ECSP: An Efficient Clustered Super-Peer Architecture for P2P Networks

by

Juan Li

M.Sc., Institute of Software, Chinese Academy of Sciences 2001

B.Sc., Northern Jiaotong University, 1997

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming

to the required standard

The University of British Columbia

August 2003

Abstract

Peer-to-peer (P2P) computing has become increasingly popular in recent years. It offers many attractive features, such as self-organization, load-balancing, availability, fault tolerance, and anonymity. However, it also faces some serious challenges. In this thesis, we propose an Efficient Clustered Super-Peer P2P architecture (ECSP) to overcome the scalability and efficiency problems of existing unstructured P2P systems, using a semi-centralized hierarchical structure: With ECSP, peers are grouped into clusters according to their topological proximity, and *super-peers* are selected from regular peers to act as cluster leaders and service providers. These super-peers are also connected to each other, forming a backbone overlay network operating as a distinct, yet integrated, application. To maintain the dynamically adaptive overlay network and to manage the routing on it, we propose an application level broadcasting protocol: *Efa*. Applying only a small amount of information about the topology of a network, Efa is as simple as flooding, a conventional method used in unstructured P2P systems. By eliminating many duplicated messages, Efa is much more efficient and scalable than flooding, and furthermore, it is completely decentralized and self-organized. Our experimental results prove that ESCP architecture, combined with the super-peer backbone protocol, can generate impressive levels of performance and scalability.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my gratitude to my supervisor, Dr. Son Vuong, for his wonderful guidance, inspiration and encouragement. Without him, it would have been impossible to complete this thesis. I am also grateful to Dr. Alan Hu for being my second reader and for his useful comments that have improved this thesis.

I gratefully acknowledge the financial support from the Computer Science Department at UBC in the forms of TA and RA as well as the support from the SCE project funded jointly by the CITR Network of Centers of Excellence, the Advanced Systems Institute of BC, and the industrial Partners: Telus Mobility, Alcatel, Magnetas Games and Proface INDE.

I would also like to thank all my family members for their consistent support during my graduate studies at UBC.

JUAN LI

The University of British Columbia

August 2003

Chapter 1

Introduction

Prior to the remarkable rise of Napster [1], systems for sharing and exchanging information between computers were limited to the client-server model such as World Wide Web (WWW), Local Area Networks (LANs), and File Transfer Protocol (FTP) programs. Recently, however, more and more people and organizations are using the Internet; and more and more applications are using the network and consuming bandwidth. The system has expanded vastly beyond its original client-server design. Peer-to-peer (P2P) technology has begun to flourish just as this exponential growth has taken off, and its development is now recognized as one of the most important trends in the computer business today. Through direct exchange among peers, P2P technology enables efficient sharing of computer resources and services including information, files, processing cycles and storage. Because P2P systems distribute costs to all participating nodes, namely the resources required for sharing data, they have many advantages, such as adaptability, self-organization, fault-tolerance, load-balancing mechanisms, and the ability to utilize large amounts of resources. Recently, P2P has been subject to active research due to the impact of P2P applications on Internet traffic [12].

1.1 Motivation

One of the most challenging problems in P2P research is the difficulty of locating content in an efficient and scalable way. Particular content is located by accessing the node(s) that manage content when the names or attributes of the desired content are specified.

In very large networks, it is not always easy to find desired resources. For any given system, the efficiency of any search technique depends on the needs of the application. Currently, there are two types of P2P lookup services widely used for decentralized P2P systems [2]: *structured* searching mechanism and *unstructured* searching mechanism.

Structured systems such as Tapestry [4], Pastry [5], Chord [6], and CAN [7] are designed for applications running on well-organized networks, where availability and persistence can be guaranteed. In such systems, queries follow well-defined paths from a querying node to a destination node that holds the index entries pertaining to the query. These systems are scalable and efficient, and they guarantee that content can be located within a bounded number of hops. To achieve this performance level, the systems have to control data placement and topology tightly within their networks. However, this results in several limitations: first, they require stringent care in data placement and the development of network topology. Thus, the tools they use are not applicable to the typical Internet environment, where users are widely distributed and come from non-cooperating organizations. Second, these systems can only support *search-by-identifiers* and lack the flexibility of keyword searching, a useful operation for finding content without knowing the exact name of the object sought. Third, these systems offer only file level sharing, and do not share particular data from within the files.

Unstructured systems like Gnutella [2] and FastTrack [20] are designed more specifically for the heterogeneous Internet environment, where the nodes' persistence and availability are not guaranteed. Under these conditions, it is impossible to control data placement and to maintain strict constraints on network topology, as structured applications require. Currently, these systems are widely deployed in real life.

The present thesis focuses on building a P2P lookup application for integration into arbitrary dynamic networks that cannot be controlled. We thus concentrate on unstructured P2P systems, which support many desirable properties such as simplicity, robustness, low requirement for network topology and supporting keyword searching. Unstructured systems operate under a different set of constraints than those faced by techniques developed for structured systems. In unstructured systems, a query is answered by flooding the entire network and searching every node. Flooding on every request is clearly not scalable, and it has to be curtailed at some point; therefore it may fail to find content that is actually in the system. Furthermore, a network that uses flooding might be bombarded with excess messages and activity, and at certain points it might fail. To address these problems, we propose a hierarchical structure and an efficient routing strategy.

1.2 Thesis contributions

This thesis addresses two major deficiencies of unstructured P2P networks: limited scalability and inefficient search mechanisms. We propose an Efficient Clustered Super-Peer (ECSP) P2P model [27]. In this model, peers are grouped into clusters according to their topological proximity. Super-peers are selected from regular peers to act as cluster leaders, responsible for locating content and maintaining the network structure for client peers. Super-peers also connect to each other, to construct a backbone overlay network.

To scale the routing on overlay networks connecting super-peer nodes, we have designed an application-level broadcasting protocol, called Efa [28]. Efa's application is not only useful to the system analyzed in the current study, but rather it is also applicable to all large, unstructured, P2P networks on the Internet. Utilizing just a small amount of topology info, Efa is almost as simple as flooding, but it can be much more efficient. We have implemented and evaluated the architecture, and experimental results have further verified the effectiveness of our mechanism. The contributions of this thesis are as follows:

- We propose a cluster based super-peer system that groups peers according to their topological proximity. The introduction of a new level of hierarchy results in greater scales of query lookup and forwarding functionality, and increases stability, scalability and performance.
- We utilize a novel method to find the nearest neighbors, whereby we can generate networks according to topology metrics.

- We design an efficient application level broadcasting protocol, Efa, to perform routing on the super-peer backbone network. Efa is much more efficient than flooding.
- We introduce a novel super-peer redundancy to improve reliability and to decrease the possibility of single-point failure.
- We have implemented our mechanism and evaluated it, both with a real network environment and with simulation tools.

1.3 Thesis organization

This thesis is comprised of seven chapters. In Chapter 2, background information and related work in P2P networking are introduced. Chapter 3 describes the system design, which includes system components and the working processes of main components. Chapter 4 presents the Efa routing algorithm and protocols to be applied onto the super-peer overlay network. Details regarding implementation and decisions made for the prototype implementation are discussed in Chapter 5. Chapter 6 describes the experiment setup, along with performance evaluations and analysis. Chapter 7 concludes the thesis and discusses potential directions for future research.

Chapter 2

Background and related work

This chapter introduces background information on P2P technology, and previous research and techniques that are important to the present study.

2.1 Peer-to-Peer computing

In this section we give a brief introduction of P2P computing, including its definition, architecture and advantages.

2.1.1 Definition

The fundamental idea of organizing computers as peers is not new. The initial manifestation of the Internet, ARPANET, was a P2P system when it was originally conceived in the late 1960s. The goal of the original ARPANET was to share computing resources around the U.S. The first few hosts on the ARPANET (UCLA, SRI, UCSB, and the University of Utah) were all independent computing sites with equal status. The ARPANET connected them together not in a master-slave or client-server relationship, but rather as equal computing peers. As the years have passed, however, client-server architectures have become more prevalent because they have provided the fastest and most cost-effective means of supporting large numbers of non technical users. This has begun to change recently, as P2P has become a hot buzzword again.

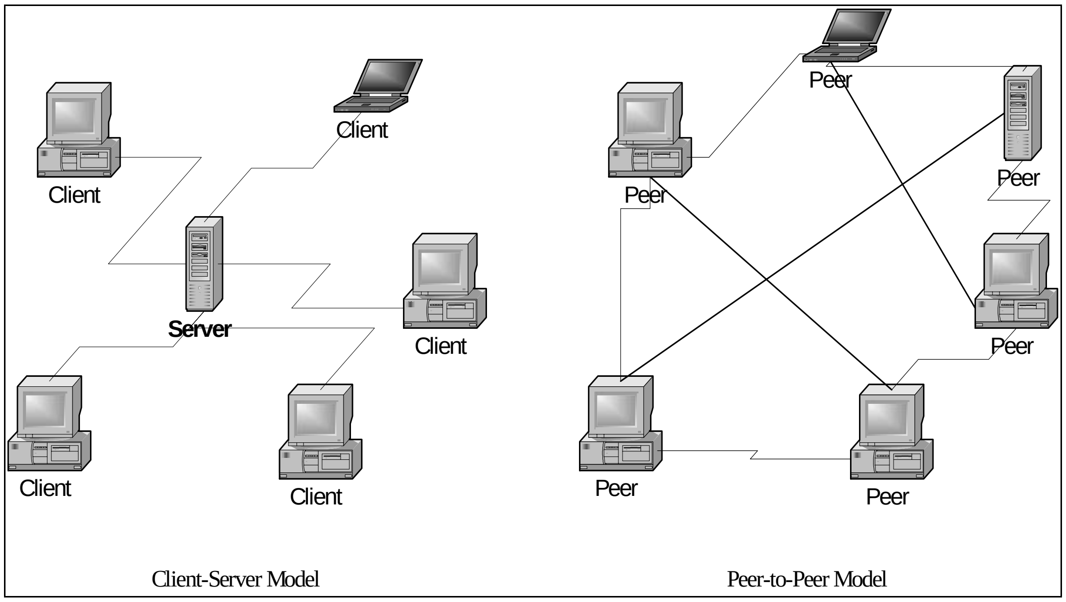


Figure 2-1 Peer-to-peer networks vs. client-server networks

As many new technologies, there is no single universally accepted definition for P2P computing. The Peer-to-Peer Working Group defines P2P computing as “sharing of computer resources and services by direct exchange between systems” [8]. Similarly, P2P can be defined as direct communication or collaboration between computers, where none act simply as a client or a server, but rather where all machines are equal peers. In a P2P system (Figure 2-1), every participating node acts both as a client and as a server, and reciprocates for its participation by providing other nodes access to some of its own resources and services, such as information, processing cycles, cache storage, and disk storage for files.

2.1.2 Architecture

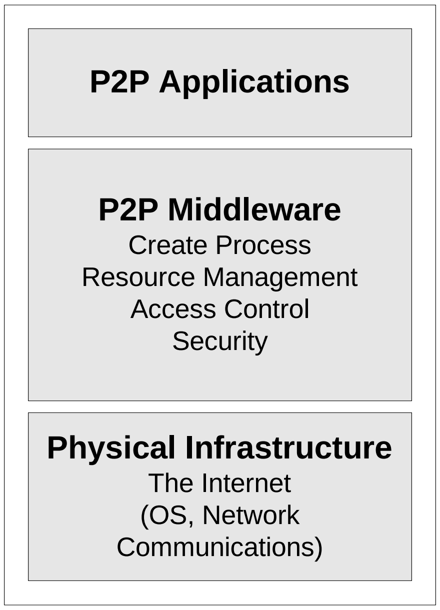


Figure 2-2 P2P architecture [22]

P2P architecture (Figure 2-2) is generally more sophisticated than client-server architecture due to the dual functionality embedded within it: a peer can behave as a client with the opportunity to switch to become a server and respond to incoming requests from other peers at any time. This is possible because of the additional middleware layer of software in P2P architecture. The middleware layer interfaces with both the physical layer of the network and the applications that are executed across the network. It supports process and resource management, access control, and security.

2.1.3 Advantages

P2P systems have many potential advantages, including the following [10]:

- Cost sharing and reduction. In a centralized system, servers that serve a large number of clients typically bear the majority of the costs of the system. In contrast, P2P architecture can spread costs over all the peers.
- Improved reliability. In P2P systems, the system burden is distributed among all participating peers, thereby increasing reliability and eliminating the chance of a single-point failure stalling the entire system.
- Resource aggregation and interoperability. Every peer in a P2P system shares certain resources, such as computing power and storage space, with other peers. Applications with P2P structures aggregate these resources to solve larger problems which cannot be resolved by the power of single peer.
- Increased autonomy. In P2P systems, users need not rely on any single centralized service provider, because local nodes can perform operations on behalf of their users. The most prominent examples of this are various file sharing systems, such as Napster, Gnutella, and FreeNet. In each case, users are able to obtain files that would not be available at any central server because of licensing restrictions. However, individuals autonomously running their own servers have been able to share files because they are more difficult to find than a server operator would be.
- Anonymity and privacy. In a central system, it is difficult to ensure anonymity because servers will typically be able to identify clients. By employing a P2P

structure in which activities are performed locally, users can avoid having to provide any information about themselves to anyone else in the network.

- Dynamism. P2P systems assume that computing environments are highly dynamic. More specifically, resources such as compute nodes enter and leave the system continuously. When applications are intended to support highly dynamic environments, the P2P approach is a natural fit.
- Enabling ad-hoc communication and collaboration. Dynamism is further related to the notion of supporting ad-hoc environments. By ad hoc, we mean environments where members come and go based perhaps on their current physical location or their current interests. Again, P2P is suitable for these applications because it naturally takes into account changes in the group of participants. P2P systems typically do not rely on established infrastructure — they build their own, as a logical overlay.

2.2 Related work

We can categorize existing P2P applications into three types of architecture [2]: *centralized* systems, *decentralized but structured* systems and *decentralized and unstructured* systems.

2.2.1 Centralized systems

This model is used by Napster [1]. Napster became famous as a music exchange system and was sentenced to go out of business due to copyright issues. Technically speaking, Napster is a pretty simple P2P system. The central server in Napster maintains directories of shared files stored at each registered user currently on the network. To retrieve files, users send requests to the central server. The central server then searches its database of files shared by other users and creates lists of files matching the search criteria, and the lists are sent back to the users. Users can then select desired files from the lists and open direct connections to other users. Files are transferred directly from one peer to another peer and not stored in the central server, so the central server only holds user information and directory information of shared files. Thus Napster is not a pure P2P system, but rather a combination of client-server and P2P functions. The principal advantage of the client-server architecture is the central index that locates files quickly and efficiently. Furthermore, because all clients have to be registered with the network, search requests reach all logged on clients to ensure that searches are as thorough as possible. However, the central server system involves a potential single point of failure on the network: when millions of nodes connect to the network, the centralized server might get overloaded, and there is also a possibility of clients receiving outdated information because the central server index is only updated periodically.

2.2.2 Decentralized but structured systems

Decentralized but structured systems have no central directory server, however they involve an organized structure. Several research groups have promoted such systems that support distributed hash table (DHT) functionality. Among them are Tapestry [4], Pastry [5], Chord [6], and Content Addressable Networks (CAN) [7]. Each of these systems uses different routing algorithms and implementation details, but the concepts inherent within them are quite similar: content is placed and retrieved according to strict rules. These systems use distributed indexing schemes based on hashing to locate content.

Basically, each participating node ID and published content name is hashed to a key. A content item is stored in the node whose hashing key is closest to the content's key, and query routing processes forward queries to neighboring nodes whose hashing keys are closer and closer to a query object's key. Below, we give a brief explanation of some typical systems that apply this model.

Tapestry [4] uses an algorithm inspired by Plaxton, Rajamaran and Richa [12], but it augments their model, which is intended for static environments, and adapts it to a dynamic node population. Tapestry maintains *O(log n)* neighbors and routes with path lengths of *O(log n)* steps.

In Pastry [5], Nodes are connected using a hypercube topology, in which adjacent nodes share some common address prefixes. Pastry routes any message concerning a given File ID toward the node whose Node ID is numerically closest among all live nodes. A set of |L| closest nodes creates the Leaf Set L, and routing can be achieved with this leaf set. To achieve more efficient routing, Pastry has another set of neighbors spread out in the key space. Routing consists of forwarding the query to the neighboring node that has the longest shared prefix with the key. Pastry has *O(log n)* neighbors and routes within *O(log n)* hops.

Chord [6] organizes hosts in a ring, and each Chord node maintains information about *O(log n)* other nodes in the finger table, allowing the ring to be quickly traversed. Data location in Chord is done by associating a key with each data item, and storing the key/data item pair at the node to which the key maps. Chord routes a key through a sequence of *O (log n)* other nodes toward a desired destination.

CAN [7] is organized into a *d*-dimensional hypercube. Each physical node is mapped to a coordinate point in the *d*-dimensional space via *d* hashing functions. Two physical nodes are neighbors if both coordinates are only different in one dimension. Queries are moved from one node towards one of its neighbor nodes, which has the numerically closest coordinate to the requested key. The lookup cost of CAN is $O(dn^{1/d})$.

All the above systems guarantee that content is retrieved within a bounded number of steps, thus greatly improving the efficiency and scalability of the P2P systems using these technologies. However, these systems require tight controls of data placement and topology within the network, they thus lack the flexibility that unstructured systems offer, and they are not applicable to the Internet environment. Also, they do not support keyword searching, which is important when users do not know exact file names in advance of their search. Although discussions of highly structured systems are quite prevalent in academic research, there is no real deployment at present and therefore only a small amount of measurement information is currently available for understanding the usability and scalability of such systems.

2.2.3 Decentralized and unstructured systems

Decentralized unstructured systems have no central server to keep track of files shared on a network, and have no control over network topology or content placement. Gnutella [3] is a notable example of this type of system.

The Gnutella file sharing protocol was designed by Nullsoft, a subsidiary of America Online (AOL). It was published on the Nullsoft web server by the developer, and was halted by AOL management shortly after the protocol was made available to the public. Up for only a few hours, several thousand downloads occurred. The protocol was reverse engineered and soon afterwards third-party clients were available and Gnutella had entered its deployment stage. The Gnutella system is a pure decentralized file sharing system. To coordinate communication among member nodes, the Gnutella protocol uses five descriptors: Ping, Pong, Query, Query-Hit and Push, as shown in Table 2-1

.

Table 2-1 Message types in the Gnutella network [21]

Type	Description	Contained Information
Ping	Announce availability and probe for other servents	None
Pong	Response to a ping	IP address and port# of responding servent; number and total kb of files shared
Query	Search request	Minimum network bandwidth of responding 'servent'; search criteria
QueryHit	Returned by servents that have the requested file	IP address, port# and network bandwidth of responding servent; number of results and result set
Push	File download requests for servents behind a firewall	Servent identifier; index of requested file; IP address and port to send file to

When joining the Gnutella network, each member node sends a Ping message that announces its presence on the network. After another member node receives a Ping message, it responds by sending a Pong message back to the initiator and forwards a copy of the Ping message to other member nodes. Nodes in the Gnutella network keep information about the ping messages in the routing table, while the TTL field of the ping message ensures that the message does not propagate itself infinitely. The Gnutella Query function enables a requesting member node to search for data files on other member nodes, on the basis of specified content. Queries are routed in the same way as with a Ping: when a member node finds a match, it responds to the requesting member node by sending a Query-Hit packet comprised of the IP address of the target member node, its port number and the data file name. A QueryHit message takes the Query's route back to the member node initiating the search, on receipt of which the searching node can download the desired data file.

Gnutella has many attractions: the query hit rates are reasonably high, the system is fault-tolerant towards failures of member nodes, and it adapts well in dynamically changing networks. From users' perspectives, Gnutella is a simple yet effective protocol. However, from a networking perspective, these attractions come at the price of very high bandwidth consumption, because search queries are flooded over the network and each node receiving a search request scans its local database for possible hits.

In summation, many papers have been written on P2P networks, and further research is ongoing, addressing different aspects and different problems that P2P networks are currently facing. Each model has its own advantages and disadvantages, but in view of the defectiveness of existing systems, a new architecture is needed. In the currently proposed cluster-based super-peer model, we have tried to address some major issues of P2P networks and to provide a solution for those problems.

Chapter 3

System design

In this chapter, we present the design of our Efficient Clustered Super-Peer (ECSP) P2P model, aiming at resolving the scalability and efficient query problems faced by unstructured P2P systems. ECSP follows a hierarchical approach; it allows *pure* P2P functions independent of infrastructure, while it provides advantages in scalability and search speed convergence. This chapter explains the details of the system design.

3.1 Multi-tier architecture

In a network, participating peers exhibit considerable heterogeneity in terms of storage capacity, processing power, bandwidth and online availability. For the best design, we should take advantage of this heterogeneity and assign greater responsibility to the peers that are capable of handling it. ECSP utilizes these differences in a hierarchical P2P design, in which peers with different capabilities take different roles. Specifically, peers in the system act as client peers and super-peers in different hierarchies.

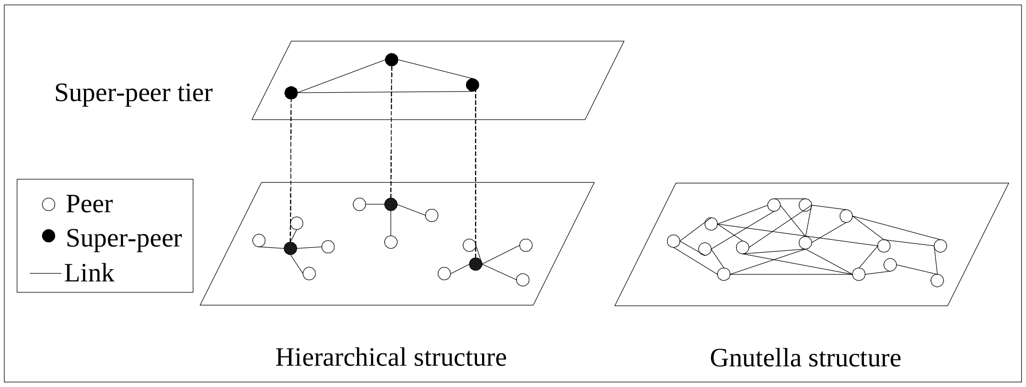


Figure 3-1 Hierarchical structure vs. Gnutella structure

Figure 3-1 illustrates the hierarchical structure, in which peers are grouped together if they are topologically close. Peers with more *resources* in the cluster can be selected as a super-peer. Super-peers act as local search hubs, building indices of the content files shared by each peer connected to them, and proxying search requests on behalf of these peers. Desirable properties for super-peers include accessibility to other peers, bandwidth and processing capacity. Super-peers with these characteristics are connected with each other and organized amongst themselves into a backbone overlay network on the *super-peer tier*. Then, an application level broadcasting protocol is designed to perform distributed lookup services on top of this overlay network. A unique well-known registration server is responsible for maintaining user registrations, logging users into the system, and bootstrapping the peer discovery process.

The hierarchical structure of this system combines advantages of both centralized and pure P2P systems: it combines the efficiency of a centralized search with the autonomy, load balancing and robustness provided by distributed search mechanisms. Compared with centralized systems, the hierarchical structure distributes the load on the central server to many super-peers; therefore no single super-peer is required to handle a very large load, nor will one peer become a bottleneck or a point of failure for the entire system. Compared with pure decentralized system like Gnutella (Figure3-1), the hierarchical structure reduces the query traffic because only super-peers participate in searching and routing. Simultaneously, more nodes can be searched because each super node proxies for many regular nodes. Therefore, the introduction of a new level in the system hierarchy increases the scale and speed of query lookup and forwarding processes. Moreover, the hierarchical structure is more stable because clusters join and leave the network less frequently than individual peers. Finally, our super-peer overlay routing protocol reduces the workload of super-peers significantly by avoiding many flooding duplications.

3.2 Main components

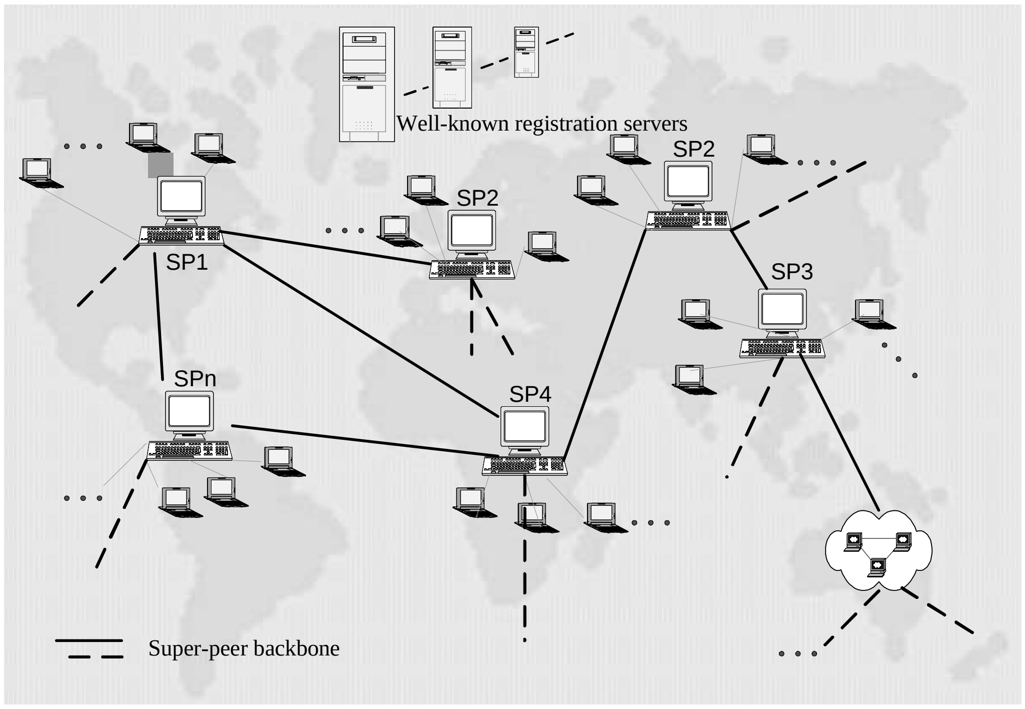


Figure 3-2 System architecture

As Figure 3-2 demonstrates, the system is composed of three main components: the well-known registration server, the super-peer, and the client peer. In this section, we explain these components in detail.

3.2.1 Well-known server

In the networks analyzed for the current study, well-known registration servers (Figure 3-3) supply yellow page services to all nodes in a network. Registration servers maintain databases of all active super-peers in the system, and when a new super-peer is added to the network, a new entry is generated in the registration server's super-peer database. The registration server then sends a neighbor list to the super-peer, which includes a set of super-peers already in the system, and the new super-peer can join the network by connecting to these neighbors. Neighbors in the neighbor list are not chosen randomly, but rather they are the nodes that are topologically closest to the new neighbor. Whenever a new client peer joins the system, it first contacts the registration server to get a super-peer list and to identify the super-peers located closest to it topologically. The algorithm of finding nearest peers will be described in section 3.3. Basing the network relationships on topological proximity in this manner can reduce network load for queries and responses. To provide scalability and load balancing, some hierarchical registration servers are essential, which contain replicas of the active registration server. Replica registration servers become active only when the main registration server is not able to provide service to nodes in the system, for example during busy periods or failing times.

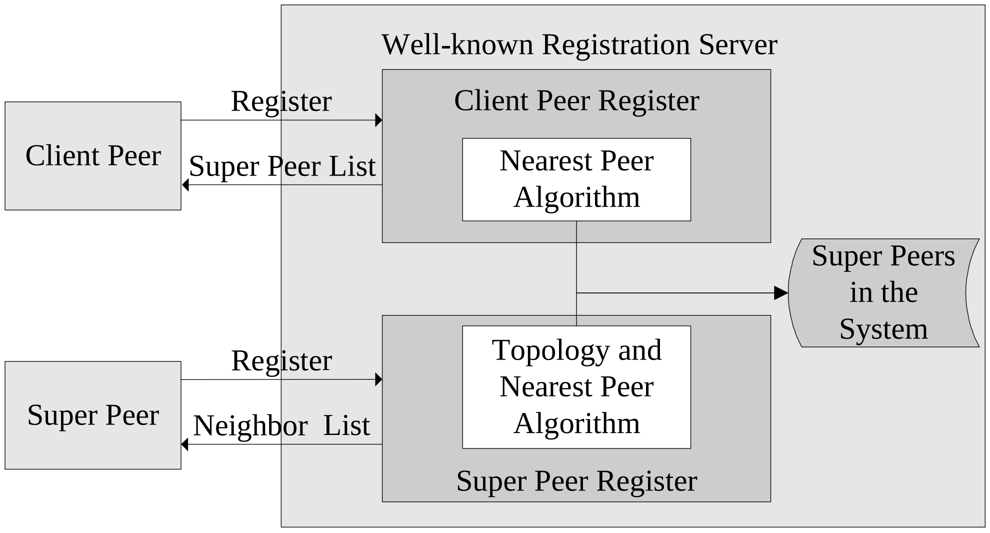


Figure 3-3 Well-known registration server structure

3.2.2 Super-peers

Super-peers are selected from regular peers according to their computing resources and bandwidth capabilities, the volume of files they store, and the behavior of being seldom offline. Super-peers act as cluster leaders and service providers for a subset of client peers, providing four basic services to the clients: *join*, *update*, *leave* and *query*.

After obtaining a super-peer list from a well-known registration server, client peers choose one super-peer from the list and connect to it. In the *join* process, client peers upload metadata describing the property of the content they will share with the network. In addition, the super-peer also stores details related to the client peer's connection, such as the IP address, bandwidth, and processing power of the client. For example, these details may include the use of T1, T2, cable or modem by a client to connect to the network. After the *join* process is completed, the client peer is ready to query content in the network, and to allow other client peers to download content from it.

When a client peer leaves the system, the super-peer removes that client peer's metadata from the index library. If a client peer ever updates its content data, it sends an *update* message to the super-peer, and the super-peer updates its index accordingly. When a super-peer receives a query from its client peer, it matches what is in its index library and forwards the query to its neighbors, who in turn forward it to some of their neighbors, according to the super-peer overlay network routing algorithm Efa. After results (or time-outs and error messages) are received from all of its neighbors, the super-peer sends the aggregated result to the requesting client peer.

As mentioned, super-peers are also connected with each other to form an application-level overlay network. The dynamic maintenance of the topology, and the efficient locating of content within this overlay network is described in the next chapter. Here, we note that super-peers are not only cluster leaders for their client peers, but also members of the super-peer overlay network. Therefore they supply interfaces to both client peers and to adjacent neighbor super-peers (Figure 3-4).

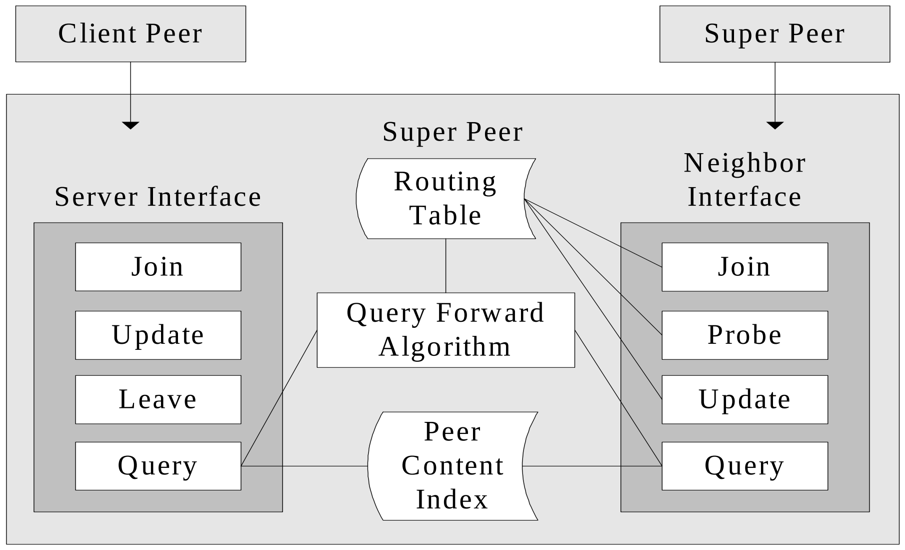


Figure 3-4 Super-peer structure

3.2.3 Client peers

In the present paper, regular peers are referred to as *client* peers to distinguish them from super-peers. In fact, they act as both clients and servers: they send requests to super-peers like clients, and receive other peers' file download requests like servers. While providing this functionality, client peers can offer easy-to-use interfaces, through which users can connect to the system, discover resources in the network and finally obtain the required content. To accomplish this, a client peer acts as both an FTP client and an FTP server. After the client peer joins the system and uploads its content metadata to its local super-peer, it initiates an FTP server on a well-known port and waits for other peers' download requests. After a client peer locates content through super-peers, it opens a connection and downloads directly from the node where the content is located. Figure 3-5 illustrates the client peer structure.

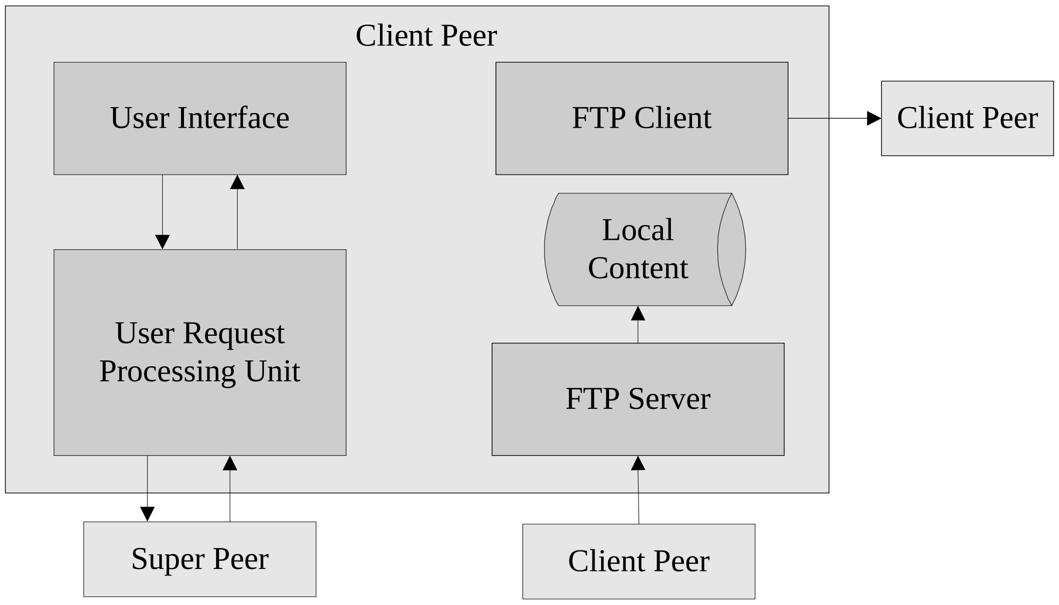


Figure 3-5 Client peer structure

3.2.4 Backup peers

The introduction of one more level of hierarchy makes the system more efficient, but the super-peer becomes a potential area of single-point failure for its cluster. When the super-peer fails or leaves the system, the entire cluster content index information is lost. To increase the reliability of the system, therefore, we introduce a backup peer as redundancy for the super-peer. Thus, every cluster has a super-peer acting as a cluster leader and a backup peer acting as a redundancy server. The backup peers are selected from the client peers too. They copy the super-peer's index table periodically, and when a super-peer fails or leaves the network, its backup peer replaces it and the cluster selects a new backup peer for redundancy. The possibility of both a super-peer and its backup peer failing simultaneously is much smaller than failure of the super-peer alone, and thus the introduction of a backup peer greatly improves a system's robustness. Furthermore, a backup peer is dynamically selected from client peers in the cluster, so there is no extra burden for the redundancy.

3.3 Exploiting locality

3.3.1 Motivation

It is beneficial to utilize physical locality and resources to increase network performance and to reduce the cost of sending messages across physical and data link layers. In effect, it is beneficial for peers to connect to super-peers that are near to them. This lowers the latency perceived by clients as well as the load for queries and responses. For the same reason, in the overlay network connecting super-peers, it is more efficient when neighbors on the overlay network are near to each other on the underlying topology. To make use of the locality, rather than randomly grouping peers, information in well-known registration servers is used to identify close neighbors among peers.

3.3.2 Algorithm

There are some existing methods to identify topological locality. A simple approach to identify closeness of peer location is to compare the first three bytes of their IP addresses. However, the correctness rate of this over-simplified approach is only about 50% [23]. Specifically, in the system used in the current study, the number of peers is large and therefore individually probing the whole set to find a near peer [14] is not a viable solution. In addition, the peers change dynamically, and any passive measurement based technique [15] may incur high error rates. Therefore, we need a method which can manage a large and dynamic network situation. After a comprehensive survey, an approach called Tiers [13] was selected for the system. Tiers scales to large application peer groups, and creates a hierarchy among peers, thereby providing an efficient and scalable solution to handle the complexity of the system.

To implement the algorithm of Tiers, the registration server maintains a hierarchy of nodes (in our case, super-peers). The hierarchy is based on the cluster of nodes: nearby nodes are grouped into a single cluster. (Here, 'cluster' has no relation with the 'cluster' of our P2P systems, it just means a group.) Each cluster has a cluster center, which has the shortest aggregate distance to all other nodes in the cluster, thus the center is an approximation of the location of all the cluster nodes.

In this process, first all the nodes are grouped into clusters at the leaf level, level *0*. Then, all cluster centers of the leaf level are grouped into clusters at level *1*.

Recursively, all cluster centers at level *i* are grouped into clusters at level *i+1*; in other words all nodes at level *i+1* are centers at level *i*. We set the cluster size between *k* to *2k-1*, where *k* is a constant. Therefore, nodes are eventually organized hierarchically as illustrated in Figure 3-6, which can be mapped to the tree structure in Figure 3-7.

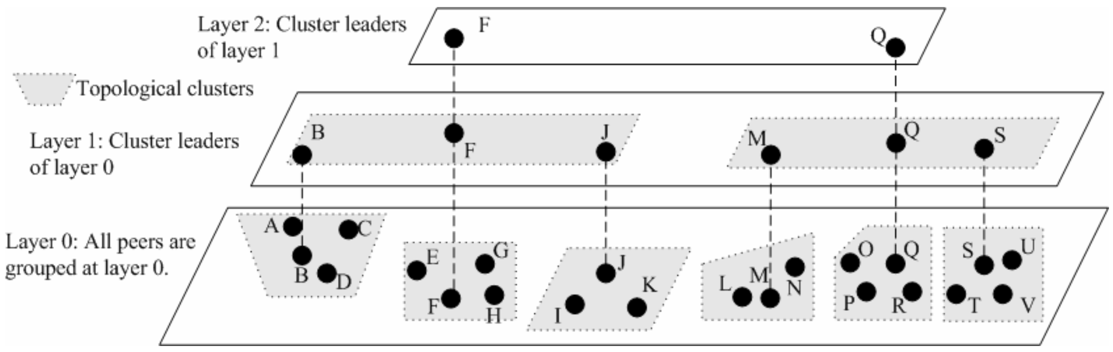

Layer 2: Cluster leaders of layer 1
Topological clusters
Layer 1: Cluster leaders of layer 0
Layer 0: All peers are grouped at layer 0.
F
Q
B
F
J
M
Q
S
A
C
B
D
E
G
F
H
J
I
K
L
M
N
O
Q
P
R
S
U
T
V

Figure 3-6 Hierarchical structure of nodes

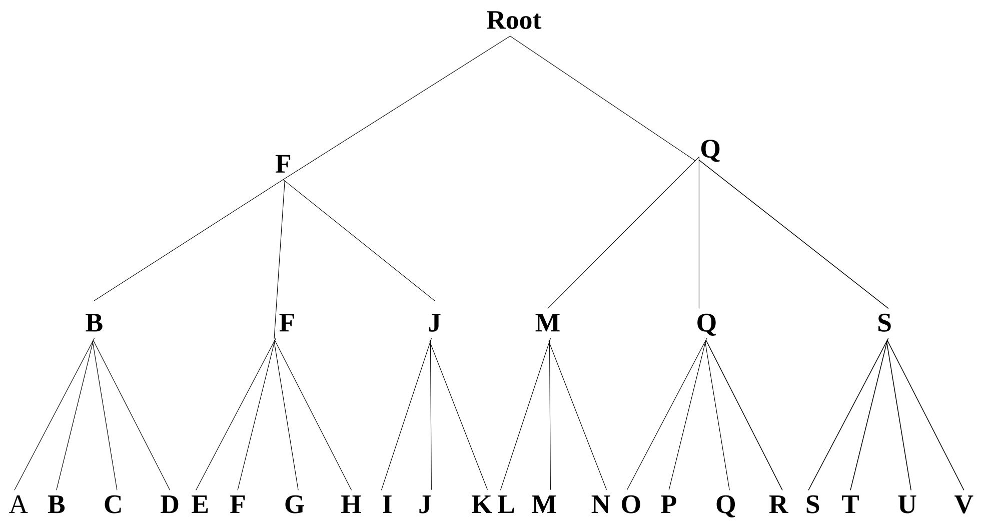


Figure 3-7 Tree corresponding to the node hierarchy

The algorithm operates from the top of the tree structure. For example, assume a new node *X* joins the system shown in Figure 3-7. It first contacts the root, in our case, the well-known registration server. The root will return its two children *F* and *Q* to *X*. Then *X* will probe these two nodes to see which one is nearer. Here we use Round Trip Time (RTT) as proximity metric. Suppose *X* finds that *Q* is nearest, then *X* will contact *Q*, and *Q* will return its three children *M, Q and S* to *X*. (In fact, except itself, only two nodes *M* and *S* are returned, because *X* already recognizes *Q*). *X* then finds that *S* is the nearest among these three. Similarly, *S* will return *U, V, T* to *X*, and at last *X* will find the nearest peer from these four nodes.

3.3.3 Algorithm analysis

As the example shows, instead of probing the whole network, the new node just requires contacting a small set of the network to identify its nearest neighbor. The size of each cluster is between *k* and *2k-1*, so the process at each level requires *O(k)* communication time. Furthermore, the height of the tree is bounded to *O(log N)*, and therefore the overhead of finding the nearest peer is *O(k*log N)*. Compared with probing every node in the system, the algorithm saves time and reduces traffic.

3.4 System operation process

This section explains the system operation process: how a client peer or a super-peer joins the system, how a user query can be checked and forwarded, and how the super-peer overlay network can be maintained and adapted to changing topologies.

Figure 3-8 illustrates the process when a client peer joins the system. To join the system, the client peer first contacts a well-known registration server (1), and the well-known registration server checks its super-peer database (2,3) to identify super-peers that are near the client peer and returns the super-peer list to the client peer (4). After receiving the super-peer list, the client peer selects and connects to one super-peer (5), and if the client peer receives a positive response from the super-peer (6) it uploads its content metadata to the super-peer (7). The super-peer saves the new peer's metadata to its index library (8) and sends a response to the client peer (9), which includes the address of the current backup peer in case the super-peer ever fails.

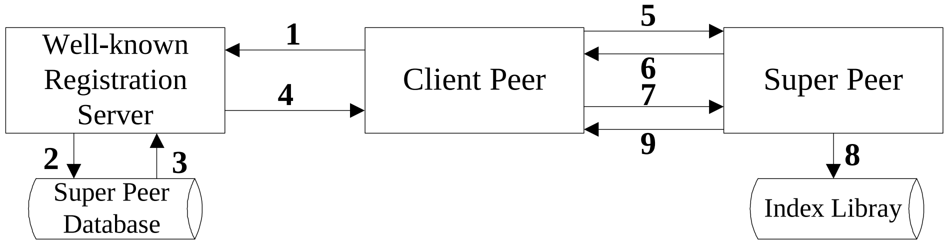


Figure 3-8 Client peer joining process

Figure 3-9 shows how a super-peer joins the system. It also contacts the well-known registration server first (1), the server saves the super-peer's profile information such as its IP address, bandwidth and geographic location into its super-peer database (2), and the server retrieves neighbor super-peers from the database according to their proximity to the new super-peer (3). Then the well-known server returns the neighbor list to the requesting super-peer (4). After receiving the neighbor list, the super-peer contacts each neighbor in the list (5), and these neighbors return their topology information to this super-peer (6), enabling the new super-peer to construct its routing table.

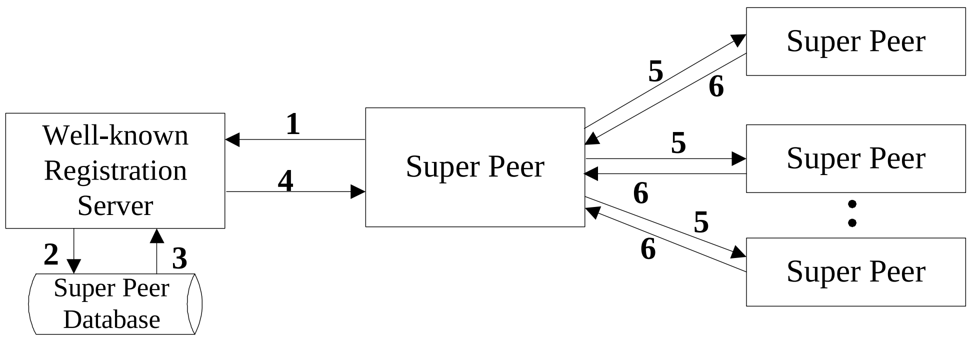


Figure 3-9 Super-peer joining process

Once a group of super-peers have been connected with each other into an overlay network, every super-peer has a local directory pointing to locally managed resources, in addition to a routing table. The algorithm for computing the routing table is explained in the next chapter. A query process is explained in Figure 3-10, where a client peer initiates a query and sends the query request to its local super-peer (1). Query messages are given Time to Live (TTL) that specifies the maximal super-peer hop steps the message may travel. The local super-peer searches the query in its index library (2), while at the same time it forwards the query to its entire neighbor super-peers (3). When a neighbor super-peer receives the forward query, it first checks its own index library to match the query (4), and decrements the TTL. If the TTL is greater than 0, it checks its routing table to obtain a forwarding list (5), then it forwards the query to neighbors in the list (6), and when it has collected all responses from the neighbors to whom it forward the query (7), it combines the local results with the results from its neighbors and returns the final result to the neighbor who sent the initial query to it (8). When the local super-peer receives responses from all of its neighbors, it returns the combined result to the client peer (9). In this query forwarding process, the lookup and forwarding processes are recursively operated on all nodes involved.

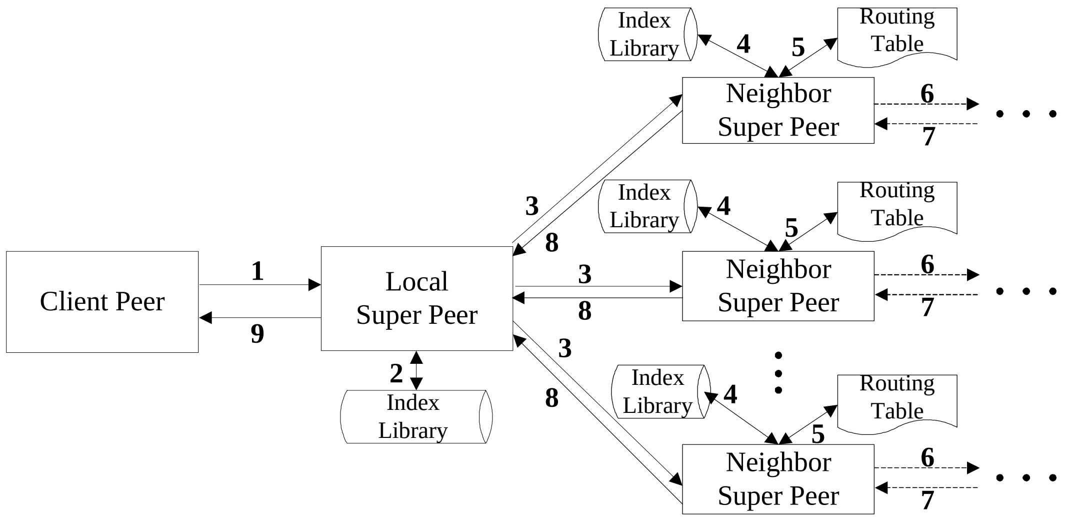


Figure 3-10 Query process

In our model, the response message is forwarded back along the reverse path of the query message, which ultimately leads to the source super-peer. Another option is letting each responder open a connection to the source, to transfer the results directly. While the first method uses more aggregate bandwidth than the second, it will not overwhelm the source with connection requests, as will the second method, and it provides additional benefits such as anonymity. Hence, we use the first method in our implementation.

Adapting to dynamically changing topologies and routing over super-peer overlay networks involve important challenges. We have designed a routing protocol, Efa, to deal with the query routing difficulties and to maintain the super-peer overlay network. The routing algorithm and protocol are illustrated at length in Chapter 4.

Chapter 4

Overlay routing

As mentioned in the previous chapter, super-peers are connected to form a backbone overlay network. Every super-peer acts as a router and forwarder for messages sent by other super-peers. This obviously involves issues of cost and scalability within the system, and therefore the design of the routing protocol on top of the overlay network is very important. In this chapter, we describe a constrained flooding algorithm, Efa, that addresses this problem. Efa's application can be useful not only to this super-peer overlay network, but also to other unstructured P2P and ad hoc networks.

4.1 Algorithm motivation

Although DHT-based (distributed hash table) structured systems scale well and perform efficiently, they are not applicable to typical Internet environments, where users are widely distributed and come from non-cooperating organizations. Node persistence and availability are not guaranteed or necessary, and users are not often willing to store large amounts of data for other unknown users. It is almost impossible to control data placement and network topology strictly. In addition, most users desire support from the system for richer queries, beyond the searches for identifiers that are offered by structured systems. Therefore the present thesis focuses on unstructured systems.

In unstructured P2P systems, no clue emerges about where content is placed, and therefore queries have to be flooded through a network to obtain results. Flooding introduces a lot of duplicated queries, particularly in highly connected networks. These duplicated queries are pure overhead; they incur extra work to deal with the queries at the receiving nodes, but they do not contribute to any increased chance of finding desired content. Most flooding-based applications use TTL to control the number of hops a query propagates, and add duplication detection mechanisms to avoid duplicated query messages. However, TTL can only help reduce the radius of the query coverage; it cannot decrease duplication within the coverage. At the same time, simple duplication detection can only avoid a very small proportion of duplications, and the network may still be overwhelmed by large quantities of duplicated messages.

Therefore, our algorithm aims at suppressing flooding by reducing the number of duplicated query messages. There are many approaches to eliminating flooding, the most popular of which uses tree-based broadcasting. In our model, the number of participant nodes can be quite large and users are widely distributed all over the Internet. Therefore it is impossible to let every node know the whole topology of the network. In addition, all tree-based approaches require huge messaging overhead, associated with construction and maintenance of the spanning tree. However, in most P2P systems, participant nodes are typically PCs at homes or offices with their own tasks, and thus they cannot afford many resources for P2P applications. In addition, they can be very dynamic, so messages updating tree structures overwhelm the network. In light of these considerations, our objective is to use limited topology information and simple computing to decrease the duplication queries created by flooding.

4.2 Algorithm description

In a well-connected network, several different paths may exist to connect two particular nodes, which is the reason that extensive duplications may be created by flooding. If node *v* can anticipate that one of its neighbors *u*, receives query messages from another path, however, then *v* does not forward the query to *u*. To achieve this type of anticipating, we use a rule directing the nodes that duplicate and forward messages while we keep track of topology information to compute the forwarding set. As the later experiment shows, although we cannot avoid all duplications, we can reduce much duplication for most widely used network topologies.

The following definitions (Table 4-1) are used in the algorithm and discussed later in this chapter.

Table 4-1 Definitions used in the routing algorithm

Symbol	**Description**
v	Current node
id(v)	Node *v*'s unique id
N(v)	Neighbor set of *v*
NN(v)	Neighbor's neighbor set of *v*
fr(u,v)	*v* is the current node, *u* is the node which forwards the query to *v*. *fr(u,v)* is the forward reaching set of *u* for the current node *v*, i.e. the immediate (no more than 2 hops away) set of nodes reached by the local flooding source *u*.
routing(u,v)	For local source *u*, current node *v*'s routing set. For example, if *u* forwards the query package to *v*, the set of nodes *v* forwards is decided by *routing (u, v)*

The algorithm in Figure 4-1 is used to compute *fr(u,v)* and *routing(u,v)*.

```
fr(u,v) = N(u) ∪ { all v' in NN(u) | id(v') < id(v)}

routing(u,v) = all v' in N(v), such that

        1.  v' ∉ fr(u,v) AND
        2.  {N(v') ∩ fr(u,v) = ∅} OR {N(v') ∩ fr(u,v) = A AND (∀v'' ∈ A AND id(v'') > id(v)) }
```

Figure 4-1 Algorithm to compute routing table

```
forward(u,v)

/*when node v receives forwarded query from its neighbor u, this algorithm decides how v forwards
this query */

If  the received query has been received before
        discard it
else
        if  u is null /* v is the node which initiates the query*/
                forward the query to N(v)
        else
                forward the query to routing(u,v)
```

Figure 4-2 Routing algorithm

The algorithm above in Figure 4.2 describes the routing process for the current node, *v*, when it receives a query from its neighbor, *u*.

We use an example to explain the algorithm. Figure 4-3 depicts a simple network topology, where *N5* is the current node. To compute *N5'*s routing table, we need to keep track of *N5*'s 2-hops neighbor info. In the implementation, we store the 2-hops topology info into a hash table, called a topology table,, as shown in Table 4.2. *fr(N(N5),N5)* is calculated with the algorithm described in Figure 4-1 and the results are listed in Table 4-3, which is useful to compute *N5*'s routing table. Finally, *N5*'s routing table is formed in Table 4-4. To compare Efa with flooding, we list table 4-5, which is what simple flooding actually used.

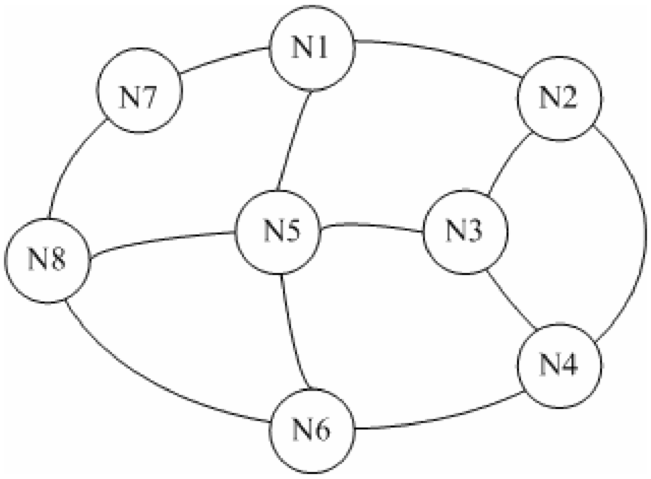


Figure 4-3 A simple network topology

Table 4-2 Topology table of N5

Neighbor *u*	***N(u)***	***NN(u)***
N1	*N2*	*N3, N4*
	N7	*N8*
N3	*N2*	*N1,N4*
	N4	*N2, N6*
N6	*N4*	*N2, N3*
	N8	*N5, N7*
N8	*N6*	*N4, N5*
	N7	*N1*

Table 4-3 Forward reaching sets of N5's neighbors

Neighbor *u*	***fr(u, N5)***
N1	*N2, N7, N3, N4*
N3	*N2, N4, N1,N 6*
N6	*N4, N8, N2, N3*
N8	*N6, N7*

Table 4-4 Efa routing table of N5

Neighbor *u*	***routing (u, N5)***
N1	*N8*
N3	*N8*
N6	*null*
N8	*N1,N3*

Table 4-5 Flooding routing table of N5

Neighbor *u*	***routing (u, N5)***
N1	*N3, N6, N8*
N3	*N1, N6,N 8*
N6	*N1, N3, N8*
N8	*N1, N3, N6*

For example, in the case of flooding, when *N5* receives message sent from *N1*, *N5* would forward the message to all of its neighbors except *N1*. Therefore, *N5* forwards the messages to *N3, N6*, and *N8*. However, if it uses Efa, *N5* does not need to forward the message to all of its neighbors, but only to those that may not be reached by *N1*. Messages from *N1* reach all neighbors of *N1*, which are *N2* and *N7*. Because *id(N2)* is smaller than *id(N5)*, the message also reaches *N2*'s neighbors, *N3* and *N4*. Finally, we get: *fr(N1,N5)={N2, N7, N3, N4}*. Therefore, when *N5* receives a message from *N1*, it does not forward the message to its neighbor *N3*, because *N3* is in the set *fr(N1, N5)*. *N5* does not forward the message to *N6* either, because *N6*'s neighbor *N4* is in *fr (N1, N5)*, and *id (N4)* is smaller than *id (N5)*. So at last, *N5* only forwards the message to *N8*.

4.3 Routing protocol

We have developed an application-level broadcasting protocol to maintain and route the backbone overlay network. We briefly describe the key components of the protocol herewith.

- Data structure: Every node maintains two tables: The first table is a topology table that stores the node's 2-hops topology information and is used to compute the node's routing table. The node creates and updates its topology table by exchanging information with its neighbors. The second table is a routing table, created and updated by utilizing the topology information from the topology table, according to the algorithm described in the last section.
- Join: The join process starts when a node joins the network. It first contacts nodes already in the network and generates a *Join* message containing its node *ID*, then it

pauses in a *waiting* status. When a node v in the network receives the *Join* message, it returns *N(v)* and *NN(v)* to the sender. The sender then adds this information to its topology table. When the node collects responses (including error messages and timeouts) from all nodes contacted, it creates its routing table according to the topology information collected.

- Probe: Periodically, nodes send "heart beat" messages to their neighbors to probe the connection between them and to update their topology tables accordingly.
- Update: A node sends its knowledge of local topology to its neighbors periodically, to keep all information up-to-date. For a node v, the update message includes *N(v)* and *NN(v)*. The receiving node can use this message to update its topology table, and later its routing table.
- Query: Queries are initiated by user applications, and forwarded to other nodes. When a node *v* receives a forwarded query from its neighbor *u*, it first checks its local content index; if there are not enough answers, it checks its routing table and forwards the query to set *routing (u,v)*.

4.4 Algorithm correctness

If a network is connected, the protocol described above guarantees that a query message is forwarded to all nodes in the network. For an arbitrary node *v*, which receives a forwarding query from its neighbor node *u*, the entry *(u,v)* in the routing table of *v* determines to which neighbors the query is forwarded. The entries in the routing table are computed according to the following principle: If *v*'s neighbor $x \in fr(u, v)$, then *v* need not forward the query to *x*, because *v* knows *x* has been reached by *u*. If $x \notin fr(u)$, but *x*'s neighbor $y \in fr(u)$, we compare the *ID* of *y* and *v*; if *v*'s ID is smaller, then *v* forwards the query to *u*, otherwise, v leaves the query for *y* that has a smaller *ID* to forward the query. Therefore, for an arbitrary neighbor of *v*, it would be reached either by *v* or by other nodes in *u*'s reaching set, whose *ID* is smaller than *v*'s. Consequently, all nodes in the network will be reached by the forwarding protocol.

4.5 Design improvement

The algorithm avoids some duplication by comparing the ID of the current node with the ID of forwarding candidate nodes and their neighbors. When a node can be reached through different paths, we always forward the message through nodes with smaller IDs, saving the traffic of forwarding the message from nodes with bigger IDs. This method decreases duplication, but it cannot generate other benefits. If we can use a weight or priority value instead of simply using node IDs to direct the query transmission path, we may improve network performance: the weight of a node can be defined according to the node's bandwidth and processing power, and when a node receives a message, it can preferentially forward the message to the node with higher weight. Therefore, a node with more bandwidth and processing power has higher priority for forwarding queries. The weight of a node can change dynamically with its current load and other factors. This change can be reflected to the routing table when nodes periodically update their routing tables. However, adding performance characteristics to algorithms requires even more data to be gathered, thus involving a tradeoff.

One major issue in unstructured systems is the challenge of selecting an appropriate TTL. The TTL decides the number of steps each query must travel. If a TTL is set too high, the query unnecessarily wanders in the network. If it is too small, a node might not locate content outside the TTL radius. An effective method for setting TTL is the Expanding Ring method developed by Cao [2]. According to Cao, expanding a ring reduces message overhead significantly compared with regular flooding with a fixed TTL.

Another important problem for P2P applications is the challenge of managing topological properties. Pangurangan, Raghavan and Upfal [24], Kommareddy, Shankar and Bhattacharjee [25], and Banerjee, Komareddy and Bhattacharjee [26] have proposed several methods to build networks with good topology properties. Combined with these methods and the techniques discussed above, our algorithm can greatly improve system performance and scalability.

Chapter 5

System implementation

This chapter presents implementation details of the ECSP framework, focusing on components and their interactions. The design presented in Chapter 3 is influenced by object oriented technology. It has been implemented in Java and Java RMI and has undergone some basic testing. The prototype system runs on Windows, Macintosh, Linux, and Sun. The system is composed of three main components: the well-known registration server, super-peer and client peer. The well-known registration server should run on a stable and robust computer, and it supplies registration services for super-peers and peers in the system. Super-peers are in charge of local cluster peer requests, such as uploads and queries. In addition, super-peers connect with each other, forming a backbone overlay network to allow client peers to share contents from throughout the system. Client software gives users tools that allow them to easily publish, search and download content in the network. In our implementation, we bind the super-peer and client peer interfaces together, because each client peer can be a potential super-peer. We also implement the routing protocol on top of the super-peer overlay network.

5.1 Deployment of Java RMI

5.1.1 RPC and RMI

One common means of communicating between processes in a distributed system is through a Remote Procedure Call (RPC).

RPC allows a thread of control in one process to call a function that runs in a thread of control in some other process, perhaps on a different machine. RPC is a high-level communication paradigm that allows programmers to write network applications using procedure calls that hide the details of the underlying network. Java Remote Method Invocation (RMI) is the object-oriented version of RPC. It essentially uses the same concept to allow programmers to transparently invoke methods on objects that reside on other computers. In this way, the object-oriented paradigm is preserved in distributed computing.

5.1.2 Interfaces

The design of an RMI remote object must include an interface. The major purpose of the interface is to describe a set of entry points that can be called from a remote process. The interface describes each entry point in terms of the information required, such as the types of arguments taken by the procedure to be called, and the information returned, such as values or output parameters of the procedure to be called. Interfaces are often specified using an interface definition language (IDL), which allows the specification of both entry points and data types that are passed (as parameters or return values) between communicating processes, much like a standard programming language.

The system has three kinds of services, represented by three interfaces: *InterfaceWellKnown, InterfaceServer,* and *InterfaceNeighbor.* The well-known server supplies an interface, called *InterfaceWellKnown,* for super-peers and peers to gain entry to the system. It supplies three methods for remote calling: *peerJoin(), spJoin()* and *spLeave()* (Figure 5-1). When regular peers join the system they call *peerJoin()* to gain entry to the system. *peerJoin()* requires the peer's IP address as input arguments, and it returns a vector, which is the super-peer list for the caller to connect to. Similarly, super-peers call *spJoin()* to register and gain entry to the system. *spJoin()* accepts one parameter *ipStr,* which is the IP address of the invoker. The well-known server adds the IP address to its database and returns the caller a neighbor list it can connect with. *spLeave()* can be invoked actively by a super-peer leaving the system or by a backup super-peer which observes the death of the current super-peer, so the well-known server can remove that super-peer from its database.

```
public interface InterfaceWellKnown extends Remote{
   Vector peerJoin(String ipStr) throws RemoteException;
   Vector spJoin(String ipStr) throws RemoteException;
   void spLeave(String ipStr) throws RemoteException;
}
```

Figure 5-1 The well-known server interface

Super-peers are servers for client peers, and neighbors of their adjacent super-peers in the backbone overlay network. Because of their dual-roles in the system, they have two interfaces: a server interface for client peers and a neighbor interface for neighbor super-peers. The server interface *InterfaceServer* (Figure 5-2) offers four methods for client peers. *clientUpload()* is invoked when client peers join the system and upload their sharing file metadata. It accepts two parameters, the client peer's IP address and the sharing files' metadata vector. *clientLeave()* is then called when a client leaves the system. *clientQuery()* deals with a client peer's query request, for which it requires two arguments and has a return value: the first argument is the client peer's IP address, and the second one is the query keyword and the return value is a result vector. *backupIndex()* is periodically called by the backup peer in the cluster, so it can get the current super-peer's local index library info. A super-peer also offers a neighbor interface for its neighbors in the backbone overlay network. The *InterfaceNeighbor* (Figure 5-3) has four methods for maintaining the overlay network's neighborhood relationship and for dealing with query lookup and forwarding tasks.

```
public interface InterfaceServer extends Remote{
String clientUpload(String ipStr, Vector vecFile) throws RemoteException;
void clientLeave(String ipStr) throws RemoteException;
Vector clientQuery(String ipStr, String strQuery) throws RemoteException;
Hashtable backupIndex(String ipStr) throws RemoteException;
}
```

Figure 5-2 Server interface

```
public interface InterfaceNeighbor extends Remote{
Vector neighborForward(String ipStr,String qId,String strQuery) throws RemoteException;
Hashtable neighborContact(String ipStr) throws RemoteException;
void neighborUpdate(String ipStr, Hashtable hashRouting) throws RemoteException;
void neighborLeave(String ipStr) throws RemoteException;
}
```

Figure 5-3 Neighbor interface

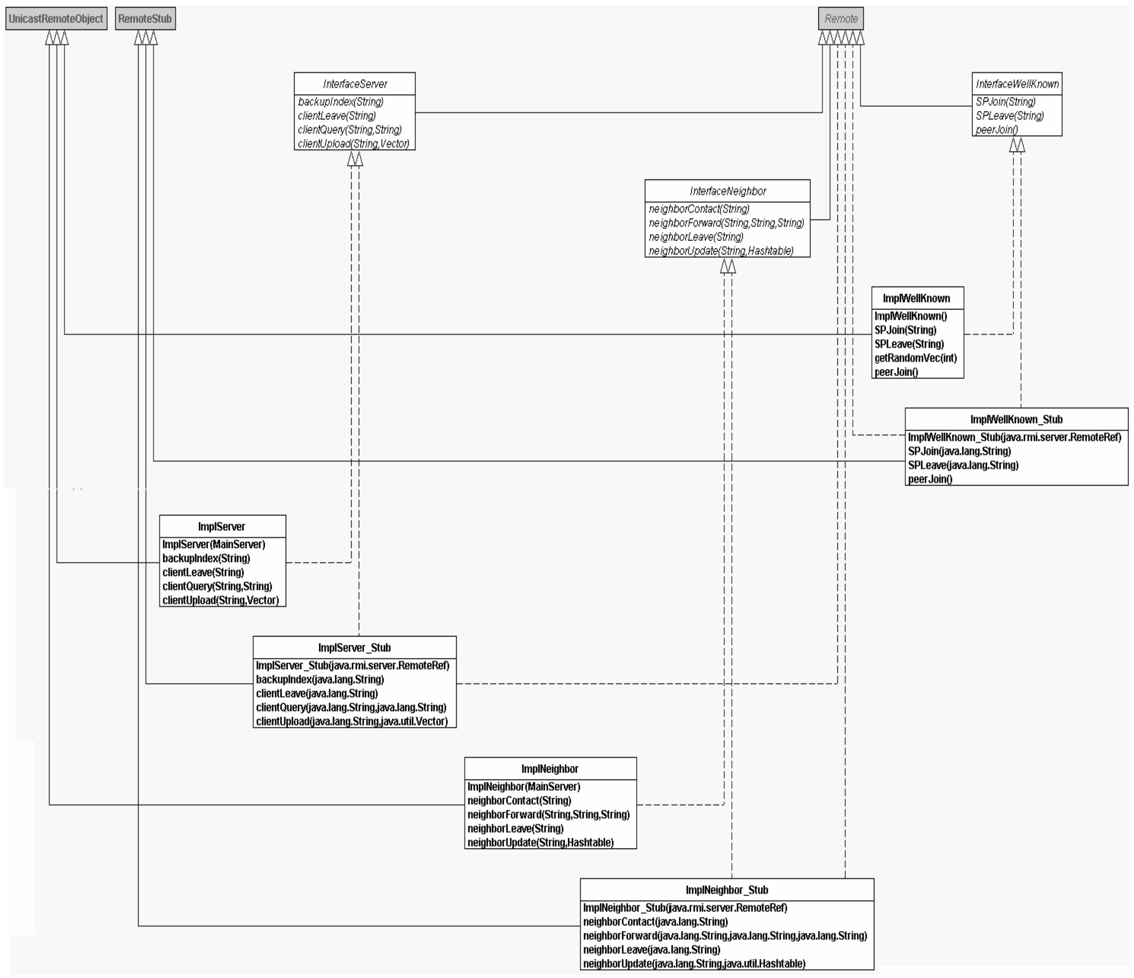

UnicastRemoteObject
RemoteStub
Remote
InterfaceServer
backupIndex(String)
clientLeave(String)
clientQuery(String,String)
clientUpload(String,Vector)
InterfaceWellKnown
SPJoin(String)
SPLeave(String)
peerJoin()
InterfaceNeighbor
neighborContact(String)
neighborForward(String,String,String)
neighborLeave(String)
neighborUpdate(String,Hashtable)
ImplWellKnown
ImplWellKnown()
SPJoin(String)
SPLeave(String)
getRandomVec(int)
peerJoin()
ImplWellKnown_Stub
ImplWellKnown_Stub(java.rmi.server.RemoteRef)
SPJoin(java.lang.String)
SPLeave(java.lang.String)
peerJoin()
ImplServer
ImplServer(MainServer)
backupIndex(String)
clientLeave(String)
clientQuery(String,String)
clientUpload(String,Vector)
ImplServer_Stub
ImplServer_Stub(java.rmi.server.RemoteRef)
backupIndex(java.lang.String)
clientLeave(java.lang.String)
clientQuery(java.lang.String,java.lang.String)
clientUpload(java.lang.String,java.util.Vector)
ImplNeighbor
ImplNeighbor(MainServer)
neighborContact(String)
neighborForward(String,String,String)
neighborLeave(String)
neighborUpdate(String,Hashtable)
ImplNeighbor_Stub
ImplNeighbor_Stub(java.rmi.server.RemoteRef)
neighborContact(java.lang.String)
neighborForward(java.lang.String,java.lang.String,java.lang.String)
neighborLeave(java.lang.String)
neighborUpdate(java.lang.String,java.util.Hashtable)

Figure 5-4 Interface and implementation

The interfaces simply describe the service of remote objects. The actual realization of the services is accomplished through implementation features. Figure 5-4 provides a diagram of interfaces and implementation classes.

5.1.3 RMI communication processes

To realize a transparent communication channel between distributed systems, RMI utilizes a registration mechanism: an RMI Registry is needed to manage remote references. When a server wants to publish particular services to remote objects, it registers the service objects to a local registry service. The registry service listens on a well-known socket and waits for remote object's lookup. When a remote object connects to the registry and obtains a remote reference, it is possible to access that object in the same way as local objects.

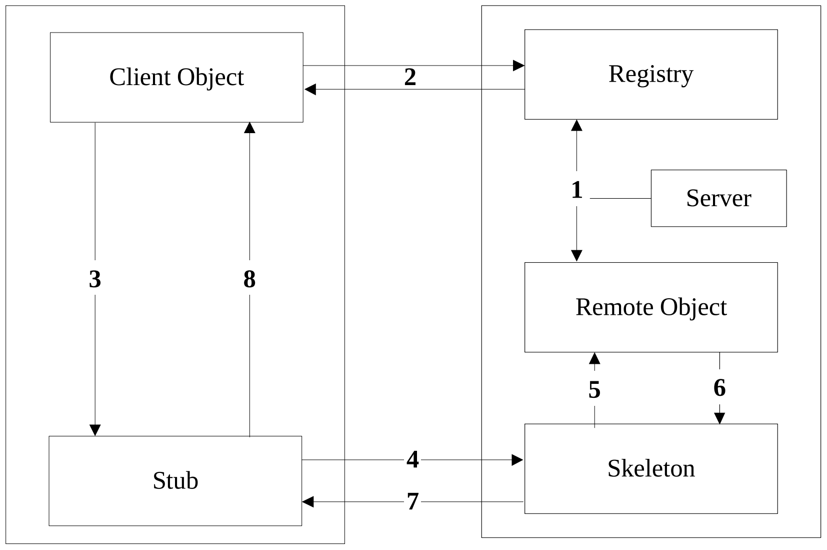


Figure 5-5 RMI communication processes [16]

The general Java RMI architecture is depicted in Figure 5-5. First a server creates a remote object and registers it to a local Registry (1). The client then connects to the remote Registry (2) and obtains the remote reference. At this point, a stub of the remote object is transferred from the remote virtual machine to the client virtual machine, if the stub is not yet present. When the client invokes a method at a remote object (3), the method is actually invoked at the local stub. The stub marshals the parameters and sends a message (4) to the associated skeleton on the server side. The skeleton unmarshals the parameters and invokes the appropriate method (5). The remote object executes the method and passes the return value back to the skeleton (6), which marshals it and sends a message to the associated stub on the client side (7). Finally the stub unmarshals the return value and passes it to the client (8).

5.2 System model

5.2.1 Core classes

There are eight core classes that provide important functionality to the system framework, listed below in alphabetical order. Any names to the right of the class names denote super classes and directly implemented interfaces.

BackupTask TimerTask

Used by backup peers to periodically backup the index library of the active super-peer.

FtpServerThread Thread

Running as an FTP server at the client peer, listening on a well-known port for downloading requests.

HashIndex

Maintaining the index library of the super-peer, supplying functions to add, retrieve update and delete indexes.

ImpWellKnown	InterfaceWellKnown	Implement the service of the well-known registration server.
ImpSever	InterfaceServer	Implement the service of super-peers to client peers.
ImpNeighbor	InterfaceNeighbor	Implement the service of super-peers to neighbor super-peers.
MainFrame	JFrame	The GUI of super/client peers.
MainServer	Thread	Running as the super-peer, it registers and starts the service, and serves requests from client peers and neighbor super-peers.
MainClient		The client peer, dealing with user requests and invoking the corresponding services of local super peers.
PanelGraph	JPanel	Drawing super-peer local topology information.
QueryRecord		Managing query requests, by checking local index libraries, forwarding results to neighbors, collecting and aggregating results and returning the results to query requesters.
Routing		Dealing with query routing: it creates and updates the routing table, and chooses routes according to routing table.
ThreadPool	Terminatable	

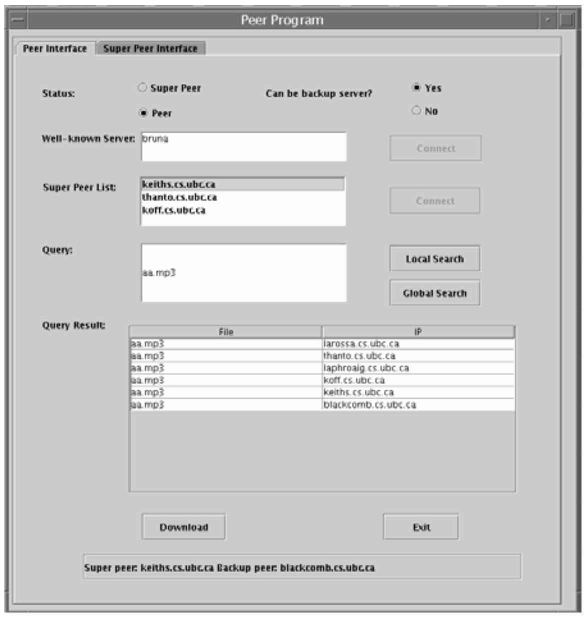
It implement a pool of threads that execute tasks assigned to it, all implementing the Runnable interface.

UpdateTask	TimerTask
Used by super-peers to periodically send their topology information to all of their neighbors.	

All of these components in the system cooperate with each other to realize the complete functionality of the system. Figure 5-9 identifies the main components and their relations, and Figure 5-10 illustrates the main operation sequences of the system.

5.2.2 GUI

Users access the system through client software. Client software is GUI based, and it interfaces with other components using events triggered by buttons. Because every client peer can be a potential super-peer and each super-peer comes from client peers, we combine the super-peer and client peer functions together in the GUI. Figure 5-6 illustrates the user interface of the client software. Because it is a prototype implementation, the GUI is very simple. Users can choose if they want to be super-peers, peers or backup peers. After a peer registers to the well-known registration server, it receives a list of super-peers. Users can choose one super-peer from the list to connect to, and then the user can query content by submitting keywords into the searching box. The system supports two types of searching: local searches and global searches. Local searches only search the content in the local cluster. It is very fast, but the results are limited. Global searches, on the other hand, study the entire network, so the user can get a complete result, but this may take longer time. Users choose each particular searching mode according to their requirements. The query results are shown in the results table, and users can choose one peer from which to download the content.



Status: Super Peer [✓ Peer]

Can be backup server? [✓ Yes] No

Well-known Server: bruna

Super Peer List:
keiths.cs.ubc.ca
thanto.cs.ubc.ca
koff.cs.ubc.ca

Query: aa.mp3

Query Result:

File	IP
aa.mp3	larossa.cs.ubc.ca
aa.mp3	thanto.cs.ubc.ca
aa.mp3	laphroaig.cs.ubc.ca
aa.mp3	koff.cs.ubc.ca
aa.mp3	keiths.cs.ubc.ca
aa.mp3	blackcomb.cs.ubc.ca

Super peer: keiths.cs.ubc.ca Backup peer: blackcomb.cs.ubc.ca

Figure 5-6 Client peer GUI

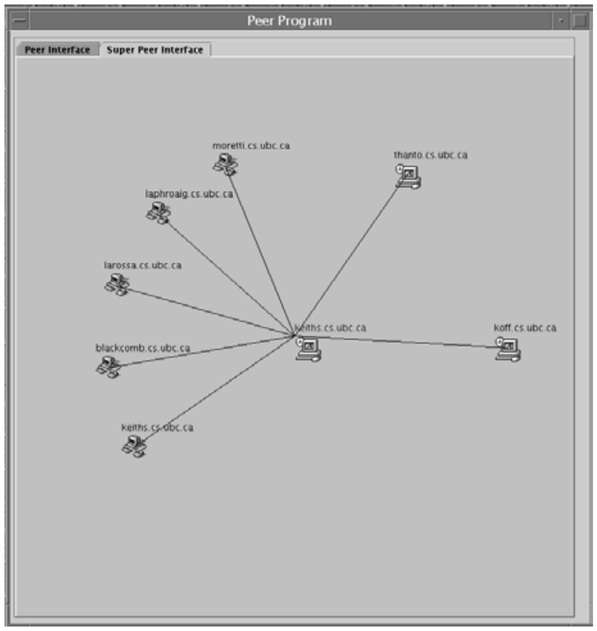


Figure 5-7 Super-peer local topology

To monitor the network connection, we display the super-peer topology information (Figure 5-7). This is only for our test convenience and we seldom see such interfaces in real P2P applications. The center of the topology graph is the local super-peer, with its client peers on the left and neighbor super-peers on the right.

5.3 Other implementation issues

5.3.1 Multithreads

It is often convenient and efficient to use multi-threads to deal with client requests. Therefore, sometimes people attempt to create unlimited threads to deal with every task. However, resources are always limited, and too many threads would exhaust system resources, and system performance will decrease quickly. On the other hand, insufficient numbers of threads may result in excessive queue growth and large latency in message delivery. Therefore, we should plan carefully on the amount of threads to be used. Instead of creating a thread for every request, or using single threads executing sequentially, we create a thread queue to deal with requests from client peers and neighbor peers, and we set the queue number at an open parameter that can be modified according to system performance. Threads must be synchronized in some way to share common resources, and moreover we need to prevent starvation and deadlock. In our implementation, we avoid deadlock by always acquiring locks in the same order, which implies that we need a lock hierarchy among classes. The lock hierarchy is a queue rather than a tree: each object in the hierarchy must have one and only one parent object (as in the Java class hierarchy), but it must have one and only one descendant as well.

5.3.2 Asynchronous communication

RMI makes the implementation of communication convenient. However, the remote method is still different from the local method: the remote invocation needs not only execution time, but also remote communication time. Sometimes, one invocation can result in several other remote invocations. For example, in our system, when a client invokes a query method in its super-peer, the super-peer will also invoke query methods from its neighbors, and these neighbors will invoke methods from their neighbors and so on. Thus, the response time of such a remote call could be quite long. If the sender always has to wait for a response, then the system will spend a lot of time waiting, rather than running applications.

To use system resources optimally and to respond faster to users, we adopt an asynchronous communication mode to deal with remote requests, thus the system can devote the majority of its time to processing rather than waiting. We create two queues: one request queue and one response queue. In asynchronous communication, the requester simply sends a request to the server’s request queue, and then proceeds in its own operations without waiting for any response from the server. Meanwhile, a demon thread runs in the server and checks the request queue periodically: if the queue is not empty, it selects one to execute from the head of the queue. There is also a demon thread checking the response queue, and according to the responses, it decides whether to forward results to the request sender or to wait for other results. For example, only when the super-peer gets all responses from its neighbors, will it return the result to the query requester, otherwise it will wait for other replies. In this way, both the peers and super-peers avoid idling there waiting for a response from the request receiver, thus improving system efficiency.

5.3.3 TTL

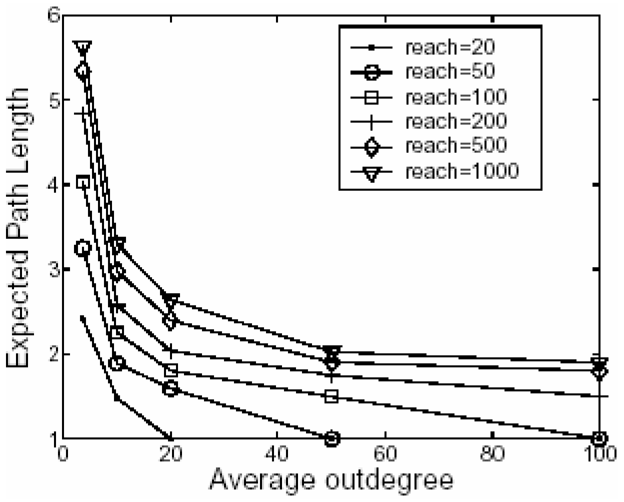


Figure 5-8 Estimating EPL given reach and out degree [19]

As we mentioned in Chapter 4, one major issue in unstructured systems is the difficulty of choosing an appropriate TTL. For a given network topology and node degree, the broadcast reaching set is a function of TTL. Because the number of query results is related to the reach, and the reach is decided by the TTL, we determine the TTL according to the number of results we expect to get. If users expect to get the entire results, we need to determine the minimum TTL required to reach all nodes. Additional query messages will create redundancy once a query has reached every node. When the desired reach covers just a subset of all nodes, finding a TTL to produce the desired reach should be made globally. One solution is predicting the estimated expected path for the desired reach and average out degree, as Figure 5-8 [19] illustrates. Another effective method to set TTL is the Expanding Ring method [2], in which expansion of the ring reduces the message overhead significantly, compared with regular flooding with fixed TTL.

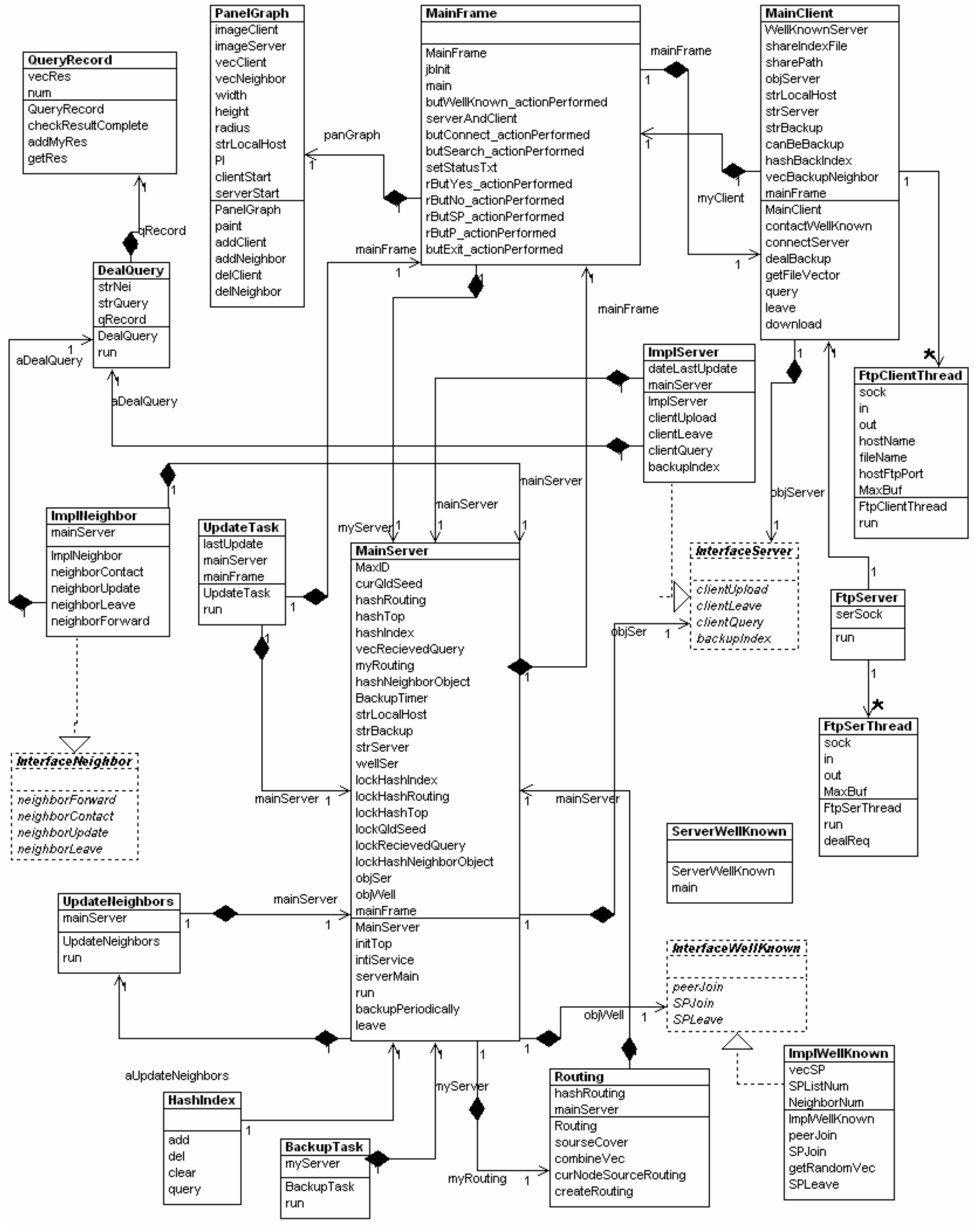


Figure 5-9 UML class diagram of main components

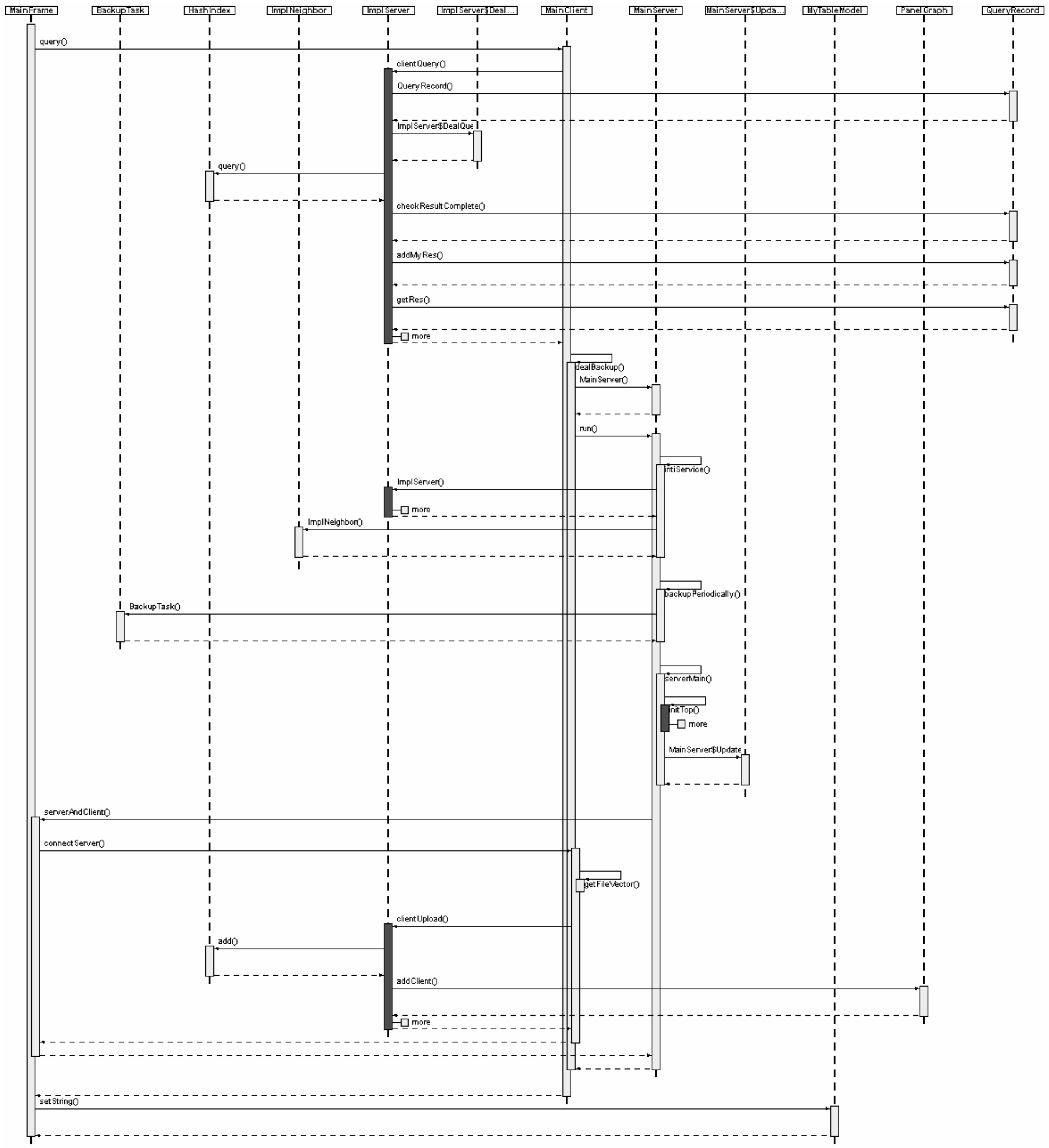


Figure 5-10 UML sequence diagram of main operations

Chapter 6

Experiments

In this chapter, we present the results of our experiment and analysis. We have performed two kinds of experiments to evaluate the system. First, we designed a simulator to evaluate the performance and scalability of the architecture and the routing protocol, because it is impossible to run the system in an Internet-based network with millions of computers. Second, we installed our P2P software on the LAN of the Computer Science Department of UBC to test and evaluate the *real* system.

6.1 Routing protocol evaluation

6.1.1 The simulator

P2P systems are not set up and maintained by a central authority; thus creating and observing a non-trivial network and measuring the performance as described in the previous chapters is a difficult task. However, simulation can help to gain insight into the behavior of the system, and therefore in order to evaluate the super-peer level overlay network's discovery mechanisms, we have developed an event-based simulator in Java. The simulator can simulate the application-level broadcasting and query searching processes with different routing algorithms and network topologies. For the simulation, we have used synchronous rounds. In each round, every node reads the messages from its input queue and handles them according to specified routing rules. Each node keeps track of the number of messages it forwards during each round. A simulation of a broadcast is initiated by having one randomly chosen node in the network send a message to all its neighbors, and the simulation ends when no more messages are forwarded per round. The results of the simulation may depend on which node was chosen to start the broadcast. We therefore perform a large number of simulation runs, with the node initiating the broadcast chosen at random. To determine results, we take the average over all the runs. Then we vary the network properties, such as topology, network size and average node fan in and fan out, repeat the tests and analyze the results.

6.1.2 Topology

To simulate the routing cost, we have generated network topologies with the typical characteristics of P2P networks. One typical characteristic of naturally formed P2P networks is a power-law distribution in their node degree. In networks like this, a new node is more likely to connect to existing nodes that are highly connected. When a node *i* joins the network, the probability that it connects to a node *j* already belonging to the network is given by

$$P(i, j) = \frac{d_j}{\sum_{k \in V} d_k}$$

where d_j is the degree of the target node, V is the set of nodes that have joined the network and $\sum_{k \in V} d_k$ is the sum of out degrees of all nodes that have previously joined the network. This means that in the network, a few nodes may have a very high degree and many may have a low degree. Barábasi and Albert [17] have proposed a model which generates topologies with a power-law characteristic and serves as a good basis to generate random topologies with the typical characteristics of P2P networks. To generate the Barábasi-Albert model topology, we use BRITE [18], a topology generator developed at Boston University. Our simulator has a parser which can parse the output file exported by BRITE, and create the target topology accordingly. However, in our super-peer based P2P structure ECSP, because we can utilize the well-known registration server to control the network topology, we can make the network with desirable topological properties, such as balanced node degree, low diameter and high connectivity. Thus, to test the system performance on such a network, we have also created two other popular network topologies: grid topology and random topology with balanced degree.

6.1.3 Experiments and analysis

The experiments conducted for this study prove the correctness of the broadcasting algorithm: for all the tested network topologies, no matter where the source nodes are located, the query messages eventually reach all nodes in the network. Therefore, Efa is an adequate alternative to simple flooding. In addition, to evaluate the performance of Efa, we have compared it with simple flooding over the same topology network and with the same guarantees offered.

A major issue in P2P networks is the load on network participants. Typically the participants are PCs at homes or offices. They always have their own working and entertainment tasks and cannot devote significant processing resources to the network when they join the P2P systems. Thus, minimizing the system overhead is an important objective for our algorithm. In our experiments, we define the overhead as the duplicated messages on the network. Figures 6-1, 6-2 and 6-3 compare the system overhead when using Efa with the overhead when using simple flooding. Our experiments run on three different network topologies: grid topology, random topology and Barábasi-Albert random topology. In the simulation, we set the TTL to "unlimited," to make the broadcast reach every node in the network. For each topology, we vary the network size and repeat the tests ten times, then compute the average results. The results in Figures 6-1 to 6-3 reveal that Efa greatly reduces network overhead for all three topologies, compared with flooding.

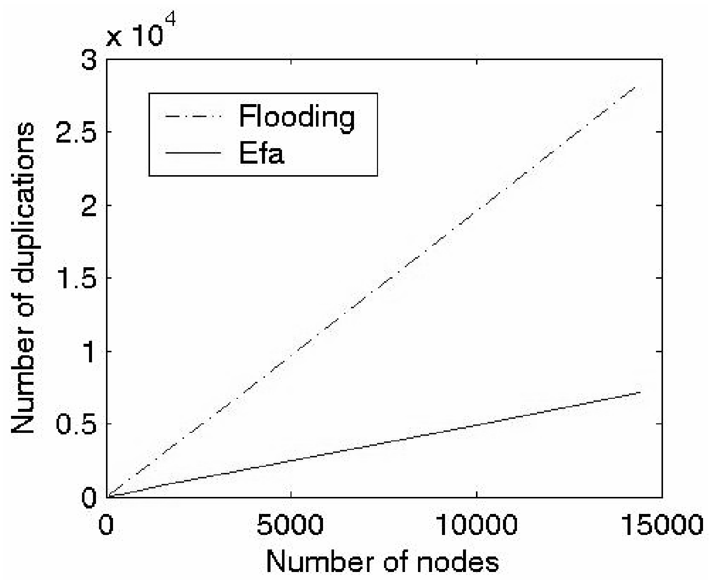


Figure 6-1 Overhead vs. network size: grid topology

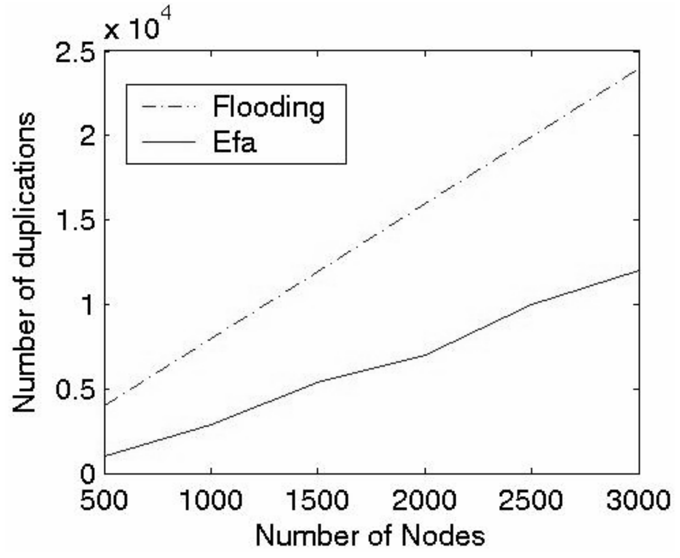


Figure 6-2 Overhead vs. network size: random topology with beginning degree: 5

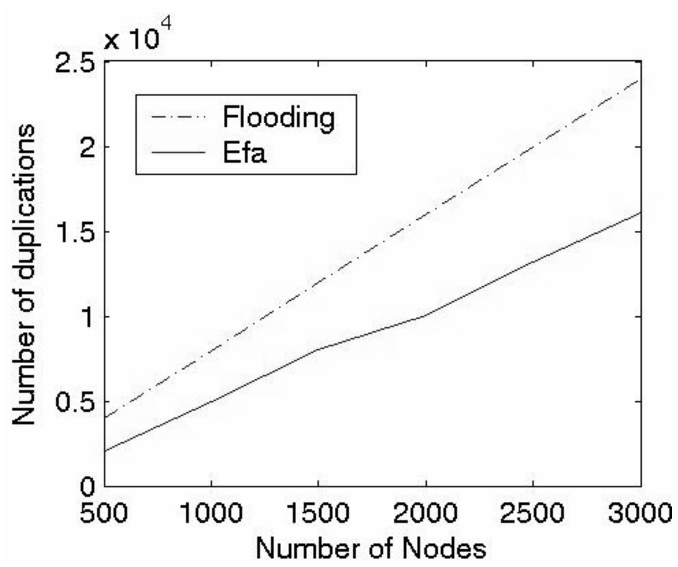


Figure 6-3 Overhead vs. network size: Barábasi-Albert topology with beginning degree: 5

Figure 6-4 depicts the relationship of network duplication ratios and network average degrees. The experiment is performed on a random topology network with 3000 nodes. The network duplications increase with the network average degree in the flooding situation. For Efa, when network average degree grows to some extent, the duplication ratio begins to decrease with the increase of average node degrees.

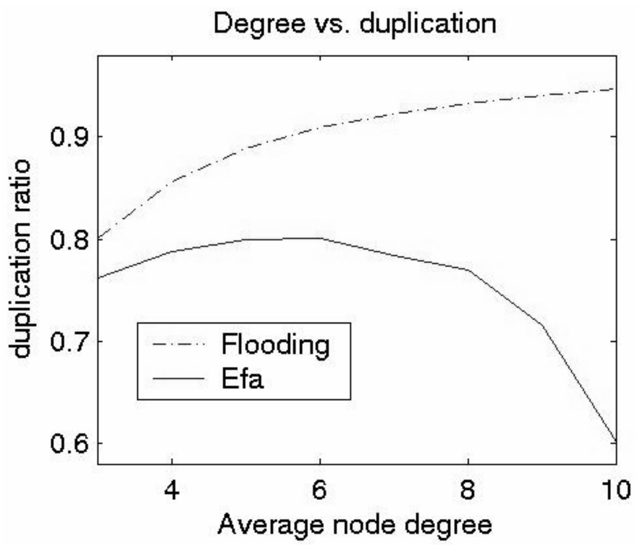


Figure 6-4 Degree vs. duplication

Like flooding, Efa also uses TTL to control the number of hops through which a query can be propagated. Figure 6-5 illustrates the system overhead in terms of the number of messages generated as the TTL increases. The experiment is also performed on a random topology network with 3000 nodes. The Efa routing produces fewer messages than flooding does when TTL increases.

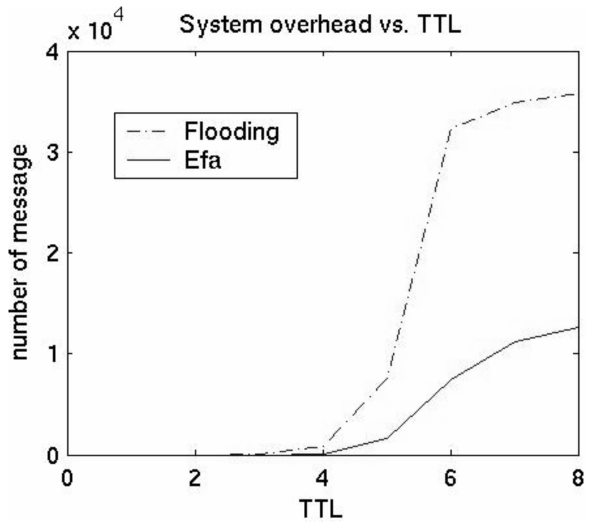


Figure 6-5 TTL vs. system overhead

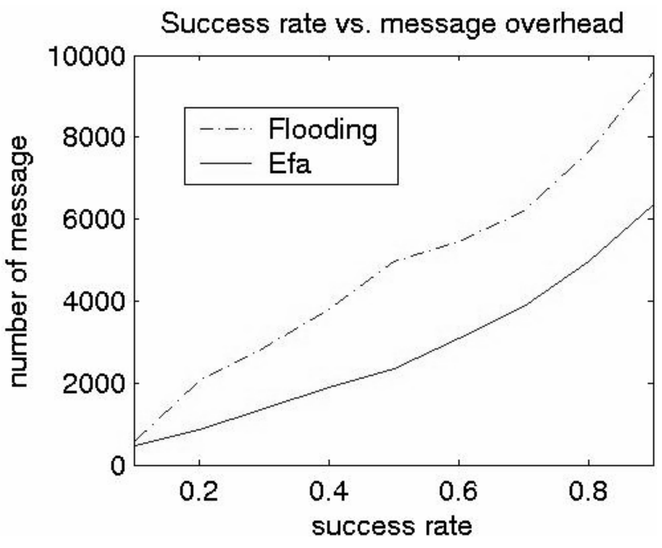


Figure 6-6 Success rate vs. system overhead

The experiment in Figure 6.6 is performed on the random topology with 3000 nodes and an average degree of 5. The content is replicated at 0.3% of the randomly selected nodes in the network. The results in Figure 6-6 identifies the relationship of query success probability to the number of messages produced in the system.

All experiments performed on different network environments demonstrate that compared with simple flooding, Efa reduces many overheads of individual nodes as well as the loads of the whole network. It achieves better performance and scalability than flooding does, especially when the network is well connected or the network size is large.

6.2 Real system evaluation

6.2.1 Experiment setup

The experiment environment is made up of 16 PCs with Intel Pentium Ⅲ 1.004 GHz processor and 256M of RAM, and all the PCs are running the Red Hat Linux 9 operating system. There are a total of 50 different files in the system. Every peer maintains 20 files and each of the files is around 5KB. To test our architecture, we randomly choose one to serve as a well-known registration server and the other 15 PCs to serve as peers. The 15 peers are grouped into three clusters. In every cluster, a peer also acts as a super-peer. To evaluate the system performance, we compare it with a Gnutella system. The topology of Gnutella is randomly generated with an average degree of 4. In both systems, to generate the network traffic peers send queries every two seconds. Because the experiments are conducted on a LAN, the transmission time between two nodes is too short to reflect the real Internet environment; therefore we add 0.1 second delay for every transition between two nodes.

6.2.2 Experiment results

We first compare the costs of ECSP and Gnutella. The number of messages forwarded by a node relates directly to the amount of resources, such as bandwidth and CPU cycles, that are consumed. We calculate the cost as the number of messages that need to be forwarded in the network to perform the queries. The costs of the Gnutella system and the super-peer system's costs are defined as follows:

For the Gnutella pure P2P systems:

$$c = 1 + \sum_{i=1}^{N} (d_i - 1) = 1 + N(d - 1)$$

c: cost in number of messages forwarded

d_i: degree of node i

d: average node degree

N: number of nodes in the network

For the super-peer P2P system:

$$c = 1 + \sum_{i=1}^{G} (d_i - 1) = 1 + (N \div s)(d - 1)$$

c: cost in number of messages forwarded

d_i: degree of super node i

d: average super node degree

N: number of nodes in the network

G: number of clusters

s: average cluster size

Obviously, clustering the peers into one more level of hierarchy can save a lot of network bandwidth. The formula for the super-peer system is not an optimal result because the above formula uses simple flooding as its super-peer level broadcast scheme. In our system, we use a more efficient algorithm, Efa, to incur less traffic. Our experiments justify the analysis above.

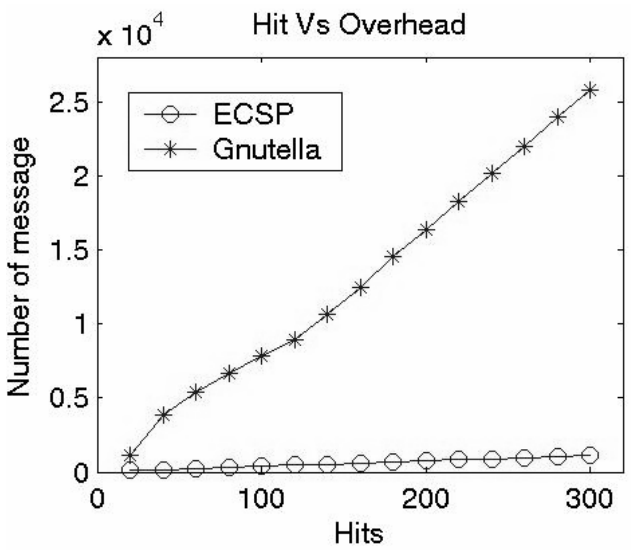


Figure 6-7 Hit rate vs. system overhead

Figure 6-7 shows the query hits and number of messages needed. To attain the same number of successful query hits, ECSP sends significantly fewer messages than Gnutella does. Figure 6-8 reviews the relationship of time consumed and system overhead: for any time period, our system creates less traffic than Gnutella does. Therefore, our system accrues lower costs than Gnutella.

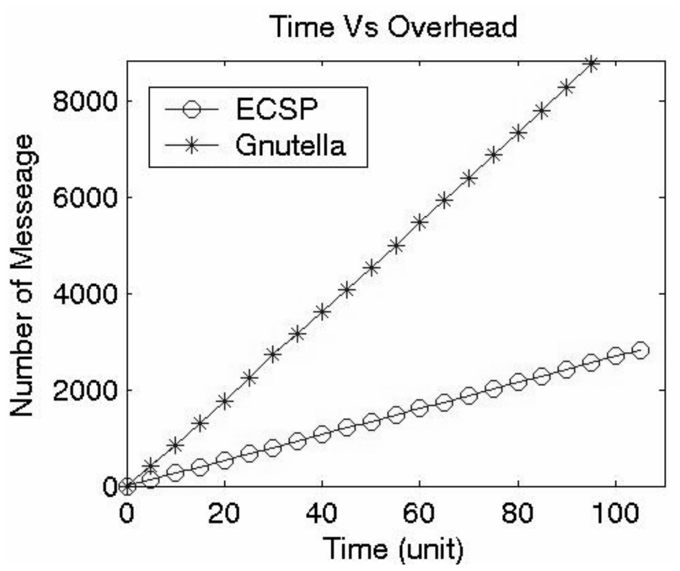


Figure 6-8 Time vs. system overhead

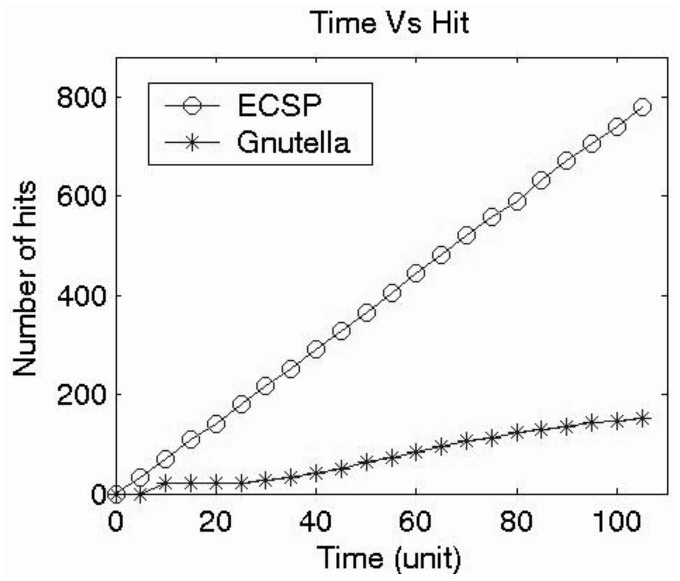


Figure 6-9 Time vs. query hits

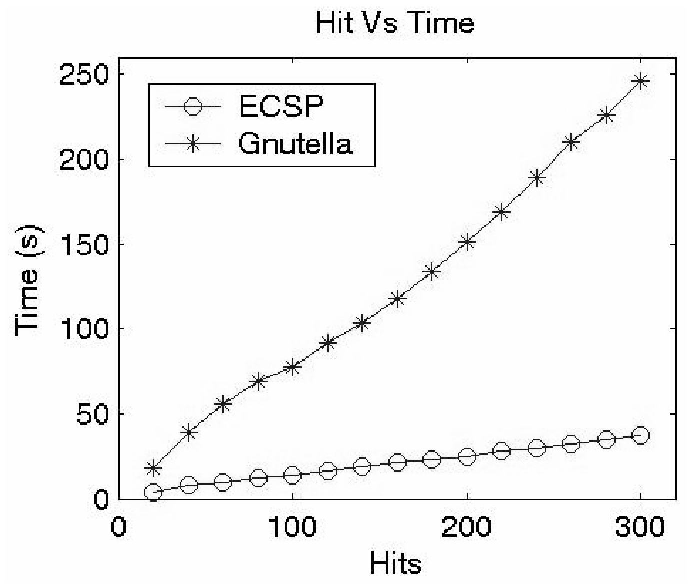


Figure 6-10 Query hits vs. completion time

Figures 6-9 and 6-10 compare ECSP and Gnutella in terms of query hits and completion time. Two observations can be drawn from these comparisons: as Figure 6-10 shows, our system uses much less time to finish the same amount of queries; with the same time limit, our system can finish more queries (Figure 6-9).

In sum, all of our experiments prove that the ECSP structure and the Efa backbone routing protocol dramatically decrease the cost of queries without decreasing the ability to satisfy queries, compared with Gnutella and simple flooding.

Chapter 7

Conclusions

7.1 Summary

P2P systems are being deployed fairly actively on the Internet. However, the existing systems address different aspects of P2P problems and none of them are perfect.

In this thesis, we investigate P2P systems currently in use, primarily on decentralized, unstructured systems. The unstructured systems are actively used by the largest community of Internet users and support many desirable properties. Two major deficiencies of unstructured P2P networks are addressed: scalability and efficient search mechanisms. Consequent to our observations, we propose a hierarchical-based super-peer structure, ECSP. The ECSP system groups peers into a two-level hierarchy according to topological proximity. Super-peers are then selected from regular peers to act as cluster leaders, responsible for locating content and maintaining the network structure for client peers. Super-peers are also connected to each other, forming a super-peer overlay network. To scale the routing on the overlay network connecting the super-peer nodes, we design an application level broadcasting algorithm: Efa. Efa's application is not limited to this system; rather it is an application-level broadcasting protocol applicable to all potentially very large, unstructured, P2P networks on the Internet. A prototype system with ECSP architecture is designed and implemented in Java. Experiments are performed both with a real network environment and with simulation tools. The experimental results demonstrate that the ESCP architecture and the overlay broadcasting algorithm achieve good performance and scalability, and they can be used to construct powerful infrastructures for very large scale, unstructured P2P environments.

7.2 Future work

Our hierarchical model and routing strategy are scalable and efficient in content locating. Several other aspects call for further investigation, however.

First, security issues need to be studied. This thesis focuses on performance issues, and security has been left out. In order to implement and use the proposed architecture satisfactorily in a real network, security mechanisms must be included. For example, an authentication process is needed to check if the received document is a genuine copy and is not tampered. In addition, some sensitive content may need to be encrypted, and in some case, a peer might be authenticated to access some subset of the resources on another peer.

Second, content-wise grouping or grouping based on specifications dictated by particular peers could be introduced, such that the model would be more efficient. The peers may wish to join some other groups even if they are not geographically located in that area. A related area of potential interest is the possible mobility of peers.

Third, efficient caching strategies can be used to reduce the path length required to retrieve an object. Therefore, the number of messages exchanged between peers is decreased. As a result, the communication latency between peers can be reduced.

Finally, there are some other implementation aspects that can be further improved. For example, content can be downloaded from multiple sources simultaneously, and search technology can be improved to support complex SQL queries.

Bibliography

[1] Napster website: http://www.napster.com/

[2] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. *Proceedings of the 16th international conference on Supercomputing*, June 2002.

[3] Gnutella website: http://gnutella.wego.com/

[4] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. "Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing," *Technical Report*, UCB/CSD-01-1141, April 2000.

[5] A. Rowstron and P. Druschel. "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems," *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, Middleware, November 2001.

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

[7] S. Ratnasamy, P.Francis, M.Handley, R.Karp, and S. Shenker. "A Scalable Content-Addressable Network," *ACM SIGCOMM*, August 2001, pp. 161-172.

[8] Peer-to-Peer Working Group website: http://www.P2Pwg.org

[9] OpenP2P.com website: http://www.openp2p.com/.

[10] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. *P2P computing. Technical Report HPL-2002-57*, Hewlett Packard Lab, 2002.

[11] SETI@Home website: http://setiathome.ssl.berkeley.edu/

[12] C. Plaxton, R. Rajaraman, and A. Richa. "Accessing Nearby Copies of Replicated Objects in a Distributed Environment," *Proceedings of the ACM SPAA*, Newport, Rhode Island, June 1997, pp. 311–320.

[13] Christopher Kommareddy, Narendar Shankar, Bobby Bhattacharjee. "Scalable Peer Finding on the Internet," *Proceedings of Global Internet Symposium, Globecom 2002*, Taipei, Taiwan, November 2002

[14] Robert L. Carter and Mark E. Crovella. "Server Selection Using Dynamic Path Characterization in Wide-Area Networks" *Proceedings of INFOCOMM*, Kobe, Japan, April 1997, p. 1014.

[15] Mark Stemm, Srinivasan Seshan, and Randy H. Katz. "A Network Measurement Architecture for Adaptive Applications," *Proceedings of INFOCOM*, Tel Aviv, Israel, March 2000.

[16] S. Campadello, O. Koskimies, K. Raatikainen, H. Helin. “Wireless Java RMI,” *Proceedings of The 4th International Enterprise Distributed Object Computing Conference*, Makuhari, Japan, September 2000, pp. 114-123.

[17] A.L. Barábasi and R. Albert, “Emergence of Scaling in Random Networks,” *Science*, October 1999, pp. 509-512.

[18] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. “BRITE: An Approach to Universal Topology Generation,” *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems- MASCOTS '01*, Cincinnati, Ohio, August 2001.

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

[20] FastTrack website http://www.fasttrack.nu/

[21] Aberer and M. Hauswirth. “Peer-to-Peer Information Systems: Concepts and Models, State-of-the-Art, and Future Systems,” *ICDE Advanced Technology Seminar*, 2002.

[22] Jonas Åslund, “Authentication in P2P Systems,” Master Thesis, 2002

[23] Balachander Krishnamurthy and Jia Wang, “On Network-Aware Clustering of Web Clients”, *Proceedings of ACM SIGCOMM'2000*.

[24] G. Pandurangan, P. Raghavan, and E. Upfal. “Building Low-Diameter P2P Networks,” *Proceedings of the 42nd Annual IEEE Symposium on the Foundations of Computer Science (FOCS),* 2001.

[25] C. Kommareddy, N. Shankar, and B. Bhattacharjee. “Finding Close Friends on the Internet”. *Proceedings of ICNP*, November 2001.

[26] S. Banerjee, C. Kommareddy, and B.Bhattacharjee. “Scalable Peer Finding on the Internet” *Proceedings of the Global Internet Symposium*, Globecom, November 2002.

[27] Son Vuong and Juan Li. “ECSP: an efficient cluster based P2P architecture”. *Proceedings of the International Conference on Internet Computing*, June 2003.

[28] Son Vuong and Juan Li. “Efa: an Efficient Content Routing Algorithm in Large Peer-to-Peer Overlay Networks”. *Proceedings of the Third IEEE International Conference on Peer-to-Peer Computing*, September 2003.