Winter 2007

FreeLib: A Self-Sustainable Peer-to-Peer Digital Library Framework for Evolving Communities

Ashraf A. Amrou
Old Dominion University

FREELIB: A SELF-SUSTAINABLE PEER-TO-PEER DIGITAL LIBRARY FRAMEWORK FOR EVOLVING COMMUNITIES

by

Ashraf A. Amrou
M.S. February 2001, Alexandria University
B.S. June 1995, Alexandria University

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
December 2007

Approved by:

Kurt Maly (Co-Director)

Mohammad Zubair (Co-Director)

Hussein Abdel-Wahab (Member)

Ravi Mukkamala (Member)

Mohamed Younis (Member)

ABSTRACT

FREELIB: A SELF-SUSTAINABLE PEER-TO-PEER DIGITAL LIBRARY FRAMEWORK FOR EVOLVING COMMUNITIES

Ashraf Amrou
Old Dominion University, 2007
Co-Directors of Advisory Committee: Dr. Kurt Maly
Dr. Mohammad Zubair

The need for efficient solutions to the problem of disseminating and sharing of data is growing. Digital libraries provide an efficient solution for disseminating and sharing large volumes of data to diverse sets of users. They enable the use of structured and well defined metadata to provide quality search services. Most of the digital libraries built so far follow a centralized model. The centralized model is an efficient model; however, it has some inherent problems. It is not suitable when content contribution is highly distributed over a very large number of participants. It also requires an organizational support to provide resources (hardware, software, and network bandwidth) and to manage processes for collecting, ingesting, curating, and maintaining the content.

In this research, we develop an alternative digital library framework based on peer-to-peer. The framework utilizes resources contributed by participating nodes to achieve self-sustainability. A second key contribution of this research is a significant enhancement of search performance by utilizing the novel concept of community evolution. As demonstrated in this thesis, bringing users sharing similar interest together in a community significantly enhances the search performance. Evolving users into communities is based on a simple analysis of user access patterns in a completely distributed manner. This community evolution process is completely transparent to the

user. In our framework, community membership of each node is continuously evolving. This allows users to move between communities as their interest shifts between topics, thus enhancing search performance for users all the time even when their interest changes. It also gives our framework great flexibility as it allows communities to dissolve and new communities to form and evolve over time to reflect the latest user interests. In addition to self-sustainability and performance enhancements, our framework has the potential of building extremely large collections although every node is only maintaining a small collection of digital objects.

ACKNOWLEDGMENT

My sincere thanks and appreciation go to the following people:

- My family; my father, my mother, and my wife for their patience and endless support.
- My advisers Dr. Kurt Maly and Dr. Mohammad Zubair for their guidance on my research and editing of this manuscript.
- Members of my graduate committee Dr Hussein Abdel-Wahab, Dr. Ravi Mukkamala, and Dr. Mohamed Younis for their valuable feedback and comments that helped me enhance this work.

TABLE OF CONTENT

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I
INTRODUCTION

1.1 Motivation

With the current growth of Internet and advances in content creation tools, a large amount of digital content is created every day. This content belongs to organizations as well as individuals from diverse communities. Examples of such content include technical reports, research papers, and datasets, generated by researchers and faculty members in universities and research centers around the world. Additional examples include lecture notes, presentations, and assessment material created by instructors in schools and universities. Sharing and utilization of such heterogeneous content represents a great challenge as it is highly distributed over machines, users, topics, and geographic locations. In addition its distributed nature, much of this content lacks standard and unified ways for organizing and accessing it. Furthermore, it lacks the appropriate metadata management and utilization. Utilization of metadata can facilitate search and discovery of content.

Many people utilize the World Wide Web to enable sharing. In this approach, content is uploaded to websites and made available for indexing by web search engines. Web search engines use crawlers to traverse the web following links from page to page. They index the content and may cache some of it as well. Once the content is indexed, it starts to appear as search results when people search using relevant keywords that exist in the content. This approach is straightforward. However, this approach suffers from poor precision. Results for a query might reach millions of items while the user is looking for

---

The model journal for this dissertation is: IEEE Transactions on Software Engineering

few relevant items. In addition, with the current web search technology, it might take days or weeks until newly added content gets indexed by search engines.

Another approach to sharing and dissemination is to build and maintain large digital collections. In this approach, each digital collection or digital library is typically dedicated to a certain topic or a certain user community. Example digital libraries of this type include ACM digital Library [1] and the IEEE computer society [25]. This approach suffers from sustainability-related issues. Some the organizations charge subscription fees to provide the necessary resources to maintain such collections. This imposes a barrier that limits participation from many individuals. And consequently, content in these libraries is very restricted. Some other organizations maintain digital libraries on pro bona basis, and as the community interest withers, these libraries cease to exist.

We need a digital library framework that is decentralized, self sustainable, and focus on individual publishers. Such framework must enable individual users to seamlessly publish and search/discover content. It also must support distributed users as well as distributed content. In addition, it must be self-sustainable in terms of securing the necessary resources required to operate and maintain the collection. Furthermore, it must promote effective use of metadata in publishing as well as in search and provide the necessary tools to enable that. The use of metadata in search can help the user specify more accurate queries that target the relevant documents. We shall discuss metadata-based search in more details later in this chapter.

1.2 Problem Statement

Individual users face many challenges when publishing, disseminating, and searching digital content. These challenges include:

- Limited resources: Individual users do not have the necessary resources to build and maintain huge collections. These necessary resources include storage space, network bandwidth, and the software programs to index, maintain, and provide access to the content. We refer to this as the Sustainability problem.
- Limited ability to effectively search and locate relevant content: This problem is in part due to the tremendous amounts of content available and the lack of good metadata that can enable effective search and discovery of content. Often, users are presented with huge result sets that contain tens of thousands of items while they are looking for few relevant documents.
- Distributed users and content: A distributed system that reflects distributed users and content usually faces performance issues when compared to a centralized system. Distributed protocols are usually more complex and have communication overhead.
- Adapting to shifting interest: User interest usually shifts between topics over time, which increases the challenges we face when building efficient solutions for dissemination and discovery for digital content.

We need a framework that addresses the above challenges and provides enhanced user experience. In the following section, we discuss our approach to address these issues.

1.3 Approach

In this dissertation, we present a peer-to-peer-based digital library framework that addresses the above issues.

First, to address the distributed nature of users and content, the proposed framework is based on a pure peer-to-peer system. In this framework, each participant uses a software client to maintain a small collection locally on the user's machine. In the general scenario, the local collection on a user's machine contains the publications of the local user as well as replicas of content from other users. When a user submits a search query, it is forwarded by the client to other users in a peer-to-peer fashion and results are collected and presented to the user. This framework suits the distributed nature of the targeted body of individual users. As we can see from this description of the framework, the storage and bandwidth resources are contributed by the individual participants. This helps alleviates the need for dedicated resources and ensures the self-sustainability of the proposed digital library framework.

Second, we utilize the key notion of *targeting relevant peers with search queries* to enhance system performance. In order to target the relevant peers, the proposed framework connects the users in the peer-to-peer network such that users having similar interest are close to each other on the network topology. This approach results in higher recall, lower bandwidth usage, and better response time for search queries as relevant results are typically within few hops away from the user submitting the query. Connecting users sharing similar interest together is an ever-evolving process that we call community evolution. In this process, the client on every node monitors and records accesses to and from its node. The client periodically performs ranking of nodes based on the access information and identifies the peers with the highest ranks. The ranking formula is based on the frequency of mutual access between the node and each peer.

After ranking is performed by a node, the node establishes friend[1] links to a few nodes from the top of its ranked list. The union of the friend links over all the nodes gives the *access network* topology. The ranking process is performed in a distributed fashion. All the information needed for ranking is stored locally by each node. This way, minimal communication between nodes is needed after the ranking is finished to update the topology. By constructing the peer-to-peer topology according the mutual interest between nodes, search queries reach relevant nodes in few hops and results are returned faster than other ad hoc topologies. Instead of having to use a large time-to-live (TTL) value for search messages to reach all nodes the whole network, which might not be feasible for large network; with our community evolution technique, a small TTL is sufficient as relevant peers are within few hops on the access network overlay topology.

And third, the proposed architecture utilizes Dublin Core metadata [16] in both publishing and search. The choice of DC metadata is for its simplicity and to enable interoperability as DC is a metadata standard utilized by an increasing number of digital libraries. Our peer-to-peer client, which we call the Freelib client, provides a GUI interface for the users to publish their metadata. Metadata can be typed in when the user publishes new content to her local collection. However, our proposed architecture is extensible. It allows addition of tools for automatic extraction and validation of metadata from full-text documents to alleviate the need for the user to manually enter the metadata. We also utilize metadata in search queries. The Freelib client provides a GUI interface for the users to specify search queries based on the metadata fields. The metadata-based

---

[1] We refer to the topology links that we build based on access patterns as "Friend links" and to their target nodes as "Friends". We use the terms "access network" and "access topology" to refer to the overall topology that consists of the links we build based on access patterns as well the nodes connected by these links.

search gives the user more power in terms of the ability to create more accurate queries that target the relevant items and avoids irrelevant ones. To further illustrate this, consider the following example. Consider a collection that contains 100 documents whose author last name is 'Cash'. Also assume that 1000 documents in this collection contain the word 'Cash' somewhere in the document. If we use metadata-based search based on the author field, the maximum number of documents we can get is 100 all of which are relevant, which represent recall of 100%. If the system returned 75 documents, then recall is 75%. All the returned documents are relevant and hence the precision is 100%. On the other hand, for a system that uses general keyword search, the maximum number of documents that could be returned is 1000, which represent 100% recall and only 10% precision. If the system returned 700 documents containing 60 documents that are relevant (author last name is 'Cash'), then recall is 60% and precision is 8.57% (60 divided by 700).

Use of metadata contributes to a significant reduction in the time spent on activities related to organizing and searching/locating digital objects. It reduces the time spent on these activities by 18.1% to 43.5% [14]. In addition to significantly reducing the cost for maintaining a digital collection, these savings results in efficient use of resources such as network bandwidth and processing power. Besides the above advantages, use of standard metadata promotes interoperability with other systems and facilitates future development and extensions.

1.4 Contributions

The main contributions of this research are *development of a self-sustainable digital library framework* suitable for individual users and *significant enhancement of*

search performance. We utilized a peer-to-peer network as the basis for building the digital library. The use of the peer-to-peer model provides the desirable features of self-sustainability and supporting distributed users and content. However, it generates many challenges especially in terms of performance. In summary, we:

- Researched and developed *an alternative framework that produces a self-sustainable digital library.*
- Devised mechanisms to support evolution of communities with diverse and dynamically changing interest areas, which *enhanced search performance in terms of response time, recall, and precision.*
- Implemented the proposed framework to show the feasibility of our approach.
- Built a testbed and developed a simulator, which enabled us to evaluate the proposed framework.

1.5 Organization of the Dissertation

The rest of this dissertation is organized as follows:

Chapter II – Background and Related Work: In chapter II, we discuss the background and overview related work in the areas of digital library and peer-to-peer systems.

Chapter III – Network Architecture: In chapter III, we present the network architecture and the protocols for building and evolving the peer-to-peer topology. These protocols include the algorithms for performing the ranking of peers according to mutual access.

Chapter IV – Node Identity and Discovery: In chapter IV, we discuss how we generate system-wide unique node identities and present our discovery protocols.

Chapter V - Client Design and Implementation: In chapter V, we present the peer-to-peer client design and describe our implementation.

Chapter VI – Performance Evaluation: In chapter VI, we describe a testbed that we have constructed and used for testing and preliminary evaluation. We discuss the challenges that we have faced during building the testbed. Then, we present our event-based simulator design and implementation. Finally, we present the performance evaluation of the proposed framework along with results from our simulation experiments.

Chapter VII – Conclusion and Future Work: In chapter VII, we present our conclusions and discuss possible ways for enhancing and extending the work presented in this dissertation.

CHAPTER II
BACKGROUND AND RELATED WORK

Our work builds on research work from various fields including digital library, peer-to-peer, social science, and information retrieval. In this chapter, we discuss the relevant background and related work in those areas.

2.1 Digital Library

Digital library is an active research field dealing with issues regarding building digital collections of items (also called digital objects) and providing services to the various users of these collections including publishers, readers/searchers, librarians/content managers, and administrators. An item in a digital collection typically contains content and metadata describing various attributes of the content. Content could be documents, images, or any type of data. Every item in a collection is associated with one or more identifiers. Unlike other information systems, a digital library maintains collections that are rich in metadata, which is essential for enabling better search and retrieval services. Metadata describe properties of the actual data such as its structure, encoding, author, date of publishing, and title. The use of metadata enables better services, promotes interoperability, facilitates maintenance, and allows future developments and extensions.

Digital libraries are usually compared to other systems such as traditional libraries, containing physical books and other publications, and the World Wide Web. In contrast to the traditional libraries; in a digital library, content is in digital format. Users of digital libraries do not have to be in a specific place to access the various services. All what a user needs to access the service are a computer and an Internet connection. Another difference between digital libraries and traditional libraries is the additional

services that are suitable for the digital format, e.g., automatic search inside the full-text which is not feasible to implement in a traditional library.

Some of the differences between the Web and digital libraries are as follows. First, digital libraries maintain well structured and well organized content. Second, digital libraries provide quality content that is, the content is usually curated. Third, the metadata a digital library maintains for each object enable easy discovery of information and resources. Fourth, they provide uniform interface to the services while web sites are not uniform. For example, web sites follow different page layouts, styles, and fonts. Digital libraries also provide better search service. In addition, as we mentioned earlier, current web search technology covers a small part of the Web's content and it might take considerable amount of time until newly added content is indexed by web search engines.

We have witnessed a dramatic increase in the number of digital libraries built recently. The most common digital library model adopted by these digital libraries is the centralized model. There are also efforts to research and adopt other models such as digital library federation, distributed, and peer-to-peer models. In the following subsections, we discuss these models in greater details.

2.1.1 Centralized Model

In this model the client is usually a web browser through which the user interacts with the digital library. User requests are submitted to the server-side and responses are sent back to the user. In this model, the collections are stored and indexed using server machines running software components such as web servers, database servers, and search engines. The server side usually consists of back-end processing engines and front-end components providing the various services to its users.

There are many professional as well as experimental digital libraries that use the centralized model. Examples of centralized digital libraries that maintain collections of publications include the ACM digital library [1] and the IEEE computer society digital library [25]. The Alexandria Digital Library (ADL) [2, 19] is a digital library containing collections of geographically referenced information. In a typical search request, the user specifies a region on the map and submits his search. The results returned are those items associated with locations within the region the user specified. FEDORA (stands for Flexible Extensible Digital Object and Repository Architecture) [17, 64] is a general purpose digital object repository system that can be used by variety of communities and for various purposes; e.g., institutional repositories, scholarly publishing, and digital preservation all might use FEDORA. Although the basic software components of these digital libraries might be distributed over machines in an intranet or the Internet, e.g., ADL [2, 19], we still consider this a centralized model. Communication between these digital libraries and their users still follow the client-server model, it uses client-server protocols such as HTTP and there is a central administrative body that decides on policies, processes, and enforcements.

2.1.2 Digital Library Federation

Digital library federation efforts aim at providing digital services spanning multiple autonomous digital library collections. The *Open Archives Initiative* (OAI) [50], preceded by the Santa Fe Convention [73], was the first organization that defined 'federation' in the context of digital libraries to promote interoperability. In OAI, two roles are defined, the *service provider* (SP) and the *data provider* (DP). The service provider is a fully functional digital library that offers services to the users. The data provider is an

autonomous digital library maintaining its local collections and its own user interface to serve its local users. In the general scenario, one or more SPs offer services that span many DPs. OAI uses the 'harvesting' of metadata method to establish a federation, however another model to realize a federation exists: the 'distributed' model. We shall next describe these two methods.

2.1.2.1 Digital Library Federation by Harvesting

In this approach, SPs harvest metadata from the individual DPs, using the harvesting protocol OAI-PMH (Open Archives Initiative-Protocol for Metadata Harvesting) [35, 50], and store the metadata on the federation server. SPs might also cache full-text. The service is provided based on the harvested metadata which usually contains links to both the original items as well as cached versions if available. The harvesting protocol is a request response client-server protocol with DPs implementing the server-side and service providers implementing the client-side. Harvesting is typically done periodically to achieve some degree of consistency with the original collections.

Example federated digital libraries include Arc [3, 39], and Kepler [33, 42]. In these digital libraries, the number of data providers is typically in the range of tens to hundreds or even few thousands. Both Arc and Kepler use the OAI-PMH protocol. Another example is the Technical Report Interchange project (TRI) [67]. TRI enables member institutions (NASA, LANL, Sandia, and AFRL) to exchange metadata of technical reports so that authorized users at any of these organizations have access to all the collections of technical reports. Every TRI member institution uses OAI-PMH to harvest metadata from other members.

2.1.2.2 Federation through the Distributed Model

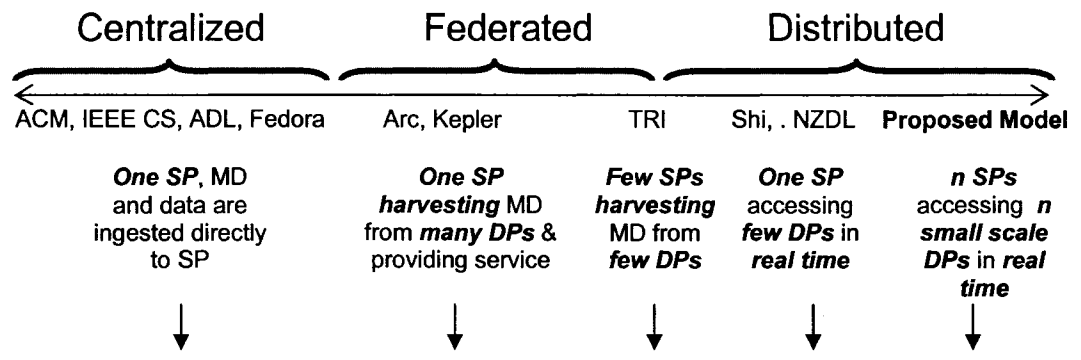
In this model, the SP is reduced to merely a unified user interface. Search requests, entered using this unified user interface, are sent in real-time to the individual member collections or digital libraries, which process them and return results to the service provider. The SP collects results from individual digital library and presents them to the user. In this approach, no metadata harvesting is used. All the communications are done in real-time and the requests are entirely processed by the individual collections. The SP uses a client-server protocol to send requests and receives results from the member digital libraries.

[60, 61] introduced a framework to federate a group of disparate digital libraries by means of specifying the input/output behavior of target collections. The authors developed a language for specifying these behaviors and an engine that could then process a user's request and translate it into the corresponding input queries of the members of the federation. Similarly the engine would use the specification to translate the output of all responding collections and transform them into one uniform format and present the merged responses back to the user. The New Zealand digital library group's distributed digital library architecture [44, 49] is an example digital library that follows this model. It enables searching multiple collections in real time through a unified user interface. The New Zealand digital library group defined a protocol for communication between the unified user interface and the individual collections. They implemented their system with only two individual member collections and they claim their architecture is extensible. This approach is not completely distributed, however. All the requests have to be entered through the unified user interface, which introduces some sort of centralization

into the system. This could cause a bottleneck as the number of users and individual collections increase.

There have been some attempts to build peer-to-peer based digital libraries. P2P-4-DL [69] is an example of such efforts. It uses a model similar to Napster [8, 46] which is a peer-to-peer application with a centralized server maintaining an index of the content. The search is done in a completely centralized fashion at the server and downloads are done in a peer-to-peer fashion. A digital library architecture based on the super-peers variant of the peer-to-peer model is presented in [12]. In this model, the OAI-PMH, which is originally a client-server protocol, is extended for use in a peer-to-peer fashion. Super-peers in the architecture harvest collections from normal peers, store and index them locally, and provide search service based on the harvested metadata. As mentioned earlier, introducing centralized components could cause bottlenecks as the number of users grows. In addition, it causes the system to be failure-prone. If the centralized component fails or gets attacked, the whole service fails.

Figure 2.1 shows the distribution spectrum in the context of digital library with example digital libraries in the various categories. In the centralized digital library model, the same exact centralized system plays both roles of DP and SP. On the other extreme, a peer-to-peer-based system, like our proposed framework, has many DP's all of which cooperate to provide the service. The term Query type in Figure 2.1 refers to whether the query is processed locally or in a distributed fashion and for the local case whether the metadata is harvested or originally local.



# SPs	1	1	Few (< 10)	1	n*
# DPs	N/A	10's to 100's or 1000's	Few (< 10)	Few	n*
Query type	Local (Local MD)	Local (Harvested MD)	Local (Harvested MD)	Distributed (Real time)	Distributed (Real Time)

*Large number of small scale autonomous SP/DP s.

Figure 2.1: Digital library distribution spectrum.

2.2 Peer-to-peer Systems

Peer-to-peer (P2P) is an alternative for the traditional client-server model for providing services over a network. Peer-to-peer systems are special cases of distributed systems in which the number of nodes is extremely large. In pure peer-to-peer systems, there are no servers. Rather, peers cooperate to provide the service to the users. Peer-to-peer systems are self-sustainable in terms of the resources the system needs such as storage, processing power, and network bandwidth. In the peer-to-peer model, these resources are contributed by the individual nodes. Although each node usually contributes few resources, with the large number of participant, these contributions form a huge distributed system with considerable amounts of resources. This gives peer-to-peer system potential for high scalability and self-sustainability as the number of participants grows. In addition to these advantages, peer-to-peer systems:

- Enable the utilization of the resources at the perimeter of the network. This is especially important as the personal computers forming the perimeter are becoming more and more powerful in terms of processing power and storage capacity;
- Have potential for high fault tolerance. Since resources and users are highly distributed, failures are usually partial. A failure typically affects some part of the system, while the rest of the system continues to provide the service. Most peer-to-peer systems utilize replication to enhance the availability of the content in presence of these partial failures; and
- Have potential for bringing together people from different geographic locations into forming virtual communities and groups.

Unlike the client-server model, machines in a pure peer-to-peer system are equal. There are no centralized powerful servers; rather, peers cooperate to provide the service. Every machine in such system is running the same software client that realizes the P2P application. Every machine is both a producer (small server) and a consumer (client) of the service, not necessarily at the same time. Due to this fact, some peer-to-peer systems such as Gnutella [21], call their software implementation a *servent* (a combination of the words 'server' and 'client').

2.2.1 Peer-to-Peer Models

There are several peer-to-peer models in use in various applications. Some peer-to-peer systems follow models that have some centralized components. Others follow a pure peer-to-peer model. Some peer-to-peer systems use a hierarchical model called super-peer model. We shall describe the main variants in the following sub-sections.

2.2.1.1 Centralized Indexing

Some models utilize centralized components. For example, Napster [8, 46], the first peer-to-peer file sharing system, uses a centralized server to index all the content available on all the connected peers., In this model, while search takes place at the server, actual downloads are performed in a peer-to-peer fashion in which nodes communicate directly with each other. It is worth mentioning that Napster, in its original form, was closed in June 2001 due to copyright issues and it is now a paid service.

2.2.1.2 Pure Peer-to-Peer Model

Freenet is an example of a peer-to-peer model; in this model, all communications including search requests are performed in a peer-to-peer fashion. Search requests are forwarded from node to node up to a certain hop count (also called time-to-live or TTL) on the overlay topology (in a P2P system all nodes have links to selected other nodes forming the network). Some systems uses breadth-first search and results are sent back to the original requester, e.g., Gnutella [21]. Other systems use depth-first search, e.g., Freenet [9, 18]. In Freent, for the purpose of anonymity, results follow the same path back to the original requester as queries were forwarded.

2.2.1.3 Hierarchical Model

In hierarchical architectures or what is also called *super peer* model, the system attempts to utilize the heterogeneity of the participating nodes. These systems introduce more powerful nodes (especially in terms of network bandwidth) to become super nodes. These super nodes are responsible for indexing content available on regular nodes and handling

search requests. Kazaa [31] is an example of a super peer network. A study of super-peer networks and guidelines for building them is available in [71].

2.2.2 Applications of Peer-to-Peer Model

Peer-to-peer concepts are utilized in many applications. For example, Domain Name Systems, invented by Paul Mockapetris in late 1983 [56, 57], and News Servers communicate and provide their services in a peer-to-peer fashion. Other more recent applications of peer-to-peer include file sharing and distributed computing (cycle sharing). In the following subsection we present some of these applications.

2.2.2.1 File Sharing

File sharing enables users to share files as implied by its names. Participants place the files they want to share in specific folders and the peer-to-peer client makes these folders available to people searching the network. Napster [8, 46] is an example of a centralized file sharing peer-to-peer system. Examples of pure peer-to-peer file sharing systems include Gnutella [21], Freenet [9, 18], and Bittorrent [6]. Kazaa [31] is an example of a super-peer file sharing system.

2.2.2.2 Distributed Computing

Distributed computing is a peer-to-peer application that enables sharing processor cycles. Each participant uses a client that monitors the local computer processor usage and when the computer is idle, it downloads and runs tasks. By contributing only relatively little processing power that is otherwise wasted, individuals create a group computer whose processing power could reach orders of magnitudes of the most powerful machine available. Computation-intensive scientific applications benefit the most from such peer-

to-peer system. Example systems include SETI@ home [59], United Devices [68], DataSynapse [15], and distributed.net [13].

2.2.2.3 Distributed Search

Distributed search is another peer-to-peer application in which search requests are submitted to individual *search sites* and results are categorized, ranked, and presented to the user. Example distributed search systems are Copernic [11], which searches many of the popular Web search engines simultaneously. Other example include RetrievalWare [55], which enables searching the files and documents on an organization intranet computers), and JXTA Search [30].

2.2.2.4 Distributed Hashtables

Another peer-to-peer application is Distributed hashtables (DHT). Distributed hashtables are distributed versions of the regular hashtable programming structure, which is very efficient in retrieving items given their identifiers. A hash table is a set of mapping from key values to digital objects. The basics operations supported by a hash table abstract data type are *insertion* in the form *insert (key, value)* which associates the given value with the specified key, and *retrieval* in the form *retrieve (key)* which returns the value associated with the specified key if there is one. A DHT is a hash table whose entries are distributed over nodes of a peer-to-peer or a distributed system. Every peer stores and provides access to part of the hash table. Entries of the DHT are usually distributed uniformly over the participating nodes for the purpose of load balancing. Insertion and retrieval requests are routed to nodes responsible for the key specified by the request. Example DHT protocols are Chord [65], Symphony [43], CAN [54], TAPESTRY [72],

and PASTRY [58]. DHT protocols utilize various overlay topologies. Some protocols use ring structure, e.g., Chord [65] and Symphony [43]. Other protocols arrange nodes in *m*-dimensional space, e.g., CAN [54].

2.2.2.5 Other Miscellaneous Peer-to-Peer Applications

Other applications of peer-to-peer include *instant messaging* and *group collaboration.* Instant messaging enables users to send and receive instant messages in real time. Example instant messaging systems include MSN messenger, Yahoo messenger, ICQ, and AOL instant messenger. Group collaboration enables multiple users to participate in group projects online. It supports services like annotation, adding comments, and sharing documents across multiple machines. These systems usually have some degree of integration with email clients and browsers. Example collaboration systems include Groove Networks [23], and IntraLinks [26].

As hardware, software and networking technologies advance, the future might witness the introduction of more sophisticated peer-to-peer applications. For example a *distributed storage* application may enable sharing at the level of storage blocks. Blocks contributed by participants form a huge storage media. In such application, portions of one file might be stored on different physical machines. Another possible application is *licensed media distribution* application that may enable legal online distribution of media in a Napster- or Gnutella-like model. In such application, file transfers are tracked, officially authorized, and would generate the appropriate revenue to the copyright holders.

2.2.3 Peer-to-Peer Search Issues

Peer-to-peer search is usually done by forwarding search requests from node to node over the peer-to-peer overlay topology. In order to prevent endless loops and for bandwidth considerations, peer-to-peer search uses a reasonable time-to-live value. This limits the *search coverage* as a consequence search requests might not reach all relevant nodes in the network. Another issue is search performance. Efficiency of the search is affected by the time taken to forward requests over the network, which might result in poor response time especially when the too many requests are being processed simultaneously. *Quality of results* is also an issue in peer-to-peer search as most of the known peer-to-peer systems lack appropriate metadata and full-text search techniques. For example, almost all of the peer-to-peer file sharing systems do search based on file names, which are most of the time misleading. This wastes user time in browsing and sometimes downloading irrelevant results.

There have been many attempts and efforts to enhance peer-to-peer search performance and quality of results. These include *topic segmentation* [5] and use of super-peer models [31, 71]. Topic segmentation clusters member nodes based on the similarity of the content such that nodes containing similar material are connected together. This approach is, however, very complex and does not evolve quickly as user interest changes. To measure the similarity of documents, these documents are represented in some data structure such as a keyword vector. A clustering algorithm is run on documents to cluster them. The distributed nature of the system adds to the complexity of this protocol.

Relevant to our proposed model are certain concepts and features of peer-to-peer search mechanisms. First, the concept of the small world property enhances the reachability and coverage of searches in P2P networks. We shall discuss the small world property in more details shortly. Second, our proposed solution evolves participants into communities of common interest using simple access pattern analysis. Similar work that infers communities in the context of the World Wide Web based on link topologies between pages was presented in [20]. That work is page-oriented. It clusters web pages into communities based on the link topology connecting those pages. It does not have the concept of accesses or the user-centric approach that we emphasize in our approach.

2.3 Social Networks and the Small-World Property

Social networks are networks of people in which nodes represent persons and a link represents a relationship (for example, the two persons know each other, or are friends, or both like classical music) between the two persons at both ends. In a large social network, any two arbitrary people were found to be connected to each other through a short chain of intermediate acquaintances [34, 70] and hence the name small-world. Small-world networks are typically characterized by two properties. The first is a small average path length between nodes. The other is a high clustering coefficient (the probability of two neighbors of a node to be neighbors themselves). The average path length between two members of the network is called the *diameter* of the network. The small world property is desirable in the context of peer-to-peer and large-scale distributed systems as messages can be routed from source to destination in a number of hops smaller than the diameter. There have been some efforts to build peer-to-peer networks whose overlay topology has this small world property. Symphony [43], for example builds a peer-to-peer network

with diameter $(\log^2(n)) / k$, where n is the number of nodes in the network and k is the number of contacts (links) per node. Chord [65] is another example with diameter $\log(n)$, where n is the number of nodes in the network. In order to maintain this small diameter, every node in Chord establishes $\log(n)$ contacts. Both of Symphony and Chord use a ring overlay topology in addition to long distance contacts. CAN [54] could be viewed as a small world network. As mentioned earlier, it arranges nodes in k-dimensional space. It has a diameter that is $O(k n^{1/k})$, where k is the number of dimensions and n is the number of nodes in the network.

Semantic small world [38] clusters nodes based on similarity of content and uses a linearization technique to arrange the set of clusters in a line. Peers establish short contacts to neighbor clusters and long contacts to clusters far away. These protocols require complicated clustering techniques and do not evolve quickly as users' interests change.

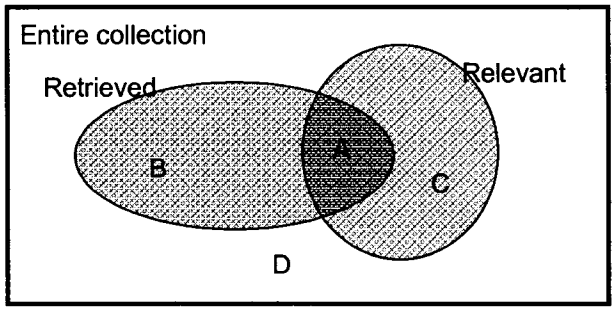
2.4 Performance Metrics

In this section we provide a brief overview of some of the performance metric used in evaluating information retrieval (IR) techniques and algorithms. We use these metrics in our performance evaluation to evaluate the quality of Freelib search results. The two main metrics are *Precision* and *Recall*. Precision (P) is defined as the fraction of the retrieved items that are relevant to the query and Recall (R) is the fraction of relevant items retrieved [4]. An optimal information retrieval algorithm would retrieve all relevant documents and only those, thus having 100% precision and 100% recall. Unfortunately, such a technique does not exist. IR algorithms face a compromise in this regard. One technique may retrieve more documents to increase recall. However, this typically

decreases the precision. Figure 2.2 shows graphical representation using a Venn diagram and calculations of precision and recall.

$$P = \frac{A}{A+B}$$

$$R = \frac{A}{A+C}$$



	Relevant	Irrelevant
Retrieved	A	B
Not retrieved	C	D

Figure 2.2: Precision (P) and recall (R): graphical representation and calculations.

CHAPTER III
FREELIB ARCHITECTURE

There exist many challenges when building a digital library suitable for the individual users to enable them publish and disseminate their content. The key challenges include the highly distributed nature of users as well as content, the lack of tools for maintaining high quality metadata, and inability of the individuals to secure the necessary software and hardware resources needed to organize and maintain the content. A framework is needed to address these challenges and build a digital library that enables the individual user to publish and search for content.

In this chapter we introduce our Freelib framework to address the above challenges. The Freelib framework builds a digital library on top of a peer-to-peer network. Our approach to sustainability is by utilizing resources contributed by the participating users of the underlying peer-to-peer network. Although each participant typically contributes little of disk storage and processing power; with the large number of nodes that characterize peer-to-peer system, massive amounts of resources can be accumulated. The use of a peer-to-peer system also addresses the issue of users and content being highly distributed. Each node in the system represents one user which might be joining from any geographic location. The Freelib framework provides its users with tools for publishing Dublin Core (DC) metadata [16]. The Framework also provides metadata-based search to its user. Metadata-based search enables the user to enter more accurate queries and hence enhance the results of Freelib search.

As stated earlier, the Freelib framework organizes participating nodes into a peer-to-peer overlay network. Links in our peer-to-peer overlay network can be categorized into two main categories, *support links* and *friend links*. The former are built based on the

Symphony protocol [43], which maintains the desirable small-world property. The latter are built based on simple access pattern analysis between peers. According to this categorization, our peer-to-peer network can be viewed as two separate overlay topologies as illustrated in Figure 3.1. The first is the *support network* which contains all the support links in the Freelib network. The other topology is the *access network*, which is the union of all the friend links in the Freelib network. It is worth mentioning that our initial Freelib design contained a third logical overlay topology, which we called the *migration ring*. The migration ring is a separate virtual ring. Our intention is to arrange the nodes such that each community occupies a small proximity on the migration ring. Our motivation for this is to facilitate discovery of communities. Once a node discovers a friend, it can discover other potential friends by probing the nodes closer to its friend on the migration ring. Once a node connects to enough friends that occupy close locations on the migration ring, it logically migrates to that area of the ring such that new nodes can discover their prospective communities. However, once we have started implementation, this three-layer model proved to be very complex. In addition, with continuous evolution of the access network, nodes whose users share similar interest get closer to each other on the access topology. Consequently, a newly joining node that connects to another similar node will be able to reach other relevant nodes through that node. Because of complexity of the topology and ability to achieve community discovery based on the access topology, we have decided to drop the migration ring and simplify our architecture. In the following sections, we discuss our current two-layer Freelib model in more details explaining the purpose of each topology is and how we build them. We follow that by describing the services Freelib offers to its users.

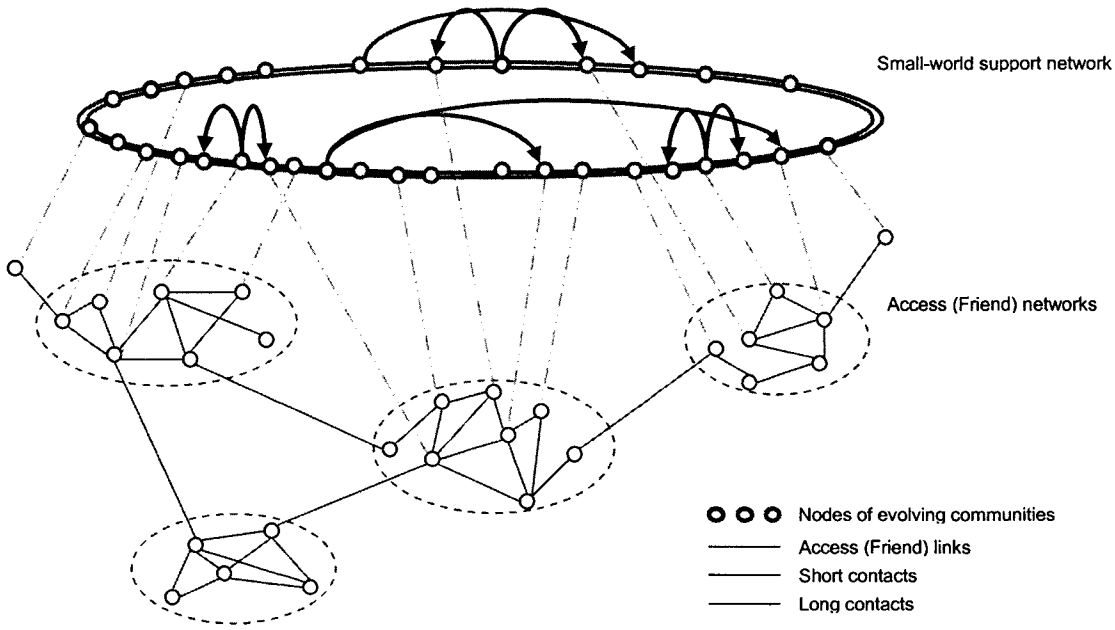


Figure 3.1: Freelib network architecture showing support network and access networks.

3.1 Small-world Support Network

The support network helps new nodes join the network and speeds up new nodes' discovery of its community. We discuss the use of the support network in discovery in chapter IV. We build the support network based on the Symphony protocol [43]. The support network topology consists of two types of links, *short contacts* and *long contacts*. Long contacts connect every node to distant nodes on that ring. Long contacts enable queries to reach to far locations on the ring. This is essential to achieve the small-world criterion. Short contacts, on the other hand, enable those queries to reach nodes in the immediate proximity. In the following subsections, we discuss theses types of contacts and the algorithms used to build them.

3.1.1 Short Contacts

The support network arranges the Freelib nodes on a virtual ring structure of unit length. When a node joins the Freelib network, its location on the support network is randomly chosen with uniform distribution over the ring. This is imposed by having a node select its ring location by sampling from a uniform distribution over the interval [0, 1) [43]. Hence, node locations are positive real numbers between 0.0 inclusive and 1.0 exclusive. Every node maintains a number of short contacts, which are links to the neighboring nodes on the ring. The minimum number of short contacts per node is one (link to the clockwise neighbor node on the ring). The more short contacts nodes establish, the more fault-tolerance is achieved. For example, to tolerate f consecutive node failures on the ring, every node must maintain f + 1 short contacts (e.g., each node maintains short contacts to next f + 1 nodes on the ring). If each node has more than one short contact, they can all be clockwise neighbors, anti-clockwise neighbors, or combination of these. Consider for example, the ring locations shown in Table 3.1. Assume that the table lists all the nodes on the support network ring with locations that range over the interval [0.5, 0.7]. The order of these nodes on the ring is: N5, N1, N4, N3, and N2. Table 3.2 shows several different choices for the list of short contacts for node N4 as an example. In order to maintain the ring topology, the minimum number of short contacts is one node.

3.1.2 Long Contacts

In addition to short contacts, Symphony [43] provides every node with a number of *long contacts*. Long contacts are links to distant nodes on the ring. The distance between two nodes on the ring is calculated by subtracting the ring locations of the two nodes and taking the absolute value. It ranges between 0 and 0.5. Having these long contacts is the

key to achieve the small-world property. The locations of the long contacts are chosen by sampling from the harmonic probability distribution (hence the name Symphony) shown in Equation 3.1.

Table 3.1: List of nodes with locations x between 0.5 and 0.7 inclusive

Node	Location on the ring
N_1	0.52
N_2	0.7
N_3	0.6
N_4	0.55
N_5	0.50

Table 3.2: Various choices of the list of short contacts for node N_4 based on ring locations in Table 3.1

Choice	List of Short Contacts
Clockwise neighbor only	N_3
Two node going clockwise	N_2, N_3
One node on each side	N_1, N_3
Two nodes on each side	N_1, N_2, N_3, N_5

$$p_n(x) = \begin{cases} \frac{1}{x \log(n)} & x \in [1/n, 1] \\ 0 & otherwise \end{cases} \qquad (Eq.\ 3.1)$$

n = # *nodes*

Using this probability distribution, the expected path length from any node to any node is shown in [43] to be $\Omega(\frac{\log^2 n}{k})$, where n is the number of nodes and k is the number of long contacts per node (usually 4 long contacts).

3.1.3 Summary of Algorithms for Building Support Network

For completeness of the discussion we overview the protocols used by Symphony [43] to build and maintain the support network. These protocols include join, leave, maintenance, failure recovery, and long contact. Here is a brief outline listing the steps of the key protocols in the support network.

Symphony Routing protocol is used to efficiently route any message to the closest node on the ring to a certain ring location:

1. The node sending the message selects from its contacts the node that takes the message closest to the given ring location.
2. This process is repeated by each node that receives the message until the message reaches its destination.

Join protocol is invoked when a node is joining the network. It consists of the following steps:

1. Choose a ring location by sampling from a uniform distribution over the interval [0, 1).
2. Send a join request to some existing node.
3. Request is forwarded using the *Symphony routing* protocol to the node that is closest to the chosen location. Closeness here is based on the real number distance

between the ring locations. That node inserts the joining node, updates its information, and sends response directly to the joining node.

4. Upon receiving the response the joining node updates its information and establishes short contacts.

5. Initiate the long contact protocol. The long contact protocol is explained below.

Leave protocol is invoked when a node is leaving the network. It simply cleans up and sends leave notifications to its contacts.

Maintenance protocol is invoked when nodes leave the network. The two nodes adjacent to the leaving node on the ring need to maintain the ring structure by establishing a short contact link between them. In addition, any node which has a long contact link to the leaving node triggers the long contact protocol explained below to establish a new long contact link to some other node.

Failure recovery protocol is to detect the failure of nodes and initiating the maintenance protocol. This protocol utilizes an application-level *ping* or *keep-alive* messages to detect when a node fails.

Long contacts protocol is invoked by a node to establish long contacts. This is done when a node joins the network, a node that is long contact to some other node leaves the network, or whenever the estimated number of nodes on the network changes. For every long contact to be established, the following is done:

1. Pick a ring location by sampling from the harmonic distribution shown in Equation 3.1. This is necessary to maintain the small-world property.

2. Send out a request to establish long contact to that location (using Symphony routing)

3. If the target node is already saturated (has already reached a predefined threshold for the number of incoming long contacts), start over.

Keep-alive protocol is the traditional application-level pinging protocol based on timers.

3.2 Access Network

The main purpose of building and evolving the access network is to enhance the search performance of Freelib. The main idea we employ in building the access network is *bringing nodes sharing common interest close to each other on the overlay topology.* Building the topology this way brings documents or items closer to nodes searching for them. So, instead of having to search the whole network, nodes need to search their immediate network proximity. This effectively reduces user waiting time for search results as relevant peers receive the search query immediately or after within few hops on the overlay topology. In addition, it saves network bandwidth as we can use smaller time-to-live for our peer-to-peer search messages. We do not have to search far on the network topology. Furthermore, this approach is expected to enhance recall and precision of the results. We evaluate the performance gains of Freelib in chapter VI.

In order to implement this approach, we need to address the following issues:

- How to characterize user interest and identify users sharing common interest, and
- How to build the overlay topology according to mutual interest between nodes.

In the following subsections, we introduce our approach to address these issues.

3.2.1 Characterizing User's Interest

Characterizing user interest and identifying users sharing common interest could be done in many different ways, some of which are explicit while others are implicit and transparent to the user. The easiest method to implement is explicit feedback, which requires users to give preference information about search results. Although the explicit feedback is accurate, users are usually too indolent to provide it. Implicit methods try to capture user preference based on analyzing user interaction with the system in a transparent way. For example, eye tracking technology and click-through data are utilized to infer user interest in many scenarios including web search [29, 32, 53]. An implicit method that does not directly rely on user actions but rather on analysis of content is topic segmentation [5]. This approach is based on measuring the similarity of the actual content contributed by different nodes and using it as an indication of similarity of their users' interest. First, documents are represented using a suitable data structure such as a keyword vector. Then, nodes are clustered based on the similarity of documents. Nodes that belong to the same cluster are then considered as having similar interest. This, however, is a complex approach that requires complex clustering algorithms and does not adapt when user interest changes between topics. We introduce a new method for characterizing user interest based on analyzing user access patterns. In our method, access of (downloading) a document or an item by a user after viewing its metadata and description is considered as an indication of user's interest in that document or item. This is analogous to using click-through data as implicit feedback. The higher the mutual access between a pair of nodes, the more similar is their interest. This method is implicit, requires no additional communications between nodes, and adapts with changes in user's

interest. Table 3.3 gives a comparison of our method for characterizing user interest to those other methods.

Table 3.3: Different ways to characterize user's interest

Method	Implicit / Explicit	Advantages	Disadvantages
User Feedback	Explicit	- Accurate	- Relies on User input; - Sometimes users are too indolent
Content Analysis (Topic segmentation based on content)	Implicit	- Could be applied before joining	- Needs complex clustering algorithms; - Does not adapt quickly with changes in user's interest
User Accesses Analysis (Implicit Feedback)	Implicit	- Requires no extra communication between nodes - Adapts to changes in user's interest	- Needs algorithms for speeding up the evolution of the network

3.2.2 The concept of Locality

In general, the concept of locality tries to predict future behavior of a system based on discovery of trend from the recent system history. This concept is applied and utilized in many fields and areas including computer architecture and business model design. It is often called the 80/20 rule. For example, by analyzing machine instruction frequencies, John Cocke in 1974 discovered that a small percentage of a computer's instructions accounted for most of the execution time [10]. Out of the whole set of machine instructions, compilers and programs utilized the simpler commands more often. This locality in instruction usage led to the evolution of processor designs like the RISC (Reduced Instruction Set Computing) architectures [10]. Another example use of the concept of locality is in the design of the memory hierarchy in modern computer systems

[24]. Spatial and temporal locality of memory addresses referenced by programs was utilized in building faster memory systems. This is achieved by introducing a smaller but very fast cache memory between the processor and the main memory. This cache is used to hold copies of the content of memory locations that are expected to be referenced in the near future based on locality. When the processor loads data from the memory system, the data is returned much faster if found in the cache. With good utilization of locality in prediction of memory references, considerable speed gain is achieved.

The concept of locality was also studied in the context of computer networks at various levels. For example, [40] studied locality with regard to end hosts (source and destination of network traffic). It was found that addresses of end hosts of traffic show a high degree of temporal locality. Temporal locality in that context was stated as: "If a pair of machines communicate, it is likely they will communicate again in the near future."[40]. Spatial locality, however, was inapplicable in that context. Another study that concentrated on the locality at finer level of granularity can be found in [45]. They studied locality at the level of communicating processes not just end hosts.

In our research, we apply the concept of locality in the context of peer accesses. Temporal locality in our case is very similar to temporal locality of end hosts described in [40]. In addition to utilizing the temporal locality, we try to enable spatial locality and make it applicable in our context. We evolve the overlay topology to reflect the mutual accesses between peers. This evolution process brings peers that have high mutual accesses, which indicates similar interest, closer to each other on the access network overlay topology. This closeness on the overlay topology represents spatial locality. Consequently, a node N_i accessing another node N_j becomes likely to access nodes close

to N_j on the overlay topology. Table 3.4 illustrates the concept of locality in the context of processor architectures and memory hierarchy. It also shows an analogy in the context of our context.

Table 3.4: Concept of locality and its analogy in Freelib

Locality	Utilized in
"small percentage of a computer's instructions accounted for most of the execution time", A result of a study by John Cocke @ IBM, 1974	RISC architectures
"Memory locations just accessed are likely to be accessed again in the near future" *"Memory locations nearby those just accessed are likely to be accessed next"*	Cache memory hierarchy
Analogy in Freelib:	
'Nodes just accessed are likely to be accessed again soon' *'Nodes with similar interest to those just accessed are likely to accessed next'*	Enhancing Freelib search by reflecting node interest in building the topology

3.2.3 Measuring Mutual Accesses between Nodes

In order to realize our approach, we need to measure mutual access between nodes, quantify it, and use it to build the overlay topology. Access in this context refers to downloading an item usually after viewing it in the result list of a search query. The simplest way to measure the mutual access between nodes is to count the number of accesses. Let $N = \{N_1, N_2, ..., N_n\}$ be the set of nodes in our Freelib network. Users' access counts can be represented by a function A whose domain is the Cartesian product $N \times N$ and whose range is the non-negative integers Z^+, that is $A: (N \times N) \rightarrow Z^+$. This function maps each ordered pair (N_i, N_j) to a non-negative integer a_{ij} that represent the

number of accesses made by node N_i to node N_j. This function could be represented as $n \times n$ matrix of non-negative integers, where n is the number of nodes. We refer to this matrix as the *access matrix*. The information about accesses that involve a certain node in our Freelib network occupies exactly one row and one column of the access matrix. For node N_i, this information occupies the i^{th} row and the i^{th} column of the access matrix. Each cell on the i^{th} row contains the number of accesses made by node N_i to the node associated with the corresponding column. We refer to these accesses as the *outgoing accesses* of N_i. Similarly, each cell on the i^{th} column contains the number of accesses made by the node associated with corresponding row to N_i. We refer to these as the *incoming accesses* to N_i. Maintenance of the access matrix is simple. Initially, all a_{ij} are initialized to zero. And every time an access is made by node N_i to node N_j, both a_{ij} and a_{ji} are incremented. Table 3.5 shows an example access matrix. In this access matrix, we can see, for example, that node N_4 has made 6 accesses to node N_1 and received 13 accesses from it.

Table 3.5: Access Matrix showing accesses for nodes N_1 to N_5

	N_1	N_2	N_3	N_4	N_5
N_1	0	17	10	13	10
N_2	7	0	9	11	15
N_3	3	11	0	14	8
N_4	6	7	14	0	11
N_5	8	5	11	6	0

Although n (the number of nodes) is very large in peer-to-peer networks, the access matrix is usually a sparse matrix as each node usually accesses a relatively small

subset of the whole set of nodes. Hence, a suitable sparse-matrix implementation could be employed to conserve space. In addition, as we shall explain shortly, we distribute the access matrix over the nodes in a way that minimizes the communication needed to perform our access pattern analysis. We next discuss use of the access matrix in peer ranking and building the topology.

3.2.4 Peer Ranking Based on Mutual Access

The purpose of peer ranking is to provide a ranked list of peers that serves as a list of candidates for building friend links. As described in the previous section, for a node N_i, mutual access with other nodes consists of two components, incoming accesses and outgoing accesses. We need to design a ranking function that has the following characteristics and features:

- It monotonically increases as number of accesses increase
- It takes into consideration both the incoming and outgoing components of the access data, and
- It allows us to assign different weights for each of the outgoing and incoming accesses.

The requirement that ranks monotonically increase with increasing accesses is clear since more accesses imply more mutual interest. Including incoming accesses in the calculation is needed to reflect mutual interest rather than one party's interest. Incoming accesses usually are assigned lower weight compared to the weight for outgoing accesses. Equation 3.2 gives our peer ranking function. It shows the calculation of the rank assigned to node N_j by node N_i.

$$R_i(N_j) = \alpha \times \frac{a_{ij}}{\sum_j a_{ij}} + (1-\alpha) \times \frac{a_{ji}}{\sum_j a_{ji}} \qquad (Eq.\ 3.2)$$

The parameter α in Equation 3.2 determines the weight for the outgoing accesses relative to incoming accesses. Possible values of α range from 0.0 to 1.0 where α =1.0 discards incoming accesses in the ranking calculation; and α = 0.0 discards outgoing accesses. We have performed some experiments and found that α should be typically set to some value around 0.8. This value ensure that the outcome of the ranking at each node is mainly derived by the local user accesses as it gives outgoing accesses higher weight relative to incoming accesses. This is very important as outgoing accesses are usually order of magnitudes smaller than incoming accesses. For example, if a community contains 100 peers accessing each other, the incoming accesses will be 100 times the outgoing accesses on average. In addition, this value of α does not completely ignore incoming accesses. They still receive some weight and they are reflected in the results of the ranking process as well.

The first partial term $\frac{a_{ij}}{\sum_j a_{ij}}$ represents the proportion of N_i outgoing accesses to node N_j. The second partial term $\frac{a_{ji}}{\sum_j a_{ji}}$ represents the proportion of N_i incoming accesses from node N_j. Notice that the value of each of these terms increase as the number of accesses (a_{ij} in the first and a_{ji} in the second) increase. Dividing by the total number of accesses in each of the two terms is not necessary. It, however, normalizes the ranks into values between 0.0 and 1.0 inclusive. In fact, R_i is a probability function with $0 \le R_i(N_j) \le 1$ and $\sum_j R_i(N_j) = 1$. The reader should also notice that all access data used

in Equation 3.2 above comes from exactly one row and one column in the access matrix. We utilize this fact in proposing a distributed implementation of the access matrix that eliminates communication between nodes regarding access pattern analysis and peer ranking.

Table 3.6: Ranks calculated based on the Access Matrix in Table 3.5 using $\alpha = 0.8$

	N_1	N_2	N_3	N_4	N_5
N_1	0	0.3303	0.185	0.258	0.2267
N_2	0.2183	0	0.2264	0.2445	0.3107
N_3	0.1121	0.2854	0	0.3747	0.2278
N_4	0.1854	0.1974	0.3584	0	0.2588
N_5	0.2588	0.2015	0.3297	0.21	0

Table 3.7: Ranked list at node N_4 calculated based on the Access Matrix in Table 3.5 using $\alpha = 0.8$

	N_4
N_3	0. 3584
N_5	0. 2588
N_2	0.1974
N_1	0. 1854

This method of ranking was inspired by the concept of locality discussed earlier in this chapter. The more mutual accesses with a node, the higher the rank it receives. After ranking peers, the ranking node sorts them in a non-descending order based on the

ranks such that nodes with highest ranks occupy the top of the list. This ranked list is then used to build the friend links for the node. As an example, Table 3.6 gives the ranks calculated based on the access matrix in Table 3.5 using $\alpha = 0.8$. Each row in this table gives the ranks calculated by the corresponding node. For example $R_1(N_2)$, which is the rank of node N_2 as calculated by node N_1, appears in the cell at the intersection of the first row and the second column. The value of $R_1(N_2)$ as given in the table is 0.3303. Table 3.7 shows the ranked list at node N_4. It lists the nodes that have mutual access with node N_4 in a non-decreasing order of ranks.

3.2.4.1 Frequency of Performing Peer Ranking

In order for the overlay topology to adapt and evolve as users' interest evolves, each node needs to perform peer ranking periodically. The question now is how often the peer ranking process should be performed by each node. This involves a tradeoff. On one hand, we need to perform ranking as often as possible to account for the most recent accesses. On the other hand, we do not want to overwhelm the nodes by executing the ranking procedure too often. In order to resolve this issue, we base the decision to trigger the peer ranking process on the actual number of new accesses that occurred since the most recent invocation of the peer ranking process. Each node maintains two values. The first value is Δ_{out}, the total number of outgoing accesses since the most recent peer ranking was performed. This value is incremented with every outgoing access. The other value is Δ_{in}, the total number of incoming accesses since the most recent peer ranking was performed. Similarly, this value is incremented with every incoming access. Every time an incoming or outgoing access is made, the node calculates Δ, the normalized total number of accesses since the most recent peer ranking was performed. The calculation of

Δ is shown in Equation 3.3. It combines Δ_{out} and Δ_{in} using the weighing factor α similar to the calculation of the peer ranks. If Δ is greater than a certain threshold $\Delta_{threshold}$, the peer ranking process is started. The value of this threshold controls how often the peer ranking process is performed by each node.

$$\Delta = \alpha \times \Delta_{out} + (1-\alpha)\Delta_{in} \qquad (Eq.\ 3.3)$$

By tuning $\Delta_{threshold}$, we control the frequency of ranking process invocations based on the number of accesses. Table 3.8 gives example values of α and $\Delta_{threshold}$ and the corresponding frequency of the peer ranking process.

Table 3.8: Example values of α and $\Delta_{threshold}$ and the corresponding frequency of the peer ranking process

α	$\Delta_{threshold}$	**Frequency of Peer Ranking**
0.5	1	Every 2 accesses (any combination of incoming and outgoing)
0.5	4	Every 8 accesses (any combination of incoming and outgoing)
1.0	1	Every outgoing access
0.8	1	Every 2 outgoing accesses, 1outgoing then 1 incoming, up to 4 incoming followed by 1 outgoing, or 5 incoming accesses

Although the above constraint enables us to have some control on the frequency of performing peer ranking, it still depends on the access rate. If the access rate is very high at some periods, the peer ranking process might still be invoked with high frequency and overload the machine. In order to have more control over the frequency of the peer ranking process invocations, we introduce another constraint. This constraint specifies a

certain minimum amount of time τ that must pass between consecutive invocations of the peer ranking process. The code in Listing 3.1 outlines the procedure for triggering the peer ranking process based on the number of new accesses Δ as well as a minimum interval between consecutive invocations of the ranking process τ.

In the next subsection, we study the effect of random accesses on out peer ranking calculations.

Listing 3.1: Triggering the peer ranking process based on the number of new accesses as well as a minimum interval

```
accessPerformed (Access a) {
        if (a is an outgoing access)
                Δout ++;
        else
                Δin ++;

        Δ = α* Δout+ (1 – α) * Δin;

        if ((Δ ≥ Δthreshold) && (time since last ranking ≥ τ))
                startRankingProcess ();
}
```

3.2.4.2 Effect of Non-friend Accesses on Peer Ranking

Non-friend accesses are those accesses that do not match the main user interest. These accesses can be categorized into two types of accesses. They can be legitimate accesses that represent temporary shift or the start of a permanent shift in user's interest into another topic or interest area. They can also result from human mistakes such as confusing the subject of a document or accidentally accessing the wrong document. Our peer ranking process could be enhanced by introducing certain techniques to speed up

discovery of permanent change in user's interest. We delay discussion of these techniques and our method for handling temporary change in user interest to later in this chapter. In this section, however, we discuss how our basic ranking process handles random accesses in general.

Table 3.9: Snapshot of an access matrix showing nodes N_1 to N_9

	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_9
N_1	0	8	10	5	6	0	0	0	0
N_2	14	0	10	8	9	1	0	2	0
N_3	3	11	0	6	12	0	0	0	0
N_4	7	4	8	0	9	0	0	0	0
N_5	6	9	10	8	0	0	0	0	0
N_6	0	0	0	0	0	0	10	8	12
N_7	0	0	0	0	0	11	0	13	14
N_8	0	0	0	0	0	9	0	0	15
N_9	0	0	0	0	0	11	15	10	0

By definition, the number of random accesses is relatively small compared to the total number of accesses. When a node performs peer ranking, random accesses typically result in smaller ranks assigned to the corresponding peers. As the user submits more searches and performs more accesses, random accesses will become more and more insignificant and the corresponding peers will typically occupy the tail of the ranked list. Consider, for example, the access matrix shown in Table 3.9. It shows the accesses for nodes N_1 to N_9. Most of the accesses of node N_2 involve nodes N_1 to N_5. However, it made one single access to node N_6 and two accesses to node N_8. These accesses look like random accesses as nodes N_6 and N_8 do not have high mutual access with N_2 or any of N_2

top ranked peers. Table 3.10 gives the ranks calculated by node N_2. It shows how nodes N_6 and N_8 receive very low ranks and, hence, occupy the tail of the ranked list. Notice also that node N_7 and node N_9 do not appear on the N_2' ranked list as they were not involved in any accesses with N_2.

Table 3.10: Ranked list at node N_2 showing ranks calculated based on the access matrix in Table 3.9 and using $\alpha = 0.8$. It shows nodes N_8 and N_6 receiving very low ranks and occupying tail of the ranked list

	N_2
N_1	0.304
N_3	0.250
N_5	0.219
N_4	0.170
N_8	0.036
N_6	0.018

3.2.5 Distribution of the Access Matrix over Nodes

As shown in the previous sections, ranking calculations performed at node N_i only uses the access data stored in the i^{th} row and the i^{th} column of the access matrix, each of which is a one dimensional array of n elements. One way to distribute the access matrix is to store at each node N_i only its associated row, which is the i^{th} row. This approach will, however, require each node N_i to request its incoming access information (the i^{th} column) from its peers when it is time to perform peer ranking. In our design, we introduce some replication of the access data to help avoid this extra communication. Each node maintains both its associated row and column. In other words, every node N_i maintains the i^{th} row and the i^{th} column of the access matrix. Although this duplicates the required

storage space, it eliminates any additional communication that could otherwise be needed to perform peer ranking.

In addition to limiting the access data maintained by each node to only one row and one column, we utilize sparse matrix implementation. As discussed earlier, each node usually accesses a small subset of the whole set of nodes and, hence, the arrays representing the i^{th} row and the i^{th} column are mostly sparse. Any suitable sparse matrix implementation could be used to conserve space. We use a variant of the coordinate storage scheme [62] in our implementation. We store the non-zero values and the corresponding rows or columns. Figure 3.2 shows the value array and column index array for storing outgoing accesses of node N_7 using the access matrix in Table 3.9. Similarly, Figure 3.3 shows the value array and the row index array for storing the incoming accesses to node N_7. In the actual implementation, however, we use unique node identifiers instead of the row/column indexes and store the access information as a hash table. The reason for this is that each individual Freelib node is identified by a unique identifier generated by the node according to [37] the very first time it joins the network.

Value:	11	13	14
Column Index:	6	8	9

Figure 3.2: Coordinate storage scheme for storing outgoing accesses at node N_7 using the access counts from Table 3.9

Value:	10	15
Row Index:	6	9

Figure 3.3: Coordinate storage scheme for storing incoming accesses at node N_7 using the access counts from Table 3.9

3.2.6 Building the Overlay Topology According to the Mutual Interest

The ranked list at each node contains the list of peers that have mutual access with the node. The peers with highest mutual access occupy the top of the ranked list. Each node chooses its friends from the top of its ranked list. By doing this, we introduce spatial locality into the topology based on mutual access. In other words, Freelib nodes with similar interest get closer to each other on the topology. Consequently, the traditional peer-to-peer forwarding of search messages takes these messages to the most relevant nodes in few steps on the overlay topology. The ranked list can be utilized to build the access topology in two different ways to realize this objective. The first, and the simplest, is the *Routing table* approach. The other is the *Active link* approach. In the following subsections, we explain these methods and discuss the advantages and disadvantages of using each of them.

3.2.6.1 Routing Table Approach

In this approach, the ranked list is used as a routing table for sending out and forwarding search queries. When a node is about to send out or forward a search request, it sends the search message to the first *R available* peers on its ranked list. Each node that receives a search request needs to acknowledge it to confirm its availability. The advantages of this approach are that it is dynamic and simple. But it does need extra communication messages for sending back the acknowledgements. Another issue with this approach is that a node does not know and does not have control on the number of nodes keeping it as a friend (we refer to these as *incoming friends* of the node as opposed to the *outgoing friends* the node selects from the top of its ranked list). This could cause as a *popular* node (one that every one is accessing) to be overwhelmed with too many requests.

3.2.6.2 Active link approach

In this approach, every node establishes active links to R peers from the top of the ranked list. These are full links that have a keep-alive mechanism such as pings. The number of friends per node, R, is typically 4 to 6. Every time the ranking process is performed, some friend links might need to be disconnected and some others might need to be established to reflect the most recent ranking results. This whole evolution process is transparent to the users of the system. When a node is saturated (i.e., the number of its incoming friends R_{in} exceeds a certain threshold), it simply rejects any new requests for establishing friend links to it.

Listing 3.2: Algorithm for establishing friend links

```
EstablishFriendLinks (RankedList rankedList) {
        create empty list newFriends
        for each peer in rankedList{
                if (peer is in currentFriends list)
                        add peer to newFriends;
                else{
                        try to establish friend link to peer;
                        if (result == OK)
                                add peer to newFriends;
                }
                if (number of peers in newFriends ≥ R)
                        break;
        }
        for each peer in currentFriends {
                if (number of peers in newFriends < R){
                        if (peer is not in newFriends)
                                add peer to newFriends;
                } else {
                        if (peer is not in newFriends)
                                dropFriend(peer);
                }
        }
        currentFriends = newFriends;
}
```

Like the routing table approach discussed in the previous section, this approach has some communication overhead due to establishing links and due to the pings. The ping (keep-alive) messages are needed in order to detect failure of friends. If every node establishes R friends, then we have $R \times n$ total friend links in a network of n nodes. Also the network will have total $c \times R \times n$ pings per unit time, where c is a constant that represents the number of ping messages per link per unit time. This is $\Omega(n)$ ping messages for the whole network. Listing 3.2 outlines the algorithm for establishing the friend links given the ranked list of peers as input. Listing 3.3 outlines the algorithm for processing a request for establishing a friend received by a node.

Listing 3.3: Processing requests to establish a friend

```
establishFriendRequest (PeerInfo peer) {
        if(numberOfIncomingFriends < R_in){
                add peer to incomingFriends;
                status = OK;
        }else{
                Status = SATURATED;
        }
        sendResponse(status);
}
```

Although this approach is more complex than the routing table approach presented in the previous section, we choose to use it in our design as it gives each node control on the number of incoming friends.

3.2.7 Community and Friends

The friend relationship between Freelib nodes can be represented by a binary relation F on the set of participating node. The binary relation F is given in Equation 3.4. An ordered pair $(N_i, N_j) \in F$ if and only if there is an friend link from node N_i to node N_j in

the access topology. The first coordinate N_i is the owner of the friend link and the second coordinate N_j is its friend contact.

$$F = \{(N_i, N_j) \mid \textit{there exist an friend link from node } N_i \textit{ to node } N_j\} \qquad \textit{Eq. 3.4}$$

Since friend links reflect mutual interest between nodes, the friend relation F evolves the participating nodes into communities of common interest. A community in our model is, however, a fuzzy concept. Membership in communities is not hardwired; rather, it evolves over time. From a node' point of view; the community is its immediate friends, their friends, and so on up to some time-to-live value h. To further explain this, we define the *h-step transitive closure* F^h to be a binary relation that includes all ordered pairs (N_i, N_j) where there is a path P on the access topology from node N_i to node N_j with $length(P) \leq h$. Equation 3.5 gives F^h according to this definition.

$$F^h = \{\forall (N_i, N_j) \exists \{(N_i, N_{k_1}), (N_{k_1}, N_{k_2}), \ldots, (N_{k_m}, N_j)\} \subseteq F \wedge m \leq h\} \qquad (Eq.\ 3.5)$$

The community of a node N_i at that point of time is the set of nodes that results from the projection on second coordinate of F_i^h. Equation 3.6 gives the community C_i of node N_i.

$$C_i = \pi_{N_j}(\{\forall (N_i, N_j) \in F^h\}) \qquad (Eq.\ 3.6)$$

The set of node in N_i' community changes over time as nodes perform ranking and reflect the results in the access topology. This feature enables communities to evolve based on users' interest.

3.2.8 Handling Changing User Interest

Sometimes users change their interest either temporarily or permanently. This is usually indicated by user starting to access different peers other than the ones accessed in the past. Using our basic ranking process discussed earlier in this chapter, it can take long time for new accesses to balance and beat the old ones. Consequently, it may take long time to get the user connected to the appropriate community that matches user's new interest. Consider, for example, the following scenario:

1. User has already joined some community.
2. User accessed friends around 100 times each.
3. User's interest is starting to shift to a different topic and she is starting to access different peers.

According to our ranking calculations presented earlier, new peers will not outrank the old ones until user accesses them more than 100 times each. This might take very long time. We need to speed up this process. The goal is to get the user connected to the appropriate community in fewer accesses. In the following subsections, we present two techniques that can be utilized to enhance our peer ranking process to deal with changing user interest. Their implementation and evaluation of these techniques is, however, left as a future work. These two techniques are *aging of accesses* and *access weights*. These techniques build on temporal locality of peer accesses, which we

discussed earlier in this chapter. The main idea is to enable peer ranks to be decided mainly based on more recent accesses.

In order to introduce ranking weights or implement aging, we need additional information about accesses. Most importantly, we need to know the time at which each access occurred. To realize this, each node maintains an access log instead of maintaining merely the access counts. Each entry in the access log represents one access. The information in each log entry includes the *unique identifier* of the peer involved in the access, the *identifier of the accessed item/document*, the *time of access*, and the *type of access* (i.e., whether it is incoming access or outgoing access). As discussed earlier, a node does not have to maintain this information for every peer in the network. Rather, it needs to account for only the peers with which it was involved in accesses, which is usually a very small subset of the whole set of nodes. The identifier of the item accessed is not used by the new ranking techniques presented in the following subsections. However, we choose to include it to enable further enhancements in the future. Figure 3.4 shows the structure of our access log entry.

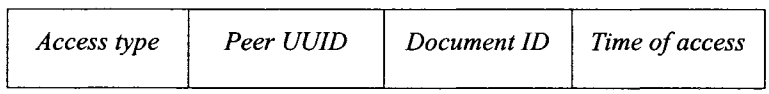


Figure 3.4: Structure of Freelib access log entry

3.2.8.1 Aging of Accesses

Aging of accesses is a technique that eliminates older accesses from the calculations of peer ranks. This allows peer ranks, and hence the order of peers on the ranked list, to be decided based on the more recent accesses. Consequently, the most recent user interest as

characterized by the ranks will be reflected in the access network topology. Aging of accesses is typically realized by utilizing the timestamp field in the access log entries. A time threshold is calculated based on a certain maximum age of accesses that should be included in peer ranking. For example, if we decide that we want to include accesses whose age is at most 5 minutes, the time threshold is calculated by subtracting 300,000 milliseconds (that is 5 minutes) from current system time. We then process the access log starting from the most recent log entry and scanning backward. We calculate the access counts for the individual nodes as well as the total number of accesses. We stop when we reach a log entry whose timestamp is earlier than the calculated time threshold. In order to make sure that we include adequate number of accesses, we augment this process with another constraint that specifies the minimum number of accesses that must be processed. The time threshold constraint is relaxed whenever its enforcement results in processing fewer accesses than the minimum required number of accesses. Listing 3.4 outlines the algorithm for enforcing aging of accesses in the peer ranking process.

3.2.8.2 Access Weights

Another way to handle changing user interest is by introducing access weights based on time of access. The main idea is to assign higher weights to more recent accesses to speed up the process of connecting the user to relevant nodes when user's interest changes. To calculate peer ranks using this method, each access is assigned a weight that is based on the time of access. And instead of merely counting the accesses, the weights for accesses that involve each peer are added up and normalized by dividing by the total weights (similar to our basic ranking). Our ranking formula based on access weight is given in Equation 3.8. It shows the calculation of the rank assigned to node N_j by node N_i. In this

Equation w_{ij} is the total outgoing access weights from node N_i to node N_j and w_{ji} is the total incoming access weights from N_j to N_i. Listing 3.5 outlines the algorithm for calculating access weights in the ranking process.

Listing 3.4: Calculating access counts using the aging technique

```
calculateAccessCounts (accessLog, maxAge, minNumberOfAccesses) {
        oldestTimestamp = getCurrentSystemTime() – maxAge;
        for (i= accessLog.size() - 1; i >= 0; i--){
                logEntry = accessLog.getLogEntryAt(i);
                if( (logEntry.getTimestamp() < oldestTimeStamp) &&
                                (i < accessLog.size() - minNumberOfAccesses))
                        break;
                if(logEntry.getType() == LogEntry.OUTGOING){
                        incrementOutgoingAccesses(logEntry.getPeer());
                        incrementTotalOutgoingAccesses();
                }else{
                        incrementIncomingAccesses(logEntry.getPeer());
                        incrementTotalIncomingAccesses();
                }
        }
}
```

Listing 3.5: Calculating access weights based on the access log

```
calculateAccessWeights (accessLog) {
        for (i= 0; i < accessLog.size(); i++){
                logEntry = accessLog.getLogEntryAt(i);
                weight = calculateWeight(logEntry.getTimeStamp());
                if(logEntry.getType() == LogEntry.OUTGOING){
                        incrementOutgoingAccesses(logEntry.getPeer(), weight);
                        incrementTotalOutgoingAccesses(weight);
                }else{
                        incrementIncomingAccesses(logEntry.getPeer(),weight);
                        incrementTotalIncomingAccesses(weight);
                }
        }
}
```

$$R_i(N_j) = \alpha \times \left(\frac{w_{ij}}{\sum_j w_{ij}} \right) + (1 - \alpha) \times \left(\frac{w_{ji}}{\sum_j w_{ji}} \right) \qquad (Eq.\ 3.8)$$

We need a suitable mathematical function to calculate the weight of the individual accesses. The required characteristic in any weight function to use is that the generated weights should decrease as accesses age. Giving higher weights to more recent accesses enables them to have more influence on the selection of friends. Consequently, fewer recent accesses will connect the user to relevant nodes when user interest changes. A variety of weight functions could be utilized to impose this behavior. In fact, the aging technique discussed in the previous section is equivalent to the weight function in Equation 3.9. It assigns a weight of 1.0 to accesses whose timestamp is more recent than a certain oldest timestamp and a weight of 0.0 otherwise. Figure 3.5 shows the corresponding graph for this weight function.

$$w = \begin{cases} 1.0 & timestamp \geq oldestTimestamp \\ 0.0 & otherwise \end{cases} \qquad (Eq.\ 3.9)$$

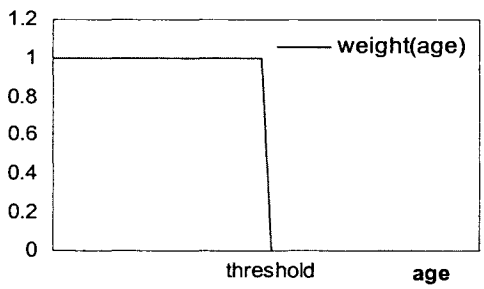


Figure 3.5: Weight function equivalent to the aging technique

Equations 3.10.a, 3.10.b, 3.10.c show some other possible weight functions. In these Equations, c is a scaling constant and M (used in Equation 3.10.c only) is the maximum relevant age. The corresponding graphs for these weight functions are shown in Figure 3.6.

$$w = \frac{1}{c \times \text{age}} \qquad (Eq.\ 3.10.a)$$

$$w = \frac{1}{\log_b(b + c \times \text{age})} \qquad (Eq.\ 3.10.b)$$

$$w = \begin{cases} 1 - \frac{age}{M} & age \le M \\ 0 & otherwise \end{cases} \qquad (Eq.\ 3.10.c)$$

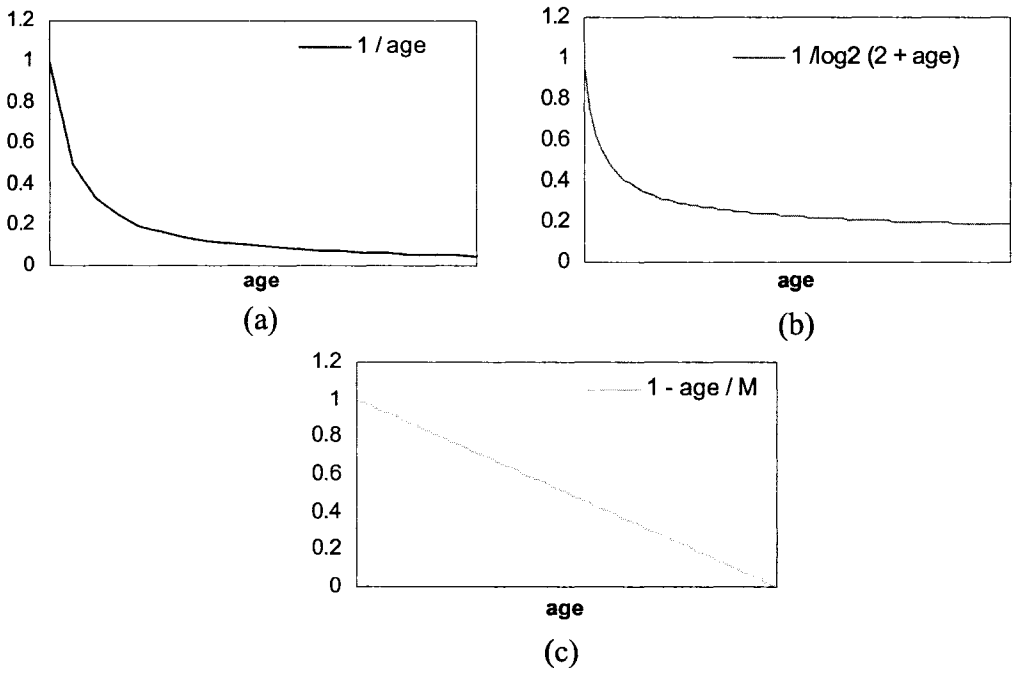


Figure 3.6: Graphs for weight functions in Equation 3.10

In the next subsection, we present a preprocessing technique to avoid irregularity of access weights that can result from long intervals of user inactivity.

3.2.8.3 Preprocessing Access Patterns

Intervals of user inactivity are common in access patterns. User activity, typically, consists of a series of sessions separated by intervals of inactivity as shown in Figure 3.7. Access patterns can be preprocessed to enhance the ranking weights and the aging technique.

Long intervals of inactivity could cause our access weights to be dominated by too few recent accesses. In other words, these long intervals of inactivity could result in large difference in weights assigned to accesses in different sessions. In the extreme case, some sessions might totally dominate the other sessions. Consider, for example, the pattern in Figure 3.8.a. If we perform the weighted ranking as described earlier, session s1 will dominate the other sessions because of the long inactive interval that separate this session from the others. This is not desirable, especially if s1 consists of few accesses. To avoid this anomaly, we need to normalize long inactive intervals, as shown in Figure 3.8.b, so that more recent sessions get relatively more weights over older ones but do not totally dominate them.

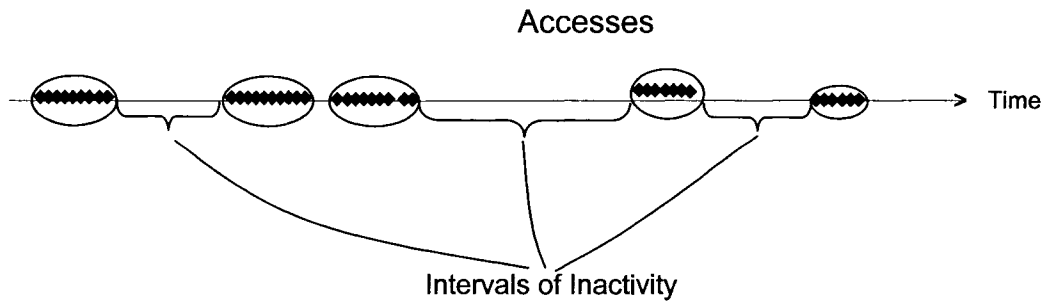


Figure 3.7: A typical access pattern consists of sessions separated by inactive intervals

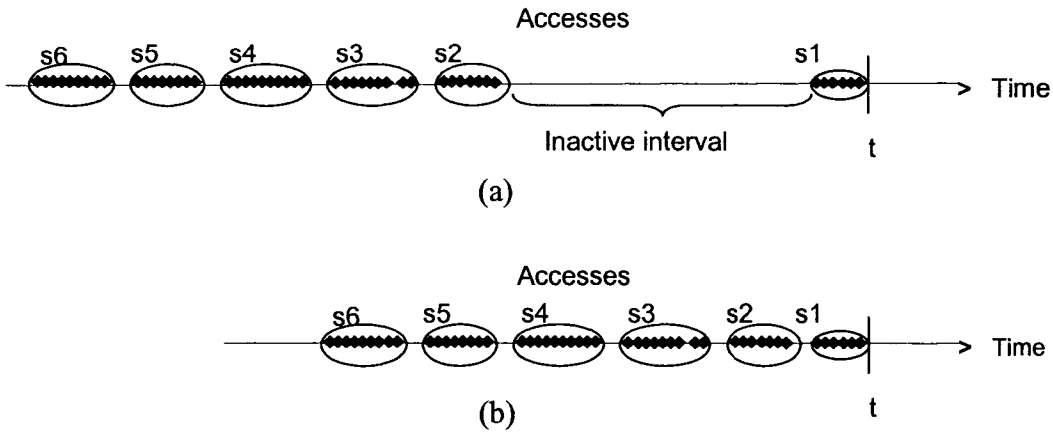


Figure 3.8: a) An access pattern; b) The same pattern after eliminating inactive intervals.

Listing 3.6: Calculating access weights along with normalization of inactive intervals

```
calculateAccessWeights (accessLog) {
        timestamp = getCurrentSystemTime();
        timeAdjustment = 0;
        for (i= accessLog.size() - 1; i ≥ 0; i--){
                logEntry = accessLog.getLogEntryAt(i);
                interval = timestamp - logEntry.getTimeStamp();
                if (interval > I_max ){
                        timeAdjustment += interval – I_max;
                }
                weight = calculateWeight(logEntry.getTimeStamp() + timeAdjustment );
                if(logEntry.getType() == LogEntry.OUTGOING){
                        incrementOutgoingAccesses(logEntry.getPeer(), weight);
                        incrementTotalOutgoingAccesses(weight);
                }else{
                        incrementIncomingAccesses(logEntry.getPeer(),weight);
                        incrementTotalIncomingAccesses(weight);
                }
                Timestamp = logEntry.getTimeStamp();
        }
}
```

Listing 3.6 outlines our algorithm for calculating the weights along with normalization of intervals of inactivity. We monitor the intervals between consecutive

accesses and normalize them if they are longer than a certain threshold I_{max}. As we scan through the access log, we maintain an adjustment value that represents the total amount by which we should adjust the timestamp of each access. This value is calculated by accumulating the total amount of time eliminated as we normalize inactive intervals.

3.2.9 Handling Temporary Change in User Interest

Temporary change in user interest is indicated by occasionally searching off-topic typically for short periods of time. An example could be a user that is searching in a main topic such as bio-informatics but occasionally searching for information about foreign countries when she is traveling. We have two issues related to this search pattern. First, we need a way to enable these temporary search requests to reach beyond the user's community in order to return the highest recall possible. Second, we need to make sure that accesses related to these temporary search queries do not affect the access topology and cause the user to get disconnected from her main community. The Freelib framework provides a global search mode that can be utilized to address the first issue. Global search uses the support network to forward the search queries and hence enables these search queries to reach beyond the user's community. When searching off topic, the user can utilize this global search mode. We discuss the global search mode in details in chapter V. To address the second issue, a configuration setting can be made available to the user to exclude these temporary accesses from the ranking calculations. Before starting to search off-topic, the user can turn off the ranking process as well as the access, which effectively prevents accesses related to temporary shift in user interest from changing the access topology.

3.2.10 Dual and Multiple Interest

Sometimes, a user might have interest in multiple topics. An example of this would be a researcher who is actively doing research in an interdisciplinary research area such as Bioinformatics. Such user will typically search equally in the two individual topics. In the case of the Bioinformatics, the individual areas are Biology and Information Technology or Computer Science. In general if a user interest spans n topics, her accesses will be distributed over these topics. Consequently, the friend list will contain peers from and be able to search those different communities. The aging and access weights techniques described earlier can also help in this scenario. When a user is actively searching in one of her topics of interest, these techniques will adjust the access topology and the user will get connected to more friends from the corresponding community. As the user starts to search in a different topic, the topology will be adjusted as well the user will get connected to more friends from the corresponding community. The implementation and evaluation of these techniques is, however, left as a future work.

3.3 Summary

In this chapter, we introduced our Freelib network architecture. We started by describing the Support network which is built based on the Symphony protocol [43]. We described the various protocols needed for building and maintaining the support network. We then introduced the Access network, which is evolved continuously based on simple access pattern analysis. We a method for capturing mutual interest between users and described a technique to rank peers at each node according to mutual user interest. Our method for capturing mutual user interest and peer ranking is an implicit and transparent one. No explicit feedback from the user is needed. We described how to build the access topology

based on the outcome of the peer ranking process and showed how our method evolves the participating users into communities of common interest. Finally we described methods and techniques for handling changing user interest. In the next chapter, we discuss how to generate system-wide unique node identities and introduce our discovery protocols.

CHAPTER IV
NODE IDENTITY AND DISCOVERY PROTOCOLS

In this chapter we introduce the 'Node Identity' and 'Discovery' protocols. In our Freelib framework, generating unique node identities is essential. Node identifiers are utilized by our access logging, ranking algorithms, and discovery protocols. Non-unique node identifiers would result in inaccurate ranking calculations and incorrect discovery results.

Taking into consideration the fact that nodes in a peer-to-peer system are autonomous and users have the freedom to join and leave the network at any time and as often as they wish, we need to provide a mechanism for *discovery of peers*. The discovery mechanism is needed for example when a node joins back after leaving as the information it has about other peers might be outdated. During the time the node is disconnected, nodes might leave the network and join back using different IP addresses and/or port numbers. In addition, nodes get a new ring location on our support network virtual ring every time they join. This emphasizes the need for a discovery protocol that enables nodes to rediscover their access contacts and friends when they rejoin. The one question that a discovery protocol answers is: given the unique identifier of a node, what is its current information (e.g., IP, port, ring location). Hence, in our context, 'discovery' refers to the finding of a node's current information given its unique identifier (UUID).

4.1 Node Identity

As discussed earlier, we need a system-wide unique node identification mechanism. This mechanism should be a distributed one as we do not want to introduce centralized components into our model. Nodes should not change their identifiers; rather, identifiers must be persistent. This is critical for the accuracy of the ranking calculation and for the

discovery protocol. This persistence requirement disqualifies session identifiers such as IP and port pairs and ring locations. These are not persistent; rather, they might change every time the user joins the network.

To fulfill these requirements, we implemented version 4 of the Universally Unique Identifier (UUID) internet draft [37]. UUID is also documented as part of ISO standard for Remote Procedure calls [27]. It is documented more recently in an International Telecommunication Union standard [28] and IETF published an equivalent RFC [36]. UUIDs are 16 bytes (128 bits) identifiers. The canonical representation of UUIDs is 32 hexadecimal digits separated by hyphens as shown in the following XML element:

<uuid>8b27b39a-a907-4103-bcef-b3e375bc355d</uuid>

4.2 Discovery

In a system with no centralized components, discovery is not simple; rather, it is a complicated task. The simplest approach to discovery is to flood the access network with the discovery request. This is, however, bandwidth inefficient. We introduce two new discovery protocols that are much more efficient in terms of bandwidth usage. These are the *DHT discovery* and the *Link discovery* methods. The following subsections cover a comparison of discovery protocols. We start by discussing the Flooding method and then we follow it by presenting our new discovery methods. Finally, we compare the bandwidth usage for these discovery protocols.

4.2.1 Discovery by Flooding

This approach uses flooding. Discovery requests are forwarded by every node to all its contacts and friends up to certain TTL. Although, this approach is straightforward to implement, it is very bandwidth inefficient as number of messages grows exponentially with increasing TTL. If we have TTL of h hops and c contacts per node on average, the number of messages per discovery request would be $\Omega(c^h)$. For example, for TTL = 7 and 10 contacts per node, the number of messages per discovery request is more than 10 millions. Although that many messages are used, the request might not reach every node in the network because of the TTL. This could cause the discovery protocol to fail to locate information about existing nodes.

4.2.2 DHT Discovery

In the DHT approach, we build and maintain a distributed hash table (DHT) for storing discovery information of Freelib nodes. According to our definition of discovery, we want to enable nodes in our network to discover current information about other nodes including their IP addresses, port numbers, and ring locations. This information represents the values to be stored in our discovery hash table. We use the unique identifiers (UUIDs) of the nodes as keys. Hence, an entry in our discovery hash table corresponds to one node and maps from that node' UUID to its current information. The basic operations supported by the hash table are *insert(UUID, nodeInfo)* and *retrieve(UUID)*. The former inserts a hash table entry with the specified *UUID* and *nodeInfo* as its key and value respectively in the distributed hash table. The latter retrieves the *nodeInfo* record associated with the specified *UUID* or *NULL* if no such key can be found in the distributed hash table. In order to perform these two operations

efficiently, we need to be able to efficiently route the corresponding messages to the node responsible for storing the corresponding DHT entry. We use Symphony [43] for this purpose.

The Symphony protocol, which we use in building the support network, routes messages efficiently to nodes occupying or close to specific ring locations. If we can compute the ring location of the node responsible for storing a discovery entry, then we can use the Symphony protocol to route the discovery message efficiently to that node. Our computation of the ring location that corresponds to a certain UUID can be performed by any node and, hence, no centralized repository is needed. Our way to achieve this is by equipping Freelib nodes with a universal one-way hash function h under which every *UUID* is mapped to one and only one ring location $loc = h\ (UUID)$. Under this mapping, the ring location that corresponds to a certain UUID always remains the same and does not change over time. At any moment, the node that manages ring location *loc* on the support network is responsible for maintaining the corresponding hash table entry (*UUID*, *NodeInfo*). We remind the reader that the node that manages a ring location is the closest node, on the support network virtual ring, to that location going anticlockwise. Every discovery request, whether insertion or retrieval, involving a *UUID* is routed using Symphony to the node at support network ring location $loc = h\ (UUID)$. Table 4.1 shows six nodes constituting an example Symphony ring. It shows for each node its ring locations, UUID, *h(UUID)*, and the node responsible for storing the discovery entry. For example h(UUID1) = 0.82, which makes node N5 responsible for storing the discovery entry for N1. N5 is the closest node (anticlockwise) to location 08.2

Table 4.1: Nodes in a Symphony ring and the corresponding nodes responsible for storing their discovery entries

Node	Ring Location	UUID	h(UUID)	Node storing discovery entry
N_1	0.120	$UUID_1$	0.82	N_5
N_2	0.251	$UUID_2$	0.452	N_3
N_3	0.372	$UUID_3$	0.001	N_6
N_4	0.714	$UUID_4$	0.741	N_4
N_5	0.800	$UUID_5$	0.130	N_1
N_6	0.928	$UUID_6$	0.401	N_3

Listing 4.1 and 4.2 outline the algorithms for sending out the requests for hash table entry insertion and retrieval respectively. These algorithms use Symphony routing protocol from [43] to route the discovery insertion and retrieval messages (the call to *SendSymphonyMessage* in Listing 4.1 and 4.2). We discussed the Symphony routing protocol in Chapter III. When a node receives a request for inserting a discovery hash table entry, it inserts it into a local hash table. And when a node receives a request to retrieve a discovery hash table entry, it returns the node information if the entry exists in its local hash table or *Unknown-Node* if the entry does not exist. The absence of a discovery entry could happen because of one or more of the following reasons. First, the node may have left the network and deleted its discovery entry upon leaving. Second, the entry might have been lost due to the failure of the node storing it. And third, the UUID in the discovery request might be invalid (invalid is this context means that UUID has never been used by any node). We shall introduce solutions to handle node failure (the

second case above) through replication in the following section. For nodes that already left the network and for discovery requests that involve invalid UUID, it is sufficient to return *Unknown-Node* response. Listing 4.3 and 4.4 outline the processing of the insertion and retrieval requests respectively by the receiving nodes.

Listing 4.1: Sending out request for insertion of discovery information

```
insertDiscoveryInfo (String uuid_str, String nodeInfo) {
        double loc = h (uuid_str);
        SendSymphonyMessage (loc, new DiscoveryInsertMessage(uuid_str, nodeInfo));
}
```

Listing 4.2: Sending out request for retrieval of discovery information

```
retrieveDiscoveryInfo (String uuid_str) {
        double loc = h(uuid_str);
        SendSymphonyMessage(loc, new DiscoveryRetrievalMessage(uuid_str));
}
```

Listing 4.3: Processing of discovery insertion request

```
processDiscoveryInsert (String uuid_str, String nodeInfo) {
        localDiscoveryHashtable.put(uuid_str, nodeInfo);
}
```

Listing 4.4: Processing of discovery retrieval request

```
processDiscoveryRetrieve (String uuid_str) {
        // if entry exists, returns the associated node info, otherwise null is returned
        return localDiscoveryHashtable.get(uuid_str);
}
```

These algorithms are straightforward. One critical feature, however, is the consistency in generating the ring locations that correspond to node UUIDs. All nodes

must use the same exact hash function whenever the ring location that corresponds to a certain UUID needs to be generated. And therefore, the ring location that corresponds to certain UUID stays the same regardless of the node that is generating it. Consequently, all discovery requests that involve a certain UUID are routed to the same ring location.

Using this method, a discovery request is routed to its destination using $\Omega(\frac{\log^2 n}{k})$ messages, where n is the number of nodes and k is the number of long contacts per node. This is an order of magnitude less than the number of messages used in the flooding method. However, there is a small overhead at the time of joining and leaving the network. When a node joins the Freelib network, it needs to: 1) initiate the insertion protocol for inserting an entry for itself using its own UUID as the key, which costs at most $\frac{\log^2 n}{k}$ messages; and 2) claim its share of the distributed hash table from the neighboring node on the support network, which costs 2 messages (a request and its response). When a node leaves, it needs to: 1) delete its entry from the distributed hash table, which costs $\frac{\log^2 n}{k}$ messages at most; and 2) transfer its portion of the distributed hash table to its neighboring node on the support network, which costs 1 or 2 messages depending on whether an acknowledgement is sent back. The total overhead per node is $2 \times \frac{\log^2 n}{k} + 4$ messages, which is still $\Omega(\frac{\log^2 n}{k})$.

Fault-tolerance of DHT discovery method can be enhanced in many ways. One way to enhance it is to have every node periodically sends its discovery entry for insertion. This recovers the discovery entry if it has been lost due to failure of the node storing it. We call this *DHT with Repetition*. A second approach to enhancing fault-

tolerance of the DHT Discovery protocol is introducing replication. This is the subject of the next subsection.

4.2.2.1 Replication of Discovery Information

Failure of Freelib nodes can adversely affect our DHT discovery protocol. If a node fails without having the chance to transfer the discovery entries it maintains to its neighboring node on the ring, these discovery entries are lost from the discovery distributed hash table. Consequently, any discovery requests asking for these entries will not be fulfilled. This can cause inconvenience and disruption of the discovery service to nodes especially if the rate of failure is high. We use replication to alleviate this issue and enhance fault-tolerance of the DHT discovery protocol. Replication is a popular technique to significantly enhance availability. If the probability of a node to fail is ***P***, and nodes fail independent from each other, then the probability of ***r*** nodes to fail is P^r . For example, if P = 0.1 and r = 3 replicas, the chance of all three replicas to fail is 0.001, which is order of magnitudes smaller.

We introduce a new replication scheme that works at the granularity of individual discovery entries, instead of replicating at the level of nodes. Instead of having pairs of nodes whose discovery information is exact replica of each others, our replication scheme selects r nodes for storing each discovery entry, where r is the number of replicas. In order for all nodes to be able to locate the replicas for a certain UUID, we use a universal hash function h to determine the ring locations of the r replicas as follows. The first ring location is calculated based on the UUID of the discovery entry using h as before, $loc = h(UUID)$. The remaining r-1 ring locations are chosen such that the whole set of r replication ring locations are equidistant on the ring. For example, for r = 2, each

discovery entry *(UUID, NodeInfo)* is stored at two ring locations that are half way across the ring from each other. These ring locations are $loc_1 = h$ *(UUID) and* $loc_2 = (loc_1 + 0.5) \bmod 1.0$. Table 4.2 shows the same nodes in our previous example, from Table 4.1, with two replicas storing the discovery entry for each node.

To further illustrate what we mean by granularity level of the individual discovery entry, which we mentioned earlier; let's consider nodes N_2 and N_6 in Table 4.2. Each of these nodes has N_3 as a first replica. However, the second replica for each of them is different. For node N_2, the second replica is N_6 whereas for node N_6, the second replica is N_5. When two nodes are selected as two replicas for a certain UUID, it does not mean that they are exact replica of each other; rather, it means that they are replicas for storing this specific discovery entry.

Our discovery replicas are all the same level and there is no notion of primary or secondary replicas. In addition, there is no internal synchronization between replicas. Rather, discovery insertion and deletion requests are sent to all the replicas. And when a node leaves, it requests deletion of its discovery entry from all its replicas. Discovery retrieval requests can be handled in several different ways, however. One option is to send discovery retrieval requests to all replicas simultaneously. Another option is to try one replica at a time. In the later case, randomization could be utilized to achieve load balancing. Alternatively, requests could be sent to the closest replica on the support ring as this reduces the number of forwarding steps required to reach the destination as discussed in [43]. In this case, if a discovery fails, the next closest replica on the discovery ring is tried. Listing 4.5 outlines the procedure for sending out discovery

insertion requests to all of three replicas. Similarly, Listing 4.6 shows the procedure for sending out discovery retrieval requests using all three replicas.

Table 4.2: Nodes in a Symphony ring and the corresponding nodes responsible for storing their discovery entries, each entry is stored at two replicas.

Node	Ring Location	UUID	h(UUID)	Node storing discovery entry
N_1	0.120	$UUID_1$	0.82	N_5, N_2
N_2	0.251	$UUID_2$	0.452	N_3, N_6
N_3	0.372	$UUID_3$	0.001	N_6, N_3
N_4	0.714	$UUID_4$	0.741	N_4, N_1
N_5	0.800	$UUID_5$	0.130	N_1, N_4
N_6	0.928	$UUID_6$	0.401	N_3, N_5

Listing 4.5: Insertion of discovery information at all three replicas

```
insertDiscoveryInfo (String uuid_str, String nodeInfo) {
        double loc1 = h (uuid_str), loc2 =( loc1 + 0.33) mod 1, loc3 = (loc1 + 0.67) mod 1;
        SendSymphonyMessage (loc1, new DiscoveryInsertMessage(uuid_str, nodeInfo));
        SendSymphonyMessage (loc2, new DiscoveryInsertMessage(uuid_str, nodeInfo));
        SendSymphonyMessage (loc3, new DiscoveryInsertMessage(uuid_str, nodeInfo));
}
```

Listing 4.6: Retrieval of discovery information from all of three replicas simultaneously

```
retrieveDiscoveryInfo (String uuid_str) {
        double loc1 = h (uuid_str), loc2 =( loc1 + 0.33) mod 1, loc3 = (loc1 + 0.67) mod 1;
        SendSymphonyMessage(loc1, new DiscoveryRetrievalMessage(uuid_str));
        SendSymphonyMessage(loc2, new DiscoveryRetrievalMessage(uuid_str));
        SendSymphonyMessage(loc3,new DiscoveryRetrievalMessage(uuid_str));
}
```

4.2.3 Link Discovery

Link discovery is an alternate, new approach to node discovery in Freelib. In this approach, each joining node establishes and maintains a discovery link to a target node whose ring location is specified by hashing the joining node unique identifier. In other words, instead of building a discovery DHT, every node establishes one discovery link to the manager of the ring location *loc = h (UUID)*, where *h* is the universal hash function, *UUID* is the unique identifier of the node. We refer to the node that establishes the discovery link as the *owner* of the discovery link and to the target node of the discovery link of as its *discovery contact*. The discovery contact of a node can be located anywhere on the support ring. This depends on the ring location generated by the hash function h and the density of nodes on the ring. When a node joins, it establishes a link to its discovery contact. Symphony routing is used when a request to establish a discovery link is sent. Whenever a node is leaving the network, it notifies it discovery contact and disconnects its discovery link. The discovery link is a live link that requires keep-alive mechanism such as periodical pings. These pings are light-weight messages that are sent directly to the destination. No peer-to-peer routing is involved at all when ping messages are sent. In order to discover information about a node N_i, a discovery request is routed using Symphony to N_i's discovery contact, which costs $\Omega(\frac{\log^2 n}{k})$ messages.

Maintenance of the discovery link is needed in two cases: 1) when the discovery contact fails; and 2) when a new node joins using a ring location that is between the location of the discovery contact and the actual ring location associated with the discovery link (which is *loc = h (UUID)*, where *UUID* is the unique identifier of the owner of the discovery link). In the first case, the predecessor of the discovery contact

becomes the new discovery contact for the owner of the discovery link. In the second case, the newly joining node becomes the new discovery contact. To further illustrate this, consider the following example scenario:

1. Initially we have node ***A*** at ring location 0.7 manages the interval [0.7, 0.8), node ***X*** at ring location 0.61 manages the interval [0.61, 0.7)
2. Node ***B*** is joining say at ring location 0.1, ***B***'s UUID hashes under h to the value of 0.75, ***B*** establishes discovery contact to ***A***
3. Now, node ***A*** is going down, node ***X***'s interval becomes [0.61, 0.8), ***X*** is node ***B***'s discovery contact, and ***B*** establishes discovery link to ***X***
4. Now, node ***Y*** is joining at location 0.74, ***X***'s interval is now [0.61, 0.74), ***Y***'s interval is [0.74, 0.8), ***Y*** is now the discovery contact for ***B***, and the link needs to be reestablished
5. Assume now that another node ***Z*** is joining at ring location 0.76, ***Y***'s interval becomes [0.74, 0.76) and it is still the discovery contact for ***B***, nothing needs to be changed.

In both cases above, failure of discovery contact and new node joining between discovery contact and the actual ring location associated with the discovery link, the discovery link needs to be reestablished, which costs $\Omega(\frac{\log^2 n}{k})$ messages. This cost can be significantly reduced. Adding little additional information to the ping messages can help in the case of failure of the discovery contact. The discovery contact needs to send information on its ring predecessor in the ping messages. When a node detects failure of its discovery contact, it knows the predecessor and it immediately establishes discovery

link to it. This effectively reduces the cost to $\Omega(1)$. In the case of a new node joining and becoming the new discovery contact (like node Y in step 4 of the example scenario above) the original discovery contact just notifies the discovery link owner, which establishes a new discovery link to the joining node. This also reduces the cost in this case to $\Omega(1)$. A separate issue is the failure of the owner of a discovery link. In this case, the discovery contact detects failure of the discovery link owner and releases its resources accordingly. No maintenance is required in this case.

4.2.4 Comparison of Discovery Algorithms

Table 4.3 shows a comparison the cost of the various discovery protocols associated with events such as joins, leaves, node failure, and sending out discovery requests. All the costs are per node cost. The repeated insertion and pings are performed periodically at a certain rate per unit time. Other costs are incurred when certain events, such as node failures and sending out discovery requests, happen. The table shows the costs of these operations for the Flooding, Plain DHT, and DHT with Repetitions, Replicated DHT, Replicated DHT with Repetitions, and Link Discovery algorithms. As mentioned earlier, the with-repetitions versions refer to variants of the discovery protocols in which the discovery insertion requests are sent periodically to avoid losing the discovery information when nodes fail.

From the time complexity shown Table 4.3, we can see that Flooding has a high cost associated with discovery requests. The number of messages grows exponentially with h (the max TTL). Plain DHT discovery and Replicated DHT significantly reduce the cost of discovery but suffer loss of discovery information as nodes fail. The versions with repetition enhance this by making that loss temporary as lost discovery entries are

recovered at the time of the next insertion requests by the nodes whose discovery entries were lost. In the last row in Table 4.3, we show the effect of node failure. The reader should not that for replicated variants, losing a discovery entry happens when all the replicas storing that entry fail. The chance for this to happen gets significantly smaller as the number of replicas increases. Link discovery beats the other DHT discovery protocols in two cases, which are *leave* and *periodical pings*. When the discovery contact of a node fails, the discovery link needs to be reestablished. This is equivalent to re-insertion of the discovery entry in *DHT with repetition*. Based on this cost comparison, we chose to implement and use the *Link discovery* protocol in our implementation of Freelib universal client. It avoids permanent loss of discovery information. In addition, it replaces unnecessary re-insertion requests, which cost $\Omega(\frac{\log^2 n}{k})$ each with simple ping messages.

Table 4.3: Comparison of the various discovery algorithms

	Flooding	**DHT Discovery**	**DHT with repetitions**	**Replicated DHT**	**Replicated DHT with repetitions**	**Link Discovery**
Join / Establish	0	$\Omega(\frac{\log^2 n}{k})$	$\Omega(\frac{\log^2 n}{k})$	$\Omega(\frac{\log^2 n}{k})$	$\Omega(\frac{\log^2 n}{k})$	$\Omega(\frac{\log^2 n}{k})$
Discovery	$\Omega(c^h)$	$\Omega(\frac{\log^2 n}{k})$	$\Omega(\frac{\log^2 n}{k})$	$\Omega(\frac{\log^2 n}{k})$	$\Omega(\frac{\log^2 n}{k})$	$\Omega(\frac{\log^2 n}{k})$
Leave	0	$\Omega(1)$	$\Omega(1)$	$\Omega(1)$	$\Omega(1)$	$\Omega(1)$
Periodical Insertion / Pings	N/A	N/A	$\Omega(\frac{\log^2 n}{k})$	N/A	$\Omega(\frac{\log^2 n}{k})$	$\Omega(1)$
Failure of *r* Node **(for non-replicated protocols, $r = 1$)**	0	1 discovery entry lost	1 discovery entry lost temporarily until next insert	1 discovery entry lost	1 discovery entry lost temporarily until next insert	1 discovery entry lost temporarily until reestablish

4.3 Summary

We started this chapter by discussing the need for a method for generating system-wide unique node identities and a specific solution: UUIDs. We then discussed the need for a discovery protocol that enables nodes to find their previously discovered friends. The straight-forward flooding technique is very inefficient and consumes considerable network bandwidth. We then introduced two new discovery protocols that are built on top of the Symphony support network. The first is DHT discovery, which builds a distributed hash table that maps node identities to node information. We discussed variants of the DHT discovery protocol that utilize repetition of the insertion operation and other variants that use replication of the discovery information. The second major discovery protocol we presented is Link discovery in which each joining node establishes and maintains a discovery link to a target node whose ring location is specified by hashing the joining node unique identifier. In this protocol, all nodes use the same hash function, which guarantee that all discovery requests for a certain node identifier are routed to the same ring location and reach the correct target node. We compare the complexity associated with all the various discovery protocols discussed in this chapter and explain our decision of implementing Link discovery in our implementation of the Freelib client based on complexity of operations and target environment of Freelib.

CHAPTER V
IMPLEMENTATION

In this chapter we present our design and implementation of the Freelib peer-to-peer client, which is implemented using the Java programming language. We start with presenting the client design. We, then, briefly describe the various modules that constitute the whole implementation.

Our implementation of the Freelib client is highly modular. It consists of modules each of which implements a specific functionality. Each module exposes one or more public interfaces to the rest of the modules and effectively hides the details of the implementation. This approach produced an easily maintainable source code as changes to the implementation of one module do not affect the other modules. We start by describing the various modules in our client design. We then describe the process flow of the various user and non-user triggered processes. This will help us demonstrate the possible interactions between the various modules. We conclude this chapter with a discussion of the various services that the current implementation provides to the Freelib user.

5.1 Implementation Modules

The block diagram in Figure 5.1 shows our Freelib client design. In the following subsections we present the various implementation modules. We group these modules according to the common functionality or service they provide and present each module group in a separate subsection. For example, the modules that provide services directly to the user are in the User Modules group. Other groups include the Networks Manager modules that are responsible for implementing the various network and topology

protocols; the Messaging modules that are responsible for communication between peers; the Log and History modules responsible for maintaining the access log, search history and the Registry, which is a central repository for peer info; and the Collection Manager module that is responsible for indexing and storing metadata and full-text.

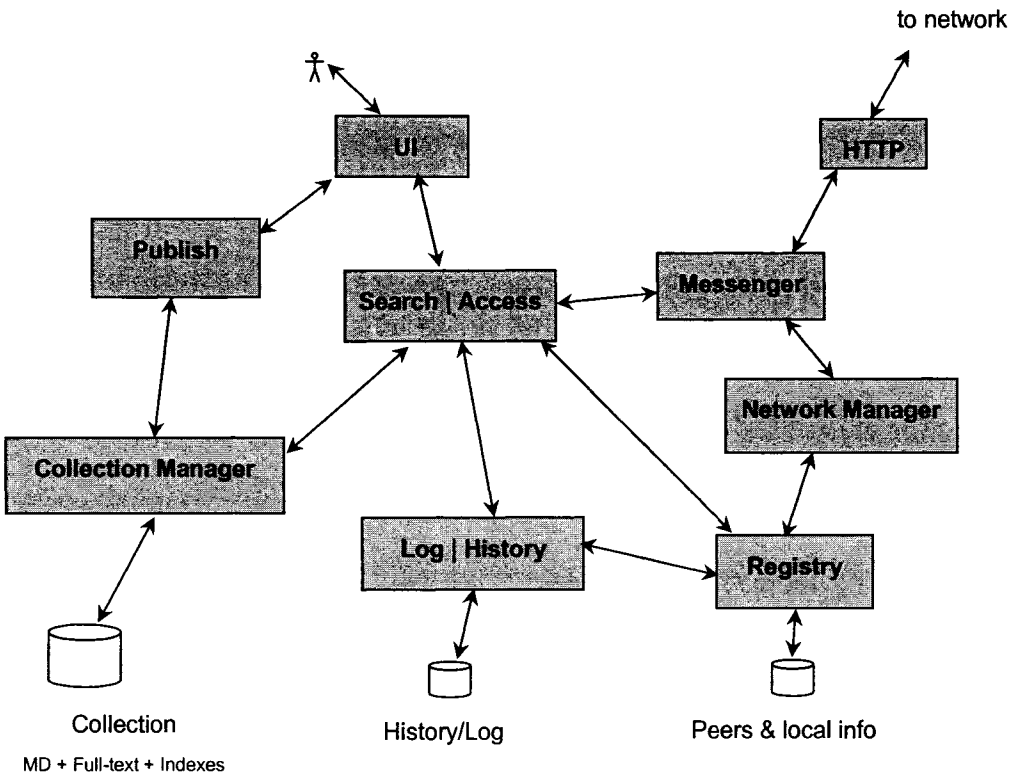


Figure 5.1: Block diagram of Freelib client

5.1.1 User Modules

The user service modules are four modules that interact with the user and implement services consumed directly by the user. The first and immediately apparent to the user is the UI module. This module provides the main graphical user interface through which the user interacts with the system and invokes various services including publishing, searching, and retrieval/access. We borrowed some of Freelib graphical user interface

from Kepler [33, 42]. The new features added in Freelib include multiple tabs for search results and some of the buttons on the main interface to support Freelib-specific functionality. . Figure 5.2 shows the main user interface of the Freelib client. At the top, there is a set of buttons that enables the user to invoke two of the main services, namely publishing and search. The other buttons in this set are the Settings button, which enables the user to configure the Freelib client; the Connect/Disconnect button, which enables the user to connect to and disconnect from the peer-to-peer network; and the Help button which displays help information to the user. To the right of the main buttons, an icon is displayed. This icon is bright when the client is connected to the network and is grayed when it is disconnected.

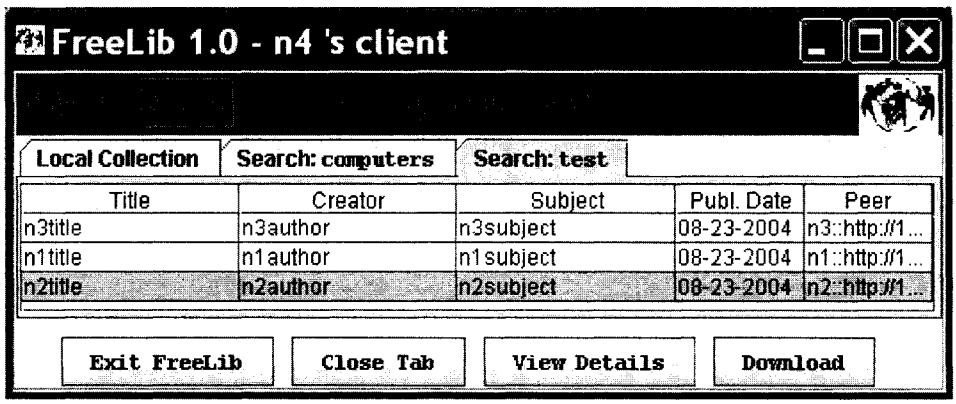


Figure 5.2: Freelib main user interface

The middle area of the main user interface consists of overlay tabs for displaying the local collection and the results for the ongoing user search queries. The first tab is dedicated for displaying the items available in the local collection. For each ongoing search query an additional tab is created to display the results for the query. The information inside each tab is presented in tabular form with each row presenting various

metadata for one item/document from the result set (or from the local collection in case of the first tab).

In addition to the main buttons at the top, another set of buttons is available at the bottom of the main user interface. These buttons include the *Exit Freelib* button for exiting the Freelib client; the *Close Tab* button for closing the current tab, the *View Details* button for viewing the detailed metadata for the selected item/document, and the *Download* button initiating download of the selected item. The local collection tab is not closable. Closing a search tab effectively releases any resources allocated for the result set associated with it. The user is always prompted when the *Exit Freelib* or *Close Tab* is clicked before the corresponding action is taken.

Figure 5.3 shows the publishing interface that is displayed when the user clicks the main publish button. It allows the user to provide the various metadata fields and choose a document to publish. Figure 5.4 shows the simple search interface, which is displayed when the user clicks the main search button. It enables the user to enter her search query. Freelib supports various types of queries including exact search, proximity search, fuzzy search, etc. We shall discuss the various query types later in this chapter. Figure 5.5 shows the configuration interface. It allows the user to edit configuration information such as port number to use and proxy information if user is behind proxy. It also allows the user to edit her user profile including name, email address, and description of local collection.

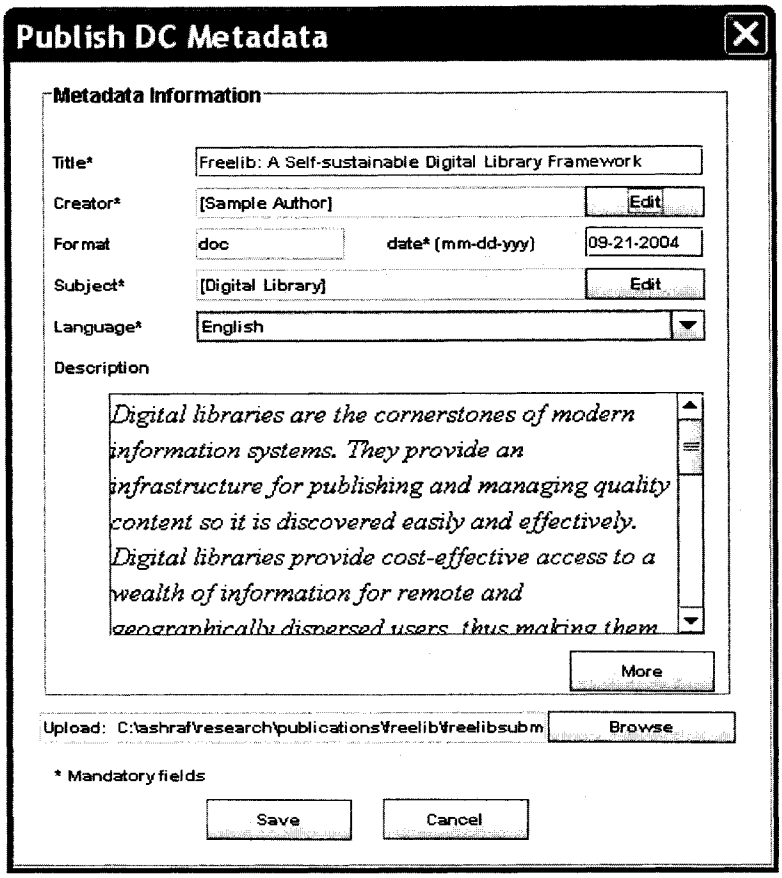


Figure 5.3: Publishing tool: It enables the user to provide metadata and choose a document for publishing

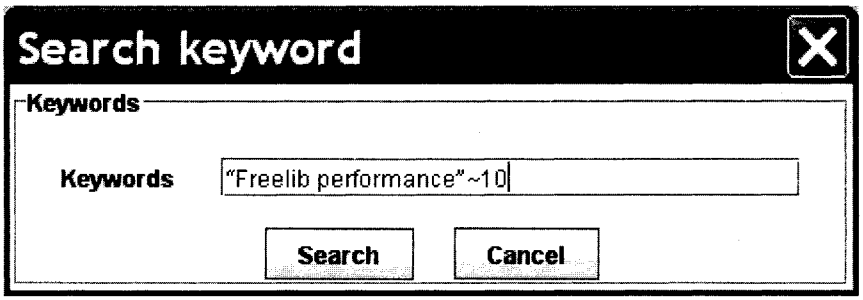


Figure 5.4: Freelib Search Interface showing a proximity search query entered. Items returned as results for this query must have the two words Freelib and performance within 10 words from each other

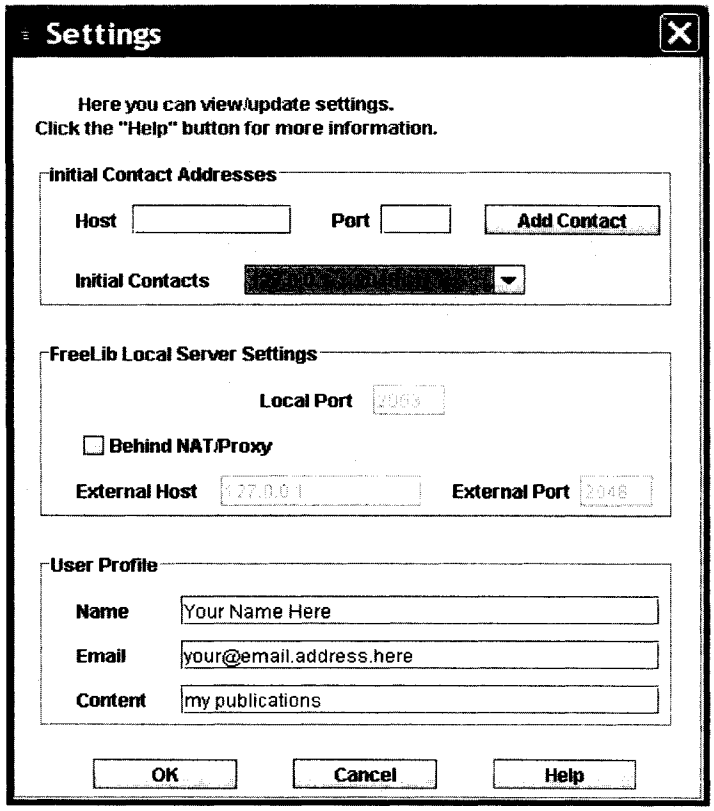


Figure 5.5: Configuration tool: It enables the user to provide configuration information and edit user profile

In addition to the UI module, there are three other modules related to user services. These are the Publish, Search, and Access modules. The publish module is invoked when the user submits metadata for publishing. Currently, this module only publishes to the local collection. In the future, replication to other nodes might be utilized for better availability of content. The Search module implements the search protocol. It is invoked by the UI module when the user enters a search query. This module sends out the search queries, collects results, and forwards them back to the UI module. By default, this module searches the local collection as well. In the future, this feature, however, should be made configurable by the user. The Access module is a simple module that is invoked by the UI module when the user clicks the download button. It sends out access requests

and handles the responses by saving the downloaded content into a specific subdirectory inside the directory structure of the installed Freelib client.

5.1.2 Messaging Modules

There are two modules that implement the messaging framework. These are the Messenger module and the HTTP module. The messaging framework is utilized by the other modules that communicate with other Freelib nodes, e.g., the Search and Access modules. The Messenger module is invoked by the other module to deliver Freelib messages to other nodes. The Messenger module supports both synchronous and asynchronous communication modes. In the synchronous mode, the sender thread blocks waiting for a response message. In the asynchronous mode, the sender thread returns immediately and the Messenger forwards response back to the module as it arrive. The HTTP module encapsulates Freelib messages in HTTP requests and sends them out to their destination. The HTTP module on the other end extracts the Freelib message from the incoming HTTP request and forwards it to the Messenger module for delivery to the appropriate module. The Messenger module exposes two public interfaces. These interfaces are the *MessageHandler* interface and the *MessengerInterface*. The messenger interface is implemented by the main Messenger class. It specifies the methods that other modules can call to send out the various message types supported. The message handler interface must be implemented by any module that wishes to receive Freelib messages. The single method declared by this interface is called by the Messenger module to deliver incoming messages to the implementing modules. The class diagram in Figure 5.6 shows the major classes and interfaces in the Messenger module. In addition, it shows the search module, as an example, implementing the message handler interface.

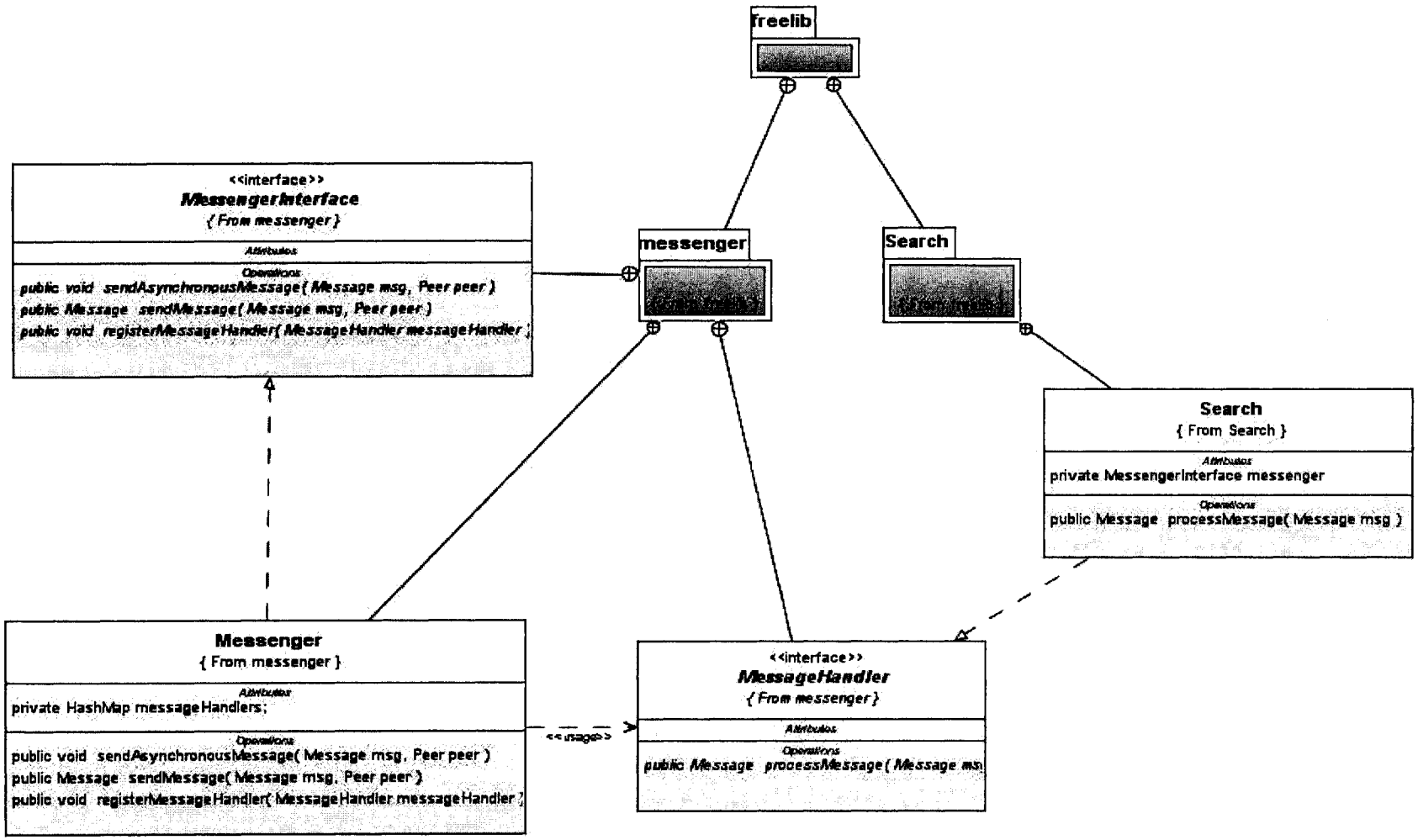


Figure 5.6: Class diagram showing the major classes in the Messenger module and the Search module implementing the MessageHandler interface

5.1.3 Network Modules

The network modules are the modules responsible for implementing and maintain the network protocols. They are four sub-modules that are included in the main Network Manager module. The first of these four modules is the Ring Manager module. The ring manager implements the join and leave protocols. In addition, it is responsible for maintaining the support network short contacts. The second module is the Long Contact module, which is responsible for establishing and maintaining the support network long contacts. The third module is the Friend Manager module. The Friend manager is responsible for establishing and maintaining friends based on the ranked list of peers. The fourth and final module is the Maintenance Manager module which is a helper module that implements some functionality used by the other network modules for maintaining their corresponding portion of the network architecture. Each of these four modules implements the message handler interface and registers with the messenger module as message handler similar to the search module as shown in Figure 5.6. This enables these modules to use the messaging framework to send and receive messages.

5.1.4 Log, History, and the Registry

These modules implement internal functionality that does not directly involve any network communication or use of the messaging framework. The Log Manager module is responsible for logging the incoming and outgoing access in which the local user is involved. This is the access log that is utilized by the peer ranking process to produce the ranked list of peer, which is in turn used for establishing the access contacts. The log manager is invoked by the access module whenever an incoming or outgoing access

occurs. Every time the peer ranking process starts, it scans through the access log, calculates peer ranks, and produces the ranked list.

The History module is not implemented in the current version of the Freelib client. It is intended for maintaining a search history of the local user. The search history can be utilized in many ways. It can be used to infer useful information about the user interest. In addition, it can be made available to the user as necessary for selectively repeating previous searches.

The Registry module is a local repository of local and peer information. It contains the Freelib client configuration information and user profile information. In addition, it contains the most recent information about peer nodes.

5.1.5 The Collection Manager

The Collection manger is responsible for maintaining the local collection. In the current implementation, the local collection contains the metadata, documents, and indexes for the content published by the local user. In future, content replicated from other nodes could be maintained by the collection manager in a separate collection. The collection manager implements a DAO (Data Access Object) design patterns to allow different implementations to be plugged into the client. The DAO pattern exposes a public interface which declares the public methods used for data access. All the details of the implementation are hidden from the user of the module. The supported collection types in the current Freelib implementation are an XML File System-based collection and a Lucene-indexed collection. The former uses XML files to save the metadata. The later uses the Apache Lucene [41] open source software for indexing the metadata. Other

collection types can be plugged as necessary, e.g., a Database collection using JDBC/ODBC.

5.2 Freelib Client Process Flow

Figure 5.1 shows the block diagram of our Freelib client. The various modules are represented by rectangles that show the module names. The lines connecting the modules represent possible interactions between pairs of modules. The possible flow of information between the various modules includes:

- **Search**, which consists of the following interactions:
    1. User clicks a Search button on the GUI and provides the search keywords,
    2. UI module creates and displays a tab for displaying the results and invokes the Search module passing the search query,
    3. Search module invokes the Log/History module to insert an entry in the search history.
    4. Search module invokes the Collection Manager to search the local collection, The collection manager responds by returning the matching results
    5. Search module queries the Registry module to get the list of peers to forward the search query. The Registry module responds by providing the list of peers. The list of peers depends on the search mode. We discuss search modes later in this chapter.
    6. The Search module invokes the Messenger module to forward the search query to the peers on the list provided by the registry module in

an asynchronous. When results from other peers arrive, the Messenger module invokes certain call-back methods in the Search module passing the results.

7. The Messenger module invokes the HTTP module to encode the messages in http requests and send them to the peers on the list. Responses from other peers are decoded by the HTTP module and the messages are presented to the Messenger module.
8. The Search module passes the results to the UI module as they arrive from the local collection as well as from other peers.

- **Access**, which consists of the following interactions:
    1. User selects an item from the results tab of an ongoing search query and clicks the Download button on the main UI,
    2. The UI module invokes the Access module passing the available information about the item of interest including its identifier and information about the peer that has that item.
    3. The Access module invokes the Log module to log the access.
    4. The Access module invokes the Messenger module to send out the download request message. The reply message is then passed back to the access module, which saves the downloaded file and invokes the UI module to notify the user.
    5. The Messenger module invokes the HTTP module to encode the message in an http request and send it to the peer. Response from the

peer is decoded by the HTTP module and the message is forwarded to the Messenger module.

- **Publish**, which consists of the following interactions:
    1. User clicks the Publish button on the main UI, fills in the metadata and attaches the full-text document.
    2. UI module invokes the Publish module to publish the item.
    3. The Publish module passes the metadata and the full-text to the Collection Manager module, which indexes the metadata and stores the full-text in the local collection
- **Incoming Search**, which consists of the following interactions:
    1. HTTP module receives a search request from another peer, decodes it and passes the search message to the Messenger module.
    2. The Messenger module passes the message payload to the Search module, which extracts the search query
    3. The Search module invokes the Collection Manager module to search the local collection, and constructs and passes a reply message to the messenger module to send to the original peer that owns the search query through the HTTP module as usual.
    4. The Search module invokes the Registry module to get the list of peer to which the search message should be forwarded, The Registry module responds with the list.
    5. The Search module invokes the Messenger module to forward the search message to the list of peers.

6. The Messenger module invokes the HTTP module to encode the message into HTTP requests and send them out.

- **Incoming Access**, which consists of the following interactions:
    1. HTTP module receives an access request from another peer, decodes it and passes the access message to the Messenger module.
    2. The Messenger module passes the payload of the message to the Access module, which extract the access request.
    3. The Access module invokes the Log module to log the access.
    4. The Access module invokes the Collection Manager to get the file name. The Access module, then, constructs a response message, attaches the full-text to the message, and invokes the Messenger module to send the message
    5. The Messenger module invokes the HTTP module to send the message.
- **Ranking**, which consists of the following interactions:
    1. Log Manager module detects that Δ, the number of accesses since the last ranking process was performed, exceeds $\Delta_{threshold}$ and verifies that enough time has passed since the last ranking process was performed. Then it triggers the ranking process.
    2. After the ranking process finishes, the Log Manager passes the new ranked list to the Network Manager to update the access topology accordingly.

3. The Friend Manager updates the topology and invokes the Registry module to store the new Friend list.

5.3 Freelib Services

Every node in the Freelib framework maintains a small collection. The size of the individual collection is usually up to few hundreds to few thousands of items; although there is no limit imposed by the framework and in fact, it can grow as needed. Freelib creates a huge virtual collection out of these smaller individual collections. The direct services Freelib offers its users are *publish*, *search* and *retrieval/access*. The framework is, however, extensible and additional services such as collaboration services can be added in the future. We discuss these services in the following subsections.

5.3.1 Publish

Publishing is the process of making items available in the collection and ready to be discovered and used by the various services. The publishing process consists of: ingesting metadata and documents, indexing them, and generating identifiers for items. In the simplest case, items are published to the local collection. However, content availability can very much be enhanced in face of node failures and disconnections by introducing content replication. Replication of content is, however beyond the scope of this work. We discuss it with other possible ways to extending this work in chapter VI.

5.3.2 Search

Search is the process of locating relevant items. Search in Freelib is mainly keyword search based on the metadata. Full-text search, which extends the search to look inside document text, is not currently supported by Freelib. It is, however, a candidate as a

short-term future extension. The search protocol in our system is based on peer-to-peer. Requests are forwarded from node to node up to some TTL.

Based on our categorization of links in our overlay topology, Freelib nodes can utilize two search modes, which are *community search* and *global search*. Community search is suitable for nodes that are already connected to enough friends. In this search mode, requests are forwarded using the friend links only. In this search mode, a smaller TTL is usually enough to get most of the relevant results as friends are close to each other on the topology. Global search, on the other hand, is suitable for newcomers, which are nodes joining for the first time and which does not having enough friends. On every hop, a request is forwarded to all friends, short, and long contacts. This mode usually uses a large TTL to enable the new nodes to discover their friends more quickly.

Various types of search queries are supported by the current Freelib client implementation. Table 5.1 gives the different search types, an example, and the meaning of each. We support these search types based on the Apache Lucene indexing engine [41] that we are utilizing in our Collection Manager module.

5.3.3 Access

Access is a simple service in Freelib. Once the user submits a search request and results start to arrive, they are displayed to the user. At this time, the user can view the metadata of any item on the result list. The user may select any of these items to download, which we refer to as access. Access is done by direct communication with the peer having the item. It does not involve any forwarding on the peer-to-peer topology.

Table 5.1: The different types of search queries in Freelib

Search Type	An Example Query	Meaning / Matching Results
Simple keyword search	*Freelib*	matches items that have the word Freelib
Proximity Search	*"Freelib performance"~10*	matches items that have the two words within 10 words from each other
Similarity Search (Using Levenshtein/Edit distance)	*roam~*	returns items that have words similar to roam e.g., roam, room, foam
Wildcards	*te?t* *archiv**	matches words like *test, text* matches words like *archive, archives, archival, etc.*
Exact phrase	*"digital library"*	matches items that have exact phrase
Field-Specific	*title: performance*	matches items that have the word in the title field
Composite Queries	*digital AND library*	matches items that have both words

5.4 Summary

In this chapter, we presented our implementation of the Freelib framework. We presented our client architecture, which consists of many modules each implementing certain functionality. We then described the process flow for the key user- as well as non-user-triggered processes in the system. These include the processes associated with events like the search, publish, access, incoming search, incoming access, and the peer ranking process. Finally, we present the services that the Freelib framework provides to its users including the search, publish and access services. We discuss the various supported search types including simple keyword search, exact phrase search, proximity search, similarity search, wildcard search, field specific search, and composite search queries.

CHAPTER VI
FREELIB PERFORMANCE EVALUATION

In this chapter, we present the evaluation of Freelib. The main objectives of the test and evaluation of Freelib are:

1. To perform rigorous testing of our implementation and ensure the correctness of all the various functions especially the network protocol, and
2. To measure the performance gain Freelib achieves over other comparable systems.

We realized the first main objective by creating a testbed using the Freelib client that we implemented. The testbed consisted of 20 machines each running Linux and having 2 GB of memory. We ran multiple nodes on every machine. These nodes used the implementation of the Freelib client. This involved using the operating system network protocols as well as the physical network. We shall discuss this in more details later in this chapter. In order to achieve our second objective, we built an event-based simulator. This simulator enabled us to simulate large networks of thousands of nodes and various community sizes. We studied different performance aspects including the quality of the results (mainly recall of the search results), the bandwidth usage, and the response time. We shall discuss this in greater details later in this chapter.

This chapter is organized as follows. We start by discussing our methodology for evaluating Freelib. Then, we present the testbed we built and present the outcome of one set of experiments that we performed using that testbed. We then show how the experiments justified the need for building a simulator. We, then, present our event-based simulator discussing the simulator design and the results of the experiments we

conducted using the simulator. We conclude the chapter with a summary of our evaluation results.

6.1 Evaluation Methodology

This section describes our approach to evaluating Freelib. We start by outlining our user model. Then we describe the reference system that we use for comparison. We conclude the section by discussing our evaluation metrics.

6.1.1 Modeling Users and Documents

In order to perform effective evaluation of Freelib, we need to resolve two main issues. First, we need to model different interest areas (communities) and how local collections are to be constructed to reflect the interest area(s) of the individual user. We use a set of keywords W to model documents, interest areas, and queries. Each interest area is represented by a set of keywords $T_i \subseteq W$. In general, some keywords might appear in more than one interest area. These are ambiguous keywords that have different meanings in different contexts. For example, the word *network* could be used to refer to computer networks as well as network of people (social network). When each node is created, the main interest area, say i, for the user is randomly chosen. Then, the local collection is built by adding documents. Each document is represented by a set of keywords $d_j \subseteq T_i$ (a subset of the set of keywords that represent the corresponding interest area). The number of documents in each collection is determined randomly. In our model, we have two types of nodes. These are *regular* nodes and *hub* nodes. Hub nodes are typically a small percentage of nodes that have larger collections and more resources especially in terms of network bandwidth. Regular nodes, on the other hand, are less powerful in terms of

resources and have smaller collections. They represent the majority of the nodes in the network.

The second issue is to accurately model and automate the user behavior. The user activity typically consists of sessions in which the user repeatedly submits a search, examines the results, and accesses some items. Hence, the main user actions that we need to model are *submitting search queries* and *accessing/downloading* selected items from search results. We model arrival of search queries as a Poisson process with λ search queries per minute. For example, assigning λ a value of 0.5 means the user submits a search every two minutes on average. For each search query, some items are randomly selected for access/download. Studies [22, 29, 63] have shown that users give attention to the first few (typically 10) top ranked results. Our software simulates this behavior by giving higher chance for downloading items that are closer to the top of the ranked list of results.

6.1.2 Performance Comparison

In our evaluation, we mainly want to measure the performance gain we can achieve using Freelib's concept of evolving communities that share a common interest. We can characterize this performance gain by comparing Freelib to a peer-to-peer system in which links between nodes are ad-hoc (do not reflect that mutual interest). We chose to compare our system with Symphony [43] for the following reason. Although overlay topology links in Symphony do not reflect any kind of mutual interest between nodes, the Symphony protocol is still better than an ad-hoc P2P network as it maintains a small-world network. We discussed the small-world property in chapter II. This property makes a peer-to-peer network to have a very small diameter relative the number of participating

nodes. This enables search queries to reach more nodes faster, and hence, Symphony has an advantage over ad-hoc systems.

We use the Symphony protocol in our support network overlay topology to maintain the small-world property but also we use it to the first few searches when a node has just joined and does not yet have friend links. Thus, the support network helps new joining nodes to discover their communities. In addition, it supports our node discovery protocols. After a few search and accesses, a new node starts to build friend links and will switch to using community search mode (that is, use the friend links to forward the search queries). In this mode, the nodes utilize the access overlay topology for submitting and forwarding search queries. So, although we use the Symphony protocol, its use in search is temporary and represents a short transient period for each new node immediately after joining the network. In fact, we could build the Freelib system using the access network only. In this case, the difference will be a slightly longer transient period for new nodes until they discover their communities. In our experiments, whenever we need to measure the performance for Symphony alone, we do so by turning off the access network protocol and having the search queries forwarded using the support network only.

6.1.3 Measurements

The measurements of interest to us involve three separate aspects of the Freelib search protocol, which directly affects the user experience of such a system. These include: 1) *quality of results*, which characterizes user satisfaction in the relevance of results to the corresponding query; 2) *query response time*, which characterizes the amount of time the user waits for results after submitting a query; and 3) *bandwidth usage*, which

characterizes how heavy the search protocol is in terms of bandwidth consumption. We discuss these measurements in greater detail in the following subsections.

6.1.3.1 Quality of Results

We measure quality of results using the two measurements which were defined in Chapter II. These are *recall* and *precision* of the results. For a certain search query, recall represents the percentage of relevant results the system returns. For example, if 100 documents are relevant to a user query and the results returned by the system include 60 of these, the recall is 60/100 or 60%. Precision, on the other hand, represents the percentage of relevant results relative to the size of the result set. To continue with our example above, if the size of the result set is 300 items, the precision is 60/300 or 20%. These quality measures are directly related to the user experience. The optimal case is when the user gets all and only the relevant items to her query. However, this is usually not achievable in part because of keyword ambiguity. In addition, peer-to-peer search queries might not reach all relevant nodes due to bandwidth limitations.

6.1.3.2 Query Response Time

Response time represents the time from submitting the query until the results arrive. This affects the user experience as it determines the amount of time the user waits for results. In traditional information retrieval systems where results for each query arrive as one response, response time can be measured accurately. In our context, however, partial results are accumulated and presented to the user as they arrive from different nodes. This makes it harder to calculate the response time. In order to resolve this issue, we introduce a recall parameter into the definition of response time. Given a certain required minimum

recall level r, we define response time for a search query to be the time from submitting the query until the moment at which the recall of the result set reaches the threshold r. For example, if r is 60%, the response time is the time from submitting the query until the recall of the result set reaches 60%. In order to further simplify our measurements of response time, we measure the number of hops on the overlay topology (overlay distance) instead of real clock time as they usually are directly proportional. As results arrive, we monitor the recall as well as the overlay distance of the node sending the results. The overlay distance is the number of hops on the overlay topology that separates the node that sends the results from the node that receives the results (the owner of the query). Consider the example results shown in Table 6.1. This table shows the results for one query. Each row represents partial results arriving from one of the peers. The information in each row includes the peer sending the results, the size of the results from the peer, the size of the results accumulated, the overlay distance of the peer relative to the owner of the query, the accumulated recall as the results arrive, and the accumulated precision (rounded to one decimal digit). The accumulated recall is calculated assuming the number of items relevant to the query is 10. Table 6.2 shows the response time in terms of overlay distance for different recall thresholds. For example, for recall threshold of r = *60%*, response time is 2 (hops) and for r = *80%*, response time is 3.

6.1.3.3 Bandwidth Usage

Bandwidth usage is an important performance measurement especially for large-scale systems such as peer-to-peer networks. These systems normally use message forwarding from node to node to propagate search queries to participating nodes. Nodes in peer-to-peer systems are autonomous and usually there is no control on the rate of submitting

search queries by each user, hence, conserving network bandwidth is very important to performance of the network.

Table 6.1: Example results for one query as they arrive from different peers; each row shows results from one node

Peer	Results Relevant/Total	Accumulated results Relevant/Total	Overlay distance (Hops away)	Accumulated recall (%)	Accumulated Precision (%)
P_1	2/4	2/4	1	20	50.0
P_2	3/3	5/7	1	50	71.4
P_3	2/7	7/14	2	70	50.0
P_4	2/5	9/19	3	90	47.4
P_5	1/2	10/21	4	100	47.6

Table 6.2: Response time calculated as distance on the overlay topology

Recall threshold *r* (%)	Response time (hops)
40	1
50	1
70	2
80	3

For the purpose of simplicity, we measure the network bandwidth in terms of application-level messages. This is a realistic approximation since Freelib search

messages are light weight and have identical size. Therefore, the bandwidth consumption due to these messages is directly proportional to the number of messages. This approach simplifies our evaluation experiments as it relieves us from measuring or simulating the finer details of the underlying physical network.

6.1.4 Experiments

Evaluation of systems such as Freelib is a rather complicated task. The evaluation parameters include the number of participating nodes (network size), the number of different interest areas, the number of nodes in each interest area, the parameters of the ranking protocols, the average size of the local collection at each node, the homogeneity/heterogeneity of nodes in terms of resources, and the average rate of search and access. We have conducted numerous experiments and gave special attention to the system aspects that directly affect the user experience. For the sake of statistical accuracy, all the measurements that we report are averaged over 3 repetitions using different seeds for the random number generators that we use.

6.2 Testbed

After completing the system design, we developed a reference implementation of the Freelib universal client with most of the key features being realized. We ran copies of the Freelib client on a cluster of 20 machines running Linux. We ran 10 of the Freelib client on each machine each of which represents one user. All nodes in the testbed used the Linux operating system network protocol stack for communicating messages and downloading content. This is true even for messages communicated between nodes running on the same machine.

The main objective for creating the testbed was to perform extensive testing of our implementation in a real life environment. The validation of the code was successful; however, we discovered the limitations of this approach. Using the 20 Linux machines available, we were able to start a maximum of 200 Freelib nodes with 10 nodes on each machine. If we try to start more nodes, the system resources get overloaded and our measurements become inaccurate. This is in part due to the multi-threaded nature of our implementation of the Freelib client. On Linux, each thread is implemented as a process. This is becomes costly especially for a heavily threaded program like the Freelib client. In fact, in order to be able to run 10 nodes on a single machine, we had to modify the code to reduce the number of threads used by each instance. Hence, we began the design of a simulator described later in this chapter.

6.2.1 Testbed Results

Figure 6.1 shows the recall as a function of TTL for two Freelib networks one with 4 friends and the other with 5 friends and for a Symphony network with 2 short contacts and 4 long contacts. As mentioned earlier, we ran the Symphony network by turning off the access network and using only the support network for search. The results shown are in Figure 6.1 represent the average of 3 separate experiments that we performed. The number of nodes in this set of experiment was 200 nodes. It shows that Freelib recall grows faster especially early in the life time of the query. For example, consider the recall at TTL = 3. At this TTL, Symphony recall is 26% while Freelib gave 90% recall. The same results can be viewed from another perspective as follows. Using 2 hops, Freelib achieves the same recall level (R = 50%) as Symphony achieves in 4 hops.

These results were encouraging. However, as discussed earlier, we could not run networks larger than 200 nodes on the testbed.

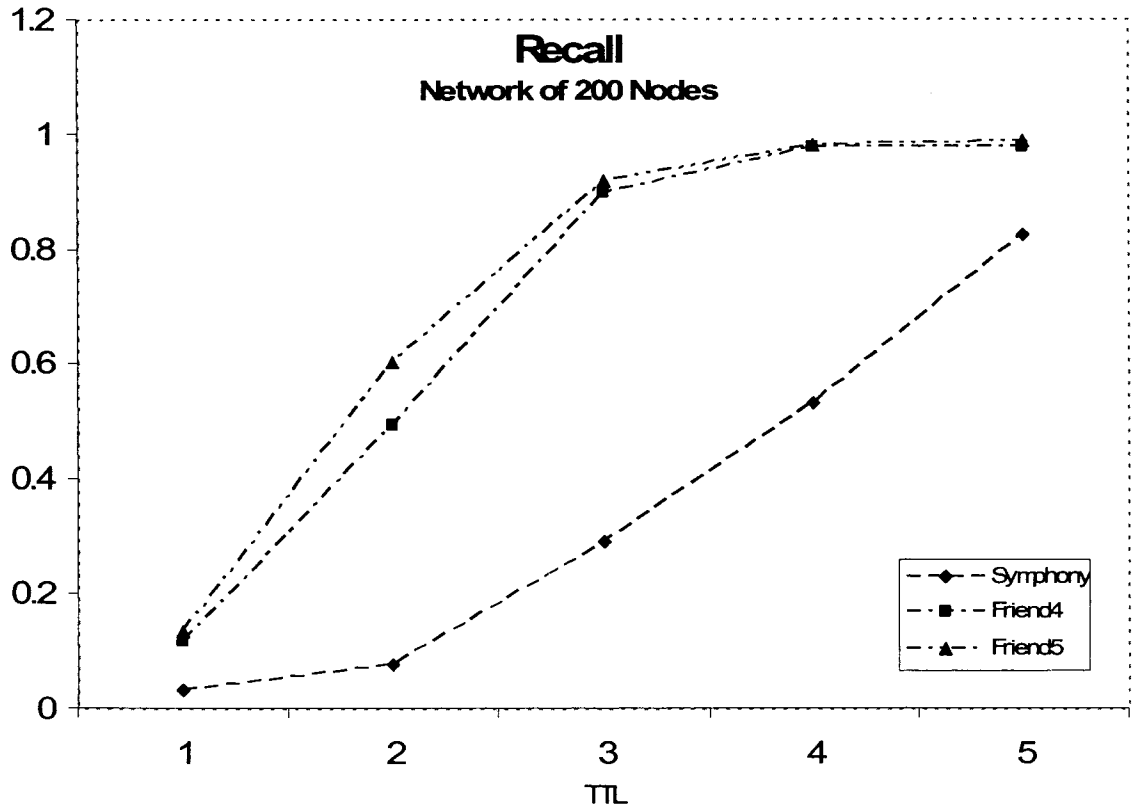


Figure 6.1: Testbed experiment: Recall as a function of TTL

6.2.2 Need for Building a Simulator

Since peer-to-peer networks are usually large-scale systems, we need to evaluate large networks of thousand of nodes. Nonetheless, by using the testbed, we were limited to running small Freelib networks of up to 200 nodes. We could not evaluate larger networks. We immediately started to study the alternatives. It was clear that we should perform our evaluation using a simulator. Simulation can enable us to omit unnecessary details of the underlying network and concentrate on simulating the higher levels of the

protocol stack. We gave consideration to existing network simulators such as NS2 [48]. However, we found these systems to be directed more towards simulating the fine details of the network. We found some existing simulators [47, 51, 52, 66] that are targeting large-scale systems such as peer-to-peer networks. However, we decided to build a simulator for Freelib. This decision was in part due to the time needed to study the inner workings of these systems and modify and use them to simulate Freelib. In addition, it was due to great deal of prior experience that we have in designing and implementing similar simulation systems.

6.3 Simulator

The main objective of designing and building a simulator is to enable the evaluations of large Freelib networks that have thousands of nodes. By building a simulator, we avoided the resource issues that we faced with the testbed. This was achieved in part by omitting the low-level network details and replacing real network communications with local method calls. Two factors guided us in making the decision of eliminating low-level networking details from the simulation. First, computer networks and protocols are becoming more and more reliable and fast. So, we can concentrate on simulating the application level protocol and leave out the low-level details. Second, other simulators that are targeting large-scale peer-to-peer systems have made the same choice. Examples of those simulators include 3LS [66], Neuro-Grid [47] PeerSim [51], and P2PSim [52]. An internet draft [7] was submitted Jan 2006 describing these simulators, their design, and their implementation languages. In addition, the results we obtained from our simulator were very similar to those we obtained from the experiment we performed on the testbed, which gave us more confidence in our decision. We start this section by

presenting the simulator design. Then we present the evaluation results we obtained by running or experiments using the simulator.

6.3.1 Simulator Design

We used an event-based approach in the simulator. The simulation engine provides an abstract class *SimulationEvent* that represents general events and an event queue that holds the events sorted in a non-descending order based on the event timestamp. Any module involved in the simulation must provide concrete event implementation by extending the abstract class *SimulationEvent*. In addition, the module needs to implement and register an event handler that will be invoked by the *Simulator* main class to process the corresponding event. The *EventHandler* interface, which is used by the simulator to refer to any event handler, must be implemented by any module that creates its own concrete events. In software engineering terms, this approach is referred to as the template patterns approach. It provides great flexibility and extensibility as new modules can be introduced without any need to changing the simulation engine.

This design has also the advantage that only one single thread is needed to perform all the work. This thread is utilized by the *Simulator* class to initialize the simulation run and process the simulation events. The class diagram in Figure 6.2 shows the interfaces and classes of the simulation engine. In addition, it shows how the Freelib messenger module provides a concrete event class *MessageEvent* and an *EventHandler* for processing message events, which is the main *Messenger* class itself. The class *MessageEvent* extends the abstract *SimulationEvent* class and the class *Messenger* implements the *EventHandler* interface.

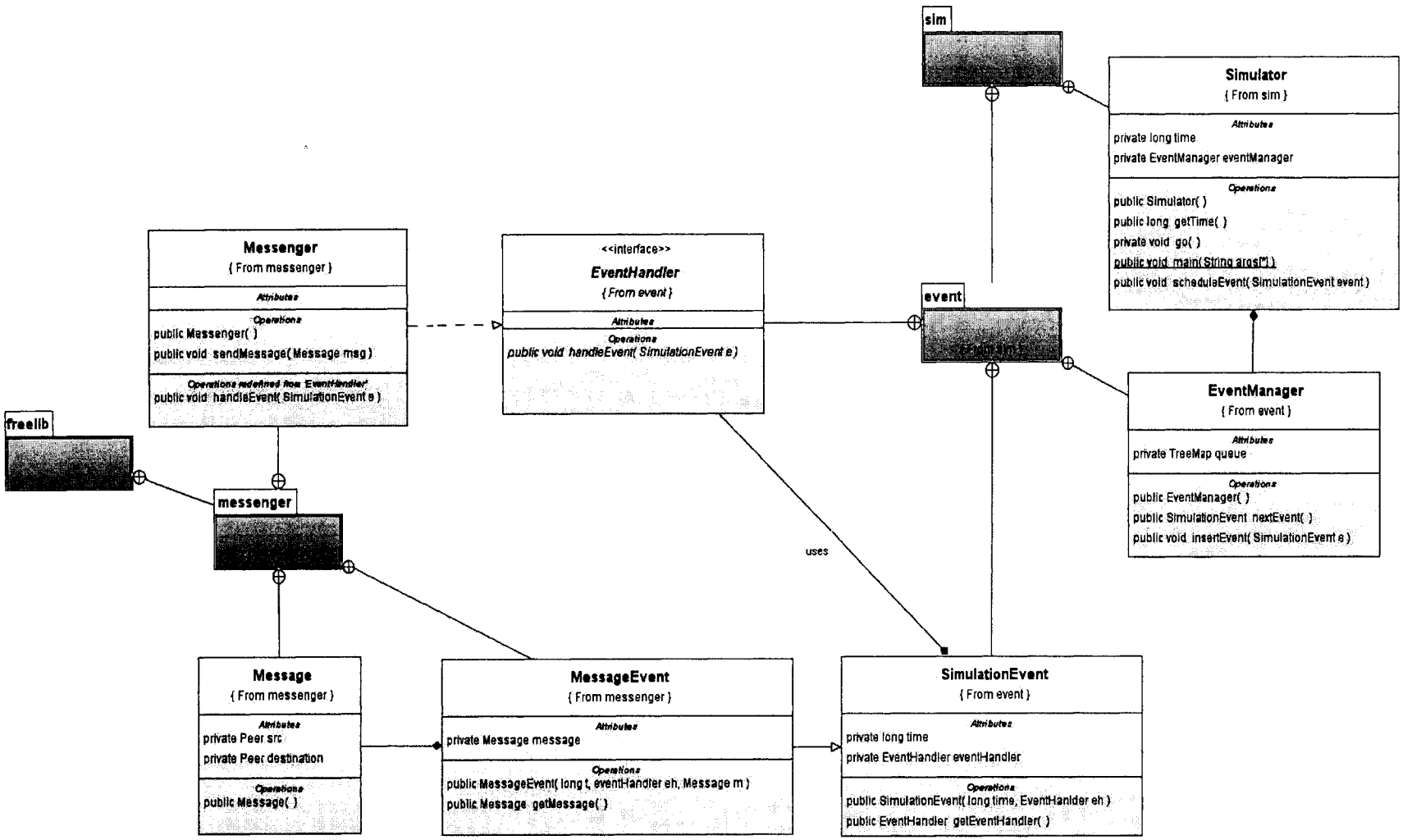


Figure 6.2: Simulator class diagram showing the simulation engine classes and the Freelib Messenger module

Listing 6.1 shows the main simulator loop. First, the simulation event at the head of event queue is fetched. Then, the simulator' internal clock is advanced to the timestamp of the event. And then, the event is processed by invoking the corresponding event handler. In this design, the event object has a reference to the appropriate event handler.

Listing 6.1: Main simulation engine loop for processing simulation events

```
processEvents () {
    SimulationEvent e;
    while (true){
        e = eventManager.getNextEvent();        // get next event
        if (null == e)                          // exit if no events to process
            break;
        time = e.getEventTime();                // advance the clock
        e.processEvent();                       // process the event
    }
}
```

The other key modules that use the event framework in the same manner as the messenger module include the *search* module used for submitting search requests and collecting results, the *access* module used for sending and responding to access requests, and the network protocols. The network protocols include the *Friend Manager* module used for maintaining the friends, the Long Contacts module used for maintaining Symphony long contacts, and the *Ring* module used for maintaining short contacts.

6.3.2 Verification and Validation of our Simulation Model

The results that we obtained from the testbed served one more objective beyond validating the code and obtaining small scale performance results. We used it to partially validate the correctness of our simulation design and implementation. We repeated the

testbed experiments using the simulator and the results were almost identical. This gives us a high degree of confidence in our simulation results. In addition, the results reported in this chapter were calculated by averaging 3 repetitions of each experiment using different seeds for the random number generators that we use in various parts of the simulations. This insures the statistical accuracy of the results.In addition to comparing simulation results to testbed results for validation of our simulation model, we took other steps to verify that the model was built correctly. For example, the code was checked by developers other than the author. In addition, flow diagrams for processing the various events were built and the code was analyzed closely to make sure that its logic follows these diagrams for each event type. Furthermore, for smaller networks, results for randomly selected queries we verified by hand to make sure the processing is correct.

Furthermore, we have performed sensitivity tests on many of the parameters including the search query arrival rate, the percentage of hub nodes, and the network delays. The objective of these sensitivity tests is to give us more confidence of our choices of the various parameter values. To perform a sensitivity test on a parameter, we repeat the experiment using different values of the parameter while fixing the values of all the other parameters. We then analyze the results to discover any discrepancies or changes that are not expected. For example, a sensitivity test on the search query arrival rate showed us that reasonable changes in search rate only resulted in changes in the duration of the simulation (wall clock time). A search rate that is too high, however, resulted in memory issues as the event queue becomes too large.

6.3.3 Experiments and Results

In this section, we present the evaluation results. We conducted sets of experiments with individual experiments in each set. Each experiment consists of 3 simulation runs using the same configuration parameters but different random number seeds. The result for an experiment is calculated by averaging the results of its constituting simulation runs. Sometimes, for the purpose of brevity and when results are similar, we present the outcome of several experiments and report on the others in the text.

Each simulation run consists of the following stages:

- Reading the configuration information: The main *Simulator* class starts by loading the configuration file. This is an XML file that contains the configuration information for the simulation run. The configuration information in that file includes the number of participating nodes, the number of different interest areas (prospective communities), the number of nodes in each interest area, the percentage of hub nodes (these are nodes that have bandwidth and hence can allow more network connection), the average rate of search and access, and the total simulation time for the run.

- Creating and initializing the participating nodes: After loading the configuration file, the *Simulator* class creates the nodes and initializes them using the configuration information.

- Activating the user module: The simulator then starts the user module on the participating nodes. This module is responsible for generating and processing the main user events such as submitting search requests and performing

accesses. In addition, this module collects the results, calculates the required measurements, and writes them to an output file.

- Processing events: The simulator then starts the main event processing loop, which is shown in Listing 6.1 above. The simulation loop ends when all user-related events are processed.

- Cleanup: Finally, the simulator notifies the various modules to cleanup for simulation shutdown. The most important of these is the user module, which needs to write any results that are still in memory to persistent storage.

We used a Windows XP Desktop machine with 3GHz Intel Pentium 4 processor and 2GB of memory to run the simulation experiments. We ran experiments for network sizes that range from 200 to 4000 nodes. The number of interest areas was in the range of 1 to 40. The number of nodes in each interest area was 100 to 700 nodes. We experimented with network with 5% as well as with 0% hub nodes. The total number of experiments was 70 experiments with different settings for the number of nodes, nodes per interest area and percentage of hub nodes. Each experiment was repeated 3 times with different seeds and results are averaged. In the following sections, we present the evaluation results.

6.3.3.1 Quality of Results

The main performance metric we use to evaluate the quality of results is the recall of the search results. It measures the percentage of the relevant results that our search protocol retrieves for a certain query. We measure the absolute recall as well as the normalized recall, which is the recall per one search message.

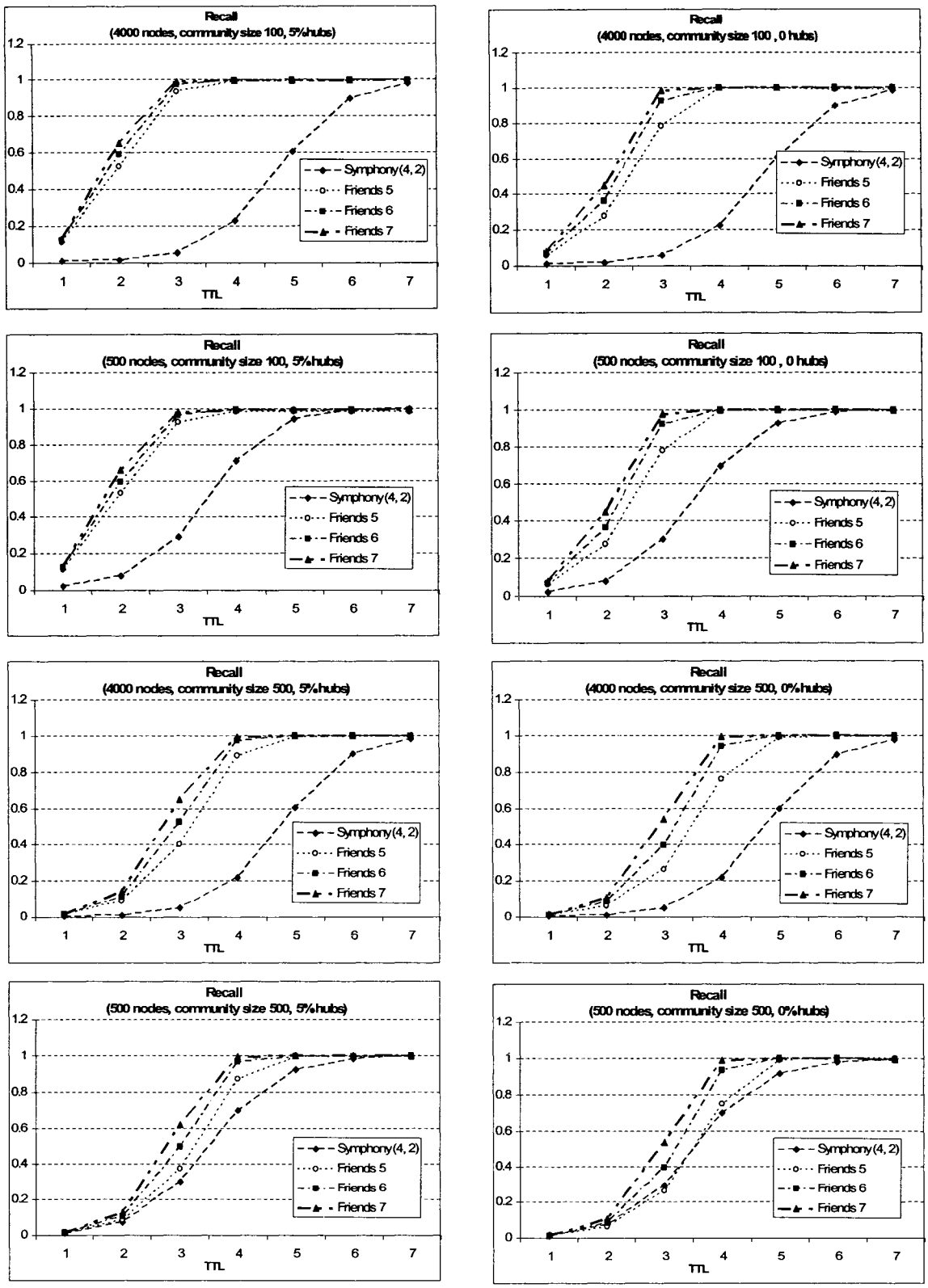


Figure 6.3: Recall vs. TTL for network sizes 500, 4000 nodes; community sizes 100, 500 nodes; and hub nodes 0% and 5 %

Figure 6.3 gives the recall for selected experiments. The other experiments gave very similar results. In Figure 6.3, we show the results for Freelib with 5, 6, and 7 friends per node and for Symphony with 2 short contacts and 4 long contacts. These results show that Freelib achieves recall that is order of magnitudes higher than Symphony early in the life-time of the query. Consider for example the network of 4000 nodes with average community size of 100 and 5% hub nodes. Freelib achieves recall level of 95% at 3 hops which is 19 times the recall Symphony achieves (5%) at the same TTL. Symphony eventually reaches the same final recall level but it takes almost double the time in most cases. The recall gap between Freelib and symphony gets larger for larger networks that consist of many communities. The recall gain from introducing hub nodes is marginal after TTL of 3. For lower TTL, the recall gain from hub nodes is up to 100%. As an example consider the Freelib networks with 4000 nodes and community size of 100 at TTL=2. In case of no hub nodes, recall is around 30%. This is half the recall of the same network with 5% hub nodes, which has recall just below 60%.

Figure 6.4 shows the normalized recall as a function of TTL. We remind the reader that normalized recall is the average recall per 1 search message. A high normalized recall means that the search protocol succeeds in targeting relevant nodes and avoids sending the search query to node that are not relevant to the search query.

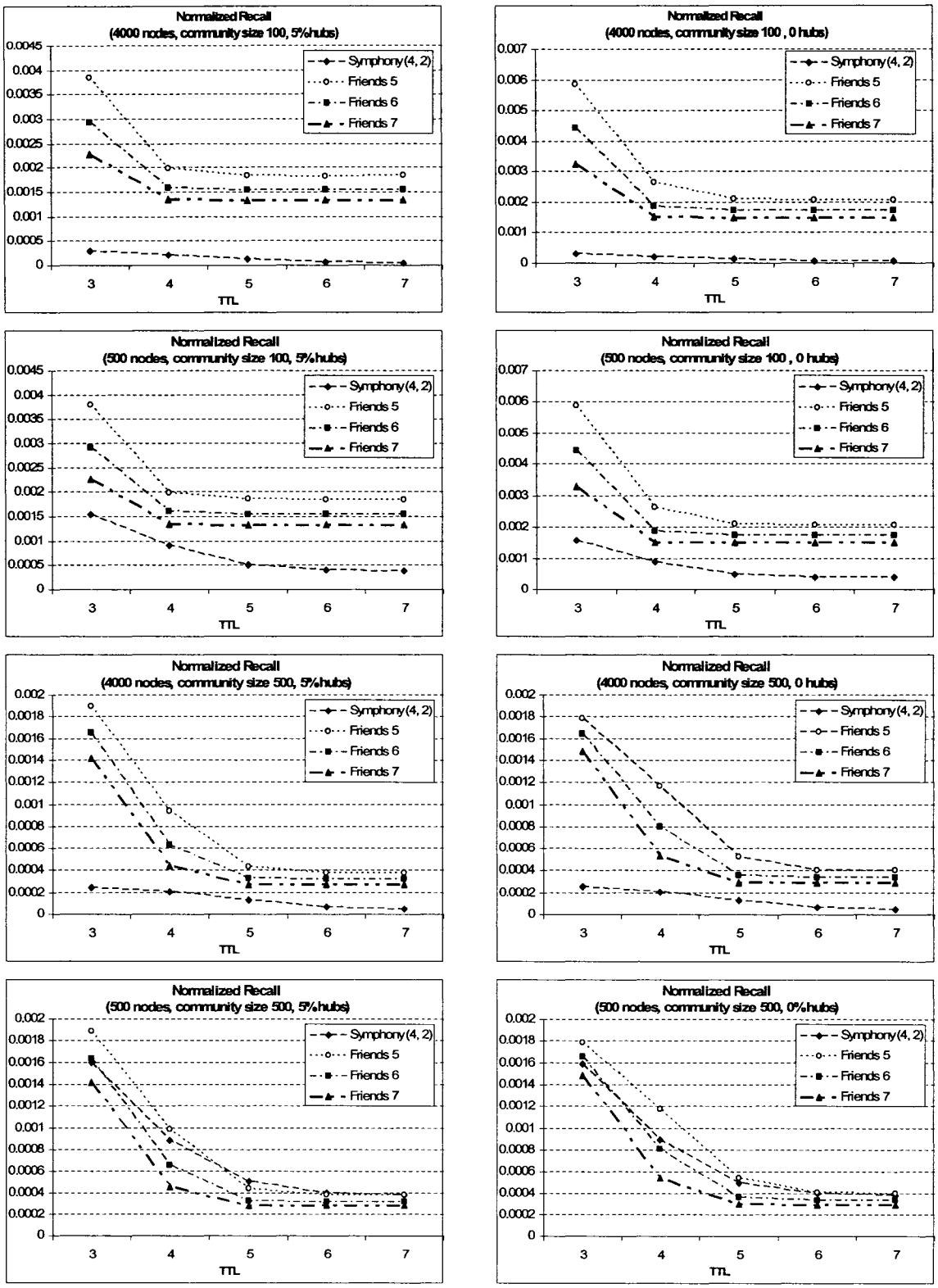


Figure 6.4: Normalized recall vs. TTL for network sizes 500, 4000 nodes; community sizes 100, 500 nodes; and hub nodes 0% and 5 %

The results in Figure 6.4 show that Freelib has a significant normalized recall gain over Symphony for large networks that consist of many communities. For example, the normalized recall for the Freelib network of 4000 nodes with community size of 100 and 5% hub nodes at TTL of 5 is 15 times the recall of the Symphony network of the same number of nodes and community size. The recall gap is larger for smaller TTL. This matches our expectation when we discussed Freelib communities earlier in chapter III. For the smaller TTL values, messages reach nodes that are close on the access topology, which are the most relevant nodes. For higher TTL, the query might reach nodes that are not relevant and hence the normalized recall decreases slightly.

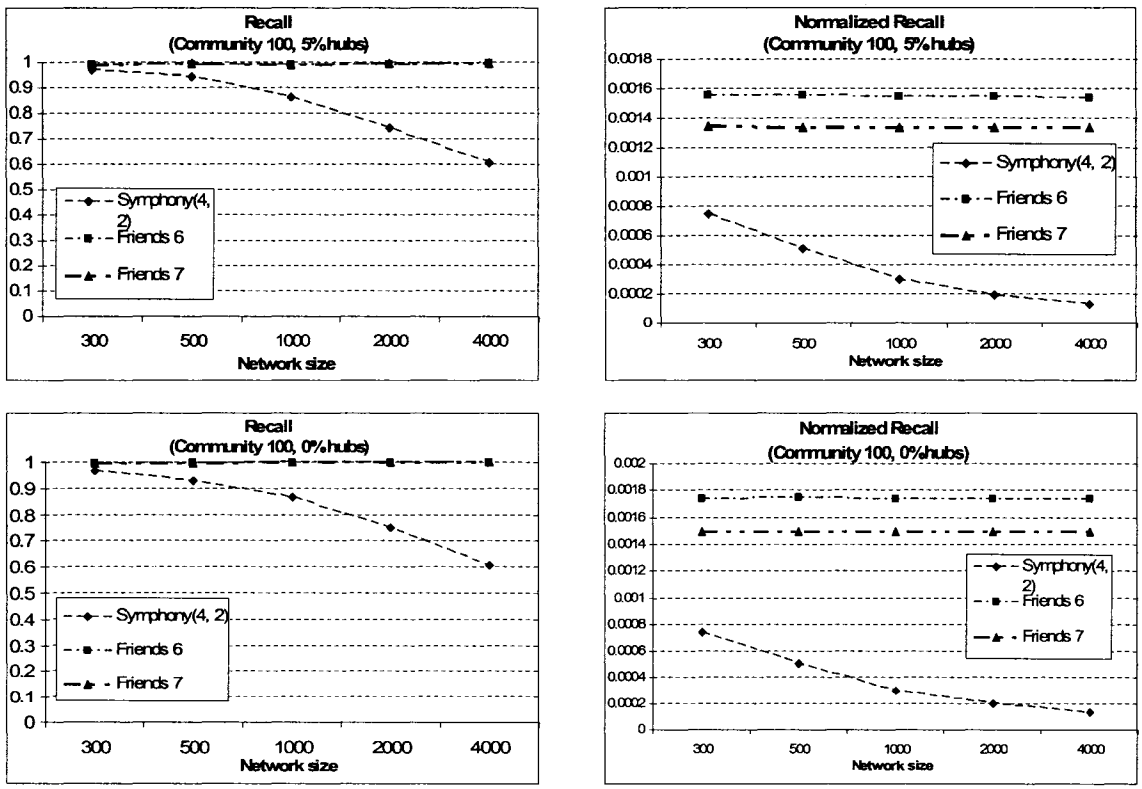


Figure 6.5: Recall and normalized recall vs. network size

In addition to measuring recall as a function of TTL, we studied how it changes with network size. Figure 6.5 shows the recall and normalized recall for networks whose sizes from 300 nodes to 4000 nodes. Community size is 100 nodes. Other community sizes gave similar results. The results in Figure 6.5 show both homogenous networks (0% hub nodes) as well as networks with 5% hub nodes. As we can see from these results, Freelib maintains high recall levels as the network size grows, while on the other hand, the recall of Symphony deteriorates. This shows the benefit of the concept of targeting relevant nodes that Freelib utilizes. For example, Symphony recall deteriorates from 95% to 60% when network size increases from 500 nodes to 4000 nodes for a community size of 100 nodes. Freelib recall stays at the same high level.

We also studied how recall changes with community size (the number of nodes per interest area). In this set of experiments, we simulated networks of size 4000 nodes, which is the largest network size that we can simulate using our current simulator. The community sizes that we studied ranges from 100 to 500 nodes. Figure 6.6 shows the recall and normalized recall for homogenous network of 4000 nodes as well as one with 5% hub nodes. The community sizes in this figure are represented as a percentage of the network size. These results show that Freelib achieves considerable recall and normalized recall gain over Symphony. The recall gain over Symphony is order of magnitudes in most cases and increases as the community size gets smaller relative to the network size, which we believe is the case in real-life peer-to-peer systems. For example, for community size of 2.5% (100 nodes), Freelib recall is above 95%. This is more than 18 times the recall we get from Symphony, which is around 5%. The results from this set of experiments also show that the recall gain from introducing hub nodes increases as the

community size increases. For community size of 2.5% (100 nodes), recall gain due to hub nodes is marginal. It increases as community size increase. For community size of 12.5% (500 nodes), the recall gain due to hub nodes is around 11%.

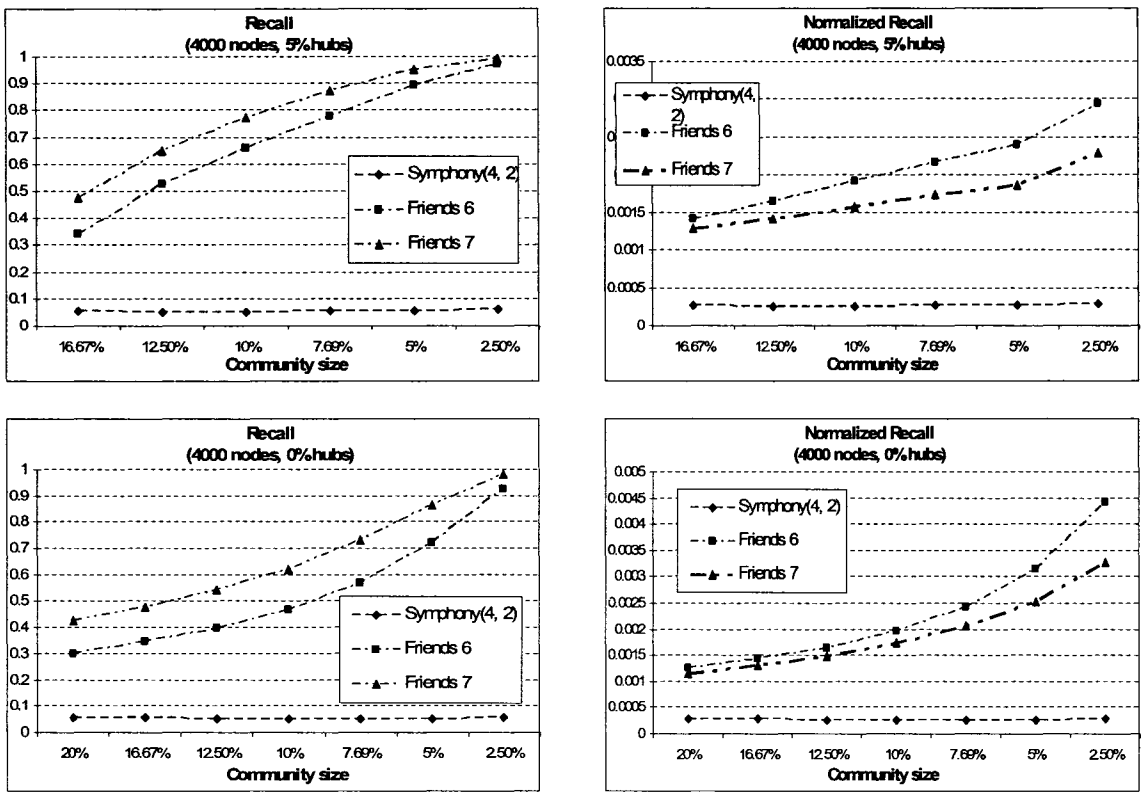


Figure 6.6: Recall and Normalized recall vs. Community size

6.3.3.2 Bandwidth Usage

We measure bandwidth in terms of the number of application level messages as discussed earlier. Conserving network bandwidth is essential especially in large scale distributed and peer-to-peer systems. We measured bandwidth usage for network sizes that ranged from 200 nodes to 4000 nodes and for community sizes from 100 to 700 nodes. We also measured bandwidth for homogenous networks as well as for networks that have hub

nodes with more resources. The results we obtained from these experiments show that Freelib achieves considerable bandwidth savings over Symphony for larger networks that consist of many communities. For small networks that consist of one community, the bandwidth savings were marginal. This is expected as, in this case, all nodes belong to the same community and therefore Freelib does not benefit from the concept of targeting relevant peers. The results also show that hub nodes have marginal effect on the bandwidth usage of Freelib. Figure 6.7 shows the bandwidth usage as a function of the recall level for some of these network configurations. The results for other networks are similar. For large networks, messages sent by Symphony were order or magnitudes larger than those send by Freelib for the same recall level. Consider for example the network of 4000 nodes with community size of 100 and 5% hub nodes. The number of messages sent by Symphony to achieve recall level of 99% was almost 20,000 messages. The corresponding number of messages for Freelib was less than 1000 messages. For the same network size with community size of 500 and 5% hub nodes, the number of messages sent by Freelib was less than 3000. For network size of 500 nodes all in one community, Freelib needed roughly the same number of messages as Symphony since it needed to reach all the existing nodes.

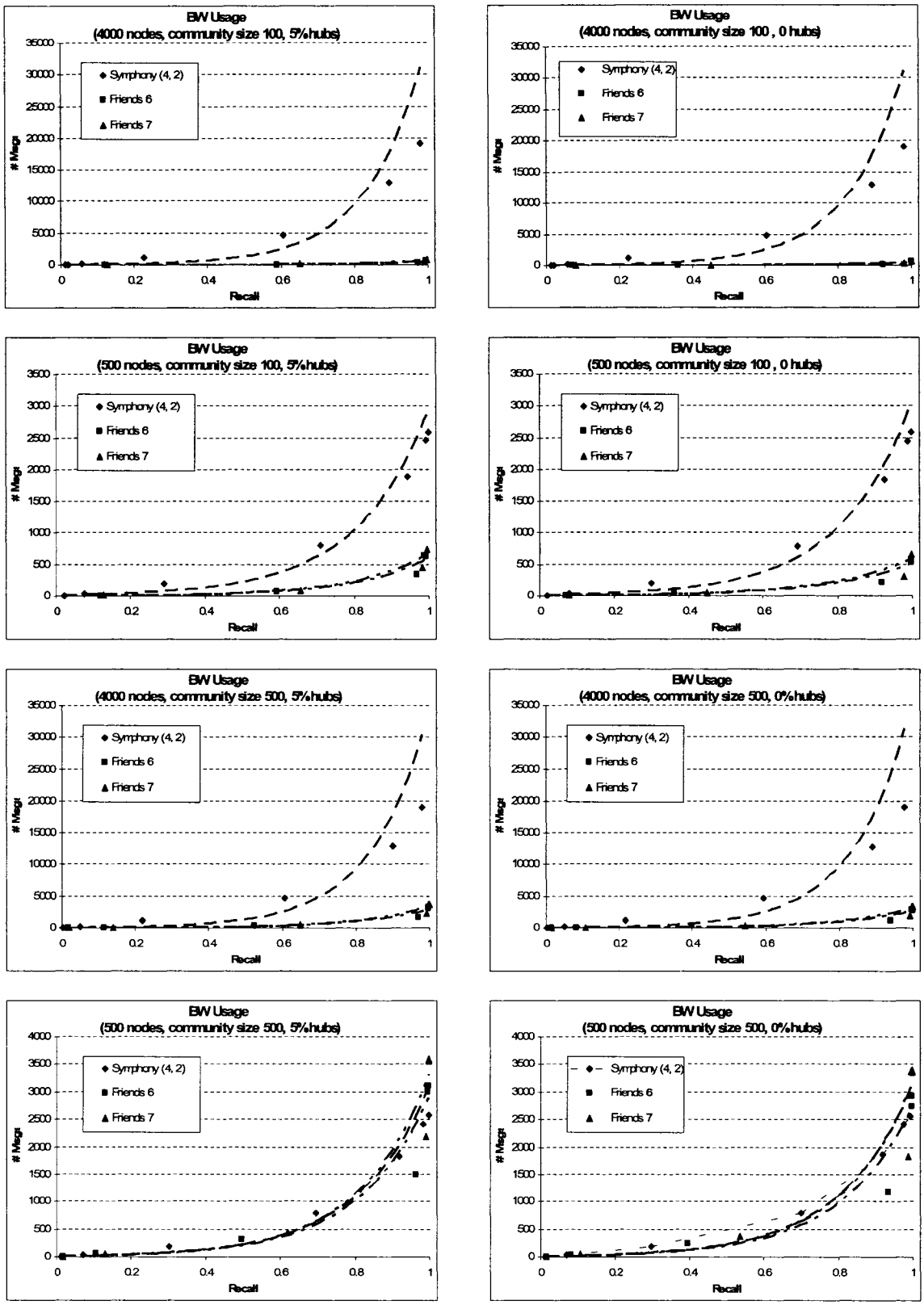


Figure 6.7: Bandwidth vs. recall for network sizes 500, 4000 nodes; community sizes 100, 500 nodes; and hub nodes 0% and 5 %

6.3.3.3 Response Time

We now look into another important performance measure. That is the response time of search queries. As discussed earlier, measuring response time in a peer-to-peer system such as Freelib is a challenge. This is due to the fact that results of a certain search query are accumulated over time as individual peers respond to the search request. Although, the users usually examine the first one or two pages of results, it is hard to draw a line that specifies the time at which the user is satisfied with the results. Consequently, we present response time as a function of the recall level achieved. For a certain minimum recall level, the response time is the minimum time at which this recall level is reached. Also as discussed earlier, we measure response time in terms of the distance (the number of hops) on the access overlay topology.

Figure 6.8 shows the response time for selected network configurations. Results for other network configurations are very similar. The top left graph in Figure 6.8 shows the response time for a network of 4000 nodes with community size of 100 and 5% hub nodes. For a recall level of 60%, the Symphony needed 5 hops while Freelib achieved the same recall level in 2 hops. We believe that in bigger networks, Freelib will be able to achieve more time savings. The response time savings for smaller networks with fewer communities was smaller. To summarize, the results in Figure 6.8 show that for larger networks that consist of many communities, Freelib achieves considerable reduction in response time over Symphony. And in general, the response time enhancement gets better as network sizes increases.

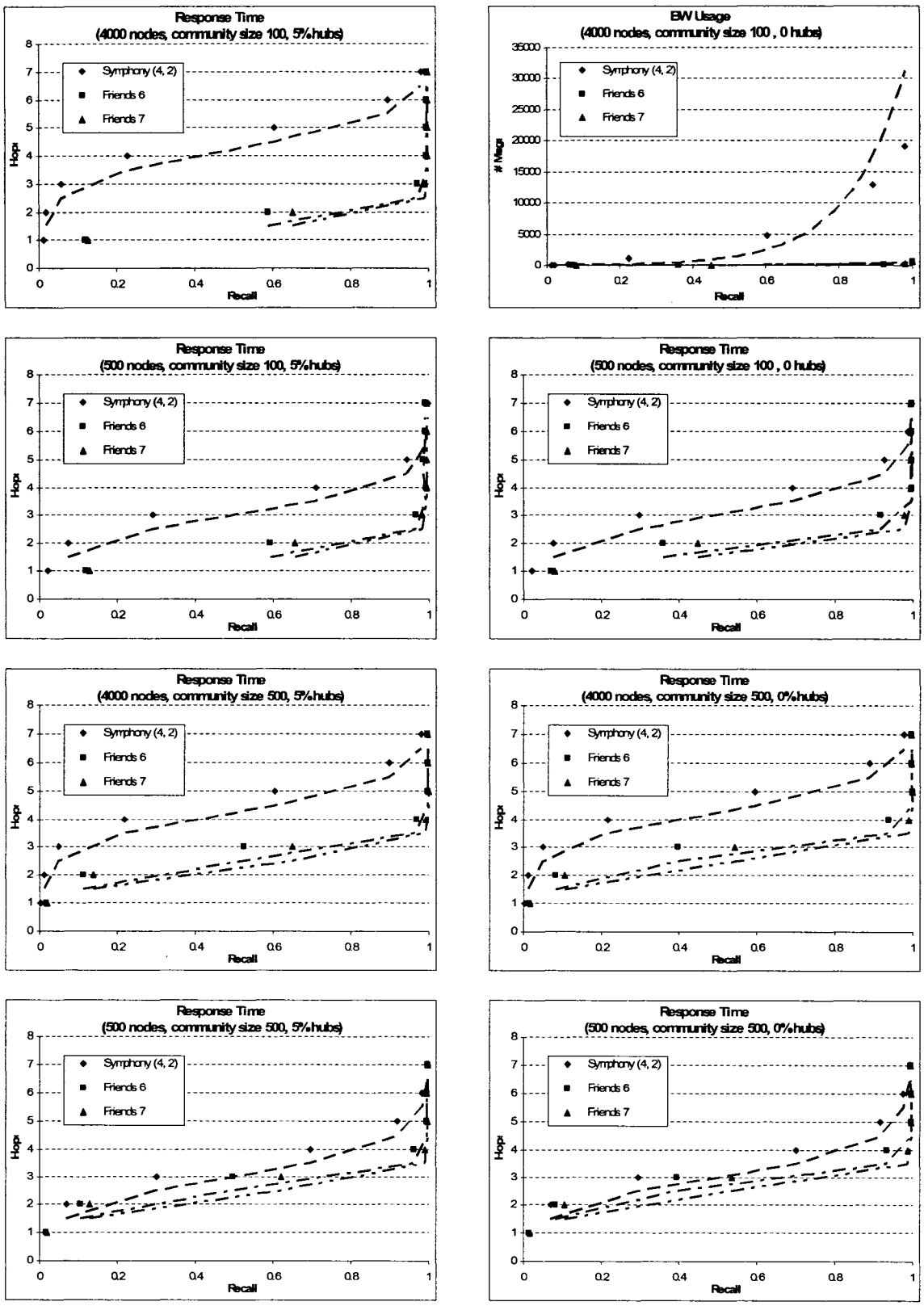


Figure 6.8: Response time vs. recall for network sizes 500, 4000 nodes; community sizes 100, 500 nodes; and hub nodes 0% and 5 %

6.4 Summary

In this chapter we presented the evaluation of the Freelib framework. We started by discussing the evaluation methodology and the measurements that we are using in the evaluation. We then presented a testbed we built for the purpose of testing our implementation and performing preliminary evaluation of the proposed work. We explained how scalability issues prevented us from running large networks using the testbed. We then presented our Freelib simulator that we built for that purpose. We presented the simulator design and the results from numerous experiments that we ran using the simulator.

From the evaluation of the Freelib system, we can conclude that Freelib succeeded in targeting relevant nodes. In general, Freelib is advantageous and achieves considerable gains over Symphony that reaches orders of magnitudes when the network is large and consists of many communities. For a smaller network that consists of few communities, the recall gain over Symphony is marginal. In addition to the recall gain, Freelib also achieved considerable savings in bandwidth and response time. The bandwidth savings is orders of magnitude in most cases. The response time savings were considerable as well and it reached 50% in many cases. In summary, Freelib achieves considerable performance gain over Symphony especially for larger networks with many communities, which we believe is the case in real-life peer-to-peer systems.

We could not however simulate networks larger than 4000 nodes using our simulator. Once the network size goes beyond this limit the event queue gets extremely large in size and the time to run the simulation becomes prohibitively too long. However, from the results we obtained, we believe that Freelib will have even better results for

larger networks. We are currently looking into several ways to improve our simulator to enable simulation of larger networks. Once we implement these ideas, we believe that we shall be able to evaluate the performance of Freelib in much larger networks than the ones whose results were presented in this chapter.

We conclude this chapter by discussing the impact Freelib can have on users and communities in a real-life deployment of such a system. As we have seen from the evaluation in this chapter, Freelib achieved significant performance gains through the new concept of evolving communities. This would greatly enhance the user experience as users obtain results faster, wait less, and get enhanced recall and precision. In addition to the performance enhancement, community evolution enables people sharing same interest to connect to each other and get together (electronically), which open the door for potential collaboration and social activities.

By evolving users into communities, Freelib is expected to enhance the scalability of the peer-to-peer network. Instead of having to search the whole network as in the case of ad hoc peer-to-peer systems, Freelib nodes need to search a much smaller subset of nodes, their respective communities. This results in order of magnitudes savings in bandwidth as shown earlier in our simulation results. The end result is the ability of the Freelib framework to scale up to unprecedented peer-to-peer network sizes while maintaining the same enhanced user experience.

CHAPTER VII
CONCLUSIONS AND FUTURE WORK

In this thesis, we introduced the Freelib framework for building digital collections on top of a peer-to-peer network. Freelib targets the individual users that want to share and disseminate their digital objects as well as search for and discover content shared by others. Freelib, like other peer-to-peer networks, uses resources of the participating nodes and does not rely on dedicated, centralized resources. All the processing power, the storage, and the network bandwidth are contributed by the participating nodes. This makes Freelib self-sustainable.

We introduced the concept of evolving an overlay topology based on the similarity of user interest. This enabled us to evolve users into logical communities of common interest where members of the same community are close to each other in the overlay topology. This novel idea contributed to a significant performance gain over traditional or ad-hoc peer-to-peer systems as shown in our simulation results presented in chapter VI. Freelib uses Dublin Core metadata [16] to enable metadata based search and discovery which provides the user with the ability to specify more accurate queries.

We developed a prototype implementation of the Freelib peer-to-peer client that realizes the ideas and protocols introduced by the Freelib framework. We used the client to build a test-bed for evaluating our framework. Within the time frame of our thesis we were able to deploy on the order of hundreds of clients and validate the protocol and our implementation. However, due to the limited number of user that we were able to emulate using the test-bed, we were not able to obtain valid performance results with the testbed that could predict the performance of large scale deployment of clients. Hence,

we developed an event-based simulator that enabled us to simulate and study networks of thousands of users.

In summary, the contributions of this work and its impact include:

- **Self-sustainability empowers individuals to share and discover content:** We introduced the concept of self-sustainability and designed a framework to build self-sustainable digital libraries on top of peer-to-peer networks and provided a reference implementation. This work can be utilized to empower individuals to organize and share their digital content and discover content published by their peers.
- **Ranking mutual user interest:** We introduced and implemented a ranking technique that captures mutual user interest transparently. This technique alleviates the need for users to provide explicit feedback.
- **Evolve the network topology to create communities:** We introduced a peer-to-peer network design that evolves the network topology based on the outcome of the ranking process. This evolves users into communities based on user interest as captured by the access pattern analysis. The immediate impact of this is better search performance in terms of higher recall and faster responses, lower bandwidth usage, and enhanced overall user experience.
- **Fast node discovery in peer-to-peer networks:** We introduced various algorithms for node discovery in peer-to-peer networks. This is a necessary feature as it enables nodes to re-discover their community after rejoining the network.

The work presented in this dissertation can be enhanced and extended in many ways. For example, replication and caching can be utilized to enhance content availability. Currently, the 'publishing service' of the Freelib client publishes content to the local user's collection. Consequently, the items stored on a node become unavailable when that node leaves the network and only become available again when it rejoins. Availability of content can be enhanced significantly by having the items published to multiple nodes and/or cached by nodes as they process search results.

We also plan to evaluate the use of an *aging* technique and the use of *weights* in the ranking process. These techniques would be particularly useful during the transition period when a user's interest shifts to a new topic. We could measure recall and response time as well as how long the transition period lasts to determine optimal parameters of the techniques. In addition to aging and utilization of weights, the ranking process can be further enhanced by enabling the user to provide explicit feedback and by fine tuning the ranking process based on the individual items accessed. For example, a user can designate a certain item or a set of items in his local collection as the most important items. Consequently, accesses that involve these key items receive higher weights in the ranking process. This can be viewed as a tool the user can utilize to guide the ranking process to best select the more relevant peers. Another area of potential enhancement is to support automatic extraction of metadata from the common document formats such as PDF and MS Word documents. This alleviates the need for the user to manually type in the metadata. Finally, we would like to study the performance of our framework for larger networks and communities. In order to be able to do this, we may need to implement a distributed simulator that runs on multiple computers.

BIBLIOGRAPHY

[1] ACM digital library: http://portal.acm.org/dl.cfm/, viewed Jan. 10, 2006.

[2] Alexandria digital library project, University of California, Santa Barbara: http://alexandria.sdc.ucsb.edu/, viewed Jan. 10, 2006.

[3] Arc project home page, Old Dominion University: http://arc.cs.odu.edu/, viewed Jan. 10, 2006.

[4] R. Baeza-Yates and B. Ribeiro-Neto, "Modern Information Retrieval", Addison Wesley 1999.

[5] M. Bawa, G. S. Manku, and P. Raghavan, "SETS: Search Enhanced by Topic Segmentation", *Proc. 26th Annual Int'l ACM SIGIR 2003*, Toronto, Canada, July 28 to Aug 1 2003.

[6] Bittorrent home pgae: http://www.bittorrent.com, viewed Jan. 10, 2006.

[7] A. Brown and M. Kolberg, "Tools for Peer-to-Peer Network Simulation", *Internet draft, draft-irtf-p2prg-core-simulators-00.txt*, Jan. 2006.

[8] B. Carlsson and R. Gustavsson, "The Rise and Fall of Napster - An Evolutionary Approach", *Proc. 6th Int'l Computer Science Conference on Active Media Technology*, Hong Kong, China, Dec. 2001.

[9] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A Distributed Anonymous Information Storage and Retrieval System", *Proc. Int'l Workshop on Design Issues in Anonymity and Unobservability, (LNCS 2009)*, Berkeley, CA, USA, July 2000, pp 46-66.

[10] J. Cocke and V. Markstein, "The Evolution of RISC Technology at IBM". *IBM Journal of Research and Development*, volume 34, (no. 1), pages 4-11. 1990.

[11] Copernic home page: http://www.copernic.com, viewed Jan. 10, 2006.

[12] H. Ding and I. Solvberg, "Metadata Harvesting Framework in P2P-Based Digital Libraries", *Proc. Int'l Conference on Dublin Core and Metadata Application*, Shanghai, China, October 11-14, 2004.

[13] Distributed.net home page: http://www.distributed.net, viewed Jan. 10, 2006.

[14] M. Doane, "Metadata, Search and Meaningful ROI, *Proc. DCMI Workshop*, Seattle, Washington, USA, 2003.

[15] DataSynapse home page: http://www.dataSynapse.com, viewed Jan. 10, 2006.

[16] Dublin Core Metadata Initiative, home page: http://dublincore.org, viewed Jan 10, 2006.

[17] Fedora project home page: http://www.fedora.info, viewed Jan. 10, 2006.

[18] Freenet project: http://www.freenetproject.org, viewed Jan. 10, 2006.

[19] J. Frew, M. Freeston, N. Freitas, L. Hill, G. Janee, K. Lovette, R. Nideffer, T. Smith, and Q. Zheng, "The Alexandria Digital Library architecture", *Proc. 2nd European Conference on Research and Advanced Technology for Digital Libraries (ECDL'98)*, pp. 61-73, Heraklion, Crete, Greece, Sept. 1998.

[20] D. Gibson, J. Kleinberg, and P. Raghavan, "Inferring Web communities from link topology", *Proc 9th ACM Conference on Hypertext and Hypermedia*, 1998.

[21] Gnutella home page: http://www.gnutella.com, viewed Jan. 10, 2006.

[22] L A. Granka, T. Joachims, and G. Gay, "Eye-Tracking Analysis of User Behavior in WWW Search", *Proc 27th annual Int'l ACM SIGIR conference on Research and development in information retrieval*, University of Sheffield, UK, July 25 - 29, 2004.

[23] Groove Networks home page: http://www.groove.net, viewed Jan. 10, 2006.

[24] J. L. HENNESSY and D. A. PATTERSON, "Computer Architecture: A Quantitative Approach", Morgan Kaufmann, San Mateo, Calif., 1990.

[25] IEEE digital library: http://www.computer.org/publications/dlib/, viewed Jan. 10, 2006.

[26] IntraLinks home page: http://www.intralinks.com, viewed Jan. 10, 2006.

[27] ISO standard: "Information technology -- Open Systems Interconnection -- Remote Procedure Call (RPC)", ISO/IEC 11578:1996.

[28] International Telecommunication Union standard, "Information technology - Open Systems Interconnection - Procedures for the operation of OSI Registration Authorities: Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 Object Identifier components", ITU-T Rec. X.667 | ISO/IEC 9834-8:2005, available at http://www.itu.int/ITU-T/studygroups/com17/oid.html, viewed Feb 2007.

[29] T. Joachims, L. Granka, B. Pang, H. Hembrooke, and G. Gay, "Accurately Interpreting Clickthrough Data as Implicit Feedback", *Proc. 28th Annual ACM Conference on Research and Development in Information Retrieval (SIGIR)*, August 15-19, 2005, Salvador, Brazil.

[30] JXTA Search home page: http://search.jxta.org, viewed Jan. 10, 2006.

[31] KaZaA home page: http://www.kazaa.com, viewed Jan. 10, 2006.

[32] D. Kelly and J. Teevan, "Implicit feedback for inferring user preference: A bibliography", *ACM SIGIR Forum*, Volume 37 , Issue 2, 2003.

[33] The Kepler project home page, Old Dominion University: http://kepler.cs.odu.edu, viewed Jan. 10, 2006.

[34] J. Kleinberg, "The Small-World Phenomenon: An Algorithmic Perspective", *Proc. 32nd ACM symposium on theory of Computing*, Portland, OR, USA, May 21-23, 2000.

[35] C. Lagoze, and H. van de Sompel, "The Open Archives Initiative: building a low-barrier interoperability framework", *Proc. the ACM/IEEE Joint Conference on Digital Libraries*, 2001.

[36] P. J. Leach, M. Mealling, and R. Salz, "UUIDs and GUIDs", *Internet draft, rfc4122.txt*, July 2005.

[37] P. J. Leach, and R. Salz, "UUIDs and GUIDs", *Internet draft, draft-leach-uuids-guids-01.txt*, August 1998.

[38] M. Li, W. Lee, A. Sivasubramaniam, and D. Lee, "A Small World Overlay Network for Semantic Based Search in P2P Systems", *The Second WWW*

Workshop on Semantics in Peer-to-Peer and Grid Computing (SemPGRID'04), New York City, NY, May 2004, pp. 71-90.

[39] X. Liu, K. Maly, M. Zubair, M. Nelson, "Arc - An OAI Service Provider for Digital Library Federation", *D-Lib Magazine*, 7(4), April 2001.

[40] M. J. Lorence, M. Satyanarayanan, "IPwatch: A Tool for Monitoring Network Locality", *ACM SIGOPS Operating Systems Review*, 24(1): 58-80 (1990)

[41] Apache Lucene home page: http://lucene.apache.org/, viewed Jan 10, 2006.

[42] K. Maly, M. Zubair, and X. Liu, "Kepler – An OAI Data/Service Provider for the Individual", *D-Lib Magazine*, 7(4), April 2001.

[43] G. S. Manku, M. Bawa, and P. Raghavan, "Symphony: Distributed Hashing in a Small World", *Proc. 4th USENIX Symposium on Internet Technologies and Systems*, 2003.

[44] R. J. McNab, I. H. Witten, and S. J. Boddie, "A Distributed Digital Library Architecture Incorporating Different Index Styles", *Proc IEEE forum on Research and Technology Advances in Digital Library*, Santa Barbra, CA, April 1998, pp 36-45.

[45] J. C. Mogul, "Network locality at the scale of processes", *ACM SIGOPS Operating Systems Review*, 10(2): 81-109 (1992)

[46] Napster home page: http://www.napster.com, viewed February 10, 2001; and Napster page at Wikipedia, http://en.wikipedia.org/wiki/Napster, viewed Jan 10, 2006

[47] NeuroGrid home page: http://www.neurogrid.net, viewed Jan. 10, 2006

[48] NS2 wiki, http://nsnam.isi.edu/nsnam/index.php/Main_Page, viewed May 11, 2005.

[49] The New Zealand Digital Library home page, University of Waikato, New Zealand: http://www.nzdl.org/fast-cgi-bin/library?a=p&p=home, viewed Jan. 10, 2006.

[50] Open Archives Initiative home page: http://www.openarchives.org, viewed Jan. 10, 2006.

[51] PeerSim home page, http://peersim.sourceforge.net, viewed Jan. 10, 2006.

[52] P2PSim home page, http://pdos.csail.mit.edu/p2psim/, viewed Jan 10, 2006.

[53] F. Radlinski and T. Joachims, "Query Chains: Learning to Rank from Implicit Feedback", *Proc. 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Chicago, Illinois, USA, August 21-24, 2005.

[54] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network", *Proc. ACM SIGCOMM 2001*, August 2001.

[55] Retrievalware home page: http://www.retrievalware.com, viewed Jan. 10, 2006.

[56] P. Mockapetris, "Domain names: Concepts and facilities", *RFC 882*, USC/Information Sciences Institute, Nov 1983.

[57] P. Mockapetris, "Domain Names - Implementation and Specification", *RFC 883*, USC/Information Sciences Institute, Nov 1983.

[58] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems", *Proc. IFIP/ACM Middleware 2001*, Heidelberg, Germany, Nov. 2001.

[59] SETI@home home page: http://setiathome.ssl.berkeley.edu, viewed Jan. 10, 2006.

[60] R. Shi, K. Maly, and M. Zubair, "Automatic Metadata Discovery from Non-cooperative Digital libraries", *Proc IADIS Int'l Conference e-Society 2003 ES2003*, pp. 735-739, Lisbon, Portugal, May 2003

[61] R. Shi, K. Maly, and M. Zubair, "Improving Federated Service for Non-cooperating Digital Libraries", *Proc Int'l conference on Digital Libraries ICDL2004*, New Dehli, India, Feb. 2004

[62] M. Silva and R. Wait, "Sparse Matrix Storage Revisited", *Proc. 2nd conference on computing frontiers*, Ischia, Italy, May 4-6 2005, pp. 230 – 235.

[63] C. Silverstein, H. Marais, M. Henzinger, and M. Moricz, "Analysis of a very large AltaVista query log", *ACM SIGIR Forum*, Volume 33, Issue 1, 1999.

[64] T. Staples, R. Wayland, and S. Payette, "The Fedora Project: An Open-source Digital Object Repository System", *Digital Library Magazine*, April 2003.

[65] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications", *Proc. ACM SIGCOMM 2001*, pp. 149-160, San Deigo, CA, August 2001.

[66] N. S. Ting and R. Deters, "3LS - A Peer-to-Peer Network Simulator", *Proc. 3rd Int'l Conference on Peer-to-Peer Computing (P2P'03)*, 2003

[67] The Technical Report Interchange (TRI) project home page, Old Dominion University: http://128.82.7.99/tri/index.html, viewed Jan. 10, 2006.

[68] United Devices home page: http://www.ud.com, viewed Jan. 10, 2006.

[69] J. Walkerdine and P. Rayson, "P2P-4-DL: Digital Library over Peer-to-Peer", *Proc. 4th Int'l conference on Peer-to-Peer Computing (P2P'04)*. Zurich, Switzerland, August 25 - 27, 2004.

[70] D. J. Watts and S. H. Strogatz, "Collective Dynamics of 'Small-World' Networks", *Nature*, 393:440-442, 1998.

[71] B. Yang, and H. Gracia-Molina, "Designing a Super-Peer Network", *Proc. 9th Int'l conference on Data Engineering*, Bangalore, India, March 5-8, 2003.

[72] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz, "Tapestry: A Resilient Global-scale Overlay for Service Deployment", *IEEE Journal on Selected Areas in Communications*, January 2004, Vol. 22, No. 1.

[73] The Santa Fe Convention home page, http://www.openarchives.org/sfc/sfc_entry.htm, viewed Jan 15, 2003.

VITA

for

Ashraf A. Amrou

EDUCATION:

- Master of Science, Computer Science, Alexandria University, Egypt, Thesis: "*Fast mining of interesting association rules for large-scale problems*", Feb 2001,
- Bachelor of Science, Computer Science, Alexandria University, Egypt, May 1995.

PROFESSIONAL CHRONOLOGY:

- E-Learning Technical Lead, Old Dominion University, Norfolk, VA, Aug 2007 – Present.
- J2EE Developer, Union Pacific Rail Road, Omaha, NE, June 2007- Aug-2007.
- Research Assistant, Old Dominion University, Department of Computer Science, Aug 2001- Dec 2006.
- Assistant Researcher, Informatics Research Institute, MCSRTA, Alexandria Egypt, Dec 1997- June 2001

RESEARCH INTERESTS:

Distributed and Peer-to-peer Systems, Digital Library, Computer Networks

SELECTED PUBLICATIONS:

A. Amrou, K. Maly, M. Zubair; "Performance Evaluation of Freelib, a P2P-based Digital Library Architecture", Proceedings of the International Conference on Digital Libraries (ICDL 2006), Dec. 5-8, 2006, New Delhi, India.

A. Amrou, K. Maly, M. Zubair; "Freelib: Peer-to-peer-based Digital Libraries", Proceedings of the IEEE 20th Int'l Conference on Advanced Information Networking and Applications (AINA 2006), April 18-20, 2006, Vienna, Austria.

A. Amrou, K. Maly, M. Zubair; "Freelib: A Self-sustainable Digital Library for Education Community", Proceedings of the World Conference on Educational Multimedia, Hypermedia and Telecommunications (Ed-Media 04), pp. 15-20, Lugano, Switzerland, June 2004.