potential problem that exists with the current implementation is the size of vectors representing both files and search queries. The size of a vector in a search query is a particular problem because search queries wrapping search vectors need to be

forwarded over many hops. This can use up a significant amount of bandwidth compared to keyword matching. The first optimisation would be to investigate whether a sufficiently accurate vector representation could be calculated from the abstract or even keywords. This optimisation would only cut down on initial computation time. The vectors are still of the same dimension. One possible solution to this is dimension reduction. Singular value decomposition [109] (SVD) is one method that seeks to reduce the number of dimensions to a singular dimension or value. With this method vectors may be reduced to one dimension thus significantly cutting down on the size. This would greatly improve searching, as search requests would require significantly less bandwidth. SVD is an important step in the process of latent semantic indexing [110] (LSI). LSI has many similarities to vector space modelling, and uses much of the same techniques. However, the key step in LSI is decomposing the term frequency matrix using SVD. An investigation into this technique could greatly improve the current method.

6.2.2 Future Investigation

Another point that needs to be addressed is that of node failure. This is especially important in the case of maintaining the persistence of index-tables. Other systems such as PAST have used replication of files to combat this problem. To what degree index-tables need to be replicated and a recovery mechanism for the reformation of the index-table needs to be investigated

Also, the use of a "keyphrase" to find users may not provide the accuracy needed in discovering other users with similar interests. Words have different meanings and one topic may be classed under several different keywords so an additional meta-layer may need to be added to form true "interest group" communities. Another point that needs more attention is the use of node-vectors in classifying a user's document set. This use of the nodevector could prove to be naive and a more accurate implementation of this idea may need to be investigated.

Further evaluation of the proposed idea is needed. The simulations show the reader that the system described in this thesis has the potential to perform well when compared with other systems, however more simulation is required. A full investigation of the system is needed; this would include the optimisations presented in the previous section and a full simulation model including the DHT routing layer. Along with this the simulation model does not accurately model real network connections. Simulations

have also only been run on a small number of nodes. To produce more accurate and realistic results a more scalable simulator needs to be developed. The current simulator can only simulate up to a small amount of nodes on a Pentium 4 machine running Windows 2000. Increasing the scalability of the simulator also means it is possible to assess how well the system scales to a larger number of users joining and how well it can deal with larger community tables.

6.2.3 Possible Extensions of the Work

Currently the system is being developed to deal with documents containing text. The system has the ability to deal with PDF files; this is through the use of the PJX Java API [91] for PDF software development. However the system has been designed to incorporate enough levels of abstraction so as to facilitate the sharing of other types of objects. It has been shown that vector space modeling and clustering methods have been extended to many data formats [71]. The system is open to this type of extension. This gives the system the ability to deal with objects other than documents containing text. Another useful application of this system is in the area of music file sharing. There has been much research done recently into the clustering of music through compression algorithms. For instance, the method described in [85] could be used to enable the formation of music communities with the developed system

Another application of this type of system is tackling the difficult problem of sharing fragments of Electronic Healthcare Records (EHCR) [111] [112] across an intranet or virtual private network. Notwithstanding obvious security difficulties, the work presented here could also form the basis for discovering and sharing record fragments between healthcare providers. For example, by integrating the indexing service presented here with a patient identification management system such as PIDS (patient identification service) [113], nodes storing "islands of information" could be grouped into a secure healthcare provider community. This would enable healthcare providers who were caring for a single patient to form a temporary community in order to link the scattered fragments of an EHCR for the patient in their care. This could be the focus of future work.

Building content sensitive communities using state of the art IR techniques has many far-reaching applications some of which have been mentioned above. The list and discussion is only limited by ones imagination. The underlying mechanisms presented in this thesis used to realise these communities may also be extracted and used to

provide a decentralised indexing service for many purposes. The true usefulness of p2p applications in general is still only being realised. As the technology moves forward new and interesting challenges like the challenges this thesis has addressed will come to light. P2P provides the curious researcher with a wealth of possibilities and numerous potential applications.

Chapter 7

Conclusion

Peer-to-peer networks have recently emerged into the spotlight as a research hot topic. P2P systems give users the power to search for and share content in a more free and non-restrictive environment. P2P is an exciting area of research with many interesting possibilities. However, p2p implementations such as Gnutella and Napster suffer from scalability and robustness issues. Also p2p systems for the most part only employ impoverished query languages. The scalability of systems like Gnutella is compromised by its search techniques and node organisation. Napster's robustness is comprised by the use of a centralised indexing service. Both systems employ basic search techniques such as Boolean or lexical keyword matching.

This thesis details the research and development of a system that aims to address the above problems. Robustness issues within a p2p system can be tackled through the use of a completely distributed approach to search, such as Gnutella. However as was stated above, Gnutella's architecture has been shown not to scale well. This thesis argues that a principle factor contributing to Gnutella's scalability issues is with its lack of node structure. Nodes join together in a random fashion, not taking into account the content a node is hosting or content it is interested in. Searches hop blindly from node to node consuming bandwidth. In an effort to overcome this problem, a content sensitive community layer approach to organising nodes was proposed and developed. The concept draws on the fact that nodes generally search for content that is similar to the content that they host. A node is described by the content it stores in the form of a content sensitive identifier, which is the average topic of content a node stores. It is proposed in this thesis that this gives a reasonable representation of the interests associated with that node. Node identifiers may be compared together to assess similarity between nodes and thus organise them into communities. This organisation means that nodes will be placed in contact with other nodes sharing similar content and with similar interests. As a result, searches can be directed at nodes where the content being searched for is more likely to be. This cuts down on the number of hops a search query must be forwarded over to find content. The idea and prototype presented here

has been implemented over a distributed hash-table (DHT). The DHT substrate layer provides a decentralised indexing service. This indexing service provides rendezvous points for nodes to discover other nodes and assess their similarity.

The system incorporates Vector Space Modelling (VSM) techniques to construct node identifiers. VSM is a state of the art information retrieval (IR) technique. VSM has proved more accurate than traditional Boolean or lexical searches. Employing VSM means that the system has improved search capability compared to other p2p systems. This thesis suggests that this technique may be employed in other p2p systems to tackle the issue of impoverished query languages associated with many p2p systems.

In conclusion, by building an indexing service on top of Pastry, it is possible to create a virtual space where users/researchers with similar interests can meet and discover each other in a distributed environment. This enables the construction of communities of users. Organisation of the network in this way has many advantages; it creates a more searchable and scalable file-sharing system. It also creates a more realistic representation of links between files by discovering the semantic relationships that exist through the use of Vector Space Modelling of documents. Structured p2p overlay systems provide exciting and interesting possibilities when combined with Information Retrieval (IR) techniques. These two emerging technologies complement each other in providing a way to share files and data in a distributed environment. However more work is needed to full realize this technique so as to be able to deploy it in a real world environment such as the Internet.

Bibliography

[1] *www.dictionary.com*. (Last accessed Aug 2005).

[2] The World Wide Web Consortium: *www.w3.org*. (Last accessed Aug 2005).

[3] Napster: *www.napster.com*.

[4] Gnutella Community. Gnutella Protocol Specification v0.4. *http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf*. (Last accessed Aug 2005).

[5] Salus, P.H. "Casting The Net: From ARPANET to INTERNET and beyond...". page: 104, Addison Wesley 1995.

[6] Kazaa: *www.kazaa.com*. (Last accessed Aug 2005).

[7] Rüdiger Schollmeier, "A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications". *Proceedings of the IEEE 2001 International Conference on Peer-to-Peer Computing (P2P2001)*, Linköping, Sweden, page: 101, IEEE Aug 2001.

[8] Stephen Pizzo. "Not waiting for godot". *O'Reilly Open P2P [Online]*, Dec 2000. *www.openp2p.com/pub/a/network/2000/05/12/magazine/napster.html*. (Last accessed Aug 2005).

[9] NullSoft, Inc. "The WinAmp MP3 Player". *http://www.winamp.com/*. (Last accessed Aug 2005).

[10] America Online, Inc. "AOL": *www.aol.com/*. (Last accessed Aug 2005).

[11] Marius Portmann, Pipat Sookavatana, Sebastien Ardon, Aruna Seneviratne. "The Cost of Peer Discovery and Searching in the Gnutella peer-to-peer File Sharing Protocol". *Proceedings of the Ninth IEEE International Conference on Networks*, Bangkok, Thailand, page: 263, IEEE Oct 2001.

[12] E. Adar and B. A. Huberman. "Free Riding on Gnutella". *First Monday*, Vol 5, No. 10, Oct 2000.

[13] S. Saroiu, P. Gummadi, S. Gribble. "A Measurement Study of Peer-to-Peer File Sharing Systems". *Proceedings of Multimedia Computing and Networking*, 2002.

[14] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky. "Seti@home-massively distributed computing for SETI". *Computing in Science & Engineering*, pages: 78-83, IEEE 2001.

[15] A. Oram. "PEER-TO-PEER: Harnessing the Power of Disruptive Technologies". CA 95472, USA, O'Reilly & Associates, Inc. March 2001. ISBN 0-596-00110-X pages: 432.

[16] A. Oram. "Opencola: Swarming Folders". *O'Reilly Open P2P Online publications*, O'Reilly July 2001.

[17] D. Qui, R. Srikant. "Modelling and Performance Analysis of BitTorrent-like Peer-to-Peer Networks". *In Proceedings of ACM Sigcomm*, Portland, Aug 2004.

[18] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. "Freenet: A Distributed Anonymous Information Storage and Retrieval System". *The ICSI Workshop on Design Issues in Anonymity and Unobservability"*. Proceedings Series: Lecture Notes in Computer Science, Vol. 2009, pages:46-67 Federrath, Hannes (Ed.) July 2000.

[19] Kelly Truelove and Andrew Chasin. "Morpheus out of the underworld". *O'Reilly Open P2P [Online]*, O'Reilly Nov 2001. *www.openp2p.com/pub/a/p2p/2001/07/02/morpheus.html*. (Last accessed Aug 2005).

[20] L. Gong. "JXTA: A network programming environment". *IEEE Internet Computing*, pages: 88-95, IEEE May/June 2001.

[21] C. Greg Plaxton, Rajmohan Rajaraman, Andrea W. Richa. "Accessing Nearby copies of Replicated Objects in a Distributed Environment". *Theory of computing systems*. Vol. 32, No. 3, pages: 241 – 280, Springer-Verlag 1999.

[22] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems". *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, pages: 329-350, Springer-Verlag Nov 2001.

[23] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. "Tapestry: An infrastructure for fault-resilient wide-area location and routing". *Technical Report* UCB//CSD-01-1141, pages: 27, U. C. Berkeley April 2001.

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

[25] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A scalable peer-to-peer lookup service for Internet applications". In Proc. *ACM SIGCOMM'01*, San Diego, CA, August. 2001. Pages: 149 – 160 ACM Press

[26] A. Rowstron and P. Druschel. "PAST: A Large-Scale, Persistent Peer-to-Peer Storage Utility". *HotOS VIII*, Schloss Elmau, Germany, page: 75-81, IEEE May 2001.

[27] N.J.A Harvey, M.B. Jones, S Saroiu, M Theimer, A. Wolman. "Skipnet: A scalable overlay network with practical locality properties". *In Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03).* Seattle, WA, March 2003.

[28] John Kubiatowicz, Davic Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, Ben Zhao "OceanStore: An Architecture for Global-Scale Persistent Storage" *Proceedings of ACM ASPLOS,* pages: 190-202, ACM Press May 2001.

[29] Hash-table definition: *http://www.nist.gov/dads/HTML/hashtab.html.* (Last accessed Aug 2005).

[30] A V Aho, J D Ullman, J. E. Hopcroft. "Data Structures and Algorithms". *Addison Wesley,* ISBN: 0201000237, January 1983.

[31] Kaufman, C., Perlman, R., Speciner, M. "Network Security: Private Communication in a Public World". Page: 53 of 504, Prentice Hall 2002.

[32] P. Rogaway, "Bucket hashing and its application to fast message authentication, Advances in Cryptology". *CRYPTO '95, Lecture Notes in Computer Science*, Vol. 963, Springer-Verlag, pages 313-328, 1995.

[33] D. Eastlake, 3rd and P. Jone. RFC 3174: *US secure hashingAlgorithm 1*, Sept. 2001.

[34] Jon Postel. "Internet Protocol". *Internet Request for Comments*, RFC 791, Internet RFC/STD/FYI/BCP Archives, September 1981. *http://www.faqs.org/rfcs/rfc791.html.* (Last accessed Aug 2005).

[35] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. "Tapestry: A resilient global-scale overlay for service deployment". *IEEE Journal on Selected Areas in Communications,* Vol. 22, No. 1, pages. 41-53, IEEE Jan 2004.

[36] G. Kessler, S. Shepard. "A Primer On Internet and TCP/IP Tools and Utilities", *Internet Request for Comments*, RFC 2151, Internet RFC/STD/FYI/BCP Archives, June 1997. *http://www.faqs.org/rfcs/rfc2151.html.* (Last accessed Aug 2005).

[37] S. Ratnasamy et al. "Routing Algorithms for DHTs: Some Open Questions". *In Proceedings of the 1st International Workshop on Peer-to-Peer Systems,* pages: 45 – 52, Springer-Verlag March 2002.

[38] Dabek, F., Zhao, B., Druschel, P., and Stoica, I. "Towards a common API for structured peer-to-peer overlays". *In 2nd International Workshop on Peer-to-Peer Systems IPTPS'03.* Pages: 33-44, Springer-Verlag Feb. 2003.

[39] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica. "Wide-area cooperative storage with CFS". *In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages: 202 – 215 ACM press 2001.

[40] Stacey P., Berry D., Coyle E., "Using Structured P2P Overlay Networks to Build Content Sensitive Communities". *In Proceedings of the Tenth International Conference on Parallel and Distributed Systems,(ICPADS'04)*, Newport Beach California, pages: 281-288, IEEE 2004.

[41] Ryan Huebsch Boon T. Loo Scott Shenker Matthew Harren, Joseph M. Hellerstein and Ion Stoica. "Complex queries in DHT based peer-to-peer networks". *Electronic Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, 2002, pages: 242 – 250, Springer-Verlag 2002.

[42] Pietro Braione, Politecnico di Milano. "A Semantical and Implementative Comparison of File Sharing Peer-to-peer Applications". *In Proceedings of the Second International Conference on Peer-to-Peer Computing (P2P'02)*, pages: 165-166, IEEE 2002.

[43] C.J. van Rijsbergen. "Information retrieval (second edition)", in London: Butterworths. 1979.

[44] E. Cohen, H. Kaplan, and A. Fiat, "Associative search in peer-to-peer networks: Harnessing latent semantics". *The 22nd Annual Joint Conference of the IEEE Computer and Communications Societies. San Francisco 2003*, pages:1261 – 1271, IEEE 2003.

[45] K. P. Sycara. "Multi agent Systems". *Artificial Intelligence Magazine*, Vol. 19:2 pages: 79-92, 1998.

[46] Kung, H. T., Wu, C. H. "Content Networks: Taxonomy and New Approaches." *In Proceedings of the Internet as a Large scale Complex System*". Oxford University Press, Santa Fe Institute Series, 2002.

[47] Ricardo Baeza-Yates, Berthier Ribiero-Neto, Berthier Ribeiro-Neto. "Modern Information Retrieval". Page: 7 of 513, Pearson Education 1999. ISBN: 020139829X.

[48] R. Wurman. "Information Anxiety". Doubleday, New York, 1989. ISBN: 0385243944.

[49] S. Lawrence, C.L. Giles. "Accessibility of Information on the Web". Nature, pages: 107-109, Feb 1999.

[50] Google: *www.google.com*. (Last accessed Aug 2005).

[51] Yahoo: *www.yahoo.com*. (Last accessed Aug 2005).

[52] C. F. Reynolds. "The Use of Colour in Language Syntax Analysis". *In Software-Practice and Experience*, Vol. 17, Issue 8, pages 513-519, John Wiley & Sons, Inc 1987.

[53] "Resource Description Framework (RDF) model and syntax specification". *W3C Working Draft WD-rdf-syntax-19981008*. See *http://www.w3.org/TR/PR-rdf-syntax/*, W3C 1999.

[54] T. Guan and K. F. Wong. "KPS - A Web Information Mining Algorithm". *The Eighth International World Wide Web Conference*, pages: 1495 - 1507 , Elsevier 1999.

[55] T. Berner-Lee, J. Hendler, O. Lassila. "The Semantic Web". *Scientific American*, Vol. 284, No. 5, pages: 34-43, 2001.

[56] K.A. Spackman, K.E. Campbell, and R.A. Cote. "Snomed rt: A reference terminology for health care". *In AMIA Annual Fall Symposium*, 1997. pages: 640-644, PubMed: PMID: 9357704.

[57] Rector AL, Gangemi A, Galeazzi E, Glowinski AE, and Rossi-Mori A. "The GALEN CORE model schemata for anatomy: Towards a re-usable application-independent model of medical concepts". *In Proceedings 12th International Congress on Medical Informatics MIE '94*, Lisbon, pages: 229-233, Lisbon 1994.

[58] ICD-10. *http://www.cdc.gov/nchs/about/otheract/icd9/icd10cm.htm*. (Last accessed Aug 2005).

[59] A. Rossi-Mori and CEN Project Team PT003, "Medical Informatics - Categorial structure of systems of concepts - Model of Representation of Semantics (ENV12264)", 1995.

[60] Luhn, H.P. "A statistical approach to mechanised encoding and searching of library information", *IBM Journal of Research and Development, 1*, pages.309-317, IBM 1957.

[61] Salton, G., Wong, A., and Yang, C. S. "A Vector Space Model for Automatic Indexing". *Communications of the ACM*, pages: 613 - 620, ACM Press, Nov 1975.

[62] Willet P. "Similarity coefficients and weighting functions for automatic document classification an empirical comparison". *International Classification, 3*, pages:138-142, 1983.

[63] Jones, W., and Furnas, G. "Pictures of Relevance: A Geometric Analysis of Similarity Measures". *Journal of the American Society for Information Science*, Vol.38 No.6, pages: 420-442,Wiley Nov 1987.

[64] Salton, G, McGill M.J. "Introduction to Modern Information Retrieval". *McGraw-Hill*, New York, 1983.

[65] Computer Programming Algorithms Directory: *http://www.algosort.com/*. (Last accessed Aug 2005).

[66] G. Salton and C. Buckley. "Term-weighting Approaches in Automatic Text Retrieval". *Information Processing and Management, 24*, pages: 513-523, Pergamon Press Ltd. 1998.

[67] Han, E-H. and Karypis, G. "Centroid-Based Document Classification: Analysis and Experimental results". *In Proceedings of the 4th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, page: 424 Springer Sep 2000.

[68] B. Zadrozny and C. Elkan. "Obtaining calibrated probability estimates from decision trees and naive bayesian classifiers". *In Proceedings of the Eighteenth International Conference on Machine Learning, 2001*, pages: 609-616, Morgan Kaufmann 2001.

[69] A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility". *ACM Symposium on Operating Systems Principles (SOSP'01)*, Banff, Canada,. Pages: 188 – 201 ACM Press Oct 2001.

[70] Quinlan, J. R.. "C4.5: Programs for machine learning". Morgan Kaufmann. 1993, ISBN :1-558-60238-0.

[71] Jain, A.K., Murty M.N., and Flynn P.J. "Data Clustering: A Review". *ACM Computing Surveys*, Vol. 31 No. 3, pages: 264-323 ACM Press Sept 1999.

[72] Y. Suzuki, F. Fukumoto, Y. Sekiguchi. "Keyword Extraction using Term-Domain Interdependence for Dictation of Radio News".*Coling-ACL*, pages: 1272-1276, 1998.

[73] Jones. W.P. and Furnas, G.W. "Pictures of relevance: A geometric analysis of similarity measures". *Journal of the American Society for Information Science.* Vol. 38 No. 2, pages: 207-227. 1987

[74] P. Willet. "Recent trends in hierarchical document clustering: a critical review". *Information Processing and Management*, Vol. 24 No. 5 pages: 577-597 Elsevier Science 1988.

[75] E. M. Voorhees. "Implementing agglomerative hierarchical clustering algorithms for use in document retrieval". *Information Processing and Management*, pages: Vol. 22 No. 6, pages 465-476, Elsevier Science 1986.

[76] J.J. Rocchio. "Relevance feedback in information retrieval". In *The SMART Retrieval System--Experiments in Automatic Document Processing*, pages 313-323, Englewood Cliffs, NJ, Prentice Hall, Inc. 1971.

[77] D. R. Hill. "A Vector Clustering Technique". *Mechanised Information Storage, Retrieval and Dissemination*, North-Holland, Amsterdam, edited by Samuelson 1968.

[78] Krishna Gummadi, Ramakrishna Gummadi, Steve Gribble, Sylvia Ratnasamy, Scott Shenker, Ion Stoica. "The Impact of DHT Routing Geometry on Resilience and Proximity". *In Proceedings of ACM SIGCOMM*, pages: 381 - 394 ACM Press 2003 .

[79] Ben Y. Zhao. "A Decentralized Location and Routing Infrastructure for Fault-tolerant Wide-area Applications". *Ph.D. Qualifying Examination*, pages: 13,April 18, U.C. Berkley 2001.

[80] Mario T. Schlosser, Tyson E. Condie, and Sepandar D. Kamvar, "Simulating a P2P file-sharing network". *Appears in the First Workshop on Semantics in P2P and Grid Computing*, pages: 69-80 December 2002.

[81] Dublin core: *http://dublincore.org/*. (Last accessed Aug 2005).

[82] Ronald L. Rivest. "The MD5 message-digest algorithm". *Internet Request for Comments*, RFC 1321, Internet RFC/STD/FYI/BCP Archives, pages: 21, April 1992. *http://www.faqs.org/rfcs/rfc3797.html.*

[83] "Descriptions of SHA-256, SHA-384, and SHA-512", pages: 48, National Institute of Standards and Technology, Washington, 2000. *http://csrc.nist.gov/cryptval/shs/sha256-384-512.pdf* (Last accessed Aug 2005).

[84] ADLER-32, *http://www.faqs.org/rfcs/rfc1950.html.* (Last accessed Aug 2005).

[85] R. Cilibrasi, R. de Wolf, P. Vitanyi. "Algorithmic clustering of music", *ERCIM News R&D AND TECHNOLOGY TRANSFER*, No. 54, ERCIM News July 2003, *http://www.ercim.org/publication/Ercim_News/enw54/vitanyi.html.* (Last accessed Aug 2005).

[86] M. Castro, P. Druschel, Y. C. Hu and A. Rowstron, "Exploiting network proximity in peer-to-peer overlay networks". *International Workshop on Future Directions in Distributed Computing (FuDiCo)*, pages: 52-55, Springer-Verlag 2002.

[87] LimeWire. *http://www.limewire.com*. (Last accessed Aug 2005).

[88] Stephen A. Stelting, Olav Maassen "Applied Java Patterns". Pages: 608, Prentice Hall 2001, ISBN: 0130935387.

[89] JAMA: *http://math.nist.gov/javanumerics/jama/*. (Last accessed Aug 2005).

[90] "Porter Stemming Algorithm" *Readings in Information Retrieval*, pages: 513, Morgan Kaufmann, ISBN 1-55860-454-4.

[91] *PJX: http://www.etymon.com/epub.html.*

[92] The MathWorks Inc. "MATLAB Reference Guide", *http://www.mathworks.com/*. (Last accessed Aug 2005).

[93] D. L. Less, H. Chuang, K Seamons. "Document Ranking and the Vector-Space Model". *IEEE Software*, Vol 14, no 2, pages: 67-75, March/April 1997.

[94] Carlos R Cunha, Azer Bestavros, Mark E Crovella, "Characteristics of WWW client-based traces". *Technical Report TR-95-010*, pages: 18, Boston University June 1995.

[95] UCB/LBNL/VINT "Network Simulator NS (version 2)" *http://www-mash. cs.berkeley.edu/ns/*. (Last accessed Aug 2005).

[96] J. K. Ousterhout. "Tcl and the Tk Toolkit". Addison-Wesley, 1994.

[97] J. Postel, J. Reynolds. "File Transfer Protocol (FTP)". RFC 959, October 1985.

[98] J. Postel, J. Reynolds. "Telnet Protocol Specification". RFC 845, May 1983.

[99] A. Felmann, A. Gilbert, P. Huang, W. Willnger. "Dynamics of IP Traffic: A study of the Role of Variability and the Impact of Control". *Proceedings of ACM/SIGCOMM*, pages: 301-313, 1999.

[100] A. Veres, M. Boda. "The chaotic Nature of TCP Congestion Control". *Proceedings of IEEE INFOCOM*, pages: 1715-1723, March 2000

[101] P. Sinha, T. Nandagopal, N. Ventitaraman, R. Sivakumar, V. Bharghaven. "WTCP: A Reliable Transport Protocol for Wireless Wide-Area Networks". *Wireless Networks*, Vol. 8, No. 2-3, pages: 301-316, 2002.

[102] Scalable Simulation Framework (SSF). *http://www.ssfnet.org*. (Last accessed Aug 2005).

[103] J. H. Cowie, D. M. Nicol, A. T. Ogielski. "Modeling the global internet". *Computing in Science and Engineering*, January 1999.

[104] The Java Network Simulator (JNS) *http://homepages.cs.ncl.ac.uk/einar.vollset/home.formal/jns.html*.

[105] Javis, Network Visualisation tool, *http://cs.baylor.edu/~donahoo/NIUNet/javis.html* . (Last accessed Aug 2005).

[106] JiSt (Java in Simulation Time), *http://jist.ece.cornell.edu/*. (Last accessed Aug 2005).

[107] J-Sim, *www.j-sim.org*. (Last accessed Aug 2005).

[108] IEEE Explore, *http://ieeexplore.ieee.org/*. (Last accessed Aug 2005).

[109] M. T. Heath, A. J. Laub, C. C. Paige, R.C. Ward. "Computing the SVD of a product of two matrices". SIAM J. Sci. Statist. Comput., Vol. 7, pages: 1147-1159, 1987.

[110] S. T. Dumain. "Latent Semantic Indexing (LSI)". *The Second Text Retrieval Conference (TREC2)*, National Institute of Standards and technology Special Publication, pages: 105-116, 1994.

[111] Jane Grimson, William Grimson, Damon Berry, Gaye Stephens, Eoghan Felton Dipak Kaltra, Pieter Toussaint, Onno W. Weier. "A CORBA-based integration of distributed electronic healthcare records using the Synapses approach". *IEEE Transactions on Information Technology in Biomedicine*, pages: 124-138, IEEE Engineering in Medicine and Biology Society 1998

[112] Jane Grimson, Eoghan Felton, Gaye Stephens, William Grimson and Damon Berry. "Interoperability issues in sharing electronic healthcare records - the Synapses approach". *Proceedings of Third IEEE International Conference on Engineering of Complex Computer Systems*, pages: 180-185, IEEE 1997.

[113] "Person Identification Service Specification (PIDS"). *Formal specification*, formal/01-04-04, version 1.1, OMG 2001.
http://www.omg.org/docs/formal/01-04-04.pdf (Last accessed Aug 2005).

[114] "Unified Modeling Language (UML), v1.5". *Formal specification*, formal/03-03-01, version 1.5, OMG 2004.
http://www.omg.org/cgi-bin/doc?formal/03-03-01. (Last accessed Aug 2005).

[115] Omondo EclipseUML: *http://www.omondo.com/*. (Last accessed Aug 2005).

[116] Eclipse: *http://www.eclipse.org/*. (Last accessed Aug 2005).

Glossary

Access scheme - Plaxton et al's set of algorithms for read, insert and delete access to objects within their proposed network.

Agent - A concept associated with network simulation. Agents attach to the application and are used to simulate network protocol behaviour.

Agglomerative hierarchical clustering - A clustering algorithm used to group together "similar" objects.

AHC - See Agglomerative hierarchical clustering.

Angle based metric – A similarity metric determined by the angle between two vectors.

API - See application programming interface.

Application programming interface – A set of external program libraries imported into new programs to reuse already developed software functionality.

Avalanche effect - A property of hashing algorithms refers to the algorithms ability to produce two non-numerically similar hash-codes given similar input strings.

Avalanche effect collision – When a hashing algorithm produces the same output key for two different inputs.

Average vector – The vector resulting from taking the average of a group of document vectors, see also centroid vector.

C++ - An object oriented programming language.

CAN – A DHT p2p algorithm, where nodes are organised based on a cartesian coordinate system.

Cartesian Coordinate Space - The coordinates of a point measured from an origin along an (horizontal) axis from left to right (the x-axis) and along a perpendicular (vertical) axis from bottom to top (the y-axis).

Category – Refers to the definition of the "type" of a document based on the document's contents.

Centroid classification – A document classification method.

CFS - A global storage system that uses the Chord p2p substrate.

Chord - A DHT p2p substrate. Nodes are organised in a ring similar to Pastry.

Cluster – The act of grouping together "similar" objects.

Clustering tree – A method for grouping "similar" objects. Objects that are deemed similar using a similarity metric are positioned close on a tree where as non-similar objects are positioned further away.

Community – A group of users of the Peerreach application who are deemed to have similar interests and as a result are virtually connected to each other.

Community based search – A search performed over a community of users machines.

Community Layer – Level of network activity, where connections are maintained based on user similarity, thus implementing the community concept.

Community table – A storage location used to keep track of the addresses of nodes within a community.

Cosine function – A mathematical function, which calculates the cosine of the angle between two vectors.

Cryptographic hash - Bit strings of a fixed length produced from a hash function by mapping bit strings of arbitrary length.

Decentralised p2p system – A computer network of nodes with no centralised form of control.

Density – (of nodes) The number of nodes populating a specific area within the network.

Destination node – The node a message is destined for, i.e. its intended point of delivery.

DHT - see Distributed Hash-Table.

Distributed Hash-Table – A distributed system that attempts to model its self on a hash-table but is implemented in a distributed manner.

Document frequency – The number of documents a particular word occurs from a particular set of documents.

Document vector - A vector space representation of a text document, calculated using the word frequency within a particular document and using the document frequency from a set of documents.

Dot product - A number (scalar) equal to the product of the magnitudes of any two vectors and the cosine of the angle formed between them.

EHCR - See electronic healthcare record.

Electronic healthcare record - the set of electronic information that is stored about a single patient over their lifetime.

File Id - See File Identifier.

File Identifier- A number that is used to uniquely identify a particular file.

File Vector - See document vector.

Fan out – The number and diversity of neighbouring nodes that a particular node in a DHT substrate has.

Feature vector - A feature vector characterises a document for classification in the peerreach application.

Flooding - see flood searching.

Flood searching – A method of searching for objects within a network where the search request if not forwarded to any specific node but to all possible nodes.

Fuzzy community – A group of nodes that are virtually linked together based loosely on the type of content they store.

Get function – The function used to return an object from the hash-table.

Gnutella – A specific implementation of a p2p network.

Hash-code – The output of a hashing algorithm.

Hashing algorithm – See hash function.

Hash function – A hash function takes a variable length data message and creates a fixed size message digest.

Hierarchic clustering – A specific method of clustering "similar" objects.

Index table – A list of nodes and the associated node information that is stored at an indexing node.

Indexing node – The node storing an index table, an indexing node acts as a rendezvous point within Peerreach, registering "similar" nodes.

IR – See Information Retrieval.

Information retrieval – The science of finding stored information.

Inverse document frequency – The inverse of the number of documents a particular word occurs from a set of documents. Used in the calculation of a document vector.

J2SE - The current (at time of writing) standard edition of the Java programming language.

JAMA – A basic linear algebra package for Java that is compatible with J2SE.

Java Network Simulator – A network simulator implemented in Java.

Java – An object oriented programming language.

JavaSIM – A Network simulator implemented in Java.

JDK – The Java development Kit by SunSoft.

JisT – A network simulation software package written in Java.

JNS - See Java Network Simulator.

JSIM - A network simulation software package, written in Java.

Kazaa – P2P file sharing application that uses the idea of super nodes to aid the indexing of files.

KBR - See Key based Routing.

Key – A number that used to direct a message over a network to its destination.

Key randomisation – The process of randomly assigning key's or object Id's.

Key based Routing – A message routing algorithm that uses keys, a message is routed to a point within the network that corresponds to the key, messages progress by forwarding the message to closer and closer to the destination point

Latent semantic indexing - A statistical information retrieval method designed to overcome synonomy.

Limewire – A p2p file sharing application.

Linear time clustering – A specific type of clustering algorithm.

Leaf Set – A table containing information about neighbouring nodes within a Pasty DHT.

Load – The level of network traffic concentrated at a specific point within the network

Load Balancing – Method of evenly distributing network traffic to avoid bottlenecking.

Logical hop – A hop between nodes within a DHT, ignoring the underlying network topology.

LSI - see Latent semantic indexing.

Matlab $_{TM}$ - A programming language for mathematical modelling that employs matrix arithmetic.

Multiple traffic simulator – A simulator that can simulate multiple sources of traffic within a simulated network.

NAM - See network animator.

Neighbour Table – A storage location used to keep track of "neighbouring" nodes within a network.

Network animator – A network visualisation tool.

Network distance – The "real" distance a packet or message must travel, i.e. takes into account the physical under lying distance.

Network simulator – A software tool used to simulate the behaviour of a network under various scenarios

Node – Physically a node is a computer running an instance of the PeerReach application. Virtually a node is a point within the network containing routing tables and community tables. A node acts as a servant, i.e. it serves information to the network and also becomes a client within the network. A node also represents a user within the network.

NodeId – A number used to uniquely identify a node within a network

NodeVector – A representation of a node based on the documents the node stores locally.

Non- structured search – see flood search.

NS - A network simulator.

NS-2 - A network simulator.

Object Id –Number to uniquely identify an object.

Object Location – The location of a stored object within a distributed hash table.

Observation point - Selected moments in time where a network simulator records values.

Ocean Store - A global storage system that uses the Tapestry DHT substrate.

Overlay namespace distance – The numerical difference between two nodes nodeId's.

P2P - See Peer to Peer.

PAST - A global storage system that uses the Pastry DHT substrate.

Pastry – A DHT routing substrate.

Pastry Layer – The position occupied by the Pastry substrate within a system layer view.

PDF – See portable document format.

Peer – A node within a p2p system.

Peerreach – The name of the application described by this thesis.

Peer to Peer – Distributed system with no centralised control.

Person Identification service - A specification produced by the object management group that describes how multiple identities existing in distributed systems can be merged.

PIDS - see Person Identification service.

PJX - A Java API for manipulating PDF files.

Pointer List – A list containing connection information to nodes.

Proxy searcher – A node that searches on behalf of another node.

Put function - The functionality of a hash-table that is used to store an object in the hash-table.

Registry message – An extended pastry message used to register a nodes details at a rendezvous point.

Rendevous point – A node within the network where other nodes sharing similar content can be indexed in order to discover each other.

Root node – the node, whose nodeId is numerically similar to a specific key, that becomes the root node for that key.

Routing schema – A set of routing algorithms that facilitate the reading, writing and insertion of objects within a DHT.

Routing Substrate – The software layer that handles the routing of network messages.

Search query – A message forwarded to nodes requesting a particular file.

Search vector – A vector derived from search keywords, that are used to search for documents by being compared to a documents FileVector.

Self organising – A network that organises itself with out any centralised control

Semantic – Relating to the meaning of a particular document or string of text.

SHA-1 - A hashing algorithm.

SHA-256 - A hashing algorithm.

Similarity metric – A metric to represent the similarity between two objects.

Single point of failure – A node that stores the only copy of network link information

Singular value decomposition – The process of mapping a multi dimensional space to a one-dimensional space.

SSFNet – A network simulator.

Stop word – A commonly occurring word within a document that contains no semantic meaning.

Structured (p2p) substrate – A substrate that organises nodes in a structured fashion.

Substrate (p2p) – A routing layer that sits under the application layer in a p2p software system.

Surrogate Routing – Tapestry's distributed algorithm for incrementally calculating a root node.

SVD - See singular value decomposition.

Synonyms - a word or phrase, which has the same or nearly the same meaning as another word or phrase in the same language.

Tapestry – A DHT substrate.

Taxonomy - A system for naming and organising things, especially plants and animals, into groups, which share similar qualities.

TCL – A scripting language for issuing commands to interactive programs.

TCP/IP - A set of protocols developed for the Internet in the 1970s to get data from one network device to another.

Term space – The global dictionary of terms or words used in the vector space modelling of documents.

Term frequency – The number of occurrences of a word within a specific document.

Time to live stamp – The number of logical hops a message may jump before "dying".

TTL - See time to live stamp.

UML – See unified modelling language.

Unified Modelling Language - The Unified Modelling Language is the industry-standard language for specifying, visualising, constructing, and documenting the artefacts of software systems.

Vector space modelling – A information retrieval technique where documents are represented as vectors from a term-space. Vector space modelling allows the user to search for concepts rather than specific words.

Zipf Law - The probability of occurrence of words or other items starts high and tapers off. Thus, a few occur very often while many others occur rarely.

Zipf Distribution - A distribution of probabilities of occurrence that follows Zipf's law

Appendix A

This chapter gives a code listing for the more important classes that were implemented to realise a prototype application, described in chapter five.

A.I. Package DIT.PeerReach

This is the main package within the system. This package contains the driver classes that contain `main()` and extend the Pastry Application interface.

A.I.i PeerReachAppl

PeerReachAppl is the driver class for the application. The main method is defined within this class.

```
package DIT.PeerReach;

import java.net.InetSocketAddress;
import java.util.*;
import java.io.*;
import java.net.*;
import rice.pastry.*;
import rice.pastry.direct.*;
import rice.pastry.dist.*;
import rice.pastry.standard.*;
import DIT.PeerReach.node.*;
import DIT.PeerReach.register.*;
import DIT.PeerReach.vector.*;

/**
 *  This class is the driver class for the application. This class and PeerReachNode
 *  take the same format as the similar classes provided in the Pastry API testing
 *  example "Pastry HelloWorld" At present the main creates
 *  nodes to run on one local machine, this for the purpose of testing the prototype
 *  @author Paul Stacey
 */
public class PeerReachAppl {

        private static int i;
        private PastryNodeFactory factory;
        private DistPastryNodeFactory factoryran;
        private Vector helloClients;
        private NodeHandle hint = null;
        private static RegistryManager rm;
        private NodeIdFactory idfact;
        private NetworkSimulator simulator;
        private DistPastryNode node;
        private static int port = 510;
        private int bsport = 510;
        private String userInfo;
        private Vector pastryNodes;
        private static String bshost = null;
        private String uInfo;
        private String[] interesttopic = { "x", "y", "z" };
        private String interesttopicx;
        //use the Pastry RMI protocol to send messages between peers
        //Pastry also provides the direct and wire protocol
        public static int protocol = DistPastryNodeFactory.PROTOCOL_RMI;
        private NodeVector nodeVector = null;
        private static int rmiport = 1099;

        /**
         *   Constructor.
```

```
 *   Initialise the class variables
 */
public PeerReachAppl() {
        pastryNodes = new Vector();
                factoryran = DistPastryNodeFactory.getFactory(
                        new RandomNodeIdFactory(),
                        protocol,port);
}
/*
 * This method looks for a bootstrap node to join the network, at the moment
 * it will return a null, this null is handled within the pastry class
 * RMIPAstryNodefactory
 */
private DistNodeHandle getBootstrap(boolean firstNode) {

        InetSocketAddress addr = null;
        if (firstNode && bshost != null)
                addr = new InetSocketAddress(bshost, bsport);
        else {
                try {
                        addr =
                                new InetSocketAddress(
                                        InetAddress.getLocalHost().getHostName(),
                                        bsport);

                //bottom line used when connecting to a reomte computer
                // addr = new InetSocketAddress("192.168.0.1",bsport);
                } catch (UnknownHostException e) {
                        System.out.println(e);

                }
        }

        DistNodeHandle bshandle =
                                (DistNodeHandle)((DistPastryNodeFactory)factory).ge
                                tNodeHandle( addr);
        return bshandle;
}

/*
* Creates a Pastry node
*
*/

public void MakeNode(boolean firstNode) {

        BufferedReader in =
                new BufferedReader(new InputStreamReader(System.in));
        System.out.println("\nplease enter your ip address\n");
        //The user input string is used to create the NodeId for the node
        try {
                uInfo = in.readLine();
        } catch (IOException e) {
                System.out.println(e);
                System.exit(1);
        }
        idfact = new ConcPRNodeIdFactory(uInfo);
        factory = DistPastryNodeFactory.getFactory(idfact, protocol, port);
        node = (DistPastryNode) factory.newNode(getBootstrap(firstNode));
        pastryNodes.addElement(node);
        //create a new PeerReachNode
        PeerReachNode prnode = new PeerReachNode(node, rm, rmiport, nodeVector);
        if (Log.ifp(5))
                System.out.println("\nplease enter your interest subject\n");
        try {
                interesttopicx = in.readLine();
        } catch (IOException e) {
                System.out.println(e);
                System.exit(1);
        }
```

System.out.println("\n You entered "+ interesttopicx+ " please wait while you are connected with others having the same interest");

```
                        Register(interesttopic, prnode);
}
```

```
	/**
	 *   Causes this node to be registered at various rendezvous points around the
	 *   Pastry Ring.
	 */
	public void Register(String[] interests, PeerReachNode prn) {
		rm.setNodeInstance(prn);
		rm.setInterests(interests);
		rm.setNodeVector(nodeVector);
		rm.register();
	}
	public void setNodeVector(NodeVector nvector) {
		nodeVector = nvector;
	}
	/**
	 *  This method is for testing purposes, it creates a random node. This is
	 *  used in main to fill the Pastry ring with random nodes
	 *
	 */
	public void makerandomNode(boolean firstNode) {
		NodeHandle bootstrap = getBootstrap(firstNode);
		DistPastryNode pnn = (DistPastryNode) factoryran.newNode(bootstrap);
		// internally initiateJoins
		pastryNodes.addElement(pnn);
		PeerReachNode app = new PeerReachNode(pnn, rm, rmiport, nodeVector);
	}
	/**
	 *   Main
	 */
	public static void main(String args[]) {

		PeerReachAppl preachappl = new PeerReachAppl();
		VectorManager vm = new VectorManager();
		vm.buildTrainingSet();
		vm.buildNodeVector();
		NodeVector nv = vm.getNodeVector();
		preachappl.setNodeVector(nv);
		rm = new RegistryManager(rmiport);
		preachappl.MakeNode(true);
		rmiport++;
		for (int j = 1; i < 4; i++) {
			rm = new RegistryManager(rmiport);
			preachappl.makerandomNode(false);
			rmiport++;
		}
		for (i = 1; i <= 1; i++) {
			rm = new RegistryManager(rmiport);
			preachappl.MakeNode(false);
			rmiport++;
			System.out.println(rmiport);
		}
	}
}
```

A.I.ii PeerReachNode

```
package DIT.PeerReach;

import rice.pastry.*;
import rice.pastry.client.*;
import rice.pastry.security.*;
import rice.pastry.messaging.*;
import rice.pastry.routing.*;
import rice.pastry.dist.*;
import DIT.PeerReach.message.*;
import DIT.PeerReach.register.*;
import DIT.PeerReach.community.*;
import DIT.PeerReach.vector.*;
import java.net.*;

/**
 *   This class extends the Pastry API, CommonAPIAppl.
 *   @author Paul Stacey
 */

public class PeerReachNode extends CommonAPIAppl {
```

```
private static Address addr;
private static Credentials cred = new PermissiveCredentials();
private MessageManager mm;
private RegistryManager regm;
private int rmiport;
private PRNodehandle prnh;



/**
 *  Constructor.
 *  Initialises the nodes attributes.
 */
public PeerReachNode(DistPastryNode pn,RegistryManager rm,int port, NodeVector
                                                                   nodevec) {

        super(pn);
        rmiport = port;
        DistNodeHandle dnh = (DistNodeHandle) pn.getLocalHandle();
        InetSocketAddress ipaddr = dnh.getAddress();
        regm = rm;
        mm = new MessageManager(regm);
        regm = rm;
        CommunityServer cm = new CommunityServer(this, port);
        Integer p = new Integer(port);
        String Port = p.toString();
        String RmiConnect = "CommunityManager//127.0.0.1/" + Port;
        prnh = new PRNodehandle(nodevec, RmiConnect, dnh);
}
/**
 *  Returns a PRNodehandle for this node
 */

public PRNodehandle getPRNodehandle() {
        return prnh;
}
/**
 *  Returns the port number this node is registered at
 */

public int getRMIPort() {
        return rmiport;
}
/**
 *  Returns a reference to this nodes MessageManager
 */
public MessageManager getMsgManager() {
        return mm;
}
/**
 *  This is an internal class that implements the Pastry class Address
 */

private static class HelloAddress implements Address {
        private int myCode = 0x1984abcd;
        public int hashCode() {
                return myCode;
        }

        public boolean equals(Object obj) {
                return (obj instanceof HelloAddress);
        }

        public String toString() {
                return "[HelloAddress]";
        }
}

// The remaining methods override abstract methods in the PastryAppl API.

/**
 *  Returns the address of this application.
 */

public Address getAddress() {
        addr = new HelloAddress();
        return addr;
```

```java
	}

	/**
	 * Reuturns this nodes credentials.
	 */

	public Credentials getCredentials() {
		return cred;
	}

	/**
	 * Invoked on intermediate nodes in routing path.
	 */

	public boolean enrouteMessage(
		Message msg,
		Id key,
		NodeId nextHop,
		SendOptions opt) {
		if (Log.ifp(5))
			System.out.println("Enroute " + msg + " at " + getNodeId());
		return true;
	}

	/**
	 * Invoked upon change to routing table.
	 */
	public void routeSetChange(NodeHandle nh, boolean wasAdded) {
		if (Log.ifp(5)) {
			System.out.print(
				"In "
					+ getNodeId()
					+ "'s route set, "
					+ "node "
					+ nh.getNodeId()
					+ " was ");
			if (wasAdded)
				System.out.println("added");
			else
				System.out.println("removed");
		}
	}

	/**
	 *   Invoked when this node forwards a message
	 */

	public void forward(RouteMessage msg) {

		System.out.println("\nmsg is passing through" + getNodeId());
		return;
	}

	/**
	 *   Called by the Pastry layer when a message destined for this node arrives,
	 *   the message is passed to the message manager. The message manager then
	 *   forwards then reads the message to find out what type of message it is
	 */
	public void deliver(Id key, Message msg) {
		mm.setprNode(this);
		mm.readMessage(msg);
		return;
	}
	/**
	 * Invoked by {RMI,Direct}PastryNode when the node has something in its
	 * leaf set, and has become ready to receive application messages.
	 */
	public void notifyReady() {
		if (true /*Log.ifp(6)*/
			)
			System.out.println(
			"Node " + getNodeId() + " ready, waking up any clients");
	}

}
```

A.I.iii PRNodehandle

```
package DIT.PeerReach;

import java.io.*;
import DIT.PeerReach.vector.*;
import rice.pastry.dist.*;

/**
 *   This class is a handle to a node, it is a PeerReach implementation of the Pastry *
 idea. This class produces objects that contain the details of a particular node
 *   such as a nodes nodevector etc. The class implements serializable so it may be
 *   passed over a network connection
 *   @author Paul Stacey
 */

public class PRNodehandle implements Serializable {

        private NodeVector nodevec;
        private String rmiConnect;
        private DistNodeHandle distnh;

       /**
         *  Constructor.
         *
         *  Takes in a nodes main attributes and sets them within this class
         */
        public PRNodehandle(
                NodeVector nv,
                String rmiConnection,
                DistNodeHandle dnh) {
                nodevec = nv;
                rmiConnect = rmiConnection;
                distnh = dnh;
        }
        /**
          *   Returns a NodeVector
          */

        public NodeVector getNodeVector() {

                return nodevec;

        }

        /**
          *   Returns the rmi connection for the associated node
          */

        public String getRmiConnection() {
                return rmiConnect;

        }

       /**
         *   Returns the distributed nodehandle for the associated node
         *
         */
        public DistNodeHandle geDistNodeHandle() {

                return distnh;
        }
}
```

A.II. Package DIT.PeerReach.community

The community package contains classes that are used to manage and maintain community links between community nodes. Community nodes are nodes within the network that are deemed similar using a similarity metric

A.II.i CommunityManager

```
package DIT.PeerReach.community;

import DIT.PeerReach.*;
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

/**
 *   This class acts as the manager of the classes contained within the community
 *   package, it handles interactions with other managers of the application.
 *   @author Paul Stacey
 */

public class CommunityManager extends UnicastRemoteObject implements
                             RemoteCommunityManager {

        private CommunityTable commtable;
        private PeerReachNode pn;

      /**
        *   Constructor.
        *
        *   Takes a reference to the node that instantiated this class a an argument
        *   also sets up the community table
        */

        public CommunityManager(PeerReachNode prn) throws RemoteException {

                pn = prn;
                commtable = new CommunityTable();

        }

      /**
        *  This method is called by other nodes using an rmi connection
        *  it is used to inform a community manager that they should update there
        *  community tables to include a particular node
        */
        public void updateCommunityTable(Vector nodeSet) throws RemoteException {

                int nodSetSize = nodeSet.size();
                for (int i = 0; i <= nodSetSize - 1; i++) {
                        PRNodehandle prnhandle = (PRNodehandle) nodeSet.get(i);
                        commtable.addNeighbour(prnhandle);
                }
                informCommunity(nodeSet);
        }

        /**
         *  Sets up an rmi link between the community nodes
         */
        public void createCommunityLink(PRNodehandle prnhan)
                throws RemoteException {
                commtable.addNeighbour(prnhan);
        }

      /**
        *  Called to tell a recently connected neighbour to update its community link
        *  this method is exported as a remote rmi call
        */

        public void informCommunity(Vector nset) {
                int nodeSetSize = nset.size();
                for (int i = 0; i <= nodeSetSize - 1; i++) {
                        PRNodehandle prnh = (PRNodehandle) nset.get(i);
                        String port = prnh.getRmiConnection();
                        try {
                                RemoteCommunityManager rcm =
                                        (RemoteCommunityManager) Naming.lookup(port);
                                rcm.createCommunityLink(prnh);
                        } catch (Exception e) {
                                        System.out.println("Community update client
                                        exception2: " + e);
                        }
```

```
                }

        }
}
```

A.II.ii CommunityServer

```
package DIT.PeerReach.community;

import java.rmi.*;
import DIT.PeerReach.*;

/**
 *   This class binds the community manager to the rmiregistry
 *
 *   @author Paul Stacey
 */

public class CommunityServer {

        public CommunityServer(PeerReachNode prn, int port) {

                try {

                        Naming.rebind(
                                "CommunityManager//127.0.0.1/" + port,
                                new CommunityManager(prn));
                        System.out.println(
                                "CommunityManager Server is ready. it is bound to
                                CommunityManager//127.0.0.1/"
                                        + port);

                } catch (Exception e) {
                        System.out.println("CommunityManager failed" + e);
                }

        }
}
```

A.II.iv RemoteCommunityManager

```
package DIT.PeerReach.community;

import DIT.PeerReach.*;
import java.rmi.*;
import java.util.*;

/**
 *   Interface
 *   An Interface to any class that is aimplemented as the community manager
 *   abstract methods
 *   @author Paul Stacey
 */

public interface RemoteCommunityManager extends Remote {

        public void updateCommunityTable(Vector nodeSet) throws RemoteException;

        public void createCommunityLink(PRNodehandle prnhan)
                throws RemoteException;

}
```

A.III. Package DIT.PeerReach.message

The message package contains classes that are used to read and create messages. Within the prototype the Message Manager assumes a registry message. All messages that arrive in the deliver method of PeerReachNode are passed to the message manager.

A.III.i MessageManager

```
package DIT.PeerReach.message;

import rice.pastry.messaging.*;
import DIT.PeerReach.*;
import DIT.PeerReach.register.*;
import DIT.PeerReach.vector.*;

import java.io.*;

/**
 *   This class handles interactions between the messaging package and the rest of the
 *   application. The Message manager deals with any messages passed from the pastry
 *   Layer
 *   @author Paul Stacey
 */

public class MessageManager implements Serializable {

        private PeerReachNode pn;
        private RegistryManager rm;
/**
 *   Constructor
 *   Takes in a reference to the registry manager
 */

        public MessageManager(RegistryManager regm) {
                rm = regm;
        }
       /**
        *  Reads messages passed up from the pastry layer
        */
        public void readMessage(Message msg) {

                System.out.println("\nMessage manager is reading message!");
                PRMessage message = (PRMessage) msg;
                MessageReader read = new MessageReader(msg);
                String type = read.getMsgType();
                // at the moment assumes a registry message
                if (true) {
                        PeerReachNode prn = message.getPeerReachNode();
                        NodeVector nv = message.getNodeVector();
                        rm.registerNode(prn, nv);
                }
        }

       /**
        *  Sets the the instance of the node
        */
        public void setprNode(PeerReachNode prn) {

                pn = prn;
        }
}
```

A.III.iii PRMessage

```
package DIT.PeerReach.message;

import java.net.*;
import rice.pastry.messaging.*;
import rice.pastry.NodeId;
import DIT.PeerReach.vector.*;
import DIT.PeerReach.*;

/**
 *   This class represents a message that can be passed to the pastry layer. PRMessage
 *   is an extension of a Pastry Message
 *
 *   @author Paul Stacey
 */

public class PRMessage extends Message {
```

```
        public NodeId source;
        public NodeId target;
        public NodeVector nvec;
        private int msgid;
        private PeerReachNode prn;

        /**
         *   Constructor
         *
         *   Initialises all the attribute of the message
         */
        public PRMessage(
                Address addr,
                NodeId src,
                NodeId tgt,
                int mid,
                NodeVector nv,
                PeerReachNode pn) {
                super(addr);
                source = src;
                target = tgt;
                msgid = mid;
                nvec = nv;
                prn = pn;
        }

        /**
         *   Returns the NodeVector from the message
         *
         */

        public NodeVector getNodeVector() {

                return nvec;
        }

        /**
         *  Returns a reference to the node that sent the message
         */

        public PeerReachNode getPeerReachNode() {

                return prn;
        }
        /**
         *  returns the IP address of the node, currently just returns the local IP.
         */
        public InetAddress getIPaddr() {

                InetAddress Ipaddr = null;

                try {
                        Ipaddr = InetAddress.getLocalHost();
                } catch (UnknownHostException e) {
                        System.out.println(e);
                }
                return Ipaddr;

        }
}
```

A.VI. Package DIT.PeerReach.register

A.VI.iii IndexTable

```
package DIT.PeerReach.register;

import java.util.*;

import DIT.PeerReach.vector.*;
import DIT.PeerReach.*;
```

```
/**
 *  This class contains methods used to register nodesin an index table, currently a
 *  hashtable is used, future implementations of this class will create and maintain a *
 clustering tree.
 *
 *  @author paul stacey
 */
public class IndexTable{

Vector indxtable = new Vector(); // note this will eventually become a cluster tree!
Hashtable intble = new Hashtable();
PeerReachNode peernode;

        /**
         * Add node to the index table
         */
        public void addNode(PeerReachNode pnnn, NodeVector nv){
                peernode = pnnn;
                PRNodehandle prnh = pnnn.getPRNodehandle();
                indxtable.addElement(prnh);
                intble.put(pnnn,nv);
        }

        /**
         * Removes a node from the index table
         */

        public void removeNode(){
        }

       /**
        * search the index table for a similar node to the given nodevector
        */

       public Vector searchTable(NodeVector search){
                //int i = 0;
                //PeerReachNode prn = (PeerReachNode) indxtable.get(i);
                //NodeVector nodev = (NodeVector) intble.get(prn);
                //return prn;
                return indxtable;
        }
}
```

A.VI.iv RegistryManager

```
package DIT.PeerReach.register;

import rice.pastry.NodeId;
import rice.pastry.join.*;
import rice.pastry.NodeHandle;
import DIT.PeerReach.*;
import DIT.PeerReach.message.*;
import DIT.PeerReach.community.*;
import rice.pastry.messaging.*;
import rice.pastry.routing.*;
import rice.pastry.security.*;
import DIT.PeerReach.vector.*;
import java.rmi.*;
import java.util.*;

/**
 *   This class handles interactions between classes in the registry package and the
 *   rest of the application. Registy Manager contains methods to register nodes,
 *   search the index tables and forwards search queries.
 *
 *  @author paul stacey
 */

public class RegistryManager {

        private int msgid = 0;
        private static Address addr;
        private static Credentials cred = new PermissiveCredentials();
        private ConcRegistryIdFactory crf;
        private PeerReachNode prn;
```

```
        private IndexTable indx = new IndexTable();
        private NodeHandle hint = null;
        private String[] registryWrds;
        private NodeVector nodeVector;
        private int Port;

        public RegistryManager(int port) {
                //Port = port;
                crf = new ConcRegistryIdFactory();
        }

        /**
         *   Set the nodevector
         */
        public void setNodeVector(NodeVector nvec) {
                nodeVector = nvec;
        }
        /**
         * Routes the registry request to the node with the nodeId nid
         */
        public void routeRegistryRegst(NodeId nid) {
                Message msg =
                        new PRMessage(addr, prn.getNodeId(), nid, ++msgid, nodeVector,
                        prn);
                prn.routeMsg(nid, msg, cred, new SendOptions());
        }

        /**
         * Registers the node pnn and its nodevector nv in the index tables
         */

        public void registerNode(
                PeerReachNode pnnn,
                NodeVector nv) { //mayb add in a node Handle here
                indx.addNode(pnnn, nv);
                //this will change to update the community or be a community table
                findSimilarNodes(nv, pnnn);
        }

        /**
         * Searches the index table for similar nodes to the given node and its
         * nodevector
         */

        public void findSimilarNodes(NodeVector nodevec, PeerReachNode pnr) {

                Vector prn = indx.searchTable(nodevec);
                returnSimNodes(prn, pnr);

        }
        /**
         * Returns a set of nodes that were found to be similar to a given node
         */

        public void returnSimNodes(Vector nodeSet, PeerReachNode prnnn) {

                int port = prnnn.getRMIPort();
                System.out.println(
                  "in the registry manager/retunrSimNodes this is the port:" + port);
                try {
                // RemoteCommunityManager rcm = (RemoteCommunityManager) Naming.lookup
("rmi://127.0.0.1/CommunityManager");
                        RemoteCommunityManager rcm =
                                (RemoteCommunityManager) Naming.lookup(
                                        "CommunityManager//127.0.0.1/" + port);
                        System.out.println("The rmi connection appears to be
                        successful!");
                        rcm.updateCommunityTable(nodeSet);
                } catch (Exception e) {
                        System.out.println("Community update client exception1: " + e);
                }
        }
```

```java
        /**
         *  Registers a given node in the index table, this method is called by
         *  PeerReachAppl
         */
        public void register() {

                JoinAddress addr = new JoinAddress();
                String reg;
                NodeId regid = new NodeId();
                Credentials cred = new PermissiveCredentials();
                int size = registryWrds.length;
                for (int i = 0; i <= size - 1; i++) {
                        reg = registryWrds[i];
                        crf.setRg(reg);
                        regid = crf.generateNodeId();
                        routeRegistryRegst(regid);
                }
        }

        /**
         * Sets the interests or keywords of the node that instantiated this registry *
         manager
         */

        public void setInterests(String[] interests) {
                registryWrds = interests;
        }
        /**
         *  sets the reference of the node that instantiated this registry manager.
         */

        public void setNodeInstance(PeerReachNode pn) {
                prn = pn;
        }

}
```

A.VII. Package DIT.PeerReach.vector

A.VII.i FileVector

```java
package DIT.PeerReach.vector;

import Jama.*;
import java.io.*;

/**
 *  This class implements the interface PRVector, it is the vector representation of
 *  a file within the system. The actual vector is a Jama one dimension natrix.
 *  @author Paul Stacey
 */
public class FileVector implements PRVector, Serializable {

        private Matrix vec;
        private int size;
        /**
         * Constructor
         */

        public FileVector() {
        }
        /**
         *  Constructor
         *  initialize the size of the matrix, at tprsent this is the size of the
         *  training set
         */
        public FileVector(int s) {

                size = s;
                vec = new Matrix(size, 1);

        }
```

```java
	/**
	 * Returns the size of the vector/matrix
	 */

	public int getSize() {

		return size;
	}
	/**
	 * Adds a zero to the vector at the specified point
	 */
	public void addZero(int index) {

		double zero = 0;
		vec.set(index, 0, zero);
	}
	/**
	 * Adds a value at the specified point within the vector
	 */

	public void addValue(double value, int index) {

		vec.set(index, 0, value);
	}
	/**
	 * Outputs the vector to the screen
	 */

	public void printVec() {

		vec.print(1, size);

	}
	/**
	 * returns the Jama Matrix wrapped by this class
	 */
	public Matrix getMatrix() {
		return vec;
	}
	/**
	 *  Compares another filevector to this FileVector and returns a similarity
	 *  index between 0 and 1
	 */
	public double compare(FileVector v) {
		Matrix compare = v.getMatrix();
		double cmpn2 = compare.norm2();
				System.out.println("this is the l2 norm of the vector
		compare" + cmpn2);
		double vecn2 = vec.norm2();
		System.out.println("this is th l2 norm of the other vector" + vecn2);
		double n2prod = cmpn2 * vecn2;
		System.out.println("this is the product of the two l2 norms" + n2prod);
		double dprod = dotProduct(vec, compare);
		System.out.println(
			"this is the value returned from the dot product method" + dprod);
		double simindex = dprod / n2prod;
		System.out.println(simindex);
		return simindex;
	}
	/**
	 * Performs the dot product of two vectors
	 */
	public double dotProduct(Matrix v, Matrix c) {
		double product;
		double dotproduct = 0;
		for (int i = 0; i <= 54; i++) {
			double velem = v.get(i, 0);
			double celem = c.get(i, 0);
			product = velem * celem;
			dotproduct = dotproduct + product;
		}
		return dotproduct;
	}
}
```

A.VII.ii NodeVector

```
package DIT.PeerReach.vector;

import Jama.*;
import java.io.*;

/**
 *  This class implements the interface PRVector, it is the vector representation of
 *  a Node within the system. The actual vector is a Jama one dimension natrix.
 *  @author Paul Stacey
 */

public class NodeVector implements PRVector, Serializable {

        private Matrix centroid;
        private int size;
        /**
         * Constructor
         */
        public NodeVector() {
        }
        /**
         *  Constructor
         *  initialize the size of the matrix, at tprsent this is the size of the
         *  training set
         */
        public NodeVector(Matrix cent) {

                centroid = cent;

        }
        /**
         * Returns the size of the vector/matrix
         */
        public int getSize() {

                return size;
        }
        /**
         * Adds a zero to the vector at the specified point
         */
        public void addZero(int index) {

                double zero = 0;
                centroid.set(index, 0, zero);
        }
        /**
         * Adds a value to the vector at the specified point
         */
        public void addValue(double value, int index) {

                centroid.set(index, 0, value);
        }

        /**
         * Prints the NodeVector to screen
         */
        public void printVec() {

                centroid.print(1, size);

        }
        /**
          *Returns the Jama Matrix wrapped by this class
          */
        public Matrix getMatrix() {

                return centroid;
        }
        /**
          *  Compares a FileVector to this NodeVector
          */

        public double compare(FileVector v) {
                Matrix compare = v.getMatrix();
```

```java
		double cmpn2 = compare.norm2();
		System.out.println("this is the 12 norm of the vector compare" + cmpn2);
		double vecn2 = centroid.norm2();
		System.out.println("this is th 12 norm of the other vector" + vecn2);
		double n2prod = cmpn2 * vecn2;
		System.out.println("this is the product of the two 12 norms" + n2prod);
		double dprod = dotProduct(centroid, compare);
		System.out.println(
			"this is the value returned from the dot product method" + dprod);
		double simindex = dprod / n2prod;
		System.out.println(simindex);
		return simindex;
	}
	/**
	 *  Compares a NodeVector to this NodeVector
	 */
	public double compare(NodeVector v) {
		Matrix compare = v.getMatrix();
		double cmpn2 = compare.norm2();
		System.out.println("this is the 12 norm of the vector compare" + cmpn2);
		double vecn2 = centroid.norm2();
		System.out.println("this is th 12 norm of the other vector" + vecn2);
		double n2prod = cmpn2 * vecn2;
		System.out.println("this is the product of the two 12 norms" + n2prod);
		double dprod = dotProduct(centroid, compare);
		System.out.println(
			"this is the value returned from the dot product method" + dprod);
		double simindex = dprod / n2prod;
		System.out.println(simindex);
		return simindex;
	}

	/**
	 * Performs the dot product of two vectors
	 */
	public double dotProduct(Matrix v, Matrix c) {
		double product;
		double dotproduct = 0;
		for (int i = 0; i <= 54; i++) {
			double velem = v.get(i, 0);
			double celem = c.get(i, 0);
			product = velem * celem;
			dotproduct = dotproduct + product;
		}
		return dotproduct;
	}
}



package DIT.PeerReach.vector;

import Jama.*;
/**
 * This is an Interface to a class that is a vector representation of an object
 * @author Paul Stacey
 */

public interface PRVector {

	public void addZero(int index);

	public void addValue(double value, int index);

	public void printVec();

	public Matrix getMatrix();

	public double compare(FileVector v);
	public double dotProduct(Matrix v, Matrix c);

}
```

A.VII.iii TrainingSet

```
package DIT.PeerReach.vector;

import java.util.*;


/**
 *  This class implements the TrainingSet used in vector space modeling of files and
 *  nodes
 *
 *  @author Paul Stacey
 */

public class TrainingSet {

        private WrdWeightList wl = new WrdWeightList();
        private String stopList = "";
        private Vector vector = new Vector();
        private ArrayList stpfileList = new ArrayList();
        private ArrayList stmdfileList = new ArrayList();
        private ArrayList removedwords = new ArrayList();
        private ArrayList stpList = new ArrayList();
        private int stpIndx = 0;
        private int fleIndx;
        private int flesize;
        private int stpsize;
        private int initflesize;

        /**
          *  This class implements the TrainingSet used to calculate vector space
          *  representations of files and nodes
          *
          *  @author Paul Stacey
          */

        public TrainingSet(ArrayList file, String stList, WrdWeightList wwl) {

                wl = wwl;
                stopList = stList;
                stpfileList = stopList(file, stopList);
                stmdfileList = stemmer(stpfileList);
                buildTrainingSet(file);

        }
        /**
          *  Returns an stemmed list of words of the arraulst al
          */

        public ArrayList stemmer(ArrayList al) {
                ArrayList stL = new ArrayList();
                Stemmer st = new Stemmer();
                st.stemList(al);
                stL = st.getStemmedList();
                return stL;
        }
        /**
          *  Returns the arraylist with stopwords removed
          */

        public ArrayList stopList(ArrayList file, String stoplist) {
                boolean test = true;
                flesize = file.size(); // get size of the array
                initflesize = flesize;
                //read in stop list file and convert to a arraylist
                InputReader ip = new InputReader();
                stpList = ip.getFile(stoplist); // read in stop list to arralist
                stpsize = stpList.size(); //get size of array
                //first element of stoparray list
                for (int j = 0; j <= stpsize - 1; j++) {
                        fleIndx = 0;
                        test = true;
                        Object stpElement = stpList.get(stpIndx);
                        String stpElem = stpElement.toString();

                        while (test) {
```

```
                        Object fleElement = file.get(fleIndx);
                        String fleElem = fleElement.toString();
                        int ans = fleElem.compareTo(stpElem);
                        // compares lexically if equal
                        if (ans == 0) {
                                Object removedStrng = file.remove(fleIndx);
                                String remove = removedStrng.toString();
                                flesize = file.size();
                                removedwords.add(remove);
                        }
                        fleIndx++;
                        if (flesize - fleIndx <= 0) {
                                test = false;

                        }
                }
                stpIndx++;
        }
        file.trimToSize();
        return file;
}
/**
 *  Builds the trainingset
 */
public void buildTrainingSet(ArrayList filelist) {

        Object word;

        int arraysize = filelist.size();
        HashMap tF = new HashMap(arraysize);

        for (int i = 0; i <= arraysize - 1; i++) {

                word = filelist.get(i);
                boolean contain = vector.contains(word);

                if (contain == true) {
                } else {
                        vector.add(word);
                        updateTrainingSet(word);
                }
        }
}
/**
 *  Updates the TrainingSet
 */
public void updateTrainingSet(Object word) {

        wl.updateWrdWeightList(word);
        System.out.println("..............we are updating the word list!");
}

}
```

A.VII.iv VectorFactory

```
package DIT.PeerReach.vector;

import java.util.*;
/**
 *  Interface to a class that can create vector space representations
 *
 *  @author Paul Stacey
 */
public interface VectorFactory {
        public HashMap getTfidf(ArrayList tf);
        public HashMap buildTf(ArrayList file);
}
```

A.VII.v VectorManager

```
package DIT.PeerReach.vector;
```

```
import java.util.*;

import DIT.PeerReach.persistence.*;
import Jama.*;

/**
  * This class is the manager of the vector package
  *
  * @author Paul Stacey
  */

public class VectorManager {

        private ArrayList flevectorlist;
        private int initcapacity = 300;
        private ArrayList list = new ArrayList();
        private HashMap wwwl = new HashMap();
        private PersistentStore s = new PersistentStore();
        private TrainingSet tset;
        private VSMFileVectorFactory fileVector;
        private String stoplist =
          "C:/eclipse2/eclipse/workspace/masters/DIT/PeerReach/vector/van rijsbergen stop
        list.txt";
        private WrdWeightList wwl = new WrdWeightList();
        private ArrayList fleVector = new ArrayList();
        private int count = 0;
        private String name = "";
        private String path =
                "C:/eclipse2/eclipse/workspace/masters/DIT/PeerReach/vector/";
        private int countdown = 10;
        private String countdwn = "building training set";
        private NodeVector nvec = null;
        public ArrayList getFile(String Path) {
                InputReader reader = new InputReader();
                list = reader.getFile(Path);
                return list;
        }



        /**
          * Build Training Set
          *
          */
        public void buildTrainingSet() {
                for (int i = 0; i <= 10; i++) {
                        Integer cnt = new Integer(count);
                        name = cnt.toString();
                        String file = path + name + ".txt";
                        //read test file into an array list of words
                        fleVector = getFile(file);
                        //construct the file vector
                        //Need to change this design, too slow
                        tset = new TrainingSet(fleVector, stoplist, wwl);
                        count++;
                        System.out.println(countdwn + "\n" + countdown);
                        countdown--;
                        countdwn = "";
                }
                s.writeObject(wwl, "trainingset");
                System.out.println("the traininig set has been saved");
        }
        /**
          * Read the hard saved training set and retrun it as type WrdWeightList
          *
          */

        public WrdWeightList readTrainingSet() {
                WrdWeightList wrdwl = (WrdWeightList) s.readObject("trainingset");
                return wrdwl;
        }
         /**
          * Builds NodeVector from the FileVectors of the saved files
          *
           */
        public void buildNodeVector() {
```

```
		flevectorlist = new ArrayList();
		Matrix centroid = null;
		count = 0;
		WrdWeightList wrdwl = readTrainingSet();
		for (int i = 0; i <= 10; i++) {
			Integer cntname = new Integer(count);
			String fname = cntname.toString();
			String filename = path + fname + ".txt";
			fleVector = getFile(filename);
			HashMap wrddocfrqset = wrdwl.getHashMap();
			ArrayList trainwrdst = wrdwl.toArrayList();
			int wwlsize = trainwrdst.size();
			fileVector = new VSMFileVectorFactory(fleVector, stoplist,
				wrdwl);
			System.out.println("We are now building the file vectors!");
			FileVector fVector = fileVector.getFileVector();
			s.writeObject(fVector, fname);
			flevectorlist.add(fVector);
			System.out.println("filevector added" + flevectorlist);
			count++;
		}

		int size = flevectorlist.size();
		for (int i = 0; i <= size - 1; i++) {
			Object fv = flevectorlist.get(i);
			FileVector fvv = (FileVector) fv;
			int vsize = fvv.getSize();
			System.out.println("size of matrix is " + vsize);
			System.out.println("File Vector" + fvv);
			Matrix vector = fvv.getMatrix();
			if (i == 0) {
				centroid = vector;
			}
			if (i != 0) {
			centroid = centroid.plus(vector);
			}
		}
		nvec = new NodeVector(centroid);
		s.writeObject(nvec, "NodeVector");
	}
	/**
	 *  Return NodeVector, reads it from the persistent storage
	 *
	 */
	public NodeVector getNodeVector() {

		NodeVector savedNvec = (NodeVector) s.readObject("NodeVector");
		return savedNvec;
	}
}
```

A.VII.vi VSMFileVectorFactory

```
package DIT.PeerReach.vector;

import java.util.*;

import DIT.PeerReach.persistence.*;
/**
 *  This class provides methods of building vector representations of Objects
 *  @author Paul Stacey
 */

public class VSMFileVectorFactory implements VectorFactory {

	private int stpIndx = 0;
	private int fleIndx;
	private int flesize;
	private int stpsize;
	private int initflesize;
	private ArrayList tf = new ArrayList();
	private ArrayList fileList = new ArrayList();
	private ArrayList stmdfileList = new ArrayList();
	private ArrayList stpfileList = new ArrayList();
	private ArrayList stpList = new ArrayList();
```

```
private ArrayList removedwords = new ArrayList();
private HashMap termFrequency = new HashMap();
private HashMap training = new HashMap();
private String stopList = "";
private FileVector fleVector;
private FileVector fileVector = new FileVector();
private PersistentStore store = new PersistentStore();

/*
* Constructor
* Initialises the class variables
*
*/

public VSMFileVectorFactory(
        ArrayList file,
        String stList,
        WrdWeightList wwl) {
        stopList = stList;
        fileList = file;
        stpfileList = stopList(fileList, stopList);
        stmdfileList = stemmer(stpfileList);
        System.out.println("Building the term-frequncy table and updating the
        training set this may take a while please wait!\n");
        termFrequency = getTfidf(stmdfileList);
        fileVector = buildVector(wwl, termFrequency);
}

public FileVector getFileVector() {
        return fileVector;
}
/**
* This method instantiates an object that implements a
* stemming algorithm to stem the words with the file
*
*/

public ArrayList stemmer(ArrayList al) {

        ArrayList stL = new ArrayList();
        Stemmer st = new Stemmer();
        st.stemList(al);
        stL = st.getStemmedList();
        return stL;
}
/**
  * This method removes all the stoplist words from the file
  * a the stop list can be easily changed and passed to this method
  *
  */
public ArrayList stopList(ArrayList file, String stoplist) {

        boolean test = true;
        flesize = file.size(); // get size of the array
        initflesize = flesize;
        //read in stop list file and convert to a arraylist
        InputReader ip = new InputReader();
        stpList = ip.getFile(stoplist); // read in stop list to arralist
        stpsize = stpList.size(); //get size of array
        //first element of stoparray list
        for (int j = 0; j <= stpsize - 1; j++) {
                fleIndx = 0;
                test = true;
                Object stpElement = stpList.get(stpIndx);
                String stpElem = stpElement.toString();
                while (test) {
                        Object fleElement = file.get(fleIndx);
                        String fleElem = fleElement.toString();
                        int ans = fleElem.compareTo(stpElem);
                        // compares lexically if equal this may not be what we
                        //want, check!
                        if (ans == 0) {
                                Object removedStrng = file.remove(fleIndx);
                                String remove = removedStrng.toString();
                                flesize = file.size();
                                removedwords.add(remove);
                        }
```

```
                        fleIndx++;
                        if (flesize - fleIndx <= 0) {
                                test = false;

                        }
                }
                stpIndx++;
        }
        file.trimToSize();
        return file;
}

/*
 * Method that returns the tf-idf representation of the file
 */
public HashMap getTfidf(ArrayList tf) {

        HashMap test = buildTf(tf);
        return test;
}

// buildTf is quite slow, there is definitely room to speed up this algorithm,
//review latter
public HashMap buildTf(ArrayList file) {
        tf = file;
        int numOccour = 0;
        Object searchOb;
        Object search = null;
        int arraysize = tf.size();
        HashMap tF = new HashMap(arraysize);
        for (int i = 0; i <= arraysize - 1; i++) {
                numOccour = 0;
                searchOb = tf.get(i);
                for (int j = 0; j <= arraysize - 1; j++) {
                        search = tf.get(j);
                        String search1 = searchOb.toString();
                        String search2 = search.toString();
                        int compare = search1.compareTo(search2);
                        if (compare == 0) {
                                numOccour = numOccour + 1;

                        }
                }

                Integer numOcc = new Integer(numOccour);
                boolean contain = tF.containsKey(searchOb);
                if (contain) {
                } else {

                }
                Object x = tF.put(searchOb, numOcc);
                //recheck positioning of this update, may have performance
                //implications
        }
        int size = tF.size();
        //System.out.println("the size of the idf is ="+ size);
        return tF;
}
/**
  *  Builds a vector given the training set and a term frequency list
  *
  */

public FileVector buildVector(WrdWeightList training, HashMap termFreq) {

        int N = 10;
        Vector dict = training.getDictionary();
        int size = dict.size();
        HashMap wrddocfreqset = training.getHashMap();
        ArrayList trainwrdset = training.toArrayList();
        fleVector = new FileVector(size);
        for (int i = 0; i <= size - 1; i++) {
                Object term = trainwrdset.get(i);
                Integer docfi = (Integer) wrddocfreqset.get(term);
                double dfi = docfi.doubleValue();
                boolean contains = termFreq.containsKey(term);
                if (contains) {
```

```
                                        Integer trmfreq = (Integer) termFreq.get(term);
                                        double tfreq = trmfreq.doubleValue();
                                        double vecterm = tfreq * Math.log(N / dfi);
                                        fleVector.addValue(vecterm, i);

                                } else {
                                        fleVector.addZero(i);
                                }
                        }

                        return fleVector;
                }

}
```

A.VII.vii VSMStemmer

```
package DIT.PeerReach.vector;

import java.util.*;

/**
  * An Interface to a class that implements a stemming algorithm
  * @author Paul Stacey
  */

public interface VSMStemmer {

        public void stemList(ArrayList al);

        public ArrayList getStemmedList();
}
```

Appendix B

This Appendix lists the m-files written in the initial experiments on Vector Space modeling techniques detailed in section 6.1.1. Three m-files were programmed to realise the experiment.

B.I. M-File 1

```
% Create an array to store results
ext = '.txt';
resultsarray = {};

%Read in 1000 most common english words
common = textread('1000mostcommonEnglwords.txt','%s','delimiter','\n ','whitespace','
');
common = upper(common);
commonlen = length(common);
repeat = 1;
for repeat = 1:3;
i= 1;
counter = 1;
doccount=textread('doccount.txt','%s','delimiter','\n','whitespace',' ');
docpointer = 1;
S = length(doccount);
for docpointer = 1:S
    if docpointer == 1
     docnum = 'one'
    elseif docpointer == 2
       docnum = 'two'
    elseif docpointer == 3
       docnum = 'thr'
    elseif docpointer == 4
       docnum = 'for'
    elseif docpointer == 5
       docnum = 'fiv'
    elseif docpointer == 6
       docnum = 'six'
    elseif docpointer == 7
       docnum = 'sev'
    elseif docpointer == 8
       docnum = 'eig'
    elseif docpointer == 9
       docnum = 'nin'
    elseif docpointer == 10
       docnum = 'ten'
    elseif docpointer == 11
       docnum = 'elv'
    elseif docpointer == 12
       docnum = 'twl'
    elseif docpointer == 13
       docnum = 'tur'
    elseif docpointer == 14
       docnum = 'frt'
    else
       docnum = 'fit'
    end
     doc = doccount(docpointer);
     doc = char(doc);
     document = [doc,ext];

% Create cell array by reading in the txt file
file = textread(document,'%s','delimiter','\n  ( ) . , " , ','whitespace',' ');
% Converts all text to upper case because Matlab is case sensitive
     file = upper(file);
% Extract all common words
     for counter = 1:commonlen
         word = common(counter);
```

```
commonmatch = strmatch(word,file,'exact');
        file(commonmatch)= '';
        counter = counter +1;
    end

% get the length of the cell array
    l = 1;
    L = int2str(l);
    y = length(file);

% Create variable cellnum and set it equal to 1
    cellnum = 1;
    %i= 1;
    j= 1;

% for loop to read all elements of cell array and find out how may
% times they occur
    for cellnum = 1:y;
% Read element corresponding to cellnum of %cell array
        x = file(cellnum);
        X = char(x);
% Compare x to elements in array to see if it exists, if not add it
% to array
        k = strmatch(X,resultsarray,'exact');
        RAlen = length(resultsarray);
            p = length(k);
            if p == 0;
                resultsarray{i,j} = X;
                resultsarray{i,j+2} = L;
% Do a str match on file to see how many times x occurs in document
                match = strmatch(x,file,'exact');
                matchsize = length(match);
                MATCHSIZE = int2str(matchsize);
                % store matchsize in array ;
                resultsarray{i,j+1} = MATCHSIZE;
                resultsarray{i, j+3}= docnum;
            i = i+1;
         elseif k > RAlen
            resultsarray{i,j} = X;
                resultsarray{i,j+2} = L;
% Do a str match on file to see how many times x occurs in document
                match = strmatch(x,file,'exact');
                matchsize = length(match);
                MATCHSIZE = int2str(matchsize);
% Store matchsize in array
                resultsarray{i,j+1} = MATCHSIZE;
                resultsarray{i, j+3}= docnum;
            i = i+1;
          else
%If it is in the array already and occurs in the new document we %want to increment N
and increment the number of times it has %occurred
                match1 = strmatch(x,resultsarray,'exact');
            match1 = match1(1);
                matchN = resultsarray(match1, j+3);
            MATCHN = char(matchN);
        if MATCHN == docnum
        cellnum = cellnum +1;
        else
            match2 = strmatch(x,file,'exact');
            matchsize = length(match2);
            m = resultsarray(match1, j+1);
        orgmatch = char(m);
        orgmatch = str2num(orgmatch);
        orgmatch = orgmatch + matchsize;
            MATCHSIZE = int2str(orgmatch);
        resultsarray{match1, j+1} = MATCHSIZE;
        h = resultsarray(match1, j+2);
        H = char(h);
        H = str2num(H);
        H = H+1;
        H = int2str(H);
        resultsarray{match1, j+2} = H;
        resultsarray{match1, j+3} = docnum;
      end
      end
      resultsarray;
    end
```

```
    docpointer = docpointer+1;
end
repeat = repeat + 1;
end
save resultsarray
vectorconstruct
```

B.II M-File 2

```
% Program to calculate the vector representation of a document given % an already
constructed array of words

% load in new text doc

file = textread('14.txt','%s','delimiter','\n','whitespace',' ');
len = length(resultsarray);
j =1;
vec = zeros(len,1);
for i = 1:len;
i
% read first line of results array
    x = resultsarray(i);
% does doc contain this word?
    match = strmatch(x,file,'exact');

    p = length(match);
    if p == 0;
   % no? -> add zero to vector
       vec(j,1) = 2 ;
   % yes? get frequency multiply by log(N/dfi)
    else
        freq = length(match);
        N = resultsarray(i,3);
        N = char(N);
        N = str2num(N);
        dfi = resultsarray(i,2);
        dfi = char(dfi);
        dfi = str2num(dfi);
        term = freq*log(N/dfi);
        vec(j,1) = term;
        j = j+1; % increment vector pointer
        i = i+1; % increment pointer in resultsarray
    end
end
 vecnew = vec;
```

B.III. M-File 3

% program to calculate the vector representation of a document given

% an already constructed array of words load in new text doc

```
ext = '.txt';
docpointer = 1;

for docpoint = 1:S

   doc = doccount(docpointer);
    doc = char(doc);
    document = [doc,ext];

file = textread(document,'%s','delimiter','\n','whitespace',' ');
len = length(resultsarray);
j =1;
vec = zeros(len,1);
for i = 1:len;

% read first line of results array
    x = resultsarray(i);
% does doc contain this word?
    match = strmatch(x,file,'exact');
```

```
    p = length(match);

    if p == 0;
% no? -> add zero to vector
       vec(j,1) = 2 ;
% yes? get frequency, multiply by log(N/dfi)
    else
        freq = length(match);
        N = resultsarray(i,3);
        N = char(N);
        N = str2num(N);
        dfi = resultsarray(i,2);
        dfi = char(dfi);
        dfi = str2num(dfi);
        term = freq*log(N/dfi);
        vec(j,1) = term;
        j = j+1; % increment vector pointer
        i = i+1; % increment pointer in resultsarray
    end
end

   if docpointer == 1
    vec1 = vec
   elseif docpointer == 2
      vec2 = vec
   elseif docpointer == 3
      vec3 = vec
   elseif docpointer == 4
      vec4 = vec
   elseif docpointer == 5
      vec5 = vec
   elseif docpointer == 6
      vec6 = vec
   elseif docpointer == 7
      vec7 = vec
   elseif docpointer == 8
      vec8 = vec
   elseif docpointer == 9
      vec9 = vec
   elseif docpointer == 10
      vec10 = vec
   elseif docpointer == 11
      vec11 = vec
   elseif docpointer == 12
      vec12 = vec
   elseif docpointer == 13
      vec13 = vec
   elseif docpointer == 14
      vec14 = vec
   else
      vec15 = vec
   end
%vec1 = vec
docpointer = docpointer + 1;
end
save vec1
save vec2
save vec3
save vec4
save vec5
save vec6
save vec7
save vec8
save vec9
save vec10
save vec11
save vec12
save vec13
save vec14
save vec15
```

Appendix C

This Appendix contains the java code written to realise the simulator. The Appendix only lists two classes, Simulation and PeerReachNode. These two classes are the main classes of interest within the simulator.

C.I. Package DIT.PeerReach.simulator

C.I.i Simulation

```
package DIT.PeerReach.simulator;

import java.util.ArrayList;
import java.util.Date;
import java.util.Hashtable;
import java.util.LinkedList;
import java.util.Random;
import drcl.comp.Component;
import drcl.comp.Port;
import drcl.comp.Util;
import drcl.sim.event.SESimulator;
import drcl.data.DoubleObj;
import drcl.comp.tool.Plotter;
import DIT.PeerReach.vector.NodeVector;

/**
 * This is the main simulation class, this class sets up the
 * simulation parameters and variables. The plotter is also
 * configured within this class to   * output simulation data
 * @author Paul Stacey
 */

public class Simulation extends Component {

        private Plotter plot;
        //      number of nodes a node can create links with
        private int communityConnectivity = 4;
        private double queryHops_plot = 0;
        private double queryHops_plot1 = 0;
        private int numberofNodes;
        private int cat = 8;
        public static final int DESCRIPTOR_ID = 0;
        private ConfigManager configManager;

        private double finishTime = 7641000.17365;
        ;
        public int totalNoOfFiles;
        public static final int SOURCE = 1;
        public static final int QUERY = 2;
        public static final int QUERY_HIT = 3;
        private Port forkPort;
        private Port commandPorts[];
        private Port plotterPort;
        private Port plotterPort1;
        private Port thisPlotterPort;
        private Port thisPlotterPort1;
        private PeerReachNode[] nodes;
        public static Hashtable fileHashtable;
        public static Hashtable confusionHashtable;
        private Connection[] connections;
        private Network network;
        private ArrayList bandwidthArray;
        private SESimulator runtime;
        private int descriptorIDCounter;
        private int satisfiedConnectivity = 0;
        private int totalConnected = 0;
```

```
        private boolean end1 = false;
        private boolean end2 = false;
        private int hits = 0;
        private double hitTime;
        long startTimeStamp;
        /**
         * Contructor
         *
         * Sets up simulation parameters
         */
        public Simulation(String configFilesPath, int numberNodes) {
                System.out.println("starting simulation");
                System.out.println("loading configuration files");
                numberofNodes = numberNodes;
                plot = new Plotter();
                // Setting up the Rendezvous points
                // This bypasses the use of Pastry for simulation
                // purposes
                for (int i = 1; i <= cat; i++) {
                        RendezVous RV = new RendezVous();
                        Integer catnum = new Integer(i);
                        String catNumber = catnum.toString();
                        RV.storeObject("C" + catNumber);
                }
        configManager = new ConfigManager(configFilesPath, numberofNodes);
                System.out.println("initialising simulator");
                //initialise datastructures
                nodes = new PeerReachNode[numberofNodes];
                runtime = new SESimulator();
                fileHashtable = new Hashtable();
                confusionHashtable = new Hashtable();
                commandPorts = new Port[numberofNodes];
                connections = new Connection[numberofNodes];
                forkPort = addForkPort("forkPort");
                createTopology(configManager);
                network = new Network(numberofNodes);
                setConnections(network);
                Util.setRuntime(this, runtime);
                Util.setRuntime(network, runtime);
                //set up the plotter
                Port plotPort1 = new Port();
                Port plotPort2 = new Port();
                plotterPort = plot.addPort(plotPort1, "0", "0");
                plotterPort1 = plot.addPort(plotPort2, "0", "1");
                thisPlotterPort = addPort("PlotterPort");
                thisPlotterPort.connect(plotterPort);
                thisPlotterPort1 = addPort("PlotterPort1");
                thisPlotterPort1.connect(plotterPort1);
                Util.setRuntime(plot, runtime);
                descriptorIDCounter = 0;
}



        /**
         * Method that builds the topology or the links between nodes
         */

        private void createTopology(ConfigManager configManager) {

                int connectivity;
                Random random = new Random(System.currentTimeMillis());
                TopologyConfig topconf = configManager.getTopologyConfiguration();
                ArrayList nodeVectorlist = configManager.getNodeVectorList();
                Hashtable nodeVectortoNodeIdList =
                        configManager.getNodeVectortoNodeIdList();
                NodeVector currentNodeVector;
                NodeVector currentNV;
                NodeVector currentprunedNodeVector;
                NodeVector currentprunedNodeVector1;
                ArrayList neighbourNodeVectorlist = null;
                int neighbourID = 0;
                int neighbor;
                //set up random links between nodes
                for (int i = 0; i < numberofNodes; i++) {
                        PeerReachNode node = new PeerReachNode((int) i, random, this);
                        nodes[i] = node;
```

```
                        Util.setRuntime(node, runtime);
                        commandPorts[i] = addPort("commandPort" + i);
                        commandPorts[i].connect(node.getPort("commandPort"));
                }
                //Set up node links from the rendezvous points
                for (int i = 0; i < numberofNodes; i++) {
                        PeerReachNode currentNode = nodes[i];
                        neighbourNodeVectorlist = new ArrayList();
                        LinkedList currentNeighborList =
                                configManager.getTopologyConfiguration().getPeerList(i);
                        LinkedList prunedNeigbourList = new LinkedList();
                        LinkedList randomNeigbourList = new LinkedList();
                        int[] temp_prunedNeigbourList;
                        //need to prune list, max 4
                        Integer[] neighborArray =
                                (Integer[]) currentNeighborList.toArray(new Integer[0]);
                        //setup random node link topology
                        Random generator = new Random();
                        for (int x = 0; x < 3; x++) {
                                int gnode = generator.nextInt(numberofNodes);
                                randomNeigbourList.addLast(new Integer(gnode));
                        }
                        currentNode.setRandomNeighborList(randomNeigbourList);
                        for (int j = 0; j < randomNeigbourList.size(); j++) {
                                neighbor = ((Integer)
(randomNeigbourList.get(j))).intValue();
                                PeerReachNode neighborNode = nodes[neighbor];
                                currentNode.connectMessageRouteTo(neighborNode);
                        }
                        //setting up the network topology
                        for (int j = 0; j < currentNeighborList.size(); j++) {
                                neighbourID = neighborArray[j].intValue();
                                if (prunedNeigbourList.contains(new Integer(neighbourID)))
{
                                        System.out.println("neighbour already registered");
                                } else if (neighbourID == i) {
                                        System.out.println("Cant connect to self");
                                } else {
                                        currentNodeVector =
                                                (NodeVector)
nodeVectorlist.get(neighbourID);
                                        neighbourNodeVectorlist.add(currentNodeVector);
                                        prunedNeigbourList.addLast((new
Integer(neighbourID)));
                                        System.out.println(
                                                "This is the pruned list" +
prunedNeigbourList);
                                }
                                if (prunedNeigbourList.size() >= communityConnectivity) {
                                        break;
                                }
                        }
                        currentNode.setNeighbourNodeVectorlist(neighbourNodeVectorlist);
                        currentNode.setNodeVectortoNodeIDTable(nodeVectortoNodeIdList);
                        currentNode.setNeighborList(prunedNeigbourList);
                        //Set up the cmessage connection routes
                        for (int j = 0; j < prunedNeigbourList.size(); j++) {
                                neighbor = ((Integer)
(prunedNeigbourList.get(j))).intValue();
                                PeerReachNode neighborNode = nodes[neighbor];
                                currentNode.connectMessageRouteTo(neighborNode);
                        }

                }
        }

        /**
         * Set the connections within the Network, this connects all the necessary ports
         */
        private void setConnections(Network network) {
                for (int i = 0; i < numberofNodes; i++) {
                        int inBandwidth =
                                configManager
                                        .getNodesConfiguration()
                                        .getNodeConfiguration(i)
                                        .getInBandwidth();
                        int outBandwidth =
```

```
                        configManager
                                .getNodesConfiguration()
                                .getNodeConfiguration(i)
                                .getOutBandwidth();
                Connection connection =
                        new Connection((int) i, inBandwidth, outBandwidth);
                connections[i] = connection;
                PeerReachNode currentNode = nodes[i];
                currentNode.getPort("connectionPort").connect(
                        connection.getPort("nodePort"));
                currentNode.getPort("connectionEventPort").connect(
                        connection.getPort("nodeEventPort"));
                connection.getPort("networkPort").connect(
                        network.getPort("nodeConnectionPort" + i));
                connection.getPort("networkEventPort").connect(
                        network.getPort("nodeConnectionEventPort" + i));
                Util.setRuntime(connection, runtime);
        }
}
/**
 * Method to return a reference to the configuration manager
 */

public ConfigManager getConfigManager() {
        return this.configManager;
}

/**
 * Method to return a reference to the PeerReachNode with id i
 */
public PeerReachNode getNode(int i) {
        return nodes[i];
}
/**
 * Method that returns the number of nodes currently active in the system
 */
public int getNumberofNodes() {
        return numberofNodes;

}
/**
 * Method that returns the next descriptor Id
 */
public int getNextDescriptorID() {
        return descriptorIDCounter++;
}

/**
 * Method, this is the contract that describes the behaviour of this component,
 * data arriving on the simulations ports is handled within this method
 *
 */
protected void process(Object data, Port inPort) {
        if (data instanceof String) {
                if (data.equals("START_SIMULATION")) {
                        startSimulation();
                        forkAt(forkPort, "END_SIMULATION", finishTime);
                        startTimeStamp = (new Date()).getTime();
                } else if (data.equals("END_SIMULATION")) {
                        EndSimulation();
                        System.exit(0);
                }
        }
}
/**
 * Method,starts the simulation and runs it until finishTime is reached.
 * The time unit of finishTime is in second.
 */
public void start() {
        GlobalVariables.start_();
        Util.inject("START_SIMULATION", forkPort);
}
/**
 * Method to update the plot when any relevant data is produced
 */
public void updatePlot(String type) {
        System.out.println("updating plotter");
```

```
        long currentSimTime = runtime.getWallTimeElapsed();
        //update the relevant plot
        if (type == "query1" & end1 == false) {
                System.out.println("Updating plot");
                DoubleObj testset = new DoubleObj();
                queryHops_plot = queryHops_plot + 1;
                testset.setValue(queryHops_plot);
                send(thisPlotterPort, testset, (double) 0.0);
                plot.show(0);
        }
        //update the relevant plot
        if (type == "query2" & end2 == false) {
                DoubleObj testset1 = new DoubleObj();
                queryHops_plot1 = queryHops_plot1 + 1;
                testset1.setValue(queryHops_plot1);
                send(thisPlotterPort1, testset1, (double) 0.0);
                plot.show(0);
        }

}
/**
 * Method to keep track of the number of hits are returned from query 1
 */
public void numberHits1() {
        hits = hits + 1;
        if (hits == 3) {
                end1 = true; //finishes plot1

        }
}
/**
 * Method to keep track of the number of hits are returned from query 2
 */
public void numberHits2() {
        hits = hits + 1;
        if (hits == 3) {
                end2 = true; //finishes plot 2

        }
}
/**
 * Method that initiates the actual simulation, called when the set up has been
done
 */

private void startSimulation() {
        for (int i = 0;
                i < configManager.getEventsConfiguration().getNoOfEvents();
                i++) {
                EventConfig eventConfig =

configManager.getEventsConfiguration().getEventConfiguration(i);
                String action = eventConfig.getAction();
                if (action.equals("QUERY")) {
                        System.out.println(
                                "  query time="
                                        + eventConfig.getIssueTime()
                                        + " node="
                                        + eventConfig.getNodeID()
                                        + " fileID = "
                                        + eventConfig.getActionParameterID());
                        QueryEvent queryEvent =
                                new QueryEvent(eventConfig.getActionParameterID());
                        sendAt(
                                commandPorts[(int) eventConfig.getNodeID()],
                                queryEvent,
                                (double) (eventConfig.getIssueTime()));
                }
                sendAt(
                        commandPorts[(int) eventConfig.getNodeID()],
                        action,
                        (double) (eventConfig.getIssueTime()));

        }
}

/**
```

```
 * Method, ends simulation, called when the finishtime has been reached
 */
private void endSimulation() {
        System.out.println("Exiting NOW");
        printTopology();
        double avg = getAvgNeighbour();
        System.out.println("the average neigbour size is" + avg);
        plot.showAll();
        // a wait for loop to give the plotter time to be displayed
        for (int f = 0; f <= 1000000000; f++);
}


/**
 * Method, Prints the simulation topology
 */
public void printTopology() {
        System.out.println("topology:");
        for (int i = 0; i < nodes.length; i++) {
                PeerReachNode node = nodes[i];
                System.out.println("  node: " + i);
                LinkedList neighborList = node.getNeighborList();
                Integer[] neighborArray =
                        (Integer[]) neighborList.toArray(new Integer[0]);
                String neighbors = "";
                for (int j = 0; j < neighborArray.length; j++) {
                        neighbors += " " + neighborArray[j];
                }
                System.out.println("    neighbors:" + neighbors);
                Connection connection = connections[i];
                System.out.println(
                        "    bandwidth(in/out): "
                            + connection.getInBandwidth()
                            + "/"
                            + connection.getOutBandwidth());
        }
        System.out.println(
                "this is the number of hits"
                        + hits
                        + "it took"
                        + hitTime
                        + "to get the first hit");
}
/**
 * Method, calculates the average size of the nodes community tables
 */
public double getAvgNeighbour() {
        ArrayList neighbourSizes = new ArrayList();
        int tot = 0;
        double avgNum;
        int numNeighbours;
        for (int i = 0; i < nodes.length; i++) {
                PeerReachNode node = nodes[i];
                LinkedList neighborList = node.getNeighborList();
                Integer[] neighborArray =
                        (Integer[]) neighborList.toArray(new Integer[0]);
                numNeighbours = neighborArray.length;
                neighbourSizes.add(i, new Integer(numNeighbours));
        }
        for (int j = 0; j < neighbourSizes.size(); j++) {

                if (j == 0) {
                        tot = ((Integer) neighbourSizes.get(j)).intValue();
                } else {
                        tot = tot + ((Integer) neighbourSizes.get(j)).intValue();

                }
        }
        avgNum = tot / neighbourSizes.size();
        return avgNum;
}

/**
 * Main method
 *
 */
public static void main(String args[]) {
```

```
		String configFilesPath = "C:/config/";
		Simulation simulation = new Simulation(configFilesPath, 10);
		simulation.start();
	}

}
```

C.I.ii PeerReachNode

```
package DIT.PeerReach.simulator;

import java.util.ArrayList;
import java.util.Hashtable;
import java.util.LinkedList;
import java.util.Random;
import DIT.PeerReach.vector.FileVector;
import DIT.PeerReach.vector.NodeVector;
import DIT.PeerReach.persistence.*;
import drcl.comp.Port;

/**
 * Represents the community level PeerReach Node
 *
 * @author Paul Stacey
  */

public class PeerReachNode extends Node {

	private int nodeID;
	private Random rand = new Random();
	private int connectivity;
	private String fileMatchingPolicy;
	private int ttl;
	private int routingTableSize;
	private int hostCacheSize;
	private int maxUploads; //need to use this in the upload section
	private int maxDownloads; //need to use this in the download section
	private Simulation simulation;
	private NodeConfig nodeConfig;
	private NodeVector nodeVector;
	private int[][] routingTable;
	private int insertPos; // for routingTable
	private int hostCache[];
	private ArrayList neighbourNodeVectorlist;
	private Hashtable nodeVectortoNodeIdList;
	private ArrayList nodeIdHintList;
	private LinkedList communityList;
	private LinkedList randomNeighbourList;
	private int numHints = 4; //number of community nodes to forward to.
	private int[] FileVectorList;
	private int[] randomNodeList;
	private float simMetric;
	private Port commandPort;
	private boolean firstQueryHitReceived;
	private int availableConnectivity;
	private long speed;
	private boolean active;
	private PersistentStore s = new PersistentStore();
	private String path = "C:/config/";
	private String cat1;
	private double filesimMetric = 0.3;

	private String cat2;

	/**
	 * Constructor
	 *
	 * Initialises node variables
	 */

	public PeerReachNode(int nodeID, Random random, Simulation simulation) {

		super(nodeID, random);
		this.nodeID = nodeID;
		this.simulation = simulation;
```

```
		nodeConfig =
			simulation
				.getConfigManager()
				.getNodesConfiguration()
				.getNodeConfiguration(
				(int) this.nodeID);
		connectivity = nodeConfig.getMaxConnectivity();
		FileVectorList = nodeConfig.getFileVectorList();
		cat1 = nodeConfig.getCat1();
		cat2 = nodeConfig.getCat1();
		nodeVector = nodeConfig.getNodeVector();
		ttl = nodeConfig.getPingTTL();
		routingTableSize = nodeConfig.getRoutingTableSize();
		hostCacheSize = routingTableSize;
		active = nodeConfig.getActive();
		availableConnectivity = connectivity;
		communityList = new LinkedList();
		insertPos = 0;
		initRoutingTable();
		commandPort = this.addPort("commandPort");
		firstQueryHitReceived = false;
	}
	/**
	 * Returns the list of node Ids of nodes directly
	 * connected to this node
	 */

	public Integer[] getNeighbors() {
		return (Integer[]) communityList.toArray(new Integer[0]);
	}


/**
	* Sets up a list of nodes and their asociated vectors
	*/

	public void setNeighbourNodeVectorlist(ArrayList nvVectorList) {
		neighbourNodeVectorlist = nvVectorList;
	}
	/**
	* Sets up a list of nodes and their asociated vectors
	*/
	public void setNodeVectortoNodeIDTable(Hashtable nodIdtoNodeVector) {
		nodeVectortoNodeIdList = nodIdtoNodeVector;
	}
	/**
	* Returns a list of handles to nodes conectedto this node
	*/

	public LinkedList getNeighborList() {
		return communityList;
	}
	/**
	* Sets the community tables for this node
	*/

	public void setNeighborList(LinkedList list) {
		communityList = list;
	}
	/**
	* Sets up the randomly connected neighbour tables
	*/

	public void setRandomNeighborList(LinkedList randomList) {
		randomNeighbourList = randomList;
	}
	/**
	* Initiates the routing table
	*/
	private void initRoutingTable() {
		routingTable = new int[routingTableSize][2];
		for (int i = 0; i < routingTableSize; i++) {
			routingTable[i][Simulation.DESCRIPTOR_ID] = -1;
		}
	}
	/**
	* Adds new route to the routing table
```

```
*/
private void insertRoute(int messageDescriptorID, int sourceID) {
        routingTable[insertPos][Simulation.DESCRIPTOR_ID] = messageDescriptorID;
        routingTable[insertPos][Simulation.SOURCE] = sourceID;
        insertPos = (int) (++insertPos % routingTableSize);
}
/**
* Checks if messageDescriptorID is present
*/
private boolean hasRoute(int messageDescriptorID) {
        for (int i = 0; i < routingTableSize; i++) {
                if (routingTable[i][Simulation.DESCRIPTOR_ID]
                        == messageDescriptorID) {
                        return true;
                }
        }
        return false;
}

/**
* Searches for files similar to searchVector
*/
public boolean searchFile1(FileVector searchVector) {
        boolean hit = false;

        for (int i = 0; i < FileVectorList.length; i++) {
                int file = FileVectorList[i];
                //read in file
                FileVector fv =
                        (FileVector) s.readObject(path + "/" + cat1 + "/" + file);
                double result = fv.compare(searchVector);
                if (result >= filesimMetric) {
                        hit = true;
                        simulation.numberHits1();
                }

        }
        System.out.println("No matching files forwarding query");
        return hit;

}
/**
* Searches for files similar to searchVector
*/

public boolean searchFile2(FileVector searchVector) {
        boolean hit = false;

        for (int i = 0; i < FileVectorList.length; i++) {
                int file = FileVectorList[i];
                //read in file
                FileVector fv =
                        (FileVector) s.readObject(path + "/" + cat1 + "/" + file);
                double result = fv.compare(searchVector);
                if (result >= filesimMetric) {
                        hit = true;
                        simulation.numberHits2();
                }

        }
        System.out.println("No matching files forwarding query");
        return hit;

}
/**
* The J-Sim process, data srrives through this nodes port
* into this method,
* this is the contract of this component
*/
protected void process(Object data, Port inPort) {
        //System.out.println("received something in the node");
        if (inPort == commandPort) {
                System.out.println("received on command port");
                //received a command from Simulation
                if (data instanceof QueryEvent) {
                        int fileID = ((QueryEvent) data).getEventParameterID();
                        if (fileID == 1) {
```

```
                                        generateQuery(fileID);
                                } else if (fileID == 2) {
                                        generateQueryRandom(fileID);
                                }
                        }

                } else if (inPort == connectionPort) {
                        System.out.println("received on the connection port");
                        if (data instanceof ProtocolPacket) {
                                ProtocolPacket inPacket = (ProtocolPacket) data;
                                receiveProtocolPacket((ProtocolPacket) data);
                                Query query = (Query) inPacket;
                                int type = query.getFileId();
                                System.out.println(
                                        "the type of the query that has arrived is" +
type);
                                if (type == 1) {
                                        processQuery(query);
                                        System.out.println(
                                                "received a query for community based
approach....now processing");
                                } else if (type == 2) {
                                        System.out.println(
                                                "received a query for random based
approach....now processing");
                                        processQueryRandom(query);
                                }

                        }
                } else {
                        System.out.println("sending to super node");
                        super.process(data, inPort);
                }
        }
        /**
        * Process a query over random node links
        */
        private void processQueryRandom(Query query) {

                simulation.updatePlot("query2");
                if (!hasRoute(query.getDescriptorID())) {
                        insertRoute(query.getDescriptorID(), query.getSenderID());
                        FileVector searchCriteria = query.getSearchCriterias();

                        if (query.getTTL() > 1) {
                                relayQueryRandom(query);
                        }
                        boolean hit = searchFile2(searchCriteria);
                }
        }


/**
        * Return the id of this node
        */
        public int getNodeId() {
                return nodeID;
        }

        /**
        * Return the list of nodes this node is randomly connected to
        */

        private int[] getRandomNodeIdlist() {

                randomNodeList = new int[randomNeighbourList.size()];
                for (int i = 0; i < randomNeighbourList.size(); i++) {

                        Integer randomnode = (Integer) randomNeighbourList.get(i);
                        int node = randomnode.intValue();
                        randomNodeList[i] = node;
                }
                return randomNodeList;

        }

        /**
```

```
* Return best nodes to forward search query to
*/
private ArrayList getHint(FileVector searchVec) {

        ArrayList nodeVectorHintList = new ArrayList();
        ArrayList similarityList = new ArrayList();
        ArrayList closestNodeVectorList = new ArrayList();
        for (int i = 0; i < neighbourNodeVectorlist.size(); i++) {
                NodeVector currentNodeVector =
                        (NodeVector) neighbourNodeVectorlist.get(i);
                //convert searchvec to a nodeVector vector to comapre both
                //NodeVector currentSearchVector = new
NodeVector(searchVec.getMatrix());
                double simmetric = currentNodeVector.compare(searchVec);
                nodeVectorHintList.add(i, currentNodeVector);
                similarityList.add(i, new Double(simmetric));
                if (i > 2) {
                        for (int j = 0; j <= 2; j++) {
                                double currentSimMetric =
                                        ((Double)
similarityList.get(j)).doubleValue();
                                if (simmetric > currentSimMetric) {
                                        nodeVectorHintList.add(i,
currentNodeVector);

                                }
                        }
                }
        }
        for (int j = 0; j <= 2; j++) {
                NodeVector closestNodeVector =
                        (NodeVector) nodeVectorHintList.get(j);
                closestNodeVectorList.add(closestNodeVector);
        }
        return closestNodeVectorList;
}

/**
* Generate a query to be forwarded over random node connections
*/
private void generateQueryRandom(int fileID) {

        System.out.println(
                "A query is being generated.....and forwarded randomly");
        int file1 = FileVectorList[0];
        Integer File1 = new Integer(file1);
        String fileone = File1.toString();
        String path1 = path + cat1;
        FileVector searchVector =
                (FileVector) s.readObject(path1 + "/" + fileone);
        int[] randomNodeIdList = getRandomNodeIdlist();
        int size = 25;
        int descriptorID = simulation.getNextDescriptorID();
        for (int i = 0; i < randomNodeIdList.length; i++) {
                this.insertRoute(descriptorID, nodeID);
                int receiverID = randomNodeIdList[i];
                Query query =
                        new Query(
                                nodeID,
                                receiverID,
                                size,
                                descriptorID,
                                Simulation.QUERY,
                                ttl,
                                (int) 0,
                                (int) 20);
                System.out.println(
                        "this is the query attribute:"
                                + nodeID
                                + " "
                                + receiverID
                                + " "
                                + size
                                + " "
                                + descriptorID
                                + " "
```

```
                                        + Simulation.QUERY
                                        + " "
                                        + ttl
                                        + " "
                                        + (int) 0
                                        + " "
                                        + (int) 20);
                        query.setFileId(fileID);
                        query.setMinSpeed(0);
                        query.setSearchCriterias(searchVector);
                        sendProtocolPacket(query);

                }

        }

        /**
        * Generate a query to be forwarded over community links
        */
        private void generateQuery(int fileID) {

                System.out.println("A query is being generated.....");
                int file1 = FileVectorList[0];
                Integer File1 = new Integer(file1);
                String fileone = File1.toString();
                String path1 = path + cat1;
                FileVector searchVector =
                        (FileVector) s.readObject(path1 + "/" + fileone);
                nodeIdHintList = getHint(searchVector);
                int size = 25;
                int descriptorID = simulation.getNextDescriptorID();
                if (nodeIdHintList.size() == 0) {
                        System.out.println("Node has no community connections!");
                }
                for (int i = 0; i < nodeIdHintList.size(); i++) {
                        this.insertRoute(descriptorID, nodeID);
                        System.out.println(
                                " Node: "
                                        + nodeID
                                        + " generated QUERY with descriptorID "
                                        + descriptorID
                                        + " for FILE "
                                        + fileID);
                        //get the nodId asociated with this NodeVector
                        NodeVector currentnv = (NodeVector) nodeIdHintList.get(i);
                        int receiverID =
                                ((Integer)
nodeVectortoNodeIdList.get(currentnv)).intValue();
                        Query query =
                                new Query(
                                        nodeID,
                                        receiverID,
                                        size,
                                        descriptorID,
                                        Simulation.QUERY,
                                        ttl,
                                        (int) 0,
                                        (int) 20);
                        System.out.println(
                                " Node: "
                                        + nodeID
                                        + " sent QUERY with descriptorID "
                                        + descriptorID
                                        + " to Node: "
                                        + receiverID);
                        query.setMinSpeed(0);
                        query.setSearchCriterias(searchVector);
                        query.setFileId(fileID);
                        sendProtocolPacket(query);
                }

        }
```

```
/**
* Process a query that has arrived to be forwarded over
* community links
*/

        private void processQuery(Query query) {
                simulation.updatePlot("query1");
                if (!hasRoute(query.getDescriptorID())) {
                        insertRoute(query.getDescriptorID(), query.getSenderID());
                        FileVector searchCriteria = query.getSearchCriterias();
                        if (query.getTTL() > 1) {
                                relayQuery(query);
                        }
                        boolean hit = searchFile1(searchCriteria);
                }
        }

        /**
        * Forward query over random links
        */
        private void relayQueryRandom(Query query) {
                FileVector searchCriteria = query.getSearchCriterias();
                int[] randomNodeIdList = getRandomNodeIdlist();
                int size = 25;
                int descriptorID = simulation.getNextDescriptorID();
                for (int i = 0; i < randomNodeIdList.length; i++) {
                        int receiverID = randomNodeIdList[i];
                        System.out.println("Forwarding query to" + receiverID);
                        if (query.getSenderID() != receiverID) {
                                Query newQuery = query.propogate(receiverID);
                                newQuery.setFileId(query.getFileId());
                                sendProtocolPacket(newQuery);
                        }

                }

        }

        /**
        * Forward query over community links
        */
        private void relayQuery(Query query) {
                FileVector searchCriteria = query.getSearchCriterias();
                nodeIdHintList = getHint(searchCriteria);
                int size = 25;
                int descriptorID = simulation.getNextDescriptorID();
                if (nodeIdHintList.size() == 0) {
                        System.out.println("Node has no community connections!");
                }
                Integer[] neighbors = (Integer[]) communityList.toArray(new Integer[0]);
                for (int i = 0; i < nodeIdHintList.size(); i++) {
                        NodeVector currentnv = (NodeVector) nodeIdHintList.get(i);
                        int receiverID =
                                ((Integer)
nodeVectortoNodeIdList.get(currentnv)).intValue();

                        System.out.println("Forwarding query to" + receiverID);
                        if (query.getSenderID() != receiverID) {
                                Query newQuery = query.propogate(receiverID);
                                newQuery.setFileId(query.getFileId());
                                sendProtocolPacket(newQuery);
                        }
                }
        }
}
```

Appendix D

This Appendix supports the unified modelling language [114] (UML) diagrams that are included within the main text. The UML diagrams have been generated using the Omondo UML plugin [115] developed for the Eclipse Java development studio [116]. Omondo supports the OMG UML version 1.5 specification. The discussion below gives an explanation of the various components that appear in the UML diagrams.

D.I. Unified Modelling Language 1.5

To give a brief explanation of the UML diagrams that appear in the main text a UML example is presented. A class is drawn with three compartments separated by horizontal lines. The top name compartment holds the class name; the middle list compartment holds a list of attributes; the bottom list compartment holds a list of operations or methods.

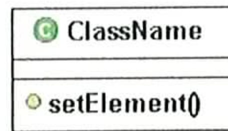

Figure D.i. UML Class

An association between two classes is drawn as a solid line.

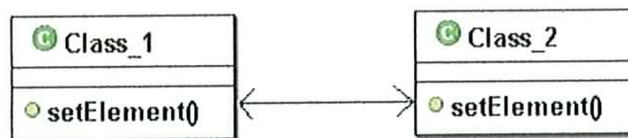

Figure D.ii. Association

A dependency is represented by a dashed line with an open arrow.

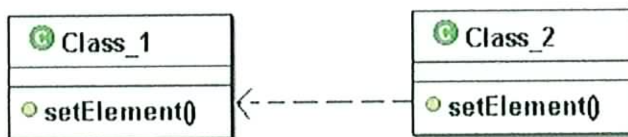

Figure D.iii. Class Dependency

Inheritance between two classes is represented by a sold line with a closed rectangle.

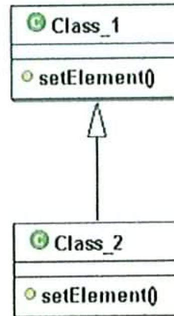

Figure Div. Inheritance

An interface is a classifier and is shown using only the name compartment with the name of the interface and the <<Interface>> tag. The relationship from a classifier to an interface that it supports is shown by a dashed line with a solid triangular arrowhead.

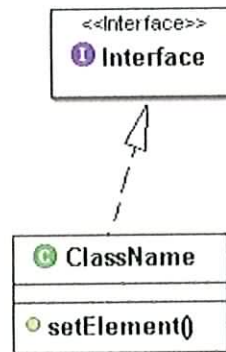

Figure E.v. Relationship from a Classifier to an Interface

A group of classes grouped within the same package is represented as is seen below in figure D.v.i. Interactions between packages are shown using associations and dependencies as described above.

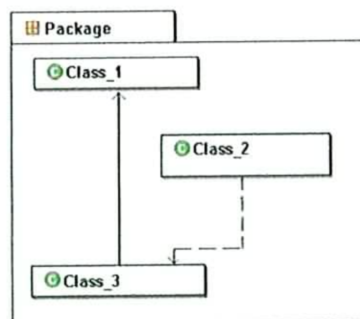

Figure D.vi.. Representation of a Package