University of Louisville
ThinkIR: The University of Louisville's Institutional Repository

Electronic Theses and Dissertations

5-2010

Architecture analysis of peer-to-peer network structure and data exhanges for distribution of contraband material.

Nicholas Miles
University of Louisville

Follow this and additional works at: https://ir.library.louisville.edu/etd

Recommended Citation

Miles, Nicholas, "Architecture analysis of peer-to-peer network structure and data exhanges for distribution of contraband material." (2010). *Electronic Theses and Dissertations.* Paper 974.
https://doi.org/10.18297/etd/974

This Master's Thesis is brought to you for free and open access by ThinkIR: The University of Louisville's Institutional Repository. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of ThinkIR: The University of Louisville's Institutional Repository. This title appears here courtesy of the author, who has retained all other copyrights. For more information, please contact thinkir@louisville.edu.

ARCHITECTURE ANALYSIS OF PEER-TO-PEER NETWORK STRUCTURE AND DATA EXCHANGES FOR DISTRIBUTION OF CONTRABAND MATERIAL

By

Nicholas Miles

A Thesis Submitted to the Faculty of the

Graduate School of the University of Louisville

in Partial Fulfillment of the Requirements

for the Degree of

Master of Engineering

Department of Computer Engineering and Computer Science
University of Louisville
Louisville, Kentucky

May 2010

ARCHITECTURE ANALYSIS OF PEER-TO-PEER NETWORK STRUCTURE AND DATA EXCHANGES FOR DISTRIBUTION OF CONTRABAND MATERIAL

By

Nicholas Miles

A Thesis Approved on

August 2, 2010

by the following Thesis Committee:

____

Thesis Director: Dr. Olfa Nasraoui

____

Dr. Ibrahim Imam

____

Dr. Michael Losavio

____

Dr. Dr Ayman El-baz

ABSTRACT

Architecture Analysis of Unstructured Peer-to-Peer Networks

By

Nicholas Miles

May 10, 2010

Because of the anonymity that P2P networks provide, they are an ideal medium for the exchange of contraband material such as child pornography. Unfortunately, not much research has been conducted on how to best monitor these types of networks for contraband searching and sharing activity. This thesis proposes techniques to advance the state of the art in peer to peer data exchange monitoring and detection of nodes that participate in distributing and sharing contraband material.

Because of the legal considerations in working with a live P2P network and the technical difficulty in developing and testing a "surveillance" system for P2P networks, a simulator was developed that attempts to accurately simulate the behavior of users on P2P networks based upon empirical data collected from several researchers.

With the help of the simulation platform that has been developed, a complete methodology for monitoring "contraband" activity and reporting the most prolific contraband users has been created. This methodology, if implemented on an actual P2P network, should allow the detection of members of the network who are the most active sharers and distributors of contraband material.

Contents

1 INTRODUCTION **1**

1.1 Motivation . . . 1

1.2 Objectives . . . 2

1.3 Thesis Contribution . . . 3

1.4 Summary . . . 4

2 LITERATURE REVIEW **5**

2.1 Overview of Gnutella Network Protocol . . . 5

2.1.1 Gnutella v0.4 Packet Structure . . . 7

2.1.2 Query Routing in Gnutella . . . 7

2.2 P2P Network Data Characterization . . . 10

2.2.1 Live Network Data Capture . . . 10

2.2.2 Contraband Material . . . 11

2.3 P2P Network Structure . . . 12

2.4 File Propagation in P2P Networks . . . 13

2.4.1 Neighborhood Size . . . 13

2.4.2 The Compartmental Model for P2P Networks . . . 14

2.5 P2P Network Usage Behavior . . . 15

2.6 Finding Good Peers . . . 16

2.6.1 Metrics . . . 16

2.6.2 Peer Selection Algorithm . . . 18

2.7 Summary 18

3 METHODOLOGY **20**

3.1 Overview of Proposed Methodology 21

3.2 Estimation model for the number of needed listening nodes 22

3.3 Simulator Overview 23

3.4 P2P Network Generation and Initialization 27

3.4.1 P2P Network Architecture Generation 27

3.4.2 P2P Node Classifications 30

3.4.3 P2P Network File Distribution 32

3.5 P2P Network Behavioral Simulation 32

3.6 Listening node behavior and actions 35

3.7 Simulator Data Capture 35

3.8 Listening Node Placement Algorithm 37

3.8.1 percQueryHits Function 37

3.8.2 Importance Function 37

3.9 Node Ranking Algorithm 40

3.10 Conclusion 40

4 RESULTS **42**

4.1 Simulator Validation 42

4.1.1 Graph Generation 43

4.1.2 File Distribution 45

4.1.3 Simulation Execution 46

4.1.4 Query Routing 49

4.2 P2P Network Surveillance Results 53

4.2.1 Estimation of the Number of Listening Nodes Needed 53

4.3 Listening Node Placement Algorithm Results 57

4.4 Contraband Node Ranking 59
4.5 Conclusion 60

5 CONCLUSION 62

5.1 Publications 62
5.2 Limitations 62
5.3 Future Directions and Goals 63
5.4 Final Remarks 64

A Simulator Source Code 65

A.1 main.py 65
A.2 filePool.py 67
A.3 node.py 73
A.4 p2pClient.py 77
A.5 p2pSim.py 80
A.6 priorityQueue.py 82
A.7 gnuTellaNetwork.py 84
A.8 graphGenerators.py 86
A.9 queryLogger.py 89
A.10 configure.py 93
A.11 stats.py 95
A.12 staticVars.py 97
A.13 nodeRanking.sh 98
A.14 rankOutput.py 98

List of Figures

2.1 Pictorial representation of structured vs. unstructured P2P networks. 6

2.2 Packet forwarding in an unstructured P2P network 9

2.3 Diagram of a sample P2P Network 16

3.1 Node labels after Dijkstra's algorithm has executed. With TTL=2, reachable nodes are colored in yellow. 23

3.2 Import graph for p2p simulator 26

4.1 Log-Log graph of degree rank plot for 5000 node random graphs generated using unconstrained and modified constrained Barabasi algorithm. 43

4.2 Small graph generated using the unmodified Barabasi algorithm . . . 44

4.3 Log-log plot for the degree-rank of the file distribution 45

4.4 Partial call graph for simulator run function 47

4.5 Partial call graph for doQuery function. 48

4.6 Call graph for p2pClient 50

4.7 Diagram of a p2p network to test the query routing code 51

4.8 Nodes viewable vs. listening nodes with TTL=4 (average from 100 different runs for each number of listening nodes) 54

4.9 Nodes viewable vs. Listening nodes with TTL=5 (average from 100 different runs for each number of listening nodes) 56

4.10 Effect of the placement algorithm on the number of contraband queries intercepted 58

4.11 Report composition vs. actual network composition 60

List of Tables

2.1 Descriptor Header - Gnutella Protocol 7

2.2 P2P Protocol Descriptor Descriptions 8

2.3 QUERYHIT Descriptor 8

2.4 Media type mappings from [4] 10

2.5 Traffic relating to illegal material, from [5] 11

2.6 Compartmental Model Classes from [7] 14

2.7 Metrics calculated for RMK algorithm. 17

3.1 Experimental design for listening node estimation experiment 24

3.2 Simulator Software Components 25

3.3 Configuration parameters 28

3.4 Behavioral Classes for Nodes based on data from [4] 31

3.5 NETWORK_MAKEUP ranges 31

3.6 Example mean values for time between searches for each class 33

4.1 Percentage of nodes viewable with TTL=5 56

4.2 Summary of data from testing the listening node placement algorithm 57

4.3 Percision for node ranking algorithm 61

List of Algorithms

2.1 Peer Selection Algorithm 19
3.1 Determining the reachable nodes from more than one listening node . 24
3.2 Modified Barabasi Algorithm 29
3.3 Pseudo code for query search activity. 34
3.4 Pseudocode for percQueryHits function. 38
3.5 Pseudocode for averNumHops function. 39
3.6 pseudo code for Importance function. 39
3.7 Node Ranking Algorithm 41

CHAPTER 1

INTRODUCTION

1.1 Motivation

Unstructured Peer-to-Peer (P2P) networks allow for the anonymous, un-filtered, and un-regulated transmission of content between multiple parties. This content can include anything that can be digitally stored - video , audio, software, text documents - and shared without any supervision from a governing body. These dynamic P2P networks can contain thousands to millions of individuals peers. All a peer needs to connect is one of many freely available software clients that can be downloaded from the Internet.

Because of the anonymity of a P2P network, it provides a perfect environment for the exchange of illegal contraband such as child pornography and the unauthorized distribution of copyrighted works, all without fear of intervention by law enforcement. Even documents containing secret information related to national security or corporate trade secrets have been leaked out over P2P networks either intentionally or accidentally due to mis-configured P2P software clients. A recent example is the leakage of the blue prints for Marine One (a helicopter used the transport the President of the United States of America) [12].

It is therefore becoming more important to understand how these networks are

structured and how activity can be monitored and analyzed on them. The contributions of our research are important, both for developing active forensic techniques for law enforcement, and improving ways in which existing P2P protocols can be modified to increase anonymity.

1.2 Objectives

This thesis has two goals:

1. An analysis of the properties of a P2P network, and the properties of traffic activity under different conditions in varying information exchange scenarios.

2. An approach to monitoring traffic on a P2P network that will garner as much contraband network activity as possible. In order to accomplish this, a methodology for analyzing P2P network usage and the behavior of the peers within will need to be developed in order to target the most prolific contraband nodes.

Due to unknown legal considerations and risks when analyzing data from an actual P2P network, a simulation platform for an unstructured P2P network should to be developed. Large P2P networks have a substantial amount of activity and some P2P network protocols are quite elaborate. It is therefore important to determine which subset of protocol functions the simulator will need to implement.

Once a simulation platform has been created, a methodology will need to be developed for collecting P2P network traffic. A set of "listening nodes" will need to be used to intercept and collect P2P network traffic. A determination of the number of listening nodes needed to collect adequate data representative of the entire network will need to be made. Also, an optimal placement algorithm for listening nodes will need to be implemented in order to target contraband nodes and collect the most interesting traffic.

Once the best method for data collection has been determined, a method for acting upon the data collected will need to be created. Since P2P networks are designed to be anonymous, node probing techniques will need to be explored for discovering the IP addresses and identifying information of nodes.

Finally, results collected from the listening will need to be analyzed and a ranking algorithm will need to be developed to help determine the most prolific contraband nodes.

The specific objectives of this thesis can be summarized as follows:

- Garner a basic understanding of unstructured P2P networks, and their associated protocols.
- Develop a simulation platform that will accurately simulate the most important P2P protocol functions.
- Implement a listening node placement algorithm and test it for utility.
- Develop node probing techniques to discover identifying information of nodes.
- Develop a ranking algorithm for determining the most prolific contraband nodes.

1.3 Thesis Contribution

The major contribution of this thesis is a complete approach for monitoring a P2P network for contraband file sharing activity. This thesis will present a guide for best placement of listening nodes, a methodology for collecting identifying information using probing, and a way to rank contraband nodes to find the most prolific contraband sharers.

This thesis will also be concerned with developing a simulation platform that can be used to efficiently test listening node placement algorithms and active listening

techniques to discover contraband nodes. This simulator will be able to generate large graphs of connected nodes and simulate the behavior between those nodes.

The methodology proposed in this thesis is generalized so that its suggestions could be implemented on any real P2P protocol given enough research.

1.4 Summary

Chapter 1 showed that the motivation behind the proposed methodology were the need to identify nodes on unstructured P2P networks that may be sharing contraband. It will also point out the flaws in P2P protocols that can be exploited to unravel the shroud of anonymity that users believe P2P networks provide.

Chapter 1 also presented the main objectives and contributions of this thesis, that is to develop a methodology for collecting and analyzing P2P network traffic and developing a P2P network simulator that will allow the methodology to be evaluated.

Chapter 2 will present a review of previous work that will be useful in accomplishing the goals of this thesis. Chapter 3 will present the proposed methodology for both simulating and surveying a P2P network. Chapter 4 will cover results from the implementation of our proposed methodology. Finally, chapter 5 will present our final summary and future work.

CHAPTER 2

LITERATURE REVIEW

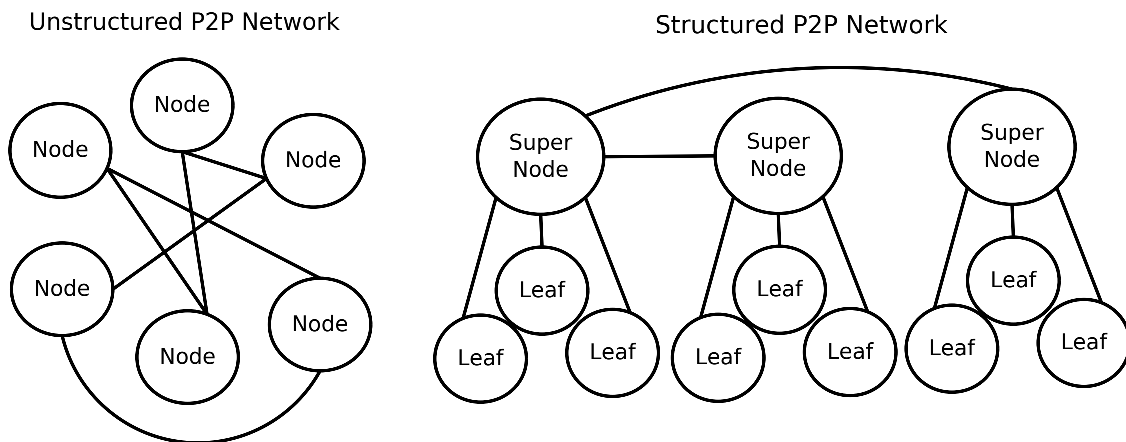
In chapter 1, the motivation, objectives, and proposed methodology for this thesis were covered. This chapter covers literature that describes the structure and composition of actual P2P networks, which is very important information needed to create an accurate simulator. This chapter also covers a novel node placement algorithm which would have great utility in placing listening nodes in the best position for collecting contraband traffic on a P2P network.

2.1 Overview of Gnutella Network Protocol

For the purposes of this thesis, we will focus our efforts on analyzing the Gnutella v0.4 version protocol. This protocol is completely unstructured unlike the new v0.6 protocol used in many freely available P2P clients [2, 3] today. An unstructured P2P network is defined as a network of arbitrarily connected nodes, in which all nodes have the same role.

In a structured P2P network such as Gnutella v0.6 the nodes may serve different roles. For example, in v0.6 a node can either be classified as a "super node" or a "leaf node". The super nodes are connected to each other and the leaf nodes are connected to the super nodes (see figure 2.1). When a user (on a leaf node) queries for a file, the query is sent to the user's respective super node and the super node relays the query

Figure 2.1: Pictorial representation of structured vs. unstructured P2P networks.



to its connected super nodes. Super nodes will only relay the query to leaf nodes that might possibly have a file that matches the query. This cuts down on broadcast traffic and makes the network more scalable.

The Gnutella v0.4 utilizes the TCP/IP protocol and operates at the application layer. File downloads are done outside the Gnutella network utilizing the hypertext transfer protocol (HTTP). The Gnutella network just facilitates searching for files.

Most P2P clients use some sort of a "host cache" service to find peers with which to connect. This is usually in the form of a centralized web server that contains an arbitrary list of nodes allowing connections. Most P2P software clients have a "phone home" mechanism built into them that updates the list on the central server. Once a peer finds hosts to connect to, they establish TCP connections to those hosts. When the TCP connection is created, an ASCII encoded connection string containing the protocol version is sent:

```
GNUTELLA CONNECT/.4\n\n
```

The host should respond with the following ASCII encoded string:

```
GNUTELLA OK\n\n
```

Table 2.1: Descriptor Header - Gnutella Protocol

Field	Description
Descriptor ID	16-byte unique ID for descriptor. Usually generated randomly by the client.
Payload Descriptor	0x00 = PING, 0x01 = PONG, 0x40 = PUSH, 0x80 = QUERY, 0x81 = QUERY HIT
TTL	Time to live value - 1 byte
HOPS	Hops value - 1 byte
Payload length	Length of payload - 4 bytes

2.1.1 Gnutella v0.4 Packet Structure

A Gnutella v0.4 packet consist of a descriptor header and a descriptor. A descriptor is what is passed among Gnutella clients to communicate with each other. The descriptor header is 22 bytes in length and contains five fields (see table 2.1).

The TTL and HOP fields in the Gnutella packet descriptor header control how deep into the P2P network the packet is allowed to travel. Each time a packet is relayed to another host in the network, the TTL field is decremented by one, and the HOPS field is incremented by one. When the TTL is equal to zero, the packet is no longer forwarded.

The ID field is generated randomly each time a packet is created by a node. It is a 16 byte random ID that uniquely identifies the packet descriptor. The ID field is primarily used to help route the Gnutella descriptors.

The Gnutella 0.4v protocol has five major descriptors for gnutella packets built into the protocol: QUERY, QUERY HIT, PING, PONG, and PUSH. [1] See table 2.2 for complete descriptions of each descriptor.

2.1.2 Query Routing in Gnutella

QUERY descriptors contain two fields: minimum speed, and search criteria. A QUERYHIT descriptor is used to respond to a QUERY descriptor and contains six

Table 2.2: P2P Protocol Descriptor Descriptions

Descriptors	Description
PING	Used to search for reachable hosts on the P2P network. A peer that receives a packet with a PING should respond with a PONG.
PONG	A PONG is sent in response to a PING. A PONG should contain information about how much a peer is sharing and the IP address of one other connected node.
QUERY	Used to search for files on the P2P network. A peer receiving a QUERY packet should respond with QUERYHIT if it has a file with a name matching the query.
QUERY HIT	Used to respond to a QUERY.
PUSH	Used to create a reverse connection with a peer sharing a file. This is used to bypass firewalls.

Table 2.3: QUERYHIT Descriptor

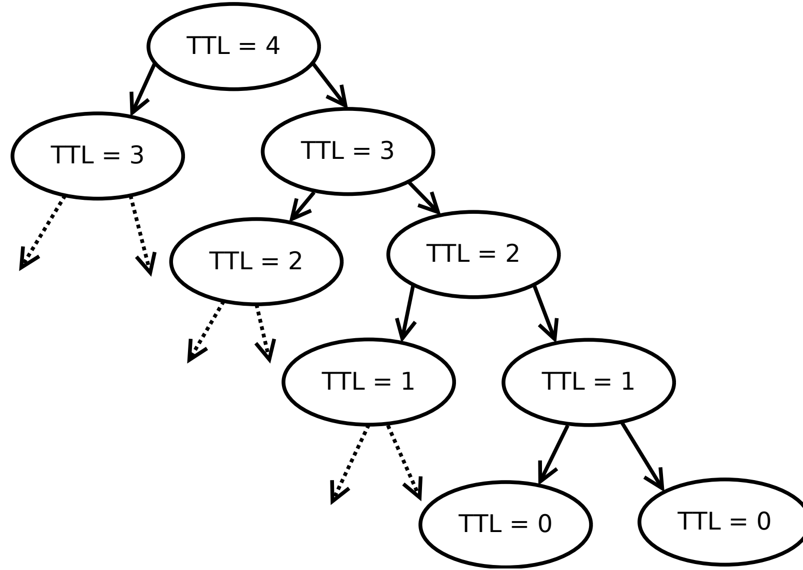
Field	Description
Number of Hits	Number of matching files in result set.
Port	Port that client can connect to via http to download files.
IP Address	IP address of responding host.
Speed	Bandwidth of responding host
Result Set	Filenames that match the query string.
Servant ID	Unique ID of the servant, for use with the PUSH descriptor.

fields (see table 2.3).

When a user wants to search for a file, it sends a packet with a QUERY descriptor containing the name of the file to search for in the search criteria field. It can also specify a value in the minimum speed field. The TTL in the descriptor is initialized to some value that should be around six since higher amounts will cause poor network performance, and HOPS is set to zero. The descriptor ID is initialized to some unique random value. The user forwards this packet to all its connected peers and those peers in turn forward the packet to all their connected peers. Each time the packet is forwarded, the TTL is decremented, until eventually reaching the value TTL=0, and the packet is no longer forwarded(see figure 2.2).

When a node responds to a query, it sends a packet with the QUERYHIT descrip-

Figure 2.2: Packet forwarding in an unstructured P2P network



tor. Nodes should only respond to the QUERY if their bandwidth is greater than what is specified in the minimum speed field, and if they have files matching what is in the search criteria field. The descriptor ID field in the descriptor header for the QUERYHIT descriptor is set to the same value as the QUERY descriptor ID. This packet is routed back along the same path that the query took to reach the node.

In order to route QUERYHIT descriptors back, each node keeps a table of descriptor IDs for QUERY descriptors it has received and which neighbor node it came from in a routing table. Nodes only forward QUERYHIT packets back to neighbor nodes for which they have a corresponding entry in the routing table. The QUERYHIT packet contains the IP address and HTTP port of the computer responding to the query and a result set of the files matching the query. The node that originally conducted the search can then connect to nodes that responded to the queries via a direct HTTP connection to download the file(s).

In order to speed up download time, many nodes hosting the same file may be connected to simultaneously and different sections of the same file can be downloaded from different nodes in parallel. Cryptographic checksums may be used to verify the

Table 2.4: Media type mappings from [4]

Media Type	File Extensions
Audio	mp3, m4a, wma, ogg,wav,flac
Video	mpg, mpeg, avi, wmv, qt, rm, ram, asf, divx, mov
Image	jpg, jpeg, gif, png, bmp
Document/Program	exe, vbs, bat, iso, pdf, zip, tar, gz, doc, xls, ppt, mdb, txt, csv

integrity of each file slice that is downloaded and the entire file as a whole once it is completely downloaded.

2.2 P2P Network Data Characterization

2.2.1 Live Network Data Capture

P2P networks are used to share a wide array of data types. These types include: audio, video, music, software programs, and documents. In order to accurately simulate behavior on the simulator, it is important to know what types of files are searched for most and how many of the queries are contraband queries.

In[4], the author attempts to characterize the data on P2P networks. In order to do this, queries are captured from a node passively listening to network traffic on a live P2P Gnutella network. A piece of custom software named “QueryCapture” was developed in order to accomplish this.

There are four well-known media types that are used (see table 2.4). These classifications are made based on the file extension found in the query string of the QUERY descriptor. According to these classifications, [4] found that around 1% of queries are for images, 2% are for docs/programs, 20% are for videos, 21% are for audio, and the remaining 56% are unknown and since they would not be classified. This is primarily because a majority of the query strings in the QUERY descriptor are hashed.

The paper mentions that 47.8% of all query strings on the P2P network are hashed using SHA-1. This makes determining the actual query string difficult because SHA-1

Table 2.5: Traffic relating to illegal material, from [5]

Reviewer	**Date**	**QUERY Descriptors**		**QUERYHITS Descriptors**	
		# Contraband	% Contraband	# Contraband	% Contraband
Reviewer 1	March 5	122	1.2	206	2.1
	March 12	156	1.6	295	3.0
	March 19	158	1.6	195	2.0
Reviewer 2	March 5	142	1.4	234	2.3
	March 12	184	1.8	297	3.0
	March 19	174	1.7	208	2.1

is an un-reversible one way hash. However, by keeping a database of collected plain text file queries and their associated hash values, it is possible to recover some of the unknown hashed file queries.

Additionally, queries were classified as non X-Rated or X-Rated. This was done by searching the query file strings for 44 keywords known to be associated with X-Rated material. According to the results, around 15% of the queries on a P2P network are X-Rated, with a standard deviation of 2.59%.

2.2.2 Contraband Material

A study was conducted in an IEEE research article in 2006 [5] to determine what percentage of behavior on P2P networks, and what percentage of network activity is related to the distribution of illegal child pornography. QUERY and QUERYHIT descriptors were collected and reviewed by two independent reviewers who helped classify which descriptors were related to illegal material (see table 2.5).

Empirical evidence shows that on average, 1.6% of queries, and 2.4% of query responses are for illegal child pornography. Furthermore, 57% of those nodes (chosen at random) where found to be hosting only child pornography contraband material. Only 17% percent of those nodes sharing child pornography contraband had less than 50% of their hosted content be something other than contraband. This evidence suggest that the nodes that are sharing illegal contraband are very active in

their sharing behavior, and that their primary purpose for using the P2P network is obtaining and/or distributing illegal child pornography contraband.

2.3 P2P Network Structure

To correctly simulate an unstructured P2P network, the topology of actual P2P networks must be understood. Several researchers [6, 7] who have created gnutella network crawlers that can extract a graph corresponding to the network topology in a particular period of time. They showed that the connectivity of the nodes in a P2P network follows a power law (zipf) distribution.

In order to determine if a network follows the power law of zipf distribution, a log-log degree rank (sorted list of number of node connections) plot is created. If the resulting plot is linear, then the distribution is most likely power law or scale free.

Many computer networks are scale free, such as the Internet topology, world wide web, and P2P networks. There are several algorithms available for generating scale free networks. One of the most well known is the Barabási–Albert model [8]. It is a preferential attachment algorithm, meaning that when new nodes connect to the network they have a higher probability to connect to nodes with high degree. More specifically, the probability of a new node connecting to node i is the degree of node i (k_i) divided by the sum of the degrees of all nodes in the network:

$$P(k_i) = k_i / \sum_j k_j$$

While P2P networks are roughly scale free networks, most P2P clients have a default limit on the number of connections that can be made. While most clients allow users to adjust the connection limit, a lot of users will not bother to change the settings. This may account for some of the "noise" in the data collected in [6]. This limit on the number of connections distinguishes P2P networks from true or ideal

power law networks.

2.4 File Propagation in P2P Networks

[9] proposes an analytical model for file propagation on the Gnutella P2P network. In this model, a node initiates a search by flooding the network with queries. This flooding is exactly the same as the search process detailed in section 2.2. The average neighborhood size is the average number of nodes the search query is able to reach, and is analyzed below.

2.4.1 Neighborhood Size

Recall that section 2.3 described how unstructured P2P networks follow a power law. [9] lays out a recursive generating function for the node degree probability distribution that is based on the power law. This generating function for degree x is defined as follows:

$$G_0(x) = \sum_{k=0}^{\infty} p_k x^k$$

In this equation p_k is the probability of a node having degree k, which can be calculated by using $p_k = Ck^{-\tau}$ (where C and τ are the power law constants) since the distribution follows a power law. According to [9], the average number of neighbors in one hop is $z_1 = G_0'(1) = \sum_k kp_k$, and the average number of neighbors in two hops is $z_2 = G_0''(1) = \sum_k k^2 p_k$. These equations can be manipulated algebraically to form an equation for the average number of neighbors$m$ hops away:

$$z_m = \left[\frac{z_2}{z_1}\right]^{m-1} z_1$$

Since there are up to TTL hops in a gnutella network, the equation for the average

Table 2.6: Compartmental Model Classes from [7]

Class	Description
$P_{S_{ON}}/P_{S_{OFF}}$	Number of nodes (online and offline) wishing to download a particular file.
$P_{E_{ON}}/P_{E_{OFF}}$	Number of nodes (online and offline) downloading the file.
$P_{I_{ON}}/P_{I_{OFF}}$	Number of peers (online and offline) with copy of file.
$P_{R_{ON}}/P_{R_{OFF}}$	Number of peers (online and offline) not interested in file.

search neighborhood size is:

$$z_{avg} = \sum_{i=1}^{TTL} z_i$$

2.4.2 The Compartmental Model for P2P Networks

The paper continued with a "compartmental" model for P2P networks in which there are four main classes split into two components (see table 2.6). This analytical model models the propagation of one particular file in an unstructured P2P network. This model assumes that the P2P network does not change size during the time it is being modeled. [7] then goes on to describe a set of differential equations based off the law of mass action which could be used to determine the change in number of nodes belonging to the classes described in table 2.6 over time. An example of one of the equations would be the change of $P_{E_{ON}}$over time (the number of online peers downloading a particular file):

$$\frac{dP_{E_{ON}}}{dt} = \lambda z_{AVG} P_{S_{ON}} P_{I_{ON}}/N_p + r_1 P_{E_{ON}} + r_2 P_{R_{ON}} - \lambda_{OFF} P_{S_{ON}} + \lambda_{ON} P_{E_{ON}}$$

Where:

- $1/\lambda$ is the "rate at which peers generate queries"
- N_p is the total number of peers

- r_1 is the "rate at which peers terminate active downloads"

- r_2 is "rate at which peers renew interest in downloading a file after deleted it"

- $(1/\lambda_{ON}, 1/\lambda_{OFF})$ is the "average peer on and off time durations".

2.5 P2P Network Usage Behavior

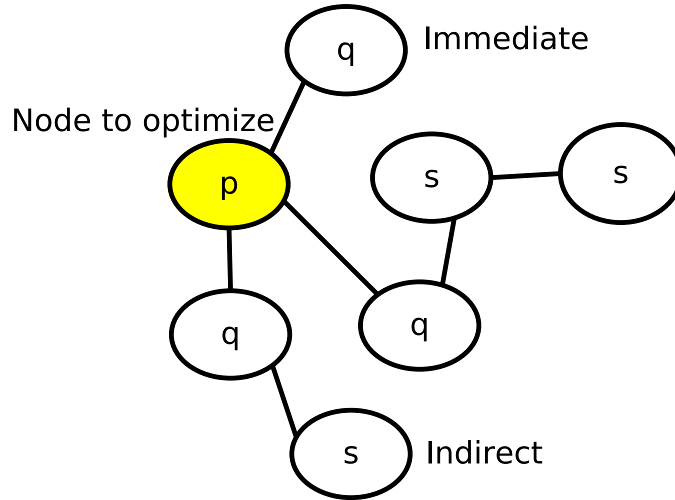
[10] proposes a heuristic method for determining peers that introduce a particular file into a P2P network. This method is based off of a hypothetical model of how files are first introduced and distributed in a P2P network. According to [10], when new files are introduced to a P2P network, they are first announced outside the network. This may happen through news outlets, message boards, chat rooms, and external communication mechanisms.

After the file is initially introduced, the hypothesis is that there is a spike in query traffic for the file that has been introduced. Once the file has been downloaded a couple of times by other nodes on the network, it will become impossible to determine which node introduced the file into the network.

In order to determine who introduces a file into a P2P network, the IP address of the node must be determined. The IP address is the only identifying information that could be used to track down the location and owner of the node. Therefore [10] suggests a method of repacking intercepted queries and sending them out to determine the IP addresses of machines hosting the file that has been queried for.

If these repackaged queries are run after a large spike in query activity, it may be possible to determine which IP address introduced the file to the network before it gets downloaded on too many machines. [10] suggest that this could be done with the help of a database system to store the queries and associated IP addresses of machines hosting the queries.

Figure 2.3: Diagram of a sample P2P Network



2.6 Finding Good Peers

According to Section 2.1, nodes obtain a list of peers to connect to from a centralized caching service. It is sometimes useful to determine which of these nodes will return better search results. In this way, the "horizon" (or number of peers to query / TTL value) can be decreased. [11] proposes a method for ranking peers based on their importance. This method is described below.

2.6.1 Metrics

In [11]'s method there are two types of peers on the network - immediate and indirect. Immediate peers q are peers directly connected to a node p (the node on which we are trying to find good peers to connect to). An indirect peer s, is a peer in which a "path from p exist" (see figure 2.3).

In order to determine good peers to connect to, each immediate peer j that is connected to a node i has four different metrics that are calculated. These metrics, shown in table 2.7, need to be calculated taking the indirect peers into account. Given a peer p with indirect peers s and with direct peers q, the average number of hops

Table 2.7: Metrics calculated for RMK algorithm.

Metric	Description
$Importance(j)$	A metric describing the relative importance of node j to i. Nodes with high importance are connected to several direct and/or indirect nodes sharing similar interests.
$percQueryHits(j)$	Query Hits sent by node j to i divided by total Query Hits from all nodes sent to i.
$avgNumHops(j)$	Average distance (or number of hops) of j from indirect peers who respond to queries from i.
$connectionTime(j)$	Amount of time j has been connected to i.

can be calculated as follows:

$$avgNumHops_p(q) = \frac{QueryHits_p(q) + \sum_{s,s:indirect}(QueryHits_{p,q}(s) * numHops_p(s))}{QueryHits_p(q) + \sum_{s,s:indirect} QueryHits_p(s)}$$

Where $QueryHits_p(q)$ is the number of query hits p receives from direct node q, $QueryHits_{p,q}(s)$ is the number of query hits received from and indirect node s being routed through direct node q, $numHops_p(s)$ is the distance or number of hops of node s from node p, and $QueryHits_p(s)$ is the number of query hits that node p receives from an indirect peer s.

The percentage of query hits for each indirect node s can be calculated using the following equation:

$$percQueryHits_p(s) = \frac{QueryHits_p(s)}{\sum_{i,i:all-contributing-peers} QueryHits_p(i)}$$

Where $QueryHits_p(s)$ is the number of query hits node p receives from s, and $QueryHits_p(i)$ is the number of queries p receives from a contributing peer of p (an indirect or direct peer that returns a query hit) denoted as i.

The most important metric is the *Importance*. The *Importance* metric can change over time as nodes enter and leave the network, so it is calculated with respect to time. The importance of a direct node q of node p with respect to time t can be

calculated as follows:

$$Importance_p(q,t) = \alpha * \frac{percQueryHits_p(q)}{avgNumHops_p(q)} + (1-\alpha) * Importance_p(q,t-1)$$

Where α is a weight factor. This calculation is an "exponentially weighted" average over time. Therefore, a higher α will give more of an indication of the importance in the current network state, whereas a lower α will give a result that represents more of the average importance over time.

2.6.2 Peer Selection Algorithm

The peer selection algorithm in [11] runs on a continual basis to evaluate the quality of connected peers to a given node p. First, a node q is selected from the immediate peers of q representing the smallest $Importance_p(q,t)$. An indirect peer s is selected with the maximum percentage of query hits [$percQueryHit_p(s)$]. If the percentage of query hits of s is higher than the percentage of query hits of q [$pecQueryHits_p(q)$], node p connects to it. If the maximum number of connections (MAX_CONNECTIONS) is exceeded for peer p, then p disconnects from q. See algorithm pseudo code in 2.1.

According to [11], the peer selection algorithm should be run more often for highly active networks compared to inactive or stagnant networks to obtain the best peer connections.

2.7 Summary

Chapter 2 reviewed research on the Gnutella protocol, one of the most common protocols used for unstructured P2P networks. It also presented research studying the basic network topology and structure of P2P networks, as well as a research model for

Algorithm 2.1 Peer Selection Algorithm

```
INPUT: peer p
OUTPUT: Algorithm will make/break connections with other nodes on
        network as needed.
PeerSelection(peer p) {
        Loop n times where n = number of immediate peers connected to p
                Find immediate peer q with least Importance;
                Find indirect peer s with maximum percQueryHits;
                if( percQueryHits(s) > percQueryHits(q) ) {
                        connect to peer s
                        if( number of connections > MAX_CONNECTIONS ) {
                                disconnect from peer q
                        }
                }
        }
}
```

file propagation and usage behavior. This research will be very useful in developing a P2P simulator.

Finally, a methodology for selecting the best peers to connect to was described, which should be very useful in a network surveillance scenario.

All the resarch covered in this chapter should be extremely useful for the purposes of this thesis. However, none of the research described a complete approach for monitoring a network for illegal acivity. Nor has any research papers been discovered on simulating searching behaviors on P2P networks. It is the aim of this thesis to contribute original research to these areas.

CHAPTER 3

METHODOLOGY

In chapter 2, we reviewed some of the research related to unstructured P2P networks. Several papers [1, 4, 5, 6, 7, 9] described the composition and structure of P2P networks, but none described a way to generate or construct a realistic network representative of real life P2P networks for simulation purposes, nor any methodology to determine which nodes are participating in exchanging contraband material.

Many of the papers described analytical models of P2P networks [9, 11], but none have actually evaluated what percentage query traffic a set of listening nodes would be able to monitor on a realistic network.

Several works described the Gnutella protocol [1, 10, 11, 7] and how to listen, collect, and interpret traffic. An optimal node placement algorithm [11] for information retrieval was discussed, but its purpose was for use in P2P client software in order to find information while conserving bandwidth and limiting the amount of traffic overhead in the network. An approach that integrates some of the ideas of the node placement algorithm into the listening node would allow us to primarily target contraband traffic. But this needs some modification.

Only one paper [10] suggested a listening technique that involved collecting and evaluating heuristics from query traffic and performing further active network probing based upon those heuristics. No papers described an actual implementation or

simulation of these techniques.

3.1 Overview of Proposed Methodology

This work presents a comprehensive approach to monitoring P2P traffic and identifying the most prolific contraband nodes. The approach can be broken down into the following steps:

1. Based on the size of the P2P network, determine the number of listening nodes needed to collect the majority of P2P network queries.
2. Randomly place this required number of listening nodes on the P2P network.
3. Start monitoring queries.
4. Have listening nodes or probe nodes run their own monitoring of intercepted queries that are determined to be contraband.
5. Store Query Hit responses from probe nodes. Listening nodes will continually make/break connections with neighboring nodes based on results from the above probing to insure an optimal targeted placement near nodes that share contraband.
6. After a given period of time, evaluate the captured Query Hits and rank nodes that are sharing contraband material.
7. Generate a report of the most prolific contraband sharing nodes based on the above rankings.

The following sections will describe in detail the implementation of each step in the methodology described above. Furthermore, this chapter will describe in detail the implementation of a simulator that allows for the evaluation of the methodology.

3.2 Estimation model for the number of needed listening nodes

In order to get an accurate sample of query traffic from the P2P network as a whole we must first determine the number of listening nodes to use. An analytical model for calculating this number would be difficult to derive, hence an optimization based estimator was developed.

The software calculator works off of a graph representative of a real p2p network. A Barabasi graph generation algorithm (as describe in section 2.3) was used to generate various random graphs to study.

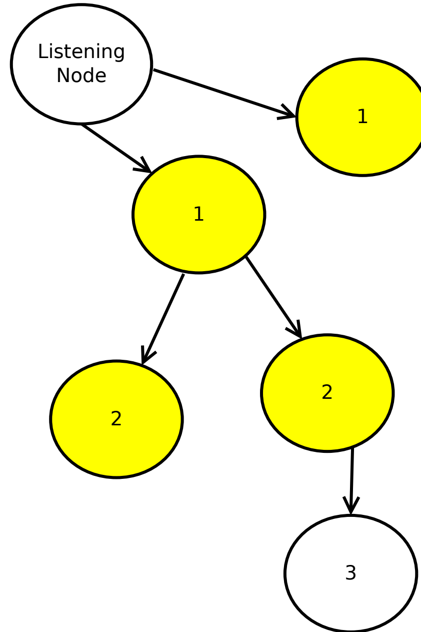
In order to determine what percentage of nodes from which listening nodes can receive traffic, Dijkstra's algorithm was utilized [15]. Dijkstra's algorithm is a single source shortest path algorithm for graph data structures. It is assumed for our generated graph that every edge has a weight of one.

When Dijkstra's algorithm is run, it tries to find the shortest path from the source node it is run on to every other node on the network. The result of this algorithm is that every node gets labeled with its distance from the source node. These distances are equivalent to the number of "hops" it would take a message to arrive at a given node from the source node.

To determine how many nodes are "visible" from a given listening node, Dijkstra's algorithm is executed using this listening node as a source. Any node that has a resulting calculated distance or label less than the TTL value is "visible" from our listening node (see figure 3.1). This means that any traffic sent from our listening node should reach a node that has a label less than the TTL value. Therefore, the listening node should also receive all queries sent from nodes with labels less than the TTL value.

For more than one listening node, Dijkstra's algorithm must be run from each

Figure 3.1: Node labels after Dijkstra's algorithm has executed. With TTL=2, reachable nodes are colored in yellow.



listening node and the list of reachable nodes must be updated after each execution of Dijkstra's algorithm. See the pseudo code in algorithm 3.1 for a complete description of this process.

In order to get a good estimate of the number of nodes needed for a given network size, we must conduct an experiment using the procedure in algorithm 3.1. Our experiment will vary three factors: network size, number of listening nodes and TTL value. Each factor has several levels shown in table 3.1. The experiment will be replicated several hundred times using different randomly generated graphs for each combination of factors and their associated levels, and the results of each replication will be averaged together.

3.3 Simulator Overview

In order to evaluate this methodology, a simulator will need to be developed that accurately models the topology of an actual P2P Gnutella network. This simulator

Algorithm 3.1 Determining the reachable nodes from more than one listening node

```
(global variables:
        G = random power law graph;
        listening_node_set {} = set of random listening nodes in G;
        TTL = 5 # can be any value , many P2P clients use 5 or 6 )

get_percent_reachable(G, listening_node_set) {
        reachable_node_set = {} # Unique set of
                                # reachable listening nodes

        for each node N in listening_node_set {
                # the following line will run Dijkstra's
                # algorithm and label every node with its
                # distance from N
                run Dijkstra's algorithm from node N;

                for every node M in G {
                        if(label on M <= TTL) {
                                add M to reachable_node_set;
                }

                x1 = number of nodes in G;
                x2 = number of nodes in reachable_node_set;
                return x2/x1;
        }
}
```

Table 3.1: Experimental design for listening node estimation experiment

Factor	Level Values	Repetitions (Graphs Generated)
Network Size	1600, 3200, 6400, 12800, 25600, 51200	100
TTL	4,5	100
Number of Listening Nodes	1-20	100

Table 3.2: Simulator Software Components

Component Name	**Description**
priorityQueue	Priority queue data structure that stores and executes events that happen on a P2P network.
graphGenerators	Used to generate the topology of a P2P network.
node	Simulates a node on a P2P network.
p2pClient	This component interfaces the node component with the file pool. This allows for searching and sharing of files.
filePool	Global pool of files shared on the P2P network. Peers pull files they share from this file pool.
gnutellaNetwork	Represents the P2P network as a whole. It contains a set of nodes connected together with a file pool.
stats	Component that stores all the statistics for the simulator.
p2pSim	Component that sets up everything needed to execute a simulation.
queryLogger	Logs queries and probes from listening nodes into a sqlite database.

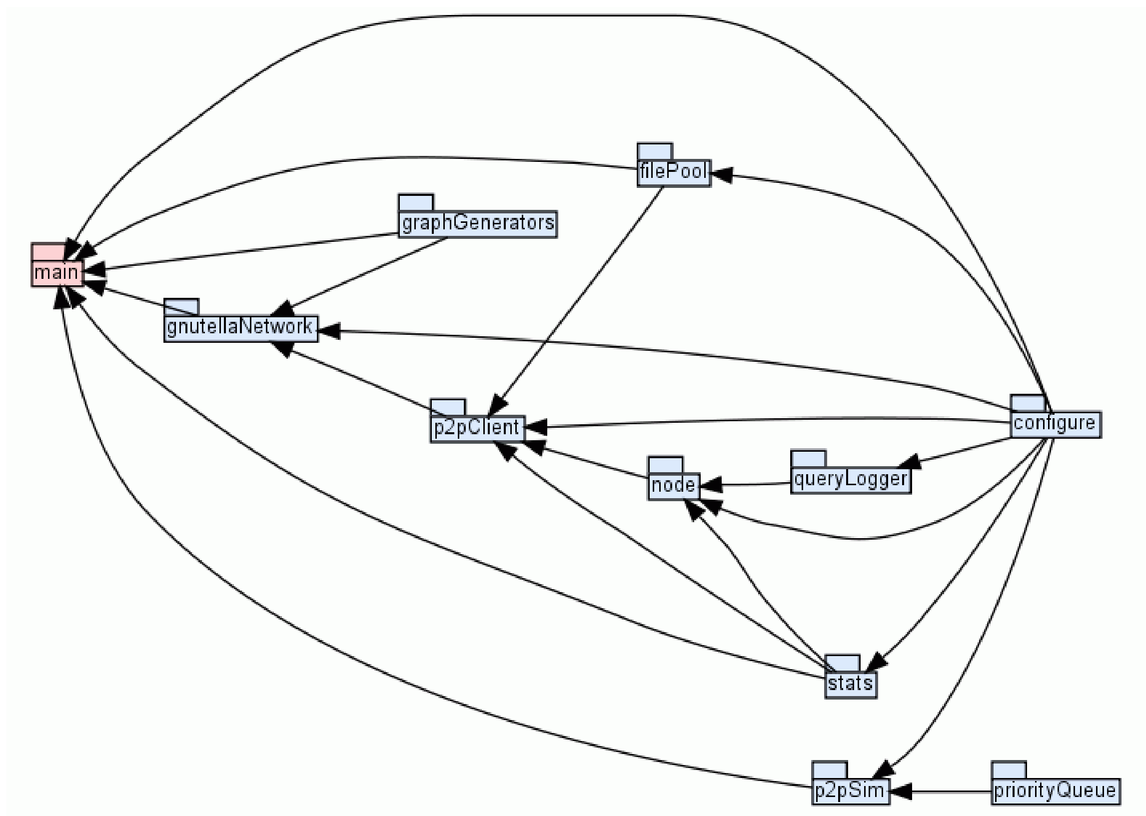
will consist of several components that are described in table 3.2.

The simulator is implemented through the main python class. This class initializes and generates a random P2P network architecture (with the help of the graphGenerators module) with its associated nodes all contained in the gnutellaNetwork class and passes it into the p2pSim class. The p2pSim class has a run method that will execute the simulator for a given number of hours specified in the configuration module (specified by NUM_HOURS parameter). The queryLogger class and stats module store all the results of the experiment. See figure 3.2 for a visual description on how these classes are connected together.

The simulator has a node and p2pClient class that implement the base functionality of a node on a p2p network. This functionality includes running queries and responding to them. The p2pClient class utilizes the filePool class in order to determine what files to share and search for.

The configuration module contains several configuration parameters that will be covered in the next few sections. See table 3.3 for a complete listing of all configuration

Figure 3.2: Import graph for p2p simulator



parameters.

3.4 P2P Network Generation and Initialization

In order to simulate a P2P network, it must first be generated. In order for the simulation to be run, the following steps must be executed:

1. Create all P2P nodes.

2. Assign each node a classification which will determine what types of files the node will be sharing and also the behavior of the node.

3. Create connections between all the nodes.

4. Generate and share files on the nodes.

The following subsections describe how the simulated network is generated and initialized.

3.4.1 P2P Network Architecture Generation

The simulator utilizes a modified version of the Barabasi-Albert algorithm [8] described in section 2.3. The modification added to the algorithm is the ability to set a maximum degree value. This modified algorithm more closely models actual P2P networks as most P2P clients have a default maximum connection limit for the number of neighboring peers that they can connect to. This modified algorithm is shown in Algorithm 3.2.

This algorithm is implemented in the graphGenerators class described in table 3.2. The gnutellaNetwork class utilizes the graphGenerators class to setup the peers and their connections. The parameters NUM_NODES, MAX_EDGES_PER_NODE,

Table 3.3: Configuration parameters

Parameter Name	Description
DATABASE_FILENAME	File name of sqlite database.
LISTENING_NODE_ALG_IN	Time interval for listening node to re-run placement algorithm (in minutes).
IMPORTANCE_ALPHA	Alpha constant of RMK/node placement.[11]
TTL	Time to live value.
LISTEN_NODES	Number of listening nodes.
NUM_HOURS	Number of hours to simulate.
GRAPH_TYPE	Graph algorithm to use when generating topology. (defaults to modified-barabasi)
NUM_NODES	Number of nodes on simulated network.
MIN_EDGES_PER_NODE	Minimum number of connections for each node on network. (used by modified barabasi).
MAX_EDGES_PER_NODE	Maximum number of connections per node. (used by modified barabasi).
SHARED_VARIANCE	Random variance from degree of node for proportion of files shared.
CONTRIB_RATIO_ CONTRABAND	Contribution ratio for contraband nodes.
CONTRIB_RATIO_NORMAL	Contribution ratio for normal files.
INIT_CONTRABAND_ FILE_POOL_SIZE	Initial contraband file pool size.
CONTRABAND_M	Percent of contraband nodes on network.
INTENTIONALLY_MIXED_M	Percent of intentionally mixed nodes on network.
UNINTENTIONALLY_MIXED_M	Percent of unintentionally mixed nodes on network.
NORMAL_M	Percent of normal nodes on network.
NETWORK_MAKEUP	Network makeup. Composed of CONTRABAND_M, INTENTIONALLY_MIXED_M, UNINTENTIONALLY_MIXED_M and NORMAL_M
CONTRABAND_DETECTION	Contraband detection rate for listening nodes.
activityProbabilities	Store probability for each class running a query per unit time.
cSearchProbabilities	Stores probability of each class running a contraband query.
INTENTIONALLY_PROP	Stores file distribution proportions of intentionally mixed class.
UNINTENTIONALLY_PROP	Stores file distribution proportions of un-intentionally mixed class.
CONTRABAND_PROP	Stores file distribution proportions of contraband class.
NORMAL_PROP	Stores file distribution proportions of normal class.

Algorithm 3.2 Modified Barabasi Algorithm

```
INPUT: minimum edges per node, max edges per node
INPUT: number of nodes
OUTPUT: RANDOM GRAPH G
function barabasi_modified(min_edges_per_node,
                           max_edges_per_node, num_nodes){
        repeated_nodes = empty list;
        targets = empty list;
        G = empty graph;
        for i=0 to min_edges_per_node {
                add node labeled i to graph G;
                add node labeled i to targets;
        }
        for i=min_edges_per_node to num_nodes{
                add new node N labeled i to graph G;
                add edges between nodes in target list and N;
                add targets to repeated_nodes list;
                for(each node in targets list){
                        add node N to repeated_nodes list;
                clear target list;
                while(length of target list <
                                min_edges_per_node){
                        x = random node in repeated_nodes;
                        if (degree of node x >
                                        max_edges_per_node){
                                remove all nodes equal to x
                                from repeated_nodes list;
                        } else {
                                add x to target list;
                        }
                }
        }
        Return G;
}
```

and MIN_EDGES_PER_NODE are present in the configuration module and can be altered to affect the graph generation.

Additionally, the number of files shared by each node is roughly proportional to the degree of the node after the graph generation algorithm has completed. There is a configuration parameter named SHARED_VARIANCE that describes the random variation from the degree used when determining the amount of files to share for each node.

3.4.2 P2P Node Classifications

There are four classes of nodes for the P2P simulator (see table 3.4 for a description of each class) . These classification help determine what proportion of the two types of files (contraband and non-contraband or normal files) will be sharing and the behavior of the node.

Nodes are assigned classifications after the network architecture has been generated using random roulette wheel selection. The proportions of nodes assigned to each of the four behavior classes is described in the following configuration parameters:

- NORMAL_M - percent of nodes classified under normal class.
- CONTRABAND_M - percent of nodes classified under contraband class.
- INTENTIONALLY_MIXED_M - percent of nodes classified under the intentionally mixed class.
- UNINTENTIONALLY_MIXED_M - percent of nodes classified under the unintentionally mixed class.

The configuration variable NETWORK_MAKEUP is a list of ranges (see table 3.5) used in a roulette wheel random selection process for each node. It is generated using the four parameters described above.

Table 3.4: Behavioral Classes for Nodes based on data from [4]

Status	Class Name	General Description	File Makeup	Behavior
Innocent	Normal	These nodes only share and search for normal, non-contraband material.	Only Normal Files.	Least active node in terms of search activity.
Innocent	Unintentionally Mixed	These nodes are mostly sharing normal files but they might occasionally on accident search for and/or share a contraband file.	These nodes mostly share normal files but some might have a few contraband files shared.	Much more active search behavior than normal nodes.
Suspicious	Intentionally Mixed	These nodes knowingly share and search for both normal and contraband files.	These types of nodes share equal amounts of contraband and normal files.	Second most active search behavior.
Suspicious	Contraband	These nodes only share and search for contraband material.	Only Contraband Files.	Most active search behavior.

Table 3.5: NETWORK_MAKEUP ranges

Class	Range
Normal	[0, NORMAL_M)
Unintentionally Mixed	[{NORMAL_M + CONTRABAND_M + UNINTENTIONALLY_MIXED_M}, 1]
Contraband	[NORMAL_M, {NORMAL_M + CONTRABAND_M})
Intentionally Mixed	[{NORMAL_M + CONTRABAND_M}, {NORMAL_M + CONTRABAND_M + UNINTENTIONALLY_MIXED_M})

3.4.3 P2P Network File Distribution

The file distribution used in the simulator is also based on a Zipf law as described in section 2.3. A single instance of the singleton class named filePool (described briefly in table 3.2) is what all nodes on the network use to pull the files they share. The filePool class has two types of files, contraband and normal files. Whenever a node needs a file name to search for, it calls the getRandNormalFile() or getRandContrabandFile() function which returns a normal or contraband file name that the calling node is not currently sharing. These functions pull files from two Zipf distributions - one for normal files and one for contraband files.

When the simulator starts, the filePool is first initialized with a given number of files specified by the INIT_NORMAL_FILE_POOL_SIZE and INIT_CONTRABAND_FILE_POOL_SIZE parameters. The genNormalFileList() and genContrabandFileList() functions are used to determine what files each node will be sharing when the simulator starts. The simulator assumes that each node is contributing a small proportion of its shared files that are not downloaded from other nodes. The first thing these functions do is add additional files to the file pool based on the CONTRIB_RATIO_NORMAL and CONTRIB_RATIO_CONTRABAND contraband parameters. The genContrabandFileList() and genNormalFileList() then utilize the getRandContrabandFile() and getRandNormalFile() functions to determine the rest of the files to be shared on each node.

3.5 P2P Network Behavioral Simulation

The behavior of nodes on the network is based upon the node classifications described in table 3.4. Since the network is event driven, every search is run after it is de-queued from the priorityQueue module listed in table 3.2. The next search event is en-queued at the end of the current search event. The time between search

Table 3.6: Example mean values for time between searches for each class

Class	Contraband	Intentionally Mixed	Unintentionally Mix	Normal
λ (in seconds)	1.0/5.0	1.0/8.0	1.0/15.0	1.0/20.0

events is pulled randomly from an exponential distribution. The simulator currently pulls from four different exponential distributions (one for each classification). An exponential distribution was used because it is assumed that query events occur cont-tinously and independently of one another, therefore following a Poisson process [16]. The following equation describes how the distribution is generated:

$$T = \frac{-lnU}{\lambda}$$

Where:

- T is randomly generated value
- U is randomly generated uniform value in range (0,1)
- λ is the mean value of the distribution (in this case, the time between searches being executed)

The "activityProbabilities" probability configuration parameter listed in table 3.3, describes the mean value (λ) to be used in the exponential distributions for each class. See table 3.6 for example values that are used in simualtor.

The types of searches (contraband or non-contraband) executed by each node is also dependent on the classification of the node. The parameter "cSearchProbabilities" configuration parameter describes the probabilities for all classes of nodes running a contraband query. When the simulator determines what type of file a node should search for, it first looks up the "cSearchProbability" for the class the node belongs to. A random number R is then generated between 0 and 1. If R is less than the

Algorithm 3.3 Pseudo code for query search activity.

```
priorityQueue: global variable for storing the events
INPUT: node to execute search on
execute_search(node) {
        generate random float R in range [0,1]
        x = cSearchProbability for class of node;

        if(R < x) {
                node.search(getRandContrabandFile());
        } else {
                node.search(getRandNormalFile());
        }

        get random value Rex from exponential distribution
        belonging to class of node;
        next_query_time = current time + Rex;

        if(next_query_time < simulator end time) {
                # add next search for node to event queue
                # if it is within simulation time interval
                priorityQueue.enqueue(next_query_time);
        }
}
```

looked up "cSearchProbability" value then the node executes a search for contraband, otherwise it searches for a normal file.

The pseudocode describing the complete search behavior for a given node is listed in Algorithm 3.3. The simulator en-queues search events for all the nodes when the simulator starts. When a search event is run for a node, the event first determines the type of file to search (using the "cSearchProbability" value). The event generates a random time interval using the exponential distribution associated with the class of the node. Finally the event ends by en-queuing another event in the event queue to execute at the current time value plus the random time value.

3.6 Listening node behavior and actions

In order to monitor a P2P network, a set of listening nodes will need to be utilized. The number of listening nodes needed is dependent on the size of the network and a method for determining the number of these nodes was described in section 3.5.

In order to get identifying information from nodes on the network, our node will need to send out its own queries. A listening node's primary purpose is to capture and store queries. Each listening node will maintain a database of keywords, hashes, and strings that are known to be associated with searches for contraband material. When a listening node comes across a query that has a search string that matches the contents of its database, it will issue the exact same query on its own. This query is known as a "probe".

The purpose of a probe query is to gather identifying information on nodes nearby that may be sharing a contraband file. With the help of these "probe" queries, the listening nodes should then be able to capture enough information that will allow other algorithms to be applied based on the results of the probes. These algorithms will be described in the following sections.

3.7 Simulator Data Capture

The simulator captures both general statistics by using the singleton *globalStatsCapture* class and by the *queryLogger* class which logs queries captured by listening nodes in an sqlite database. Both the node placement algorithm and ranking algorithm utilize these data sources. The *globalStatsCapture* class keeps track of the following statistics:

- Total number of query messages sent between nodes.
- Total number of unique queries executed.

- Total number of unique contraband and normal queries executed.
- Total number of queries captured by listening nodes.
- Total number of contraband and normal queries captured by listening nodes.
- Total number of query packets intercepted by listening nodes.

The sqlite database logs each query captured by listening nodes. This database stores queries and query responses intercepted by listening nodes as well as any query probes that are sent out by listening nodes. The following information is logged when a query or query response is intercepted by a listening node:

- Identification of listening node that intercepted the query.
- Identification of neighboring node that relayed the query to the listening node.
- Unique ID of query.
- The number of hops the query has taken to reach the listening node.
- The text in the search query.

In addition to the information above, the responding node ID is also logged in the database. When a listening node executes a probe, the following data is logged into the sqlite database:

- Identification of listening node.
- Unique ID of query.
- listen_node text, q_uuid text, filename text

3.8 Listening Node Placement Algorithm

The simulator implements the node placing algorithm (see algorithm 2.1) described in section 2.6. The primary purpose of the algorithm is to find nodes on the P2P network that might return better search results and make connections to them. Since listening nodes only send queries (probes) that are for contraband, the algorithm should have the effect of placing listening nodes so that they intercept more contraband traffic.

Each listening node utilizes Algorithm 2.1 to re-evaluate its connected neighboring nodes on a time interval specified in the simulator configuration (LISTENING_NODE_ALG_INT). The calculations needed to perform the algorithm can be done with the help of SQL queries to the sqlite database described in section 3.7. There are two main functions to the algorithm - one called *calcPercQueryHits* that determines the percentage of query hits on one node from another, and one named *Importance* which returns the importance metric for a given neighbor node.

3.8.1 percQueryHits Function

This function returns the percentage of query hits from a given node, S, sent to a listening node, P, over the total number of query hits received by node P. In order to determine this percentage, the results of two sql queries must be divided. See Algorithm 3.3 for pseudo code.

3.8.2 Importance Function

The *importance* is a function that measures the exponentially weighted average over time of the percentage of query hits that go through an indirect node, Q, divided by the average number of hops those queries take to reach node Q. With the simulator,

Algorithm 3.4 Pseudocode for percQueryHits function.

```
INPUTS: listening node p, neighbor node q
calcPercQueryHits(node p, node q) {
        // node_orig is the listening node the sent the probe
        // and node_resp is the node that responded to the
        // probe
        Q1 = "SELECT COUNT(*) FROM query_response_log
                        WHERE node_orig=p AND node_resp=q";
        Q2 = "SELECT COUNT(*) FROM query_response_log
                        WHERE node_orig=p";

        Q1_RES = execute_sql_query(Q1);
        Q2_RES = execute_sql_query(Q2);

        if(Q2_RES != 0) {
                return Q1_RES/Q2_RES;
        } else {
                return 0;
        }
}
```

the average number of hops (averNumHops) can be found by running a few SQL queries and performing some arithmetic. See Algorithm 3.5 for the pseudo code for calculating the averNumHops metric.

The Importance function has a weighting factor α (described in the parameter IMPORTANCE_ALPHA in the configuration model). Smaller values of α have more of a “smoothing” effect on the calculation - meaning that the Importance function will return a value closer to the previously calculated value. Larger values can be used to get a more “rapid response” - meaning that the previous average has less weight, in situations where there is a large amount of activity. The complete pseudo code for calculating the importance is show in Algorithm 3.6.

Algorithm 3.5 Pseudocode for averNumHops function.

```
INPUTS: listening node p, neighbor node q
avgNumHops(node p, node q) {
        # src_node is neighbor node that the listening
        # node received the query from
        Q1 = "SELECT SUM(hops) FROM query_response_log
        WHERE src_node='q' AND node_orig='p'";
        Q2 = "SELECT count(*) FROM query_response_log
        WHERE src_node='q' AND node_orig='p'";

        Q1_RES = execute_sql_query(Q1);
        Q2_RES = execute_sql_query(Q2);

        if(Q2 != 0) {
                return Q1_RES/Q2_RES;
        } else {
                return 0;
        }
}
```

Algorithm 3.6 pseudo code for Importance function.

```
global variable array prev_importance_calculations[num_nodes];
Importance (node p, node q) {
        # alpha variable is a constant that would
        # be set somewhere else outside this function
        imp = alpha * (perQueryHits(p,q)/averNumHops(p,q)) +
                (1- alpha);
        imp = imp *
                pre_importance_calculations[index of node p];
        return imp;
}
```

3.9 Node Ranking Algorithm

The node ranking algorithm works on the aggregate information collected in the sqlite database. First a query is run to find the number of contraband file that nodes are found to be sharing such files from the query response log database. Then the total number of query responses intercepted for each node that has been logged is calculated. Finally, a simple ranking algorithm is run. There are two configurable parameters in the configuration module: RANK_WEIGHT_SHARING, and RANK_WEIGHT_INTERCEPTED. RANK_WEIGHT_SHARING is a weight that is applied to the number of contraband files shared by a particular node in the log, and RANK_WEIGHT_INTERCEPTED is a weight applied to the number of intercepted queries for each node in the log. A ranking value for each node in the query response log sharing contraband, is then calculated and those results are sorted to determine the most prolific and active contraband sharers discovered while monitoring the P2P network. See Algorithm 3.7 for the pseudocode for the ranking algorithm.

3.10 Conclusion

In this section we presented a complete methodology for both simulating and monitoring a P2P network along with its information echange scenarios. An experiment for estimating the number of listening nodes needed was described, as well as an algorithm for improving the placement of listening nodes on a P2P network. This chapter concluded with an algorithm, for ranking contraband nodes, which utilizes captured traffic from the listening nodes.

Algorithm 3.7 Node Ranking Algorithm

```
INPUT: SQLITE DB Generated By Listening Node
OUTPUTS: sorted list of most prolific/active contraband nodes
CONSTANTS: RANK_WEIGHT_INTERCEPTED (or RWI in this function)
        RANK_WEIGHT_SHARED (or RWS)
rank_nodes() {
        # list that will store list of tuples containing
        # indentification of node and ranking score
        ranking_tuple_list = []

        Q1 = "
        SELECT COUNT(filename) as num_shared, node_orig
        FROM (SELECT DISTINCT filename, node_orig FROM
        query_response_log WHERE filetype='contraband')
        AS tbl GROUP BY node_orig ORDER BY 'node_orig'
        ";

        Q2 = "
        SELECT COUNT(*) as num_qr, node_orig FROM
        query_response_log WHERE filetype='contraband'
        GROUP BY node_orig
        ORDER BY 'node_orig'
        ";

        for each row (r1,r2) in (Q1,Q2) {
                # r1[node_orig] should equal r2[node_orig]
                name = r1[node_orig]
                score = r1[num_shared] * RWS +
                        r2[num_qr] * RWI;
                add tuple (score, name) to
                ranking_tuple_list;
        }
        sort ranking_tuple_list by score;
        return ranking_tuple_list;
}
```

CHAPTER 4

RESULTS

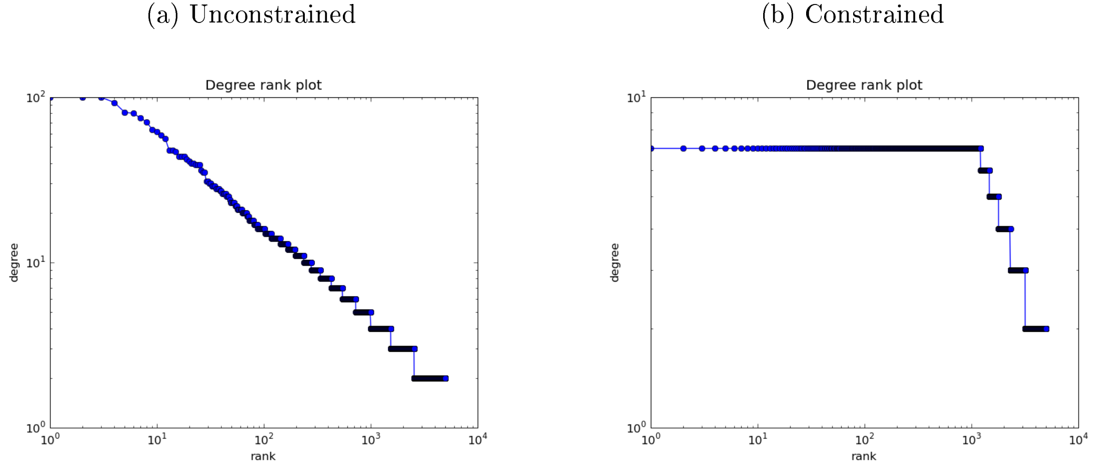
Chapter three presented the proposed methodology and implementation of a simulator for collecting traffic, improving listening node placement, and ranking contraband nodes on an unstructured P2P network. This chapter will be divided into two main sections. The first section will contain results validating the simulation software and its individual software components. The second section will cover results of the proposed methodology of network surveillance for discovering the most prolific contraband nodes.

4.1 Simulator Validation

Each component of the simulator must be tested independently. In order for the simulator to execute correctly, it must perform the following functions correctly:

- Generate a correct random graph structure for the network architecture.
- Distribute files to nodes randomly using a Zipf distribution.
- Enqueue and execute events for the simulator at time intervals specified by an exponential distribution.
- Query and query response messages must be routed correctly through the network.

Figure 4.1: Log-Log graph of degree rank plot for 5000 node random graphs generated using unconstrained and modified constrained Barabasi algorithm.



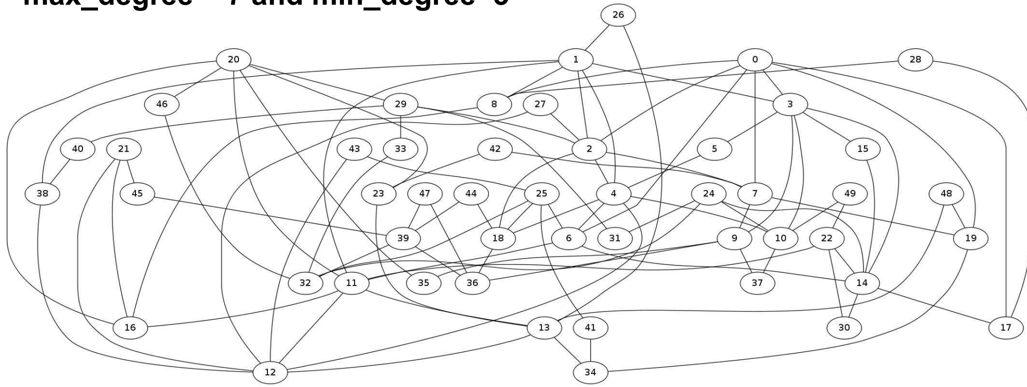
(a) Unconstrained (b) Constrained

This section contains the results of testing for each of the functions described above.

4.1.1 Graph Generation

The p2p simulator developed for this project uses a modified Barabasi graph algorithm as described in Section 3.4. This algorithm is implemented in the graphGenerators module. When setting the MAX_EDGES_PER_NODE equal to NUM_NODES, the algorithm should run as a normal Barabasi algorithm and generate a graph that follows the power law. In order to verify this, a log-log plot of the degree-rank for the graph is generated. If the graph follows a power law, a straight line should be formed on the degree rank plot. The first plot on Figure 4.1 is the log-log plot of the degree-rank for the simulator implementation of the unconstrained Barabasi algorithm on a 5000 node graph with a minimum of five edges per node. Because the points on the graph form a straight line, it is concluded that the implementation is correct and generates a graph that conforms to the power law.

Figure 4.2: Small graph generated using the unmodified Barabasi algorithm

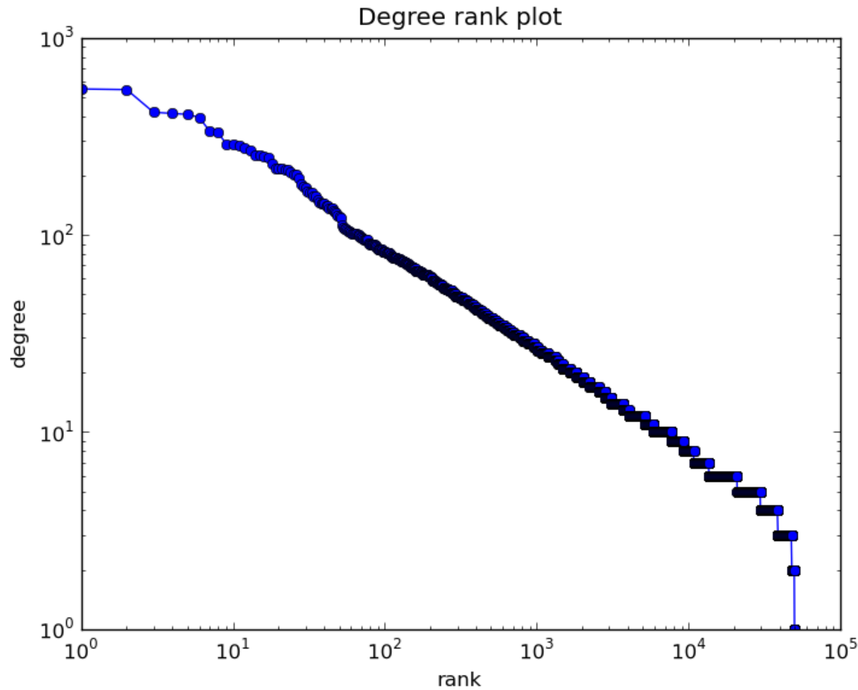


A visualization of a small graph generated by the unmodified Barabasi algorithm can be used to further confirm that the algorithm is functioning correctly. Figure 4.2 depicts a small, 50 node graph generated using the implementation of the unmodified Barabasi algorithm in this simulator. From inspection of the graph in the figure, it does seem to follow a power law because only a few of the nodes have a majority of the edge connections.

The second graph in figure 4.1 is of a graph that has been constrained to a maximum of ten edges per node and a minimum of four. The points on the graph follow a straight horizontal line until the end of the x-axis where the points start to fall into a downward linear slope. The horizontal line at the beginning of the graph accurately depicts a majority of the nodes that have been saturated with the maximum number of connections. The tail end of the graph depicts a small subset of nodes that are yet to be fully saturated with connections. This small subset of nodes still follows the power law with their degree distribution because they were added towards the end of the graph generation algorithm described in section 3.4.1 and haven't had the chance to fully saturate their connections.

Therefore, based on the analysis of the log-log graphs, it can be concluded that the simulator accurately generates the graphs it was intended to.

Figure 4.3: Log-log plot for the degree-rank of the file distribution



4.1.2 File Distribution

Like in Section 4.1.1, the same methodology of generating log-log graphs of the degree-rank can be used to verify the file distribution code that the simulator uses. The degree of every node is simply the number of files the node is sharing. Figure 4.3 is the log-log plot of degree-rank for the file distribution.

We can see from the figure that the points form a line that is approximately linear, therefore following a Zipf distribution. The slight "noise" and curvature in the line is due to the fact that the filePool class utilizes four separate Zipf distributions (since there are four separate classes of nodes, according to their roles as described in section 3.4.2) when distributing the files. Each node pulls files from its associated distribution according to its assigned classification.

The noise in this graph is more representative of log-log graphs of degree-rank

plots for file distributions in real P2P networks. This comparison is based on the graphs in the studies [4, 7, 9] described in Section 2.2.1.

It is concluded based on the log-log graph that the file distribution code in the simulator is working correctly because the graph both follows a Zipf law and is comparable to log-log graphs generated from data collected on actual P2P networks.

4.1.3 Simulation Execution

In order to evaluate the simulator, a function call graph was generated of the simulators *runSim()* function. The *runSim()* function executes a complete simulation as defined by the variables in the configuration module. The simulation performed contained no listening nodes nor was it executing any listening node placement algorithms or storing search queries. The purpose of the call graph is to trace the events/functions being called during execution of the simulator and insure it is running as expected. Figure 4.4 is a partial call graph for the runSim function.

A function called *_initFirstEvents* is called from *runSim* that adds search query events for every node on the simulated P2P network to the priority event queue so the simulator can start. From the call graph, every time the *_runQuery* function/event is de-queued and executed by the priority queue, it calls the *doQuery* function on a given node on the P2P and adds another *_runQuery* function for that node scheduled to run in the future to the priority queue. This is expected behavior as described in Section 3.5.

The *doQuery* function in the call graph shown in Figure 4.4 calls several other functions as depicted in Figure 4.5. This figure depicts calls to an “expovariate” function for generating random exponential numbers that are used to calculate the next execution time for the *_runQuery* event that is put into the priority event queue. The code behind the random exponential distributions that are responsible for determining the time between queries is part of the core python library and has already

Figure 4.4: Partial call graph for simulator run function

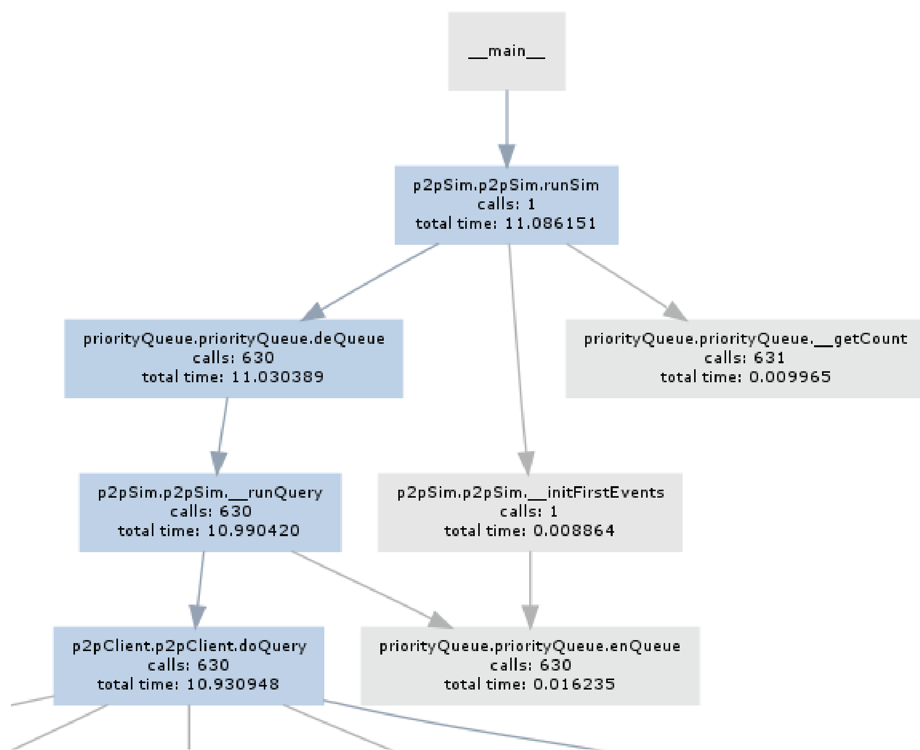
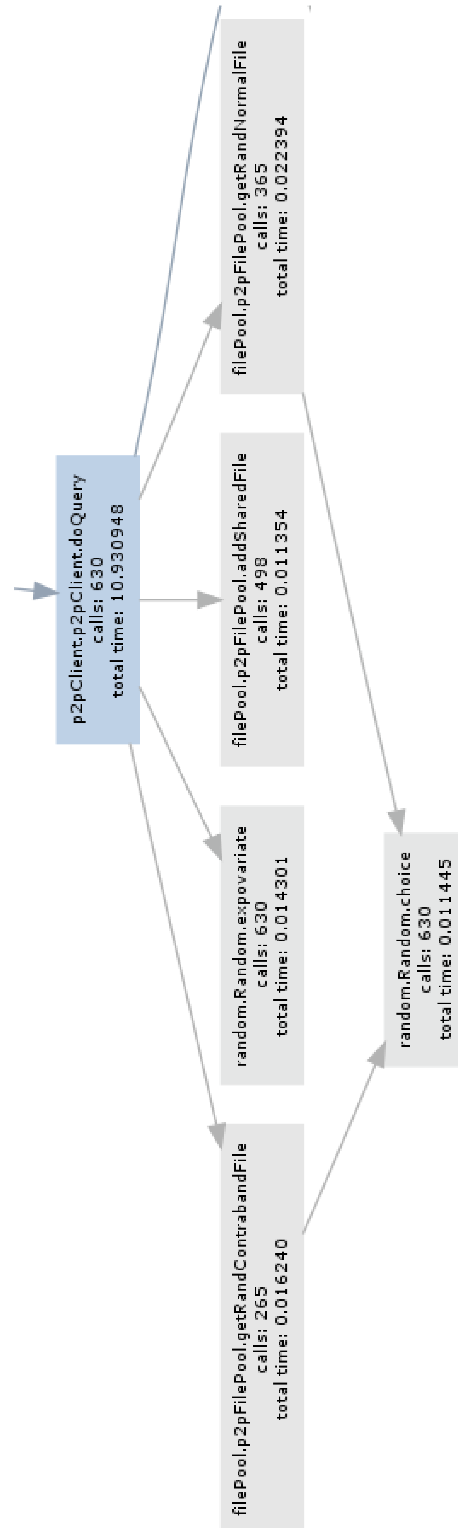


Figure 4.5: Partial call graph for doQuery function.



been evaluated and tested by the python developer community for correctness [13]. Figure 4.5 also shows calls to the *getRandNormal* and *getRandContraband* functions that are used to grab file names that the nodes can use when searching.

Figure 4.6 illustrates how each node in the simulator operates. Every time a search is started, the *addUniqueQuery* function is called to increment the unique query counter in the stats class. Several functions to generate a random universally unique id (uuid) for the query message to be sent out are also executed at the start of a search. The query function is called each time a query message is passed between node objects on the simulator, so it is the most called function. We can see from the graph that the *queryResponse* as well as the *addQueryResponse* function (in the stats class) are ran frequently in response to the queries.

The function call graphs presented in this section outline the behavior of the simulator during execution. Based on the output of the graphs, the simulator is behaving as expected and functioning properly.

4.1.4 Query Routing

The most complicated part of the simulator is the query routing code. In order for the results of the simulation to be valid, this part of the simulator must be carefully tested. In order to verify correctness, a test case was developed and debugging code was added to the p2pClient class to print out an execution trace.

A graph of five connected nodes as shown in Figure 4.7 was created. "Node A" in the network is to execute a search query for file "a". The expected results of this search is that the query should be broadcasted to all the nodes since the TTL for our test case is five. Then , "Node D" should forward a query back to "Node A" through "Node B". Below is the debug output from the simulator for the test case:

```
** connection request from node b on node a
** connection request from node c on node b
```

Figure 4.6: Call graph for p2pClient

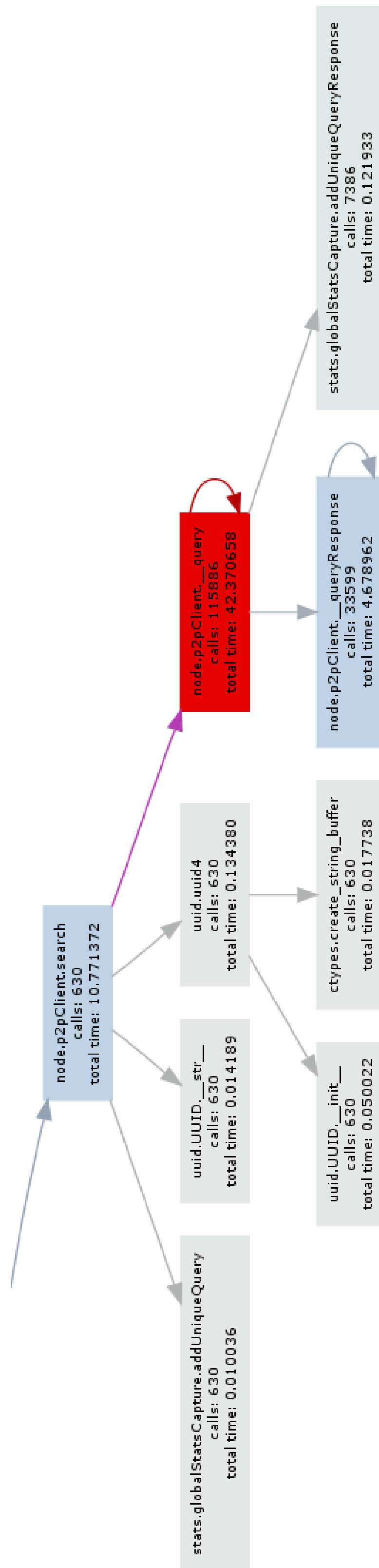
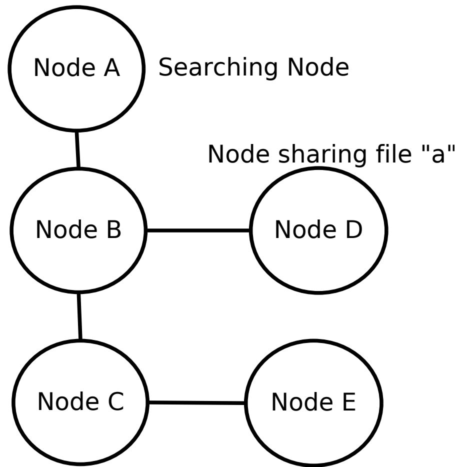


Figure 4.7: Diagram of a p2p network to test the query routing code




```
** connection request from node d on node b
** connection request from node e on node c
** node a received query for file "a" from node a with uuid:
23f3c260-2c36-488f-a4dc-74ba197983a4
** node b received query for file "a" from node a with uuid:
23f3c260-2c36-488f-a4dc-74ba197983a4
** node c received query for file "a" from node b with uuid:
23f3c260-2c36-488f-a4dc-74ba197983a4
** node e received query for file "a" from node c with uuid:
23f3c260-2c36-488f-a4dc-74ba197983a4
** node d received query file file "a" from node b with uuid:
23f3c260-2c36-488f-a4dc-74ba197983a4
** Node d has file a, sending QueryHit
** Node b received query hit...
** Node b forwarding Query Hit...
** Node a received query hit...
** Node a found file "a" it was looking for!
```

Based on the trace from the simulator, the query routing code is working as

expected - the query was broadcasted to all the nodes and the query result was sent back only through “Node b”. In order to verify that the query is not broadcasted past the scope enforced by the TTL value, the TTL was lowered to two. In this case, all the nodes except “Node E” should receive the query. Below is the output of the test:

```
** connection request from node b on node a
** connection request from node c on node b
** connection request from node d on node b
** connection request from node e on node c
** node a received query for file "a" from node a with uuid:
423cd321-4311-434e-ad13-94912aea1132
** node b received query for file "a" from node a with uuid:
423cd321-4311-434e-ad13-94912aea1132
** node c received query for file "a" from node b with uuid:
423cd321-4311-434e-ad13-94912aea1132
** node d received query file file "a" from node b with uuid:
423cd321-4311-434e-ad13-94912aea1132
** Node d has file a, sending QueryHit
** Node b received query hit...
** Node b forwarding Query Hit...
** Node a received query hit...
** Node a found file "a" it was looking for!
```

Based on the output of the simulator, the query was not sent to “Node e”, therefore the TTL limitations are being enforced as expected.

Since the simulator is routing queries and query responses correctly according to the test cases described above and the scope of the query broadcast is being limited by the TTL value correctly, the query routing code implemented in the simulator can be verified as correct.

4.2 P2P Network Surveillance Results

The proposed methodology for surveillance of a p2p network has three phases:

- Estimation of listening nodes needed to monitor the network.
- Use of a listening node placement algorithm to target contraband nodes.
- Ranking of contraband nodes based on the results of the captured traffic.

The following sections will present results of the three phases outline above.

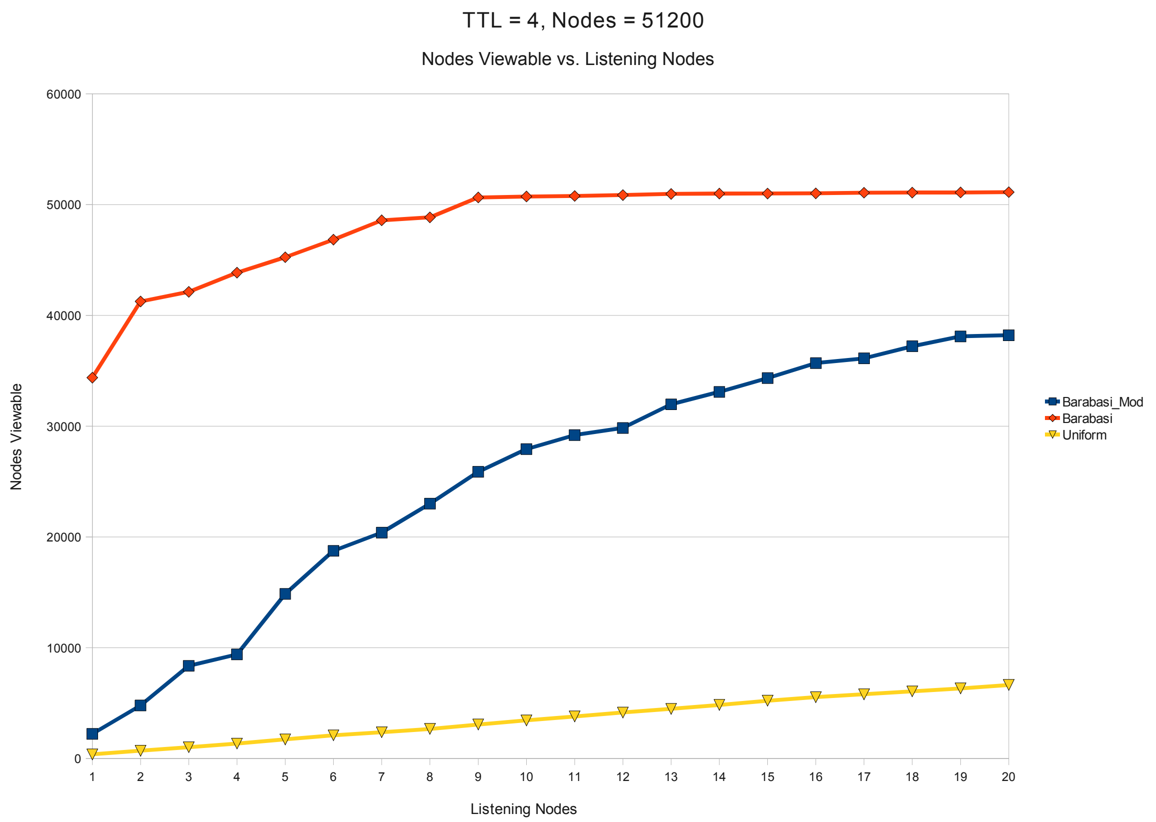
4.2.1 Estimation of the Number of Listening Nodes Needed

Section 3.2 describes the methodology for estimating the number of listening nodes needed to capture a majority of the queries on a p2p network. Figure 4.7 shows the average number of nodes viewable on a p2p network with 51,200 nodes for three different types of graphs.

The red line is for graphs generated by the unmodified version of the Barabasi-Albert algorithm and the yellow line is for graphs generated using a uniform random graph generation algorithm. Since there are no limitations on the degrees of the nodes in the graphs generated by the unmodified Barabasi algorithm, a search query is able to propagate to a large percentage of nodes on the network. This is aided by the fact that a minority of the nodes contain a majority of the connections to other nodes on the network. Therefore, p2p networks that have an architecture that closely follows an unrestricted power law require only a few listening nodes to capture a majority of the search queries executed on the network.

The blue line in figure 4.8 entitled "barabasi_mod" is for graphs that are generated by the modified Barabasi algorithm described in Section 3.4.1. This is the type of

Figure 4.8: Nodes viewable vs. listening nodes with TTL=4 (average from 100 different runs for each number of listening nodes)



graph that the simulator described in this thesis utilizes. It requires more listening nodes to capture a majority of the query traffic than the graphs that are generated by the unmodified Barabasi algorithm because there is a maximal limit on the degree of each node. For the experiment, the minimum degree was set to five for each node and the maximum degree was set to eight.

The yellow line in the graph is for networks generated using a uniform random graph generation algorithm. Since there is no clustering of edges in these types of graphs or any nodes that act as hubs with a high degree, search queries do not reach as high a percentage of nodes compared to graphs generated by the other two algorithms.

Figure 4.9 was generated using a higher TTL value of five which is a typical default value for many P2P clients. Since the TTL value is higher, queries will be propagated to a higher percentage of nodes on the network, which explains the quicker response in figure 4.9 to the number of listening nodes vs. the response in Figure 4.8.

Table 4.1 shows the percentage of nodes viewable based on the number of listening nodes used with TTL=5. This table is based on a realistically sized network of 51,200 nodes. Based on the data in the table, with more than eight listening nodes, 95% or more of the queries on the p2p network should be captured. With a higher TTL, eight listening nodes would be able to capture much more than 95% of the queries on the network.

Based on the results of this experiment, a minimum of 8 listening nodes should be used in order to monitor a p2p network of around 50,000 listening nodes (which is a typical size for a p2p network based on research presented in Section 2.2.1) in order to capture more than 95% of the queries. For larger networks, this experiment would need to be re-run with a larger graph size in order to obtain an accurate estimation.

Figure 4.9: Nodes viewable vs. Listening nodes with TTL=5 (average from 100 different runs for each number of listening nodes)

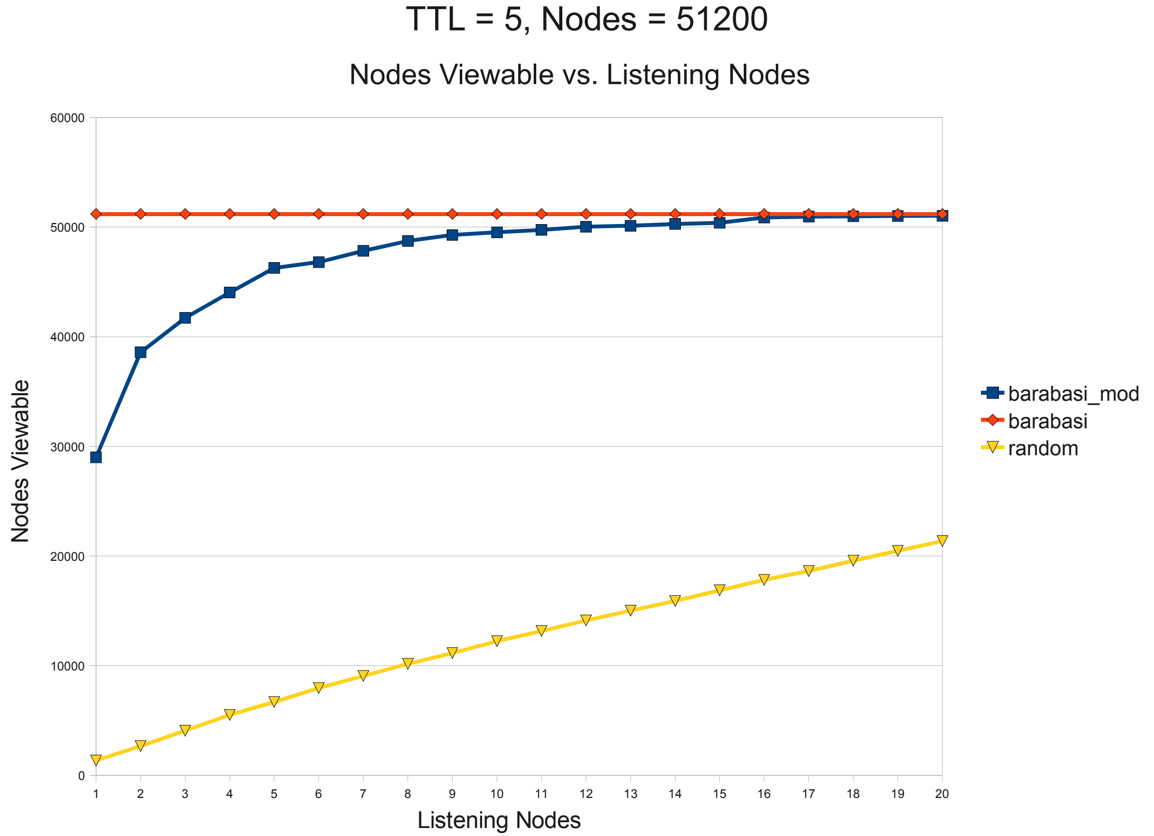


Table 4.1: Percentage of nodes viewable with TTL=5

Number of Nodes	Listening Nodes	Number Reachable	Percentage Reachable
51200	1	25000	49%
51200	2	38432	75%
51200	3	40777	80%
51200	4	44699	87%
51200	5	45587	89%
51200	6	46927	92%
51200	7	48238	94%
51200	8	48677	95%
51200	9	49120	96%
51200	10	49523	97%

Table 4.2: Summary of data from testing the listening node placement algorithm

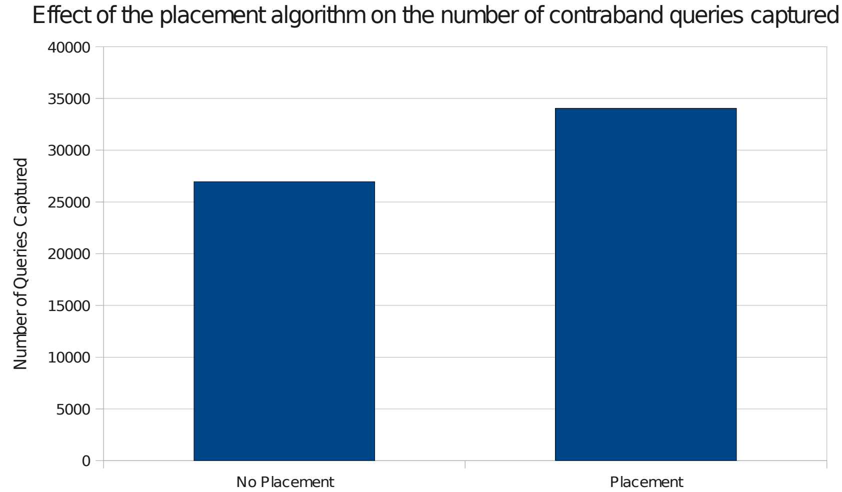
Using node placement algorithm (yes/no)	Average number of probes sent in response to contraband	Average number of query responses captured	Average number of queries captured	Average hop count of query responses
Yes	14590	5804	45089	2.75
No	11543	5556	37488	2.83

4.3 Listening Node Placement Algorithm Results

This Section evaluates the effectiveness of the node placement algorithm described in section 3.8. Several simulations were run, some using and others not using the node placement algorithm on the listening nodes. Table 4.2, summarizes the data from 20 simulations, 10 of which are used the listening node placement algorithm. The simulations were performed using a 5000-node network and ten listening nodes. The timespan to simulate was set at two hours, with a 5-minute interval period between the listening node placement algorithm being run on the listening nodes. From the Table 4.2, it looks like the node placement algorithm is helping listening nodes to obtain more query and query response messages on the P2P network. The increase in number of queries captured is most likely due to the fact that contraband nodes have the highest query activity rates. Therefore, optimizing listening nodes to receive more traffic from contraband nodes will result in many more query messages being captured.

The most important metric from Table 4.2, is the number of probe queries sent in response to contraband (remember that listening nodes send out their own queries for contraband file names they intercept). If the placement algorithm is working correctly, the listening nodes will issue many more probes in response to intercepted contraband queries. Taking into account the standard deviation of the data collected, the 90% confidence interval for this value for nodes **not** using the algorithm is [10858, 18321],

Figure 4.10: Effect of the placement algorithm on the number of contraband queries intercepted



while the 90% confidence interval for nodes **using** the algorithm is [9641, 1444]. Using Student's t test for statistical significance, it can be shown that the difference in the probe count average value between the two data sets of nodes **not** utilizing the placement algorithm and nodes **using** the placement algorithm is significant up to around 86% confidence.

The average number of query responses captured is directly linked to the number of probes sent. The more probes sent out by listening nodes, the more query response messages listening nodes should receive. Therefore, since listening nodes utilizing the placement algorithm sent more probes on average, it make sense that they would have received more query response messages as well.

The average number of hops is lower in the data collected for listening nodes utilizing the node placement algorithm. This make logical sense, because the whole goal of the algorithm is to directly connect the listening nodes with peers who will be able to provide query responses with the minimum amount of hops possible.

From examination of the number of contraband queries intercepted, the listening nodes intercepted on average 26,964 contraband queries **without** the help of the listening node placement algorithm and an average of 34,061 queries **with** the help

of the listening node placement algorithm (see Figure 4.10). This represents a 26% increase in the amount of contraband traffic that our listening nodes were able to capture as a result of the placement algorithm.

4.4 Contraband Node Ranking

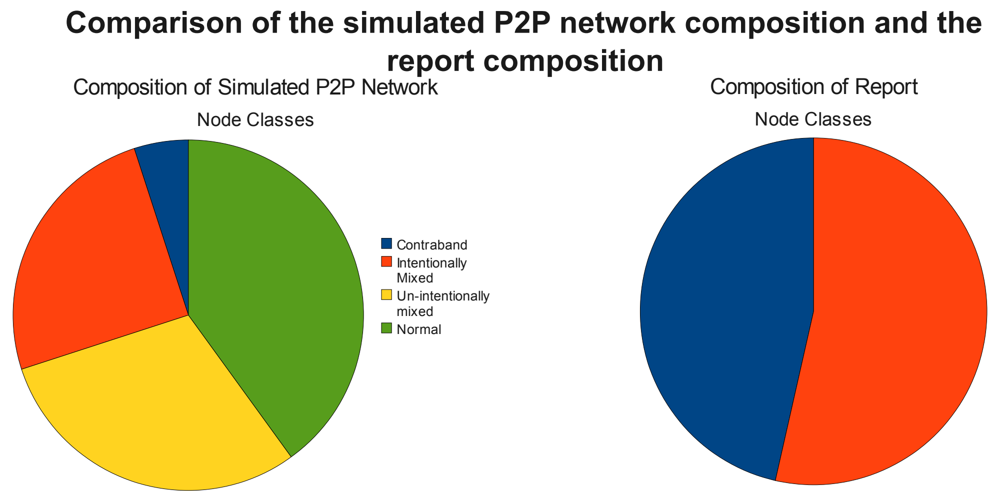
The contraband node ranking algorithm is executed at the end of the data capture period. In order to evaluate the effectiveness of the ranking algorithm, a simulation was performed with 5000 nodes for a short time span of two hours. The simulator captured a total of 5693 captured query responses from 384 different source nodes.

The ranking algorithm was used to determine the top 30 contraband nodes from the captured data set. Out of the top 30, 13 of the nodes were of the class "contraband" and the rest were of the class "intentionally-mixed". Both of these classes are "suspicious", so the ranking algorithm did a good job of only reporting nodes intentionally sharing contraband (none of the normal or unintentionally mixed nodes made it into the top thirty ranking).The composition of the report, and the composition of the network the report was generated from can be seen in figure 4.11.

Another simulation was executed using the same parameter. This time, node classifications were recorded during initialization of the network so precision and recall could be calculated. Based on the node classification records, 1528 of the nodes should have been found suspicious. Recall is the ratio of how many suspicious nodes were discovered by listening nodes, divided by how many actual listening nodes that are on the network. From the database records, the listening nodes discovered 10% of the contraband nodes on the network.

The percision is how many correct suspicious nodes were reported, divided by the number of nodes reported. Table 4.3 shows the precision for various report lengths. The algorithm was found to be 100% percise when reporting less than 15 nodes from

Figure 4.11: Report composition vs. actual network composition



a network of 5000 nodes, for a listening period of two hours.

4.5 Conclusion

This chapter presented empirical results verifying the behavior of the simulation software that has been developed. Both the graph generation and the file distribution code were analyzed, as well as the code that simulates the behavior of nodes on the network. From the examination of the call graphs, log-log graphs, and debug trace outputs from the simulator, the simulator was found to be working as expected.

The proposed number of listening nodes estimation methodology seemed to provide good results that should allow the number of listening nodes required to monitor a P2P network to be estimated fairly accurately.

Evaluation of the listening node placement algorithm showed that it was effective in increasing the number of intercepted contraband query and query response

Table 4.3: Percision for node ranking algorithm

Number Reported Suspicious	Actual Suspicious	Actual Nonuspicious	Percision
5	5	0	100%
10	10	0	100%
15	15	0	100%
20	19	1	95%
25	22	3	88%
30	24	6	80%
35	27	8	77%
40	29	11	72%

messages.

Finally, the ranking algorithm did an excellent job of reporting the most active and prolific node classes that are responsible for sharing and searching for contraband material. Furthermore, the ranking algorithm is simple and fast to execute using a database management system, which would be a realistic scenario if the algorithm was utilized in an actual system, monitoring live network activity.

CHAPTER 5

CONCLUSION

Because of the anonymity that P2P networks provide, they are an ideal place for the exchange of illegal contraband material such as child pornography. They present a difficult challenge for law enforcement, because there are no structured and proven methodologies for monitoring P2P networks.

This thesis outlined a novel approach towards monitoring P2P networks that allows for prolific contraband users to be discovered. A simulator had to be developed in order to test the approach because of the legal issues that may occur when monitoring real P2P networks.

5.1 Publications

The legal issues with monitoring a P2P network have been discussed in a separate paper [14] that has spawned out of the research conducted for this thesis.

5.2 Limitations

Because of the computational overhead in running simulations, the current results were based on a rather limited amount of data. In order to more thoroughly evaluate the methodology, many more simulations would need to be executed.

The simulator was programmed in the python scripting language that allows for rapid software development. This was done in order to complete the project within a reasonable amount of time. Using a programming language such as C or C++ that compiles down to an optimized binary would lead to a simulator which has a much faster execution time. Furthermore, python does not have the capability to use more than one processor with threads. Rewriting the simulator in another language would allow the simulator to be run much faster on machines with multiple processors or multiple processor cores.

While the simulator was developed to allow nodes on the network to break/make connections dynamically, this functionality was not implemented due to the processing overhead involved. Implementation of the simulator in a compiled language should help this limitation.

Finally, the methodology outlined in this thesis is designed for P2P networks utilizing the 0.4 version of the gnutella protocol. The 0.6 version of the gnutella protocol introduces a structured P2P network with “super nodes” and “leaf nodes” as described in Section 2.1. The methodology presented in this thesis would have to be adapted to work on later versions of the gnuTella networking protocol.

5.3 Future Directions and Goals

The obvious next step for the methodology presented in this thesis is to implement a complete system that can monitor an actual live P2P network. There are many obstacles that must be overcome to allow this to occur. First, the legal implications of monitoring a live P2P network must be addressed. It may be possible to circumvent these implications by running the software under the supervision of law enforcement.

Second, modern P2P networking protocols no longer utilize the unstructured P2P network architecture assumed in this thesis. The methodology outlined in this thesis

will have to be adapted to modern P2P protocols. Modern P2P networks have "super nodes" and "leaf nodes". The purpose of a super node is to act as a gateway and only forward queries to its listening nodes for files that they might actually have. This greatly reduces the query broadcast traffic.

There are two methods that could be used to overcome this limitation, including forcing the listening nodes to be super nodes and tricking the super node to forward all queries to a listening leaf node regardless of the actual content.

Another direction for research would be more empirical data collection. The current research papers on empirical studies of actual P2P networks is quite limited. One avenue of research that has not been explored much by any researcher (at the time of this thesis) is the behavior of individual users on a P2P network and the times between their search queries. More empirical data on these subjects would lead to a more accurate simulation.

Finally, the necessary keywords and patterns must be developed to match file names for contraband traffic. This will most likely require consultation with law enforcement.

5.4 Final Remarks

Implementing the methodology outlined in this thesis on a live P2P network should be a fairly straightforward process. Based on the results of the simulator, it appears that this approach would be very effective and provide law enforcement with a tested methodology for monitoring P2P networks for illegal contraband.

Furthermore, the simulation platform discussed in this thesis is a good outline for a more advanced implementation. By creating a simulator, ideas can be quickly tested and easily verified. This will allow for more improvement and development of ideas for surveying a P2P network for contraband activity.

APPENDIX A

Simulator Source Code

A.1 main.py

```
#!/usr/bin/env python
# p2p Discrete Event Simulator
# By: Nicholas Miles
# for p2p Project

"""p2p Simulator. By: Nicholas Miles
G{importgraph}
"""

# code optimizer
import psyco
psyco.full()

from configure import *
# for making a trace graph of the application
#import pycallgraph
import p2pSim
import gnutellaNetwork
import stats
import graphGenerators
import filePool
import datetime

debug = 0

print "Simulator starting..."

simulator = p2pSim.p2pSim()
```

```
network = gnutellaNetwork.p2pNetwork()
network.debug = debug

global_stats = stats.globalStatsCapture()

print "Setting up network..."
if(GRAPH_TYPE == BARABASI_GRAPH):
    network.generateNetwork(global_stats, NUM_NODES, GRAPH_TYPE,
                            MIN_EDGES_PER_NODE,
                            MAX_EDGES_PER_NODE)
elif(GRAPH_TYPE == UNIFORM_RANDOM_GRAPH):
    network.generateNetwork(global_stats, NUM_NODES, GRAPH_TYPE,
                            MIN_EDGES_PER_NODE)

print "Initializing listening nodes..."

network.initListenNodes()
print "Running Simulator..."
simulator.debug = 0

# start or simulation trace here
#pycallgraph.start_trace()

simulator.runSim(NUM_HOURS, network)

# end simulation trace
#pycallgraph.make_dot_graph("call_graph.png")

print "Statistics: "
print "\t Total Unique Queries: %d"% \
       global_stats.num_unique_queries
print "\t Total Unique Query Reponses: %d"% \
       global_stats.numUniqueQueryResponses
print "\t Total Unique Contraband Queries: %d" % \
       global_stats.num_unique_contraband_queries

print "\t Query to Response Ration: %f"% (float(
                                    global_stats.numUniqueQueryResponses)/
                                    global_stats.num_unique_queries)
print "\t Total Unique Normal Queries Intercepted by Listening Nodes: %d" %\
global_stats.getNumUniqueListenNomalCaptured()
print "\t Total Unique Contraband Queries Intercepted by Listening Nodes: %d" %\
global_stats.getNumUniqueListenContrabandCaptured()
```

A.2 filePool.py

```
#!/usr/bin/env python
"""Contains class that manages global file pool containing normal and contraband
file names """
import random
from configure import *
import matplotlib.pyplot as plt
import matplotlib

class p2pFilePool():
    """This class generates and stores a pool of global filenames for peers to
    pick from"""

    file_pool_normal = None
    file_pool_contraband = None

    contraband_pool_size = 0
    normal_pool_size = 0

    file_pool_normal_select = None
    file_pool_contraband_select = None

    node_file_assignments = None

    new_files = None

    __single = None

    def __init__( self ):
        if p2pFilePool.__single:
            raise p2pFilePool.__single
        p2pFilePool.__single = self

        self.file_pool_contraband = set()
        self.file_pool_normal = set()
        self.node_file_assignments = {}
        self.file_pool_normal_select = []
        self.file_pool_contraband_select = []
        self.__initList()

    def __initList(self):
        for i in range(0, INIT_CONTRABAND_FILE_POOL_SIZE):
            self.file_pool_contraband.add("c" + str(i))
            self.file_pool_contraband_select.append("c" + str(i))
```

```
    for i in range(0, INIT_NORMAL_FILE_POOL_SIZE):
        self.file_pool_normal.add("n" + str(i))
        self.file_pool_normal_select.append("n" + str(i))

def __growPoolContraband(self):
    """Internal function that grows contraband file pool size by one and
    returns name of file added"""
    t_str = "c" + str(self.contraband_pool_size)
    self.contraband_pool_size += 1
    self.file_pool_contraband.add(t_str)
    self.file_pool_contraband_select.append(t_str)
    return t_str

def __growPoolNormal(self):
    """Internal function that grows normal file pool size by one and returns
    name of file added."""
    t_str = "n" + str(self.normal_pool_size)
    self.normal_pool_size += 1
    self.file_pool_normal.add(t_str)
    self.file_pool_normal_select.append(t_str)
    return t_str

def addNode(self, node):
    """Adds an entry in file pool for given node. Each node using file pool
    should call this function first.
    @type node: node object
    @param node: Node object whose files we want to keep track of.
    """
    self.node_file_assignments[node] = set()

def degreeRankPlot(self, filename):
    """Makes a degree rank plot for file distribution.
    @type filename: string
    @param filename: file to save graph as
    """
    dict_temp = {}

    for x in self.node_file_assignments:
        for i in self.node_file_assignments[x]:
            if x not in dict_temp:
                dict_temp[x] = 1
            else:
                dict_temp[x] += 1
    l_sorted = []
    for x in dict_temp:
        l_sorted.append(dict_temp[x])
```

```
    l_sorted.sort(reverse=True)

    plt.loglog(l_sorted,'b-',marker='o')
    plt.title("Degree rank plot")
    plt.ylabel("degree")
    plt.xlabel("rank")
    plt.savefig(filename)

def genContrabandFileList(self, node, num_files):
    """Returns a list of contraband files and assigns to a node. Picked
    from a Zipf distribution.
    @type node: node object
    @param node: Node object to assign files to.
    @type num_files: number
    @param num_files: number of files to return.
    @rtype: list
    @return: List of filenames to share out.
    """
    f_name = ""
    for i in range(0, int(num_files*CONTRIB_RATIO_CONTRABAND)):
        f_name = self.__growPoolContraband()
        for j in range(0, int(num_files*CONTRIB_RATIO_CONTRABAND)):
            self.file_pool_contraband_select.append(f_name)
        self.file_pool_contraband.add(f_name)

    for i in range(0, num_files - int(num_files*CONTRIB_RATIO_CONTRABAND)):
        #if(self.isFullShare(node, CONTRABAND_TYPE)):
        #       break
        #else:
        f_name = self.getRandContrabandFile(node)

        self.node_file_assignments[node].add(f_name)
        self.file_pool_contraband_select.append(f_name)

    return self.node_file_assignments[node]

def genNormalFileList(self, node, num_files):
    """Gets a list of normal files and assigns to a node. Picked from a
    zipf distribution.
    @type node: node object
    @param node: Node object to assign files to.
    @type num_files: number
    @param num_files: number of files to return.
    @rtype: list
    @return: List of filenames to share out.
    """
```

```
        f_name = ""
        for i in range(0, int(num_files*CONTRIB_RATIO_NORMAL)):
            f_name = self.__growPoolNormal()
            for j in range(0, int(num_files*CONTRIB_RATIO_NORMAL)):
                self.file_pool_normal_select.append(f_name)
            self.file_pool_normal.add(f_name)

        for i in range(0, num_files - int(num_files*CONTRIB_RATIO_NORMAL)):
           # if(self.isFullShare(node, NORMAL_TYPE)):
            #   break
            #else:
            f_name = self.getRandNormalFile(node)

            self.node_file_assignments[node].add(f_name)
            self.file_pool_normal_select.append(f_name)

        return self.node_file_assignments[node]

    def getRandContrabandFile(self, node):
        """Returns a contraband file name that the node is currently not
        sharing. Files to search for are andomly pulrled from zipf distribution
        based on query activity.
        @type node: node object
        @param node: Node on which we want to find a new file string.
        @rtype: string
        @return: Name of file.
        """
        r_choice = random.choice(self.file_pool_contraband_select)

        if(r_choice in self.node_file_assignments[node]):
            pool = self.file_pool_contraband - \
            self.node_file_assignments[node]
            r_list = \
            [i for i in self.file_pool_contraband_select if i in pool]
            if(len(r_list) == 0):
                return self.__growPoolContraband()
            else:
                return random.choice(r_list)

        return r_choice

    def getRandNormalFile(self, node):
        """Returns a normal file name that the node is currently not sharing.
        Files to search for are randomly pulled from zipf distribution based on
        query activity.
        @type node: node object
        @param node: Node on which we want to find a new file string.
```

```
        @rtype: string
        @return: Name of file.
        """
        r_choice = random.choice(self.file_pool_normal_select)

        if(r_choice in self.node_file_assignments[node]):
            pool = self.file_pool_normal -\
            self.node_file_assignments[node]
            r_list = \
            [i for i in self.file_pool_normal_select if i in pool]
            if(len(r_list) == 0):
                return self.__growPoolNormal()
            else:
                return random.choice(r_list)

        return r_choice

    def addSharedFile(self, node, filename):
        """Adds a file that is now shared by node (e.g. file that has been
        downloaded by peer
        @type node: node object
        @param node: Node on which we want to assign a new file to.
        @type filename: string
        @param filename: Filename string.
        """
        self.node_file_assignments[node].add(filename)
        if(filename[0] == 'c'):
            self.file_pool_contraband_select.append(filename)
        else:
            self.file_pool_normal_select.append(filename)

class __testNode:
    pass

if __name__ == "__main__":

    print "Testing file pool class..."
    file_pool = p2pFilePool()

    n1 = __testNode()
    n2 = __testNode()

    file_pool.addNode(n1)
    file_pool.addNode(n2)

    print ""
    print "Initial Contraband File Pool:"
```

```
print file_pool.file_pool_contraband
print "Initial Normal File Pool:"
print file_pool.file_pool_normal

file_pool.genNormalFileList(n1,3)
file_pool.genContrabandFileList(n1,4)

file_pool.genNormalFileList(n2, 3)
file_pool.genContrabandFileList(n2, 2)

print ""
print "New Contraband File Pool:"
print file_pool.file_pool_contraband
print "New Normal File Pool:"
print file_pool.file_pool_normal

print ""
print "Contraband selection pool:"
print file_pool.file_pool_contraband_select
print "Normal selection pool:"
print file_pool.file_pool_normal_select

print ""
print "File assignments:"
print "N1:"
print file_pool.node_file_assignments[n1]
print "N2:"
print file_pool.node_file_assignments[n2]


f1 = file_pool.getRandContrabandFile(n1)
f2 = file_pool.getRandNormalFile(n1)
print " "
print "Adding files:"
print str(f1) + "," + str(n1)
print str(f2) + "," + str(n2)
print " "

file_pool.addSharedFile(n1, f1)
file_pool.addSharedFile(n1, f2)

print "New Random Selections Pools:"
print "Contraband:"
print file_pool.file_pool_contraband_select
print ""
print "Normal:"
print file_pool.file_pool_normal_select
```

A.3 node.py

```
""" Contains simple class to simulate a node on a gnutella p2p network """

import sys
import time
import uuid
import stats
import random
import queryLogger
from configure import *

class node():
    """Class that implements the basic functionality of Gnutella v.4
    protocol."""
    q_logger = queryLogger.queryLogger(DATABASE_FILENAME)
    db_name = "listen_node_capture.sqlite"
    name = ""
    connections = None
    ttl = TTL
    shared_files = None
    debug = 0
    #q_list = None
    num_queries = 0
    num_query_responses = 0
    uuid_list = None
    uuid_list_max_len = 20
    global_stats = None
    listening_node_flag = False
    prev_importance = {}


    def __init__(self, node_name, global_stats):
        self.global_stats = global_stats
        self.shared_files = {}
        #self.q_list = {}
        self.uuid_list = {}
        self.connections = []
        self.name = node_name

    def shareFile(self, filename):
        """ Share a virtual file on node with specified file name
        @type filename: filename string
        @param filename: Filename to share.
        """
```

```
        self.shared_files[filename] = 1

    def connect(self, node):
        """ Make a connection to another node object
        @type node: node object
        @param node: Node to connect to.
        """
        if self.debug == 1:
            print "connection requestion from node %s on node %s" \
            % (node.name, self.name)
        self.connections.append(node)
        node.connections.append(self)
        return 1

    def disconnect(self, node):
        self.connections.remove(node)
        node.connections.remove(self)

    def search(self, ttl, filename, probe=False):
        """ Runs a search on network for file...
        @type ttl: number
        @param ttl: TTL (time to live).
        @type filename: filename string
        @param filename: Filename to search for.
        """
        q_uuid = str(uuid.uuid4())
        if(self.listening_node_flag):
            if(probe):
                self.q_logger.addProbe(self, q_uuid, filename)
        self.global_stats.addUniqueQuery()
        if(filename[0] == 'c'):
            self.global_stats.addUniqueContrabandQuery()
        self.__query(ttl,0, filename, self, q_uuid, self)

    def __query(self, ttl, hops, filename, node, q_uuid, s_node):
        """ Internal function that sends query for filename with given
        ttl """
        if(self.listening_node_flag):
            self.q_logger.logQuery(self,s_node,q_uuid,hops,filename)
            if(filename[0] == 'c'):
                self.global_stats.addListenNodeCapturedQuery(
                            q_uuid, CONTRABAND_TYPE)
                if(hops != 0 and random.random() <
                            CONTRABAND_DETECTION):
                    self.search(hops, filename, True)
            else:
                self.global_stats.addListenNodeCapturedQuery(
```

```
                        q_uuid, NORMAL_TYPE)


        if(self.debug == 1):
            print (self.name+" recieved query from "+s_node.name+
                " with uuid: "+q_uuid)

        if(self.listening_node_flag):
            self.global_stats.addCapturedQuery(q_uuid)

        if(q_uuid not in self.uuid_list):
            if(len(self.uuid_list) > self.uuid_list_max_len):
                del self.uuid_list[self.uuid_list.keys()[0]]

            self.uuid_list[q_uuid] = s_node
            self.num_queries = self.num_queries + 1
            ttl = ttl - 1
            hops = hops + 1

            if(node != self):
                if(filename in self.shared_files):
                    if self.debug == 1:
                        print "Node %s has file %s, \
sending QueryHit" % (self.name, filename)
                    self.global_stats.\
                    addUniqueQueryResponse(q_uuid)
                    self.__queryResponse(hops, filename,
                        self, node, self, q_uuid)

            if( self.debug == 1):
                print "%s query for file: \"%s\" "%(self.name,
                                    filename)

            if ttl > 0:
                for x in self.connections:
                    if(x != s_node):
                        x.__query(ttl, hops, filename,
                        node, q_uuid, self)


    def __queryResponse(self, ttl, filename, src_node, node_orig,
                node_resp, q_uuid):
        """ Internal function that reponds to queries """
        if(self.listening_node_flag):
            self.q_logger.logQueryResponse(self,src_node,q_uuid,
                        node_orig, node_resp,
                        TTL-ttl,filename)
```

```
            self.global_stats.addCapturedQueryResponse(q_uuid)

        self.num_query_responses = self.num_query_responses + 1
        if(q_uuid in self.uuid_list):
            if(self.debug == 1 and node_resp != self):
                print "Node %s recieved query hit..." \
                % (self.name)
                print "Node %s forwarding Query Hit..."\
                % (self.name)

            ttl = ttl - 1

            if self == node_orig:
                if self.debug == 1:
                    print "Found file %s we've been \
looking for on node %s"% (filename, node_resp.name)
                    #we found the file so now we
                    #download it
                self.shared_files[filename] = 1

            if (ttl > 0):
                self.uuid_list[q_uuid].__queryResponse(ttl,\
                filename, self, node_orig, node_resp, q_uuid)

if __name__ == "__main__":
    s = set()

    s_m = stats.globalStatsCapture()
    print "Testing node class..."
    n1 = node("n1", s_m)
    n1.debug = 1
    n2 = node("n2", s_m)
    n2.debug = 1
    n3 = node("n3", s_m)
    n3.debug = 1
    n4 = node("n4", s_m)
    n4.debug = 1
    n5 = node("n5", s_m)
    n5.debug = 1

    n1.connect(n2)
    n2.connect(n3)
    n2.connect(n5)
    n3.connect(n4)

    n5.shareFile("a")
```

```
    print n1.shared_files
    import pycallgraph

    # start or simulation trace here
    pycallgraph.start_trace()

    n1.search(TTL,"a")

    # end simulation trace
    pycallgraph.make_dot_graph("node_call_graph.png")

    print n1.shared_files
```

A.4 p2pClient.py

```
"""Contains derived class from node class that can be used to emulate
behaviors of different user classes"""

from node import node
import random
import math
import datetime
import filePool
import stats
from configure import *

class p2pClient(node):
    '''p2pClient Class that emulate behavior of P2P client based on what
    classification the P2P node is determined to be
    G{callgraph}'''

    proportions = None
    classType = None
    numShared = None

    file_pool = filePool.p2pFilePool()

    def __init__(self, classType, numShared, name, global_stats):
        node.__init__(self,name,global_stats)

        self.proportions = []
        self.proportions.append(CONTRABAND_PROP)
        self.proportions.append(INTENTIONALLY_PROP)
        self.proportions.append(UNINTENTIONALLY_PROP)
        self.proportions.append(NORMAL_PROP)
```

```
        self.classType = classType
        self.numShared = numShared

        self.file_pool.addNode(self)
        self.__shareFiles()

    def doQuery(self, args):
        '''Runs a query and returns time next query should be run
        @type args: dictionary
        @param args: Dictionary containing 'exec_time' entry.
        @rtype: datetime
        @return: Datetime to execute next query.
        '''
        cur_time = args['exec_time']
        x = random.expovariate(activityProbabilities[self.classType])
        secs = datetime.timedelta(seconds=(x*60))
        dt_next_query = cur_time + secs

        x = cSearchProbabilities[self.classType];
        if(random.random() < x):
            f_name = \
            self.file_pool.getRandContrabandFile(self)
            self.search(TTL, f_name)
            if f_name in self.shared_files:
                self.file_pool.addSharedFile(self,
                                f_name)
        else: # run normal search
            f_name = self.file_pool.getRandNormalFile(self)
            if(f_name != -1):
                self.search(TTL, f_name)
                if f_name in self.shared_files:
                    self.file_pool.addSharedFile(self,
                                    f_name)

        return dt_next_query

    def __shareFiles(self):
        '''Internal function that loads nodes up with files to share'''
        norm_shared = self.__getNumNormalShared()
        contr_shared = self.__getNumContrabandShared()
        n_sh = self.file_pool.genNormalFileList(self, norm_shared)
        c_sh = self.file_pool.genContrabandFileList(self, contr_shared)
        for file in n_sh:
            self.shareFile(file)
        for file in c_sh:
            self.shareFile(file)
```

```
    def __getNumNormalShared(self):
        '''Returns number of normal files that should be shared by
        client'''
        return int(self.numShared * self.proportions[self.classType][0])

    def __getNumContrabandShared(self):
        '''Returns number of contraband files that should be shared by
        client'''
        return int(self.numShared * self.proportions[self.classType][1])


if __name__ == "__main__":
    gs = stats.globalStatsCapture()
    n1 = p2pClient(CONTRABAND_USER,10,"node_1",gs)
    n2 = p2pClient(INTENTIONALLY_MIXED_USER,10,"node_2",gs)
    n3 = p2pClient(UNINTENTIONALLY_MIXED_USER,10,"node_3",gs)
    n4 = p2pClient(NORMAL_USER,10,"node_4",gs)
    print "File Distributions:"
    print " "
    print "Contraband user shared files:"
    print n1.shared_files
    print "Intentionally mixed user shared files:"
    print n2.shared_files
    print "Unintentionally mixed user shared files:"
    print n3.shared_files
    print "Normal user shared files:"
    print n4.shared_files

    n1.debug = 1
    n2.debug = 1
    n3.debug = 1
    n4.debug = 1

    print " "
    print "Making connections: "
    n1.connect(n2)
    n2.connect(n3)
    n3.connect(n4)
    print " "

    print "Running search for known shared file:"
    n1.search(TTL, random.choice(n4.shared_files.keys()))

    print " "

    print "Running random search:"
```

```
    print " "
    print "Time Search Executed:"
    time = datetime.datetime(2001, 1, 1, 0, 0, 0)
    args = {}
    print time
    args['exec_time'] = time
    print " "

    print "Search Results:"
    t_ret = n1.doQuery(args)
    print " "

    print "Time of next search: "
    print t_ret
```

A.5 p2pSim.py

```
"""Contains main class that simulates p2p Network"""
from priorityQueue import priorityQueue
from configure import *
import datetime

class p2pSim:
    ''' Class that can simulate an unstructure gnuTella .4 network '''
    pq = priorityQueue()
    network = None
    debug = 0
    '''Debugging variable. Set equal to one to enable debugging'''
    end_time = None

    __single = None
    def __init__( self ):
        if p2pSim.__single:
            raise p2pSim.__single
        p2pSim.__single = self

    def runSim(self, num_hours, network):
        '''Runs p2pSimulator.
        @type num_hours: number
        @param num_hours: The number of hours to run simulator.
        @type network: p2pNetwork object
        @param network: P2P Network to simulate.
        '''
        self.network = network
```

```
        if(self.debug == 1):
            print "Loading first events..."

        base_time = datetime.datetime(2000, 1, 1, 0, 0, 0)
        self.__initFirstEvents(base_time)
        self.end_time = base_time + datetime.timedelta(hours=num_hours)

        if(self.debug == 1):
            print "Executing event loop."
        while(self.pq.Count > 0):
            if(self.debug == 1):
                print "Event deQueued."
            self.pq.deQueue()

        if(self.debug == 1):
            print "Simulation complete..."

    def __initFirstEvents(self, base_time):
        '''Internal function that initializes priorityQueue with some events.'''
        for p2p_client in self.network.node_list:
            args = {}
            args['p2pClient'] = p2p_client

            if(p2p_client.listening_node_flag == True):
                td = datetime.timedelta(minutes=LISTENING_NODE_ALG_INT)
                self.pq.enQueue(base_time + td, self.__updatePlacement, args)
            if(p2p_client.classType == CONTRABAND_TYPE):
                print p2p_client.name
            self.pq.enQueue(base_time,self.__runQuery,args)

    def __updatePlacement(self, args):
        client = args['p2pClient']
        #print "Running placement alg for node: " + client.name
        q_logger = client.q_logger
        l_imp_peer = client.connections[0]
        l_imp_val = q_logger.calcImportance(client, l_imp_peer)
        #print "importance calculation: " + str(l_imp_val)
        for n in client.connections[1:]:
            l_val = q_logger.calcImportance(client, n)
            #print "importance calculation: " + str(l_val)
            if(l_val < l_imp_val):
                l_imp_peer = n
                l_imp_val = l_val

        td = datetime.timedelta(minutes=LISTENING_NODE_ALG_INT)
        t = args['exec_time']
        n_args = {}
```

```
        n_args['p2pClient'] = client
        #print "Lowest: " + str(l_imp_val)

        n_list = q_logger.getIndirectPeerList(client)

        m_perc_peer = n_list[0]
        m_perc_val = q_logger.calcPercQueryHitsIndirect(client, m_perc_peer)
        #print "Percentage query hits: " + str(m_perc_val)

        for s in n_list[1:]:
            v = q_logger.calcPercQueryHitsIndirect(client, s)
            if(v > m_perc_val):
                m_perc_peer = s
                m_perc_val = v
            #print "Percentage query hits: " + str(v)

        if(m_perc_val > q_logger.calcPercQueryHits(client, l_imp_peer)):
            #print "Replacing peer " + l_imp_peer.name + " with peer " + m_perc_peer
            client.disconnect(l_imp_peer)
            m_peer = self.network.node_list[int(m_perc_peer[4:])]
            #print "check: " + m_peer.name
            client.connect(m_peer)

        if(t+td < self.end_time):
            self.pq.enQueue(t + td, self.__updatePlacement, n_args)


    def __runQuery(self, args):
        '''Internal function that handles query events. '''
        if(self.debug == 1):
            print "Running Query @ Time: " + str(args['exec_time'])
        client = args['p2pClient']
        n_time = client.doQuery(args)
        n_args = {}
        n_args['p2pClient'] = client
        if(n_time < self.end_time):
            self.pq.enQueue(n_time, self.__runQuery, n_args)
```

A.6 priorityQueue.py

```
"""Contains priority Queue class that can be used to schedule events."""
import heapq
import datetime

class priorityQueue:
```

```
    '''priorityQueue class can be used to store events and associated arguments
    that are to be executed at a given date/time'''
    heap = []

    __single = None

    def __init__( self ):
        if priorityQueue.__single:
            raise priorityQueue.__single
        priorityQueue.__single = self

    def enQueue(self, time, function, arg_dict):
        '''Add an item to the priority event queue to be executed
        @type time: datetime
    @param time: Specifies when event is to be executed.
        @type function: function
    @param function: Function to be executed with arg_dict when event is deQueued.
        @type arg_dict: dictionary
        @param arg_dict: Dictionary of arguments to be supplied to event function.
        '''
        arg_dict['exec_time'] = time
        heapq.heappush(self.heap, (time, [function,arg_dict]));

    def deQueue(self):
        '''Remove next event from queue and execute it'''
        temp = heapq.heappop(self.heap)[1]
        temp[0](temp[1])

    def __getCount(self):
        '''Returns number of items in event queue'''
        return len(self.heap)

    Count = property(__getCount)
    '''Number of items in event queue'''

def __test1(arg):
    print arg['printstr']

if __name__ == "__main__":

    x = priorityQueue()

    dict = {}
    dict1 = {}

    dict['printstr'] = "world!"
    dict1['printstr'] = "hello"
```

```
    dt = datetime.datetime(2000,1,1,1,0,0,0)
    dt1 = datetime.datetime(2000,1,1,0,1,0,0)

    x.enQueue(dt,__test1,dict)
    x.enQueue(dt1,__test1,dict1)
    x.deQueue();
    x.deQueue();
```

A.7 gnuTellaNetwork.py

```
"""Contains class that generates a P2P network and initializes nodes with
files
"""
import graphGenerators
from configure import *
import p2pClient
import random

class p2pNetwork:
    """Class that generates a P2P network."""

    node_list = None
    debug = 0

    def __init__(self):
        pass

    def generateNetwork(self, global_stats, num_nodes, graph_gen_alg, min_edges,
            max_edges=0):
        ''' Function that generates creates p2p network. Creates connections
        between nodes and starts sharing files.
    @type global_stats: globalStatsCapture object
    @param global_stats: Global statistics object.
        @type num_nodes: number
    @param num_nodes: Total number of nodes to generate.
        @type graph_gen_alg: algorithm type(BARABASI_GRAPH,UNIFORM_NORMAL_GRAPH)
    @param graph_gen_alg: which function to use when generating graph.
        @type min_edges: number
        @param min_edges: Minimum number of edges for any given node.
        @type max_edges: number
        @param max_edges: Max edges for any given node.
        '''
        if(self.debug == 1):
            print "Generating p2p Network..."
```

```
        print "\tGenerating random graph..."
    G = None
    if(graph_gen_alg == BARABASI_GRAPH):
        G = graphGenerators.barabasiRandomGraph(num_nodes, min_edges,
                                                max_edges)
    elif(graph_gen_alg == UNIFORM_RANDOM_GRAPH):
        G = graphGenerators.uniformRandomGraph(num_nodes, min_edges)

if(self.debug == 1):
    graphGenerators.displayDegreeRankPlot(G)

    self.node_list = list()
    if(self.debug == 1):
        print "\tInitializing clients with files..."
    self.__initializeClients(num_nodes, G, global_stats)
    if(self.debug == 1):
        print "\tMaking peer connections..."
    self.__makeConnections(G)
    if(self.debug == 1):
        print "\tNetwork creation complete..."

def initListenNodes(self):
"""Initializes a set number of listening nodes"""
number = LISTEN_NODES
x = 0
while ( x < number):
    l_node = random.choice(self.node_list)
    l_node.listening_node_flag = True
    x += 1


def __initializeClients(self,num_nodes, G, global_stats):
    """Initializes all the nodes for the network"""
    sorted_degree_rank = graphGenerators.sortDegreeRank(G)
j = 0
    for i in sorted_degree_rank:
        curr_node = i[1]
        std_dev = len(G[curr_node]) * SHARED_VARIANCE
        num_shared = random.normalvariate(len(G[curr_node]), std_dev)
        self.node_list.append(p2pClient.p2pClient(self.__getClass(),
                                                  num_shared,
                                                  "node" + str(j),
                        global_stats))
    j += 1

def __makeConnections(self, G):
    """Internal function that connects all the nodes together."""
```

```
        for i in range(0, len(self.node_list)):
            for n in G[str(i)]:
                self.node_list[i].connect(self.node_list[int(n)])
                del G[n][str(i)]

    def __getClass(self):
        """Returns class type. Selected using roulette wheel."""
        x = random.random()
        for i in range(0,4):
            if(x < NETWORK_MAKEUP[i]):
                return i

if __name__ == "__main__":

    network = p2pNetwork()
    network.debug = 1
    network.generateNetwork(NUM_NODES, BARABASI_GRAPH,MIN_EDGES_PER_NODE,
                            MAX_EDGES_PER_NODE)
```

A.8 graphGenerators.py

```
"""Set of various functions to generate random graphs."""
import random
import matplotlib.pyplot as plt
import matplotlib

def __displayDegreeRankPlot(degree_sequence):
    plt.loglog(degree_sequence,'b-',marker='o')
    plt.title("Degree rank plot")
    plt.ylabel("degree")
    plt.xlabel("rank")
    plt.savefig("degree_histogram.png")
    plt.show()

def displayDegreeRankPlot(G):
    dg = sortDegreeRank(G)
    dg_n = []
    for i in dg:
        dg_n.append(i[0])
    dg_n.reverse()
    __displayDegreeRankPlot(dg_n)

def uniformRandomGraph(num_nodes, edges_per_node):
    """Returns uniform random graph with given number of nodes.
    @type num_nodes: number
```

```
    @param num_nodes: Number of nodes in graph.
    @type edges_per_node: number
    @param edges_per_node: Number of edges per node
    @rtype: graph
    @return: Graph containing connections between nodes.
    """
    G = {}

    node_list = []

    for i in range(0,num_nodes):
        G[str(i)] = {}
        node_list.append(str(i))

    for i in range(0,num_nodes):
        j = len(G[str(i)])

        while j < edges_per_node:
            a = random.choice(node_list)

            while(len(G[a]) > edges_per_node or a == node_list[i]
        or node_list[i] in G[a]):
                a = random.choice(node_list)

            ix = node_list[i]
            iy = a
            G[ix][iy]=1
            G[iy][ix]=1

            j = j + 1

    return G

def barabasiRandomGraph(num_nodes, min_edges_per_node, max_edges_per_node):
    """Generates Barabasi Random Graph with min edges and max edges per node
    @type num_nodes: number
    @param num_nodes: Number of nodes in graph.
    @type min_edges_per_node: number
    @param min_edges_per_node: Minimum number of edges per node
    @type max_edges_per_node: number
    @param max_edges_per_node: Maximum number of edges for any node
    @rtype: graph
    @return: Graph containing connections between nodes.
    """
    repeated_nodes = []
    G = {}
    targets = []
```

```
    for i in range(0, num_nodes):
        G[str(i)] = {}

    for i in range(0, min_edges_per_node):
        targets.append(str(i))

    source = min_edges_per_node

    while source < num_nodes:
        for x in targets:
            G[x][str(source)] = 1
            G[str(source)][x] = 1

        repeated_nodes.extend(targets)
        repeated_nodes.extend([str(source)]*min_edges_per_node)

        targets = set()

        while len(targets) < min_edges_per_node:
            x = random.choice(repeated_nodes)
            if(len(G[x]) < max_edges_per_node):
                targets.add(x)

        source += 1

    return G

def sortDegreeRank(G):
    """Returns sorted degree rank list of nodes
    @type G: graph
    @param G: Graph to inspect.
    @rtype: list(tuples)
    @return: List of tuples containing (num_edges, node_name)
    """
    node_rank = []
    for i in range(0, len(G)):
        node_rank.append((len(G[str(i)]), str(i)))
    node_rank.sort()
    return node_rank

if __name__ == "__main__":
    G = barabasiRandomGraph(5000, 4, 100)
    print "Testing barabasi graph generation..."
    print ""
    print "Nodes in graph:"
    print len(G)
```

```
    print ""
    print "Number of connections on first node:"
    print len(G[str(0)])
    print ""
    print "Connections on node 0:"
    print G[str(0)]

    print ""
    print "Degree rank sorted node count length:"
    print len(sortDegreeRank(G))
```

A.9 queryLogger.py

```
#!/usr/bin/env python
"""  Used to log query data to an sqlite database """
import sqlite3
from configure import *
import sys
import os

class queryLogger:
    """Singleton class used to log query data to an SQL Lite DB"""

    dbName = None
    __single = None
    db = None

    def __init__(self, dbName):
        if queryLogger.__single:
            return queryLogger.__single
        queryLogger.__single = self
        self.dbName = dbName
        if(os.path.exists(dbName)):
            os.remove(dbName)
        self.db = sqlite3.connect(dbName)
        self._initDB()

    def logQuery(self, listen_node, src_node, q_uuid, hops, filename):
        '''Logs Queries Listening Nodes pick Up
        @type listen node: p2p node
        @param listen_node: listen node object
        @type src_node: p2p node
        @param src_node: src node object
        @type q_uuid: uuid string
        @param q_uuid: uuid of query
```

```
    @type hops: integer
    @param hops: number of hops query has taken
    @type filename: filename string
    @param filename: string of filename searched for
    '''
    cur = self.db.cursor()
    cur.execute("INSERT INTO query_log VALUES (?,?,?,?,?)",(
       listen_node.name, src_node.name, q_uuid, hops, filename))
    self.db.commit()
    cur.close()

def logQueryResponse(self, listen_node, src_node, q_uuid, node_orig,
                   node_resp, hops, filename):
    '''Logs Queries Listening Nodes pick Up
    @type listen node: p2p node
    @param listen_node: listen node object
    @type src_node: p2p node
    @param src_node: src node object
    @type q_uuid: uuid string
    @param q_uuid: uuid of query
    @type node_orig: node
    @param node_orig: originating node
    @type node_resp: node
    @param node_resp: responding node
    @type hops: integer
    @param hops: number of hops query has taken
    @type filename: filename string
    @param filename: string of filename searched for
    '''
    cur = self.db.cursor()
    cur.execute("INSERT INTO query_response_log VALUES (?,?,?,?,?,?,?)",(
       listen_node.name, src_node.name, q_uuid, node_orig.name,
       node_resp.name, hops, filename))
    self.db.commit()
    cur.close()

def addProbe(self, listen_node, q_uuid, filename):
    '''Add a probe query that node has sent out into network
    @type listen node: p2p node
    @param listen_node: listen node object
    @type q_uuid: uuid string
    @param q_uuid: uuid of query
    @type filename: filename string
    @param filename: string of filename searched for
    '''
    cur = self.db.cursor()
    cur.execute("""INSERT INTO query_probes VALUES (?,?,?)""",
```

```
                (listen_node.name, q_uuid, filename))
        self.db.commit()
        cur.close()

    def _initDB(self):
        cur = self.db.cursor()
        cur.execute('''create table query_log
        (listen_node text, src_node text, q_uuid text, hops int,
        filename text)''')
        cur.execute('''create table query_response_log
        (listen_node text, src_node text, q_uuid text,
        node_orig text, node_resp text, hops int,
        filename text)''')
        cur.execute('''create table query_probes
        (listen_node text, q_uuid text, filename text)''')
        self.db.commit()
        cur.close()

    def calcPercQueryHits(self, p, q):
        cur = self.db.cursor()
        cur.execute('''SELECT COUNT(*) FROM query_response_log WHERE
                node_orig=?''', [p.name])
        r = cur.fetchone()
        v1 = r[0]
        cur.execute('''SELECT COUNT(*) FROM query_response_log WHERE node_orig=?
                AND src_node=?''', [p.name, q.name])
        r = cur.fetchone()
        v2 = r[0]
        if(v1 != 0):
            res = float(v2)/float(v1)
        else:
            res = 0.0
        cur.close()
        return res

    def calcPercQueryHitsIndirect(self, p, q_name):
        cur = self.db.cursor()
        cur.execute('''SELECT COUNT(*) FROM query_response_log WHERE
                node_orig=?''', [p.name])
        r = cur.fetchone()
        v1 = r[0]
        cur.execute('''SELECT COUNT(*) FROM query_response_log WHERE node_orig=?
                AND node_resp=?''', [p.name, q_name])
        r = cur.fetchone()
        v2 = r[0]
        if(v1 != 0):
            res = float(v2)/float(v1)
```

```
        else:
            res = 0.0
        cur.close()
        return res

    def getIndirectPeerList(self, p):
        res = []
        cur = self.db.cursor()
        cur.execute('''SELECT DISTINCT node_resp FROM query_response_log WHERE
                    node_orig=?''', [p.name])

        for row in cur:
            res.append(row[0])

        cur.close()
        return res

    def calcAverNumHops(self, p, q):
        cur = self.db.cursor()
        cur.execute('''SELECT SUM(hops) FROM query_response_log WHERE
                    src_node=? AND node_orig=?''', [q.name, p.name])
        r = cur.fetchone()
        v1 = r[0]
        cur.execute('''SELECT COUNT(*) FROM query_response_log WHERE
                    src_node=? AND node_orig=?''', [q.name, p.name])
        r = cur.fetchone()
        v2 = r[0]
        if(v2 != 0):
            res = float(v1)/float(v2)
        else:
            res = 0.0
        cur.close()
        return res

    def calcImportance(self, p, q):
        if(q.name not in p.prev_importance.keys()):
            p.prev_importance[q.name] = 1.0

        if(self.calcAverNumHops(p,q) == 0):
            return 0
        else:
            imp = IMPORTANCE_ALPHA * (self.calcPercQueryHits(p, q)/
                                    self.calcAverNumHops(p,q))
            imp += (1.0-IMPORTANCE_ALPHA) * p.prev_importance[q.name]
            p.prev_importance[q.name] = imp
            return imp
```

A.10 configure.py

```
'''Contains global configuration variables.'''
import random
from staticVars import *

""" File that contains global configuration variables for simulator, and some
variable definitions..."""

######################## Global Configuration Variables ########################
################################################################################

#sqlite database filename
DATABASE_FILENAME = 'db.dat'
'''Filename of sqlite database'''

# Time interval for listening nodes to re-run
# listening node placement algorithm (in minutes)
LISTENING_NODE_ALG_INT = 5
'''Time interval for listening nodes to rerun placement algorithm. In minutes'''

# Alpha constant value for RMK algorithm...
IMPORTANCE_ALPHA = .9
'''Alpha constant for RML algorithm'''

# Time to Live (TTL)
TTL = 5
'''Time to live (TTL) value'''

# Number of listening nodes...
LISTEN_NODES = 10
'''Number of listening nodes'''

# Number of hours to simulate
NUM_HOURS = 2.0
'''Number of hours to simulate.'''

# Graph algorithm used to generate topology
GRAPH_TYPE = BARABASI_GRAPH
'''Graph algorithm to use when generating topology.'''

# Number of nodes to simulation
NUM_NODES = 5000
'''Number of nodes on simulated network.'''
```

```
# Min edges per node, used in Barabasi and Uniform Graph Generation...
MIN_EDGES_PER_NODE = 4
'''Minimum number of connections for each node on network.'''

# Max edges per node - used in Barabasi Graph Generation...
MAX_EDGES_PER_NODE = 8
'''Max number of connections for each node on network. Used with Barabasi
Algorithm.'''

# variance of number of files shared where "files shared" = "degree of node"
SHARED_VARIANCE = .25
'''Random variance from degree of node for proportion of files shared.'''

# Percentage of unique files contraband nodes is contribute...
CONTRIB_RATIO_CONTRABAND = .40
'''Contribution ratio for contraband files.'''

# Percentage of unique files normal nodes contribute...
CONTRIB_RATIO_NORMAL = .30
'''Contribution ratio for normal files.'''

# Initial contraband file pool size
INIT_CONTRABAND_FILE_POOL_SIZE = 5
'''Initial Contraband file pool size'''

# Initial normal file pool size
INIT_NORMAL_FILE_POOL_SIZE = 5
'''Initial Normal file pool size'''

########## Network Makeup - Proportions of Various Classes on Network ##########
# Percent *Contraband* nodes on network
CONTRABAND_M = .05
'''Percent of contraband nodes on network'''

# Percent *Intentionally Mixed* nodes on network
INTENTIONALLY_MIXED_M = .25
'''Percent of intentionally mixed nodes on network'''

# Percent *Unintentionally Mixed* nodes on network
UNINTENTIONALLY_MIXED_M = .30
'''Percent of unintentionally mixed nodes on network'''

# Percent *Normal* nodes on network. Do not need to specify. Should work out
# to be (1.0 - (.05 + .25 + .30)) = .40
# ** Do not edit this line **
NORMAL_M = 1 - (CONTRABAND_M + INTENTIONALLY_MIXED_M + UNINTENTIONALLY_MIXED_M)
'''Percent of normal nodes on network - does not need to be specified'''
```

```
# Sum of (CONTRABAND+INTENTIONALLY+UNINTENTIONALLY+NORMAL) should equal one

# ** Do not modifiy this line, used for routlette wheel selection... **
NETWORK_MAKEUP = [CONTRABAND_M,\
                CONTRABAND_M+INTENTIONALLY_MIXED_M,\
                CONTRABAND_M+INTENTIONALLY_MIXED_M+UNINTENTIONALLY_MIXED_M,\
                1.0]
############################## END Network Makeup ##############################

# detection probabilites for contraband queries
CONTRABAND_DETECTION = .75

# Chance of having a query activity per unit of time (one second)
# [CONTRABAND, INTENTIONALLY MIXED, UNINTENTIONALLY MIXED, NORMAL]
activityProbabilities = [1.0/5.0, 1.0/8.0, 1.0/15.0, 1.0/20.0]
'''Stores probability of each class running a query per unit time'''

# Chance of having a query run for contraband on each of the classes
# [CONTRABAND, INTENTIONALLY MIXED, UNINTENTIONALLY MIXED, NORMAL]
cSearchProbabilities = [ 100.0/100.0, 50.0/100.0, 15.0/100.0, 0.0/100.0]
'''Stores probability of each class running a contraband query'''

# File composition
#<USER_PROP> = [proportion normal shared, proportion contraband shared]
CONTRABAND_PROP = [0.0, 1.0]
'''Stores file distribution proportions of contraband class'''

x = random.uniform(0.4,0.6)
INTENTIONALLY_PROP = [x, 1.0 - x]
'''Stores file distribution proportions of intentionally mixed class'''

x = random.uniform(0.10,0.20)
UNINTENTIONALLY_PROP = [1.0 - x, x]
'''Stores file distribution proportions of unintentionally mixed class'''

NORMAL_PROP = [1.0, 0.0]
'''Stores file distribution proportions of normal class'''
```

A.11 stats.py

```
from configure import *

"""Contains classes to capture and process statistics from simulator."""
```

```
class globalStatsCapture:
    """ Singleton class that captures global statistics for simulator. """

    num_unique_queries = 0
    '''Counter for number of unique queries executed'''
    num_unique_contraband_queries = 0

    uuid_set_queries = set()
    uuid_set_query_responses = set()
    uuid_set_qr = set()

    uuid_listen_contraband = set()
    uuid_listen_normal = set()

    __single = None
    def __init__( self ):
        if globalStatsCapture.__single:
            raise globalStatsCapture.__single
        globalStatsCapture.__single = self

    def resetStats(self):
        """Resets stats."""
        self.num_unique_queries = 0
        self.uuid_set_queries.clear()
        self.uuid_set_query_responses.clear()
        self.uuid_set_qr.clear()

    def addListenNodeCapturedQuery(self, uuid, type):
        """Adds to query counter for listening nodes..."""
        if type == NORMAL_TYPE:
            self.uuid_listen_normal.add(uuid)
        elif type == CONTRABAND_TYPE:
            self.uuid_listen_contraband.add(uuid)

    def getNumUniqueListenContrabandCaptured(self):
        """Gets number of unique contraband queries captured by listening nodes.
        """
        return len(self.uuid_listen_contraband)

    def getNumUniqueListenNomalCaptured(self):
        """Gets number of unique normal queries captured by listening nodes."""
        return len(self.uuid_listen_normal)

    def addUniqueQuery(self):
        """Adds to unique query counter."""
        self.num_unique_queries += 1
```

```
    def addUniqueContrabandQuery(self):
        self.num_unique_contraband_queries += 1

    def addUniqueQueryResponse(self, uuid):
        """Adds to unique query reponse counter."""
        self.uuid_set_qr.add(uuid)

    def addCapturedQueryResponse(self, uuid):
        """Adds captured query response"""
        if(uuid not in self.uuid_set_query_responses):
            self.uuid_set_query_responses.add(uuid)

    def addCapturedQuery(self, uuid):
        """Adds captured queries"""
        if(uuid not in self.uuid_set_queries):
            self.uuid_set_queries.add(uuid)

    def getNumUniqueQueryResponses(self):
        return len(self.uuid_set_qr)

    def getNumCapturedQueryResponses(self):
        return len(self.uuid_set_query_responses)

    def getNumCapturedQueries(self):
        return len(self.uuid_set_queries)

    numUniqueQueryResponses = property(getNumUniqueQueryResponses)
    numCapturedQueries = property(getNumCapturedQueries)
    numCapturedQueryResponses = property(getNumCapturedQueryResponses)
```

A.12 staticVars.py

```
################################################################################
############## Simulation Variable Definitions - **Do Not Edit!** ##############
# Don't touch these....
CONTRABAND_USER = 0
INTENTIONALLY_MIXED_USER = 1
UNINTENTIONALLY_MIXED_USER = 2
NORMAL_USER = 3

CONTRABAND_TYPE = 0
NORMAL_TYPE = 1
UNIFORM_RANDOM_GRAPH = 0
BARABASI_GRAPH = 1
```

A.13 nodeRanking.sh

```
#!/bin/bash
sqlite3 db.dat "SELECT COUNT(filename), node_resp FROM
(SELECT DISTINCT filename, node_resp FROM query_response_log WHERE
filename LIKE '%c%') AS tbl GROUP BY node_resp ORDER BY 'node_resp';">out1.txt

sqlite3 db.dat "SELECT COUNT(*), node_resp FROM query_response_log WHERE
filename LIKE '%c%' GROUP BY node_resp ORDER BY node_resp;">out2.txt

./rankOutput.py
```

A.14 rankOutput.py

```
#!/usr/bin/python
# Node ranking algorithm

#CONSTANTS
RWS = .8
RWI = .75
NUM_RESULTS = 30
#END CONSTANTS

fh1 = open("out1.txt",'r')
fh2 = open("out2.txt",'r')
l1 = fh1.readlines()
l2 = fh2.readlines()

score_list = []
for index in range(0, len(l1)):
    l1e = l1[index].split('|')
    name = l1e[1].strip()
    ns = float(l1e[0])
    l2e = l2[index].split('|')
    nq = float(l2e[0])
    score = (ns * RWS) + (nq * RWI)
    score_list.append((score,name))

score_list.sort()

print "Top Ranking Nodes:"
for index in range(0,NUM_RESULTS):
    print score_list[index]
```

Bibliography

[1] The Gnutella Protocol Specification v0.4. Clip2. Web. 25 Feb. 2010, (www9.limewire.com/developer/gnutella_protocol_0.4.pd).

[2] Shareaza Official Website,(http://shareaza.sourceforge.net/).

[3] Limewire Official Website, (http://www.limewire.com/).

[4] J. Miller: Characterization of Data on the Gnutella Peer to Peer Network, First IEEE Consumer Communications and Networking Conference, IEEE, 2004.

[5] IEEE DISTRIBUTED SYSTEMS ONLINE 1541-4922: Peer-to-Peer: Is Deviant Behavior the Norm on P2P File- Sharing Networks?, IEEE Computer Society Vol. 7, No. 2; February 2006

[6] Ripeanu, et.al.: Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design, IEEE Internet Computing, 2002.

[7] D. Stutzbach and R. Rejaie: Capturing Accurate Snapshots of the Gnutella Network, in Global Internet Symposium, Miami, FL, 2005, pp. 127–132.

[8] Barabási, A.-L. and Albert, R.: Emergence of Scaling in Random Networks, Science, 1999, vol. 286, no. 5439, pp. 509-512.

[9] Krishna K. Ramachandran, Biplab Sikdar: Modeling malware propagation in Gnutella type peer-to-peer networks. IPDPS 2006

[10] R. Ieong, P. Lai, K. Chow, F. Law, M. Kwan and W. Tse: A Heuristic Approach for Investigating the P2P Foxy Network. University of Hong Kong, Hong Kong, China

[11] Murali Krishna Ramanathan, Vana Kalogeraki, and Jim Pruyne: Finding good peers in peer-to-peer networks. In Proc. of the Intl. Parallel and Distributed Processing Symposium, Fort Lauderdale, USA, 2002.

[12] NBC News and msnbc.com. Report: Obama Helicopter Security Breached. MSNBC, 1 Mar. 2009. Web. 20 Feb. 2010. (http://www.msnbc.msn.com/id/29447088/).

[13] Python Documentation - Generate pseudo-random numbers, (http://docs.python.org/library/random.html)

[14] “Assessing the Legal Risks in Network Forensic Probing”. Advances in Digital Forensics V Revised Selected Papers. Vol. 306. Berlin: Springer, 2009. 255-66. Print. IFIP Advances in Information and Communication Technology.

[15] Dijkstra, E. W. "A note on two problems in connexion with graphs". Numerische Mathematik 1: 269–271, 1959.

[16] Banks, Jerry, Nelson, Berry, & Nicol, David. (2009). “Discrete-event system simulation”. 2009-06.