REDUCING QUERY OVERHEAD THROUGH ROUTE LEARNING IN UNSTRUCTURED PEER-TO-PEER NETWORKS

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Selim Çıraçı
August, 2005

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. İbrahim Körpeoğlu(Co-supervisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Ali Aydın Selçuk

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Enis Çetin

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet B. Baray
Director of the Institute

ABSTRACT

REDUCING QUERY OVERHEAD THROUGH ROUTE LEARNING IN UNSTRUCTURED PEER-TO-PEER NETWORKS

Selim Çıraçı
M.S. in Computer Engineering
Supervisors: Assist. Prof. Dr. İbrahim Körpeoğlu, Prof. Dr. Özgür Ulusoy
August, 2005

In unstructured peer-to-peer networks, such as Gnutella, peers propagate query messages towards the resource holders by flooding them through the network. This is, however, a costly operation since it consumes node and link resources excessively and most of the time unnecessarily. There is no reason, for example, for a peer to receive a query message if the peer does not own any matching resource or if the peer is not on the path reaching to a peer holding a matching resource.

Semantic routing is a technique that tries to forward the queries only to those nodes where replies are likely to come from. In this thesis, we present “Route Learning”, a semantic routing scheme, which aims to reduce query traffic in unstructured peer-to-peer networks by utilizing a well-known estimation technique called “Parzen Windows”. In Route Learning, peers try to find the most likely neighbors through which replies can be obtained for submitted queries. In this way, a query is forwarded only to a subset of the neighbors of a peer, or is dropped if no neighbor, which is likely to return a reply, is found. The scheme has also mechanisms to cope with variations in user submitted queries, like changes in the keywords. This way the scheme can also evaluate the route for a query for which it is not trained. The proposed scheme consists of three phases: training, evaluation, and recursive learning. The last phase enables the scheme to adapt itself to changes in a dynamic peer-to-peer network. Our simulation results show that our scheme reduces the bandwidth overhead significantly without scarifying user satisfaction compared to a pure flooding based querying approach.

Keywords: Peer-to-peer query routing, Parzen Windows estimation, query caching, distributed hash tables.

ÖZET

PLANSIZ PEER-TO-PEER SISTEMLERDE "ROTA ÖĞRENME" YÖNTEMI ILE SORGULAMA YÜKÜNÜN AZALTıLMASı

Selim Çıraçı
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticileri: Assist. Prof. Dr. İbrahim Körpeoğlu, Prof. Dr. Özgür Ulusoy
Ağustos, 2005

Gnutella gibi belli bir planı olmayan peer-to-peer (P2P) sistemlerde, akranlar sorgu mesajlarını kaynak tutucularına sel baskını yöntemi ile P2P ağı üzerinden aktarır. Fakat bu operasyon çok verimli değildir, çünkü sel baskını yöntemi akranların bant genişliği gibi kaynaklarını gereksiz yere harcamaktadır. Örneğin, bir akranın bir sorgu mesajına cevap verecek kaynağa sahip olmadığı, veya o kaynağa gidecek yolu bilmediği zaman, bu sorgu mesajının rotasını belirlemede görev almasına gerek yoktur.

Anlamsal rota belirleme tekniği sorgu mesajlarını sadece cevapların gelebileceği akranlara yollamaya çalışan bir mekanizmadır. Bu tezde plansız P2P sistemlerde sorgulama yükünü "Parzen Windows" sınıflandırma algoritmalarını kullanarak azalatmaya çalışan ve bir anlamsal rota belirleme sistemi olan "Route Learning" (Rota Öğrenme) sistemini sunuyoruz. Sunduğumuz bu sistemde, akranlar sorgu mesajlarını cevapların yüksek olasılıkla gelebileceği komşu akranlara iletirler. Bu sayede, bir sorgu mesajı bir akranın, sel baskını tekniğinde olduğu gibi, tüm komşularına değil, sadece bu komşuların bir kısmına aktarılır. Eğer bir akranın komşularının gelen bir sorguya cevap veremeyecekleri tahmin edilirse, o sorgu mesajı düşürülür. Sistemimiz ayrıca sorgulardaki anahtar sözcük değişikleri gibi dinamik yapılar ile başa çıkabilecek mekanizmalara sahiptir. Böylece sistemimiz daha önceden görmediği sorguların bile rotasını tahmin edebilir. Sistemimiz üç fazdan oluşmaktadır; bunlar: öğrenme, değerlendirme ve yeniden öğrenmedir. Yaptığımız testler sonucunda sistemimizin sorgularda bant genişliği kullanımını, kullanıcı memnuniyetini çok düşürmeden, yüksek düzeyde azalttığı görülmüştür.

Anahtar sözcükler: Peer-to-peer sorgulama yönlendirmesi, Parzen Windows tahmin yöntemi, sorgunun ön belleğe alınması.

Acknowledgement

I would like to express my gratitude to my thesis supervisors Assist. Prof. Dr. İbrahim Körpeoğlu and Prof. Dr. Özgür Ulusoy for their encouragement, support and belief in my work.

I would like to thank Assist. Prof. Dr. Ali Aydın Selçuk for giving his time to discuss various aspects of my thesis work. His suggestions on using Radix 32 and the organization of probabilistic approachess were of great help.

I would like to thank Assist. Prof. Dr. Ali Aydın Selçuk and Prof. Dr. A. Enis Çetin for spending their valuable time to evaluate my thesis.

I thank to members of the Networking and Systems Group at Bilkent University for their comments and recommendations. I also thank Technical Research Council of Turkey (TÜBITAK) for their support to the project with grant numbers EEEAG-103E014 and 104E028.

Finally, I would like thank my family; my father, my mother, my sister, my borther-in law and my nephew for their support and understanding throughout my thesis study. I especially want to thank my father for his guidance and making me who I am right now.

To My Father; in hopes that I would be respected as much as him someday...

Contents

1 Introduction **1**

2 Related Work **5**

2.1 Work on Gnutella Network Measurements 6

2.2 Work on Reducing Query Overhead 8

3 Gnutella Protocol **11**

4 Observations From Gnutella Protocol **14**

4.1 Gnutella Crawler 15

4.2 Measured Parameters 15

4.2.1 Number of Keywords Contained in a Query 15

4.2.2 Repetition Rate of Keywords in Queries 16

4.2.3 Initial TTL Values of Queries 16

4.2.4 Peers' Contribution to the Network 17

4.2.5 Query Hit to Query Ratio 17

4.2.6 Repeated Queries 18

4.2.7 TTL Values of Repeated Queries 18

4.3 Observations 19

5 Route Learning **25**

5.1 Background 25

5.2 Framework 30

5.3 Route Learning scheme 31

6 Experiments and Results **44**

6.1 Experiments 44

6.2 Results 47

7 Conclusion **55**

8 Future Work **58**

List of Figures

3.1 A successful Gnutella handshake between two peers. 12

3.2 Header of a Gnutella Protocol Message. 12

3.3 Gnutella download fails due to firewall. 13

4.1 Distribution of number of keywords seen in query messages. . . . 20

4.2 Repetition count of keywords. a) Repetition count of keywords versus the rank of keywords. Keywords are ranked according their frequency of occurance in query messages. b) Log of repetition count of keywords versus log of rank of keywords. 21

4.3 Cumulative distribution function (CDF) of the number of files shared by a peer. Most of the peers (95%) share less than 1000 files. 22

5.1 Parzen Windows Estimation with a 1-D feature space. Volume size is 1, then P(3/2)=(4+5)/19/1/. 27

5.2 Parzen Windows training example. 29

5.3 Parzen Windows estimation example. 30

5.4 Dictionary mapping function. 36

5.5 Radix 32 based mapping function. 37

6.1 Simple P2P Network Model. 45

6.2 Number of errors made by the dictionary function in exact match experiment conducted with 6800 queries. 49

6.3 Number of errors made by the Radix 32 based mapping function in exact match experiment conducted with 6800 queries. 49

6.4 Dictionary function behavior on modified queries; experiment conducted with 6800 queries. 50

6.5 Radix 32 based mapping function modification behavior; experiment conducted with 6800 queries. 51

6.6 Bandwidth Used in Gnutella Experiment conducted with 6800 queries; 90% reduction to flooding requirements 52

6.7 Bandwidth used by flooding and Route Learning. 53

8.1 Fallacy scenario: a) training phase, b) evaluation phase. 58

List of Tables

4.1 The Gnutella protocol messages observed in trace data and their fraction of occurrances. 19

4.2 Initial TTL values seen in query messages and their percentages. Most queries observed in the traces have an initial value of 4. . . . 22

4.3 The count of repeating the same query string by a peer. 81% of peers that repeated a query sting have repeated the string 2 times. 23

6.1 Simulation results. 54

Chapter 1

Introduction

Peer-to-peer (P2P) communication paradigm has gained significant importance since it allows people to become active participants in a huge community. In P2P networks, nodes, that are also called peers, can act both as servers and clients allowing the resources to be distributed and the load that is put onto one server to be shared. This provides a great advantage over the client-server paradigm, where a failure of the server causes all the resources to become unreachable. However, the distributed nature of P2P networks and resources brings the problem of locating resources in a huge network. This problem is also known as *location lookup*. To determine the location of a resource, a query is generated and sent through the P2P network to be received by one or more peers which can answer the query. This process is called *querying*.

Querying in unstructured P2P networks like Gnutella [1] is performed by flooding the query through the network. In flooding, each node receiving the query forwards the query to all of its neighbors. Although it is simple and robust, this approach, however, causes too much bandwidth to be wasted, because most of the time the query is forwarded towards neighbors from where no answer returns. There is also the possibility of receiving the same answer from multiple peers even though one answer would be enough.

There exists many studies so far that come with efficient schemes and protocols

to solve this problem. Semantic routing [11] is one of them, which aims to reduce querying overhead by trying to find the nodes where an answer can come from and forwarding the queries towards those nodes The scheme that we propose in this paper, which we call *Route Learning*, is also a semantic routing technique, that is inherently a classification problem. Our proposed scheme learns routes and makes routing decisions for query messages based on this learning.

There are also caching based query routing techniques proposed to reduce the querying overhead. Many of these techniques, that are discussed in Chapter 2, cache a query as a whole. However, our observations that are reported in Chapter 4 show that only 15% of the queries are repeated, which implies that the reduction in the overhead can be at most nearly 15%. Besides this, our observations also show that the repetition rate of keywords follows a power-law distribution as we discuss also in chapter 4. From this observation one can conclude that some keywords are repeated at a higher rate than the average query repetition rate. Therefore, making use of this high keyword repetition rate may greatly reduce the querying overhead. Our *Route Learning* scheme makes use of this repetition rate by applying a classification technique to estimate the next hop of a query by using its keywords as features.

Our scheme starts with a process in which a node learns resources reachable by its neighbors. We call this phase of our scheme the *training*. During the training phase, nodes store query messages they receive before forwarding them to all their neighbors using the flooding approach. The same nodes, upon arrival of query hit messages, find the matching query messages that were stored earlier and record the number of results returned to the queries. These recorded results are used in the next phase of the scheme, the *evaluation* phase, to make intelligent routing decisions.

Intelligent routing decision allows a node to forward a query only to a subset of neighbors. It is, however, still possible a query to be forwarded to all the neighbors, or to be dropped depending on the *containing* probability estimated for each neighbor. The containing probabilities are estimated for each keyword seen on the query. The containing probability for a keyword and neighbor indicates

the likelihood of finding the resource matching the keyword through that neighbor. Flooding the query in the evaluation phase occurs only when the containing probability is found to be *not trained* for each neighbor. Although flooding is not desirable in the evaluation phase, it allows the scheme to continue to learn routes. In this way, the scheme can adapt itself to the dynamic nature of peer-to-peer networks, such as node disconnections or new node arrivals.

We evaluated our scheme through simulations. Our simulation results show that our scheme reduces the bandwidth overhead of querying significantly compared to a pure flooding based approach that is not using any intelligent routing. More specifically, a P2P network with our scheme consumes only around 10% of the bandwidth that would be consumed in a pure flooding based P2P network. Our simulation results also show that the quality of answers in a P2P system using our scheme is not reduced considerably compared to the quality of answers in flooding based systems.

Another contribution of this thesis is derivation of characteristics information about some of the important parameters of the Gnutella network. The Gnutella network is one of the greatest P2P networks that is currently under use over Internet. Therefore it is important to know about its characteristics. For example, people who will do simulation studies about this network may benefit from the model and parameter characteristics derived through monitoring the current operational network.

We have done extensive monitoring of Gnutella traffic and then analyzed the captured Gnutella messages. Using this analysis we have derived characteristics of some of the important parameters such as keyword repetition rate, distribution of initial TTL values, etc.

In summary, the contributions of the thesis are as follows:

- Proposal of a scheme, which we call Route Learning, to use for reducing the querying overhead in Gnutella and similar unstructured P2P networks. This is crucial in these networks since querying is based on flooding which causes extensive control traffic overhead.

- Evaluation of the performance of the proposed scheme. The performance evaluation is done both through performing real-time experiments in a test bed consisting of a custom-built Gnutella crawler attached to the real Gnutella network, and through simulations. What is needed for simulation is a model and characteristic information about some important Gnutella parameters. And this is obtained as a result of the next contribution.
- Deriving a model and characteristic information about some of the parameters of the Gnutella network and its user population. The results of this contribution have been used in the simulations done as part of this thesis, and can also be used for other simulation studies.

The thesis is organized as follows. In Chapter 2, the recent research that is related to our work is described. A brief description of the Gnutella protocol is provided in Chapter 3. The observations extracted from the Gnutella network is shown in Chapter 4. An introduction to classification problems, the P2P framework we have assumed, and the Route Learning scheme we propose is provided in Chapter 5. The experiments we conducted to evaluate our routing scheme are described in Chapter 6, and the performance results are presented in Chapter 6. In Chapter 7 we summarize our conclusions. Finally, in Chapter 8 we describe some improvements that we are considering to test on the scheme.

Chapter 2

Related Work

Peer-to-peer (P2P) technology is a relatively new technology that enables formation of huge overlay networks and that allows users to become active participants in those networks without requiring a pre-installed server system. Each node, called servent or peer, in a P2P network is both a server and a client, and has equal responsibilities with other peers. This allows resources to be distributed among users, thus the probability of a resource becoming unreachable is reduced. In the standard client/server paradigm, on the other hand, the server is the only owner of the resource, thus the resource becomes unavailable if the server crashes. The distribution of resources is the major motivation for considering P2P technology as one of the most important applications of Internet.

Today, thanks to the advances in Internet technologies, one can find many P2P systems some of which are operational and used by thousands of people and some of which are of research interest. These systems can be classified roughly into there main groups [2], which are unstructured, loosely structured and structured P2P systems. In an unstructured P2P network like Gnutella [1], the location of the resources is completely independent from the formation of the overlay network. In structured P2P systems, on the other hand, a node is given a virtual address/location determined by the resources the node is sharing. Lastly in loosely structured P2P systems the resources supply routing hints to the intermediate nodes. Unstructured P2P networks can further be divided

into three groups, which are pure, hybrid and centralized. In pure unstructured networks each node has equal responsibilities while in others some nodes take special responsibilities like holding an index of the resources the neighbors are sharing. An example to hybrid systems is Kazaa [3] network; where nodes with high-bandwidth connections become super-nodes, which have the responsibility of routing network messages. Normal nodes, which are not active participants in routing, connect to super-nodes and send the index of their shares to the super-nodes.

Considering the number of users that will connect to Internet in the future, it is obvious that the P2P networks of the Internet should accommodate large number of nodes. Thus the only P2P network type that can accommodate such large number of nodes is *unstructured* P2P systems. The major problem in such P2P systems is, however, resource location (querying), which can be summarized as given a keyword set S finding the resources whose meta-data or name matches all the keywords in S. Current unstructured P2P systems, propagate query messages by flooding the messages, which increases the bandwidth requirements of the network so that nodes with low bandwidth connections cannot keep up with the query traffic. Our study, in this thesis aims at reducing the querying overhead by applying some well-known classification technique. There are many related protocols and schemes that share the same goal that we do, and we discuss some of the most related ones to our scheme in Section 2.2. As we have discussed in the Introduction chapter, to generate close to real life simulations in evaluating our scheme, we have conducted some measurements and experiments in the live Gnutella network. Therefore, before discussing the related studies that aim on reducing the query overhead, we first discuss some related work on Gnutella network measurements and analysis.

2.1 Work on Gnutella Network Measurements

There exist several studies on the measurement and analysis of several P2P networks. The study on [12] lists some of the important parameters that should be

considered when simulating a P2P file sharing network. In this study, a model for some of the parameters are derived from real world observations, and the parameters considered are separated into two groups. The first group of parameters are related with the distribution of resources in the P2P network, and the second group of parameters are related with modeling the behavior of peers. The main difference of our study from [12] is that we try to characterize P2P network parameters using traces collected by custom P2P crawlers. We also investigate some parameters that are not investigated in [12]. A list of parameters that are important for Gnutella simulations is also reported in [18].

The authors of [4] conducted an analysis of the Gnutella network using crawlers, like we did. They logged for an hour the query and query hit messages seen at three different points on the Gnutella network. The study of the logged messages is focused on the detailed analysis of repeated queries, the TTL values seen in the queries, and the inter-arrival times of submitted queries. We also analyzed some aspects of repeated queries and the TTL values of user submitted queries. But we are more focusing on the characterization of initial TTL values set in the queries, and on the characterization of inter-arrival times of repeated queries. A similar study to [4] is presented in [14]. That study, however, is more focused on content analysis of queries. It derives and lists some popular keywords that are used in submitted queries. In this aspect, the work also resembles to what we did, but we are also trying to find a model for the repetition count of popular keywords.

The study presented in [13] also uses crawlers to collect message traces from Napster and Gnutella networks. It plots cumulative distributions of peer characteristics such as the number of resources shared, the uptime of peers, and the bandwidth capacity of peers. Our observations also focus on similar parameters such as the number of resources shared by peers, but we also try to come up with a model that can be used to generate similar values for these parameters in simulation studies.

2.2 Work on Reducing Query Overhead

There exists a substantial amount of work on reducing query traffic overhead in an unstructured P2P network. Here we briefly describe some of these works that are closely related to what we present in this thesis.

A study about Gnutella query traffic characteristics shows that queries in a Gnutella network exhibit a significant amount of locality, namely queries with the same or similar search strings are common [4]. Based on this observation, a caching scheme is proposed in which every peer stores query strings, time-to-live (TTL) values, and the corresponding query hit messages while the peer is relaying Gnutella queries and query hit messages. Then, upon receiving a query message, a peer scans the TTL and query string pairs stored in itself for a match to the query string and the TTL value stored in the received query. If such an entry is found, the peer returns the corresponding query hit (i.e., the answer) as a reply to the query. Similar to that study, our system also makes use of the locality in queries. However, rather than indexing the query string as a whole, we index each of the keywords seen in a query, separately. This helps us to cope with changes in user submitted query strings. Another difference of our work from [4] is that the peers in our scheme do not answer queries on behalf of their neighbors or any other peer. They estimate the most likely neighbors where replies can come from and forward the query messages only towards those neighbors. In other words, we do not cache the query hit messages that originate from peers holding the matching files at intermediate peers where the query hits are forwarded.

Applying a distributed hash table (DHT) structure on P2P networks is another approach to improve the scalability and also the exact-match hit rate. Gnutella developers have adopted a DHT-like scheme where peers index their shares into a binary hash table and send it to their neighbors at certain time intervals [7]. The problem with a hash table based structure is that it can only support exact match queries. An *n-gram* based technique to allow fuzzy queries is described in [5]. This idea is borrowed by [6] to bring DHTs to P2P systems with a complex query structure. These approaches are significantly different from ours in that we are not requiring peers to hash their resources and distribute them

to their neighbors, which causes extra messaging overhead. We just require each peer to learn about the resources managed at or reachable by their neighbors by only monitoring the query and query-hit messages sent to or received from their neighbors. The query and query hit messages are already part of the base Gnutella protocol, hence we are not causing extra control traffic overhead.

The routing method proposed in [8], may seem related to Route Learning since both schemes use a probabilistic approach. However, the only feature common between their scheme and ours is that both aim at reducing the query overhead in the network. The protocol shown in [8] makes use of caches and forwards queries to neighbors with a probability called "broadcast probability". Route Learning, on the contrary, does not make use of caches and forwards the queries to neighbors depending on the knowledge that is collected about neighbors.

Most of the *learning* based routing protocols focus on clustering peers that have similar interests together. In the scheme proposed in [9], peers associate to the networks using two connection methods. The first connection method, called the *random connection*, is the method used by Gnutella in which a peer chooses the neighbors to connect to by availability; i.e. if a neighbor can accept the new comer then the connection is established. In the second connection method, called the *attractive connection*, the new comer establishes a connection according to the answers it has received. If a peer X has supplied most of the answers, then the new comer establishes a connection with peer X. This way, most of the queries submitted will be answered at one hop distance. A method similar to this is proposed in [10]. In that method peers develop knowledge about their environment by using a neural network technique. The main difference of our work from [10] and [9] is that in our system peers do not change connections. Changing neighboring connections is not considered to be good solution since peer-to-peer networks are highly dynamic networks in which nodes frequently disconnect or connect to the network. Also *Route Learning* has methods to cope also with changes in keywords, thus it does not only rely only on keyword repetitions but it also uses similarity between keywords as a means to reduce the querying overhead further. Finally, Route Learning can dynamically adjust itself to the changes in the topology of a P2P network. The changes in topology can

be due to new connections or disconnections.

Chapter 3

Gnutella Protocol

In Gnutella network, peers form an overlay network over Internet by opening point-to-point TCP connections to each other. To join the overlay, a new coming peer has to discover a small subset of the active overlay participants. This discovery is done by querying the host caches, which hold the IP addresses of some of the high-uptime participants. Each Gnutella compatible P2P client comes with a set of predefined host cache addresses. After discovering a set of the peers to join, a newcomer initiates Gnutella handshake with a peer in that set (Figure 3.1). During this handshake, both the newcomer peer and the peer that is already part of the Gnutella network indicate to each other the Gnutella protocol version they are using and the extensions they support [1]. If the peer that is already part of the Gnutella network can accept the connection request from the new coming peer, it indicates this by sending an OK message. If, on the other hand, the peer cannot accept the connection, it indicates the reason why it cannot accept the connection and provides the newcomer with a set of peers it knows. This way the newcomer can discover other peers without further querying the host cache.

After a successful connection establishment, peers start exchanging Gnutella protocol messages. A Gnutella message header, shown in Figure 3.2, consists of a *global unique identifier* (*GUID*) field, a *time-to-live* (*TTL*) field, a *hops* field, a *payload type* field, and a *payload length* field. The GUID is used to overcome routing loops that may occur in the overlay. To prevent routing loops, a peer

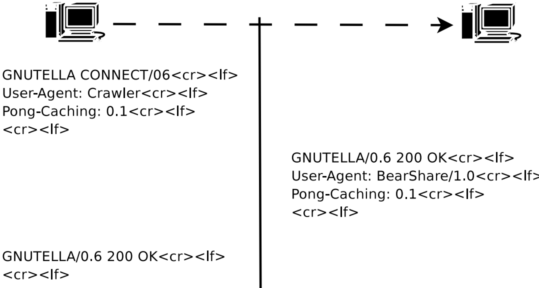


Figure 3.1: A successful Gnutella handshake between two peers.

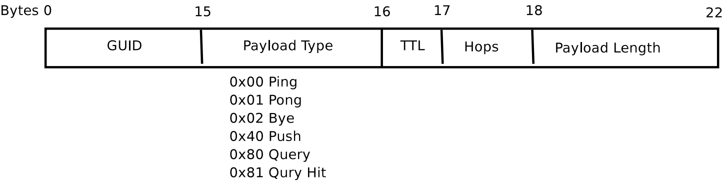


Figure 3.2: Header of a Gnutella Protocol Message.

receiving two messages with the same GUID ignores the second one. Each peer receiving a Gnutella message increases the hops count value in the message by one and also decreases the TTL value by one. When the TTL value of a message reaches to zero, the message is not forwarded anymore. The payload type field is used by peers to distinguish different types of Gnutella messages. There are five types of Gnutella messages, which are *Query*, *QueryHit*, *Bye*, *Push*, *Ping*, and *Pong* messages.

A *Query* message contains the user submitted query string as its payload. Peers in Gnutella network maintain a *routing table* that stores the *GUID*'s of the queries they have received. A peer receiving a *Query* message extracts the GUID from the header and searches that GUID in its routing table. If the GUID is previously seen, in other words an entry with the same GUID appears in the routing table, then query is dropped; this way Gnutella becomes "free" from routing loops. After checking the query in the routing table, the peer forwards the query to all of its neighbors, except to the neighbor that has sent the query.

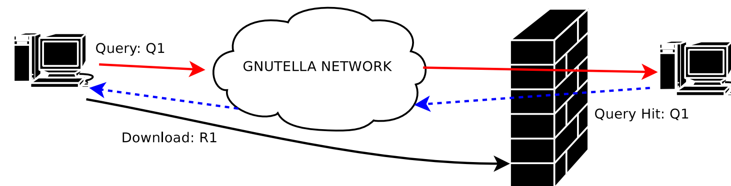


Figure 3.3: Gnutella download fails due to firewall.

The peer, after forwarding, checks its shared resources for a match to the query string included in the Query message. If the peer has resources that match the query string, it sends a Query Hit message back. The Query Hit message is set the same TTL value as the *hops* field of the corresponding Query message. The payload of the Query Hit message contains the physical address of the originator and the names of the resources that match the corresponding query. To download a resource, the user of a node in the Gnutella network selects the resource from the list of resources received via *Query Hit* messages. Then the Gnutella protocol opens a connection to the peer which is going to supply the resource the user wants to download, and the download process starts. If for any reason this connection attempt fails (Figure 3.3), the peer that wants to download a file sends a Push message through Gnutella network to the peer that is going to supply the resource. The Push message contains the sender's physical address and the TCP port number from which it can accept connections. Upon receiving the Push message, the supplying peer opens a connection to the peer requesting the resource.

The *Ping* and *Pong* messages are used to exchange topological information. When a peer receives a Ping message, it answers back with at least 10 Pong messages each containing the physical addresses of other peers that are collected again by sending Ping messages. *Bye* is used by a peer to indicate its disconnection from the network to its neighbors.

Chapter 4

Observations From Gnutella Protocol

Having models and characteristics information about important parameters of a network, including a P2P network, enables us to accuratly simulate the network to do performance analysis and protocol verification. There is already some work describing some models for some of the parameters of the Gnutella network, but these are not covering all important parameters necessary for all types of simulations. Therefore, we first identified some parameters of the Gnutella network and its user population, that can be useful for the performance evaluation of our proposed Route Learning scheme; and then we did extensive Gnutella network monitoring and analysis to obtain knowledge about the characteristics of these parameters[19].

While doing this, we followed a methodology similar to the one described in [14]. We programmed a custom Gnutella crawler to collect Gnutella network traces. Using the crawler we gathered large sets of data and logged them on a local disk. The logged data includes various Gnutella protocol messages that suit our measurement goals. After logging the Gnutella messages, we also probed numerous nodes, whose addresses are obtained from the logged messages, in order to have an idea about the duration of node uptimes.

In this chapter, we first briefly introduce the measurements and analysis that we conducted on the live Gnutella Network. Then we describe briefly our Gnutella crawler that is used to collect Gnutella protocol messages transported over a portion of the Gnutella network. We then introduce and describe some of the P2P network parameters that we identified and that we tried to model and estimate using the message traces we obtained via our crawler.

4.1 Gnutella Crawler

Our Gnutella crawler is written in Java and follows the Gnutella protocol specification version v0.6 [1]. First, the crawler connects to the HTTP address *gwebcache2.limewire.com:9000/gwc* to collect physical addresses of some active peers. It then starts opening connections to those peers and also builds its own host cache from the physical addresses collected via unsuccessful connection attempts and Pong messages. After connecting to three peers successfully, the crawler starts monitoring and logging Gnutella messages considering the parameters we are going to discuss in mind.

4.2 Measured Parameters

The simulation of a Gnutella network requires consideration of a lot of parameters. We focused only a subset of all possible parameters and tried to understand the nature of the values of these parameters in the Gnutella network. We now introduce the parameters we focused on, and describe how the related traces are collected to obtain the characteristics of these parameters.

4.2.1 Number of Keywords Contained in a Query

For semantic routing techniques, keywords in a query define routing rules for that query. Thus, the more keywords a query has, the more information the routing

technique can extract about the query's route. It is widely believed that P2P users submit short queries consisting of one or two keywords, so its difficult to apply semantic routing techniques. To test this belief, we have programmed the crawler to collect 10 thousand queries from five different connection sets (each set consisting of different nodes). After collecting the data, the queries are tokenized with “`._*()",;:!?`” deliminators to extract the keywords and then each keyword is counted. To combine the counts from different connection sets, the averages of the counts is taken.

4.2.2 Repetition Rate of Keywords in Queries

It is a fact that in P2P networks there exist some popular resources which are queried a lot. Many protocols that try to improve search quality rely on repetition rate of keywords in queries. So it is important to develop a model for popular keywords for such techniques.

To develop this model, we have used the tokenized queries of the previous parameter and hashed each keyword using Java's string class, which hashes strings by adding the integer values of each character in a string. These hashed keywords are used as a key to index the hash table holding the number of accesses made to the cells. We have given the highest rank of 1 to the mostly accessed cell, which in turn is the keyword with the highest repetition.

4.2.3 Initial TTL Values of Queries

For P2P simulations, the initial TTL values set in Query messages play an important role, since Query messages can travel longer distances with a higher TTL value which increases the chance of finding the resources requested by the query. The TTL value in a query is also important for determining the bandwidth required for various protocols. Gnutella protocol specification [1] states that TTL values in queries should be set to 7 and many P2P simulators follow this specification. However, the fact that many Gnutella clients today use shorter initial TTL

values makes TTL an important parameter to achieve realistic P2P simulations.

To keep track of TTL values, while collecting query data for the previous parameters we have also programmed the crawler to log the *TTL* and *hops* values of the received queries. The initial TTL values are calculated by adding these two values. Again averages of several collected data sets are used to obtain the final estimates.

4.2.4 Peers' Contribution to the Network

Distribution of resources to peers in a P2P simulation should also be handled carefully, since the query hit rate is directly affected by this parameter. Some previous studies show that %25 of the Gnutella peers do not share any files at all, and %7 of peers share 100 files [14].

To collect the required data to estimate the distribution characteristics of resources, the crawler has been programmed to collect 10 thousand Pong messages from five different connections sets. The collected Pong messages contain the physical addresses of the nodes sending the Pong messages, and the total number and size of resources shared by these nodes.

4.2.5 Query Hit to Query Ratio

Although peers' contribution to the network greatly affects the Query Hit messages returned to Query messages, the popularity of the shared resources is another important factor that can affect the Query Hits, since popular resources will be queried more than other ones. So it is not only important how many files a peer shares, but it is also important what kind of files the peer shared. It is hard to model the popularity of shared resources, however, collecting the number of Query messages with matching Query Hit messages in the Gnutella network may give an idea. Assuming, for example, $x\%$ of the queries in the collected data have a matching Query Hit message, we then can adjust the popularity parameter in

a simulation so that the chance of getting a Query Hit to a Query message is $x\%$.

To find the Query to Query hit ratio, our crawler uses a hash table. This hash table holds the GUID of a Query message as a key and stores the corresponding Query Hit message as data. Upon receiving a Query message, the crawler inserts a null Query Hit message, which has zero as the hit-count, to the hash table. Since the Query Hit message has the same GUID as a Query message, upon receiving a Query Hit message, the crawler searches the GUID of the message in the hash table, and if found, the Query Hit message is inserted to the table. By collecting the Query Hit messages in this way, found the chance of getting a Query Hit to a submitted Query message.

4.2.6 Repeated Queries

When the P2P network does not return any results to a query submitted by a user, the query is re-submitted by the user or the P2P client software. Thus, it may be important to model this behavior for simulation of caching systems.

In order to find out how many queries are repeated in a five different query sets each containing 10 thousand queries, we have hashed the query string in a Query message together with the *hops* value of the message, again by using Java's string class. If two different queries are hashed to the same cell, then that query is marked as a repeated query. Although it is impossible to know which peer has submitted the query when the hops value is greater than 1, two queries with the same query string and the same hops value have a very high probability of being repeated, thus we have used this method to recognize repeated queries.

4.2.7 TTL Values of Repeated Queries

When a user of a P2P system re-submits a query, it provides some advantage for the P2P client to send the query to the network with a higher TTL value. Although Gnutella specification does not mention this, some clients may have

adapted this approach in order to increase search quality. This makes it important to analyze the TTL values in repeated queries. In order to analyze this behavior, the crawler also logs the TTL values in queries that are recognized as repeated queries.

4.3 Observations

Message Type	Occurence
Query	91%
Query Hit	1%
Ping	8%

Table 4.1: The Gnutella protocol messages observed in trace data and their fraction of occurrances.

In this section we present our results about the characteristics of the parameters that we have described. Before that, however, we would like to present a table describing the overall Gnutella traffic characteristics (Table 4.1). In this table, we did not include the Pong messages since they are sent whenever a node receives a Ping message. The overhead of flooding is clearly seen in the figure; %91 of the Gnutella traffic consists of Query messages. Thus the need for a protocol that reduces this overhead is clear.

In Figure 4.1, the distribution of the number of keywords submitted in a query is shown. Our analysis of the related traces shows that 68477 queries out of 100 thousand queries contain less than 5 keywords. We found 4 as the mean of the number of keywords that can be seen in a query. Queries with just one keyword constitute the 10% of all queries we analyzed. The Figure 4.1 indicates that users tend to submit more descriptive queries instead of submitting single-keywords queries. It is also interesting to notice that 1561 queries out of 100 thousand queries contain more that 7 keywords which makes around 1.5% of all the queries analyzed.

Figure 4.2 shows the repetition count of keywords in user submitted queries.

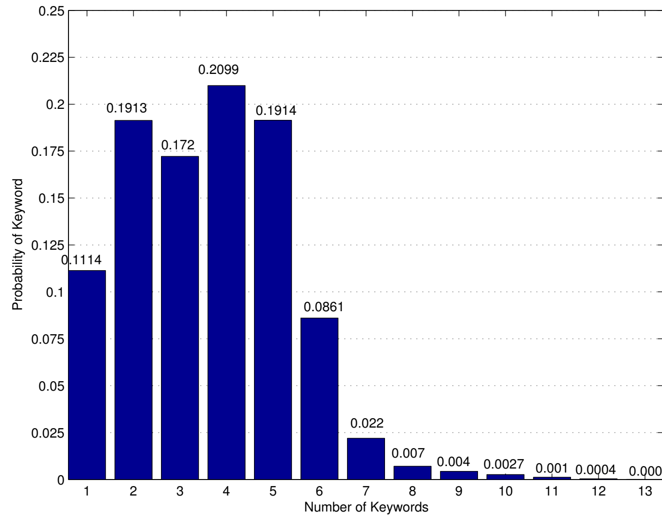
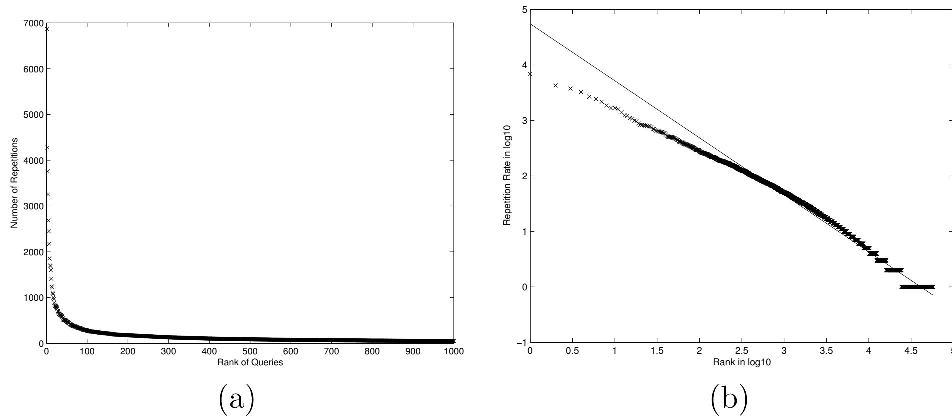


Figure 4.1: Distribution of number of keywords seen in query messages.

In plotting the graphs in the figure, we first ranked all the keywords with respect to their repetition count. In Figure 4.2-a, the x-axis is the rank of the keywords, and the y-axis is the repetition count of the keywords with respect to those ranks. The keyword that has the highest repetition count has rank 1, the keyword that has the next highest repetition count has rank 2, and so on. The analysis of this plot shows that the repetition count of keywords obeys a power-law distribution with respect to the rank of keywords. We think this is due to popularity of some keywords. For example, in our traces, keywords "`mp3`" and "`divx`" are the two most popular keywords and they have very high reparation counts (i.e. we can see them in many queries). Since the curve on the graph is steeply decreasing, we only plotted the repetition counts up to rank 1000. Otherwise it was difficult to identify the curve on the graph. To better show that the repetition count of keywords obey a power-law distribution, we plotted the repetition count versus rank of keywords in logarithmic scale, and fit a polynomial with degree 1 to the curve obtained in this manner. The Figure 4.2-b shows the plot in logarithmic scale with the fitted polynomial (in this plot we did not limit the rank). The fitted polynomial has coefficients -1.028 and 4.74 (i.e., . it is the line described by equation $y = -1.028 \times x + 4.74$).



(a) (b)

Figure 4.2: Repetition count of keywords. a) Repetition count of keywords versus the rank of keywords. Keywords are ranked according their frequency of occurance in query messages. b) Log of repetition count of keywords versus log of rank of keywords.

The distribution of initial TTL values observed in Query messages are shown in Table 4.2. As can be seen from the table, majority of the Gnutella clients (89%) set the initial TTL value to 4 in a Query message. The clients setting the initial TTL value to 3 constitute around 11% of the peers. The number of clients setting the initial TTL value to something else is less than 1% and therefore negligible. We also tested what happens if a client tries to submit Query messages with larger initial TTL values than 4. For this we modified our Gnutella client so that it submits queries to the network with TTL values larger than 4. We have noticed that majority of the clients around us have lowered the TTL value to 4. We believe that Gnutella developers have taken such an action to lower the overhead introduced by the flooding mechanisms used for disseminating the queries.

In Figure 4.3, we show the cumulative distribution function of number of files shared by a peer. On the x-axis we have the number of files shared, and on the y-axis we have the fraction of peers sharing number of files that is less than or equal to the corresponding value indicated on the x-axis. From the figure we see that 50 peers out of 420 peers share zero files. In other words, nearly 10% percent of peers do not share any files. The figure also reveals that only around 5% of

TTL	% of Messages with TTL
3	11%
4	89%
5 and greater	< 1%

Table 4.2: Initial TTL values seen in query messages and their percentages. Most queries observed in the traces have an initial value of 4.

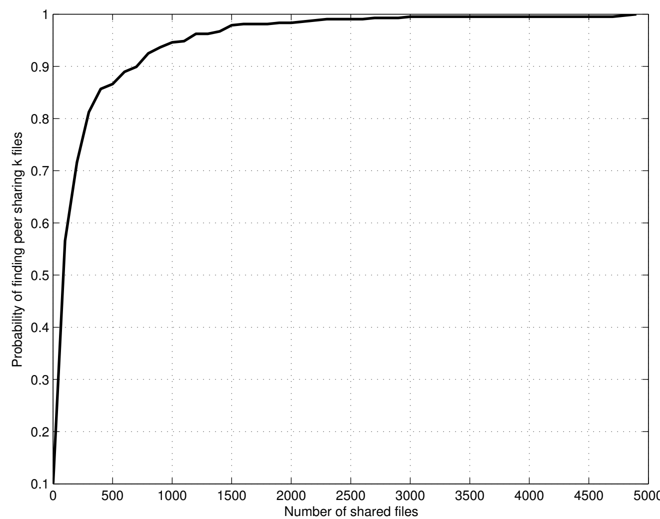


Figure 4.3: Cumulative distribution function (CDF) of the number of files shared by a peer. Most of the peers (95%) share less than 1000 files.

peers share more than one thousand files. These are not suprising results since it is a quite well-known fact that only a small percentage of peers in a P2P network share huge numbers of files. It is also interesting to notice that although many peers indicate that they share small number of files, these shared files are quite large in size (around 2 GB). This leads us to believe that in Gnutella network users tend to search and download large files which in turn causes peers to share large files.

Although Query to Query Hit ratio greatly depends on the queries submitted, from Table 4.1 one can see that Query Hit messages constitute only %1 of the overall P2P message traffic observed in the traces. This is quite a small fraction

indeed.

Repetition Count	% of Repetitions
2	81%
3	14%
4	3%
5	1%
More than 5	1%

Table 4.3: The count of repeating the same query string by a peer. 81% of peers that repeated a query sting have repeated the string 2 times.

We also looked how many times a query string is repeated by a peer in submitting queries. A query string can be repeated by a peer because the results obtained in previous query submissions may not be found satisfactory by the peer. Out of the 100 thousand queries observed, we have identified 15678 queries as repeated queries. This constitutes 15% of all the queries observed. Table 4.3 shows that majority of the queries are repeated twice (81% of all queries). The percent of queries that are submitted three times is 14%. We have found that only 2 queries are submitted to the network more than 5 times. These two queries have all "?" as query strings, which we believe are used by peers to discover all the names of the resources shared by their neighbors, although nothing about this is mentioned in Gnutella protocol specification. Our inter-arrival time analysis for repeated queries shows that on average there is a 1242291.09 msec time interval between the repeated queries. This corresponds to around 21 minutes between each repeated query, which is a reasonable time, since a user re-submits a query after the arrival and inspection of the previous results. Our TTL analysis for repeated queries shows that the initial TTL values of these 15678 repeated queries are not increased by the clients submitting these queries. Given that majority of the queries are repeated only twice, we can say that a Gnutella user is satisfied with the results after a second submission that comes after a sufficient inter-arrival time (around 21 minutes). Since the mean uptime of Gnutella peers are around 60 minutes [4], we conclude that there is no need for an increase in the TTL of the repeated queries for the purpose of getting better results, and therefore we find the decision made by Gnutella developers about not to increase

the TTL values in repeated queries to be correct; since by the time the query is re-submitted new nodes would join the network so there is no need to increase the TTL value of a query.

Chapter 5

Route Learning

In this chapter, we detail our proposed solution, *Route Learning*, to the querying overhead problem seen in unstructured P2P networks. Before going into the details, however, we first provide some insights into the classification problem and one of its solutions in the next section, since we use a technique from this domain as part of our Route Learning solution. Then, in Section 5.2, we define the properties of the P2P network environment for which our solution can be applied. Finally, in Section 5.3 we provide the details of our solution.

5.1 Background

The solution that we propose in this thesis for reducing query overhead is based on a solution of the general *classification* problem. Therefore, in this section we briefly describe what a classification problem is and give one popular solution approach for it.

A classification problem consists of a classifier and two or more classes to be chosen. The classifier tries to classify a given object according to its features; thus ideally the features that belong to a class should be distinct from other classes.

The classification problem mainly consists of five phases which are data collection, feature extraction, modeling, training and evaluation [15]. The classifiers are distinguished according to their learning phase; one type requires pre-knowledge of the objects to be classified; in other words it is trained with a set of samples whose class is known previously. This type of learning is called supervised learning. The second type of classifiers does not require prior knowledge, and this type of learning is called non-supervised learning. Route Learning classifier belongs to the first type, which is therefore detailed in this paper. The aim of supervised learning is to provide the classifier with some information about the classes which determine the location of each class in the feature space. For example, assume that we are classifying the objects with red color. For this problem we have two classes, red and non-red, and each class' features are the red, green, blue (RGB) values of the objects. Here we model a very simple classifier that works according to nearest-neighbor rule, where an object is labeled with the class whose features are nearest to the features of the object. In the learning phase, we provide the classifier with some samples and specifically mark which class each sample belongs to. Then, in the evaluation phase we provide an object to the classifier which labels the object with the class whose "color" is nearest to the object we have provided.

The simple classifier we have used in the above problem gives good results (twice the error rate of a Bayesian classifier [15]) with unlimited samples. However, in most cases we do not have that many samples. Thus we have to use classifiers that take a statistical approach. The primary statistical solution to a classification lies in the Bayesian decision theory (see Equation 5.1).

$$P(\omega_i \mid \bar{x}) = \frac{P(\bar{x} \mid \omega_i)P(\omega_i)}{\int_i P(\bar{x} \mid \omega_i)P(\omega_i)} \tag{5.1}$$

The term $P(\bar{x}|\omega_i)$ in Equation 5.1 is called the likelihood or the class conditional density. Estimation of this parameter is the crucial part of the classification problem [15]. If the probability distribution of the likelihood is known, then we could use some parameter estimation techniques to find the parameters of the

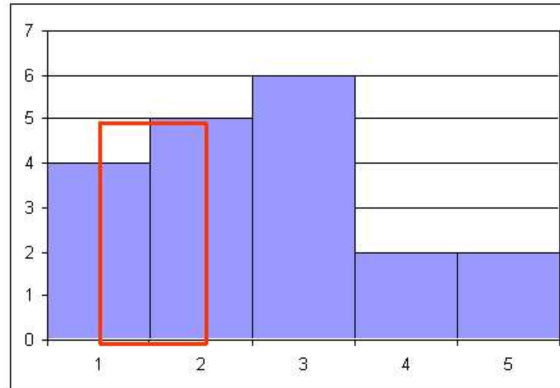


Figure 5.1: Parzen Windows Estimation with a 1-D feature space. Volume size is 1, then P(3/2)=(4+5)/19/1/.

distribution. However, in many cases we do not have this valuable information, thus we use some non-parametric techniques like *Parzen Windows*.

Next, we provide a simple description of Parzen Windows estimation and algorithms of a generic Parzen Windows classifier. The theoretical background of the non-parametric techniques can be found in [15]. The continuous probability that a vector **x** will fall into a very small region R is given in Equation 5.2.

$$P(\bar{x}) \cong \frac{\bar{k}/n}{V} \tag{5.2}$$

In this expression, k stands for the number of samples that fall in volume (V) enclosed by R, and n stands for the total number of samples in the set. To estimate this probability we can take two approaches. In the first approach, which is used by Parzen Windows estimation, we can take the volume fixed and count the number of samples in the volume (Figure 5.1). In the second approach, each time we can select a constant number of samples, and then calculate the volume that encloses each of these samples. After this brief explanation, we can detail the algorithms of a simple Parzen Windows classifier.

The *training* phase of a Parzen Windows classifier is very simple, where samples are placed into the feature space of the class they belong to, which is a

d-dimensional array (see Algorithms 1 and 2). In the *evaluation* phase, the number of samples within the volume centered at the given feature vector is counted for each class. Then the probabilities are estimated according to the formula given in Equation 5.1. Eventually the feature is assigned to the class that gives the maximum probability.

Algorithm 1 Parzen Windows Classifier with 1-D features, Training Phase

---

TRAIN(**TrainingSet** T)
for $i = 1$ **to** LENGTH(T) **do**
  $featureSpaces[T[i].class][T[i].features]$ += 1;
end for

Algorithm 2 Parzen Windows Classifier with 1-D features, Evaluation Phase

---

EVALUATE(**Feature** F, **int** $WindowSize$)
for $i = 0$ **to** $NoOfClasses$ **do**
  $count[i] = 0$;
end for
for $j = 0$ **to** $NoOfClasses$ **do**
  for $i = F$ - $WindowSize$ / 2 **to** $F + WindowSize$ / 2 **do**
    $count[j]$ += $FeatureSpaces[j][i]$;
  end for
end for
for $i = 0$ **to** $NoOfClasses$ **do**
  $probability[i]$ =
  ($count[i]$ / $FeatureSaces[i].totalNoOfSample$) / $WindowSize$);
end for
return MAXCLASS($probability$)

To have a better understanding about how these algorithms work, let's look at an example. Assume, we want to extract faces from images. The easiest way to accomplish this task is designing a classifier that classifies each pixel in an image by its color. For this problem, we have two classes which are *Face* and *non-Face* and we are using RGB values of a pixel as our features; obviously our classifier of choice is Parzen Windows classifier. As shown in Figure 5.2, during the training phase we supply the classifier two images. The first one is the original training image and second one is the "mask" image which tells the classifier to which class the pixel belongs to. In this example, the black and white image is

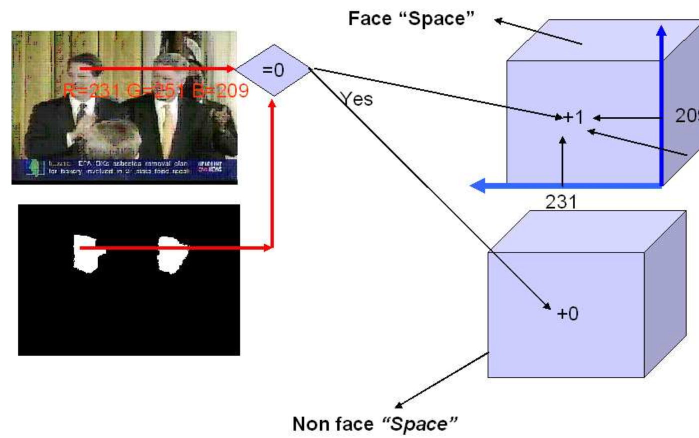


Figure 5.2: Parzen Windows training example.

the mask image and the white regions belong to *Face* class. The classifier, reads each image pixel by pixel and if the pixel read from mask is white then the pixel value from the training image is placed into the *Face* class. If, on the other hand, the pixel value is black, the pixel value from the training image is places into *non-Face* class.

During the evaluation phase, the classifier is supplied with the image to classify (Figure 5.3). Again, the image is read pixel by pixel; however, this time the classifier calculates the likelihood of each pixel by searching the pixel in each feature space. This is done by using the pixel value to index the feature space and centering the volume at the point whose coordinates are the pixel's RGB values. Then, the classifier counts the number of pixels that fall into the region enclosed by the volume. In this example, we assumed that each class is equally likely, thus there is no need to multiply the likelihoods with class conditional densities so we can classify by just looking at the likelihoods.

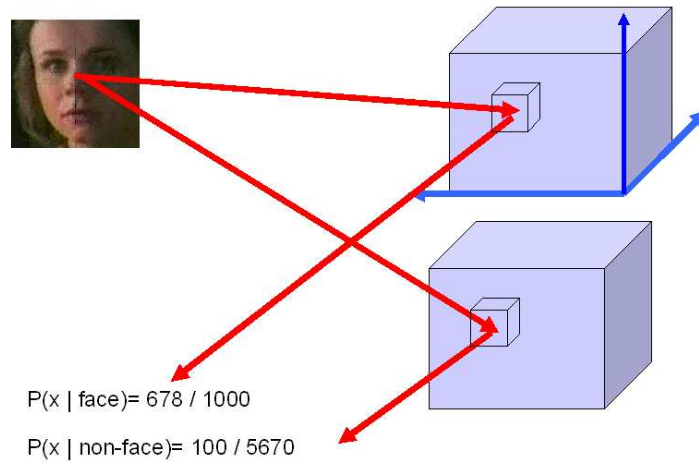


Figure 5.3: Parzen Windows estimation example.

5.2 Framework

In this section we present the features of the reference P2P network where our proposed scheme can be used. The features of the reference network are very similar to the features of Gnutella architecture and its protocol.

- The P2P system is an overlay network and it is connected, i.e. there is at least one path between any two peers.
- Each message has a TTL value and the message is forwarded between peers until TTL reaches to 0.
- Each query message has a unique identifier (QUID). The peer receiving two or more query messages with the same unique identifier and TTL responds to one of these messages. This is necessary because of the routing loops in an unstructured P2P network.
- A query is formed by one or more keywords that are separated by a deliminator character like space.
- A peer can only answer a query if it has a resource whose name or meta-data contains all the keywords in the query.

- A query hit message has the same unique identifier as the query message it is responding to.
- A query hit message is routed through the same path as the query message is routed, but in reverse direction.
- A query hit message contains all the resources that matched the query in the peer sending the hit message.

5.3 Route Learning scheme

In Route Learning scheme that we propose, a peer tries to estimate the most likely neighbors the replies to queries may come from. Peers calculate this estimation based on the knowledge that is gradually built from query and query hit messages sent to and received from the neighbors.

Route Learning inherits its basic idea from the classification problem where a peer having n neighbors has n classes to choose from. Each class corresponding to a neighbor i can be used to find out the probability of having the resource at or reachable by neighbor i. We call this probability the *containing probability* of that class. Since we do not know the probability distribution of user submitted queries, to solve this classification problem, we have used an adapted version of Parzen Windows density estimation technique. To adapt the problem to P2P domain we have made some modifications to the basic Parzen Windows technique whose details can be found in [15].

The scheme, which will be executed by all peers, maps (hashes) each keyword in a query to a point in a 1-D feature space that is created for each neighbor the peer has. Each point k of the feature space has two numbers associated with itself. The first number (*answer-count*) is the number of answers returned for keywords mapping to point k, and the second number (*query-count*) is the number of queries made to point k. The system estimates the containing probability by applying division on these two numbers.

The proposed scheme investigates the keywords in a query and indexes them separately in order to cope with varieties in user submitted queries. For example, assume that a user has submitted a query that is composed of keywords $K1$, $K2$, and $K3$. If the system would have indexed these three keywords together (not separately), then it would estimate the containing probability for a query with keywords $K1$ and $K2$ as zero. Besides investigating each keyword separately, the scheme should also have a carefully designed mapping algorithm that maps related keywords (e.g., two keywords where one is the substring of the other) to points that are close to each other in the feature space. In this way, the Parzen Windows' window function can include these keywords' probabilities in estimating the containing probabilities. As an example, assume that a user has submitted a query with a single keyword "Matrix" and a peer that has forwarded the query has learned about the neighbor that has supplied the answer. Now, assume that another user submits a query "MatrixDvd", and the Parzen Windows in our peer is not trained with this keyword. The peer will then broadcast the query to all its neighbors (wasting network resources) to learn about the neighbors that can provide the answer. However, routing the query to the neighbor that had supplied the answer to the previous query could help, since these two keywords are related with each other. If these keywords can be mapped close to each other in the feature space, then the window function can include the containing probability of the keyword "Matrix" and can send the query only to the neighbor that had supplied the answer for "Matrix".

Route Learning scheme has three phases, which are respectively *training*, *evaluation*, and *recursive learning*. Next, we describe the training phase and the mapping algorithm that we have used in mapping keywords to points of feature space. Then, we introduce the evaluation phase and the rank decision algorithm. Finally, we explain the recursive adaptation procedure, which tries to cope with changes in the network.

5.3.0.1 Phase 1: Training

Upon joining to the network, a peer starts running the training algorithm, which is an adjusted version of the Parzen Windows's training algorithm (Algorithm 3). In this phase of the scheme, the peer discovers its environment, the resources that are reachable through its neighbors, by accounting and monitoring the query and query hit messages. The training algorithm takes three parameters, which are respectively the number of other peers connected to the peer (i.e. the number of neighbors), feature space size, and train count. The train count parameter determines the number of query messages that should be sent towards neighbors in order to accumulate knowledge about the resources managed by or reachable through the neighbors. To extract the keywords from queries we have tokenized queries with the same character set used by many Gnutella clients, which contains “`._*()",;:!?`” characters. Here the size of feature space becomes a problem, since, there is no limit in the length of the keywords submitted in a query; we need to create an infinite feature space. However, creating such a feature space, that will implemented using arrays, would be impossible, thus we need to limit the feature space size. Having a very small feature space will cause queries to be forwarded to every neighbor (like flooding). This will cause a lot of query overhead, deviating from the main purpose of our scheme. However, it may have more success in getting replies. On the other hand, having a very large feature space will cause queries to be forwarded in a more selected and directed manner to neighbors, hence decreasing the query overhead. But the chance of getting replies to queries may be less in this case compared to using a small feature space.

The training phase is divided into three parts. The first part shown in Algorithm 3, starts by creating the feature spaces and close bucket hash tables for each neighbor the newly joined peer has. The hash tables are used to store the keyword maps and arrival times of each query that will be used during the training session. In a Gnutella type of network, the peers do not send a reply message when they do not manage a resource that matches a query they receive. Therefore, in order to declare a query forwarded by a peer as not contained in the neighborhood of the peer we use a timeout mechanism. It is important to note

Algorithm 3 Training Algorithm

```
ROUTELEARNTRAIN(int NumberOfNeighboors,
    int FeatureSpaceSize, int TrainCount)
for i = 1 to NumberOfNeighboors do
  FeatureSpaces[i] = sc CreateFeatureSpace(FeatureSpaceSize);
  queries_hash_table[i] = CREATEHASHTABLE();
end for
Train = 0;
while Train < TrainCount do
  query = GETQUERY();
  keyword_maps = MAP(query.data);
  query_data.maps = keywords_maps;
  query_data.startTime = currentTime;
  for i = 1 to NumberOfNeighboors do
    queries_hash_table[i].INSERT(query.QUID, query_data);
    for j = 1 to LENGTH(query_data.maps) do
      INSERTQUERY(query_data.maps[j], NumberOfNeighboors);
    end for
  end for
  STARTTIMER(query_data);
  BROADCAST(query);
end while
```

Algorithm 4 Timeout Fuction

```
ROUTELEARNTIMEOUT(QUID Q, int TimeOutFrom)
qdata = queries_hash_table[TimeOutFrom].GET(Q);
queries_hash_table[TimeOutFrom].REMOVE(Q);
```

Algorithm 5 Query Hit Fuction

```
ROUTELEARNQUERYHIT(Query_Hit Q, int AnswerFromPeer)
count = COUNTANSWERS(Q.data)
qdata = queries_hash_table[AnswerFromPeer].GET(Q.QUID);
if qdata == NULL then
  return;
end if
for i = 1 to LENGTH(qdata.maps) do
  INSERTANSWER(qdata.maps[i], count, AnswerFromPeer);
end for
```

here that the CREATEFEATURESPACE function, after allocating the memory for the feature space, sets both query-count and answer-count values of each feature space point to -1, which denotes the system is not trained. After creating the feature spaces, the training algorithm starts listening to the query messages and upon receiving a query message the algorithm calculates the points where each keyword in the query maps to; inserts these mappings (points) to the queries hash table as well as the feature space of each neighbor, and then starts the timer for the query. Inserting mappings of keywords into a feature space is done by calling the INSERTQUERY function whose pseudo-code is shown in Algorithm 6. For each keyword, this function checks the feature space point it is mapping to and if the point's values (answer-count and query-count) are equal to -1, meaning the system is not trained earlier with keywords that are mapping to this point, then the query count for that point is set to 1 and the answer-count is set to 0. However, if the values at a point are not equal to -1, which means the system is started being training earlier with keywords mapping to this point, then the query count value is increased by one.

Algorithm 6 Insert Query function

```
INSERTQUERY(Query_Data q, int NumberOfNeighboors)
for j = 1 to NumberOfNeighboors do
  for i = 1 to LENGTH(q.maps) do
    if FeatureSpaces[j][q.maps[i]].QueryCount == -1 then
      FeatureSpaces[j][q.maps[i]].QueryCount = 1;
      FeatureSpaces[j][q.maps[i]].AnswerCount = 0;
    else
      FeatureSpace[j][q.maps[i]].QueryCount++;
    end if
  end for
end for
```

When a timer for a query expires, the second part of the algorithm shown in Algorithm 4 is executed and the query is removed from the hash table of queries. In this way, further training of the system with this query is stopped, since there is no reply coming for the query from anyone of the neighbors.

The last part of the training phase (Algorithm 5) is executed upon the arrival

$$F(S) = \sum_{i=0}^{|S|} (S(i) - 97) \times i^{-1} \tag{5.3}$$

Figure 5.4: Dictionary mapping function.

of query hit messages. In this part, when a query hit message is received, the mappings of the keywords seen earlier in the corresponding query are extracted from the queries hash table by using the $QUID$ of the query hit message. Then, for each mapping, the feature space point's *answer-count* is increased by the number of answers contained in the query hit message. The *query-count* value at these points is increased by one minus the number of answers returned in the query, since the first part of the algorithm increases this value by one. These steps are shown in Algorithm 7.

Algorithm 7 Insert Answer function

INSERTANSWER(**int** *keywordMap*, **int** *AnswerCount*, **int** *AnswerFromNeighboor*)
$FeatureSpace[AnswerFromNeighboor][KeywordMap].QueryCount$ += ($AnswerCount$ - 1);
$FeatureSpace[AnswerFromNeighboor][KeywordMap].AnswerCount$ += $AnswerCount$;

As we have discussed before, we require the mapping algorithm to map related keywords to close points in the feature space. To support this, we have come up with two different mapping algorithms. Both of these algorithms use the feature space as dictionary. The mapping function for the first one, which we call *the dictionary mapping function*, is shown in Figure 5.4.

Although the function in Figure 5.4 maps related keywords to close points, it has many parameters that should be considered. Since we are mapping a continuous space (keyword space) to a discrete space (one dimensional array) of limited size, we need a distinctive rule that separates two neighboring cells of the array. This rule is the precision between two neighboring cells and it greatly affects the match rate of the system. The second mapping algorithm, we have considered, is based on Radix 32, which is shown in Figure 5.5. This algorithm

$$F(S) = \sum_{i=0}^{|S|} (S(i) - 97) \times 32^{-i} \tag{5.4}$$

Figure 5.5: Radix 32 based mapping function.

requires exactly 32^i cells to avoid collisions, where i is the number of characters in the longest keyword. Since not many peers would support such a large storage space, a limitation on the number of characters the dictionary maps is needed. Such a limitation also helps in coping with dynamic queries. Assume that the system ignores characters after the ith character in the keyword, then changes that occur after the ith character (the limit) would not be reflected on the decision making process. Changes that occur before the ith character are handled by the window function. Another use of this limitation is that it inherently sets the cell separation rule; the precision of the system, is 32^{-i}.

5.3.0.2 Phase 2: Evaluation

This phase of the scheme can be executed at a peer after the training phase is finished, i.e. after the peer has observed enough number of query messages to acquire knowledge about its neighbors and the resources reachable by them. At that time, the peer becomes ready to make intelligent decisions about which routes to forward the queries through. Algorithm 8 is the evaluation algorithm used in this phase. This algorithm is also an adaptation of the Parzen Windows evaluation algorithm. As it is seen in the algorithm, route learning does not route a query it has received to all the neighbors; but routes to a subset of the neighbors. The size of this subset is fixed and expressed with the value of the $toSendPeers$ parameter. When a query arrives, the algorithm first determines the mapping of each keyword in the query. Then, for each neighbor and keyword, the volume around the mapping of the keyword whose size is determined with the $windowSize$ parameter is applied in order to find the range of cells that will be used in estimating the containing probability. Then, from the cells that fall within the volume the *containing* probabilities are retried, which is the ratio of *answer-count* and *query count*. The containing probability of a keyword is

then calculated by adding these *containing* probabilities. To decide to which neighbors to forward the query, we use a ranking mechanism that is similar to the one used in NeuroGrid [10]. Our decision algorithm, however, completely ignores neighbors whose containing probability for a query is found to be zero to save further bandwidth. The decision function, shown in Algorithm 9, gives the highest rank to non-zero containing probabilities of all keyword matches and the lowest rank to the containing probabilities of one keyword matches. The containing probability of a neighbor for a query is calculated by *multiplying* the retrieved containing probabilities of each keyword (seen in the query) from the feature space of that neighbor. Thus, according to this formula, if one keyword's containing probability is found to be zero for a neighbor's feature space, then the query's containing probability becomes zero, for which there is no need to forward the query to. Here, we have used multiplication since in Gnutella queries, each keyword is implicitly connected to other keywords with logical "*and*" operation. That is, a peer receiving a query with keywords $K1$ and $K2$ can only answer the query if a resource's name contains both keywords $K1$ and $K2$. If, during query evaluation, a keyword is found to be not trained, i.e. the containing probability is equal to -1, the neighbor is given a lower rank. If the query is found to be not trained for all neighbors, then all neighbors will be added with rank 0 and containing probability equal to 1. This triggers the evaluation algorithm to re-execute the training algorithm only for this query, as shown in Algorithm 8.

After ranking each neighbor, the evaluation algorithm continues by calculating the set of neighbors where the query will be forwarded to. To achieve this, the algorithm calls $FINDMAXS()$ function with the highest rank, which sorts the results obtained from each neighbor's feature space and returns the top $toSendPeers$ number of neighbors for that rank. If the number of neighbors at that rank is lower then the $toSendPeers$ the evaluation algorithm lowers the rank and re-executes the steps described above[1]. This process continues until $toSendPeers$ number of neighbors is added to the set of neighbors that will receive the query. Then the queries are forwarded to these neighbors.

During the training and evaluation phase, if the scheme does not capture some

[1]The highest rank is equal to the number of keywords in the query.

Algorithm 8 Evaluation Algorithm

```
RouteLearnEvaluate(Query Q, int toSendPeers,
        int WindowSize)
maps = Map(Q.message);
probs = RankDecision(maps, NoOfNeighboors, WindowSize);
if all elements in probs[0] is 1 then
  Train(Q);
end if
for i = Length(probs) down to 1 do
  sendpeers = sendpeers
    ∪ FINDMAXS(probs[i], toSendPeers);
  if Length(sendpeers) != toSendPeers) then
    toSendPeers -= Length(sendpeers);
    continue;
  else
    break;
  end if
end for
for j = 1 to Length(sendpeers) do
  FowardQuery(Q, sendpeers[j]);
end for
if expected disconnection period is reached then
  randpeer = SelectRandomPeer(neighbors - sendpeers);
  ForwardQuery(Q, randpeer);
  RouteRecursiveLearn(count,
    sendpeers ∪ randpeer, threshold);
end if
```

Algorithm 9 Rank Decision Algorithm

```
RankDecision(Maps M, int NumberOfNeighboors,
        int WindowSize)
currentNeighbor = 0;
for j = 1 to NumberOfNeighboors do
  rank = Length(M);
  neighborProb = 1;
  for i = 1 to Length(M) do
    allUnknown = true;
    for k= M[i]-(WindowSize / 2) to M[i]+WindowSize / 2 do
      if FeatureSpaces[j][k] != -1 then
        currentProb += FeatureSpaces[j][k];
        allUnknown = false;
      end if
    end for
    if allUnknown then
      rank -= 1;
    end if
    if currentProb == 0 then
      rank = -1;
      break;
    end if
    neighborProb *= currentProb;
  end for
  if rank != -1 then
    allProbs[rank].Add(j, neighborProb);
  end if
end for
return allProbs;
```

query hit messages, then for popular keywords the scheme may decide to drop the queries. For example, assume in the training phase the scheme has received query "matrix avi" and this query hasn't generated hit messages. "Avi" here is a popular keyword and have a high probability of being repeated. Now lets assume a query "eternal sun shine of the spotless mind avi" is received. Since "avi" is seen before and no answers has received, the query will be dropped. However, the keywords other then "avi" are not seen before (unknown to the node), thus making a decision based on only one keyword generates a fallacy. To fix this, we change the decision algorithm to count keywords that are unknown and that did not generate hit messages. If the unknown keyword count is greater than the number of keywords that did not generate a query hit message, the query is broadcast. If, on the hand, the keywords that did not generate hit messages is greater, then the query is dropped. We call this modified version of the Route Learning $VoteRouteLearning$.

Another method, which we did not include in our experiments, for calculating the set of peers to forward the query to is to use a threshold value. For the highest rank, the algorithm can find the neighbors with containing probabilities greater then a threshold value and these neighbors should receive the query. If no such neighbors are found, then the algorithm can move to a lower rank.

5.3.0.3 Phase 3: Recursive Learning

This phase of the algorithm allows a peer to probe for changes in the neighboring peers and adapt its routing knowledge accordingly. The peer can find out about a failure that occurred at one hop distance with the help of ping/pong messages, and on such failures the peer may want to connect to other peers. Upon establishing the new connection, the peer restarts the training phase of the algorithm only for the new connection. On the other hand, adapting to neighborhood changes at distances from $TTL = 1$ to $TTL = maxTTL$ value, requires more attention. For example, a peer at two hops distance can change one of its connections or can completely disconnect from the network. In these types of neighborhood changes, the peers' knowledge about their environments becomes outdated. Still, the peers

can update their knowledge by observing the query and query hit messages, since the neighborhood changes have a direct effect on the number of results returned to a query. So at certain time intervals, broadcasting the query, listening to query hit messages and retraining the Parzen windows according to the answers received would allow dynamic adaptation.

The naive method described above would fail if one considers the frequent disconnections of peers. A better way is to add a randomly selected peer from the list of neighbors that won't receive the query, to the list of peers that the query will be forwarded. This can be seen in Algorithm 8.

We present the "Recursive Learning" algorithm (Algorithm 10), to avoid unnecessary training of peers' Parzen Windows when there is no or slight change in the neighborhood. The algorithm starts by waiting for the query hit messages from all the peers that the query is sent. From the query hit messages, the real probabilities of finding the resources are calculated. Then these probabilities are compared with the probabilities obtained from the Parzen Windows estimation method. If the estimated probability is greater or less then the real probabilities by a threshold, then the peer is marked for retraining.

Algorithm 10 Recursive Learning Algorithm

```
ROUTERECURSIVELEARN(float ExpectedProbability,
        Set setOfPeersQuerySent, float Threashold, Maps M)
for i = 1 to LENGTH(setOfPeersQuerySend) do
  for j = 1 to LENGTH (M) do
    ex_count[j] = FeatureSpace[i][M[j]].AnswerCount /
      FeatureSpaces[i][M[j]].QueryCount;
  end for
  query_hit = RECEIVEQUERYHITMESSAGES(setOfPeersQuerySent[i],
    timeout);
  if timeout occured then
    for j = 1 to LENGTH(M) do
      prob[j] = 0;
    end for
  else
    for j = 1 to LENGTH(M) do
      prob = COUNTRESOURCES(query_hit.message) /
        FeatureSpace[i][j].QueryCount;
    end for
  end if
  for j = 1 to LENGTH(M) do
    if (ex_count[j] - threshold > prob[j]) || (ex_count[j] + threshold < prob[j])
    then
      MARKFORRETRAINING(setOfPeersQuerySent[i]);
      break;
    end if
  end for
end for
```

Chapter 6

Experiments and Results

In this chapter, we first describe the experiments that we have conducted to evaluate our *Route Learning* scheme, and then detail our findings. While discussing the experiments, we also describe our methodology of evaluation and simulation model.

6.1 Experiments

Route Learning protocol has too many parameters and testing each of these parameters at a simulator with many nodes may lead to a very uncontrolled experiment. To overcome this, we have decided to simulate first a small but representative part of a peer-to-peer system (Figure 6.1). The simulator has one Route Learning enabled *root* node with six neighboring nodes, one of which is assumed to be connected to the whole network and queries come from this node. Other five nodes are leaf nodes that share resources. With this setup we have conducted the following experiments.

1. *Exact match error experiment:* This experiment aims at proving that the protocol works for repeated queries, and also tries to find the value for the window size parameter. If this parameter is set to be a large number, then

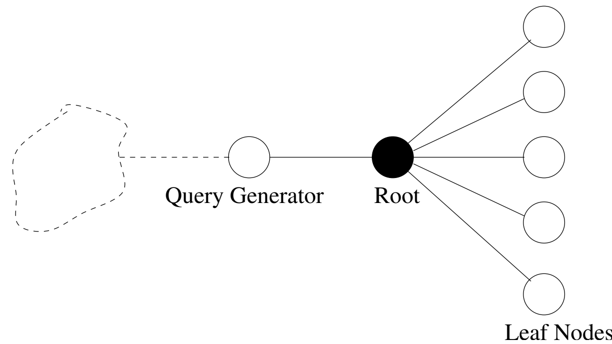


Figure 6.1: Simple P2P Network Model.

the protocol would estimate incorrect probabilities and forward the query to wrong peers even in exactly repeating queries.

The experiment works as follows: each leaf node randomly selects 20 queries from the query data collected by our Gnutella crawler. The keywords of those 20 queries selected by a leaf node are then associated with simulated resources shared by that leaf node. In this way, the leaf node can successfully answer the queries containing those keywords.

In the training phase, the "network" queries each of these resources. Then in the evaluation phase, these resources are re-queried and the *root* node tries to determine the peer managing the resource. However, the root node is only allowed to send the query to the node that has the highest containing probability and rank. If the *root* node evaluates the peer managing the resource incorrectly, declares the resource as not contained in the network, or broadcasts the query (which is not a desired behavior), then the simulator marks this as an error. The test continues until all the queries in the collected data are used.

2. *Modification error experiment:* This test aims at determining the protocol's behavior on close keywords and show how rank decision algorithm improves the protocol's decision making process. The test environment is similar to the previous test, however, in the evaluation phase the "network" modifies at least one keyword in the query by adding random characters at the end of each keyword. We have specifically not used random locations while

padding the random characters because this may produce completely irrelevant keywords and broadcasting may not become an error. The definition of the cases that are declared as errors are the same as in the previous test.

3. *Gnutella run:* In this experiment we measure the bandwidth usage of Routing Learning scheme and compare it to pure broadcasting. The "network" issues queries to the root peer in the order they are received by the crawler. The root peer trains itself with the first n queries it has received, and then starts the evaluation phase, in which $toSendPeers$ parameter is set to 1. We try the test with different values of n to show how training affects the decision making process.

Simulating a P2P network is considered to be a thumbling block since there are many parameters to consider and using unrealistic approximations for one of these parameters may produce unrealistic results. Especially the performance of Route Learning scheme greatly depends on the dynamic behavior of peers; frequent disconnections may cause the "knowledge" of the peers to become outdated. For this reason, after finding a good parameter set, we have decided to evaluate the scheme by modifying our Gnutella crawler to run Route Learning scheme virtually over flooding. As a normal Gnutella client, the crawler starts searching for other peers to connect to and after connecting to 4 peers the crawler starts executing the Route Learning scheme. For each received query, the crawler logs the decision made, which can be to drop, broadcast or forward the query, and the bandwidth used by the scheme. After this loging step, the query is flooded. Upon arrival of a query hit message, the crawler logs the number of answers received and the neighbor the answer came from. From these logs we extract the following results:

1. *Bandwidth used:* The bandwidth used, in terms number of messages, of the scheme is measured according to the decisions it has made. If the query is dropped then the bandwidth used is marked as 0, if on the other hand the query is broadcast the bandwidth used is set to the number of neighbors the crawler has. We have set the $toSendPeer$ parameter to one, thus when a query is forwarded it is forwarded to only one neighbor which in turn uses one message.

2. *Answer Rate:* It is obvious that for some queries Route Learning scheme is going to return less results then flooding since some queries are dropped or forwarded to only one neighbor. Thus it is important to compare how Route Learning reduces the answer rate of flooding. As indicated, the logs include the answers could have returned if the scheme is used and the answers returned by flooding. By examine the answers returned columns of the logs we are able to find out and compare the answer rate of Route Learning to flooding.

It is indicated by [4] that only 10% of queries in Gnutella receive a response message. This low response rate may not directly show the effects of using Route Learning instead of flooding on responses. For this reason have selected to program a simulator in which responses to queries are greater then Gnutella. To generate an environment that is similar to Gnutella, we have used queries and pong messages collected by our crawler. Each peer in the simulation acts as a peer collected in the pong messages. This way a close to real life scenario is generated in terms of the number of items the peers is sharing. Studies show that P2P networks' topology exhibit a power-law network. To add this behavior in our simulations, we have used the "Power Law Out Degree" algorithm presented in [17]. Each peer randomly selects and issues a query from the 30000 queries collected by the crawler. Peers also select the resources they are going to share during the simulation from these queries. The TTL values of the queries are set to 4 since 91% of queries are submitted to the network with TTL 4. We ran the simulation with flooding, Route Learning, Vote Route Learning and random walk based protocols. For each submitted query the simulator logs the number of results returned to query and the bandwidth used.

6.2 Results

In this section we present the results and evaluate the scheme. However, before going on to the results we describe the parameters used for the Route Learning scheme. For Radix 32 based dictionary function, the cell separation distance is set

to 32^{-5} since the average keyword length is 5. Because of this, the system requires an array of 32^5 cells. However a better implementation, one that uses hash tables instead of arrays, as shown in Algorithm 11, can overcome this requirement. When we inserted 6800 queries, collected by our Gnutella crawler, to a hash table with size 32^4, we achieved a 40% load factor. As a result, the hash table is initiated with size set to 32^4, and a peer is allowed to re-hash if 90% load factor is achieved for the hash table.

Algorithm 11 Hash-table version of the feature_space operations

```
RouteLearnInsertToFeatureSpace(hashtable feature_space,
        int AnswerCount, int QueryCount, Maps M)
toInsert = CreateCell(AnswerCount, QueryCount);
for i = 1 to Length(M) do
  feature_space.HashTableInsert(M[i], toInsert);
end for

RouteLearnCalculateContainingProb(hashtable feature_space,
        Maps M, int windowSize)
qprob = 1;
for i = 1 to Length(M) do
  prob = 0
  for j = M[i].data - windowSize / 2 to M[i] + windowSize / 2 do
    fromTable = feature_space.HashTableLookUp(j);
    if fromTable != NULL then
      prob += fromTable.answerCount / fromTable.queryCount;
    end if
  end for
  qprob *= prob;
end for
```

The Figure 6.2 shows the number of incorrect routing decisions made in a system using *Hash-1* for the exact match experiments conducted with 6800 queries. As can be seen from the figure, as the window size increases the number of errors increases significantly. With large window sizes the increase is much sharper. When window function of size 10000 is used, for example, the number of errors becomes 2428 (out of 6800 queries). This corresponds to an unacceptable error rate. Therefore, using a window size of 10000 is not suitable for a system with

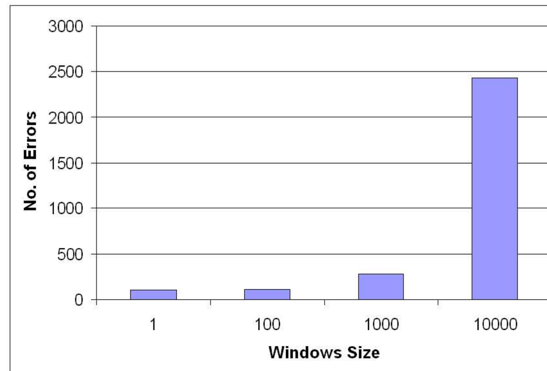


Figure 6.2: Number of errors made by the dictionary function in exact match experiment conducted with 6800 queries.

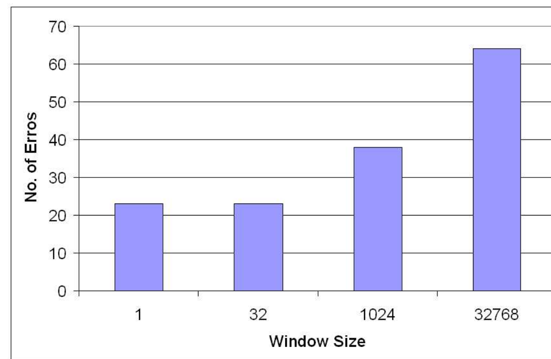


Figure 6.3: Number of errors made by the Radix 32 based mapping function in exact match experiment conducted with 6800 queries.

parameters that have values as in this experiment. Even with window size that are much smaller than 10000, the error rate is still too high. This means that the system is placing unrelated keywords in close proximity and this causes incorrect decisions to be made. Such high error rates in exact match queries are not acceptable for route learning scheme to be usable in practice.

Figure 6.3 shows the behavior of the scheme on exact match queries when Radix 32 based dictionary function is used. The graph in the figure has a similar pattern as of the Figure 6.2. But the number of errors is significantly reduced in this case, indicating that Radix 32 is better in placing unrelated keywords into

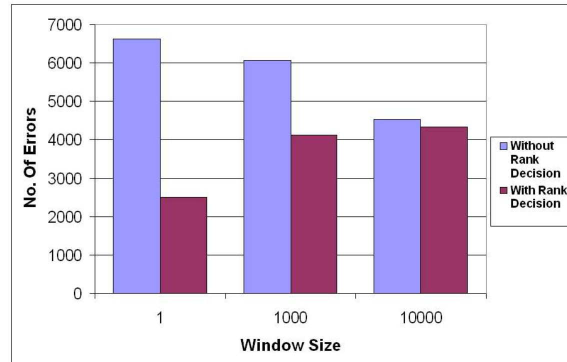


Figure 6.4: Dictionary function behavior on modified queries; experiment conducted with 6800 queries.

a feature space. The sharp increase in number of errors with a window size of 32768 recalls the sharp increase in number of errors noticed when the dictionary function of Figure 5.4 with a windows size of 10000 is used. The absolute increase with Radix 32, however, is not as much as the increase with dictionary function of Figure 5.5. Therefore, in a system using Radix 32 based mapping function, we can not immediately declare the window size of 32768 as not suitable for the parameters we have used.

The results of the modification experiments conducted with 6800 queries are shown in Figures 6.4 and 6.5. With window size equal to 1, the dictionary function has predicted the routes of almost all the queries incorrectly. This is an expected result since with such a small window size the system will perform just like a hash function. But interestingly, increasing the window size does not significantly decrease the error rate. The benefits of the rank decision algorithm (Algorithm 9) can be seen clearly from the Figure 6.4, however, the results are not satisfactory. *Hash-1* was also performing unsatisfactorily in the exact query experiments. Therefore, we conclude that the hash function of Figure 5.4 does not perform well in our Route Learning scheme and we exclude that function from the rest of the experiments.

While discussing the Radix 32 based mapping function, we have mentioned

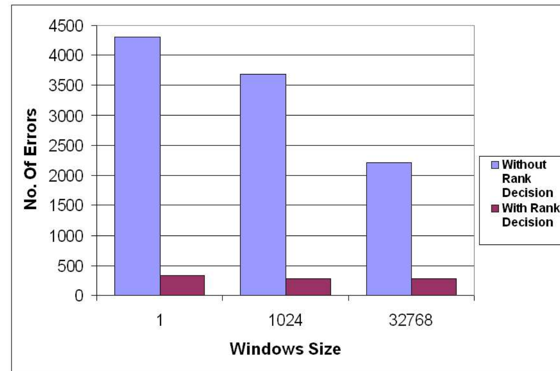


Figure 6.5: Radix 32 based mapping function modification behavior; experiment conducted with 6800 queries.

that the immunity of the system to modified queries greatly depends on the window size used. This prediction can easily be seen in Figure 6.5 since increasing the window size decreases the number of errors significantly. The high number of errors recorded without the usage of rank decision algorithm are caused by keywords that are shorter then the window sizes coverage area. For those keywords, the system has declared the queries that include these keywords as not trained; but this is not the case. It is important to notice that the system has predicted the routes of 2493 queries out of 6800 queries correctly although the window size was 1. This is because these queries are made of keywords whose length is equal to or more than 5 characters, so the Radix 32 mapping function has simply ignored the padding, and therefore made the correct decision. With rank decision algorithm, Radix 32 based function has scored even better results compared to the dictionary function. There is a slight increase in the number of errors when a window size of 32768 is used. Therefore, considering the execution times and the errors, we have decided that 1024 is a good value for the window size. We use this in the experiments we will discuss next.

Figure 6.6 shows the results of the "Gnutella run" test conducted with Radix 32 based mapping function with window size of 1024. If broadcasting is used on the system of this experiment, the bandwidth required, in terms number of messages, would be *34000 (5*6800)*. With route learning method, however, only

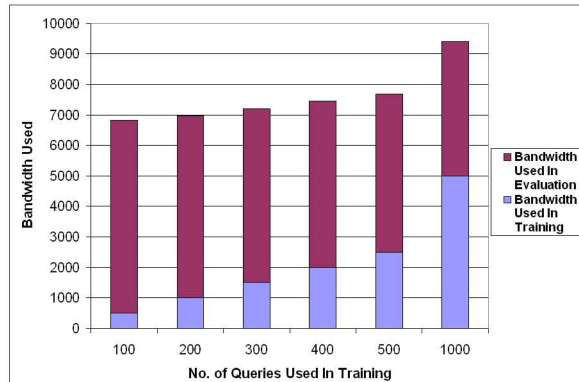


Figure 6.6: Bandwidth Used in Gnutella Experiment conducted with 6800 queries; 90% reduction to flooding requirements

about 20% of this bandwidth is used in the system. Hence improvement over broadcasting is clear. As the system gets trained with more queries, the bandwidth wasted in the evaluation phase is reduced. This is because the system starts declaring less queries as not trained when the training count increases. Although training the system with more queries reduces the bandwidth used in the evaluation phase, reductions becomes insignificant after some point. In Figure 6.6, for example, when the system is trained with 1000 queries, we see that less bandwidth is used in the evaluation phase, this is, however, not a significant improvement compared to the bandwidth used in the evaluation phase of a system that is trained with 500 queries. From this we can conclude that training the system with large number of queries does not improve the system's overall performance significantly to justify the cost of training. Hence, by looking to Figure 6.6, we can say that training the system with around 100 queries is adequate.

We have collected 100000 queries in 6 hours for the Gnutella tests. The peer's life time analysis conducted by [16] states that most of the session durations are short and the median duration is found to be 60 minutes. Thus by crawling the Gnutella network for 6 hours guaranties neighborhood changes. In fact the change can easily be seen in Figure 6.7, as the time increases the inter arrival of query messages also increases resulting in the decrease in flooding's bandwidth usage. This decrease can be caused by disconnection of peers. In Figure 6.7 the

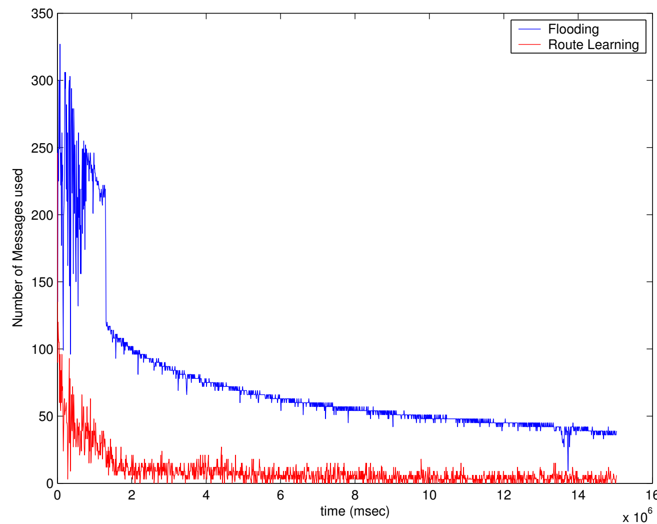


Figure 6.7: Bandwidth used by flooding and Route Learning.

bandwidth used in terms of number of messages is plotted against time. As can be seen, at the initial phase Route Learning uses the same bandwidth as the flooding; this is because of training. However, as soon as training phase finishes, Route Learning's bandwidth usage drops sharply below flooding. One can explain this sharp decrease with keyword repetitions. Since most of the keywords seen during the training phase is also seen while evaluating, and since for these keywords no query hit message is generated, Route Learning drops the queries containing these keywords. In our experiments, out of 100000 queries only 13 have received a query hit message and route learning in the evaluation phase was able to find answers for 5 of these queries, yielding a %38 answer rate.

To find better understanding of the answer rate we have programmed the simulator to return answers to %30 of the queries submitted to the network; which is 3 times the ratio one can find in Gnutella [4]. We ran the simulation with 100 peers and each has submitted 100 queries uniformly selected from the 30000 queries collected by the crawler. We have set Route Learning $toSendPeer$ parameter to 1 so that Route Learning is allowed to send the query to the neighbor with the highest rank and containing probability. To induce comparable results, we have set the random walk based protocol to send the query to only one neighbor.

From the total 10000 queries flooding has returned answers to 3000 of them.

Protocol	Number of Answers Returned	Answer Rate	Bandwidth Used
Route Learning	1285	~42%	~23%
Vote Route Learning	1853	61%	45%
Random Walk	210	7%	2%

Table 6.1: Simulation results.

In Table 6.1 the number of answers returned and answer rate of each protocol are shown. Route Learning was able to return 42% of the answers returned by flooding. This answer rate is close to the one that was observed in Gnutella. However, Route Learning was able to record this answer rate by using 23% of the bandwidth used by Gnutella. By doubling the bandwidth used, Vote Route Learning was able to return answers to 61% of the queries.

Chapter 7

Conclusion

The contribution of this thesis is two fold:

- Derivation of models and characteristics information for some of the important parameters of Gnutella, an unstructured P2P network that is very popular and widely used;
- Providing a scheme called *Route Learning* to reduce the query overhead in flooding based P2P networks like Gnutella, and performing a thorough evaluation of the proposed scheme via extensive simulation experiments and also via experiments done by using a custom-built crawler attached to the Gnutella network.

The derivation of models and characteristics information of some important Gnutella parameters is done by collecting real network traces via our crawler attached to the live Gnutella network used by thousands of people. As already mentioned by several studies, we have verified that a large portion of Gnutella protocol messages seen on a Gnutella network is constituted by Query messages which are disseminated through a simple and inefficient flooding mechanism. This clearly indicates the need for more clever algorithms for disseminating queries in unstructured P2P networks to reduce the messaging overhead and to provide better scalability.

Our results also indicate that most submitted queries contain query strings that consist of multiple keywords, as opposed to the common assumption in various simulations that a query consists of a single keyword. We also found that repetition count of keywords seen in a P2P network obeys a power-law distribution with respect to the rank of keywords where the keyword that is repeated the most has a rank of 1. We also verified the fact that not all peers contribute to a P2P network at the same level. A small portion of peers share a large portion of all files available in the network. Our traces also revealed the fact that the same query string is not repeated too much by the same peer. Also a peer does not increase the initial TTL (time-to-live) values of repeated queries to enlarge the search horizon. We have found that most submitted queries have an initial TTL value of 4, and even though a peer submits a query with a larger TTL value, the neighboring peers immediately reduce the TTL to a value below 4.

We think that our findings can be important for P2P network simulation studies that are looking for models and information about some of the important parameters of P2P networks.

As the second contribution of the thesis, we propose the *Route Learning* scheme, as an adaptation of a classification problem to P2P networks, for reducing query overhead in flooding based unstructured P2P networks like Gnutella. In this scheme, peers gradually built knowledge about their environment, so that they can predict the neighbors where an answer can come from. This process results in less bandwidth usage in querying. The main difference of the scheme presented from a classification problem is that it continuously tries to adapt its knowledge in order to cope with frequent changes occurring in the neighborhood. Therefore, the scheme's learning period never ends, as it is the case in Bayesian Recursive Learning. Route Learning's efficiency in predicting the routes greatly depends on the repetition of the keywords. The analysis of P2P networks have shown that keywords submitted to the network show a Zipf distribution. Therefore, a high number of repetitions on some keywords is expected. However, Route Learning also considers the similarity between keywords and tries to relate these keywords in order to save bandwidth. Due to the Zipfian distribution of keywords in queries, which is shown in Figure 4.2, by relating close keywords, *Route*

Learning is able to search a keyword at different regions of the popularity graph. For example, from out traces we have found that "Matrix" is a popular keyword; however, "MatrixDVD" is not that popular. By relating "MatrixDVD" with "Matrix" *Route Learning* is able to map a not so popular keyword to a very popular keyword and return answers to this not so popular query without the need to broadcast the query.

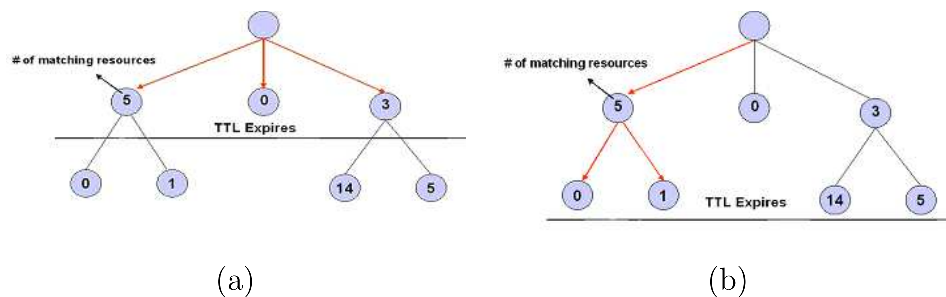
We have considered a very simple similarity metric, the distance between keywords when they are mapped to the feature space. The results of the modification test have shown that Route Learning most of time relates close keywords with each other and forwards the query to only one neighbor. Better mapping functions like Soundex could give better results, thus we are considering in conducting further experiments. Overall, as we describe in Chapter 8, the scheme presented in this thesis can be improved in many ways, but even in its simplest from, Route Learning greatly saves bandwidth as can be seen in Figure 6.6.

Chapter 8

Future Work

As we have mentioned in our conclusions, we are considering to test the performance of the Route Learning scheme with different mapping functions. We specifically want to test the scheme with *Sound-Ex* algorithm, since it is widely accepted for relating close keywords. For example, many libraries today number and sort their books according to the output of the Sound-Ex algorithm.

Route Learning in its current state does not consider TTL values in user submitted queries and leaves the updates to *recursive learning* phase of the protocol. The fallacy behind this approach can be best described with the following scenario (Figure 8.1-a and 8.1-b). Assume we have a P2P network setup shown as Figure 1, and in the training phase of the algorithm, the *root* node receives query



(a) (b)

Figure 8.1: Fallacy scenario: a) training phase, b) evaluation phase.

$q1$ which has a TTL value equal to 1. The root node broadcasts the query and the node 2 returns the largest result set for query $q1$. However, the siblings of the root node do not forward or broadcast the query, since the TTL value of the query expires. In the evaluation phase, assume that the same query with a TTL value equal to 2 arrives at the root node which falsely estimates the route of the query as node 2, since node 2 had returned the largest result set for $q1$. However, at TTL level 2, node 5 has the largest number of resources matching the query. For a solution to this scenario, we are considering using the TTL levels of queries as a second feature. In this way, the feature space becomes 2-dimensional.

Besides the above issue, the Route Learning scheme only addresses the query overhead problem in Gnutella network. Many other P2P networks, however, may use the Route Learning scheme as well. For example, the Kazaa network uses a random walk based protocol for routing the queries between super-nodes. Using Route Learning for Kazaa, however, may greatly increase the answer count returned in queries. Thus we are considering conducting performance experiments on use of Route Learning in Kazaa network as well.

Bibliography

[1] Gnutella protocol v0.6. Available at http://rfc-gnutella.sourceforge.net/developer/testing/index.html.

[2] A. T. Stephanos, “A Survey of Peer-to-peer File Sharing Systems”, WHP-2002-03, Athens University of Business and Economics, 2002.

[3] Kazaa http://www.kazaa.com

[4] E. P. Markatos, “Tracing a large scale Peer-to-Peer System: An hour in the life of Gnutella”, IEEE/ACM Int. Symp. on Cluster Computing and the Grid, 2002.

[5] I. H. Witten, A. Moffat and T. C. Bell, “Managing Gigabytes: Compressing and Indexing Documents and Images”, Second Ed. Morgan Kaufmann,1999.

[6] M. Harren, J. M. Hellerstein, R. Huebesch, T. B. Loo., S. Shenker. and I. Stoica, “Complex Queries in DHT-based Peer-to-peer Networks”, IPTPS LNCS 2429, 2002.

[7] C. Rohrs, “Query Routing for the Gnutella Network”, available at http://rfc-gnutella.sourceforge.net/src/qrp.html

[8] A. D. Menasce, L. Kanchanapalli, “Probabilistic Scalable P2P Resource Location Services”, ACM Sigmetrics Performance Evaluation Review, vol. 30, no. 2, 2002, pp. 48-58.

[9] N. C. Hang, S. K. Cheung and C. C. Hang, “Advanced Peer Clustering and Firework Query Model in the Peer-to-Peer Network”, International World Wide Web Conference, 2003.

[10] S. Joseph. “NeuroGrid: Semantically Routing Queries in Peer-to-Peer Networks”, International Workshop on Peer-to-Peer Computing, 2002.

[11] S. Joseph, “P2P MetaData Search Layers”, International Workshop on Agents and Peer-to-Peer Computing, 2003

[12] M. T. Schlosser, T. E. Condie and S. D. Kamvar, “Simulating a File-Sharing P2P Network”, Workshop on Semantics in P2P and Grid Computing, 2002.

[13] D. Zeinalipour-Yazti and T. Folias, “Quantitative Analysis of the Gnutella Network Traffic”, TR-CS-89, Dept. of Computer Science, University of California, Riverside, 2002.

[14] S. Saroiu,P. K. Gummadi and S. D. Gribble, “A Measurement Study of Peer-to-Peer File Sharing Systems”, Multimedia Computing and Networking, 2002.

[15] O. D. Duda, E. P. Hart and G. D. Strok, “Pattern Classification”, Second Edition, Wiley Interscience, 2000.

[16] M. Ripeanu and I. Foster, “Mapping the Gnutella Network -Macroscopic Properties of Largescale P2P Networks”. IEEE Internet Computing Journal, 6(1), 2002.

[17] C. R. Palmer and J. G. Steffan, “Generating Network Topologies That Obey Power Laws”, Global Internet Symposium, 2000.

[18] M. Karakaya, I. Korpeoglu, O. Ulusoy, “GnuSim: A General Purpose Simulator for Gnutella and Unstructured P2P Networks”, Technical Report BU-CE-0505, Department of Computer Engineering, Bilkent University, August 2005.

[19] S. Ciraci, I. Korpeoglu, O. Ulusoy, “Characterizing Gnutella Network Properties for Peer-to-Peer Network Simulation”, International Symposium on Computer and Information Sciences, 2005.