Towards the Detection of Encrypted Peer-to-Peer File Sharing Traffic and Peer-to-Peer TV Traffic Using Deep Packet Inspection Methods

August 2009

David Alexandre Milheiro de Carvalho

Towards the Detection of Encrypted Peer-to-Peer File Sharing Traffic and Peer-to-Peer TV Traffic Using Deep Packet Inspection Methods

DISSERTATION

Submitted to University of Beira Interior in partial fulfillment of the requirements for the Degree of

MASTER OF SCIENCE

in

Information Systems and Technologies

by

David Alexandre Milheiro de Carvalho
(5-year Bachelor of Science)

Network and Multimedia Computing Group
Department of Computer Science
University of Beira Interior
Covilhã, Portugal
www.di.ubi.pt

Copyright © 2009 by David Alexandre Milheiro de Carvalho. *All right reserved. No part of this publication can be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the previous written permission of the author.*

Title image: Heraldry of the University of Beira Interior.

Towards the Detection of Encrypted Peer-to-Peer File Sharing Traffic and Peer-to-Peer TV Traffic Using Deep Packet Inspection Methods

Author: David Alexandre Milheiro de Carvalho
Student Number: 2274
E-mail: david@di.ubi.pt

Abstract

This dissertation is devoted to the study of Peer-to-Peer (P2P) network traffic identification, using Deep Packet Inspection (DPI) methods. The approach followed in this work is based on the analysis of the content of a packet payload, being paid particular attention to the cases where encryption or obfuscation is used.

The protocols and applications under study along this dissertation are organized into two main categories: P2P file sharing (BitTorrent, Gnutella and eDonkey) and P2P TV (Livestation, TVU Player and Goalbit). The history of P2P and its major milestones are briefly presented, along with their classification according to the functionalities they provide and the network protocol architectures being used by them. Studies on the evolution and current state in the detection of P2P traffic are particularly detailed, as they were the main motivation towards the detection of both encrypted P2P file sharing and P2P TV traffic.

The detection of Peer-to-Peer traffic is accomplished by using a set of open source tools, emphasizing Snort, Wireshark and Tcpdump. Snort is used for triggering the alerts concerning this kind of traffic, by using a specified set of rules. These are manually created, based on the observed P2P traffic protocol signatures and patterns, by using Wireshark and Tcpdump. For the storage and visualization of the triggered alerts in a user friendly manner, two open source tools were used, respectively, MySQL and BASE.

Finally, the main conclusions achieved in this work are briefly exposed. A section dedicated to future work contains possible directions that may be followed in order to improve this work.

Supervisor: Dr. Mário Marques Freire, Full Professor at the
Department of Computer Science, University of Beira Interior.

Preface

First of all, I would like to thank to my supervisor, Professor Mário Marques Freire, for giving me the opportunity and credit for integrating his dynamic investigation team. During the period when I was working in the MsC thesis, his support, guidance and most important, motivation, were a constant presence whether regarding technical issues or any other matter. He also provided the means so I could perform all the activities, without having limitations of any kind. This work has been partially funded by Fundação para a Ciência e a Tecnologia through TRAMANET Project contract PTDC/EIA/73072/2006.

I am also grateful to University of Beira Interior, particularly to the Department of Computer Science and to the Network and Multimedia Computing Group, for providing excellent work conditions and such a pleasant environment for researchers and students.
I would also like to express my gratitude to Pedro Ricardo de Morais Inácio and João Vasco Paulo Gomes, both PhD students under the supervision of Professor Mário Marques Freire, for expressing their support for this work.

Precious tips about the LaTeX formatting system were provided to me by Professor Simão Melo de Sousa, which allowed me to improve the writing of this thesis. He also guided me for several times, allowing me achieve the pretended results, for which I would like to express my sincere gratitude.

A special thank you to my mother Maria Deolinda and my brother Luís Miguel, for having faith in me through all these years, not only regarding my academic or professional course, but also in every single personal project in which I was involved in. Finally, I would like to thank to my wife Elisabete for her motivation, support and understanding during this first year of our marriage, in which, unfortunately, I could not be as present as I would like to. For many months, most of my free time was dedicated to this work, abdicating on many opportunities of spending time. For her, my truly gratitude and love.

David Alexandre Milheiro de Carvalho
Covilhã, Portugal

Contents

Preface iii

Contents v

List of Figures ix

List of Tables x

1 Introduction 1
- 1.1 Focus . . . 1
- 1.2 Problem Definition and Goals . . . 2
- 1.3 Thesis Organization . . . 3
- 1.4 Main Contributions . . . 4

2 Peer-to-Peer Systems 5
- 2.1 Brief Perspective of P2P History . . . 5
- 2.2 P2P Definition . . . 9
- 2.3 Classification . . . 10
  - 2.3.1 Functionalities . . . 10
  - 2.3.2 Architecture . . . 10
- 2.4 P2P Traffic Evolution . . . 20
  - 2.4.1 CAIDA . . . 20
  - 2.4.2 ipoque . . . 21
- 2.5 State of Art in P2P Detection . . . 27
  - 2.5.1 Legal Issues . . . 27
  - 2.5.2 Classification of Mechanisms for P2P Traffic Detection . . . 28
  - 2.5.3 Currently Available DPI Software . . . 30
  - 2.5.4 Currently Available DPI Hardware . . . 35

3 Experimental Testbed **39**
3.1 Introduction . . . 39
3.2 Lab of the Network and Multimedia Computing Group . . . 39
3.3 Hardware . . . 41
3.4 Network Configurations . . . 42
3.4.1 Firewalls . . . 42
3.4.2 Traffic Forwarding . . . 44
3.5 DPI and Network Software . . . 46
3.5.1 Snort . . . 46
3.5.2 Barnyard . . . 51
3.5.3 Apache . . . 53
3.5.4 MySQL . . . 53
3.5.5 BASE . . . 54
3.5.6 Wireshark . . . 56
3.6 P2P File Sharing Protocols and Applications . . . 57
3.6.1 BitTorrent Protocol . . . 58
3.6.2 eDonkey . . . 59
3.6.3 Gnutella . . . 60
3.7 P2P TV . . . 61
3.7.1 LiveStation . . . 62
3.7.2 TVU Player . . . 63
3.7.3 Octoshape . . . 64
3.7.4 Goalbit . . . 65
3.7.5 Joost . . . 65

4 P2P Traffic Detection **67**
4.1 Introduction . . . 67
4.2 BitTorrent . . . 68
4.2.1 BitTorrent Application . . . 68
4.2.2 Vuze Application . . . 71
4.3 Gnutella . . . 76
4.3.1 LimeWire . . . 76
4.3.2 GTK-Gnutella . . . 82
4.4 eDonkey . . . 86
4.4.1 eMule . . . 86
4.4.2 aMule . . . 92
4.5 P2P TV . . . 95
4.5.1 Livestation . . . 95
4.5.2 TVU Player . . . 97
4.5.3 Goalbit . . . 101

5 Conclusions and Future Work **105**
5.1 Conclusions . . . 105
5.1.1 BitTorrent . . . 106
5.1.2 Gnutella . . . 106
5.1.3 eDonkey . . . 107
5.1.4 P2P TV . . . 108
5.2 Future Work . . . 109
5.2.1 Combining DPI and Behavior Methods . . . 110
5.2.2 Mobile P2P . . . 110
5.2.3 Defeating Encryption . . . 110
5.2.4 Snort Inline . . . 111
5.2.5 Snort Performance Measurement . . . 112

Bibliography **113**

Appendix **119**

A Snort rules for eDonkey **121**
A.1 Client/Server TCP . . . 121
A.2 Client/Server UDP . . . 124
A.3 Client/Client TCP . . . 126
A.4 Extended Client/Client TCP . . . 130
A.5 Extended Client/Client UDP . . . 132
A.6 KAD Client/Client UDP . . . 133

B Snort Rules for Gnutella **139**
B.1 General Gnutella TCP . . . 139
B.2 LimeWire TCP . . . 140
B.3 LimeWire UDP . . . 141
B.4 GTK-Gnutella UDP . . . 143

C Snort Rules for BitTorrent **145**
C.1 General BitTorrent TCP . . . 145
C.2 Vuze Plain Encryption TCP . . . 146
C.3 External TCP Rules . . . 147
C.4 General BitTorrent UDP . . . 148
C.5 Vuze UDP . . . 149
C.6 External UDP Rules . . . 150

D Snort Rules for Livestation **151**

E Snort Rules for TVU Player **153**
E.1 TVU Player UDP . . . 153
E.2 TVU Player TCP . . . 154

F Snort Rules for Goalbit **155**
F.1 Goabit Protocol 155
F.2 Goalbit - BitTorrent 156

List of Figures

2.1 P2P Architecture. . . . 11
2.2 P2P Centralized Architecture. . . . 12
2.3 P2P Purely Decentralized Unstructured Architecture. . . . 13
2.4 P2P Hybrid Decentralized Unstructured Architecture Based in Supernodes. . . . 14
2.5 P2P Hybrid Decentralized Unstructured Architecture Based in Hubs. . . . 15
2.6 P2P Hybrid Decentralized Unstructured Architecture based in Trackers. . . . 16
2.7 The Chord lookup. . . . 17
2.8 The Kad Lookup Tree. . . . 18
2.9 Distance calculation using XOR metric 18
2.10 P2P Decentralized and Loosely Structured Architecture. . . . 19
2.11 Distribution of P2P Protocols in Germany, October 2006. . . . 21
2.12 Distribution of P2P protocols in Europe, October 2006. . . . 22
2.13 BitTorrent Traffic Share in Germany, October 2006. . . . 22
2.14 Relative P2P Traffic Volume, 2007. . . . 23
2.15 Protocol Proportion Changes relative to 2007. . . . 26
2.16 ipp2p function to identify Gnutella UDP traffic. . . . 32
2.17 BitTorrent and eDonkey search patterns used in l7-filter. . . . 34
2.18 Arbor eSeries e30 35
2.19 Arbor eSeries e100 35
2.20 ipoque PRX-5G 35
2.21 ipoque PRX-10G 35
2.22 Sandvine PTS 14000 36
2.23 Detection Efficiency for Encrypted Potocols 38

3.1 Experimental testbed at NMCG laboratory. . . . 40
3.2 Microsoft Windows XP® firewall configuration for allowing eMule TCP traffic. 43
3.3 Smoothwall NAT example configuration. . . . 45
3.4 Snort Architecture. . . . 47
3.5 Snort HTTP Preprocessor Configuration; /etc/snort/snort.conf file. . . . 48
3.6 MySQL Logging – Snort Configuration. . . . 48

3.7 Example of a Created Snort Rule for P2P BitTorrent Tracker Request Traffic. . 50
3.8 Snort Inline Drop Mode Example. 51
3.9 Snort Inline Replace Mode Example 51
3.10 BASE Main Interface. 55
3.11 BASE Alert Selection. 55
3.12 Wireshark filter for HTTP protocol. 57

4.1 Snort HTTP Preprocessor Configuration. 96
4.2 Proportion of Snort rules triggered for Goalbit traffic. 104

List of Tables

1.1 P2P protocols and their aplications considered in this dissertation. 3

2.1 P2P Evolution Time Line. 8
2.2 P2P Geographical Distribution. 20
2.3 Geographical Traffic Distribution, 2007 24
2.4 Geographical P2P Protocol Distribution, 2007. 24
2.5 Volume of encrypted P2P traffic, 2007. 25
2.6 Protocol Class Proportions 2008-2009. 26
2.7 Proportion of encrypted and unencrypted BitTorrent and eDonkey traffic in Germany and Southern Europe. 27
2.8 DPI versus Traffic Flow Behavior Methods 29
2.9 Unencrypted P2P Protocol Detection Efficiency. 37
2.10 Unencrypted P2P Protocol Regulation Efficiency 37

3.1 Characteristics of the Hardware Used and Their Software Installations. 41
3.2 P2P Application Ports. 42
3.3 Snort sid-msg.map File Format. 53

4.1 Characteristics of experiences and their detection results for BitTorrent traffic. . 69
4.2 Characteristics of experiences and their detection results for BitTorrent traffic. . 70
4.3 Characteristics of experiences and their detection results for BitTorrent traffic. . 71
4.4 Characteristics of experiences and their detection results for BitTorrent traffic. . 71

4.5 Characteristics of experiences and their detection results for Vuze traffic. 73
4.6 Characteristics of experiences and their detection results for Vuze traffic. 73
4.7 Characteristics of experiences and their detection results for Vuze traffic. 74
4.8 Characteristics of experiences and their detection results for Vuze traffic. 74
4.9 Comparison of the detection results obtained for BitTorrent and Vuze applications, using the same torrent file. 75
4.10 Characteristics of experiences and their detection results for Vuze traffic. 75
4.11 Characteristics of experiences and their detection results for Vuze traffic. 76
4.12 Characteristics of experiences and their detection results for LimeWire DHT traffic, with TLS encryption settings off. 78
4.13 Characteristics of experiences and their detection results for LimeWire DHT traffic, with TLS encryption settings on. 78
4.14 Characteristics of experiences and their detection results for LimeWire traffic, with TLS encryption settings on. 79
4.15 Occurrence of false positives in the tests reported in table 4.14. 80
4.16 Characteristics of experiences and their detection results for LimeWire traffic, with TLS encryption and DHT settings on. 80
4.17 Characteristics of experiences and their detection results for LimeWire traffic, with TLS encryption and DHT settings on. 81
4.18 Characteristics of experiences and their detection results for LimeWire traffic with DHT disabled and TLS encryption settings on. 81
4.19 Characteristics of experiences and their detection results for GTK-Gnutella traffic, with TLS encryption settings on. 83
4.20 Characteristics of experiences and their detection results for GTK-Gnutella traffic with TLS encryption settings on. 84
4.21 Characteristics of experiences and their detection results for GTK-Gnutella traffic with TLS encryption settings on. 86
4.22 Pattern Structure for eDonkey, Kad and Kadu. 87
4.23 Characteristics of experiences and their detection results for eMule traffic without obfuscation. 90
4.24 Characteristics of experiences and their detection results for eMule traffic with obfuscation. 91
4.25 Characteristics of experiences and their detection results for aMule traffic with obfuscation. 92
4.26 Characteristics of experiences and their detection results for aMule traffic with obfuscation. 94
4.27 Characteristics of experiences and their detection results for TVU Player traffic. 99
4.28 Characteristics of experiences and their detection results for TVU Player traffic, using Snort *threshold* option. 100
4.29 Characteristics of experiences and their detection results for Goalbit traffic. . . 103

Chapter 1

Introduction

1.1 Focus

Among all types of internet traffic, Peer-to-Peer (P2P) has the biggest share. Although it may be hard to quantify, recent studies published by the German network hardware manufacturer *ipoque* [1], suggest that 50 to 70% of the internet overall traffic in Europe is P2P. Its popularity has been growing through the years, as the Internet grew itself along with the resources available for download.

P2P, initially seen by many as illegal distribution networks, gradually evolved until many companies noticed its potential for their own product distribution. So nowadays, besides copyrighted protected content shared through P2P networks, there are also available many open source software distributions, TV shows from open channels, promotional material from movie companies, music studios, etc.

Although P2P may have some advantages comparably to other protocols, specially when downloading files which size can easily reach the Gigabytes order, its excessive utilization might lead to network congestion. System administrators can be forced to apply restrictions to its use, in order to maintain the required network quality within the organization boundaries and to the Internet. Without those restrictions, the efficiency of critical applications that might exist and require a considerable bandwidth, can be easily compromised. On the other hand, there has been an effort in the design of P2P applications in order to keep their stealth using proxies, tunnels, and even encryption.

In this work, Deep Packet Inspection (DPI) methods are used towards encrypted P2P file sharing traffic and P2PTV traffic detection. This is accomplished by using a set of open source tools, emphasizing Snort, Base, MySQL and Wireshark to respectively detect, visualize, store and manually identify P2P network traffic payload patterns.

1.2 Problem Definition and Goals

Recent versions of P2P software can use methods to achieve stealthiness. When network administrators and Internet Service Providers (ISPs) started restricting this kind of traffic, either by completely blocking it or by using *Traffic Shapping* methods (controlling network traffic, by delaying packets that meet certain criteria) to slow it down, programmers developed countermeasures like enabling tunneling and proxy support to avoid this. Therefore, disabling some TCP or UDP ports in a *firewall* may not be enough anymore, since now P2P traffic can be easily tunneled under popular protocols, like Hypertext Transfer Protocol (HTTP), which, in most organizations, simply cannot be blocked at all. In the worst scenario, along with tunneling and proxying, encryption can be used, adding more difficulty to the detection of P2P traffic. Thus, methods that can only analyze the source and destination communication ports are not enough anymore.

There are two main approaches for traffic classification [2], [3]: Based on traffic flow behavior and based on payload inspection. The difference between them, is that while in the first one, traffic classification is done by studying its behavior, through inter arrival time, packet length, etc, the second approach uses header and payload inspection of a TCP/IP packet. Both have advantages and disadvantages, and should not be considered from start as mutual exclusive alternatives. In fact, they can work as complementary solutions to the same problem, as they provide each other a tool that can confirm their results.

The main advantage of the use of DPI when compared to its alternative, is precision. Most traffic has well known signatures, that can be easily identified by DPI classifiers. On the other hand, it can be more time consuming, since the hardware or software classifier may need to read the entire payload of a packet to identify a known pattern.

The work described in this dissertation provides a solution, using DPI, to detect P2P file sharing traffic and P2PTV traffic for some of their most popular applications. These are widely used among internet users, and therefore, all combined, they represent the majority of the P2P generated traffic. The main purpose of the first well known P2P protocols was to enable file sharing between users, but there has been an increasing number of P2P networks for sharing contents like TV shows, radio broadcasts and enabling other services such as Voice Over IP(VoIP), as computer multimedia capabilities and available network bandwidth increased. This work contemplates three major P2P file sharing protocols, each one with two different applications. The reason for this is that, just like in many other situations, applications tend to use slightly different implementations for a given protocol, so it was important to test which were the common and specific payload patterns among them. As for P2PTV, four of the most well known applications were studied, but due to licensing issues, the results obtained for Octoshape could not be included in this work. The studied protocols and respective applications are listed in table 1.1.

Protocol	Application
BitTorrent	BitTorrent Vuze
eDonkey	eMule aMule
Gnutella	Limewire Gtk-Gnutella
P2PTV	Livestation TVUPlayer Goalbit

Table 1.1: P2P protocols and their aplications considered in this dissertation.

The main goal of this work, is to obtain P2P traffic payload patterns through DPI, that can successfully identify the protocols and particularly the applications, listed in table 1.1. Whenever possible, these patterns will also be able to detect P2P traffic for the given protocols, even when the applications are running with encryption or obfuscation settings on.

These patterns will be be coded as Snort [4] rules, as this is perhaps the most popular open source Network Intrusion Detection System (NDIS) that also allows protocol analysis and content searching/matching and is currently at a very mature development stage. Further details about all the software used during this work are presented in chapter 3.

1.3 Thesis Organization

The present chapter briefly introduces the motivations and goals for this work and show the organization of this document in advance. The second chapter is dedicated to the study of P2P networks. The existing architectures are shown, their usage and purpose during the last years, thus enabling to compare it with other major network protocols. There are also displayed results from studies comparing P2P protocols usage according to its network share and respective geographical region.

The Test Lab Setup is described in the third chapter. The reasons for the operating systems choice, as well as the P2P applications installed, are detailed. It is payed special attention to the tools that were used to allow P2P traffic identification and logging. along with the network setup of the lab and other important details that made possible to achieve the results.

The fourth chapter details the methods and procedures that allowed P2P traffic detection for the studied protocols, including the description and reason for the creation of the most important Snort rules for each protocol and application. Several test results are presented for each P2P protocol, as the respective rule set had increased and improved.

The final chapter is dedicated to the conclusions achieved and related future work. The focus is mainly set on the results achieved and on a short presentation of mechanisms that might overcome the difficulties caused by the use of encryption by P2P applications.

1.4 Main Contributions

This section describes, in the opinion of the Author, main contributions resulting from this research programme for the advance of the state of art about detection of peer-to-peer traffic.

The first contribution of this dissertation is the proposal of a method and its validation for identification of peer-to-peer traffic generated by most representative file sharing applications, namely for the BitTorrent and Vuze implementations of the BitTorrent protocol, for the Limewire and GTK-Gnutella implementations of the Gnutella protocol, and for the eMule and aMule applications of the eDonkey network. The research work devoted to the detection of obfuscated traffic generated by eMule has been accepted for presentation at the 1st International Conference on Advances in P2P Systems (AP2PS 2009) [5], to be held in Sliema, Malta, on October 11-16, 2009. Our research group was also invited to present advances about the detection of encrypted BitTorrent traffic in an international conference about security technology. Therefore, the corresponding research work carried out along this dissertation will also be object of publication.

The second contribution of this dissertation is the proposal of a method and its validation for identification of peer-to-peer traffic generated by most representative television applications (P2P TV), namely for Livestation, TVU Player and Goalbit applications.

Chapter 2

Peer-to-Peer Systems

2.1 Brief Perspective of P2P History

The main concept behind P2P networks is not entirely new. In fact, it exists as long as the the Internet itself. In 1967, during the Cold War, the Advanced Research Projects Agency (ARPA), of the United States Defense Department, sponsored the development of a computer network that could link existing smaller heterogeneous ones as well as future technologies [6]. The interest of the military in such a network was to possess the technology that would ensure computer network availability even in case of a nuclear strike.

> "The Original ARPANET connected UCLA, Stanford Research Institute, UC Santa Barbara and the University of Utah not in a client/server format but as equal computing peers." [7]

In the early days, the Internet was much more open then today and, basically, any two machines could reach each other. At that time there was no need for *Firewalls*, since the few people who had access to the Internet were mostly researchers, working cooperatively. Two of the first applications (still in use today) were the Telecommunications Network protocol (Telnet) and File Transfer Protocol (FTP), for remote terminal access and file transfers, respectively. Although they were client/server applications, every connected machines could have two different roles. One host that was previously the client, could act as the server not long after. From this model, two still widely used and more complex systems that include P2P components, Usenet and DNS, have emerged.

Usenet

Usenet news is a system that enables computers to copy files between them, without any central control, which is the concept of P2P networks after all. It was created in 1979 by Tom Truscott and Jim Ellis while Duke University graduate students, to allow to read and post public messages (called articles or posts, and collectively termed news) to one or more categories, known as newsgroups. This would be a replacement for the existing announcement software at the University [8]. It was based on the Unix-to-Unix-copy protocol

(UUCP), which allowed an Unix machine to connect to another, exchange files with it and then disconnect. These could be e-mails or any sort of file.

Usenet is a great example of decentralized structures on the Internet, since there is not any central authority that controls the news system, not even for adding new newsgroups. Nowadays, the Network News Transport Protocol (NNTP) is used by Usenet, to allow newsgroups discovery more efficient and exchange messages in each group.

> "Usenet's systems for decentralized control, its methods of avoiding a network flood, and other characteristics make it an excellent object lesson for designers of peer-to-peer systems." [7].

DNS

DNS stands for Domain Name System and its purpose is to enable name address to Internet Protocol (IP) conversion. [1] This is what allows one to browse the Internet using a Fully Qualified Domain Name (FQDN) like *www.di.ubi.pt*, for example, instead of its less practical IP address notation of 193.136.66.5. It was introduced in 1984 and its initial goal was to provide a better solution than what was used before. Instead of using a regularly updated single local stored *hosts.txt* text file, to hold all that information to match a FQDN to its corresponding IP address, DNS uses both characteristics of a hierarchical model and a P2P network. The features that provided its scalability, which allowed it to grow exponentially through the years, have been the starting point for much more recent P2P protocols. One of those features is that it allows hosts to act both as clients and servers, just like in nowadays P2P networks, due to the design of the protocol itself. DNS has to replicate and propagate requests across the Internet as new sites are added and changed frequently.

Another DNS feature is its hierarchical model, that allows one server to follow the chain of authority for a given domain, although any server can generally query one another. This also enables response improvement, since the load is distributed locally across the Internet.

Caching is another characteristic of DNS, which enables DNS replies to be stored locally in a host for a given time, improving the response time of these systems. When a host searches for the corresponding IP address of a given name, it performs a query to the nearest name server. If that server does not have information regarding that DNS record, it then recursively forwards it to the domain name authority of the intended resource, which can reach the Internet root name servers. "As the answer propagates back down to the requester, the result is cached along the way to the name servers so the next fetch can be more efficient." [7]

The 1990's

In the nineties, big companies like Boeing, Amerada Hess and Intel, adopted P2P technology to increase their computing power, without the need of acquiring new mainframes. This was achieved by using their already existing machines, which, most of times, were not using by far all their computing and storage capacity.

[1] The reverse process of obtaining a name address through an IP address is called reverse DNS Lookup.

> "Intel has been using the technology since 1990 to slash the cost of its chip-design process. The company uses a homegrown system called *NetBatch* to link 10,000 computers, giving its engineers access to globally distributed processing power. Within two years of implementing this, they eliminated new mainframe purchases and mothballed several that they already had." [9]

Pat Gelsinger was Intel's chief technology officer at the time (nowadays senior vice president and co-general manager of Intel Corporation's Digital Enterprise Group) and said they "had eliminated new mainframe purchases within two years of adopting *NetBatch* and have saved an estimated $500 million over the decade that it had been in use."

Amerada Hess, a multinational oil and energy company, also used P2P networking with its *Beowulf Project* still in use today [10]. It initially connected 200 Dell desktop PCs running Linux to handle complex seismic data interpretation, and replaced a pair of IBM supercomputers.

> "We're running seven times the throughput at a fraction of the cost" [9].

Napster

Perhaps one of the most well known P2P applications of all time was *Napster*. It was created by Shawn Fanning while a freshman at Northeastern University, in May 1999 and it spread quite fast among college and universities students. Napster enabled its users to download music files directly from other computers (peers), but it was not a pure P2P network. A simple explanation of its operation mode can be presented like this: A local installed program in the client would do the music search and then send the results to a central server. When a user intended some file, it would send a query to the indexing server, whom returned the file locations to the client. Then, the communications were done directly between the peers. This dependency on central servers at the initial stage of the communications allowed this network to be shutdown in July 2001, after being sued by the Recording Industry Association of America (RIAA) in December 1999 and the rock band Metallica in April 2000. Not long after, non-dependent central server networks (some sill active today) emerged, allowing them to operate even in case of legal actions are taken to bring them down.

Nowadays P2P is widely used. Besides its evident advantages for file sharing applications, later described in 2.2, it started to be used for many others such as instant messaging, media streamming, etc, as shown in table 2.1.

Year	Application	Type
1999	Napster	File Sharing
	DirectConnect	File Sharing
2000	Gnutella	File Sharing
	eDonkey	File Sharing
2001	Kazaa	File Sharing
2002	eMule	File Sharing
	BitTorrent	FileSharing
2003	Skype	Telephony
2004	PPLive	Streaming
2005	TVAnts	Streaming
	PPStream	Streaming
	SopCast	Streaming
2006	WoW Patch Dist.	File Sharing
	Symella	Mobile P2P
	SymTorrent	Mobile P2P
2007	PeerBox	Mobile P2P
	Joost	Video on Demand
	Vuze	Video on Demand; File Sharing
2008	Goalbit	Open Source - Streaming
2009	OneSwarm	Privacy Preserving for File Sharing

Table 2.1: P2P Evolution Time Line.

There are many other P2P networks in the research, educational and general applications area as described in *Internet2 Peer-to-Peer Working Group* at [11]. Just to refer a few:

- Research Applications: Intel Philanthropic Peer-to-Peer Program, SETI@home, World-wide Lexicon Project
- Educational Applications: eduCommons, Edutella
- General Applications: Chord Project, Groove Networks, JXTA, LOCKSS, The Metadata3 Project, etc

The advantages of P2P networking are so comprehensive, that even the latest Microsoft Windows operating system Windows Vista includes a P2P application for program, documents and desktop sharing. This is called *Windows Meeting Space*, successor of *Windows NetMeeting*.

> "Windows Meeting Space gives you the ability to share documents, programs, or your desktop with other people whose computers are running Windows Vista" [12].

Windows Meeting Space features are listed and detailed in Windows Vista SP1 local Help and Support. They allow to take advantage of cooperation in a LAN and can be used for:

- Sharing the desktop or any program with other meeting participants.
- Distribution and co-editing of documents.
- Distribution of notes to other participants.
- Connection to a network projector for presentation purposes.

By using P2P technology, Windows Meeting Space allows to automatically set up an ad hoc network for the tasks mentioned above. This way, it is possible to use it even when no network is available.

2.2 P2P Definition

P2P, in a computer context, refers to a network where each node has identical responsibilities and capabilities, can act as both client and server and it can start a communication with any other node. The main characteristics of P2P networks are: Low operation costs, fault tolerance and scalability. An example of a commonly accepted definition is one that can be found in [13]:

> "Peer-to-Peer Computing (Networking) Peer-to-Peer Computing (P2P Computing) is a type of distributed computing using P2P technologies that employ distributed resources to perform a function in a decentralized manner. Some of the benefits of a P2P computing include: improving scalability by avoiding dependency on centralized points; eliminating the need for costly infrastructure by enabling direct communications among clients; and enabling resource aggregation."

Since there is no need for central servers, any equipment connected to such a network provides additional resources, whether if it is bandwidth, storage, or computing power. No expensive hardware is needed, like in the Client/Server model, to support the operations for which the network is designed.

A permanent or temporary failure in a node or even in a group of nodes, does not compromise the entire network, because alternative network paths can be established between the nodes, so the resources can still be available and thus enabling fault tolerance.

Regarding scalability, this kind of network can increase until virtually no limit, allowing more and more shared resources each time a new node is included. The word virtually was used, because in practice, performance and usability in very large P2P networks may be affected. This happens particularly in a Purely Decentralized P2P architecture, where all peers perform exactly the same functions and no indexing servers exist. Although this is the best example of a P2P network, some recent protocols abdicate this architecture because it proved to be ineffective. P2P architectures will be further detailed in section 2.3.2.

2.3 Classification

P2P networks have evolved so much during the last years, that they are not generally associated only with file sharing programs anymore. Several architectures have been developed and adopted for a given purpose. P2P networks can be classified according to the functionalities they provide and their architecture.

2.3.1 Functionalities

Since the introduction of P2P networks, their applications have largely increased as many saw their enormous potential. From the late 90's music file sharing to proprietary gaming, audio and video streaming technology, they seem to be far from reaching their utility. These networks are currently available for:

- Content Distribution

  File Sharing (Gnutella, eDonkey, BitTorrent)

  Media Streaming (TVUPlayer, PPLive, Livestation, TVants, Goalbit, Joost)

- Distributed Computing

  SETI@Home

  Berkeley Open Infrastructure for Networked Computing (BONIC)

- Communications

  VoIP (Skype, SightSpeed, Aimini)

  Instant Messaging (AOL Instant Messenger, BLA Messenger, Yahoo!Messenger)

2.3.2 Architecture

P2P networks can also be classified according with their architecture. This is the way the peers communicate with each other in an overlay network. The existing categories are the result of constant evolution since the first centralized architecture until the most recent ones.

Not long after the shutdown of the Napster centralized network, completely decentralized ones such as Gnutella 0.4 emerged, providing the absence of a point of failure, as this network did not depend on a server or group of specific servers to operate. More recent architectures, had both characteristics of centralized and decentralized ones, relying on central servers for better resource location than those of purely decentralized architectures, although without depending completely on them. Features from more recent architectures have been recently incorporated into some well known P2P protocols as alternative searching mechanisms, completely independent from central servers and also providing all the other characteristics that made them so popular. All these architectures will further detailed in this section. The following figure represents all the P2P architectures along with some of the protocols that use them.

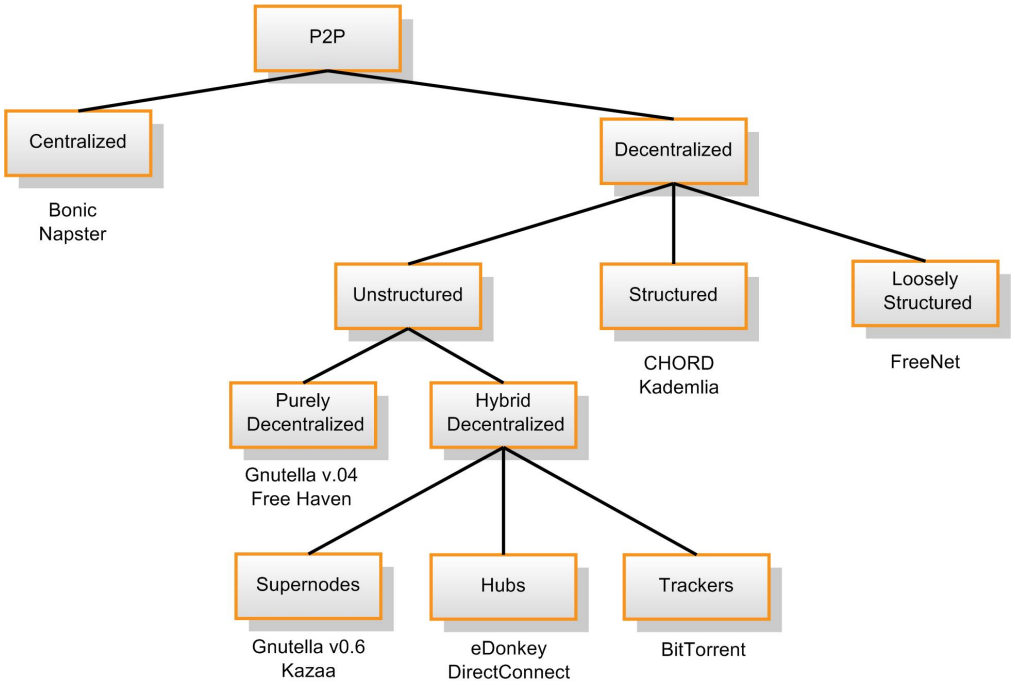

Figure 2.1: P2P Architecture.
Adapted from [14].

Centralized

A centralized P2P network is one that depends on a single or very few servers to operate [15]. These are responsible for indexing the information about the resources and the respective location (peer). When a peer in the network requests for some file, it connects firstly to the central server, which provides it the information about the peers containing that intended resource. After that, file transfers will be executed directly between the peers. Later, the indexing server will update its database including this latest peer also as a provider for such file.

Napster is the most well known example of the first P2P file sharing networks that used a centralized server, as it was already mentioned in section 2.1. In fact, all the existing architectures are the result of the success of this first P2P network. Centralized P2P systems provided some key benefits when compared to the later decentralized ones. This is the reason why some of the most popular P2P protocols still in use today have some of their features. These allow:

- Rapid and efficient file searching
- Discovery of all peers
- Registration of users to access network resources

By the other hand, when compared to decentralized P2P networks, centralized systems have the following disadvantages:

- Vulnerable to censorship and technical failure - Single network point of failure
- Possible overload of the server due to the demanding of popular data
- Central indexation might lead to oudated data, depending on periodically updates

It was the single point of failure characteristic in Napster, that allowed that the server shutdown in 2001 implied all the network failure.

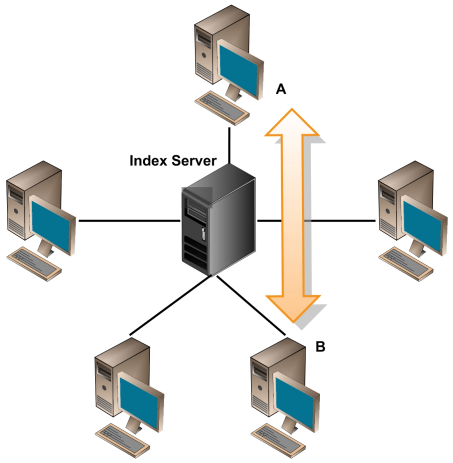

Figure 2.2: P2P Centralized Architecture.
Adapted from [14].

Figure 2.2 shows an example of a P2P centralized architecture where indexing tasks are done by a single server. For file transfers, the peers connect directly to each other.

Decentralized and Unstructured

A Decentralized P2P Network [15] is one that does not depend on a single server to operate, unlike in the Centralized architecture. This was the next evolutionary step taken, so that even in case of a legal order to shutdown a server, this would not compromise the entire network.

In an Unstructured Architecture, peers organize them self in a random graph topology. This means that peer links are established arbitrarily. Also, there is no correlation between a peer and the content managed by it. An example of an Unstructured Purely Decentralized P2P Network is the Gnutella version 0.4. When a client wants to connect to the network, it uses a *bootstrapping server* to connect at last to one peer. The problem with this model is that the search mechanism is inefficient, generating a considerable amount of traffic. When a peer wants to find some content, since there is no information about a resource and its location, it has to *flood* [2] the network with search requests and they may not even be

[2]In this context, flooding the network happens when many requests keep being sent to a network in order to find the location of specific resource.

resolved. The Unstructured Purely Decentralized P2P Network Architecture is displayed in figure 2.3.

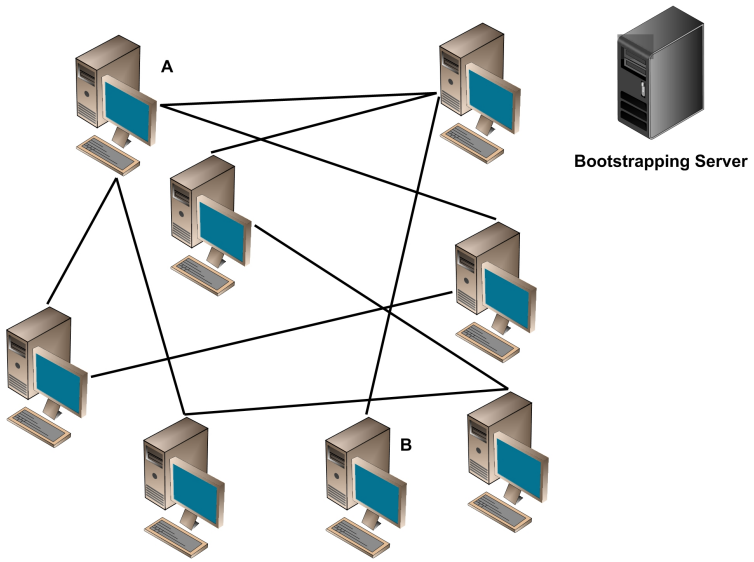

Figure 2.3: P2P Purely Decentralized Unstructured Architecture.
Adapted from [14].

Hybrid Decentralized Unstructured

The Hybrid Decentralized Unstructured Architecture [15] evolved to resolve the problem of inefficient search, typical of the previously presented Purely Decentralized Unstructured P2P Networks, in which there are no mechanisms for resource indexation. This P2P model has three subsets: Based in *Supernodes*, *Hubs* or in Distributed Servers and *Trackers*.

Hybrid Decentralized Unstructured Architecture Based in Supernodes

This architecture relies on the concept of *Supernode* (or *Ultrapeer*) which was introduced in protocols such as the Gnutella version 0.6 [16], Skype and the FastTrack based Kazaa application. These *Supernodes*, as the name implies, are more that the "regular" network peers. They can be elected automatically and also configured manually, if a user has enough resources (bandwidth, computing power) available and decides to contribute to a better network. They provide more scalability, as it is easier to keep information about any new resources available and better searching mechanisms as well. Another of their features, is that they allow multiple source downloads even from peers running different applications. Figure 2.4 shows a Hybrid Decentralized Unstructured Architecture Based on Supernodes, of which Gnutella v0.6 is an example.

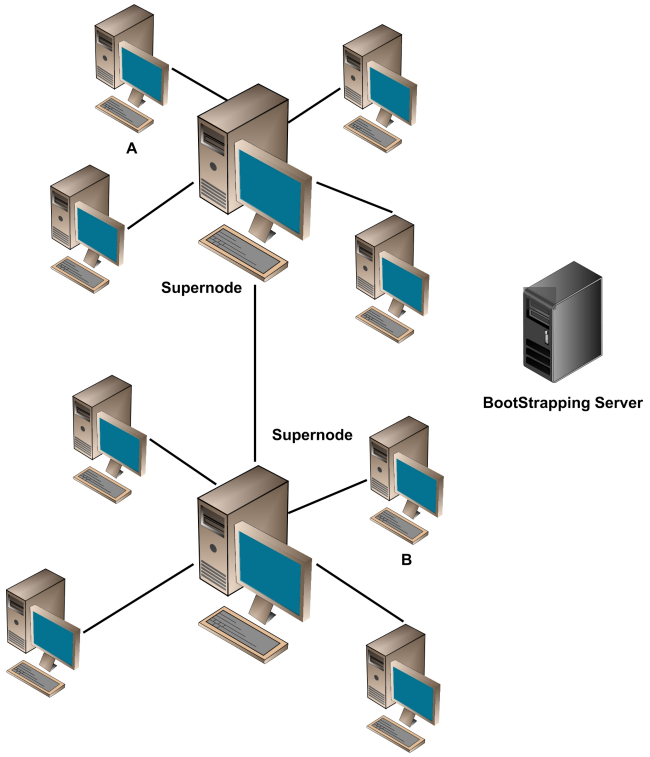

Figure 2.4: P2P Hybrid Decentralized Unstructured Architecture Based in Supernodes. Adapted from [14].

Hybrid Decentralized Unstructured Architecture Based in Hubs

In this kind of architecture, the P2P network contains hundreds of independent distributed servers [15] and files can be partially shared as they are downloaded. This is possible because they are equally split into several *chunks* [3] and when one of them is complete, it can automatically be shared. One can download many *chunks* simultaneously, each from a different location. This is called *Swarming*. Figure 2.5, shows the Hybrid Decentralized Unstructured Architecture based on Hubs used by the eDonkey [17] network (also called eDonkey2000 or simply ed2k).

Although the ed2k network had been shutdown by the Swiss and Belgium police in 2006, it is still very active today. At that time, *eMule* and *Shareaza* had already outnumbered the ed2k client, enabling other servers to keep the network alive. A user who intends to use a ed2k client, just has do download a text file usually also available at the site from which the application is being downloaded, containing several servers and respective IP addresses. These servers are then imported to the application itself, so that when it runs, it connects to one of those available servers. Most ed2k clients can be configured to automatically add new servers to the list as they are discovered.

[3] A chunk is a portion of a file. It varies according to the protocol being used and the size of the original file being downloaded.

The following message is displayed when one accesses the official eDonkey2000 (ed2k) site at [17].

"The eDonkey2000 Network is no longer available. If you steal music or movies, you are breaking the law. Courts around the world – including the United States Supreme Court – have ruled that businesses and individuals can be prosecuted for illegal downloading. You are not anonymous when you illegally download copyrighted material. Your IP address is xxx.xxx.xxx.xxx and has been logged. Respect the music, download legally."

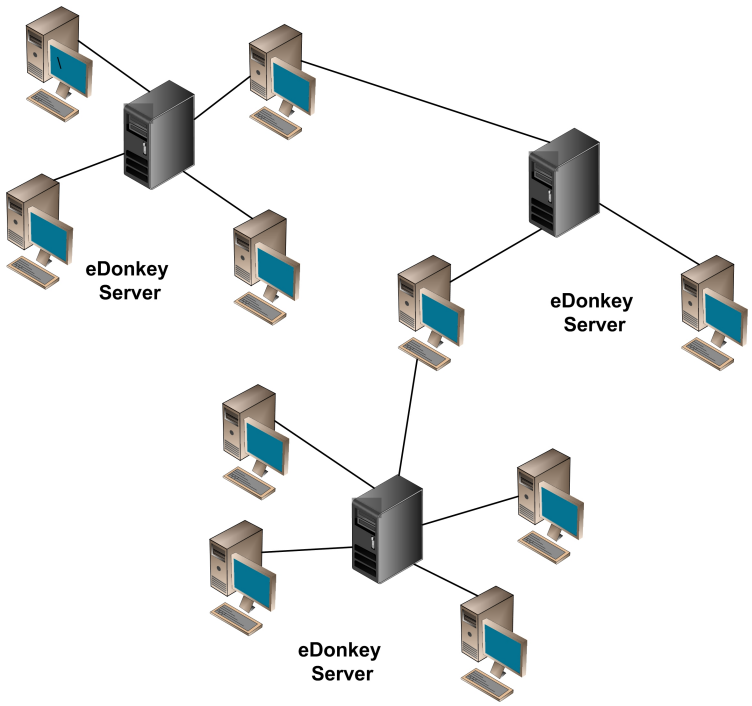

Figure 2.5: P2P Hybrid Decentralized Unstructured Architecture Based in Hubs. Adpated from [14].

Hybrid Decentralized Unstructured Architecture Based in Trackers

BitTorrent [18] is perhaps the most well known protocol that uses the Hybrid Decentralized Unstructured Architecture Based in Trackers [15]. It has the *tracker* and Web server as its main components. When a client intends some file, it downloads the *torrent* file, generally from a Web server. This torrent file contains metadata about the shared files and about the computer that coordinates the file distribution, called the tracker. A peer must have a torrent file for the intended download and connect to the specified tracker, so that it can obtain updated information about the peers to download from.

Just like the Hybrid Decentralized Unstructured Architecture Based in Hubs, the tracker based model also enables Download Swarm and the upload of partially completed files.

Recent BitTorrent applications like *Vuze*, do not necessary need a *tracker*, since they can use other mechanisms like the Distributed Hash Table, described bellow in sub section Decentralized and Structured, to obtain the resource location. Figure 2.6 shows a Hybrid Decentralized Unstructured architecture based on trackers of wich BitTorrent is an example.

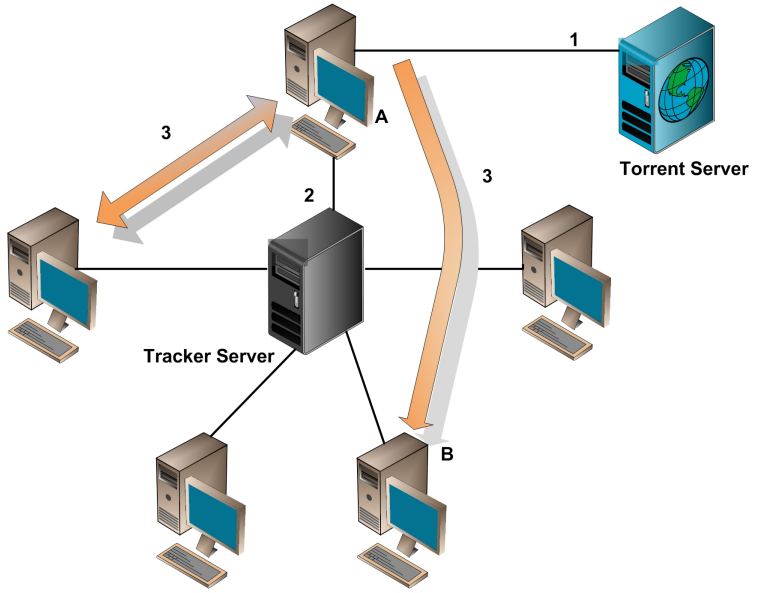

Figure 2.6: P2P Hybrid Decentralized Unstructured Architecture based in Trackers. Adapted from [14].

Decentralized and Structured

The main issue about Decentralized Unstructured P2P Networks [15], is their scalability limitation. This is particularly true in the case of Purely Decentralized P2P, since the mechanisms they use for content searching is quite inefficient. Recent P2P networks tend to use a Decentralized Structured architecture, to ensure that any peer can efficiently route a search to one another. This allows that even rare content can be more easily obtained than it Purely Decentralized Unstructured P2P Networks, where some search requests may not ever be answered at all.

The Decentralized Architecture requires a well defined topology with the data bound to it. The most common type of structured P2P network is the Distributed Hash Table (DHT) [19]. This is obtained by *hashing* [4] node information (nodeID), which can be the IP or MAC address of the node, the filename identification (dataID) and then the content is stored at the node whose nodeID is closest to the dataID.

However, there are some constraints in this mapping. Any particular node can disappear anytime, making the routing table hard to maintain. The load of the nodes should be equal,

[4]Hashing is the process of generating a fixed size alphanumeric code by applying a hash function to the initial input.

to avoid bottlenecks and, although this architecture enables keyword searching, the obtained results may be quite inaccurate.

Two examples of DHT protocol implementations are:

- CHORD
- Kademlia (KAD)

A common aproach for CHORD implementation is described in [20], being its main steps the following:

1. Assign random (160-bit) ID to each node
2. Define a metric topology on the 160-bit numbers, that is, the space of keys and node IDs
3. Each node keeps contact information to O(log n) other nodes
4. Provide a lookup algorithm, which finds the node, whose ID is closest to a given key. Implies a metric that identifies closest node uniquely
5. Store and retrieve a key/value pair at the node whose ID is closest to the key

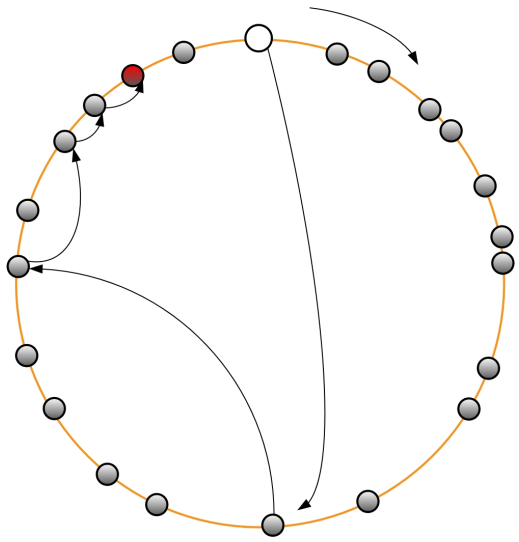

Figure 2.7: The Chord lookup.
Adpated from [20, 14].

In figure 2.7, one can see that queries are routed recursively to neighbors whose IDs are closer to that of the destination, with a total of *log n* hops, since according to [14],

> "Each step halves the topological distance to the target. So we have expected log n hops to the target."

Kademlia (KAD) [21] is another DHT system and uses the XOR metric and there is also a maximum number of *log n* hops from the source to destination nodes. Kad introduces another feature called the XOR metric, to determine the distance between any two nodes X and Y given by:

$$d(X,Y) = X \oplus Y \tag{2.1}$$

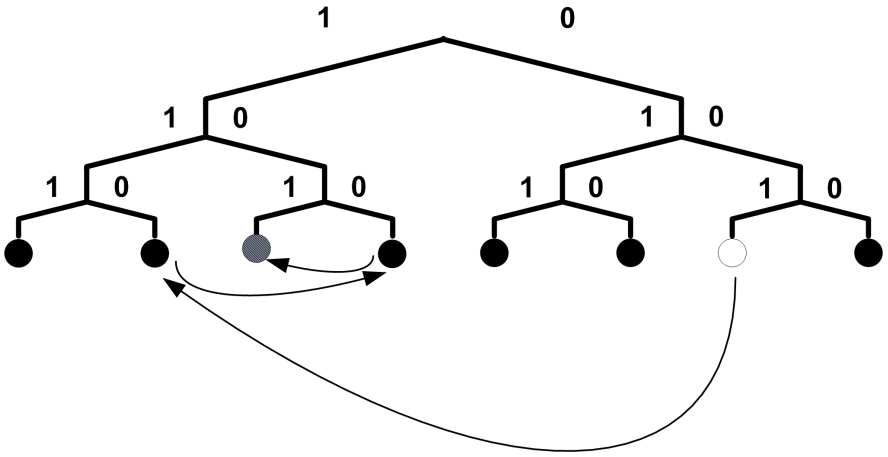

Figure 2.8: The Kad Lookup Tree.
Adpated from [21].

As one can see in figure 2.8, nodes in the same subtree are closer together than they are with nodes in other subtrees. These subtrees are built by using the hashed generated nodeID and the less different its bit representation is from another, the closer they are the tree. So one can easily verify that given any two bit arrays, differences at the higher order bits have a greater influence in distance calculation that low order bits.

$$\begin{array}{r} \mathbf{0}10\mathbf{1}01 \\ \mathbf{1}10\mathbf{0}01 \\ \hline \mathbf{1}00\mathbf{1}00 \end{array}$$

distance = $1 \cdot 2^5 + 1 \cdot 2^2$

Figure 2.9: Distance calculation using XOR metric

In other words, given any to peers, their position in the tree is given by an array of binary values. The closer they are, the less different they will be on the higher order bits. Only the positions containing different bit values, which are in fact the distance between any two peers, are considered for distance calculation. The conversion of the resulting binary value to decimal gives the actual distance between peers.

Decentralized and Loosely Structured

Decentralized and Loosely Structured P2P Systems are a particular case of Decentralized Structured ones. The overlay structure is not strictly specified as before, as it is either formed based on hints or probabilistically.

> "Loosely structured systems are a special type of structured systems where the peers estimate where is more likely that the resource will be found to route searches. The routing algorithm uses a heuristic, based on local information, and does not guarantee that the resource will be located. A well-known loosely structured network is FreeNet." [22]

In this kind of systems, data is identified by a key and the search is *lexicographically* [5]. Query responses are cached along the search path, as they are forward to a node neighbor. Initially, random decisions are made locally at the nodes to route the search path. As it evolves, nodes begin to cluster data whose keys are similar. Figure 2.10 shows the Decentralized and Loosely Structured Architecture.

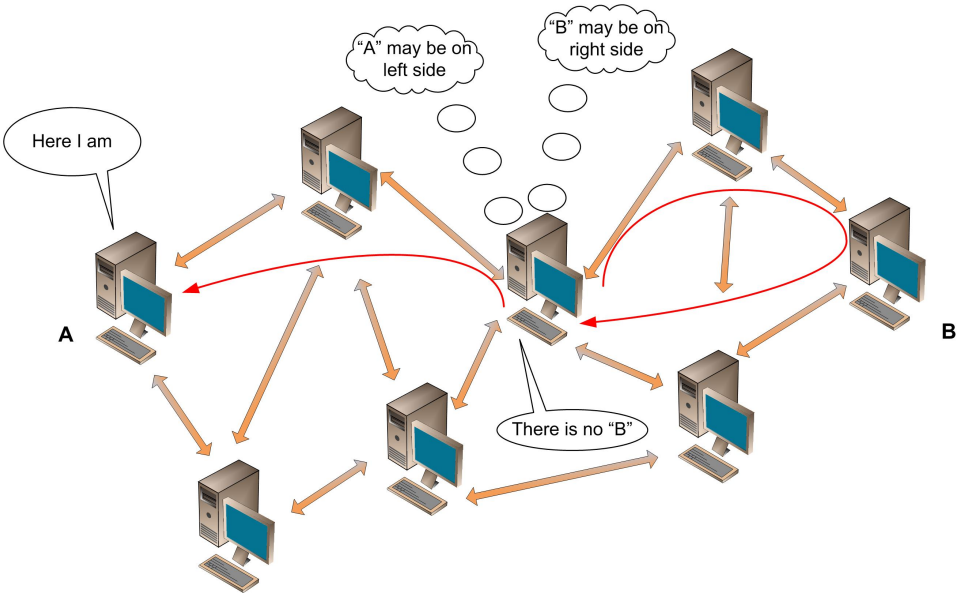

Figure 2.10: P2P Decentralized and Loosely Structured Architecture. Adapted from [23].

[5] Lexicographical refers the process of enabling a search through similar dictionary keys.

2.4 P2P Traffic Evolution

2.4.1 CAIDA

There are several web sites where one can access to worldwide information about the average routers response time, percentage of packet loss and traffic volume, such as Internet Traffic Report [24] or Internet Pulse [25]. Statistics like these are collected by ISP themselves, or by companies or organizations with access to some their edge router statistics or those of other institutions. They can provide general information about the Internet traffic of a certain location or even for a country, but since not all of it is accounted, those statistics are not 100% accurate.

When more detailed information is intended, a good starting point might be the Cooperative Association of Internet Data Analysis (CAIDA) site [26]. Nevertheless, obtaining information about P2P traffic is particularly hard.

In the beginning, P2P applications used well known ports for communicating, just like in the Client/Server model. Later, when they became popular and unwanted by many ISPs some organizations, due to the considerable amount of traffic generated by them, their programmers started to include random port functionalities so they could go unnoticed. Many P2P applications nowadays support encryption or obfuscation, which makes them difficult to detect and, consequently, to account. Table 2.2 contains information about worldwide P2P traffic share. More recent and complete information will be further displayed in this section.

Geographic Location	Year	P2P %
Europe	2005	60-80
	2006	79-93
North America	2003	8
	2004	14
	2003-04	9.19-70
	2006	21-35.1
Asia	2002	21.53
	2005	1.34
	2008	1.29

Table 2.2: P2P Geographical Distribution.
Adapted from [26].

These numbers were obtained by statistical or behavioral classification and by packet inspection "[...]the most reliable method of detecting an application (if unencrypted), which however is fraught with legal and privacy issues." [26]. These legal and privacy issues will be further detailed in section 2.5.1.

2.4.2 ipoque

Specific information about P2P traffic can be obtained, for example, at the *ipoque* company site [27]. Ipoque was founded in 2005 in Leipzig, Germany and it provides deep packet inspection solutions for Internet traffic management and analysis. Many of their products are used in big companies and ISPs with several thousands and even millions of subscribers. Since 2006, ipoque has been conducting annual detailed studies about P2P traffic share and applications. Initially, it was more focused in Germany, being later extended to the rest of Europe and nowadays worldwide, involving eight ISP and three Universities.

> "For the third year in a row, ipoque executives Klaus Mochalski and Hendrik Schulze conducted a comprehensive report measuring and analyzing 1.3 petabytes of Internet traffic." [27]

ipoque - P2P Survey 2006

For the first of these studies [28], from March to October 2006, most of the data was gathered in Germany. However, it provides a comprehensive overview of all P2P Internet traffic in Europe. In this period, 70% of all nightime Internet traffic in Germany was P2P, versus the 30% at daytime. This shows how important was for ISP and companies to have better means to identify P2P traffic, so they would be able to block it, or, more likely, to shape it [6]. According to this study, BitTorrent overtook eDonkey in popularity in Germany and together they were responsible for more than 95% of all P2P traffic.

Figures 2.11 and 2.12 show, respectively, the share of P2P Protocol distribution in Germany and the rest of Europe in 2006.

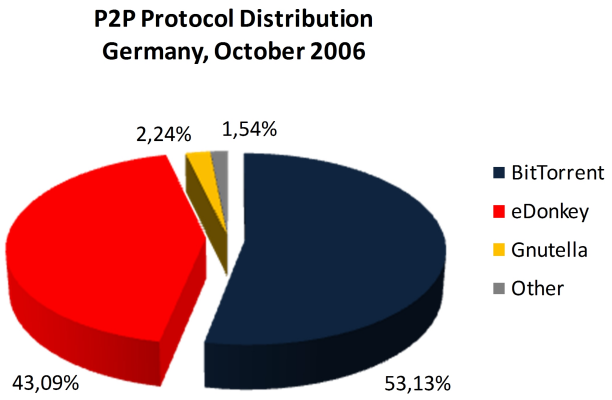

Figure 2.11: Distribution of P2P Protocols in Germany, October 2006. Adapted from [28].

[6] Traffic shaping is the ability to control the priority of packets according to some criteria.

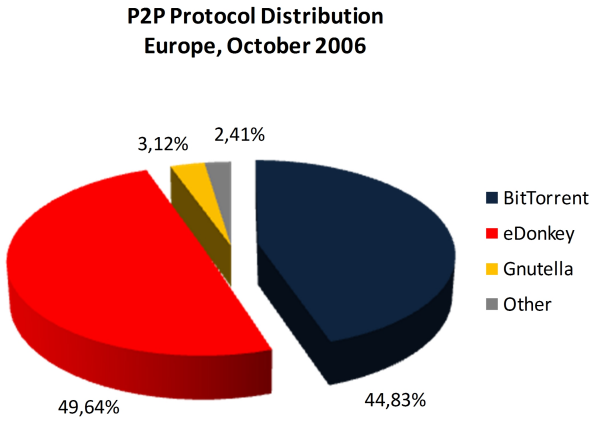

Figure 2.12: Distribution of P2P protocols in Europe, October 2006.
Adapted from [28].

Although the values of German and European P2P protocol distribution were slightly different, any of the previous charts provides an approximate scenario of the other.

As for the contents being shared, these were mainly movies, music and video games, followed by a growing share of eBooks and audio books, as one can see in figure 2.13, relative to German BitTorrent P2P traffic.

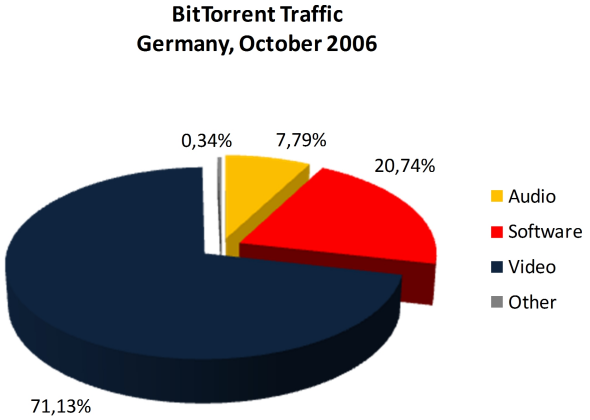

Figure 2.13: BitTorrent Traffic Share in Germany, October 2006.
Adapted from [28].

ipoque - Internet Study 2007

In 2007, ipoque conducted another study about Internet traffic [29]. Besides P2P file sharing protocols, it also included Skype, video streaming, instant messaging and file hosting. An interesting fact is that only BitTorrent and eDonkey were considered among P2P file sharing protocols, mainly due to their greater popularity and because the task of analyzing traffic content is very time consuming, since it "involves a substantial amount of manual work" [29].

More regions were included regarding the study of 2006, representing over one million users in Australia, Eastern Europe, Germany, the Middle East and Southern Europe. "The data were gathered using ipoque's PRX Traffic Manager installed at customer sites." [29] According to this study, P2P was producing more traffic in the Internet then all other applications combined. Its average proportion from August to September 2007 ranged regionally between 49% in the Middle East and 83% in Eastern Europe, reaching peaks of over 95% at nightime. Another interesting fact was that 20% of P2P traffic (BitTorrent and eDonkey) already used encryption. The worldwide amount of P2P traffic in 2007 is shown in figure 2.14

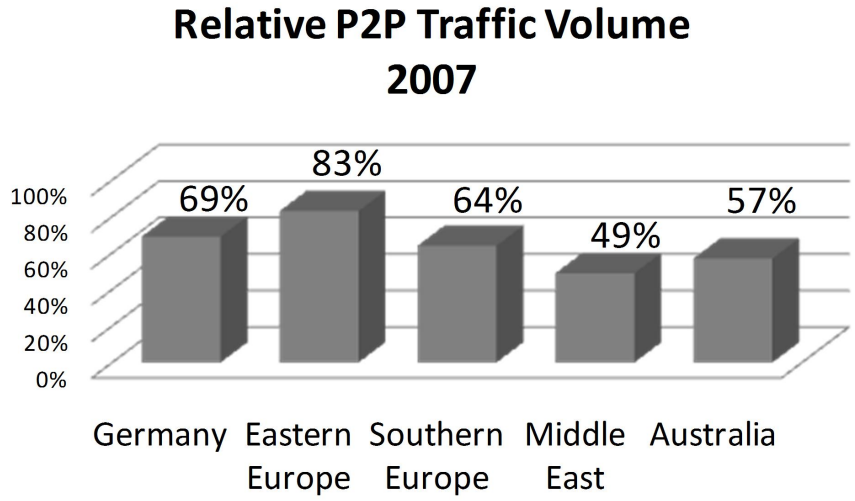

Figure 2.14: Relative P2P Traffic Volume, 2007.
Adapted from [29].

Table 2.3 shows detailed information about geographical traffic distribution.

It is important to notice that Web embedded audio and video streaming, like YouTube [30], was counted separately from HTTP traffic. Nevertheless, P2P protocols were by far those that generated the larger volume of traffic.

Protocol	Germany	East. Europe	South. Europe	Middle East	Australia
P2P	69,25%	83,46%	63,94%	48,97%	57,19%
HTTP	10,05%	-	-	26,05%	-
Streaming	7,75%	-	-	0,7%	0,02%
DDL	4,29%	-	-	8,66%	-
VoIP	0,92%	-	-	0,57%	0,51%
IM	0,32%	-	-	0,24%	0,36%
E-Mail	0,37%	-	-	0,79%	-
FTP	0,5%	-	-	0,62%	-
NNTP	0,08%	-	-	0,23%	-
Tunnel/Enc.	0,32%	-	-	1,65%	-

Table 2.3: Geographical Traffic Distribution, 2007
Adapted from [29].

Comparatively to 2006, P2P traffic has still grown in 2007, but it did not outperform the overall traffic growth. The main reason for this was the growing of Direct Download Link (DDL) services such as MegaUpload [31], RapidShare [32], etc. At that time, BitTorrent had become the most popular P2P protocol worldwide. The only region where eDonkey was still leading, was in Southern Europe with a share of 57% of all P2P traffic. In Eastern Europe DirectConnect had a high P2P traffic share of 29%. In Australia Gnutella share reached 9% of all P2P traffic, but the most significant traffic volumes were for the eDonkey and BitTorrent protocols, with a share of 14% and 73% respectively [29] . Table 2.4 shows the P2P protocol distribution across the same geographical areas as in table 2.3

Protocol	Germany	East. Europe	South. Europe	Middle East	Australia
BitTorrent	66,70%	65,71%	40,09%	56,21%	73,40%
eDonkey	28,59%	2,66%	57,05%	38,51%	13,58%
Gnutella	3,72%	1,90%	2,23%	3,10%	8,84%
DirectConnect	0,52%	28,72%	0,18%	0,39%	0,28%
Other	0,47%	1,01%	0,45%	1,97%	3,90%

Table 2.4: Geographical P2P Protocol Distribution, 2007.
Adapted from [29].

Since 2005 that BitTorrent clients BitComet and Azureus suported encryption. Later in 2006, eMule was one of the first eDonkey clients to use obfuscation. An important part of this study included statistics about the use of encryption/obfuscation in P2P traffic. Table 2.5 shows geographic encrypted/obfuscated P2P traffic distribution share.

	BitTorrent	eDonkey
Germany	18%	15%
Midle East	20%	13%
Australia	19%	16%

Table 2.5: Volume of encrypted P2P traffic, 2007.
Adapted from [29].

As one can see in table 2.5, the values relative to the usage of encryption for BitTorrent and eDonkey protocols are very similar for each region. Just like in 2006, there is much more information available in this report, covering P2P content by type and even a ranking for BitTorrent and eDonkey most shared data.

ipoque - Internet Study 2008/2009

ipoque latest study is relative to 2008/2009 [1]. More regions were included and now they are Northern Africa, Southern Africa, South America, Middle East, Eastern Europe, Southern Europe, Southwestern Europe and Germany. The data from more than one million users was analyzed, which reached 1.3 petabytes. It was collected at eight ISPs worldwide and three universities. The main conclusions were the following:

- P2P generates most traffic in all regions
- The proportion of P2P traffic has decreased
- BitTorrent is still number one of all protocols, HTTP second
- The proportion of eDonkey is much lower than last year
- File hosting has considerably grown in popularity
- Streaming is taking over P2P users for video content

Table 2.6 shows the protocol class proportions for 2008/2009. An interesting conclusion was that P2P traffic share has decreased in all regions. This does not mean necessarily there is less P2P traffic than in 2007, “but only that P2P has grown slower than other traffic” [1]. According to ipoque, precise comparison results with previous years were only possible for Germany and Middle East. This is due to the changing of many participating measurement points for this study.

Protocol	S. Africa	S. America	E. Europe	N. Africa	Germany	S. Europe	M. East	SW Europe
P2P	65,77%	65,21%	69,95%	42,51%	52,79%	55,12%	44,77%	54,46%
Web	20,93%	18,17%	16,23%	32,65%	25,78%	25,11%	34,49%	23,29%
Streaming	5,83%	7,81%	7,34%	8,72%	7,17%	9,55%	4,64%	10,14%
VoIP	1,21%	0,84%	0,03%	1,12%	0,86%	0,67%	0,79%	1,67%
IM	0,04%	0,06%	0,00%	0,02%	0,16%	0,03%	0,5%	0,08%
Tunnel	0,16%	0,1%	-	-	-	0,09%	2,74%	-
Standard	1,31%	0,49%	-	0,89%	4,89%	0,52%	1,83%	1,23%
Gaming	-	0,04%	-	-	0,52%	0,05%	0,15%	-
Unknown	4,76%	7,29%	6,45%	14,09%	7,84%	8,86%	10,09%	9,13%

Table 2.6: Protocol Class Proportions 2008-2009.
Adapted from [1].

In figure 2.15, it is possible to see the most relevant traffic changes since 2007.

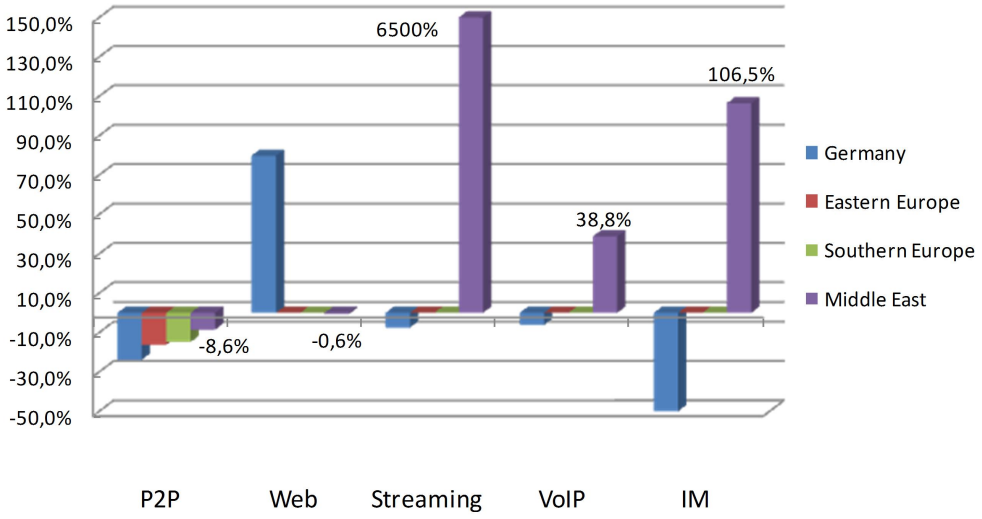

Figure 2.15: Protocol Proportion Changes relative to 2007.
Adapted from [1].

There might be several reasons for the decrease of P2P share relative to other protocols. Many ISPs are nowadays concerned about this issue and started to *throttle* [7] P2P traffic. Even not all of them use these mechanisms, the existence of throttled peers in a P2P network may be enough to reduce its overall download capacity, thus discouraging its users. Another reason might be the increasing number of alternatives for file sharing like DDL, already mentioned previously. This can reduce P2P traffic to rise HTTP instead. On the other hand, in the past few years, there has been an increasing of legislation concerning software piracy in many countries. Many of data shared in these networks is copyright-protected material, whether they are movies, music, eBooks, etc. Although there are very few cases

[7] To throttle traffic means to be able to set its priority according to some criteria.

of prosecution, authorities launch operations against these networks which may dissuade many users.

As for encrypted/obfuscated P2P traffic, the 2008/2009 study only provides information about BitTorrent and eDonkey in Germany and Southern Europe. It is only possible to compare its evolution in Germany, since it is the only region common to both 2007 and 2008/2009 reports. For eDonkey, the relative amount of obfuscated traffic remains almost unchanged. It increased 1% comparatively to 2007 reaching 16% of the overall eDonkey traffic. Encrypted BitTorrent also increased but at a greater proportion, with a value of 23% in 2008, 5% more than in the previous year. According to this study, "In Southern Europe, the disparity in encryption usage between these two most popular networks is even greater" [1].

The higher encrypted BitTorrent traffic share might be justified by more frequent releases and updates for their most known clients (like Vuze, formerly Azureus), unlike the few of eMule and aMule, the most popular eDonkey clients. Many of the latest improvements in this software allow new functionalities for encryption/obfuscation, so more releases might translate into less plain data exchange. Table 2.7 shows the relation between encrypted and unencrypted BitTorrent and eDonkey traffic in Germany and Southern Europe.

	BitTorrent		**eDonkey**	
	Encrypted	Unencrypted	Encrypted	Unencrypted
Germany	22,81%	77,19%	16,08%	83,92%
Southern Europe	26,21%	73,79%	7,03%	92,97%

Table 2.7: Proportion of encrypted and unencrypted BitTorrent and eDonkey traffic in Germany and Southern Europe.

Adapted from [1].

2.5 State of Art in P2P Detection

2.5.1 Legal Issues

P2P traffic detection has caught the attention of several companies focused on traffic filtering and optimization. There has been an increasing demand by ISPs for solutions of this type in order to keep competitive in providing services for their clients. An overloaded network with P2P traffic, means a slower connection for all users. One can easily understand that a subscriber with a much slower connection than that he had contracted, might want to change to another ISP who can guarantee a better service.

A study conducted from August to December of 2006 by the former Internet traffic management company Ellacoya (now integrated into Arbor Networks [33]) analyzed the data of about 2 million Internet users and assigned them into five categories: " "bandwidth hogs," "power users," "up and comers," "middle children," and "barely users." As it turns

out, bandwidth hogs only make up about 5% of the entire Internet-using audience, but generate about 43.5% of the total traffic. Conversely, another 40% of users (the barely users) make very light use of the Internet and only generate about 3.8% of traffic. The remaining 55% of users generate the remaining 50% of traffic." [33]

As one can see, a small share of users is the one who uses most resources and may slow down the network connection for all the others. Many ISPs nowadays depend on very expensive hardware, acquired from companies like Arbor Networks [33], Sandvine Incorporated [34] or ipoque [27] just to cite a few, to apply traffic policies to the entire network and maintain the quality of their services.

The use of DPI (if not by itself, then combined with other technologies) to identify P2P traffic, brings up another current issue: privacy. As people have more information about the methods used by ISPs to control/shape their traffic, they tend to be more concerned about the protection of their personal information. This issue was initially discussed in the United States and in Canada, but currently there is going on a worldwide heated debate concerning Net Neutrality, particularly in the European Union [35], where it achieved and enormous publicity since 2008. That was the time when Malcom Harbour [36] presented the report *Electronic communications networks and services, protection of privacy and consumer protection* [37], commonly known as the *Harbour Report* or *Telecoms Package*. The following citation was taken from [38], which is one of the many organizations committed to fight against some changes proposed in that report.

> "On May 6th, pressure from EU citizens has meant that the Directives that attempted to privatize the Internet were not passed in the vote in the European Parliament. This Autumn the Package will be negotiated again. "[38]

Another example concerning the Net Neutrality issue came from the Office of the Privacy Commissioner of Canada, which asked the Canadian Radio-television and Telecommunication Commission (CRTC) to initiate a public proceeding to review the Internet traffic management practices of ISPs, from November 2008 to February 23 of 2009. More information can be found at [39].

Maybe the most well known case of a legal action applied to an ISP, is the one of Comcast Corporation [40], the largest provider of cable services in the United States and the second largest ISP. According to [41, 42, 43], Comcast used the hardware of the Canadian company Sandvine in late 2006 to send forged TCP RST (reset) packets, disrupting multiple protocols used by peer-to-peer file sharing networks. This has prevented some Comcast users from uploading files. After a lot of controversy and many unhappy subscribers, the US communications regulator, the Federal Communications Commission (FCC), has ordered Comcast to stop treating P2P traffic differently from other on August 21, 2008 [44].

2.5.2 Classification of Mechanisms for P2P Traffic Detection

The traffic generated by first generation P2P applications was relatively easy to detect due to the fact that these applications used well-defined port numbers. However, nowadays, the traffic generated by P2P applications may be very difficult to detect because P2P applications may change the default service port or use port 80, for example, which is assigned

for HTTP traffic and therefore, may not even be blocked in most organizations. Besides, they may use encryption or obfuscation options making very difficult to detect this kind of traffic. On the other side, link speeds are reaching 1-10 Gbps in local area networks, which may become infeasible the detection since the processing speed may not match the line speed and capturing every packet may pose severe requirements in terms of processing or caching capacity. The use of encryption/obfuscation by many recent P2P applications provides them the theoretical advantage against DPI, although, as it will be shown later in section 2.5.4 (see figure 2.23), there are some claims about its possible detection, at least for some traffic portions of a given P2P protocol.

Recently, several approaches have been proposed to detect P2P applications. These techniques may be classified into two main categories [45, 2]: (i) based on payload inspection or signature-based detection, and (ii) based on flow traffic behavior or classification in the dark. Deep packet inspection methods inspect the packet payload to locate specific string series, which are called signatures that identify a given characteristic, a given protocol or a given application, where as methods based on traffic behavior attempt to detect and classify possible protocols or applications without looking into the payload contents.

Some approaches have been proposed for traffic identification using behavior-based methods. The method based on transport-level connection patterns relies on two heuristics for P2P traffic classification [45]:

(a) It involves the simultaneous use of TCP and UDP by a pair of communicating peers.

(b) Regarding the connection patterns for (IP, port) pairs, the number of distinct ports communicating with a P2P application on a given peer will likely match the number of distinct IP addresses communicating with it.

The behavioral method based on entropy reported in [2] requires the evaluation of the entropy of the packet sizes in a given time window and works on-thefly. Several approaches requiring the analysis of some fields of the header of TCP or IP packets for flow-based P2P traffic detection have been proposed based on machine learning [46, 47], support vector machines [48, 49] and neural networks [50]. This kind of methods may be used for high-speed and real-time communications with encrypted traffic or unknown P2P protocols. The main drawback is the possible lack of accuracy in the identification of P2P traffic.

	Advantages	Drawbacks
DPI	Great precision Less False-Positives	New or unknown protocols Use of Encryption Privacy issues Performance
Traffic Flow Behavior	Better Performance Encrypted Traffic Privacy guaranteed	Less Precision

Table 2.8: DPI versus Traffic Flow Behavior Methods

.

Some of the advantages of using Traffic Flow Behavior Methods over DPI are notorious, specially when it comes to performance and privacy issues. As referred previously in section 2.5.1, concerns about the legal aspects of analyzing packets payload have increased and there have already been cases where this practice was condemned.

DPI has not as much potential use for encrypted connections, due to the nature of encrypted traffic itself, unless encryption is broken somewhere between peers. Although this might be very hard to achieve, it is at least possible through a Man-in-The-Middle attack in one of the communication end-points. After one captures the key exchange, he can use its machine to impersonate an actual peer and decrypt all the P2P traffic. Then, it would be "simply" a matter of applying DPI to check against well known protocol signatures. This approach was not followed for this work due to privacy issues and its great complexity for the available for this project.

Since the introduction of encryption/obfuscation on many P2P clients, many open source software developers withdrew their focus on DPI as this became a very hard and time consuming task, on which no guaranteed results can be expected. Nevertheless, the purpose of this work was to study the possibility to detect encrypted P2P file sharing traffic and P2P TV traffic (mainly proprietary, from which scarcely information is available).

2.5.3 Currently Available DPI Software

In the beginning, P2P applications used a specified port or range of ports. Blocking this traffic, was just a matter of creating some firewall rules on the hardware of software based router, to disable communications on them. If disabling it was no an option, one could even define a minor priority for that traffic, so that the network performance would not be affected.

The next step in the evolution of P2P applications, which is still a default on most of them when running their installer, was the randomization of their TCP and UDP ports. The previous approach became useless, since one could not just block random ports hoping to detain the unwanted traffic. As a countermeasure, network administrators applied more restrictive policies to the incoming and outgoing packets. An usual way to do this is to block everything, except for incoming traffic for essential services provided by the company or institution itself, or for specific allowed outgoing communications. This last one is not so much taken into account for two main reasons. The first one is that t here is a lot more tendency for one to care more about what is allowed to enter in its network than on what goes out of it. The second is more related to the required maintenance of a system like this. In an University or research center, for example, there are usually less restrictive policies for outgoing traffic than at a commercial company. There can be the need to access many different external software and services for investigation and teaching purposes, which, with an established outgoing traffic blocking policy, would need constant firewall rules updates.

Even with just a few allowed ports for external communication, P2P applications were not defeated yet. They started to use "'traffic impersonation"', which consists in using the same ports used by applications like HTTP (TCP port 80), that can not be blocked in most organizations. To successfully identify P2P traffic, it was now necessary to use a different

approach focused on the payload of a packet, instead of the source or destination port used by it. This is called DPI, as already been introduced previously in chapter 1.

In the following, a description about commercial and open source DPI solutions is presented. Commercial solutions include both software and hardware, while the open source approaches are only available as software.

WFilter

There are several commercial solutions available for filtering contents in Web, e-Mail, Instant Messaging filtering and even P2P. One of them is the awarded *WFilter Enterprise*, available at the IMFirewall site [51]. IMFirewall Software Co., Ltd. is located in Nanjing, China and it was founded in 2004 with a strong focus on internet filtering and content management.

Although it was not intensively tested during this work (neither its results were accounted, since only a 15 day trial version was available for use), this software showed to be quite effective on detecting unencrypted P2P traffic for protocols such as BitTorrent and PPLive (P2P TV). More tests were needed to evaluate its potential capabilities.

According to its description, WFilter can detect and block P2P and several other protocols using pattern matching, in other words, DPI. WFilter features are:

- Define a file extension list forbidden from being download.
- Can be installed on a single Windows machine for a small network(1-500 Users)
- Analyzes network traffic to do monitoring
- Should be deployed at a location where it can see all Internet traffic
- Protocol Analysis

  P2P - Define policies to block over 30 P2P protocols.

  Messenger Clients

  File Transfer

  Online Streaming

  Emails - including use of SSL

IMFirewall also provides information about its supported protocols and applications, such as TCP and UDP ports used by them. During this work, it was interesting to notice that this list has slightly increased since December 2008 until the beginning of May, mostly in what concerns to online streaming, which is a good indicator of the current validity of DPI.

Another approach, is to use open source software for exactly the same purpose. Although there were not found results for any study comparing the effectiveness of commercial and open source solutions during this work, these last have one significant advantage over their alternatives. One can read the source code, modify it, or even add new features into it, according to its needs, as in the case of the following studied applications.

ipp2p

One example of an open source P2P traffic classifier is *ipp2p*[52], sponsored by the ipoque company. This software uses an extended iptables/netfilter architecture so it can "'easily"' be integrated with any recent Linux system. To do this, one has to execute the following steps:

- Possess the ipp2p, linux kernel and iptables source code
- Compile ipp2p specifying the kernel and iptables source locations
- Copy libipt_ipp2p.so to the iptables library directory, usually located at /usr/lib/iptables/
- Load the newly created kernel module (ipt_ipp2p.ko for 2.6.x kernels) to be able to use ipp2p module in iptables. Preferably with *modprobe* instead of the ipp2p documentation suggested *insmod*

When up and running, ipp2p enables P2P traffic detection by applying search patterns into the payload of a packet, obtained with ipp2p iptables module . If the traffic matches a specified rule, iptables can drop such traffic, lower its priority, shape it into a given bandwidth, or simply log it.

ipp2p version 0.8.2 was used to study its pattern matching mechanisms. It is written in C Language and its source code is distributed across three files.

- ipt_ipp2p.c (pattern matching file)
- ipt_ipp2p.h
- libipt_ipp2p.c (main file)

The following code was extracted from **ipt_ipp2p.c** and it detects Gnutella UDP traffic, by searching the first three and nine bytes for the strings **GND** and **GNUTELLA** respectively.

```
/*Search for UDP Gnutella commands*/
int udp_search_gnu (unsigned char *haystack, int packet_len)
{
unsigned char *t = haystack;
t += 8;
if (memcmp(t, "GND", 3) == 0) return ((IPP2P_GNU * 100) + 51);
if (memcmp(t, "GNUTELLA ", 9) == 0) return ((IPP2P_GNU * 100) + 52);
return 0;
}/*udp_search_gnu*/
```

Figure 2.16: ipp2p function to identify Gnutella UDP traffic.

According to ipp2p documentation and source code, this version detects the following P2P protocols:

- All known eDonkey/eMule/Overnet TCP and UDP packets
- All known Direct Connect TCP packets

- All known KaZaA TCP and UDP packets
- All known Gnutella TCP and UDP packets
- All known BitTorrent TCP and UDP packets
- All known AppleJuice TCP packets
- All known WinMX TCP packets
- All known SoulSeek TCP packets
- All known Ares TCP packets
- Experimental

  All known Mute TCP packets

  All known Waste TCP packets

  All known XDCC TCP packets (only xdcc login)

It is important to notice that these rules were made only with plain traffic (no encryption/obfuscation) in mind. Nevertheless, as it will be detailed in chapter 4, it is possible to use them to detect some traffic of P2P applications, even when they are configured to only use encrypted outgoing and incoming communications.

l7-filter

Another popular open source traffic classifier is *l7-filter* available at [53]. It is an Application Layer Packet Classifier for Linux, which explains the *l7* [8] in its name.

l7-filter also reads information from iptables/netfilter, like ipp2p, but the process to compile it is a little bit more complex since one has to patch the linux kernel. Complete information can be obtained at [53]. It is necessary to obtain:

- 2.4 or 2.6 Linux kernel source (2.6 strongly preferred) from kernel.org
- iptables source from [54]
- "l7-filter kernel version" package (netfilter-layer7-vX.Y.tar.gz)
- "Protocol definitions" package (l7-protocols-YYYY-MM-DD.tar.gz)

According to the source code of the 18 December 2008 version, l7-filter can detect 111 protocols, 2 types of malware, 16 common file types and 12 additional traffic signatures. It has builtin support for major P2P protocols like BitTorrent, eDonkey, Gnutella, Ares, and many many more. Unlike ipp2p (specific to P2P detection), where all the pattern matching

[8] Layer 7, which is commonly abbreviated as L7, represents the Application Layer in the OSI network model.

for the protocols is done in a single C file, l7-filter uses a separate file for each and uses regular expression patterns.

The following excerpt code shown in figure 2.17 was extracted from bittorrent.pat and edonkey.pat l7-filter protocol files and specify the pattern matching for BitTorrent and eDonkey respectively. These are not so "'fined tunned"' as other existing patterns on those files, but are easier to understand and display.

```
BitTorrent

# This pattern is "fast", but won't catch as much

^(\x13bittorrent protocol|azver\x01$|get /scrape\?info_hash=)


eDonkey

# matches everything and too much

^(\xe3|\xc5|\xd4)
```

Figure 2.17: BitTorrent and eDonkey search patterns used in l7-filter. Extracted from bittorrent.pat and edonkey.pat available at [53].

For BitTorrent traffic, it will check a packet payload against the following values:

- Hexadecimal value **13**, followed by the string "'bittorrent protocol"'
- string "'azver"' followed by the hexadecimal value **01**
- string "'get /scrape\?info_hash="'

For eDonkey, it will check if the first bytes, in hexadecimal format, against the values **e3**, **c5** and **d4**. As it is mentioned in the comment referring do eDonkey in table 2.17, that pattern will match all eDonkey traffic and many more, causing a high number of false positives [9]. Due to the large number of existing eDonkey messages and those specific to some applications like eMule, called eDonkey extensions, these patterns can be tunned to detect more specific messages as it will be shown in 4.4.1. Nevertheless, the false-positives obtained will be inevitable. In case a packet matches one of the above patterns, the l7-filter module for iptables enables it to trigger one of the usual actions: Drop, lower its priority or log it.

Just like ipp2p, l7-filter default P2P pattern files were intended to work with plain data payloads. There is o guarantee that they will work with encrypted or obfuscated traffic, although they might detect some P2P traffic for protocols in which, parts of the transfer or control messages are transmitted in plain data. There has not been seen many development specifically concerning encrypted P2P detection for open source software, as it depends on volunteers to keep this work. Moreover, it is a very time consuming and hard task, without guaranties from the start that any successful results will be achieved.

[9] In this context, false positives are traffic that is mistakenly classified as one protocol, when in fact it belongs to another.

2.5.4 Currently Available DPI Hardware

Due to the enormous amount of P2P traffic traveling daily through the Internet, many companies, institutions, and ISPs, have been forced to apply restrictions to it for policy reasons, or to guarantee the network performance for users or subscribers. The methods and tools available for this job have hugely evolved to keep up with the encryption/obfuscation features of recent applications. Since simple firewall rules to the state of the art hardware, a long way has been traveled. Just like in a war, where the appearance of a new weapon implies a matching counter measure, a successful method to detect P2P traffic forces developers to the find new or better alternatives to keep it stealth.

Nowadays there are several specialized DPI Hardware manufacturers. The following figures show some of equipment already mentioned of Arbor Networks, ipoque and Sandvine Incorporated, along with some of its key features.

Figure 2.18: Arbor eSeries e30 (4 Gbps; 64000 subscribers). Taken from the Arbor eSeries © Data Sheet [55].

Figure 2.19: Arbor eSeries e100 (20 Gbps; 500000 subscribers). Taken from the Arbor eSeries © Data Sheet [55].

Figure 2.20: ipoque PRX-5G (4 Gbps; 500000 subscribers; 20 million concurrent flows). Taken from the ipoque PRX Traffic Manager series © Data Sheet [56].

Figure 2.21: ipoque PRX-10G (75 Gbps; 6 million subscribers; 240 million concurrent flows). Taken from the ipoque PRX Traffic Manager series © Data Sheet [56].

Figure 2.22: Sandvine PTS 14000
(80 Gbps). Taken from the Sandvine Policy Traffic Switch series © Data Sheet [57].

As one can easily see, this is state of the art DPI hardware. It reaches hundreds and in some cases even exceeds the million dollar price per unit, making them affordable only a restrict number companies. Among them are some of the largest ISPs, like the already mentioned Comcast, that are willing to make an investment of this order to maintain their network traffic optimized and access many other provided tools, like for supporting network integrity. It is important to notice, that all of the previous models shown before provide more features than DPI, but this last one is the most important for this work.

One relevant question one can ask, is how effective these systems are. To answer it, the European Advanced Networking Test Center (EANTC) [58] decided to conduct a six months test with the most representative P2P filtering manufacturers.

EANTC is a German public limited company (AG) located in Berlin. Until 1999 EANTC was part of the Interdepartmental Research Center for Networking and Multimedia Technology of the Technical University of Berlin (TUB). It provides independent network tests for many companies, consulting and training for its clients.

There were invite 28 vendors of P2P filtering products to participate in an evaluation from April to October 2007. This study was published later in March 2008 and it is available at [58]. Some of the invited were Allot Communications, Cisco Systems Inc., Arbor/Ellacoya Networks Inc., F5 Networks Inc., Huawei Technologies Co. Ltd., Narus Inc., Sandvine Inc., Packeteer Inc., Juniter Networks Inc., as well as a host of lesser known startups. From all of those, only five agreed to take part in this study but only under the condition that if their results were not those which they expected, they could withdraw at any moment without being included in the report. At the end, three of the five participating vendors decided not to include their results on the report ...

The only two vendors that agreed with publication where Arbor/Ellacoya, based in the U.S.A., and ipoque GmbH, a German vendor. The hardware they used was respectively Arbor eSeries e30 and Ipoque PRX-5G. These tests also included a network performance evaluation which was not related to P2P traffic and, therefore, will not be detailed in this work.

Efficiency of P2P Protocol Detection

To verify the P2P protocols detection accuracy, there were used thirteen different P2P clients using a total of ten different protocols. For each of the major P2P protocols (BitTorrent, eDonkey, and Gnutella), two different clients were used, since there might be some slightly differences in protocol implementations for each client (like it will be shown in chapter 4. To reproduce the actual conditions in which the hardware would mostly run at the costumer location, there was also included Web sessions, video streams, file transfer, emails and other applications along with the P2P traffic. The results achieved are listed in the following table.

P2P Protocol	**Arbor eSeries e30**	**Ipoque PRX-5G**
BitTorrent	82%	97%
eDonkey	97%	88%
Gnutella	76%	96%
FastTrack	1%	97%
MP2P	86%	96%
iMesh	0%	47%
FileTopia	33%	23%
WinMX	7%	0%
SoulSeek	1%	5%
DirectConnect	77%	78%

Table 2.9: Unencrypted P2P Protocol Detection Efficiency.
Adapted from [58].

P2P Protocol Regulation

It was performed another test to measure the capacity of this hardware to limit the bandwidth used by P2P applications, by 25%, 50% and 75% of their transmitted bandwidth. Table 2.10 shows the P2P protocol regulation efficiency for 25% and 75% of the bandwidth limit.

	25%		**75%**	
	Arbor eSeries e30	ipoque PRX-5G	Arbor eSeries e30	ipoque PRX-5G
BitTorrent	88%	88%	90%	100%
eDonkey	36%	63%	40%	67%
Gnutella	83%	93%	57%	63%
FastTrack	27%	91%	97%	78%
MP2P	93%	92%	92%	93%
iMesh	0%	43%	0%	97%
FileTopia	32%	16%	85%	4%
WinMX	19%	0%	0%	0%
SoulSeek	0%	0%	0%	2%
DirectConnect	12%	63%	24%	58%

Table 2.10: Unencrypted P2P Protocol Regulation Efficiency
Adapted from [58].

It is possible to see, from both tables 2.9 and 2.10, that the most popular P2P protocols are those that are most detected and consequently, better regulated. This is due to the amount of effort applied in the study of those protocols, since their traffic share is much larger than that of all the other P2P protocols combined.

Encrypted P2P Protocol Detection Efficiency

To conclude this study about the current state of P2P detection, it will be shown another test, included in the same study as the previous ones, to evaluate the amount of detected encrypted/obfuscated P2P traffic. "Both vendors explained that their detection of encrypted protocols did not actually employ a mechanism to break the encryption in the various protocols, but found a way to detect the traffic and/or bit pattern created." [59].

The P2P protocols tested with active obfuscation features were eDonkey Plain-Header encryption (clear-text data, header encryption only), BitTorrent Plain-Header encryption (clear-text data, header encryption only), BitTorrent Full-Stream encryption (RC4 header and data encryption), Filetopia Full-Stream encryption (AES header and data encryption) and Freenet Full-Stream encryption (AES header and data encryption).

As one can see in figure 2.23, although it is possible to detect some share of encrypted P2P traffic, in this test both eDonkey and DirectConnect came out "undefeated", suggesting that there is still an opportunity to explore this matter, either using DPI, behavior-based methods, or any other method or combination between them.

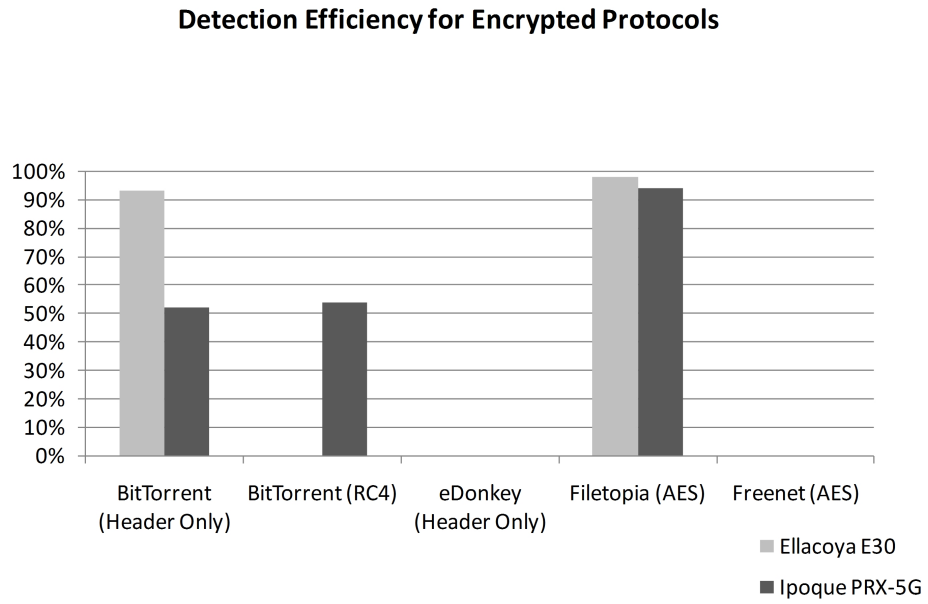

Figure 2.23: Detection Efficiency for Encrypted Potocols
Adapted from [60].

Chapter 3

Experimental Testbed

3.1 Introduction

This chapter is dedicated to the description of the lab environment. It will be detailed the network setup, the hardware and the software that were used and their configurations, since detection results can depend on their settings. Finally, the traffic classifier and the software to store, generate and visualize its reports will be described, being NIDS Snort, MySQL, Barnyard and BASE, respectively used for this purpose.

This chapter is organized in seven sections. Section 3.2 describes the physical characteristics of the laboratory where this work took place and its logical network connections as well. All the hardware used in this work is displayed in section 3.3, which also contains information about the operating systems and P2P applications they run. Section 3.4 describes the necessary network configurations that were necessary to allow P2P traffic and its interception so that it could be analyzed. These include both Microsoft Windows XP® [61] and iptables [54] firewall settings and specific traffic forwarding mechanisms. The DPI software and all the others that interact with it are described in 3.5. Snort [4] and Barnyard [62] are particularly detailed as they provide the most important tools for this work. The two final sections 3.6 and 3.7 are respectively dedicated to the description of the P2P File Sharing protocols and applications and the P2P TV applications that were used.

3.2 Lab of the Network and Multimedia Computing Group

The laboratory of the Network and Multimedia Computer Group (NMCG) [63] laboratory is part of the Department of Computer Science of University of Beira Interior. Almost all of this work was conducted in this laboratory (mainly by remotely connecting to the systems stationed there), as its facilities provide the requirements for projects of this nature, involving several network resources.

For many teachers and students, an internet connection is enough for most of their work and research. However, in cases such as this particular work, it may be necessary to allow specific incoming and outgoing traffic. Already expecting these needs, its traffic is separated from other labs and classrooms with its own *VLAN*, to guarantee a minimum

impact on performance and security, since only traffic from and to the lab circulates in its network.

All outgoing and incoming traffic for servers, workstations and laptops used at this lab, is controlled by a computer running Smoothwall Express 3.0. It is a network administration specific Linux distribution, from SmoothWall Open Source Project [64], a branch of the commercial company Smoothwall [65], which provides Internet Security and Web Filtering products. Although the SmoothWall Express 3.0 version has not the same capabilities as the commercial products, there is a huge community of developers and users, who provide support and additional plugins through internet fora such as the official one reported in [64]. This enables powerful extended possibilities at a near zero cost, wich was the main reason for its choice during the NMCG lab planning and deployment.

This Lab has twenty four 8 Position 8 Contact (8P8C) sockets, connecting to an Enterasys C2H128-48 switch through UTP Ethernet Enhanced Cat5 cabling. The switch then connects to the network backbone of the Department of Computer Science building de Informatica building, an Enterasys E7 just one floor above, via an optical fibre uplink, which in turn, connects to the rest of University of Beira Interior (UBI) through Center of Computer Science (CI). All external communication with UBI is made through an Enterasys SSR main router, located at CI. Figure 3.1 shows the experimental testbed at NMCG laboratory.

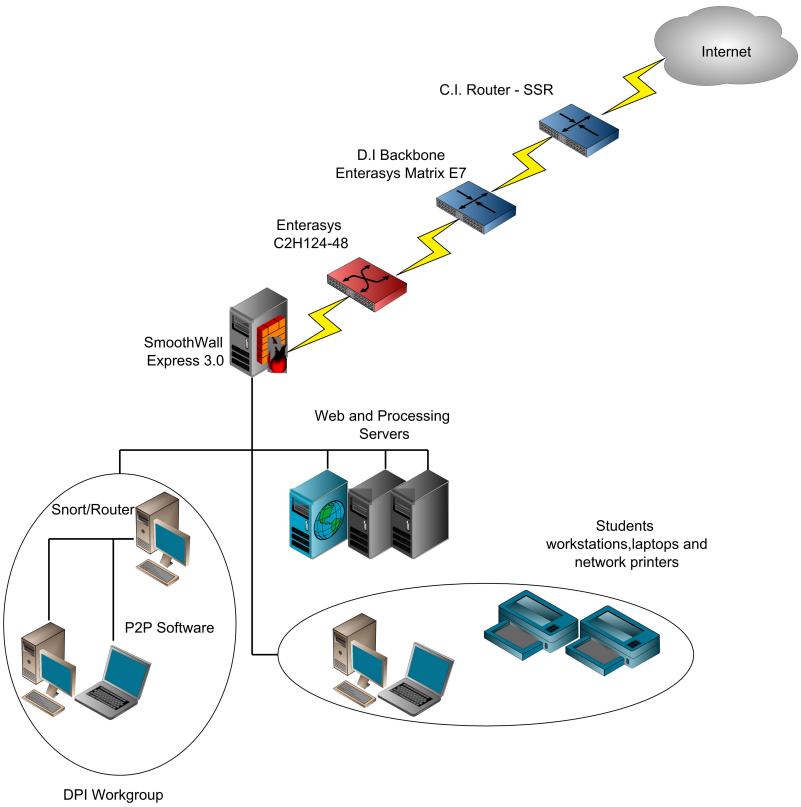

Figure 3.1: Experimental testbed at NMCG laboratory.

Most of the data and results were collected in the NMCG laboratory. However, an Internet connection through the Portuguese ISP *Cabovisão* was also used to collect and compare protocol and application signatures. During this work, there were not any visible restrictions to both connectivity and download/upload speed using any of these two kind of connections.

3.3 Hardware

To run P2P software, it is not usually necessary a great computing power. Usually, the most important feature is the size of the hard disk. When dealing with P2P file sharing programs, transfered files can easily reach a few gigabytes, since they are mostly movies, videos, music albums, games, etc. Real time network monitoring requires a lot more of memory and CPU. Therefore there were used more recent machines for the most critical applications, like the traffic classifier Snort [4], or the analysis engine BASE [66] or even the packet analyzer Wireshark [67]. As for running P2P software, pretty old machines were used, since they were mainly used for this purpose. The main characteristics of the hardware used in this work are listed in table 3.1, as well as their software information.

Type	**Operating System**	**CPU**	**RAM**	**Software**
Workstation	Fedora 9	Core 2 Duo 2.66GHz	1 GB	Snort Wireshark BASE Barnyard Gtk-Gnutella Livestation
Workstation	XP SP3	Pentium III 800MHz	512 MB	BitTorrent Vuze eMule aMule Limewire Livestation TVU Player Goalbit
Laptop	Vista Sp1;Fedora 9	Core 2 Duo 2.4GHz	3 GB	Wireshark eMule TVUPlayer Livestation Goalbit
Laptop	Mac OS X (10.5)	PowerPC G4 1GHz	768 MB	Vuze Livestation TVUPlayer

Table 3.1: Characteristics of the Hardware Used and Their Software Installations.

3.4 Network Configurations

To guarantee that all incoming and outgoing traffic generated by P2P applications in the NMCG laboratory could be analyzed, it was necessary to change some network configurations for the workstations and laptops where they were running. These machines constitute the Deep Packet Inspection Workgroup (DPI Workgroup), shown in figure 3.1. The main configurations were:

- Opening of specific TCP and UDP ports in firewalls;
- Traffic forwarding through Network Address Translation (NAT).

3.4.1 Firewalls

The use of firewalls is widespread and it is most likely that all internet users have them installed and minimally configured. Many available files in P2P networks have viruses, trojans and other malicious software, so one can assume that most users are cautious enough to protect their machines and data. Therefore, all the machines in the DPI Workgroup, regardless their operating system or purpose, also had their firewalls on, to replicate the conditions in which most P2P users will find themselves.

Most of the P2P file sharing installation programs created random communications ports, instead of the well known ports for a given protocol. The purpose of this feature is to avoid their detection, but it only works when a simple port based traffic classifier is being used, unlike some recent software firewalls, like the previously mentioned WFilter in 2.5.3, which already include DPI features. The fixed ports used by the tested applications for incoming traffic are listed bellow in table 3.2.

Application	**Port**	
	TCP	UDP
BitTorrent	17785	17785
Vuze	60249	60249
Gtk-Gnutella	10293	10293
Limewire	-	28793
eMule	35872	7075
aMule	4662	4672
Livestation	80	
TVUPlayer	80 3950	80 3902
Goalbit	2706	-

Table 3.2: P2P Application Ports.

Most of this software was running in windows operating systems and the first time each of this applications started, one of the following options had to be selected:

1. Unblock this program, despite the security risk

2. Keep blocking this program

3. Keep blocking this program, but ask me again later

Obviously, option number 1 was always selected, allowing from that moment on, the windows firewall to accept communication ports opened by the software that triggered the event. The only ports which were necessary to open manually, refer to aMule and eMule, in windows operating systems, and Gtk-Gnutella in linux. These are listed in table 3.2.

Figure 3.2 shows a simple Microsoft Windows XP® Service Pack 3 firewall configuration for eMule. It is important to highlight, that the scope option was not important in this case, since the traffic that arrived at this machine, with a private IP address, had been be previously filtered.

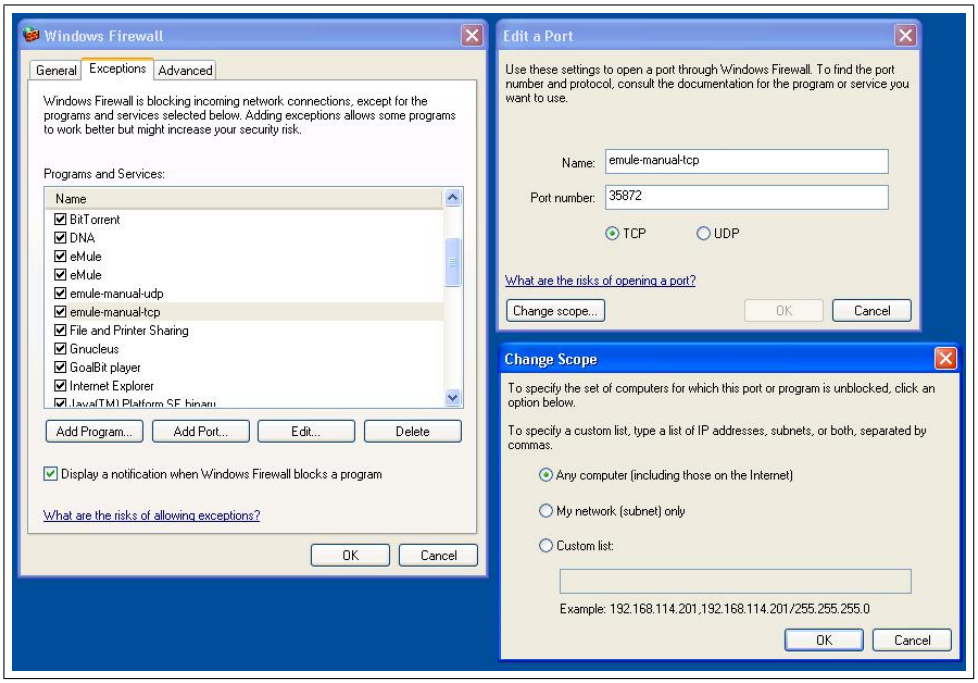

Figure 3.2: Microsoft Windows XP® firewall configuration for allowing eMule TCP traffic. Screenshot taken from a Microsoft Windows XP® [61] workstation.

As for Gtk-Gnutella, two simple *iptables* [54] rules were created. Iptables is part of an open source packet filtering framework, in linux 2.4.x and 2.6.x kernels. Previous versions were *ipchains* and *ipfwadm* for linux kernels 2.2.x and 2.0.x respectively. The rules were added into */etc/sysconfig/iptables*, the main firewall configuration file in Fedora 9 Linux, in order to allow or deny network traffic. The first one is for TCP and the second for UDP traffic.

1. `-A INPUT -m state -state NEW -m tcp -p tcp -dport 10293 -j ACCEPT`

2. `-A INPUT -m state -state NEW -m udp -p udp -dport 10293 -j ACCEPT`

3.4.2 Traffic Forwarding

The reason for using traffic forwarding, was to enable that all P2P traffic in the DPI Workgroup could be routed through the Snort classifier so it could be analyzed. To accomplish that, it was necessary to set the *default gateway* [10] on the machines where the P2P software was running to the IP address of the Snort classifier. This gateway was running Fedora 9 Linux and all the firewall rules and traffic redirection was done by using iptables.

After setting the *default gateway* value for all the machines running P2P applications in the DPI Workgroup, the first thing to be done was to forward their communications through the Snort system, which now was also set as their router. This was done by using a simple iptables rule, that *masquerades* the traffic originated from internal machines to outside of their network. This is accomplished by changing the source IP address to that of the router and, when a response to that traffic arrives, iptables can redirect it correctly by maintaining a special table of original addresses and ports being used. This is called the Network Address Translation table (NAT). The commands for masquerading two of the used machines running P2P applications, with IP addresses 10.0.5.5 and 10.0.5.114 were respectively (1) and (2):

1. `iptables -t nat -A POSTROUTING -s 10.0.5.5 -j MASQUERADE`
2. `iptables -t nat -A POSTROUTING -s 10.0.5.114 -j MASQUERADE`

NAT was also setup to redirect incoming traffic, again through the machine were Snort was installed, so it could reach the pretended P2P applications, whether if it was a response or a request to them. So after the firewalls have been opened for this, more iptables rules were added to allow communications to get to their final destination. In the following excerpt, the IP addresses 10.0.5.5 and 10.0.5.6, refer respectively, to a P2P application system and the Snort classifier.

- `iptables -t nat -A PREROUTING -d 10.0.5.6 -p tcp -dport 35872 -j DNAT -to 10.0.5.5:35872 #eMule`
- `iptables -t nat -A PREROUTING -d 10.0.5.6 -p udp -dport 7075 -j DNAT -to 10.0.5.5:7075 #eMule`
- `iptables -t nat -A PREROUTING -d 10.0.5.6 -p tcp -dport 4662 -j DNAT -to 10.0.5.5:4662 #aMule`
- `iptables -t nat -A PREROUTING -d 10.0.5.6 -p udp -dport 4672 -j DNAT -to 10.0.5.5:4672 #aMule`

NAT played another important role in allowing external access to the DPI Workgroup from a specified location. This was particularly useful during this work, since it allowed

[10] A standard network parameter, to indicate the IP address of the device used to route traffic outside of the local network.

to avoid almost any physical presence in the Lab for a given task. The Smoothwall firewall can have several external IP addressess, which, combined with ports defined by the network administrator, can be used to forward specific traffic. An example of this, was when accessing the Linux Snort Classifier, in a private network, through a Secure Shell (SSH) application. Here, a Web interface was used to access Smoothwall via HTTPS, that automatically generated the apropriate iptables rule.

Figure 3.3 shows part of the SmoothWall firewall and port forward configurations. One can see in the *Port and protocol forwarding section* that incoming traffic towards IP address 193.136.67.242 and TCP port 50002 is to be forward to IP 10.0.5.6 and port 22, to enable SSH access.

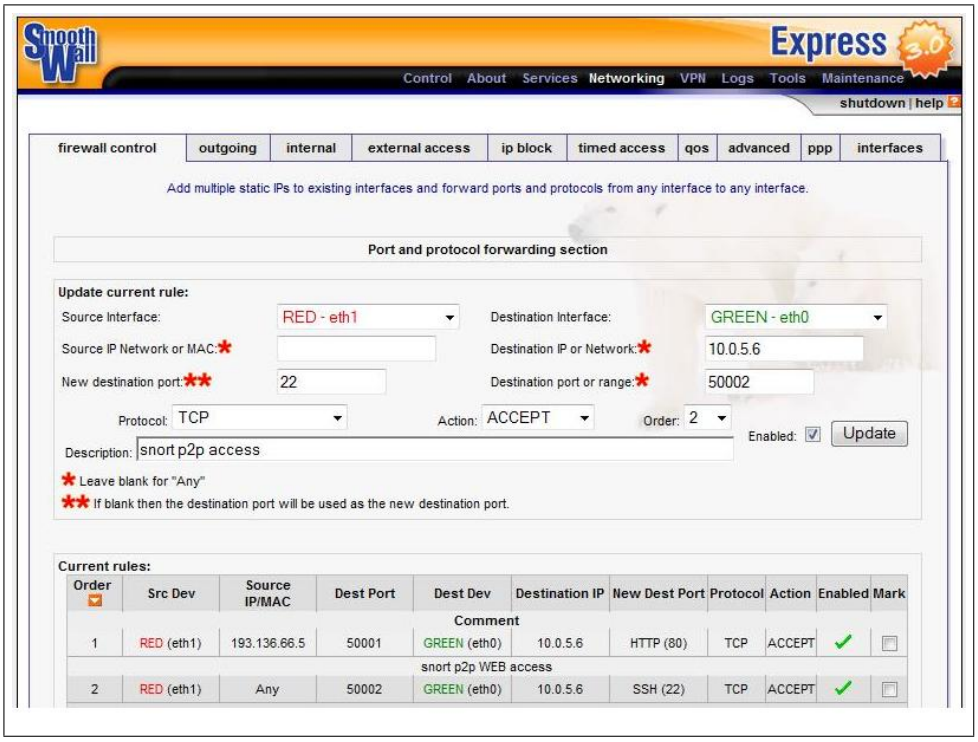

Figure 3.3: Smoothwall NAT example configuration.
Screenshot taken from SmoothWall Express 3.0 [64].

Remote Desktop Connection (RDC) to a Windows XP® system at the NMCG laboratory was another example of traffic forwarding into the private network. This was just a little bit more complex to achieve than in the previous case, because the *default gateway* on these machines was set not to the Smoothwall Express, but to the machine running the Snort Classifier [11]. So instead of forwarding traffic once, an extra step had to be done. The first one, similar to the shown in figure 3.3, but with the destination port set to TCP 3389, the default RDC port. In the second stage, incoming TCP traffic to port 3389 on the Snort classifier was forwarded to its final destination - The actual Windows XP® workstation. This

[11] The complete NMCG network schema is shown in figure 3.1, on page 40.

was accomplished by the following iptables rule, where the Snort IP address is 10.0.5.6 and one of the windows workstations is 10.0.5.5 :

```
iptables -t nat -A PREROUTING -d 10.0.5.6 -p tcp \\
--dport 3389 -j DNAT --to 10.0.5.5:3389
```

3.5 DPI and Network Software

This section is devoted to the applications involved in traffic capture and analysis and alert classification and storage. All of them are widely used open source software distributed under the GNU General Public License [68] and have a vast support community and constant developing. These were the main reasons for their choice, along with the fact that they have proven through the years to be a stable and reliable technology for projects with an identical or superior dimension than this one.

3.5.1 Snort

Snort was created by Martin Roesch in 1998, as a lightweight Network Intrusion Detection System (NIDS), comparatively to existing commercial solutions at that time. Over the years it evolved into a more feature rich technology, becoming the most popular open source NIDS. The Snort architecture [4] consists of the following components, represented in figure 3.4.

- Packet Decoder
- Preprocessors
- Detection Engine
- Logging and Alerting System
- Output Modules

Its operation can be briefly resumed as follows: Basically, Snort is a packet sniffer. However, it can also process incoming packets that match some previously specified criteria. The Snort Packet Decoder first performs all the work to prepare the data for the detection engine. It supports the Ethernet, SLIP and PPP mediums. This data is then sent to the Preprocessors, which verify if a packet should be analyzed. If this is the case, those packets are then checked against a set of rules using the detection engine. When a rule applies to a packet, then an output will be generated through the configured output modules.

The detection engine is at the heart of Snort. It is responsible for analyzing every packet based on the Snort rules that are loaded at runtime. The detection engine separates the Snort rules into what is referred to as a chain header and chain options. The common attributes such as source/destination IP address and ports identify the chain header. The chain options are defined by details such as the TCP flags, ICMP code types, specific type of content, payload size, etc. The detection engine recursively analyzes each and every packet based

on the rules defined in the Snort rules file. Any rule that matches the decoded packet, triggers the action specified in the rule definition. A packet that does not match any Snort rule is simply ignored by the engine and forward towards its initial destination.

Logging and alerting are two separate subcomponents. Logging allows you to log the information collected by the packet decoder in human readable or *tcpdump* format. One can configure alerts to be sent to a file or a database.

The Output Modules enable Snort logs and alerts to written in plain text files, systems logs, database formats like MySQL, Postgresql, ODBC, MS SQL Server or ORACLE, or even the unified(binary) format to be used by Barnyard, described in 3.5.2.

Figure 3.4 shows how the Snort components work together.

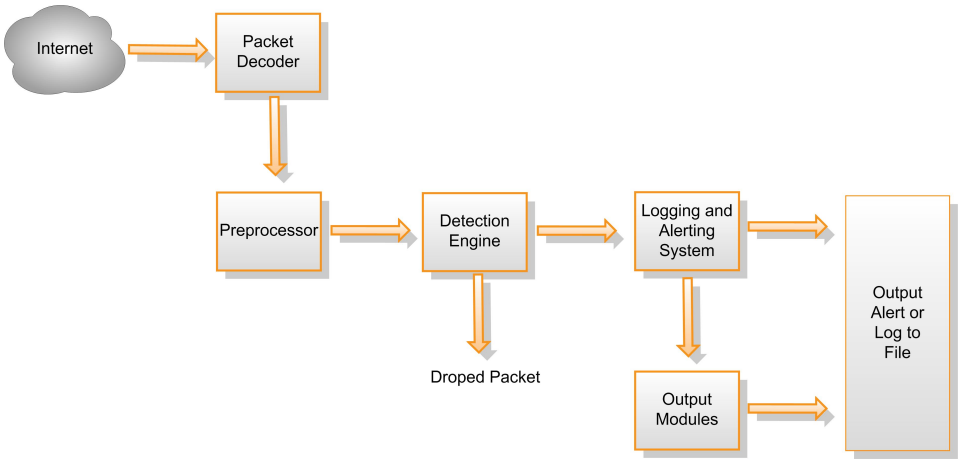

Figure 3.4: Snort Architecture.
Adapted from [69].

Installation and Configuration

Snort-2.8.3.1-1.i386 was built from the source code available for download at [4], after extracting it as a regular *TarBall* [12]. Then, it is just necessary to compile it, assuming that all library dependencies to make it work with other software are already satisfied. Usually, when integration with a MySQL Database is wanted, just like in this particular work, it is just necessary to execute the following commands in the extracted source code folder.

1. ./configure –with-mysql
2. make
3. make install

Snort installed its executable, libraries, manuals and configuration resources under */usr/sbin/, /usr/lib/snort/, /usr/share/man/man8/ and /etc/snort/*, respectively. After integrating Snort

[12] A TarBall is a very common software distribution format, in which a single Tape Archive(TAR) file is created from a file or sets of files and then compressed with Gzip or Bzip.

with the Fedora services interface, using Fedora command line configuration tool *chkconfig*, operating it was just a matter of executing *service snortd [command]* with administrative privileges, where *command* was mainly start, stop or restart.

The main configuration file is snort.conf. It is a text file with a pretty easy to read syntax, were the following settings can be made it its distinct sections:

1. Set the variables for your network
2. Configure dynamic loaded libraries
3. Configure preprocessors
4. Configure output plugins
5. Add any runtime config directives
6. Customize the rule set

In section 1 of this file, the **var HOME_NET [10.0.5.0/24]** and **var EXTERNAL_NET !$HOME_NET** were set. This tells Snort that the local network is 10.0.5.0/24 and the external network is everything that is not internal.

Another configuration made to this file was into the HTTP preprocessor, in its section 3. This necessity arose after noticing that some expected alerts [13] were not triggered by Snort. The reason for this was that the expected strings that would trigger the alert, had not a fixed position in the packet payload. It was necessary to alter the preprocessor definitions so that, for testing purposes, the entire payload would be analyzed. This was done by the following configurations:

```
preprocessor http_inspect_server:  server default profile all ports
{ 80 8080 8180 } oversize_dir_length 300
flow_depth 1460
```

Figure 3.5: Snort HTTP Preprocessor Configuration; /etc/snort/snort.conf file.

The Snort logs and alerts are initially stored into text files, if no other configuration is done. Shortly after, they started to be written into a MySQL Database after it was installed and configured. This was achieved by the following configuration line in section 4:

```
output database:  log, mysql, user=snort password=xxxxxxx dbname=snort host=localhost
```

Figure 3.6: MySQL Logging – Snort Configuration.

Snort alerts are can be triggered by its own shipped rules or user defined ones. They are included in the *snort.conf* file in section 6. There are initially 55 files under the default rule folder in */etc/snort/rules* for Snort version 2.8.3.1. These go from virus threats to Web

[13] These alerts are specific to P2PTV application Livestation.

attacks and many more. For this work, another folder was used to separate Snort distribution ruleset from the new one. Its location was */etc/snort/rules_testing* and contained one file for each studied P2P protocol. These were include by editing the *snort.conf* file in section 6 with the following contents:

- include /etc/snort/rules_testing/p2p.gnutella.rules
- include /etc/snort/rules_testing/p2p.bittorrent.rules
- include /etc/snort/rules_testing/p2p.edonkey.rules
- include /etc/snort/rules_testing/p2p.tv.rules

Snort rules are formed by the Rule Header and Rule Options. According to [4], the Rule Header contains information about:

- Rule Actions:

  alert - generate an alert using the selected alert method, and then log the packet

  log - log the packet

  pass - ignore the packet

  activate - alert and then turn on another dynamic rule

  dynamic - remain idle until activated by an activate rule , then act as a log rule

  drop - make iptables drop the packet and log the packet

  reject - make iptables drop the packet, log it, and then send a TCP reset if the protocol is TCP or an ICMP port unreachable message if the protocol is UDP.

  sdrop - make iptables drop the packet but do not log it.

- Protocols:

  TCP

  UDP

  ICMP

  IP

- IP Addresses
- Port Numbers
- The Direction Operator:

  $>$ - source to destination

  $<>$ - bidirectional

- Activate/Dynamic Rules

As for the Rule Options, they are the heart of the Snort intrusion detection engine. They are divided in the following categories, according to [4]:

- General - These options provide information about the rule but do not have any affect during detection (examples: *msg, rev, sid* respectively for output message, rule revision id, rule internal id)
- Payload - These options all look for data inside the packet payload and can be interrelated
- Non-payload - These options look for non-payload data
- post-detection - These options are rule specific triggers that happen after a rule has "fired."

An example of a created Snort rule is listed bellow. It was extracted from */etc/snort/rules_testing/p2p.bittorrent.rules* and will be further detailed in section 4.2.1.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:P2P BitTorrent outbound
- tracker request"; flow:to_server,established; content:"GET"; offset:0; depth:4;
content:"/scrape"; distance:1; content:"info_hash="; offset:12; content:"User-Agent:";
offset:80;classtype:policy-violation; sid:1000305; rev:1;)
```

Figure 3.7: Example of a Created Snort Rule for P2P BitTorrent Tracker Request Traffic.

An important note about the *sid:* subsection under general categories of the Rules Options, is that it will be used later in this work in chapter 4 to uniquely identify Snort rules. This information allows output plugins to identify rules easily, and should be used with the *rev* keyword to specify its version (revision). It should be an integer satisfying conditions:

- $<$100; Reserved for future use
- 100-1,000,000; Rules included with the Snort distribution
- $>$ 1,000,000; Used for local rules (user defined)

In figure 3.7, *sid* as a value of 1000305, which indicates it is a user defined rule, not originally included in the snort distribution.

Snort Inline

Latest versions of Snort, including the one used for this work, allow a feature named *Inline Mode*. While Snort reads packets from libpcap, when using the Inline mode this is done via iptables. This latest has to be compiled so that the *libipq* library is installed, allowing Snort Inline to interact with iptables. After this, three types of rules can then be used in Inline mode.

- drop - Drop the packet using iptables and log it via usual Snort means.

- reject - As previously, but send a TCP reset if the protocol is TCP or an ICMP port unreachable if the protocol is UDP.
- sdrop - Drop the packet without logging it.

It is advised to run two instances of Snort if one pretends to both drop packets and generate alerts. This way, each instance runs a different rule set, distinguishing the traffic to logged and that to be dropped. Due to time limitations, these capabilities were not tested during this work, as it will be further mentioned in section 5.2.4.

The rule displayed in figure 3.8 shows an example of a drop rule, which blocks incoming traffic for HTTP servers on their well known ports, for 600 seconds, after the "root.exe" content is being the detected in the Uniform Resource Identifier (URI) field.

```
 drop tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-IIS CodeRed v2
root.exe access"; flow:to_server,established; uricontent:"/ro ot.exe"; nocase;
sticky-drop:  600,src; reference:url,www.cert.org/advisories/CA-2001-19.html;
classtype:web-application-attack; sid:1256; rev:8;)
```

Figure 3.8: Snort Inline Drop Mode Example.

Snort Inline also allows packet content replacement, provided that the new string and that to be replaced have the same length. A simple example is shown in figure 3.9

```
alert tcp any any <> any 80 (msg:  "tcp replace"; content:"GET"; replace:"BET";)
```

Figure 3.9: Snort Inline Replace Mode Example

Due to time limitations, these capabilities were not tested during this work, being left for future study.

3.5.2 Barnyard

Barnyard is a fast output system [62] for Snort, to enable it to keep up with a busy network. Snort logs, without any special configuration, are stored directly into text files, or, on a more refined environment, into one of its supported database formats shown previously in 3.5.1. When an alert or log is triggered by a Snort rule, it has to be converted to text format, since it is originally obtained through the binary format of *tcpdump* [70]. More processing is needed and eventually it may cause Snort to skip some IP packets from analysis. On a busy network, specially if the logs are stored in a database instead of text files, it could have even a greater impact, due to all the extra operations to be done until a successful table insert.

During the P2PTV traffic detection, the number of alerts reached the million for a few times, since all UDP traffic was being accounted for statistical accuracy of the created rules. At that time, although there were not detected any packets skipped by snort [14], it made sense to prevent this situation. Barnyard was the perfect solution since it can process binary logs

[14] A Snort recent feature allows it to display collected traffic statistical information, including packets being skipped.

and alerts in the background, releasing Snort of this time consuming task. There can be a little delay from the time where an alert is generated to its visualization, but never enough to compromise a real time analysis.

Barnyard was installed from source code available at [62]. Its installation and configuration was quite simple. After downloading the Barnyard *TarBall*, the following commands were run in the extracted source code folder, to compile it with MySQL support and to copy its configuration file to the proper location so that Snort could use it.

1. ./configure –enable-mysql
2. make
3. make install
4. cp /usr/local/src/barnyard-0.2.0/etc/barnyard.conf /etc/snort

Subsequently, there were added two configuration lines into barnyard.conf, to enable it to log alerts and logs into the MySQL Snort database.

- output alert_acid_db: mysql, database snort, server localhost, user snort,password xxxxxx, detail full
- output log_acid_db: mysql, database snort, server localhost, user snort,password xxxxxx, detail full

After that, Snort was easily configured, by editing snort.conf, to use Barnyard instead of logging directly (as it was been doing until mid January 2009) to the MySQL Snort database. The following changes took place:

1. supression of the configuration line "‘output database: log, mysql, user=snort password=xxxxxx dbname=snort host=localhost"’, created in 3.5.1
2. added configuration lines

   "‘output alert_unified: filename /var/log/snort/snort.alert, limit 128"’

   "‘output log_unified: filename /var/log/snort/snort.log, limit 128"’

As one can see in 2, text format logs and alerts were replaced by the binary(unified) format, stored at the default Snort log folder with a limit of 128 MB. After this limit is reached, another one will be created with a different time stamp, and so one. The final configuration, was to create and edit the *barnyard.waldo* with the following contents:

1. /var/log/snort
2. snort.log
3. 1237312691 (will vary)
4. 0

This tells the Barnyard daemon, through the barnyard.conf file were the *WALDO_FILE* was set with WALDO_FILE="/var/log/snort/barnyard.waldo", the folder were the Snort logs are, their prefix (snort.log), time stamp generated suffix (like in 3; it changes every time snort daemon restarts) and the initial value of "'0"' which tells barnyard the number of Snort alerts already processed.

Barnyard was added as a system service using Fedora command line configuration tool *chkconfig*. This way, it can be easily enabled or disabled on the machine startup or any other Linux run level [15], allowing the task of stopping, starting or restarting it to be easier using the command *system SERVICE_NAME [command]*.

Another important setting was to edit the */etc/snort/sid-msg.map*. Without it, all Snort alerts were identified by the rule ID (an integer), which it was not very practical to visualize using BASE, described in 3.5.5. Previously, their description (added by the "'msg:"' parameter within a rule) was used automatically to this purpose. To make a correspondence between the rule ID and its desired description, the sid-msg.map has to obey to the following format:

SID	MSG	Optional References	Optional References
2000357	BitTorrent Traffic	bitconjurer.org/BitTorrent/protocol.html	

Table 3.3: Snort sid-msg.map File Format.

3.5.3 Apache

Apache is an open source Web server, widely used for corporate, educational and domestic environment. It is a multi-plattform application available at [71], which origins go back to the year of 1995. It was initially based on the National Center for Supercomputing Applications (NCSA, at the University of Illinois) httpd 1.3 and the first official public release (0.6.2) was available in April 1995. Finally on December 1 1995, Apache 1.0 was released.

Apache makes part of the Fedora installation media and was installed along with the operating system. It was kept quite simple, so no Apache configurations were needed for itself during this work, since its goal was only to serve a single web site for BASE [66], which purpose and configurations settings will be described in section 3.5.5.

3.5.4 MySQL

MySQL is a popular open source Relational DataBase Management System (RDBMS). It is a *Cross-Platform* [16] software available at [72], with its initial release in the distant year of 1995. MySQL is owned by the swedish company MySQL AB, a subsidiary of the american giant Sun Microsystems.

[15] Linux run levels are identified by integers from 0 through 6. The most used are 1 for single user, 3 for network with multiuser support without graphical login and 5 for full network multiuser mode.

[16] Cross-Platform software is is one that can be compiled to run on multiple computer platforms.

The Fedora installation media includes MySQL and many other related packages, to provide inter-operability with a vast number of services. An example of this is *php-mysql*, which provides files and libraries necessary for *PHP* to use a MySQL database. Version 5.0.51 was installed from RPMs along with other related MySQL packages.

Its configuration was kept minimal for this work. The Snort database was created with the provided */usr/share/snort-2.8.3.1/schemas/create_mysql* script, which besides creating the initial 16 database tables, also inserted initial data into them, thus enabling immediate Snort operations.

Sometimes, depending on the Snort processed traffic volume, the database could easily reach hundreds of Megabytes and, for once or twice, this value even reached the Gigabytes order. This had a serious impact on logs and alerts visualization, since hundreds of thousands of table rows had to be read, arranged and then displayed in a web interface. To avoid this, after a few runs by Snort or if some pretended statistics or results have been collected, the database table rows could be easily removed by two ways:

- Manually

  Using "'delete * from *tablename*"'

  Using an available graphical interface like *MySQL Administrator* or *MySQL Query Browser*

- Using subsequently detailed BASE web interface itself [17], selecting the option *Cache & Status, Clear Data Tables*

Either way, none of these procedures affected later analysis, but improved by far the performance of the visualization process.

3.5.5 BASE

BASE stands for Basic Analysis and Security Engine. It is an open source software that enables to visualize Snort logs and alerts , in a more user friendly way, using a web browser as the interface. It collects data from Snort MySQL database and it allows to perform administrative tasks on its specific tables and those of Snort.

BASE installation was quite simple and its configuration minimal. Although it can be obtained at [4], under the contributions and data analysis section, version 1.4.1 was installed from an RPM from *http://rpm.pbone.net*. The reason for this was to minimize the configurations necessary for it to work, and to guarantee the maximum integration possible with the rest of its related software, which was also mostly installed from RPMs.

Its configuration file was automatically copied to */etc/httpd/conf.d/*, the default folder in Fedora for apache addons and contains only an *alias* [18] to its filesystem location and default configurations for web access. The user configuration process itself started on the first web access to the address *http://localhost/base*, where Snort was also installed and it was a straight through process. It was just necessary to provide some Snort and MySQL

[17] Examples of BASE Web browser interfaces are shown in figure 3.10.

[18] An Apache alias is a setting that allows a name used in a browser URL to be redirected to another location.

configuration details, after which six additional tables where created in the Snort database schema, providing the visualization functionalities using a simple Web browser.

Figures 3.10 and 3.11 are just one example of the BASE interfaces for the logs and alerts generated by Snort, after being processed by Barnyard.

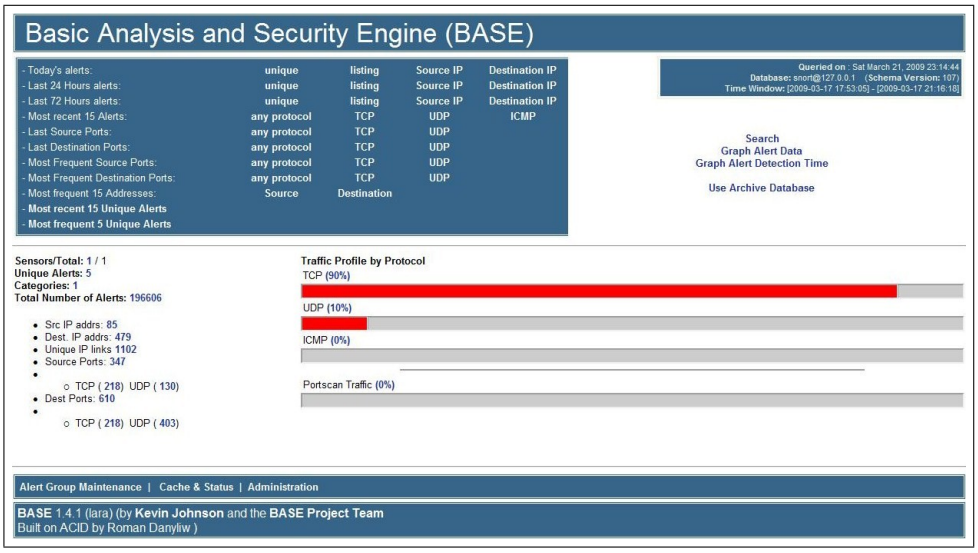

Figure 3.10: BASE Main Interface.
Screenshot taken from BASE [66] main interface.

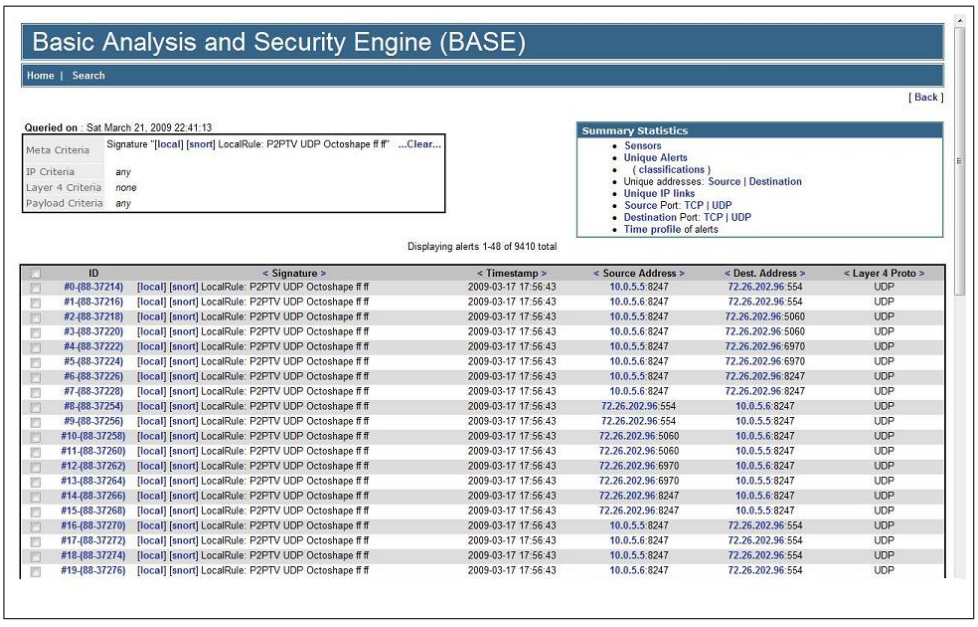

Figure 3.11: BASE Alert Selection.
Screenshot of a specifc BASE [66] Snort alert.

3.5.6 Wireshark

Wireshark is perhaps the most well known network protocol analyzer and it is the successor of Ethereal, whose origins date back to 1998. It has a large community of developers and contributors (about 609) and supports 935 network protocols. It is commonly used in industry and educational institutions and some its main features are [67]:

- Live capture and offline analysis
- Deep inspection of hundreds of protocols
- Standard three-pane packet browser
- Multi-platform: Runs on Windows, Linux, OS X, Solaris, FreeBSD, NetBSD, and many others
- Captured network data can be browsed via a GUI, or via the TTY-mode TShark utility
- Read/write many different capture file formats: tcpdump (libpcap), Pcap NG, Catapult DCT2000, Cisco Secure IDS iplog, etc
- Coloring rules can be applied to the packet list for quick, intuitive analysis
- Output can be exported to XML, PostScript, CSV, or plain text

This application was used in Windows (version 1.0.4), Linux (version 1.0.5) and even OS X from Apple (version 0.99.6), as a support tool to analyze and identify pretended traffic. Its installation on every of the above operating systems was quite simple. For windows, it is just a matter of downloading and executing the installer, available at [67]. Wireshark makes part of the many Linux distributions, so in case it is not automatically included during the installation of the system, one just has to use the proper packet manager to make it available for use. As for OS X, Wireshark was installed through *darwin ports*, a very complete and automated command line software management package. It run over *X11* [19], almost exactly the same way as in windows or Linux.

For most of the times, Wireshark run on the Snort classifier itself, because all traffic in the DPI Workgroup was routed by it. To not overload Snort, since it was running Barnyard to process its logs and alerts, and also a MySQL database and accepting external SSH connections, traffic was mostly captured through *tcpdump* in a linux shell. This way, the capture task run in background, saving the output to a binary file, which Wireshark could import later so the traffic could be analyzed. It can be very useful, when one intends to capture or display a specified protocol, port or traffic direction, or even perform a search in *ASCII* or Hexadecimal format inside a packet payload. Figure 3.12 shows a screen from Wireshark, where a filter was applied to display only HTTP traffic.

[19] X11 is an open source implementation of the X Window System.

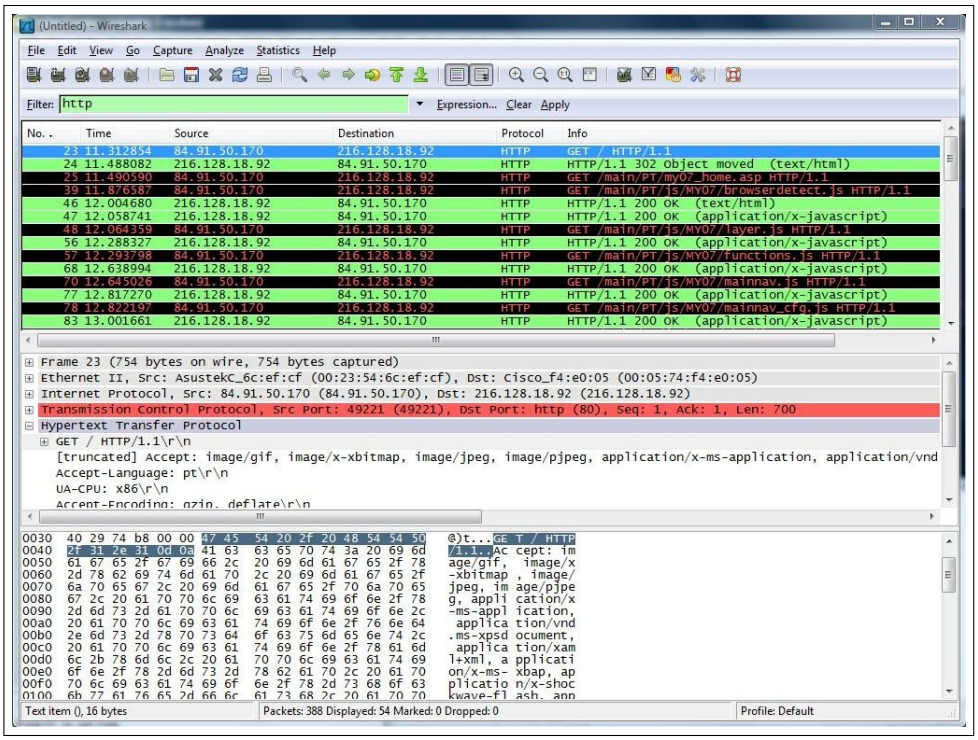

Figure 3.12: Wireshark filter for HTTP protocol.
Screenshot taken from the Wireshark [67] application.

3.6 P2P File Sharing Protocols and Applications

The choice for the P2P software and its respective operating system, were mainly influenced by its worldwide popularity, resource availability and ability to use encryption or obfuscation, since not all client software allows them. These are two different methods that programmers use to avoid Traffic Shaping or bloking. While encryption is a two-way data transformation (encrypt/decrypt) by applying a cryptographic algorithm, thus providing strong protection, obfuscation is a one-way transformation process. It can be achieved, for example, by changing the order a well known data structure, or generating some extra information to "'confuse"' possible interceptors. Any of them is quite successful when trying to achieve stealthiness using P2P applications, like it will be shown in the next chapter.

For each studied protocol, there were tested at least two applications listed in table 1.1, in page 3 and their data was collected in the server were the Snort sensor was running, which also acted as the default gateway for computers running P2P software. This was done to guarantee that all traffic generated by these applications passed through the sensor, so that it could be analyzed.

3.6.1 BitTorrent Protocol

The BitTorrent protocol [18] belongs to the Unstructured, Hybrid Decentralized, Tracker based architecture. It is perhaps the most widely used P2P protocol, specially when it comes to downloading large files. It uses a feature named *tracker*, which is a server that assists the communication between peers using the BitTorrent protocol. It is also, in the absence of extensions to the original protocol, the only major critical point, as clients are required to communicate with the tracker to initiate downloads. Clients that have already begun downloading also communicate with the tracker periodically to negotiate with newer peers and provide statistics; however, after the initial reception of peer data, peer communication can continue without a tracker.

One feature that allows BitTorrent to be so efficient for downloading large files is *swarming*. The concept behind it is that bandwidth usage is not optimized. Each computer has unused, excess uploading bandwidth even when they are busy downloading. BitTorrent works by breaking big files into many smaller files. When a file is available for download, each user interested in it starts to download a different part of the file. As soon as "chunk" is completed, it starts to automatically be uploaded for others to download. Eventually everyone gets all of the parts of the file and this is the reason why BitTorrent works so well for large downloads, even being recommended by some open source Linux operating system distributions, for example.

Nowadays, trackerless communications are possible by using decentralized overlay networks such as DHT. BitTorrent uses DHT to find resources without the dependency of central servers. Those DHT tables may have information about peers, relative distance, hash of a given file part (chunk).

Most BitTorrent clients, such as BitTorrent itself, also use Peer exchange (PEX). This provides another method to gather peer information, in addition to trackers and DHT. Peer exchange checks with known peers to see if they know of any other peers, improving the network fault-tolerance capability.

BitTorrent application

A popular implementation of the BitTorrent protocol is the BitTorrent application available at [73]. This is the original implementation of the protocol, and it is often called "'Mainline"' for this reason. Originally, it was an Open Source software written in *Python*, available for Windows, Linux and OSX from Apple. However, since versions 6.x, it has been based on μTorrent, written in C++ and available only for computers running Windows operating systems. It enables encryption, which is another reason for its choice during this work. Users can also create their own *.torrent* files, which enables them to publish their own content.

Recently, a new feature became available and it is called BitTorrent DNA. It is a service that enables acceleration for downloads and streams from Content Delivery Networks (CDNs) and is distributed along the freeware BitTorrent client, or can be downloaded separately and might be included in other popular downloaded applications and content. An example of this is becoming popular within the Gaming Industry, where the software may

use DNA to obtain game updates. "Whenever DNA is bundled with an application, the installation process explains DNA and its operation." [73]

Vuze Application

Another studied BitTorrent application was *Vuze* [43], formerly know as *Azureus*. It is Java application that can be installed in Windows, Linux or OS X from Apple. This is one of the most popular BitTorrent clients nowadays, providing stealth capabilities like proxying, tunneling and encryption. Although it has a very intuitive interface, it allows advanced users to access an expert mode, in which they can enable more complex settings. Vuze enables separate channel searching for Music, Video and Games, which quickly allow content search in its own network, even for unexperienced users. Recent versions allow to include popular torrent sites in the search request, like *btjunkie, jamendo, mininova, etc*. This search list can even be updated by the user. Just like the BitTorrent application, users can create their own *.torrent* files. Vuze was the first BitTorrent client to implement DHT.

3.6.2 eDonkey

eDonkey is a Hub Based, Hybrid Decentralized P2P network. It was created by the Meta-Machine Corporation in the year 2000 and achieved popularity mainly in Europe.

This network resides on both clients and servers to get the best of centralized and decentralized architectures. Centralized ones such as Napster, had already showed its weakness by depending on a single or a few central servers to index the information. This results in low fault tolerance and easy to achieve network shutdowns when legal actions are taken, like it happened in 2001 with Napster. With the Decentralized architecture, used for example by the Gnutella protocol, this problem does not occur anymore, since it is a pure P2P decentralized network where central servers are inexistent. Nevertheless, this architecture as still some issues, mostly concerning the enormous ammount of traffic between the peers generated by search requests. Using the Hybrid Decentralized architecture, eDonkey still relies on central servers to ensure better search mechanisms, but these are widely spread across the Internet and thus provide high fault tolerance.

Hashing mechanisms using MD4, are used so that search results are improved comparatively to simple name search. Files are split into 9500 KB "chunks" each with a 128 bit hash, which allows swarming (like BitTorrent) besides improving search accuracy.

eDonkey2000 was the original client software for this P2P network, but it became unavailable in September 2005, after receiving a cease and desist order by the Recording Industry Association of America (RIAA). Currently its website [17] shows only the following message:

> *"The eDonkey2000 Network is no longer available [...] Your IP address is xxx.xxx.xxx.xxx and has been logged. Respect the music, download legally."*

Nevertheless, the eDonkey network is still up by using other clients such as eMule, aMule, Shareaza or MLDonkey just to cite a few. Maybe the only difficulty is to obtain an updated eDonkey Server List (some are available at [74]), after which connections to servers will be available and therefore, to the eDonkey network.

eMule

One of the most successful P2P applications is eMule [74], launched in 2002 for Windows® operating systems and programmed using C++. It supports eDonkey and, since versions v0.40, the structured decentralized KAD network. This allows eMule to reduce its server dependency by providing mechanisms for direct search between peers.

Since version 0.47b, eMule provides protocol obfuscation, which was the main reason for its choice during this work. Although eMule is one of the most used eDonkey clients, there are nowadays many others forked from the initial project, such as StulleMule, Xtreme and Neomule, just to cite a few. This late one was even tested during this work, but no data was collected with it.

aMule

aMule is a another well known eDonkey client available for several platforms at [75]. It was initially based on the xMule source code, which in turn was based on the lMule project, which was the first attempt to create an eMule like client to Linux systems. Currently it shares code with eMule Project, so the features are quite similar between them, being the most notorious the graphical user interface.

aMule can be compiled to be run in a modular way, so that its main functionalities can be started as a daemon and the other features can be set in one of the following interfaces:

- aMuleCMD - Command-line client
- aMuleGUI - The usual graphical interface
- aMuleWEB - Web interface through a built-in Webserver

Just like eMule, aMule also provides protocol obfuscation, which makes it very intended for many P2P users.

3.6.3 Gnutella

Gnutella version 0.6, is a Hybrid Decentralized, Unstructured architecture based in Super Nodes (Ultrapeers), unlike its predecessor version 0.4, which was Purely Decentralized P2P network. In the latest architecture (see figure 2.3), searches generate too much traffic between peers and their results might not be very accurate, as all the peers have the same status in the network and therefore, no dedicated indexing servers exist. When using the Hybrid Decentralized architecture based in Super Nodes (as shown in figure 2.4), scalability is improved as special nodes or peers are introduced into the network, providing indexing and caching features that allow better search performance and results. This is the main reason why most Gnutella clients nowadays use this architecture. Any user with a fast Internet connection and some free disk space, can contribute to the improvement of the network by becoming a Super Node. This can be done very easily by simply selecting the intended application mode in the GUI configuration, which is generally *leaf mode* of *Super Node*.

For studying Gnutella version 0.6 traffic, it was used LimeWire 4.18.8 in Windows and GTK-Gnutella 0.96.5 in Linux. The choice for these two applications was mainly influenced by their popularity and consequently resource availability and, most importantly, for allowing the use of TLS encryption.

LimeWire

Limewire is a Java application and therefore it is available at [76] for all operating systems. It is part of the original Gnutella network implementation and led to several other applications such as Acquisition, Cabos and FrostWire, just to cite a few. Besides Gnutella, it also supports BitTorrent as an additional protocol. The main reasons for its choice were its popularity and the ability to use TLS encryption for its traffic.

LimeWire is available under two versions. A freeware (LimeWire) and a payed version named LimeWire Pro, with built in enhanced features such as optimized search results, faster downloads and connections to more sources.

No matter what LimeWire version one is using, peer location and content searching are optimized using the *mojito DHT* [77]. This is a Kademlia DHT implementation for LimeWire, but not specific for this purpose, which enables it to be integrated with other software.

GTK-Gnutella

Gtk-Gnutella is a Gnutella client available for any Unix-like system that supports both GTK+ [20]and libxml [21] [78]. Although it has a very intuitive GUI, it is also too much simplistic, forcing some of its configurations to be done directly in the configuration files, under the *.gtk-gnutella* folder in the user home directory. The most important for this work was to enable TLS support, which was done by editing the *config_gnet* file and setting **tls_enforce = TRUE**.

Like Limewire, it is one of the few Gnutella clients that can also be configured to use TLS, wich was quite important for its choice. Gtk-Gnutella also provides DHT overlay network to locate peers and content, using the Kademlia DHT implementation.

3.7 P2P TV

P2P TV is becoming popular each day. It has been growing mainly due to the worldwide availability of large event transmissions such as the World and European Football championships, the 2008 Olympic Games in Beijing, the European Song Festival and, more recently, the Inauguration of Barack Obama as the 44th President of The U.S.A, on January 20 this year.

In the beginning, P2P TV applications were mostly based on Chinese broadcasts and peers, but there has been a remarkable growth of available channels. Other country based

[20]GTK+ is a open source package for creating Graphical User interfaces.

[21]Libxml is a XML C parser and toolkit.

P2P TV software is also multiplying, enabling worldwide broadcasts to reach a higher number of Internet users.

P2P TV advantages are notorious when comparing to traditional streaming mode, where any user pretending a stream connects to a unique server or set of servers. Independently of the amount of users a client/server system like this supports, bottlenecks are inevitable. A solution for a media content distribution company in a situation like this, could be to use geographically distributed servers to allow network load balancing, but at large costs. P2P TV allows any stream receiving peer to also become a provider, without the need of acquiring any other hardware. The scalability possibilities are therefore much higher when using this architecture and it also allows to overcome some geographical issues concerning the client and provider locations, that might influence the connection to cause low quality transmissions. Nevertheless, this problem still persists with some P2P TV networks for specific transmissions, as it is frequent to receive a message of the type "This stream is not available for your region" on many applications.

Some of P2P TV main characteristics are:

- Low infrastructure and maintenance cost
- Absence of physical obstacles
- Quality of Service (QoS) not guaranteed
- Less control of content distribution - When compared to traditional broadcasting

Quality and availability of the streams depend on the amount of users connected to the network, either by using specific P2PTV application such as TVU Player, or, more recently, by running provider's Web browser plugins like Octoshape, that allow users to watch TV in their favorite media player. More connected users means better stream quality, since every peer is a potential broadcaster as well.

After initial tests with many P2PTV applications, mostly based in China, like PPLive and TVAnts, it soon became clear that although most of their GUI were available in english, sooner or later messages in a foreign language in some configuration or pop-up window would appear, causing one to randomly selection of a given option that unexpectedly originated an awkward behavior. This happened twice for PPLive. Thus, in this work, only European and American P2PTV applications were used and they are LiveStation, TVU-Player, GoalBit and Octoshape. Results obtained with Octoshape were not included in this work due to legal issues.

3.7.1 LiveStation

LiveStation is a United Kingdom based P2P TV application that allows users to customize their channel list according to their preferences. This can be done either by using the application GUI itself, or by accessing the LiveStation web site at [79]. To use this functionality, one must previously create a free account where these settings will be stored and later imported every time the user loads the application.

Besides user provided worldwide channels (currently 4495), LiveStation ensures the streaming quality of partner broadcasters such as BBC World News, Al Jazeera, Bloomberg Television, France 24 and ITN just to cite a few. To start watching or listening any LiveStation or user provided channel, one just has to select it from the personalized list in the pleasant and easy to use GUI of the application.

LiveStation also provides instant messaging support for a given channel, which is a feature that has been gaining popularity not only for P2P TV but P2P client applications in general.

3.7.2 TVU Player

TVU Player is a product from the TVU Networks, available at [80]. The company was created in 2005 and is headquartered in Mountain View, California, U.S.A., with Asia Pacific offices in Shanghai, China. Besides TVU Player, the are also currently being developed the following applications:

- TVUPlayer_OSX - The TVU Player for Apple's OS X operating System, running on a Intel processor
- TVU Mobile - Player for 3G Mobile phones
- TVU Global - Correspondence between channels and the broadcaster location
- TVUVOD - Video on Demand

The TVUPlayer application has been downloaded 25 million times by viewers in over 200 countries. It uses a technology named Real-time Packet Replication (RPR), which enables the delivery of a live TV signal, of up to HD quality, to millions of TV viewers around the globe using a single TVUBroadcast appliance and a single broadband connection. Bandwidth required to broadcast does not increase proportionally with the number of viewers. So, according to TVUNetworks, *"this technology allows TVU broadcasters to achieve massively lower broadcast costs than with today's streaming technology."* [80]. With the RPR technology, content is delivered live, without being stored on TVU's or viewers' hard disks, avoiding legal issues.

One reason for the success of TVUPlayer, is its "democratic" broadcast concept, since any amateur or local broadcasters can become global broadcasters even if just using very few resources such as a videocamera and a Windows or Linux PC with a broadband Internet connection and the free TVUBroadcast application.

TVU Networks provide content rights management tools to allow broadcasters to limit their coverage to specific regions and also personalized advertising, targeted to viewers according to their geographical location. It has worldwide channel guide, that include news, sports, movies, music and many others, including those of broadcasting networks such as Fox News, ABC, NBC, CBS and many Asian broadcasters. Its interface is very intuitive and allows easy channel selection through its guide and search options. It is composed by three main panes. The upper is for searching and selecting media type, the left for channel selection and also displaying its ID and country origin and the last is for visualization. In

the left pane, each channel is presented with one of three logotypes. These are company registered logotypes, the TVU Networks logotype and the Windows Media Player one. For this work, only those of belonging to companies or the TVU Networks logotype were used, due to streaming protocol differences which will be further detailed in 4.5.2.

3.7.3 Octoshape

Octoshape is a streaming media client and server application, created by the Danish company Octoshape ApS [81], founded in 2003 by Stephen Alstrup and Theis Rauhe. It is available as an Adobe Flash Player plugin and it works on every major browser using Windows, Linux or Mac. Octoshape is oriented for major international broadcasters around the world and Content Delivery Networks (CDN), as it allows them to minimize their bandwidth requirements for large broadcasts.

Its technology is based on P2P streaming and is called *Grid Casting*. Their main differences, are that P2P uses a tree-structure so that a signal can only be received from a single computer in that overlay network at a time, while in a grid, every computer is a unit that is hierarchically equal to the other computers. This enables a stream to be received from a number of computers on the grid simultaneously avoiding bottlenecks, since the data is coming from multiple sources. Received data is then assembled from the several sources to recreate the stream.

Octoshape started to achieve popularity in 2008, when it was used by the European Broadcasting Union (EBU) to broadcast the Eurovision Song Contest via Internet. In the present year it also "helped CNN shatter the Internet live streaming record for the 2009 Presidential Inaugurations, where CNN reported 1.34 Million simultaneous users during the swearing in of President Obama" [81]. The companies listed bellow, use the Octoshape technology for streaming their contents.

- CNN.Com Live
- EBU : Eurovision Song Contest
- NBA Leage Pass Broadband
- Nascar RaceView
- 2008 Olympics Asia Delivery
- VRT : Tour de France

The complete list of its characteristics is available at [81], but the most important are the that it is platform independent, works with all major browsers and its codec independent technology allows Flash, Windows media, AAC+, MP3 etc.

Octoshape has been criticised for its license terms. Octoshape's EULA, amongst other things, prohibits the user from monitoring their own data traffic, or utilizing the records that their firewall or anti-virus software may record. The following citation was taken from the Octoshape End User License Agreement and it is also available during the plugin installation.

> "You may not collect any information about communication in the network of computers that are operating theSoftware or about the other users of the Software by monitoring, interdicting or intercepting any process ofthe Software. Octoshape recognizes that firewalls and anti-virus applications can collect such information,in which case you not are allowed to use or distribute such information. " [82]

The knowledge of this clause, long after many work on its traffic detection had been done, prevented the inclusion of the achieved results in this dissertation.

3.7.4 Goalbit

Unlike the previous P2P TV applications in this section, Goalbit [83] is available under the GNU General Public Lincense [68]. Developed by Uruguayan programmers, it runs on GNU/Linux, Solaris, and Microsoft Windows and it uses BitTorrent streaming (based on the BitTorrent protocol), in which a stream is decomposed into several flows sent by different peers to each client. In order to measure the peers perceived quality, it is used the recently proposed Pseudo-Subjective Quality Assessment (PSQA) technology, on which one can obtain information at [84].

Goalbit has a very simple interface with four initial Uruguay TV channels and allows one to add more channels using a *goalbit* file or an URL. It also allows any user to become a broadcaster after a few network, media capture and output settings have been done. Its supported input and media formats are:

- Input media: File, Video acquisition (DV, webcam), HTTP/MMS/FTP, UDP/RTP Unicast/Multicast, TCP/RTP Unicast, DVD, VCD, SVCD, etc.
- Supported formats (video and audio): MPEG-1, MPEG-2, MPEG-4, DivX, WMV, MP3, OGG, WMA

Goalbit provides GnuTLS features for transport security, but these settings are very basic since they only concern session expiration time and number of resumed sessions.

3.7.5 Joost

Another initially studied P2P TV application was Joost [85]. Its development started in 2006, after the creators of Skype [86] and Kazaa [87] Niklas Zennstrom and Janus Friis sold it to eBay [88] in 2005. The goal of Joost was to offer a free application for viewing TV on the Internet, supported by commercial ads, but briefer and less frequent than those on regular TV. In October 2008, Joost introduced a web-based version of this software to allow in-browser viewing and in December of that year, the application was discontinued to adopt a permanent browser based approach [85]. For this work, only the in-browser version was tested.

Joost network relies on several components. These include Web servers, data servers responsible for holding information about the available TV shows and, finally, servers used for managing the P2P network. The video distribution is based on on a proprietary video plugin called *Joost Plugin*, which downloads parts of the intended video using several simultaneously sources.

> "Joost uses a peer-to-peer (P2P) network, which means that you don't pull the video from one specific source, but you pull bits of the video from the other peers (a.k.a. people like you) who are on Joost." [85]

Just like many of the so called P2P TV applications, Joost does not operate as a regular TV broadcaster, but more as a Video on Demand (VoD) service. In this kind of service, users are given the chance to select the programs to watch according with their preferences, organized through categories such as Sports, Animation, Comedy, Documentaries, Science-Fiction, etc. Although it was not possible to obtain more information, Joost and partner broadcasters such CBS conducted tests regarding live video streaming in 2008. Until the present moment, it was not possible to verify if this kind of distribution is already available, since only the usual short videos seem to be displayed. Another P2P TV (VoD) example is Babelgum [89].

Joost inherited its proprietary encryption features from Skype, with the purpose of protect the transmission, but according to the techial report in [90], it is used to bypass security controls. This may be the reason why it was not possible to identify specific Joost traffic in this work. Nevertheless, it was observed that the communications using the Web Joost plugin always used TCP port 80 and therefore they were classified as HTTP traffic.

As a parallel study, there were installed several other P2P TV applications and plugins to test their features and the provided channel list. These applications were Babelgum, Abacast [91] (which company was kind enough to send a reply concerning a technical query) and the open source Mint [92] and Alluvium [93] applications. Zattoo P2P TV application [94] is not yet available in Portugal.

Chapter 4

P2P Traffic Detection

4.1 Introduction

This chapter contains information about the procedures concerning P2P traffic detection and the results obtained by them, for the protocols already mentioned in table 1.1. Although some P2P applications use the same protocol, there might be, in some cases, some slightly different implementations. This was the main reason for using at least two applications for each studied P2P File Sharing Protocol, so that the detection results could be compared. On the other hand, P2P TV protocols are mainly proprietary and used by a single application.

The detection of P2P traffic was accomplished by using a set of open source tools, emphasizing Snort, Wireshark and Tcpdump, respectively for the process of triggering and detecting the alerts. Along with some logs, the alerts were visualized by using a Web interface provided by BASE, which connects to a MySQL database where they are stored.

The procedure for the creation of Snort rules is pretty much the same for all protocols and applications during this work . Along with the rules provided by the Snort distribution for a given protocol or application (no rules were provided for the studied P2P TV applications), new rules were manually introduced, as protocol signatures and traffic patterns were being detected. To obtain the most accurate possible rules, the traffic through the Snort classifier was kept to minimal, so that it would be easier to focus on the intended traffic. Nevertheless, most of this work was done remotely, away from the NMCG lab, which forced Snort to analyze other network traffic than P2P, such as HTTP, Windows Remote Desktop Connection (RDC), SSH, etc. In fact, this was quite worthy, since it enabled the testbed setup to run in similar circumstances of those of deployed P2P classifiers, which also have to deal with network traffic generated by a vast number of applications and then to correctly identify P2P among it.

The identification of P2P traffic patterns was done by collecting incoming and outgoing traffic from the workstations running P2P applications. This was mostly done using Tcpdump, specially when predicting large amounts of traffic, so that the output would be stored in a binary file using the less system resources as possible, allowing the traffic to be later analyzed by Wireshark in a more user friendly manner. In many situations a filter was applied during the capture, so that RDC or SSH traffic from the remote connections to the NMCG

lab were not considered for later visual analysis. When a frequent pattern was detected, a Snort rule was manually coded based on that pattern, on the position within the payload and on any other useful information that could improve the effectiveness of that rule. If the initial tests were satisfactory, these rules were then included on the Snort rule set for that P2P protocol or application and considered for the detection statistics, visualized through BASE and its MySQL database. These tasks were performed for all the applications included in this work.

This chapter is organized as follows: Sections 4.2.1 and 4.2.2 are dedicated to the detection of BitTorrent traffic using BitTorrent and Vuze applications respectively. The results for the detection of Gnutella protocol version 0.6 are divided among sections 4.3.1 and 4.3.2, concerning LimeWire and GTK-Gnutella applications. For the detection of the eDonkey protocol there were used eMule and aMule applications, in sections 4.4.1 and 4.4.2 respectively. As for the study of P2P TV traffic, four applications were initially used. Due to legal issues already described in section 3.7.3, only Livestation, TVU Player and Goalbit were included in this chapter, respectively in sections 4.5.1, 4.5.2 and 4.5.3.

4.2 BitTorrent

4.2.1 BitTorrent Application

BitTorrent application version 6.1.2 was configured so that it would **only** allow bidirectional **encrypted connections**, in other words., both outgoing and incoming traffic had to be encrypted, so that communication was possible with other BitTorrent clients (applications). Nowadays, users tend to use these settings to avoid being throttled or blocked by their ISPs. As a consequence, there are not so many sources available to download if one does not use the "‘Forced"’ setting for outgoing encrypted traffic, since other clients are mostly configured to deny "‘legacy connections"’, thus not allowing unencrypted connections. These settings are configured under the menu Options → Preferences → BitTorrent → Protocol Encryption. To only use encrypted connections, the **Outgoing** combo box must be set with the value **Forced** and **Allow incoming legacy connections** must be unchecked.

In all of the following tests, the setting **Ask the tracker scrape information**, also under Options → Preferences → BitTorrent → was always checked. This enables the client to obtain newer peers and provide statistics about their availability. Although it is not mandatory, specially if other mechanisms are used to obtain peer information like the DHT, it can be useful to maintain updated records about resource availability. It is important to notice that if this setting is unchecked, there is no traffic for *BitTorrent tracker request* and, consequently, the rules for detecting it are never triggered. For this work, it was kept checked for studying the frequency of communications to the tracker.

Besides P2P, there was also SSH, HTTP and RDC traffic through Snort during all the following tests. The first two tests were conducted with the previous mentioned settings and with DHT disabled, so that BitTorrent would not generate too much control traffic, making it harder to detect. The following rules were triggered:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"P2P BitTorrent Outgoing announce
request"; flow:to_server,established; content:"GET"; offset:0; depth:4; content:"/announce";
distance:1; content:"info_hash="; offset:4; content:"event=started"; offset:4;
classtype:policy-violation; sid:1000301; rev:1;)
```

Snort Rule 1000301. Rule for detection of traffic generated through BitTorrent.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P BitTorrent Outgoing
tracker request"; flow:to_server,established; content:"GET"; offset:0; depth:4;
content:"/scrape"; distance:1; content:"info_hash="; offset:12; content:"User-Agent:";
offset:80; classtype:policy-violation; sid:1000305; rev:1;)
```

Snort Rule 1000305. Rule for detection of traffic generated through BitTorrent.

Table 4.1 shows detailed information about the test results.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
17-01-2009	20:34	21:58	280791	107825488	22	18.4	1000301 1000305	1 1
27-01-2009	21:31	21:44	23175	10546443	1.2	3.0	1000301 1000305	1 1

Table 4.1: Characteristics of experiences and their detection results for BitTorrent traffic.

So, even with DHT disabled, two snort rules for TCP traffic are frequently triggered. In this case it happened only once, due in part to the small the amount of BitTorrent traffic. In the following tests, one can confirm a greater occurrence of them. Once again it is important to emphasize, that if the **Ask the tracker scrape information** was unchecked, rule 1000305 would never be triggered at all.

For the next tests, four more rules were introduced. They refer to DHT traffic, and use UDP unlike the previous ones. They are listed bellow.

```
alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P BitTorrent UDP -
Outgoing DHT for trackerless comunication request (d1:ad2:id20)"; content:"d1:ad2:id20";
nocase; depth:11; classtype:policy-violation; sid:1000306; rev:2;)
```

Snort Rule 1000306. Rule for detection of traffic generated through BitTorrent.

```
alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P BitTorrent UDP -
Incoming DHT for trackerless comunication request (d1:ad2:id20)"; content:"d1:ad2:id20";
nocase; depth:11; classtype:policy-violation; sid:1000307; rev:3;)
```

Snort Rule 1000307. Rule for detection of traffic generated through BitTorrent.

```
alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P BitTorrent UDP -
Incoming DHT for trackerless comunication response (d1:rd2:id20)"; content:"d1:rd2:id20";
nocase; depth:11; classtype:policy-violation; sid:1000308; rev:3;)
```

Snort Rule 1000308. Rule for detection of traffic generated through BitTorrent.

```
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P BitTorrent UDP -
Outgoing DHT for trackerless comunication response (d1:rd2:id20)"; content:"d1:rd2:id20";
nocase; depth:11;classtype:policy-violation; sid:1000309; rev:3;)
```

Snort Rule 1000309. Rule for detection of traffic generated through BitTorrent.

Rules 1000306 and 1000307 could be combined into a single one. The only advantage in specifying them independently, is that this way it is possible to easier distinguish incoming from outgoing traffic. The same thing happens with rules 1000308 and 1000309 and it will be recurrent during this work. Table 4.2 shows more information about the test allowing the use of UDP and DHT.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
01-02-2009	23:01	23:21	71783	46023309	15	6.1	1000301 1000305 1000306 1000307 1000308 1000309	3 2 1562 689 24 30

Table 4.2: Characteristics of experiences and their detection results for BitTorrent traffic.

As one can easily see, enabling the useful DHT feature allows to successfully identify UDP traffic for *trackerless requests* and *trackerless responses*.

Two additional rules were triggered during the tests on the BitTorrent application. They are available at [95] and were included in this work for test purposes. They are listed bellow.

```
#http://www.emergingthreats.net/rules/emerging-p2p.rules
#By David Bianco
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"ET P2P BitTorrent DHT
ping request"; content:"d1\:ad2\:id20\:"; depth:12; nocase; threshold:
type both, count 1, seconds 300, track by_src; classtype:policy-violation;
reference:url,wiki.theory.org/BitTorrentDraftDHTProtocol; sid:2008581; rev:1;)
```

Snort Rule 2008581; Obtained from [95].

```
#http://www.emergingthreats.net/rules/emerging-p2p.rules
#By David Bianco
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"ET P2P BitTorrent DHT get_peers
request"; content:"d1\:ad2\:id20\:"; nocase; depth:12; content:"9\:info_hash20\:"; nocase;
distance:20; depth:14; content:"e1\:q9\:get_peers1\:"; nocase; distance:20; depth:17;
threshold:  type both, count 1, seconds 300, track by_src; classtype:policy-violation;
reference:url,wiki.theory.org/BitTorrentDraftDHTProtocol; sid:2008584; rev:1;)
```

Snort Rule 2008584; Obtained from [95].

Rule 2008581 is identical to the locally developed 1000306. They share some of their content, more exactly **d1:ad2:id20**. Even though, rule 1000306 triggered 614 times against a single one of 2008581. With these additional rules included and also enabling the DHT features, it was possible to obtain the results listed in table 4.3.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
03-02-2009	20:47	20:59	20434	8642013	0.14	3.4	1000301	3
							1000305	3
							1000306	614
							1000307	222
							1000308	17
							1000309	11
							2008581	1
							2008584	1

Table 4.3: Characteristics of experiences and their detection results for BitTorrent traffic.

Another test was conducted in the same circumstances than the previous, but generating a bit more traffic. For this, it was select a torrent file for a drama movie released in 2008. The results obtained are listed in table 4.4

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
07-02-2009	19:53	22:57	231536	134571450	63.5	46.7	1000301	2
							1000305	2
							1000306	8423
							1000307	4258
							1000308	57
							1000309	31

Table 4.4: Characteristics of experiences and their detection results for BitTorrent traffic.

As one can see, rules 1000306, 1000307, 1000308 and 1000309 are triggered much often than 1000301 and 1000305. This is because when DHT is enabled, peers communicate frequently with each other to check for data and peer availability. As for rule 1000301, it is only triggered when a peer tells another that it is interested in some file shared by it and this usually occurs only just before beginning the download of another *chunk*. If the *scrape* feature is disabled, through the **Ask the tracker scrape information** option, rule 1000305 is not triggered at all, since communication with the tracker with the *scrape* content does not occur. The complete set of Snort rules created for the detection of BitTorrent traffic is provided in appendix C.

4.2.2 Vuze Application

Vuze also uses the BitTorrent protocol, and so, also belongs to the Unstructured, Hybrid Decentralized, Tracker based architecture. Vuze was chosen for being one of the most popular BitTorrent applications and since it is the successor of Azureus, it inherited all of its features, including its encryption capabilities. Version 4.1.0.0 was installed in windows

and tested with different configurations, as its interface is more complete (and complex) than that of the BitTorrent application.

One main difference between these two applications, is that Vuze allows to select two encryption types: Plain and RC4. While Plain encryption is least CPU intensive than RC4, it does not provide so much stealth capabilities since the payload itself is not encrypted.

Just like the BitTorrent Application, rule 1000305 is never triggered unless *scraping* is active. This is accomplished by checking the **Enable scraping** option under menu Tools → Options → Tracker → Client → Scrape. In all the following cases it was kept checked for studying the frequency of communications to the tracker. Another default option in every of the following tests, was the *Allow non-encrypted incoming connections* unchecked, so that only encrypted traffic could reach Vuze.

Besides P2P, there was also SSH, HTTP and RDC traffic through Snort during all the following tests. All the previously rules used for the BitTorrent Application detection, already listed in 4.2.1, were also used for Vuze, but a few more have been specifically created for it. P2P applications have sometimes slightly different implementations of the protocols and also possess different features, which generate different traffic signatures.

The following rules are specific for Vuze when using Plain encryption. It is important to notice that rule 1000314 and 1000315 could be written into a single one, but that would not allow to easily distinguish the source and destination of the traffic. The same happens to 1000316 and 1000317.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P Vuze Plain Encryption
Outgoing BitTorrent_Handshake"; flow:to_server; content:":BT_HANDSHAKE3:"; nocase;
classtype:policy-violation; sid:1000314; rev:2;)
```

Snort Rule 1000314. Rule for detection of traffic generated through Vuze.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P Vuze Plain Encryption
Incoming BitTorrent_Handshake"; flow:to_server; content:":BT_HANDSHAKE3:"; nocase;
classtype:policy-violation; sid:1000315; rev:2;)
```

Snort Rule 1000315. Rule for detection of traffic generated through Vuze.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P Vuze Plain Encryption
Outgoing BitTorrent Azureus_Handshake"; flow:to_server; content:"AZ_HANDSHAKE"; offset:8;
depth:12; nocase; classtype:policy-violation; sid:1000316; rev:1;)
```

Snort Rule 1000316. Rule for detection of traffic generated through Vuze.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P Vuze Plain Encryption
Incoming BitTorrent Azureus_Handshake"; flow:to_server; content:"AZ_HANDSHAKE"; offset:8;
depth:12; nocase; classtype:policy-violation; sid:1000317; rev:1;)
```

Snort Rule 1000317. Rule for detection of traffic generated through Vuze.

Another introduced rule, although it occurred only in one test session, was taken from [95] and is listed bellow.

```
#http://www.emergingthreats.net/rules/emerging-p2p.rules
# By Chich Thierry
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"ET P2P BitTorrent peer
sync"; flow: established; content:"|0000000d0600|"; offset: 0; depth: 6;
reference:url,bitconjurer.org/BitTorrent/protocol.html; classtype: policy-violation; sid:
2000334; rev:8;)
```

Snort Rule 2000334; Obtained from [95].

Disabled DHT, Plain Encryption

The following tests were conducted with DHT disabled, Plain encryption and the default settings previously mentioned. Table 4.5 shows detailed information about the test results, while downloading Fedora 10 Live CD.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
02-02-2009	22:01	22:12	31990	11914192	3.62	0.1	1000301 1000305 1000314 1000316	2 5 16 16
02-02-2009	22:45	23:03	89838	46923131	16.69	2.13	1000301 1000305 1000314 1000316 1000334	1 2 1 1 34
03-02-2009	23:06	23:41	48695	21455082	7.18	1.56	1000301 1000305 1000314 1000316	1 4 3 3

Table 4.5: Characteristics of experiences and their detection results for Vuze traffic.

For the next test it was used a different torrent file, for downloading a movie from 1954. The idea was to generate more download/upload traffic for a less pretended resource, to generate more DHT search requests. The results are shown in table 4.6.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
06-02-2009	14:40	16:36	524075	264170469	191.41	23.4	1000301 1000305 1000314 1000315 1000316 1000317	20 11 283 2 267 1

Table 4.6: Characteristics of experiences and their detection results for Vuze traffic.

As one can observe, the fact that most influences the number of triggered alerts, is the amount of data that was exchanged between Vuze and the tracker and also between other peers.

Enabled DHT, RC4 Encryption

In this section, there were conducted tests to verify if it was possible to detect RC4 encrypted Vuze traffic, just like when using the Plain Encryption. Although it is more CPU demanding, it makes it harder to detect, since the well known pattern "|13|BitTorrent protocol" is never sent in clear text.

Initially, using all the previous defined rules, only number 1000301 e 1000305 were triggered. To emphasize the fact of rule 1000305 only appears when **Enable scraping** option is checked, the second row of the following table shows traffic statistics when scraping is disabled, unlike the first ant third rows. Another important note, is that information shown in the first and second row, was collected locally, that is, without any other traffic than P2P trough Snort, unlike in most tests when there is also SSH, HTTP and RDC traffic. Nevertheless, this had absolutely no influence in the test results, since the alerts triggered were the same and there were also no false positives. Table 4.7 concerns the traffic statistics for downloading the trailer of an animation movie released in 2008.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
06-02-2009	15:55	17:19	65426	36687992	27.78	1.33	1000301 1000305	7 4
06-02-2009	17:57	18:22	92662	59369991	49.77	0.26	1000301	4
07-02-2009	11:51	12:05	94858	58819111	49.84	0.23	1000301 1000305	2 3

Table 4.7: Characteristics of experiences and their detection results for Vuze traffic.

The statistics displayed in table 4.8, concern the download of a dramatic movie released in 2008. This exact torrent file was also used with BitTorrent Application, but this time with significant more download traffic.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
07-02-2009	12:16	15:29	526976	278167515	160.29	52.75	1000301 1000305	6 9

Table 4.8: Characteristics of experiences and their detection results for Vuze traffic.

As one can notice, more alerts for rules 1000301 and 1000305 were accounted with Vuze than for the same movie download using BitTorrent (complete results for BitTorrent are displayed in table 4.4). Table 4.9 compares the amount of traffic with the alerts counted.

	Download	Uploade	1000301	1000305
BitTorrent	63,5 MB	46,7 MB	2	2
Vuze	160,29 MB	52,75 MB	6	9

Table 4.9: Comparison of the detection results obtained for BitTorrent and Vuze applications, using the same torrent file.

Comparing tables 4.4 and 4.8, one can notice that rules concerning DHT traffic (rules 1000306,1000307,1000308 and 1000309) were not triggered in Vuze. In fact, neither of the previous tests triggered any of those. This originated more focused tests on DHT rules. After many research, the conclusion was that the DHT protocol implementations from Vuze and BitTorrent applications are different, although they are both based on *kadmelia*, described at 2.3.2.

The following Snort rules number 1000310 and 1000311 were created separately, although they could be combined into a single one by specifying the bidirectional operator <>. This way the alerts would be triggered independently of the traffic flow direction, but for testing and accounting purposes they were kept this way.

```
alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P Vuze UDP - Outgoing
DHT"; content:"d1:c0:1:n0:1"; nocase; classtype:policy-violation; sid:1000310; rev:2;)
```

Snort Rule 1000310. Rule for detection of traffic generated through Vuze.

```
alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P Vuze UDP - Incoming
DHT"; content:"d1:c0:1:n0:1"; nocase;classtype:policy-violation; sid:1000311; rev:2;)
```

Snort Rule 1000311. Rule for detection of traffic generated through Vuze.

With the introduction of these rules, it was now possible to detect incoming and outgoing Vuze DHT traffic. Table 4.10 shows information about the rules triggered during the Fedora 9 Live CD download, with scraping enabled and also SSH, HTTP and RDC traffic, as usual.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
07-02-2009	14:08	17:29	1119829	819865361	691.84	13.30	1000301 1000305 1000310 1000311	9 15 37 12

Table 4.10: Characteristics of experiences and their detection results for Vuze traffic.

After being able to detect Vuze DHT traffic, with the rules presented above, there were still two questions needing an answer. The DHT rules that triggered with BitTorrent application never worked with Vuze. It had been necessary to create specific ones for it. But then, could Vuze and other BitTorrent applications interact via DHT, if tracker communications were disabled (when no central servers are used to obtain information about peers), since

their DHT implementations may differ ? If so, could this traffic be detected ? The answer to both is yes. After some research it was possible to find a compatible DHT mode for Vuze. This implementation allows Vuze to fully interact with other BitTorrent applications using the so called *Mainline DHT plugin*, available at [96]. After adding this plugin into Vuze, it was necessary to generate some traffic to check if the "regular" DHT communications were taking place and also, if they would trigger the rules 1000306, 1000307, 1000308 and 1000309, already show in 4.2.1. When this was confirmed, it was performed the same test as in table 4.8. One rule was triggered for the first time. It was taken from [95], just like other Snort rules previously introduced in section 4.2.1 and its code is listed bellow.

```
#http://www.emergingthreats.net/rules/emerging-p2p.rules
#By David Bianco
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"ET P2P BitTorrent DHT nodes
reply"; content:"d1\:rd2\:id20\:"; nocase; depth:12; content:"5\:nodes"; nocase;
distance:20; depth:7; threshold:  type both, count 1, seconds 300, track by_src;
classtype:policy-violation; reference:url,wiki.theory.org/BitTorrentDraftDHTProtocol;
sid:2008583; rev:1;)
```

Snort Rule 2008583; Obtained from [95].

Table 4.11 lists all the triggered rules for Fedora 9 Live CD download. It is notorious the amount of *Mainline* DHT traffic detected, after the installation of the respective plugin into Vuze, with approximately the same overall generated and analyzed traffic as in 4.8

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
07-02-2009	18:02	20:46	1154088	815445209	691.80	14.53	1000301	3
							1000305	9
							1000306	1035
							1000307	764
							1000308	11
							1000309	11
							1000311	13
							2008583	1
							2008584	1

Table 4.11: Characteristics of experiences and their detection results for Vuze traffic.

The complete set of Snort rules created for the detection of BitTorrent traffic using Vuze is provided in appendix C.

4.3 Gnutella

4.3.1 LimeWire

The first tests with LimeWire were initially meant to verify in which conditions the connection to the Ultrapeers was possible and what traffic could be detected in this stage. If one does not successfully connect to three Ultra Peers, than is not connected to the Gnutella network and, consequently, when searching for some content to download, the following message comes up:

> "LimeWire is not currently connected to the network. Your search may not return many results until you are fully connected to the network." [76]

This application comes with following features disabled by default, under the menu Tools → Options → Advanced → Performance and their settings revealed extremely important for this work:

- Disable Ultrapeer Capabilities - Unchecked
- Disable Mojito DHT Capabilities - Unchecked
- Disable TLS Capabilities - Uncheked

Checking the first option disables LimeWire application to work as an ultrapeer, that is, it does not provide searching or allocation resources for others peers in the network. With the *Mojito DHT* enabled, one has more chances to find (correctly) the pretended resources, according to the DHT functionalities already mentioned before. As for the TLS capabilities, this one was the most important setting of all. If disabled, only for a few times the connection to the Gnutella network was successfully established, but after many hours of waiting. At least once, it took more ten hours to connect. The reason for this (just like in section 4.2.1), is that P2P users are forcing their applications to use all methods available so they can go undetected, to avoid traffic shaping or being blocked by their ISPs. Users that do not use this mechanisms find themselves isolated, since most other applications do not allow unencrypted connections to them and therefore they simply can not connect, or find enough resources to download from.

The first rule developed for Gnutella traffic detection was modified from the original Snort distribution. It is now more precise and fast, since there is less payload content to analyze when comparing it to the original. After the "/" slash, it could be specified the version "0.4" or "0.6", but to try to detect any version of the Gnutella protocol, it was kept simple. The rule is given by:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P Gnutella Outbound Connect
Request (gnutella connect)"; flow:to_server,established; content:"GNUTELLA CONNECT/";
nocase; depth:17; classtype:policy-violation; sid:1000201; rev:2;)
```

Snort Rule 1000201. Rule for detection of generic Gnutella traffic.

The following tests, displayed across the tables 4.12 and 4.13, show two different scenarios. The first one, without using TLS encryption and DHT disabled in the first row and enabled in the next. The second scenario is relative to the use of TLS, with DHT enabled on the first row and disabled on the next.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
11-02-2009	21:24	21:49	7471	660444	-	-	1000201	587
	22:03	22:16	5297	466585	-	-	1000201	412

Table 4.12: Characteristics of experiences and their detection results for LimeWire DHT traffic, with TLS encryption settings off.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
11-02-2009	22:20	22:21	834	124126	-	-	1000201	2
	22:30	22:32	726	159803	-	-	1000201	3

Table 4.13: Characteristics of experiences and their detection results for LimeWire DHT traffic, with TLS encryption settings on.

In table 4.12, with TLS disabled, connection to the ultrapeers was never achieved although the application run for much more time than in 4.13. More traffic was generated and that enabled rule 1000201 to trigger many more times.

When TLS was enabled, in 4.13, the connection to the ultrapeers was established very quickly. The test was then stopped immediately, but enabling to capture rule 1000201.

In both previous scenarios, the use of DHT had absolutely no influence in the establishment of the connection to the Gnutella network, which is is solely relative to the use or not of TLS encryption. It was possible to observe that even thought TLS encryption enabled, the *GNUTELLA CONNECT/* content in the payload, concerning the connection between the peer (leaf) and the servent (ultrapeer), could still be detected. This suggests that not all TCP traffic is encrypted, at least from the early beginning.

LimeWire - TLS Encryption

All the following tests were performed with the TLS encryption feature set on LimeWire. Even though, observing the originated traffic during some tests, it was possible to detect some patterns. The following rules were introduced, the first one for TCP traffic, the others for UDP:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P LimeWire GET uri-res
afinada"; flow:to_server,established; content:"GET /uri-res/n2r"; nocase; depth:16;
content:"urn:sha1:"; distance:1; content:"X-Gnutella-Content-URN"; nocase; offset:124;
content:"urn:sha1:"; distance:1; classtype:policy-violation; sid:1000203; rev:2;)
```

Snort Rule 1000203. Rule for detection of traffic generated through LimeWire.

```
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P LimeWire
UDP - X-Gnutella-Content-URN"; content:!"GET /uri-resA"; nocase; offset:4;
content:"X-Gnutella-Content-URN:"; nocase; offset:124; content:"urn:sha1:";distance:1;
classtype:policy-violation; sid:1000256; rev:1;)
```

Snort Rule 1000256. Rule for detection of traffic generated through LimeWire.

```
 alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P LimeWire
UDP - X-Gnutella-Content-URN"; content:!"GET /uri-resA";nocase;offset:4;
content:"X-Gnutella-Content-URN:"; nocase; offset:124; content:"urn:sha1:";distance:1;
classtype:policy-violation; sid:1000257; rev:1;)
```

Snort Rule 1000257. Rule for detection of traffic generated through LimeWire.

It is important to notice that rules 1000256 and 1000257 use the negation operator "!". This is because the string "**X-Gnutella-Content-URN:**" made also part of the payload of several other packets which originated rules 1000254 and 1000255 (that will be introduced later). The goal of using this mechanism, was to guarantee that only traffic containing the string "**X-Gnutella-Content-URN:**" and not "**GET /uri-resA**", "**/n2r**" and "**urn:sha1:**" was detected.

Rules 1000256 and 1000257 are equivalent, except for the source and destination. As it happened before with other protocols and applications, their separate implementation is for accounting purposes only, since they could be combined into just one. Table 4.14 displays information about the traffic and rules triggered during the download of a drama, sci-fi movie, releasead in 2008.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
13-02-2009	15:51	15:56	20104	10385952	7.35	0	1000201 1000203 1000256 1000257	2 14 16 15
13-02-2009	17:42	18:26	282305	170712815	104.3	0.34	1000201 1000203 1000256 1000257	11 33 119 62
14-02-2009	19:14	22:22	1279249	788069608	646.2	0.36	1000203 1000256 1000257	81 105 56

Table 4.14: Characteristics of experiences and their detection results for LimeWire traffic, with TLS encryption settings on.

Information displayed in rows one and two in the previous table was collected with DHT enabled, but this had no influence on the results comparatively to those on the third row. Rule 1000201 is not necessarily triggered, unless when connecting the LimeWire application to the Gnutella network. This was tested for several times, for example, when resuming a download, or when maintaining an established connection to the network and than search and download new content.

In the previous tests there were triggered two false positives. They are rules 1000410 and 1000411 relative to TVU player traffic and will be discussed later in section 4.5.2. Their occurrences are relative to the tests listed in 4.14.

Test	Rule 1000410	Rule 1000411
1	20	20
2	20	19
3	13	10

Table 4.15: Occurrence of false positives in the tests reported in table 4.14.

The same ruleset was applied once again, but now, for a different movie download. This time for a 2008 animation movie, with DHT enabled. Table 4.16 contains information about the traffic and triggered rules since the start of the LimeWire application, through the search of the intended movie and almost until its conclusion.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
15-02-2009	10:04	10:31	614449	518948818	457.9	0.25	1000201 1000203 1000256 1000257	2 4 60 55

Table 4.16: Characteristics of experiences and their detection results for LimeWire traffic, with TLS encryption and DHT settings on.

Once again, enabling or disabling the DHT in LimeWire did not influence the test results, as the accounted alerts tend to be similar for the same amount of traffic.

Two other rules were triggered besides those listed previously. They are again rules 1000410 and 1000411, concerning TVU player traffic. Their occurrences were 28 and 36 times respectively.

After observing many LimeWire application originated UDP packets with Wireshark, it was possible to detect a pattern almost from the beginning of their payloads. They are composed by three content blocks in a given distance from each other, which enabled to detect additional traffic. Their ids are 1000254 and 1000255 and are listed below.

```
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P LimeWire UDP -
Outgoing GET uri-resA"; content:"GET /uri-resA"; nocase; offset:4; content:"/n2r"; nocase;
distance:6; content:"urn:sha1:"; distance:1; classtype:policy-violation; sid:1000254;
rev:2;)
```

Snort Rule 1000254. Rule for detection of traffic generated through LimeWire.

```
 alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P LimeWire UDP -
Incoming GET uri-resA"; content:"GET /uri-resA"; nocase; offset:4; content:"/n2r"; nocase;
distance:6; content:"urn:sha1:"; distance:1; classtype:policy-violation; sid:1000255;
rev:2;)
```

Snort Rule 1000255. Rule for detection of traffic generated through LimeWire.

After including these two rules into the Gnutella ruleset, another test was conducted using the same movie download as before, but with more 100 MB of downloaded traffic. The results are presented in table 4.17 and the false positives detected during the previous test were, once again, relative to rules 1000410 and 1000411, with 27 and 26 occurrences each.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
15-02-2009	11:41	12:13	696665	647774917	570.2	0.35	1000203	14
							1000254	12
							1000255	18
							1000256	18
							1000257	12

Table 4.17: Characteristics of experiences and their detection results for LimeWire traffic, with TLS encryption and DHT settings on.

The inclusion of Snort rules number 1000254 and 1000255, allowed to detect more Gnutella UDP traffic. As one can see in table 4.17, their occurrences are very similar to the previously defined ones. Another fact is that rule 1000201 was not triggered, unlike in table 4.16, although the test was executed without using the previously established Gnutella connection, in other words, LimeWire application was restarted for this test. One possible explanation for this, that requires more investigation, is that it may be possible that the application uses some ultra peer caching mechanism so it does not need to send a "regular" connect request. The only scenario where rule 1000201 was always triggered, was after an operating system restart and then open LimeWire and try to connect with ultra peers.

The following test, displayed in table 4.18, was a resume of the previous download and, consequently, rule 10002001 was not detected. DHT was disabled this time but as one can see, the results do not differ much although much less traffic was generated.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
15-02-2009	12:27	12:45	209892	145300813	116.2	0.2	1000203	5
							1000254	17
							1000255	14
							1000256	14
							1000257	17

Table 4.18: Characteristics of experiences and their detection results for LimeWire traffic with DHT disabled and TLS encryption settings on.

Again as false positives, there are 21 occurrences of rule 1000410 and 18 of 1000411. Although the traffic volume was about 4.4 times greater in table 4.17 than in 4.18, the amount of false positives relative to TVU Player traffic was not proportional. The complete set of Snort rules created for the detection of LimeWire traffic is provided in appendixes B.1, B.2 and B.3.

4.3.2 GTK-Gnutella

GTK-Gnutella 0.96.5 was solely installed on Linux, on the same machine where Snort was, just for convenience. It was setup so it would always use TLS encryption for all the following tests. Although it has a graphical interface, some configurations had to be done in the *config_gnet* file, under the user home folder *.gtk-gnutella*. The most important was the use of TLS, set by **tls_enforce = TRUE**. Some other important settings were made in the graphical interface. They included:

- Network Settings

  IP settings

  Listen Port

  Use of UDP

- Gnutella Network Mode

To change the network related settings, it was used the menu File → Preferences → Network. The default listen port was set to **10293** and the application was forced to use the external IP address **193.136.67.242**, so that incoming traffic could get to it through Snort, using the previously defined iptables rules in section 3.4.1.

The Gnutella Network Mode, configured in menu File → Preferences → Gnutella, was set to **leaf mode** so that the application worked as a regular peer. In this mode no searching or indexing functions are provided, unlike the ultra peers or ultra nodes as they are designated in GTK-Gnutella.

Just like in the LimeWire application, GTK-Gnutella does not usually achieve connection to three ultra peers (default number in most Gnutella applications) unless TLS encryption is used. If it does, then this only happens after many hours of trying and there is no guarantee about it. Once again, this happens because of most user configurations, that do not allow unencrypted connections to their own machines. Another fact observed during the tests, was that the vast majority of the ultra peers were using LimeWire as the Gnutella application.

The only rules that were both triggered with LimeWire and GTK-Gnutella, were those for TCP traffic containing the strings "**GET /uri-res/n2r**", "**urn:sha1:**" and "**X-Gnutella-Content-URN**", although they did not occur so often for GTK-Gnutlla. Rule 1000203 was already shown in the previous section and rule 1000204 examines exactly the same content, but with reverse values for the source and destination of the traffic. Rule 1000204 is listed bellow.

```
 alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P GTK-Gnutella Incoming
uri-res afinada"; flow:to_server,established; content:"GET /uri-res/n2r"; nocase; depth:16;
content:"urn:sha1:"; distance:1; content:"X-Gnutella-Content-URN"; nocase; offset:124;
content:"urn:sha1:"; distance:1; classtype:policy-violation; sid:1000204; rev:2;)
```

Snort Rule 1000204. Rule for detection of traffic generated through GTK-Gnutella.

After many tests, it became clear that TCP traffic would not be detected, or at least not often, due to the use of TLS. The first three bytes of the initial packets contain the

hexadecimal values "**16 03 01**" or "**17 03 01**" (that also appear in the beginning of many tls and ssl communications), concerning the beginning of the encrypted communication, after which only random like patterns are observed.

GTK-Gnutella, as LimeWire, does not use encryption for UDP traffic and since this protocol is enabled by default, to allow better search mechanisms using the Kademlia based DHT, some rules were created based on the observed GTK-Gnutella UDP payload.

```
 alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P GTK-Gnutella UDP
- Incoming DHTC"; content:"|60 60|"; offset:2; content:"DHTC"; offset:39; nocase;
classtype:policy-violation; sid:1000261; rev:1;)
```

Snort Rule 1000261. Rule for detection of traffic generated through GTK-Gnutella.

Using this new rules and all the previous ones for Gnutella traffic detection, there were conducted several tests, displayed in table 4.19, to evaluate their occurrences during the GTK-Gnutella application startup and connection to the network, as well as the post connection period without any user activity.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
19-02-2009	21:25	21:26	676	102536	-	-	1000261	30
19-02-2009	22:08	22:09	208	28401	-	-	1000261	2
19-02-2009	22:12	22:21	418	46865	-	-	1000261	3
20-02-2009	19:40	22:14	408	41307	- -	- -	1000261 1000204	34 43

Table 4.19: Characteristics of experiences and their detection results for GTK-Gnutella traffic, with TLS encryption settings on.

Data in the first and second rows refers to traffic analyzed since the application started, until the connection to the three Gnutella ultra peers. As one can see, the number of alerts obtained in the first test is considerably higher than those on the second row. This is due to automatic update of the file *.gtk-gnutella/ultras*, under the user home directory, that occurs every time a successful connection to the Gnutella network is established. This file contains information about the IP address and last time the ultra peer was "seen", so the next time the application starts, it has a higher probability that it will not need to send so many search requests to obtain the available ultra peers, as some are already included in that file. Less search requests will imply less rules detected.

The third and forth rows, contain data about the traffic collected during the time when the application was already open and connected to the Gnutella network. In this period, although there was no user interaction of any kind, rule 1000261 was triggered again, more times than in the two previous tests, as this lasted longer. The most interesting fact about the last test, is that rule 1000204 was triggered 43 times even though, supposedly, all TCP traffic were being encrypted with TLS.

Two more rules were later introduced in the Gnutella rule set. Their ids are 1000265 and 1000267 and concern incoming UDP traffic for the Gtk-Gnutella application.

```
alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P GTK-Gnutella UDP -
Incoming 60 60 offset 4"; content:"|C1 88|"; depth:2; content:"|60 60|"; distance:2; depth:2;
classtype:policy-violation; sid:1000265; rev:2;)
```

Snort Rule 1000265. Rule for detection of traffic generated through GTK-Gnutella.

```
alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule UDP Gtk-Gnutella
incoming 60 60 urn:sha1"; content:"|60 60|"; offset:2; content:"urn:sha1:"; offset:31;
classtype:policy-violation; sid:1000267; rev:1;)
```

Snort Rule 1000267. Rule for detection of traffic generated through GTK-Gnutella.

With these two additional rules, more tests were conducted for accounting their occurrences. The first row is relative to the data analyzed during the application startup and search for contents, while the second is for after the connection to the Gnutella network already took place and a random episode from a successful TV car show was searched and partially downloaded. The results are presented in table 4.20.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic(MB)	Alert	Count
22-02-2009	16:54	16:58	921	159759	- - -	- - -	1000261 1000265 1000267	4 194 101
22-02-2009	17:13	21:35	128084	93203930	79.87	0	1000204 1000261 1000265 1000267	1 38 1103 571

Table 4.20: Characteristics of experiences and their detection results for GTK-Gnutella traffic with TLS encryption settings on.

Once again, rule 1000204 (for TLS tunneled TCP traffic) was triggered, being impossible to identify the causes for this behavior.

Since the beginning of the present chapter, it has been shown that Snort rules have been created in pairs for a same pattern, for testing purposes. Their distinction is based in the flow direction, if it is incoming or outgoing traffic. This was quite useful because it allowed to find the following behavior.

Until now, all the GTK-Gnutella application traffic tests were conducted in the same machine where Snort was running and **only incoming UDP** traffic was being detected. After a few days of tests and research, it was possible to identify the reason for this problem and find a workaround for it.

The first thing to be done, was to create a simple snort rule that would trigger any outgoing UDP traffic. Once again, not even once that rule was triggered for traffic generated on the Snort machine. Later, the same tests were performed, but this time running GTK-Gnutella in machines in the DPI workgroup. As shown already in 3.1, the machine were Snort was running was also the gateway for all the others using P2P software, to guarantee that all traffic would be analyzed. Using this setup, Snort could correctly identify and

trigger UDP rules (never triggered before) for outgoing traffic, unlike when GTK-Gnutella was running on the same machine as Snort.

Outgoing UDP traffic originated in the Snort machine was then analyzed and one could see that the Wireshark *Info* field contained the following message: **[UDP CHECKSUM INCORRECT]**. This verification can be unchecked in the Wireshark application menu Edit → Preferences → Protocols → UDP.

So the problem was that Snort discarded packets with bad checksums by default. If one wants to alert on packets with bad checksums, it is necessary to turn on the configuration checksums option in Snort. This was done by adding the **"-k none"** parameters to the Snort startup file */etc/init.d/snortd*.

The reason for these checksum errors, if it is on the receiving side, it is because many modern network adapter drivers offload checksum calculation to the adapter itself. If they occur on the sending side, just like in this case, it looks like every packet has a checksum error, since the driver does not calculate the checksum at all. After this moment, Snort not longer discarded packets with bad checksums, thus enabling to analyze all outgoing UDP traffic. The following rules were included.

```
alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P GTK-Gnutella
UDP - Outgoing SCPA"; content:"|60 60|"; offset:2; content:"SCPA"; offset:25; nocase;
content:"VCEGTKG"; nocase; distance:2; classtype:policy-violation; sid:1000258; rev:1;)
```

Snort Rule 1000258. Rule for detection of traffic generated through GTK-Gnutella.

```
alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P GTK-Gnutella UDP -
Outgoing 60 60 offset 4"; content:"|C1 88|"; depth:2; content:"|60 60|"; distance:2; depth:2;
classtype:policy-violation; sid:1000264; rev:2;)
```

Snort Rule 1000264. Rule for detection of traffic generated through GTK-Gnutella.

```
alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P GTK-Gnutella UDP -
Outgoing 60 60 urn:sha1"; content:"|60 60|"; offset:2; content:"urn:sha1:"; offset:31;
classtype:policy-violation; sid:1000266; rev:1;)
```

Snort Rule 1000266. Rule for detection of traffic generated through GTK-Gnutella.

Right after the inclusion of these new rules, they started to be triggered immediately, as shown in the following table 4.21. There, the first row shows the results since the GTK-Gnutella application was started, until it completed a bit more than one hundred megabytes of the download of a well known BBC automotive TV show episode. The second row contains the results for the resuming download, on which, for uncertain reasons at the moment, rule 1000204 (for TCP traffic, supposedly tunneled through TLS) was triggered once again, and with the greatest occurrence so far.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
23-02-2009	14:50	15:18	180292	138293105	111.26	0	1000258 1000261 1000264 1000265 1000266 1000267	34 10 113 306 2 193
23-02-2009	16:00	19:08	-	-	587.6	117.79	1000204 1000258 1000261 1000264 1000265	1227 78 14 174 412

Table 4.21: Characteristics of experiences and their detection results for GTK-Gnutella traffic with TLS encryption settings on.

The complete set of Snort rules created for the detection of GTK-Gnutella traffic is provided in appendices B.1 and B.4.

4.4 eDonkey

4.4.1 eMule

eMule is perhaps the most well known client for the eDonkey network. Recent versions also support the structured P2P network Kademlia, enabling eMule to reduce its server dependency and this way avoid a complete network shutdown. Once again, it is important to remind that until the present day, the original eDonkey network site at *http://www.edonkey2000.com* is still closed as consequence of a lawsuit. The Kademlia network can be enabled in Options → Connection → Network → Kad, along with the use of UDP (available by default). Disabling them can reduce traceability, but also resource availability.

The most important feature of eMule for this work, is its protocol obfuscation option under Options → Security → Protocol Obfuscation. This characteristic makes the task of detecting eMule traffic much harder, as it was previously shown in figure 2.23, page 38, but not completely impossible, according to [97]. Obfuscation details can also be found there.

"By default, each eMule client (>= 0.47b) supports obfuscated connections to other clients, but doesn't actively requests them." [74]

eMule version 0.49b was used during this work. In eMule one can use the first or both of the following settings:

- Enable protocol obfuscation
- Allow obfuscated connections only (not recommended)

The first option allows eMule to use obfuscated connections whenever possible and will ask other clients to do the same when responding. When connecting to the eDonkey network, through a server, non obfuscated connections will only be used if an attempt to

establish an obfuscated one fails. The use of this setting will slightly increase the use of CPU without any other disadvantages.

Enabling the second option will force eMule to only establish and accept obfuscated connections. Any other eDonkey client that does not use or support obfuscation will be ignored and only obfuscated connections will be allowed through automatic server connect. This setting can act both as an benefit or a disadvantage though. If most of the peers that share a pretended resource are using it and one uses it to, faster downloads will be achieved since many connections can be established. But if one uses this setting and most of the peers do not, non obfuscated connections will be ignored causing less available peers and consequently slower downloads.

Nowadays, most eMule users opt to only use obfuscated connections, as it happens for other P2P network applications already mention in this work. This way connections to the eDonkey network are harder unless this setting is not specified.

eMule Traffic Detection

Using "The eMule Protocol Specification", available at [98], it was possible do adapt the well known eDonkey and extended eDonkey (used for example by eMule and aMule) message patterns defined on that document into Snort rules. As for the Kademlia protocol used by eMule, the source code of IPP2P, available at [52], was used for the same purpose.

There is also a variation of this latter protocol called Kademlia AdunanzA (Kadu). It is part of the *eMule AdunanzA* P2P client, developed by italian programmers, to overcome some limitations with their internet connection provided by a major Italian ISP, Fastweb. To create Snort rules that allowed to identify this protocol, it was used Tstat [22] source code as a reference. Due to geographical reasons, traffic using this protocol could not be conveniently tested. Table 4.22 contains information about the rules created for the P2P protocols, the message flow, number of rules created and message structure, where "." means one byte interval and *Byte* represents one byte of many of the possible values.

P2P Protocol	Message Flow	Network Protocol	Rules	Structure
eDonkey	Client → Server	TCP	16	0xE3 . . . . Byte
	Client → Server	UDP	9	0xE3 Byte
	Client → Client	TCP	28	0xE3 . . . . Byte
Extended eDonkey	Client → Client	TCP	12	0xC5 . . . . Byte
	Client → Client	UDP	4	0xC5 . . . . Byte
Kademlia	Client → Client	UDP	36	0xE4 Byte
Kademlia AdunanzA	Client → Client	UDP	36	0xA4 Byte

Table 4.22: Pattern Structure for eDonkey, Kad and Kadu.

Although it was created a considerable amount of Snort rules for eDonkey traffic, their use is meant for non obfuscated connections. Also, the results obtained during the tests at EANTC [60] also published in InformationWeek [9] were, at least, discouraging, so the

[22] Tstat stands dor TCP Statistical and Analysis Tool. It was developed at the Telecommunication Networks Group, Politecnico di Torino, Italy [99]

number of expected triggered alerts using the patterns defined in 4.22 was quite low or even null. Nevertheless, all alerts related to the use of Kademlia network were triggered, as it does not yet support protocol obfuscation. For this reason, only the most triggered rules will be presented in this section, although the complete Snort rule set is available in appendix A.

"Obfuscation is currently available for ED2k TCP and UDP, Server TCP and UDP and Kad TCP communication. Kad UDP packets are not yet obfuscatable." [74]

Bellow are listed two Snort rules for eDonkey traffic. The first one has the id 2586 and it is included in the Snort distribution. Although it is quite generic, since only analyzes the first byte of the packet content, it was not triggered a single time, not even for non obfuscated traffic. The reason for this is that it only analyzes outgoing TCP traffic having port 4242 as destination, which is not usual nowadays, since application port numbers are randomly generated at installation time. The second rule, with id 1000001, was created for this work according to the specifications mentioned in [98] and is more specific that the first one. It is only useful when using non obfuscated connections and if it occurs out of this scenario, it is certainly a false positive. This rule was not triggered often for non eDonkey traffic, but most of the times this happened it was relative to a Windows RDC connection.

```
 alert tcp $HOME_NET any -> $EXTERNAL_NET 4242 (msg:"P2P eDonkey transfer";
flow:to_server,established; content:"|E3|"; depth:1; metadata:policy security-ips drop;
reference:url,www.kom.e-technik.tu-darmstadt.de/publications/abstracts/HB02-1.html;
classtype:policy-violation; sid:2586; rev:3;)
```

Snort Distribution Rule 2586 for eDonkey.

```
 alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P eDonkey Outbound - Login
Request"; flow:to_server,established; content:"|E3|"; depth:1; content:"|01|"; distance:4;
depth:1; classtype:policy-violation; sid:1000001; rev:1;)
```

Snort Rule 1000001. Rule for detection of traffic generated through eDonkey.

Among the many eDonkey, eMule and Kad snort rules that were created, only those with higher number of occurrences are listed bellow. The reason for this is due to the high probability of low occurrences might represent false positives. It is important to notice that the patterns on which the Snort rules reside, can also occur for other network applications, since they are not very complex by nature. One already mentioned is RDC, but false positives can also be originated by other applications that, for example, use some kind of encryption feature that would generate random alike traffic.

The following rules were triggered for eDonkey or Kad networks when obfuscation was not used. They are presented here so one can compare rules occurrences later, when dealing with using obfuscated connections.

```
alert tcp any any -> any any (msg:"LocalRule:  P2P eMule - Client to Client -
Sources Request"; content:"|C5|"; depth:1; content:"|81|"; distance:4; depth:1;
classtype:policy-violation; sid:1000065; rev:1;)
```

Snort Rule 1000065. Rule for detection of traffic generated through extended eDonkey.

```
alert tcp any any -> any any (msg:"LocalRule:  P2P eMule - Client to Client -
Secure identification"; content:"|C5|"; depth:1; content:"|87|"; distance:4; depth:1;
classtype:policy-violation; sid:1000067; rev:1;)
```

Snort Rule 1000067. Rule for detection of traffic generated through extended eDonkey.

```
alert tcp any any -> any any (msg:"LocalRule:  P2P eMule - Client to Client - Public Key";
content:"|C5|"; depth:1; content:"|85|"; distance:4; depth:1; classtype:policy-violation;
sid:1000068; rev:1;)
```

Snort Rule 1000068. Rule for detection of traffic generated through extended eDonkey.

```
alert tcp any any -> any any (msg:"LocalRule:  P2P eMule - Client to Client - Signature";
content:"|C5|"; depth:1; content:"|86|"; distance:4; depth:1; classtype:policy-violation;
sid:1000069; rev:1;)
```

Snort Rule 1000069. Rule for detection of traffic generated through extended eDonkey.

```
alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD Hello Request";
content:"|E4 10|"; depth:2; classtype:policy-violation; sid:1000088; rev:1;)
```

Snort Rule 1000088. Rule for detection of traffic generated through eDonkey (KAD).

```
alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD2 Hello Request";
content:"|E4 11|"; depth:2; classtype:policy-violation; sid:1000090; rev:1;)
```

Snort Rule 1000090. Rule for detection of traffic generated through eDoney (KAD).

```
alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD2 Request";
content:"|E4 21|"; depth:2; classtype:policy-violation; sid:1000098; rev:1;)
```

Snort Rule 1000098. Rule for detection of traffic generated through eDonkey (KAD).

The previous rules were the most triggered when not using obfuscation. Rule 1000001 appears often because of a greater difficulty to connect to the eDonkey network with this setting. For the conducted tests described in table 4.23, the appearance of rules 1000306,1000307, 1000308 and 2008581 was a surprise, since they were written for DHT BitTorrent traffic and were previously introduced in section 4.2.1. In this same table, information in the first and third rows concerns the use of eDonkey network only, while the second is relative to Kad only. No obfuscation was used.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
07-03-2009	14:15	14:23	8876	1096078	-	-	1000001 1000065 1000317	166 2 1
07-03-2009	14:31	14:33	2725	487614	-	-	1000001 1000067 1000068 1000069 1000088 1000090 1000098	13 2 2 3 3 6 18
08-03-2009	10:05	10:11	14452	1875946	-	-	1000001 1000306 1000307 1000308 2008581	486 581 287 6 1

Table 4.23: Characteristics of experiences and their detection results for eMule traffic without obfuscation.

Although rules 1000317 and 2008581 occurred only once in the previous tests, their patterns are much more complex then those for eDonkey, extended eDonkey and Kad. So, it is not likely at all that these were false positives.

After the previous tests were completed, the same rule was checked against eMule obfuscated connections. The application was configured using the already mentioned settings *Enable protocol obfuscation* and *Allow obfuscated connections only (not recommended)*, to guarantee the maximum stealthiness possible. Even though, many rules were triggered and, once again, those were mainly DHT BitTorrent traffic. Nevertheless, no *.torrent* file was ever used during the tests.

Since Kad UDP obfuscation was not yet supported, most of the rules for this traffic were triggered during the tests. To the test results do not become to extensive, due to great amount of Kad rules created, only eDonkey network support was used for the following tests. The following rules were also triggered along with all previously mentioned.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P eDonkey Outbound -
Get List of Servers"; flow:to_server,established; content:"|E3|"; depth:1; content:"|14|";
distance:4: depth:1; classtype:policy-violation; sid:1000005; rev:1;)
```

Snort Rule 1000005. Rule for detection of traffic generated through eDonkey.

```
alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P eDonkey UDP Outbound
- Status Request"; flow:to_server; content:"|E3 96|"; depth:2; classtype:policy-violation;
sid:1000019; rev:1;)
```

Snort Rule 1000019. Rule for detection of traffic generated through eDonkey.

```
alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P eDonkey UDP Inbound -
Status Response"; flow:to_client; content:"|E3 97|"; depth:2; classtype:policy-violation;
sid:1000020; rev:1;)
```

Snort Rule 1000020. Rule for detection of traffic generated through eDonkey.

```
alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P eDonkey UDP
Outbound - Server Description Request"; flow:to_server; content:"|E3 A2|"; depth:2;
classtype:policy-violation; sid:1000024; rev:1;)
```

Snort Rule 1000024. Rule for detection of traffic generated through eDonkey.

```
alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P eDonkey UDP
Inbound - Server Description Response"; flow:to_client; content:"|E3 A3|"; depth:2;
classtype:policy-violation; sid:1000025; rev:1;)
```

Snort Rule 1000025. Rule for detection of traffic generated through eDonkey.

```
alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD Request";
content:"|E4 20|"; depth:2; classtype:policy-violation; sid:1000096; rev:1;)
```

Snort Rule 1000096. Rule for detection of traffic generated through eDonkey (KAD).

Using all the created Snort rules so far, the results for the most triggered rules during the download of the documentary "Inside the Space Shuttle", are presented in table 4.24. The first test used both TCP and UDP, while in the second, UDP support was disabled but even still UDP rules were still being detected.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
08-03-2009	11:04	11:24	46138	28618596	10,83	0.13	1000019	4
							1000020	4
							1000024	4
							1000025	3
							1000090	5
							1000096	18
							1000098	12
							1000306	638
							1000307	303
							1000308	11
08-03-2009	12:01	13:37	392168	211286503	60.73	22.86	1000005	58
							1000019	29
							1000020	21
							1000024	21
							1000025	21
							1000030	3
							1000068	4
							1000306	3489
							1000307	1711
							1000308	36
							1000309	6
							1000090	4

Table 4.24: Characteristics of experiences and their detection results for eMule traffic with obfuscation.

The complete set of Snort rules created for the detection of eDonkey, extended eDonkey and Kademlia protocols, is provided in appendix A.

4.4.2 aMule

aMule is another well known multi platform eDonkey client. It was initially based on the xMule source code, which in turn was based on the lMule project, which was the first attempt to create an eMule like client to Linux. During this work, it was used aMule version 2.2.3, which has a similar interface to eMule and also allows the use protocol obfuscation and Kademlia network.

aMule Traffic Detection

The same rule set was used for both eMule and aMule. Most of the rules triggered during the tests were already introduced previously in section 4.4.1. When not using obfuscation, the triggered rules and their amount were similar to those of eMule traffic, even just with a few minutes test. The exception was rule 1000002, that was detected for the first time while using aMule.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P eDonkey Inbound - Server
Message"; flow:to_client,established; content:"|E3|"; depth:1; content:"|38|"; distance:4;
depth:1; classtype:policy-violation; sid:1000002; rev:1;)
```

Snort Rule 1000002. Rule for detection of traffic generated through eDonkey.

Table 4.25 contains information about the first two tests, when obfuscation was not used. The first concerns the use of both eDonkey and Kad networks, while the second one refers to eDonkey only. No transfer operations were being done at that time.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
07-03-2009	17:11	17:20	21383	2682904	-	-	1000001 1000002 1000005 1000306 1000307 2008581	1 3 1 195 91 1
07-03-2009	17:29	17:42	7329	1313204	-	-	1000001 1000002 1000005	46 4 46

Table 4.25: Characteristics of experiences and their detection results for aMule traffic with obfuscation.

Later, longer tests were conducted using only obfuscated connections. As with the previously tested eMule, the purpose was to account the rules triggered more often, reducing the probability o being false positives. The following rules have been triggered for the first time.

```
alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P eDonkey UDP Outbound
- Get Sources"; flow:to_server; content:"|E3 9A|"; depth:2; classtype:policy-violation;
sid:1000017; rev:1;)
```

Snort Rule 1000017. Rule for detection of traffic generated through eDonkey.

```
alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P eDonkey UDP Inbound
- Found Sources"; flow:to_client; content:"|E3 9B|"; depth:2; classtype:policy-violation;
sid:1000018; rev:1;)
```

Snort Rule 1000018. Rule for detection of traffic generated through eDonkey.

```
alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P eDonkey UDP Outbound
- Search Request(enhanced version)"; flow:to_server; content:"|E3 92|"; depth:2;
classtype:policy-violation; sid:1000021; rev:1;)
```

Snort Rule 1000021. Rule for detection of traffic generated through eDonkey.

```
alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P eDonkey UDP Outbound
- Search Request"; flow:to_server; content:"|E3 98|"; depth:2; classtype:policy-violation;
sid:1000022; rev:1;)
```

Snort Rule 1000022. Rule for detection of traffic generated through eDonkey.

```
alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P eDonkey UDP Inbound -
Search Response"; flow:to_client; content:"|E3 99|"; depth:2; classtype:policy-violation;
sid:1000023; rev:1;)
```

Snort Rule 1000023. Rule for detection of traffic generated through eDonkey.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eDonkey - Client to Client -
File Request Answer"; content:"|E3|"; depth:1; content:"|59|"; distance:4; depth:1;
classtype:policy-violation; sid:1000040; rev:1;)
```

Snort Rule 1000040. Rule for detection of traffic generated through eDonkey.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eDonkey - Client to Client
- File Status"; content:"|E3|"; depth:1; content:"|50|"; distance:4; depth:1;
classtype:policy-violation; sid:1000043; rev:1;)
```

Snort Rule 1000043. Rule for detection of traffic generated through eDonkey.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eDonkey - Client to Client - View Shared
Folder or Content Denied"; content:"|E3|"; depth:1; content:"|61|"; distance:4; depth:1;
classtype:policy-violation; sid:1000052; rev:1;)
```

Snort Rule 1000052. Rule for detection of traffic generated through eDonkey.

```
alert tcp any any -> any any (msg:"LocalRule:  P2P eMule - Client to Client - eMule Info";
content:"|C5|"; depth:1; content:"|01|"; distance:4; depth:1; classtype:policy-violation;
sid:1000060; rev:1;)
```

Snort Rule 1000060. Rule for detection of traffic generated through extended eDonkey.

```
alert tcp any any -> any any (msg:"LocalRule:P2P eMule - Client to Client -
Sources Answer"; content:"|C5|"; depth:1; content:"|82|"; distance:4; depth:1;
classtype:policy-violation; sid:1000066; rev:1;)
```

Snort Rule 1000066. Rule for detection of traffic generated through extended eDonkey.

Once again, although no *.torrent* file was ever used during the tests for aMule, rules 1000306, 1000307, 1000308 and 1000309, for DHT BitTorrent Traffic, were by far the most detected. Table 4.26 refers to two tests that used obfuscation, during the download of a well known BBC TV car show. The first one triggered more rules as it was using both TCP and UDP support. Only TCP support was enabled on the second one, but even though, just like with eMule, it is possible to see that with the exception of rule 1000005, all of them are relative to UDP traffic. So even disabling UDP support on both eMule and aMule, the fact is that even in less account, UDP rules are being triggered.

Date	Start	End	Number of Packets	Volume in Bytes	Download Traffic (MB)	Upload Traffic (MB)	Alert	Count
07-03-2009	20:22	21:27	62,881	27287782	9.68	-	1000001	1
							1000005	4
							1000017	107
							1000018	162
							1000019	158
							1000020	70
							1000021	168
							1000022	46
							1000023	166
							1000024	69
							1000025	63
							1000306	2265
							1000307	1118
							1000308	18
							1000309	8
07-03-2009	21:52	23:08	817,565	636172665	130.11	157.14	1000005	58
							1000019	167
							1000020	75
							1000040	7
							1000043	7
							1000052	6
							1000060	5
							1000066	6
							1000306	2707
							1000307	1345
							1000308	11
							1000309	8

Table 4.26: Characteristics of experiences and their detection results for aMule traffic with obfuscation.

Unlike previous studied applications for a given P2P protocol, eMule and aMule did not required specific Snort rules for each. The complete set of Snort rules created for the detection of eDonkey, extended eDonkey and Kademlia protocols, is provided in appendix A.

4.5 P2P TV

One of the most recent applications for P2P networks, is video and audio streaming in real time. These can be TV or radio channels from all over the world and also Video on Demand (VOD) contents of any kind available. A user watching a TV broadcast, for example, can act simultaneously as a receiver and a broadcaster, since transmission can be forwarded to more users requesting it, originating an overlay distribution network using the available peers. The main advantage of this type of distribution, is that they provide worldwide contents unlike the traditional broadcasts, usually geographically dependent. Some of their main characteristics are:

- Low infrastructure and maintenance cost
- Absence of physical obstacles
- Quality of Service (QoS) not guaranteed
- Less control of content distribution - When compared to traditional broadcasting

For this work, it was analyzed the traffic for three well known P2PTV applications already described in 3.7. They are: LiveStation, TVUPlayer and Goalbit.

4.5.1 Livestation

LiveStation is a United Kingdom based P2P TV application that allows users to customize their channel list according to their preferences. This can be done either by using the application GUI itself, or by accessing the LiveStation web site at [79]. To use this functionality, one must previously create a free account where these settings will be stored and later imported every time the user loads the application.

Livestation Traffic Detection

LiveStation application login mechanisms are slightly different of those of HTTP access, although they both establish a TCP connection to port 80 of a LiveStation server during authentication. Since the focus of this work is P2P traffic detection, only the application traffic was analyzed, originating rules 1000401 and 1000402 listed further bellow. These are only triggered when a response to a login request is received (mostly in XML), whether it is a positive one or not. Outgoing login requests contain encrypted username and password and the rest of the transmitted information has no short and easily identifiable records to enable an effective Snort rule, without the possible occurrence of false positives. Since the Livestation streaming traffic has to occur after the login, not much more time was dedicated to

find any traffic pattern during a transmission. Once any of the following rules are triggered, even in case of 1000402 (an unsucessful login due to a mistype, for example), certainly a user intends to briefly start receiving a transmission of some type.

```
 alert tcp $EXTERNAL_NET 80 -> $HOME_NET any (msg:"LocalRule:  P2PTV
Livestation Login Successful"; flow:from_server,established; content:"<message
xsi\:type=\"xsd\:string\">Login Successful</message>";offset:680; nocase;
classtype:policy-violation; sid:1000401; rev:2;)
```

Snort Rule 1000401. Rule for detection of traffic generated through Livestation.

```
 alert tcp $EXTERNAL_NET 80 -> $HOME_NET any (msg:"LocalRule:  P2PTV Livestation Login
Failed"; flow:from_server,established; content:"<message xsi\:type=\"xsd\:string\">Login
failed";offset:680; nocase; classtype:policy-violation; sid:1000402; rev:2;)
```

Snort Rule 1000402. Rule for detection of traffic generated through Livestation.

As one can see in the previous rules, the *offset:* parameter was set to 680. This is its highest value during this entire work and it tells snort to start looking for the content specified with *content:""*, 680 bytes from the start of a packet payload to its end.

It was not possible to determine a more precise value for this parameter, since the position of the searched string *<message xsi:type="xsd:string">Login Successful</message>* often changed during the tests between 680 and 1300 bytes. Even though, these rules triggered for every successful and unsuccessful login for LiveStation version 2.5, tested in Windows, Linux and OS X 10.4 and 10.5.

Initially, Snort was not able to trigger these rules, since, by default, it only inspected 500 bytes of a HTTP server response packet due to performance issues. It was then necessary to reconfigure the HTTP preprocessor. Some of these aspects were already mentioned in page 48. This was done by editing the main Snort configuration file */etc/snort/snort.conf*.

```
preprocessor http_inspect_server:  server default profile all ports
{ 80 8080 8180 } oversize_dir_length 300
flow_depth 1460
```

Figure 4.1: Snort HTTP Preprocessor Configuration.

"This value can be set from -1 to 1460. A value of -1 causes Snort to ignore all server side traffic for ports defined in ports. Inversely, a value of 0 causes Snort to inspect all HTTP server payloads defined in ports (note that this will likely slow down IDS performance). Values above 0 tell Snort the number of bytes to inspect in the first packet of the server response." - Official Snort Documentation, available at [4]. The set of Snort rules created for the detection of Livestation traffic is provided in appendix D.

4.5.2 TVU Player

TVU Player is one of the best well known P2P TV applications and it can be obtained at the TVU Networks site at [80]. It has worldwide channel guide, that include news, sports, movies, cartoons, music and many others, including those of broadcasting networks such as Fox News, ABC, NBC, CBS and many Asian broadcasters.

Its interface is very intuitive and allows easy channel selection through its guide and search options. In its left pane, for channel selection, there are three types of logotypes just before the channel id, its name and the country origin. These are company registered logotypes, the TVU Networks logotype and the Windows Media Player one. For the following traffic tests, only channels presenting a registered logotype (official broadcasts) or that of TVU Networks, were used. The reason for this, is that during the initial tests, traffic from channels with the Windows Media Player logo, was mostly detected as Real Time Streaming Protocol (RTSP), used by several media applications, for which some Snort rules already exist.

TVUPlayer detection

TVUPlayer traffic was analyzed using its application version 2.4.1. Once again, during most of time there was also SSH, HTTP and RDC traffic, since the tests were conducted remotely. There have been created two sets of two rules each. One set for TVUPlayer UDP traffic, the one used for content streaming and the second for TCP HTTP traffic, concerning the connection to the TVU Networks site [80]. These rules are presented bellow.

```
 alert udp $HOME_NET any <> $EXTERNAL_NET any (msg:"LocalRule:  P2PTV UDP TVU Player |00
01|"; content:"|00 01|"; offset:2; depth:2; classtype:policy-violation; sid:1000410; rev:1;)
```

Snort Rule 1000410. Rule for detection of traffic generated through TVU Player.

```
 alert udp $HOME_NET any <> $EXTERNAL_NET any (msg:"LocalRule:  P2PTV UDP TVU Player |00
02|"; content:"|00 02|"; offset:2; depth:2; classtype:policy-violation; sid:1000411; rev:1;)
```

Snort Rule 1000411. Rule for detection of traffic generated through TVU Player.

```
 alert tcp $HOME_NET any -> $EXTERNAL_NET 80 (msg:"LocalRule:
P2P TVUPplayer TCP 80 - contacting server"; content:"User-Agent:
TVUPlayer";nocase;offset:23;content:"tvunetworks.com"; within:40;
classtype:policy-violation; sid:1000420; rev:2;)
```

Snort Rule 1000420. Rule for detection of traffic generated through TVU Player.

TCP traffic rules 1000420 and 1000421 are much less triggered than those for UDP 1000410 and 1000411. Obviously, TCP is only used for establish a connection to to the application main site, which enables the download of resources such as the complete channel list, peer availability, etc.

```
 alert tcp $EXTERNAL_NET 80 -> $HOME_NET any (msg:"LocalRule:  P2P TVUPplayer TCP 80
- response from server"; content:"<PRODUCT_CODE>TVUPlayer</PRODUCT_CODE>"; nocase;
offset:200; classtype:policy-violation; sid:1000421; rev:1;)
```

Snort Rule 1000421. Rule for detection of traffic generated through TVU Player.

Once the application starts receiving a stream, it can then be forwarded to one another requesting it. There was not identified any difference in the packets payload whether if it was an incoming or outgoing stream. That is the reason why the bidirectional operator was introduced for the first time in a Snort rule. It this case, it only matters to detect the pattern independently of the flow direction. The bidirectional operator is represented as "<>", as one can see in rules 1000410 and 1000411.

When receiving a stream, the amount of data can easily achieve dozens of megabytes if a few minutes, since it is (ideally) a constant flow of information. For this reason, the tests performed did not generally take longer than five to ten minutes, because it can easily flood the Snort database quite fast, since each triggered alert produces several database operations in various tables. This was even aggravated by the method used to calculate the accuracy of the UDP rules, because all UDP traffic hat to be alerted for accounting purposes, as it will be described next.

It is important to notice, that although there were not running any more applications sending or receiving UDP traffic, it is not possible to totally control this environment, since LAN broadcasts, Universal Plug and Play (UPnP; used for directly connecting network devices) or even Multicast DNS use UDP and were many times accounted as part of the total UDP traffic. This is specially true for UPnP traffic detected in the lab, originated by other machines not involved in the DPI Workgroup, so it made sense to exclude this traffic from the total UDP universe concerning P2P.

To minimize the results imprecision, it was created a simple rule to trigger on UPnP traffic, so that it would not be accounted into the total amount of UDP traffic, since it was not being used by TVUPlayer. This also could be done for Multicast DNS traffic or even any type of UDP traffic that certainly was not being used by any P2P application, but only this most signifcant one was considered.

```
alert udp $HOME_NET any <> 239.255.255.250 any (msg:"LocalRule: udp
UPnP";classtype:policy-violation; sid:1000496; rev:1;)
```

Simple UDP rule to detect UPnP traffic

So the method used to calculate the rules accuracy is given by the formula:

$$P = \frac{C_{1000410} + C_{1000411}}{T_{UDP} - T_{UPnP}} \tag{4.1}$$

In 4.1, P denotes precision, C_{ruleid} is the total accounted triggered rules for a given rule id, T_{UDP} is the total number of UDP traffic packets and T_{UPnP} is the total number of UPnP packets.

Formula 4.1 was applied to all tests conducted with TVUPlayer application version 2.4.1. In each application session, traffic from several channels including NASA TV, CBS, Fox News, Comedy Central and ABC, just to cite a few, was analyzed and classified by Snort using the previous rules. A heterogeneous sample of the obtained results are displayed in table 4.27.

Date	Start	End	Number of Packets	Volume in Bytes	Alert % in UDP Traffic	Alert	Count
20-01-2009	16:09	16:24	1008722	395694909	0,97188	1000410 1000411	156831 159604
21-01-2009	10:26	10:30	246020	186363279	0,8916	1000410 1000411	10311 16620
26-01-2009	09:55	10:20	78178	27871345	0,9883	1000410 1000411 1000420 1000421	140305 2842 52 1
26-01-2009	11:07	11:10	97322	32023332	0,982	1000410 1000411 1000420 1000421	40654 1800 22 1
26-01-2009	11:48	12:07	793454	230630139	0,9878	1000410 1000411 1000420 1000421	337340 9174 50 1

Table 4.27: Characteristics of experiences and their detection results for TVU Player traffic.

These are only some of the tests performed with TVUPlayer for several channels. As one can see in the first and second rows of tabl 4.27, rule numbers 1000420 and 1000421 were not being triggered yet at that time, since they were developed later than those for UDP. The share of UDP traffic belonging to TVUPlayer detected with these rules tends not vary much, as long as the broadcast does not fail. This happens even if there is some packet loss causing a low reception quality. The second row in the previous table contains information in such scenario and, even though, about 89% of all UDP traffic was being accounted as TVUPlayer.

It became obvious that the task of logging such an enormous account of alerts, specially when they were generated in such a small time gap, brings up performance issues at some time, no matter what hardware is being used. To be able to efficiently detect TVU Player traffic, two additional rules based on 1000410 and 1000411 were created, considering the amount of alerts triggered in a short period of time. Thus, given a time gap of ten seconds and after some account adjustments, the Snort rules 1000412 and 100413, which replaced 1000410 and 1000411 respectively, were allowed to trigger after 500 and 70 occurrences each. This provided an enormous disk space and CPU time saving, as not so much database operations need to be done, although they were already executed in background using Barnyard for that effect, as described in 3.5.2. Rules 1000412 and 1000413 are shown bellow.

```
alert udp $HOME_NET any <> $EXTERNAL_NET any (msg:"LocalRule:  P2PTV UDP TVU Player |00
01|"; content:"|00 01|"; offset:2; depth:2; threshold:  type both, count 500, seconds 10,
track by_src; classtype:policy-violation; sid:1000412; rev:1;)
```

Snort Rule 1000412. Rule for detection of traffic generated through TVU Player.

```
alert udp $HOME_NET any <> $EXTERNAL_NET any (msg:"LocalRule:  P2PTV UDP TVU Player |00
02|"; content:"|00 02|"; offset:2; depth:2; threshold:  type both, count 70, seconds 10, track
by_src; classtype:policy-violation; sid:1000413; rev:1;)
```

Snort Rule 1000413. Rule for detection of traffic generated through TVU Player.

Using all the previously defined Snort rules for TVU Player, it was now possible to compare the previous and later alerts account in table 4.28. The experiences were conducted with a few possible values for the *threshold* setting, to find a "optimal" value that could detect the application traffic, without logging superfluous information.

Date	Start	End	Old Alert-Count	New Alert-Count	Threshold	Stream Length(s)
1-5-2009	17:41	17:43	1000410-**79880** 1000411-**2920**	1000412-**29** 1000413-**4**	500 100	10 10
1-5-2009	17:51	17:54	1000410-**30144** 1000411-**1826**	1000412-**8** 1000413-**12**	500 50	10 10
1-5-2009	18:05	18:08	1000410-**129716** 1000411-**4622**	1000412-**43** 1000413-**23**	500 70	10 10

Table 4.28: Characteristics of experiences and their detection results for TVU Player traffic, using Snort *threshold* option.

The rules presented in the "New Alert-Count" column in table 4.28 revealed themselves much more appropriate than the previous ones. They provide constant information about TVU Player traffic, but suppressing redundant information that would only overload the alert database.

Without being able to specify an exact date, a Web browser plugin became available at [80]. This allowed to watch TV on line right after the automatic installation from TVU Networks website took place. Tests conducted at the beggining of May 2009, enabled to confirm that using either this plugin, or the most recent version of TVU Player at that time (version 2.4.5.1), the Snort rules were still valid and triggering exactly as before.

It was not possible to tell if TVUPlayer 2.4.1 or 2.4.5.1 used some kind of encryption for its traffic. More tests were necessary to try to identify additional patterns or eventual key exchanges that would confirm its use. The complete set of Snort rules created for the detection of TVU Player traffic is provided in appendix E.

4.5.3 Goalbit

Of all the P2P TV applications studied in this work, Goalbit is the only available under the Gnu GPL licence. This means that the software can be freely downloaded, distributed, changed and even included in other new free programs. Due to the increasing number of proprietary P2P TV software and their acceptance between viewers, it is most likely that equivalent free software can also obtain a considerable share for this type of applications soon.

Unlike the traditional streaming methods, where the initial flow is sent from a single server, or even the initial P2P streaming technology, in which a flow is distributed through an overlay tree topology and so, available from a single peer at some time, Goalbit follows the multi-source approach. This way the stream is decomposed into several flows sent by different peers to each client. Packets are then reassembled at the destination to compose the pretended flow. This technology allows better transmission quality, wich is measured using the Pseudo-Subjective Quality Assessment (PSQA) [84], as more bandwidth is available.

Using the Goalbit application is extremely easy. It allows the visualization of four initial Uruguayan TV channels which are selected in the left pane of the application. A user can also obtain additional channels by specifying an URL or a *goalbit*[23] file. Goalbit has another interesting feature which is displaying the current number of viewers and boadcasters for a given channel, along with the download and upload bandwidth in addition to the usual availability or bitrate indicator, provided in every application of this kind.

After selecting the pretended channel, visualization occurs quickly (obviously depending on its availability) after the application sets itself to use UPnP, so it can overcome the problem of passing through the Snort and Smoothwall pcs before reaching the internet. In the visualization pane, the following message is displayed right before the content starts to be buffered: *"Trying to connect through UPnP"*

During this work, it was undoubtedly the less stable of all tested P2P TV applications, even those for which no results were achieved or included here like Octoshape or Joost. This will not be due to the fact that unlike the others it is open source application, but most likely because it is on a initial development state and so, it is not yet a mature technology.

Goalbit Traffic Detection

Goalbit version 0.4.2 was tested in Windows environments. Besides Goabit application, there was also SSH, HTTP and RDC t raffic through Snort during all the following tests.

Initial communication is done using HTTP between the application and several servers on the default TCP port 80. Just like BitTorrent, Goalbit uses tracker requests sent to TCP port 6969. Besides its requirement to initiate stream downloads, these communications can occur periodically to negotiate with newer peers and provide statistics, although it is no longer necessary for BitTorrent when the download has already started.

Goalbit GnutTLS settings are accessible under the menu Tools → Settings → Advanced → GnuTLS, but these only include *Expiration time for resumed TLS sessions* and *Number*

[23] Goalbit files have similar functions to those of torrent files. They indicate the location of the resources, along with information about the stream itself.

of resumed TLS sessions. During this work, no TLS traffic negotiation has been detected while using Goalbit and so, it was not possible to confirm if TLS is being used on the stream traffic.

Three Snort rules were initially created specifically for Goalbit traffic detection. Later, it was observed that one of them was very identical to another one already previously presented in section 4.2.2, page 102, relative to Vuze traffic. Only the one taken from [95] was maintained in the Snort ruleset and it is listed bellow as rule number 2000334. The other two rules were created from scratch and are Snort rules number 1000440 and 1000441.

```
#http://www.emergingthreats.net/rules/emerging-p2p.rules
# By Chich Thierry
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"ET P2P BitTorrent peer
sync"; flow: established; content:"|0000000d0600|"; offset: 0; depth: 6;
reference:url,bitconjurer.org/BitTorrent/protocol.html; classtype: policy-violation; sid:
2000334; rev:8;)
```

Snort Rule 2000334; Obtained from [95].

```
alert tcp $HOME_NET any <> $EXTERNAL_NET any (msg:"LocalRule: P2PTV Goalbit Protocol";
content:"|10|GoalBit protocol"; depth:17; nocase;classtype:policy-violation; sid:1000440;
rev:1;)
```

Snort Rule 1000440. Rule for detection of traffic generated through Goalbit.

```
alert tcp $HOME_NET any <> $EXTERNAL_NET any (msg:"LocalRule: P2PTV Goalbit GET
/announce"; content:"GET"; content:"/announce"; distance:1; content:"protocol=goalbit";
distance:1; content:"User-Agent:"; offset:300; content:"Goalbit"; nocase; distance:1;
nocase;classtype:policy-violation; sid:1000441; rev:1;)
```

Snort Rule 1000441. Rule for detection of traffic generated through Goalbit.

Another rule, created for BitTorrent traffic and previously presented in 4.5.3, was also being triggered from the beginning of the tests and mistakenly classified has a false positive. Only later, when it was found that Goalbit used BitTorrent protocol for media streaming, its constant triggering became obvious. Just like when using BitTorrent or Vuze, this is the less triggered rule for this protocol, as it is related to the beginning of the stream download from a given source. This rule is listed bellow.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"P2P BitTorrent outbound announce
request"; flow:to_server,established; content:"GET"; offset:0;depth:4; content:"/announce";
distance:1; content:"info_hash="; offset:4; content:"event=started";offset:4;
classtype:policy-violation; sid:1000301; rev:1;)
```

Snort Rule 1000301. Rule for detection of traffic generated through Goalbit.

As one can easily see, Snort rules 1000440 and 1000441 are quite similar to others created for BitTorrent traffic. This is even more notorious when looking specifically at rules

1000440 and 1000304. While the first one searches for the pattern **|10|Goalbit protocol** from the beginning of a packets payload until a specified limit position, the latest does that for the content **|13|BitTorrent protocol**.

Several tests were performed using the previous rules for the initially available TV channels. There were used streams from thirty to three hundred seconds, so one could compare the relation between the number of triggered alerts and the transmission times. Tests were conducted with Goalbit version 0.4.2 and during this time, no streaming uploaded occurred. It is very likely that the reason for behavior is geographical, as the only channels tested were those provided by default in Goalbit and these are based in Uruguay. For optimization reasons, it is not advisable to use a peer in Portugal to redistribute the stream back to Uruguay, where most of these channel viewers reside. Information about some of the conducted tests, for the channels Canal 10 Uruguay, Tevé Ciudad and Televisión Nacional de Uruguay, are displayed in table 4.29.

Date	Start	End	Number of Packets	Volume in Bytes	≅ Stream Length (s)	Alert	Count
01-06-2009	22:56	22:57	15620	9196814	30	1000301 2000334 1000440 1000441	1 756 26 3
01-06-2009	23:05	23:06	11172	7476244	30	1000301 2000334 1000440 1000441	1 467 16 2
03-06-2009	21:13	21:19	227264	194652737	300	1000301 2000334 1000440 1000441	1 3642 24 12
03-06-2009	21:25	21:29	125264	107174184	180	1000301 2000334 1000440 1000441	1 1399 14 8
03-06-2009	21:33	21:35	46773	38215134	60	1000301 2000334 1000440 1000441	1 505 15 4

Table 4.29: Characteristics of experiences and their detection results for Goalbit traffic.

As one can see from the previous results, Snort rule 2000334 is the most triggered one regardless the amount of traffic generated and it is related peer synchronization [24]. By the other hand, rule 1000301, related to the beginning of the stream download from a given source, is the less triggered one, only with one occurrence in each of the previous tests. This behavior is also typical when using BitTorrent clients, in which a peer announces only when it is interested in some resource just before starting to download it from a given source.

[24] Peer synchronization occurs when a P2P client requests a list of stored files from another peer.

Snort rules 1000440 and 1000441 are usually triggered proportionally to the stream length. Figure 4.2 shows another perspective on the previous results.

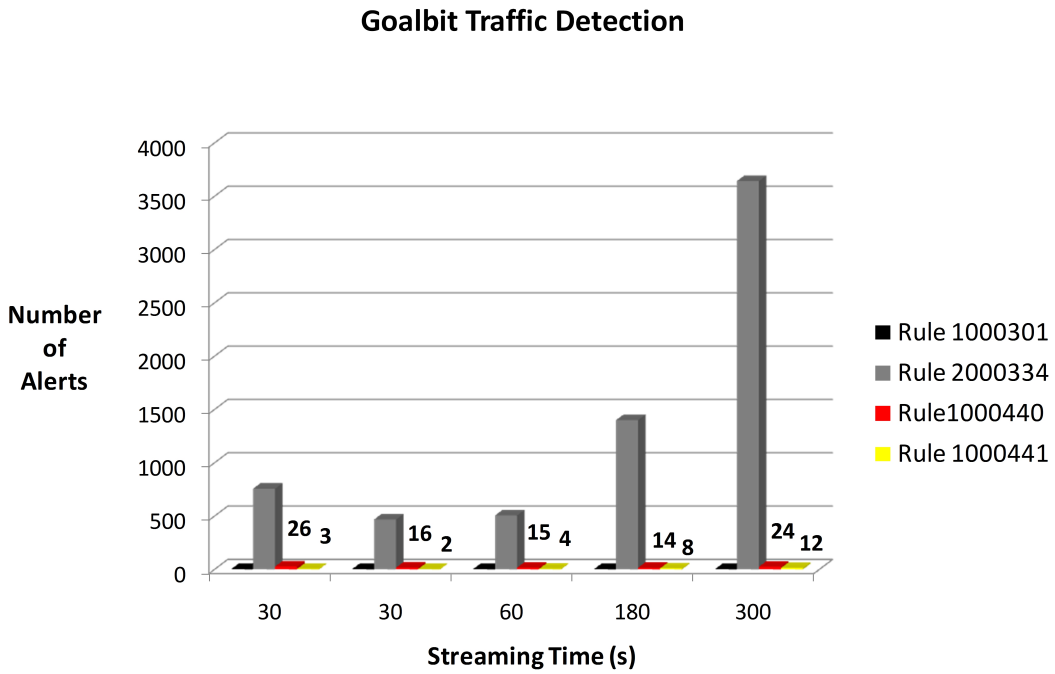

Figure 4.2: Proportion of Snort rules triggered for Goalbit traffic.

The complete set of Snort rules created for the detection of Goalbit traffic is provided in appendix F.

Chapter 5

Conclusions and Future Work

This chapter is organized in two sections. The first one shows the main conclusions achieved about the use of DPI on the detection of P2P network traffic, along with a brief resume about the amount and type of Snort rules applied for each protocol and application. The second section will be dedicated to the description of further procedures and applications, than can be used to improve the P2P detection capability by the methods used in this work and to overcome problems such as protocol encryption/obfuscation.

5.1 Conclusions

Although latest P2P applications support several methods of encryption/obfuscation, it is still possible to detect at least some of their traffic. Nevertheless, results shown in figure 2.23, translate well the difficulty of correctly classify some P2P network traffic.

In this work, most of the rules created for Snort concerned UDP traffic, as complete obfuscation is not yet fully supported for many protocols and these are becoming more frequently used as part of recent mechanisms that provide server independence. Rules for TCP traffic were still eventually triggered, even when only encrypted/obfuscated connections were allowed, but in a very small account. It is important to notice that most created TCP rules contained complex patterns, thus, hardly to be detected as false positives.

P2P applications may use slightly different protocol implementations, causing P2P rules not to be triggered in the same scenario for two P2P clients using the same protocol. This was observed when using BitTorrent and Vuze applications for the BitTorrent protocol and GTK-Gnutella and LimeWire for the Gnutella protocol. Although the tested applications are among the most well known for a given protocol, more tests were necessary to conclude if the results would be similar with other P2P software. Nevertheless, for every P2P application analyzed, its behavior was exactly the same regardless the operating system on which it was running.

The use of DPI by itself will possibly bring less results in the near future, if encryption/ofbuscation will be fully supported for both TCP and UDP traffic. The created Snort rules for P2P applications running with their encryption or obfuscation settings on, are based on the detection of some *clear* payload patterns exchanged between peers and they

will no longer work if all messages are encrypted between them. Another challenge for this approach is related with the detection of this kind of traffic under high-speed communications, in which the use of DPI mechanisms may not feasible without compromising the performance of the network.

5.1.1 BitTorrent

For BitTorrent traffic detection, either using BitTorrent client or Vuze, the use of UDP largely increases the protocol detection. When using DHT, which runs over UDP, it is possible to detect its respective outgoing and incoming communications. These are by far the most triggered rules, as they are relative to content and peer discovery.

Initially, the use of DHT in Vuze did not trigger any of previously defined rules for DHT in BitTorrent. Its protocol specification was slightly different, which caused new Snort rules to be specifically created for detecting specific Vuze DHT traffic. After discovering the *Mainline* DHT plugin for Vuze, this type of communication could be detected using exactly the same rule set as in BitTorrent.

Some traffic relative to TCP usage was still detected when using encryption, but in much less amount comparatively to UDP and only regarding an initial communication phase for each partial file download. This corresponds to rules 1000301 and 1000305, but this latest is never triggered if users uncheck the *scraping* feature, although its use allows some advantages.

Vuze allowed two encryption types: Plain and RC4. When using Plain encryption (header only), it was possible to detect four TCP rules created specifically for this purpose. These are relative to the initial communication with a peer just before the file transfer and include the *handshake* keyword.

The main conclusion for BitTorrent traffic, is that it is possible to accurately detect both TCP and UDP traffic, but mostly UDP. In the case of TCP, even using RC4 encryption, some initial messages between peers can still be possible to detect, which suggests that not all traffic is totally encrypted.

5.1.2 Gnutella

LimeWire and GTK-Gnutella were used in this work to study the P2P Gnutella protocol detection. Both support the use of TLS encryption for TCP, but even though, there were still some occurrences of the Snort rules created for this purpose. Just like BitTorrent, the greatest amount of triggered Snort rules were for UDP traffic. Its use is almost mandatory, since it is necessary for using the DHT protocol for searching and locating contents.

For LimeWire, the most triggered TCP rules were 1000201, 1000203 and 1000204. The first one, tends to be triggered in very small accounts and only when the LimeWire application connects to the Gnutella network, which can take generally less than one minute when using TLS. Detection of Gnutella UDP traffic was mostly achieved by the use of rules 1000254, 1000255, 1000256 and 1000257, relative to payloads containing the *gnutella* keyword, among with other specific patterns according to precise positions in a packet payload.

As for GTK-Gnutella using TLS encryption, rule 1000204 for TCP traffic (relative to incoming requests) was the only one triggered. None of the previously defined rules for LimeWire UDP traffic triggered even once, which suggests a completely different DHT protocol implementation. Nevertheless, the rules created specifically for GTK-Gnutella UDP (mainly rules 1000261, 1000265 and 1000267) are triggered often during any file transfer and can hardly be classified as a false-positives, due to their content complexity. For these reasons, they might be good indicators of accurate traffic detection for this application.

5.1.3 eDonkey

The identification of eDonkey traffic seemed to be the most difficult from the start, considering the studies mentioned in section 2.5.4. For its study, they were used the eMule and aMule applications.

The eDonkey, extended eDonkey and Kademlia rule set built for this this work, was undoubtedly the largest among the others. It was possible to use documentation than contained the exact patterns associated with a protocol message, to create a matching Snort rule for its detection. These rules follow a simple structure as seen in 4.22 and, therefore, can occur often as false positives for other applications. Their categories are:

- eDonkey Client/Server TCP messages
- eDonkey Client/Server UDP messages
- eDonkey Client/Client TCP messages
- Extended Client/Client TCP messages
- Extended Client/Client UDP messages
- Kademlia Client/Client UDP mesages

Similarly to other protocols, when obfuscation was not used, connection to a eDonkey server was very hard to achieve, since they mostly use this feature. Rule 1000001, relative to eDonkey network connection attempts, is the most triggered one in this scenario.

When using obfuscation, in both eMule and aMule, the most triggered rules were by far 1000306, 1000307, 1000308 and 100309. Curiously, they were created for BitTorrent DHT traffic detection but they reach the same amount of alerts as in an equivalent BitTorrent transfer. Due to a greater complexity of the patterns within these rules comparatively to the eDonkey rule set, one can claim these can hardly be false positives.

Obfuscation is not yet supported for Kademlia protocol. Although its use is optional, it allows better search mechanisms for both searching contents and nodes. For this reason, tests were mostly conducted with this feature on, just like in the majority of eDonkey client applications, thus allowing to detect every Kademlia communication.

5.1.4 P2P TV

Three P2P TV applications were used in this work. They were LiveStation, TVUPlayer and Goalbit. With the exception of Livestation, which used TCP for transmitting the media, all traffic not concerning the initial application startup is UDP, which is somehow obvious, since their goal is media streaming. Therefore, attention was mainly focused on UDP packets for traffic detection, but it was still possible to create Snort TCP rules for Livestation and TVUPlayer regarding the initial communication between the application and the network web servers for tasks such as channel list download, application version and even user login.

Livestation

It was only possible to create two TCP rules for Livestation traffic. The Livestation web site login and logout payload patterns are different from those of the Livestation application. These last can be found at a cost of higher processing, since the pretended strings occur in slightly random positions within the payload of a packet. It was necessary to configure Snort to be able to read a greater amount of a HTTP packet so it could be able to trigger on both login and logout requests. Although these rules can not be used to actually detect a media stream, they can be useful at least to detect a user intention to watch or listening to it.

All incoming streaming traffic was sent through port 80 using TCP, causing it to be mistakenly classified has HTTP traffic bye tools like Wireshark. The use of TCP for this purpose might be to guarantee transmission quality.

TVUPlayer

There were created two sets of rules for TVUPlayer traffic detection. The first one for TCP traffic, regarding initial application communication with the servers to obtain channel list and other information. These include patterns containing keywords such as *tvuplayer* and *tvunetworks* in specific packet payload positions, among with other patterns to decrease false positives probablity, which never occur for this rule set during this work. The other set is to detect streaming itself through Snort UDP rules. Initially, these rules were aimed to trigger on every TVUPlayer packet, so it would be possible to collect data about their accuracy. When results reached regularly above 98% of the total incoming and outgoing UDP traffic, these rules were modified so that they would trigger according to a specified amount of occurrences during a small period of time. This still allows to correctly identify the pretended traffic, but without logging every single packet, thus optimizing Snort and database interaction.

The rules for UDP traffic contain real simple patterns and this initially caused some false positives. Since almost all the studied P2P protocols used UDP, the hexadecimal **|00 01|** and **|00 02|** values, positioned between the second and fifth byte position in the payload, were encountered now and then when using other P2P client applications. The introduction of the previously mentioned modified rules solved this problem, as no other applications generated such a large amount of these pattern occurrences in such a small period of time.

Goalbit

Goalbit traffic detection was achieved by using two sets of two Snort rules each. The first one includes rules 1000440 and 1000441 and was specifically created for Goalbit traffic. The other contains rules 1000301 and 2000334, which were already used in sections 4.2.1 and 4.2.2. Snort rule 1000440 searches for the pattern **|10|GoalBit protocol** within the first bytes of a packet and it is very similar to the well known **|13|BitTorrent protocol** for BitTorrent traffic, from which the application is derived. As for rule 1000441, it is also very similar to others for BitTorrent and it is mainly a HTTP request containing specific Goalbit messages.

One of the rules being developed was dropped, as it was noticed that it was identical to rule 2000334, already presented for BitTorrent traffic. This is one is by far the most triggered rule when running Goalbit unlike rule 1000301 which was only triggered once in every conducted test. They are respectively relative peer synchronization and beginning of the stream download. With the exception of rule 1000301, all the others tend to be triggered in a proportional amount to the streaming time.

5.2 Future Work

Although the latest studies suggest that the P2P traffic share has lowered in the last year [1, 100], it has still an enormous impact in nowadays networks and it is predictable that it will continue to have, at least in a near future. According to studies carried by Cisco ™, P2P file sharing networks are still responsible for a 3.3 exabytes traffic volume each month [100]. Thus, P2P traffic detection (for blocking or shaping it) will probably continue, but mainly for specialized Internet hardware vendors or academic researchers, since nowadays encryption/obfuscation methods make this task harder then ever.

Briefly exposed, much more could be done concerning the topic of this dissertation. Latest P2P applications such as Vuze support the use of Proxy Servers (SOCKS V5, for example) and tests were needed to study the network traffic in those conditions. As if the detection of encrypted/obfuscated P2P traffic was not hard enough, some applications allow the use of tunneling, which consist on traffic encapsulation under another protocol. DPI allows to identify a pattern in a packet payload, regardless the TCP and UDP ports used for communication. But if one considers a given rule that will detect pretended traffic, according to a pattern specific position in a data payload, then, when using encapsulation, that position will mostly change, making the rule useless. The worst scenario involves the use of SSH. It can be used along SOCKS proxies for tunneling packets from the P2P client application towards a proxy server. This way, all P2P related traffic circulates as SSH and thus, it is virtually impossible to accurately identify any P2P traffic without applying any mechanisms to break the encryption. All the previous scenarios could also be studied, although the expected results do not seem promising.

In the opinion of the author, many of the created Snort rules could also be, at least, slightly improved. More tests are needed within a larger testbed, in order to test the accuracy of P2P traffic detection and network performance.

5.2.1 Combining DPI and Behavior Methods

Nowadays, the main challenge regarding P2P file-sharing traffic detection is concerned with on-line detection of encrypted traffic under high-speed and real-time communications, where fast P2P traffic identification is required in order to avoid network performance degradation. A possible solution to this problem may be to combine a hybrid method based on flow behavior analysis, such as the one reported in [2] and DPI. This would allow to quickly identify most of P2P traffic using flow behavior methods, so that the P2P classifier could keep up with such high-speed networks. These methods can be based on packet sizes, number of TCP and UDP ports being used simultaneously, etc. If a more precise test would be needed, then a DPI module could be dynamically called to process a given packet or flow. Such a combination would really be the best of both worlds, not only because it would reduce the amount of false negatives and false positives, but it would assure better network performance than if only DPI was used.

5.2.2 Mobile P2P

The use of mobile devices for P2P client applications can also be studied, as they are becoming more available. Nowadays, it is possible use them similarly to computers or laptops for running P2P applications for file sharing or media streaming, due to the growth of their computing capabilities.

To test the created rule set for the several P2P protocols on mobile devices, one could acquire a wireless ethernet card and use the same method as the one used in this work. All traffic to and from the mobile device should be forced to pass through Snort, via its wireless card, becoming the gateway for all existing mobile devices. Snort should also be setup to analyze traffic in this network interface using the same P2P rule set as before, to compare the P2P traffic detection accuracy in similar conditions of the tests conducted for this work.

5.2.3 Defeating Encryption

Although network hardware manufacturers such as Arbor Networks and ipoque GmbH claim that they do not use any mechanisms to break protocol encryption (see section 2.5.4, page 38), it was no possible to decrypt P2P traffic during this work. Most of the encryption methods for P2P traffic use the node (peer) id hash during the the encryption key exchange, which will cause communications between any two nodes to use a different key and so, protocol detection is even more difficult. The only mechanisms that seem to be a promising workaround for encryption, are the use of decryption modules applied to DPI. This way, encrypted P2P traffic could be decoded first and then the next step would be to analyze the *plain* content of the payload. The advantage of using such mechanisms, is that all the known protocol signatures and traffic patterns could still be used, enabling to classify an encrypted payload as if no encryption was used at all.

SSL Encryption

Recently, there has been an increasing number of companies such as SSLTech [101], which provides software packages focused on SSL decryption, mostly for network traffic originated through HTTPS. SSLTech provides both DSSL and SnortSSL and are mainly directed to HTTPS traffic. Their main features are listed bellow:

- DSSL

  Support for SSL 3.0 and TLS 1.0

  Multi-platform C library

  Built-in TCP reassembly engine

  Abstracts SSL/TLS protocol complexity

- SnortSSL

  Analyze deciphered SSL as plain TCP/IP traffic with Snort rules

  Dynamically loaded preprocessor

  Supports multiple SSL servers

Source code for both previous applications is available at SSLTech site. However, compiled binaries are only available for Windows operating systems. Since, for this work, Snort was setup and run on a Linux machine, it would be interesting to test the use of the SnortSSL preprocessor on a Windows system, using all the created rules aimed at TLS traffic for P2P Gnutella applications such as GTK-Gnutella and Limewire.

RC4 Encryption

The choice for using the RC4 algorithm in P2P protocols, such as BitTorrent, is not because it is a strong encryption algorithm, but due to its speed. It is important for P2P applications not to be overloaded with encryption/decryption tasks that might reduce the overall application performance, specially when transferring large or simultaneous multiple files.

During this work, it was not possible to find any tool or Snort module that could provide RC4 decryption. Its existence or future development, could contribute for the detection of encrypted P2P protocols such as BitTorrent.

5.2.4 Snort Inline

Latest versions of Snort allow a feature named *Inline Mode*. Instead of reading packets from libpcap, the Inline mode uses iptables for this and then allows extra functionalities to Snort like drop and reject traffic, as already described in section 3.5.1. Snort Inline also allows packet content replacement, provided that the new string and that to be replaced have the same length.

The discovery of these features came up after all the Snort, Barnyard and MySQL configurations were done. Since the testbed was stable and due some later issues regarding the study of P2P TV, it was decided not to reconfigure Snort or add another instance to it, as it

could diminish the available time to finish this work. From the documentation read at [4], the Snort Inline mode installation and configuration does not seem an extremely hard task. Nevertheless, it could be very time consuming, specially because all the previously created rules had to be modified for this mode, so that one could test if the pretended packets were blocked. If they were, it is very likely that each protocol for which snort rules were created could be blocked, as essential traffic for its operation could never reach its destination.

5.2.5 Snort Performance Measurement

Latest Snort versions, like the 2.8.3.1 used in this work, can provide useful statistics that include the total amount of received and analyzed packets, their protocol distribution, the number of alerts and logs generated and information relative to preprocessors, those which their default configurations were modified. Although these text reports look quite complete, a more careful observation allows one to conclude the lack of an important item, in my opinion. One that could provide information about the Snort rules execution time. As a future work, it would be interesting to develop a mechanism to obtain at least the medium response time between alert processing.

Nevertheless, statistics collected by Snort, as a response to its *stats* parameter, have shown that no packets were lost in the queue due to the packet inspection in all experiments (with or without obfuscation), with the exception of the average two-packet loss every time the statistics are collected, independently of the Snort load.

Bibliography

[1] Hendrik Schulze and Klaus Mochalski. Internet Study 2008/2009. Technical report, ipoque GmbH, 2009.

[2] João V. P. Gomes, Pedro R. M. Inácio, Mário M. Freire, Manuela Pereira, and Paulo P. Monteiro. Analysis of Peer-to-Peer Traffic Using a Behavioural Method Based on Entropy. In CA IEEE Computer Society Press, Los Alamitos, editor, *Proceedings of the 27th IEEE International Performance Computing and Communications Conference (IPCCC 2008), Austin, Texas, USA*, volume ISBN: 978-1-4244-3367-4, pages 201–208, December 7-9 2008.

[3] Roberto Di Pietro Angelo Spognardi, Alessandro Lucarelli. A Methodology for P2P File-Sharing Traffic Detection. In *Hot Topics in Peer-to-Peer Systems, 2005. HOT-P2P 2005. Second International Workshop on*, pages 52–61.

[4] Snort. URL: `http://www.snort.org`, last access in June 4, 2009.

[5] Mário M. Freire, David A. Carvalho, and Manuela Pereira. Detection of Encrypted Traffic in eDonkey Network Through Application Signatures. In *The First International Conference on Advances in P2P Systems. AP2PS 2009.* IARIA, October 2009.

[6] Peter H. Salus, editor. *The ARPANET Sourcebook: The Unpublished Foundations of the Internet*. Peer-to-Peer Communications, January 2008.

[7] Any Oram, editor. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly Media, Inc., February 2001.

[8] GigaNews. Newsgroups. Nonstop. Giganews Usenet History: Interview with Tom Truscott. URL: `http://www.giganews.com/usenet-history/truscott.html`, last access in June 4, 2009.

[9] Paul McDougall. InformationWeek - Business Technology News, Reviews and Blogs. URL: `http://www.informationweek.com/801/peer.htm`, last access in June 5, 2009.

[10] Beowulf Project. URL: http://www.beowulf.org, last access in June 4, 2009.

[11] Peer to Peer Working Group. URL: http://p2p.internet2.edu/, last access in June 5, 2009.

[12] Microsoft Windows Vista Help and Support. *What is Windows Meeting Space ?*, 2009.

[13] Inc. Javvin Technologies. *Network Dictionary*. Javvin Press, May 2007.

[14] Tien Tuan Anh Dinh. Security in P2P Systems. URL: http://www.cs.bham.ac.uk/~ttd/latex-beamer.pdf, last access in June 5, 2009.

[15] Fares Benayoune and Luigi Lancieri. Models of Cooperation in Peer-to-Peer Networks - A Survey. In *Third European Conference, ECUMN 2004 Porto, Portugal, October 25-27, 2004 Proceedings*, pages 327–336. Springer Berlin / Heidelberg.

[16] Gnutella Protocol Specification. URL: http://wiki.limewire.org/index.php?title=GDF\#Gnutella_Protocol_Specification, last access in June 5, 2009.

[17] edonkey. URL: http://www.edonkey2000.com, last access in June 4, 2009.

[18] BitTorrent.org. URL: http://www.bittorrent.org, last access in July 27, 2009.

[19] Sylvia Ratnasamy, Ion Stoica, and Scott Shenker. Routing Algorithms for DHTs: Some Open Questions. In *Peer-to-Peer Systems. First International Workshop, IPTPS*, pages 45–52. MIT Faculty Club, Cambridge, MA, USA, Springer Berlin / Heidelberg, Mar 2002.

[20] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM2001 Conference, San Diego, California, USA : applications, technologies, architectures, and protocols for computer communication*, pages 149–160, San Diego, California, United States, Aug 2001. ACM.

[21] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *Peer-to-Peer Systems. First International Workshop, IPTPS*, pages 53–65. MIT Faculty Club, Cambridge, MA, USA, Springer Berlin / Heidelberg, Mar 2002.

[22] Luis Rodero Merino. *Self-Adaptation Mechanisms for Efficient Resource Location in Peer-to-Peer Systems*. PhD thesis, Universidad Rey Juan Carlos, Departamento de Ingeniería Telemática y Tecnología Electrónica, 2007.

[23] The Peer to Peer Model. URL: https://www.cs.uwaterloo.ca/~iaib/cs454/notes/P2P.pdf, last access in June 5, 2009.

[24] Internet Traffic Report. URL: http://www.internettrafficreport.com, last access in June 4, 2009.

[25] The Mobile & Internet Performance Authority. URL: http://www.internetpulse.net/, last access in August 9, 2009.

[26] CAIDA - The Cooperative Association for Internet Data Analysis. URL: http://www.caida.org, last access in June 4, 2009.

[27] Ipoque. URL: http://www.ipoque.com, last access in June 5, 2009.

[28] Hendrik Schulze and Klaus Mochalski. P2P Survey 2006. Technical report, ipoque GmbH, 2006.

[29] Hendrik Schulze and Klaus Mochalski. Internet Study 2007. Technical report, ipoque GmbH, 2009.

[30] YouTube - Broadcast Yourself. URL: http://www.youtube.com, last access in August 10, 2009.

[31] MEGAUPLOAD - The leading online storage and file delivery service. URL: http://www.megaupload.com, last access in August 10, 2009.

[32] RapidShare - Easy Filehosting. URL: http://www.rapidshare.com, last access in August 10, 2009.

[33] Arbor Networks. URL: http://www.arbornetworks.com, last access in June 5, 2009.

[34] Sandvine Incorporated. URL: http://www.sandvine.com, last access in June 5, 2009.

[35] Viviane Reding. Net Neutrality and Open Networks; Towards an European Approach. URL: http://europa.eu/rapid/pressReleasesAction.do\?reference=SPEECH/08/473, last access in August 10, September 2008. European Union Conference "Network Neutrality - Implications for Innovation and Business Online".

[36] European Parliament Directory. Malcom Harbour, Chairman of the Committee on the Internal Market and Consumer Protection; European Parliament. URL: http://www.europarl.europa.eu/members/expert/committees/view.do?language=EN\&id=4538, last access in August 10, 2009.

[37] Malcom Harbour. Electronic communications networks and services, protection of privacy and consumer protection. Technical report, European Parliament, 2008.

[38] Blackout Europe - Defending the Open Internet. URL: http://blackouteurope.eu/, last access in August 10, 2009.

[39] Review of the Internet traffic management practices of Internet Service Providers; Office of the Privacy Commissioner of Canada. URL: http://www.privcom.gc.ca/information/pub/sub_crtc_090218_e.asp, last access in June 4, 2009.

[40] Comcast. URL: http://www.comcast.com, last access in June 5, 2009.

[41] Free Press. URL: http://www.freepress.net, last access in August 10, 2009.

[42] Public Knowledge. URL: http://www.publicknowledge.org, last access in August 10, 2009.

[43] Vuze. URL: http://www.vuze.com, last access in June 5, 2009.

[44] Federal Communications Commission. COMMISSION ORDERS COMCAST TO END DISCRIMINATORY NETWORK MANAGEMENT PRACTICES. URL: http://fjallfoss.fcc.gov/edocs_public/attachmatch/DOC-284286A1.pdf, last access in August 10, 2008.

[45] A. Madhukar and C. Williamson. A Longitudinal Study of P2P Traffic Classification. In *Proc. 14th IEEE Int. Symp. Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2006)*, pages 179–188. IEEE press, September 2006.

[46] Hui Liu, Wenfeng Feng, Yongfeng Huang, and Xing Li. A Peer-To-Peer Traffic Identification Method Using Machine Learning. In *International Conference on Networking, Architecture, and Storage, NAS, 29-31 July, 2007*, pages 155–160. IEEE Press, 2007.

[47] M. Soysal and E.G. Schmidt. An accurate evaluation of machine learning algorithms for flow-based p2p traffic detection. In *International Symposium on Computer and Information Sciences (ISCIS 2007)*, pages 1–6. IEEE Press, 2007.

[48] Francisco J. González-Castaño, Pedro S. Rodríguez-Hernández, Rafael P. Martínez-Álvarez, and Andrés Gómez-Tato. Support Vector Machine Detection of Peer-to-Peer Traffic in High-Performance Routers with Packet Sampling . In *Adaptive and Natural Computing Algorithms*, pages 208–217. Springer Berlin / Heidelberg, 2007.

[49] Zhong Gao, Guanming Lu, and Daquan Gu. A Novel P2P Traffic Identification Scheme Based on Support Vector Machine Fuzzy Network. In *2009 Second International Workshop on Knowledge Discovery and Data Mining (WKDD 2009)*, pages 909–912. IEEE Press, 2009.

[50] B. Raahemi, A. Kouznetsov, A. Hayajneh, and P. Rabinovitch. Classification of Peer-to-Peer traffic using incremental neural networks (Fuzzy ARTMAP. In *Canadian Conference on Electrical and Computer Engineering (CCECE 2008)*, pages 719–724. IEEE Press, 2008.

[51] IMFirewall. URL: http://www.imfirewall.com, last access in June 5, 2009.

[52] IPP2P. URL: http://www.ipp2p.org, last access in June 5, 2009.

[53] L7-Filter Application Layer Packet Classifier for Linux. URL: http://l7-filter.sourceforge.net, last access in June 5, 2009.

[54] Iptables. URL: http://www.iptables.org, last access in June 5, 2009.

[55] Arbor Networks. Deep Packet Inspection. URL: http://www.arbornetworks.com/deeppacketinspection, last access in August 11, 2009.

[56] Ipoque. PRX Traffic Manager. URL: http://www.ipoque.com/products/prx-traffic-manager, last access in August 11, 2009.

[57] Sandvine Incorporated. Policy Traffic Switch. URL: http://www.sandvine.com/products/policy_traffic_switch.asp, last access in August 11, 2009.

[58] EANTC - European Advanced Networking Test Center. URL: http://www.eantc.com, last access in June 5, 2009.

[59] Carsten Rossenhövel. Peer-to-Peer Filters: Ready for Internet Prime Time? Technical report, Internet Evolution, March 2008.

[60] EANTC - European Advanced Networking Test Center; Presentations 2006-2008. URL: http://www.eantc.com/test_reports_presentations/presentations/2006_2008.html, last access in June 4, 2009.

[61] Microsoft Corporation. Windows XP® Home Page. URL: http://www.microsoft.com/windows/windows-xp/default.aspx, last access in August 11, 2009.

[62] Barnyard - Fast Output System for Snort. URL: http://www.snort.org/dl/barnyard/, last access in June 5, 2009.

[63] NMCG - Network and Multimedia Computing Group. URL: http://floyd.di.ubi.pt/nmcg, last access in June 8, 2009.

[64] Smoothwall Open Source Project. URL http://www.smoothwall.org, last access in June 5, 2009.

[65] Smoothwall. URL: http://www.smoothwall.net, last access in June 5, 2009.

[66] BASE - Basic Analysis and Security Engine. URL: http://base.secureideas.net, last access in June 5, 2009.

[67] Wireshark. URL: http://www.wireshark.org, last access in June 4, 2009.

[68] The GNU General Public License. URL: http://www.gnu.org/licenses/licenses.html\#GPL, last access in August 11, 2009.

[69] Rafeeq Ur Rehman. *Intrusion Detection Systems with Snort: Advanced IDS Techniques Using Snort, Apache, MySQL, PHP, and ACID*. Prentice Hall, 2003.

[70] Tcpdump/Libpcap. URL: http://www.tcpdump.org, last access in June 2, 2009.

[71] The Apache Software Foundation. URL: http://www.apache.org, last access in June 4, 2009.

[72] MySQL Developer Zone. URL: http://dev.mysql.com, last access in June 5, 2009.

[73] BitTorrent. URL: http://www.bittorrent.com, last access in June 5, 2009.

[74] eMule. URL: http://www.emule-project.net, last access in June 4, 2009.

[75] aMule. URL: http://www.amule.org, last access in June 4, 2009.

[76] LimeWire. URL: http://www.limewire.com, last access in June 5, 2009.

[77] LimeWire. The Mojito DHT. URL: http://wiki.limewire.org/index.php?title=Mojito, last access in June 7, 2009.

[78] Gtk-Gnutella. URL: http://www.gtk-gnutella.sourceforge.net, last access in June 5, 2009.

[79] Livestation. URL: http://www.livestation.com, last access in June 4, 2009.

[80] TVU Networks. URL: http://www.tvunetworks.com, last access in June 5, 2009.

[81] Octoshape. URL: http://www.octoshape.com, last access in June 4, 2009.

[82] Octoshape. End User License Agreement. URL: http://www.octoshape.com/play/EULA.pdf, 2009.

[83] Goalbit. URL: http://goalbit.sourceforge.net, last access in June 8, 2009.

[84] PSQA: Pseudo-Subjective Quality Assessment. URL: http://ralyx.inria.fr/2004/Raweb/armor/uid34.html, last access in June 5, 2009.

[85] Joost. URL: http://www.joost.com, last access in August 14, 2009.

[86] Skype. URL: http://www.skype.com, last access in August 14, 2009.

[87] KaZaA. URL: http://www.kazaa.com, last access in August 14, 2009.

[88] eBay. URL: http://www.ebay.com, last access in August 14, 2009.

[89] Babelgum. URL: http://www.babelgum.com, last access in August 14, 2009.

[90] paloalto Networks. The Application Usage and Risk Report. Technical report, paloalto Networks, April 2008.

[91] Abacast Hybrid DN Solutions. URL: http://www.abacast.com, last access in August 14, 2009.

[92] Internet-Online.org. URL: http://internet-online.org/tv/, last access in August 14, 2009.

[93] ACTLab TV - Alluvium. URL: http://actlabtv.sourceforge.net/, last access in August 14, 2009.

[94] Zattoo. URL: http://www.zatoo.com, last access in August 14, 2009.

[95] Emerging Threats. URL: http://www.emergingthreats.net/rules/emerging-p2p.rules, last access in June 5, 2009.

[96] Vuze Mainline DHT Plugin. URL: http://azureus.sourceforge.net/plugin_details.php?plugin=mlDHT, last access in June 5, 2009.

[97] eMule Protocol Obfuscation. URL: http://wiki.emule-web.de/index.php/Protocol_obfuscation, last access in June 5, 2009.

[98] Yoram Kulbak and Danny Bickson. The eMule Protocol Specification, 2005. School of Computer Science and Engineering The Hebrew University of Jerusalem, Israel.

[99] Tstat - TCP Statistic and Analysis Tool. URL: http://tstat.tlc.polito.it/index.shtml, last access in March 27, 2009.

[100] Cisco ™. Cisco Visual Networking Index: Forecast and Methodology, 2008 - 2013. URL: http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360.pdf, last access in August 14, 2009.

[101] SSLTech - SSL Decryption Software. URL: http://www.ssltech.net, last access in June 5, 2009.

Appendix A

Snort rules for eDonkey

A.1 Client/Server TCP

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P eDonkey Outbound - Login
Request"; flow:to_server,established; content:"|E3|"; depth:1; content:"|01|"; distance:4;
depth:1; classtype:policy-violation; sid:1000001; rev:1;)
```

Snort Rule 1000001.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P eDonkey Inbound - Server
Message"; flow:to_client,established; content:"|E3|"; depth:1; content:"|38|"; distance:4;
depth:1; classtype:policy-violation; sid:1000002; rev:1;)
```

Snort Rule 1000002.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P eDonkey Inbound -
server accepted client"; flow:to_client,established; content:"|E3|"; depth:1; content:"|40|";
distance:4; depth:1; classtype:policy-violation; sid:1000003; rev:1;)
```

Snort Rule 1000003.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P eDonkey Outbound -
Offer Files"; flow:to_server,established; content:"|E3|"; depth:1; content:"|15|"; distance:4;
depth:1; classtype:policy-violation; sid:1000004; rev:1;)
```

Snort Rule 1000004.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P eDonkey Outbound -
Get List of Servers"; flow:to_server,established; content:"|E3|"; depth:1; content:"|14|";
distance:4: depth:1; classtype:policy-violation; sid:1000005; rev:1;)
```

Snort Rule 1000005.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P eDonkey Inbound - Server
Status "; flow:to_client,established; content:"|E3|"; depth:1; content: "|34|"; distance:4;
depth:1; classtype:policy-violation; sid:1000006; rev:1;)
```

Snort Rule 1000006.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P eDonkey Inbound -
List of Servers" ; flow:to_client,established; content:"|E3|"; depth:1; content: "|32|";
distance:4; depth:1; classtype:policy-violation; sid:1000007; rev:1;)
```

Snort Rule 1000007.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P eDonkey Inbound - Server
Identification "; flow:to_client,established; content:"|E3|"; depth:1; content: "|41|";
distance:4; depth:1; classtype:policy-violation; sid:1000008; rev:1;)
```

Snort Rule 1000008.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P eDonkey Outbound -
Search Request"; flow:to_server; content:"|E3|";depth:1; content:"|16|"; distance:4; depth:1;
classtype:policy-violation; sid:1000009; rev:1;)
```

Snort Rule 1000009.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P eDonkey Inbound - Search
Result"; flow:to_client,established; content:"|E3|"; depth:1; content: "|16|"; distance:4;
depth:1; classtype:policy-violation; sid:1000010; rev:1;)
```

Snort Rule 1000010.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P eDonkey Outbound - Get
Sources"; flow:to_server,established; content:"|E3|"; depth:1; content:"|19|"; distance:4;
depth:1; classtype:policy-violation; sid:1000011; rev:1;)
```

Snort Rule 1000011.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P eDonkey Inbound - Found
Sources"; flow:to_client,established; content:"|E3|"; depth:1; content:"|42|"; distance:4;
depth:1; classtype:policy-violation; sid:1000012; rev:1;)
```

Snort Rule 1000012.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P eDonkey Inbound -
Callback Request"; flow:to_server,established; content:"|E3|"; depth:1; content: "|1C|";
distance:4; depth:1; classtype:policy-violation; sid:1000013; rev:1;)
```

Snort Rule 1000013.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P eDonkey Inbound -
Callback Requested"; flow:to_client,established; content:"|E3|"; depth:1; content: "|35|";
distance:4; depth:1; classtype:policy-violation; sid:1000014; rev:1;)
```

Snort Rule 1000014.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P eDonkey Inbound -
Callback Failed"; flow:to_client,established; content:"|E3|"; depth:1; content: "|36|";
distance:4; depth:1; classtype:policy-violation; sid:1000015; rev:1;)
```

Snort Rule 1000015.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P eDonkey Inbound -
Message Rejected"; flow:to_client,established; content:"|E3|"; depth:1; content: "|05|";
distance:4; depth:1; classtype:policy-violation; sid:1000016; rev:1;)
```

Snort Rule 1000016.

A.2 Client/Server UDP

```
alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P eDonkey UDP Outbound -
Get Sources"; content:"|E3 9A|"; depth:2; classtype:policy-violation; sid:1000017; rev:1;)
```

Snort Rule 1000017.

```
alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P eDonkey UDP Inbound -
Found Sources"; content:"|E3 9B|"; depth:2; classtype:policy-violation; sid:1000018; rev:1;)
```

Snort Rule 1000018.

```
alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P eDonkey UDP Outbound -
Status Request"; content:"|E3 96|"; depth:2;classtype:policy-violation;sid:1000019; rev:1;)
```

Snort Rule 1000019.

```
alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P eDonkey UDP Inbound
- Status Response"; content:"|E3 97|"; depth:2; classtype:policy-violation; sid:1000020;
rev:1;)
```

Snort Rule 1000020.

```
alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P eDonkey UDP Inbound
- Status Response"; content:"|E3 97|"; depth:2; classtype:policy-violation; sid:1000020;
rev:1;)
```

Snort Rule 1000020.

```
alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P eDonkey UDP Outbound -
Search Request(enhanced version)"; content:"|E3 92|"; depth:2; classtype:policy-violation;
sid:1000021; rev:1;)
```

Snort Rule 1000021.

```
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P eDonkey UDP Outbound -
Search Request"; content:"|E3 98|"; depth:2; classtype:policy-violation; sid:1000022; rev:1;)
```

Snort Rule 1000022.

```
 alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P eDonkey UDP Inbound
- Search Response"; content:"|E3 99|"; depth:2; classtype:policy-violation; sid:1000023;
rev:1;)
```

Snort Rule 1000023.

```
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P eDonkey UDP Outbound
- Server Description Request"; content:"|E3 A2|"; depth:2; classtype:policy-violation;
sid:1000024; rev:1;)
```

Snort Rule 1000024.

```
 alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P eDonkey UDP Inbound
- Server Description Response"; content:"|E3 A3|"; depth:2; classtype:policy-violation;
sid:1000025; rev:1;)
```

Snort Rule 1000025

A.3 Client/Client TCP

```
alert tcp any any -> any any (msg:"LocalRule: P2P eDonkey - Client to Client Hello";
flow:to_server, established; content:"|E3|"; depth:1; content:"|01|"; distance:4; depth:1;
content:"16"; distance:1; classtype:policy-violation; sid:1000026; rev:1;)
```

Snort Rule 1000026.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eDonkey - Client to Client Hello - Login
Answer"; flow:to_server,established; content:"|E3|"; depth:1; content:"|4C|"; distance:4;
depth:1; classtype:policy-violation; sid:1000027; rev:1;)
```

Snort Rule 1000027.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eDonkey - Client to Client -
Sending File Part"; content:"|E3|"; depth:1; content:"|46|"; distance:4; depth:1;
classtype:policy-violation; sid:1000028; rev:1;)
```

Snort Rule 1000028.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eDonkey - Client to Client -
Request File Part"; content:"|E3|"; depth:1; content:"|47|"; distance:4; depth:1;
classtype:policy-violation; sid:1000029; rev:1;)
```

Snort Rule 1000029.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eDonkey - Client to Client
- End of Download"; content:"|E3|"; depth:1; content:"|49|"; distance:4; depth:1;
classtype:policy-violation; sid:1000030; rev:1;)
```

Snort Rule 1000030.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eDonkey - Client to Client -
Change Client ID"; content:"|E3|"; depth:1; content:"|4D|"; distance:4; depth:1;
classtype:policy-violation; sid:1000031; rev:1;)
```

Snort Rule 1000031.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eDonkey - Client to Client -
eMule Chat Message"; content:"|E3|"; depth:1; content:"|4E|"; distance:4; depth:1;
classtype:policy-violation; sid:1000032; rev:1;)
```

Snort Rule 1000032.

```
alert tcp any any -> any any (msg:"LocalRule:  P2P eDonkey - Client to Client -
Part HashSet Request"; content:"|E3|"; depth:1; content:"|51|"; distance:4; depth:1;
classtype:policy-violation; sid:1000033; rev:1;)
```

Snort Rule 1000033.

```
alert tcp any any -> any any (msg:"LocalRule:  P2P eDonkey - Client to Client -
Part HashSet Replay"; content:"|E3|"; depth:1; content:"|52|"; distance:4; depth:1;
classtype:policy-violation; sid:1000034; rev:1;)
```

Snort Rule 1000034.

```
alert tcp any any -> any any (msg:"LocalRule:  P2P eDonkey - Client to Client -
Start Upload Request"; content:"|E3|"; depth:1; content:"|54|"; distance:4; depth:1;
classtype:policy-violation; sid:1000035; rev:1;)
```

Snort Rule 1000035.

```
alert tcp any any -> any any (msg:"LocalRule:  P2P eDonkey - Client to Client -
Accept Upload Request"; content:"|E3|"; depth:1; content:"|55|"; distance:4; depth:1;
classtype:policy-violation; sid:1000036; rev:1;)
```

Snort Rule 1000036.

```
alert tcp any any -> any any (msg:"LocalRule:  P2P eDonkey - Client to Client
- Cancel Transfer"; content:"|E3|"; depth:1; content:"|56|"; distance:4; depth:1;
classtype:policy-violation; sid:1000037; rev:1;)
```

Snort Rule 1000037.

```
alert tcp any any -> any any (msg:"LocalRule:  P2P eDonkey - Client to Client -
Out of Part Requests"; content:"|E3|"; depth:1; content:"|57|"; distance:4; depth:1;
classtype:policy-violation; sid:1000038; rev:1;)
```

Snort Rule 1000038.

```
alert tcp any any -> any any (msg:"LocalRule:  P2P eDonkey - Client to Client
- File Request"; content:"|E3|"; depth:1; content:"|58|"; distance:4; depth:1;
classtype:policy-violation; sid:1000039; rev:1;)
```

Snort Rule 1000039.

```
 alert tcp any any -> any any (msg:"LocalRule:  P2P eDonkey - Client to Client -
File Request Answer"; content:"|E3|"; depth:1; content:"|59|"; distance:4; depth:1;
classtype:policy-violation; sid:1000040; rev:1;)
```

Snort Rule 1000040.

```
 alert tcp any any -> any any (msg:"LocalRule:  P2P eDonkey - Client to Client
- File Not Found"; content:"|E3|"; depth:1; content:"|48|"; distance:4; depth:1;
classtype:policy-violation; sid:1000041; rev:1;)
```

Snort Rule 1000041.

```
 alert tcp any any -> any any (msg:"LocalRule:  P2P eDonkey - Client to Client -
Requested File ID"; content:"|E3|"; depth:1; content:"|4E|"; distance:4; depth:1;
classtype:policy-violation; sid:1000042; rev:1;)
```

Snort Rule 1000042.

```
 alert tcp any any -> any any (msg:"LocalRule:  P2P eDonkey - Client to Client
- File Status"; content:"|E3|"; depth:1; content:"|50|"; distance:4; depth:1;
classtype:policy-violation; sid:1000043; rev:1;)
```

Snort Rule 1000043.

```
 alert tcp any any -> any any (msg:"LocalRule:  P2P eDonkey - Client to Client
- Change Slot"; content:"|E3|"; depth:1; content:"|5B|"; distance:4; depth:1;
classtype:policy-violation; sid:1000044; rev:1;)
```

Snort Rule 1000044.

```
 alert tcp any any -> any any (msg:"LocalRule:  P2P eDonkey - Client to Client
- Queue Rank"; content:"|E3|"; depth:1; content:"|5C|"; distance:4; depth:1;
classtype:policy-violation; sid:1000045; rev:1;)
```

Snort Rule 1000045.

```
 alert tcp any any -> any any (msg:"LocalRule:  P2P eDonkey - Client to Client -
View Shared Files"; content:"|E3|"; depth:1; content:"|4A|"; distance:4; depth:1;
classtype:policy-violation; sid:1000046; rev:1;)
```

Snort Rule 1000046.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eDonkey - Client to Client - View
Shared Files Answer"; content:"|E3|"; depth:1; content:"|4B|"; distance:4; depth:1;
classtype:policy-violation; sid:1000047; rev:1;)
```

Snort Rule 1000047.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eDonkey - Client to Client -
View Shared Folders"; content:"|E3|"; depth:1; content:"|5D|"; distance:4; depth:1;
classtype:policy-violation; sid:1000048; rev:1;)
```

Snort Rule 1000048.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eDonkey - Client to Client - View
Shared Folders Answer"; content:"|E3|"; depth:1; content:"|5F|"; distance:4; depth:1;
classtype:policy-violation; sid:1000049; rev:1;)
```

Snort Rule 1000049.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eDonkey - Client to Client - View
Shared Folder Content"; content:"|E3|"; depth:1; content:"|5E|"; distance:4; depth:1;
classtype:policy-violation; sid:1000050; rev:1;)
```

Snort Rule 1000050.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eDonkey - Client to Client - View
Shared Folder Content Answer"; content:"|E3|"; depth:1; content:"|60|"; distance:4; depth:1;
classtype:policy-violation; sid:1000051; rev:1;)
```

Snort Rule 1000051.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eDonkey - Client to Client - View Shared
Folder or Content Denied"; content:"|E3|"; depth:1; content:"|61|"; distance:4; depth:1;
classtype:policy-violation; sid:1000052; rev:1;)
```

Snort Rule 1000052.

A.4 Extended Client/Client TCP

```
alert tcp any any -> any any (msg:"LocalRule: P2P eMule - Client to Client - eMule Info";
content:"|C5|"; depth:1; content:"|01|"; distance:4; depth:1; classtype:policy-violation;
sid:1000060; rev:1;)
```

Snort Rule 1000060.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eMule - Client to Client -
eMule Info Answer"; content:"|C5|"; depth:1; content:"|02|"; distance:4; depth:1;
classtype:policy-violation; sid:1000061; rev:1;)
```

Snort Rule 1000061.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eMule - Client to Client - Sending
Compressed File Part"; content:"|C5|"; depth:1; content:"|40|"; distance:4; depth:1;
classtype:policy-violation; sid:1000062; rev:1;)
```

Snort Rule 1000062.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eMule - Client to Client
- Queue Ranking"; content:"|C5|"; depth:1; content:"|60|"; distance:4; depth:1;
classtype:policy-violation; sid:1000063; rev:1;)
```

Snort Rule 1000063.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eMule - Client to Client -
eMule File Info"; content:"|C5|"; depth:1; content:"|61|"; distance:4; depth:1;
classtype:policy-violation; sid:1000064; rev:1;)
```

Snort Rule 1000064.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eMule - Client to Client -
Sources Request"; content:"|C5|"; depth:1; content:"|81|"; distance:4; depth:1;
classtype:policy-violation; sid:1000065; rev:1;)
```

Snort Rule 1000065.

```
alert tcp any any -> any any (msg:"LocalRule: P2P eMule - Client to Client
- Sources Answer"; content:"|C5|"; depth:1; content:"|82|"; distance:4; depth:1;
classtype:policy-violation; sid:1000066; rev:1;)
```

Snort Rule 1000066.

```
 alert tcp any any -> any any (msg:"LocalRule:  P2P eMule - Client to Client -
Secure identification"; content:"|C5|"; depth:1; content:"|87|"; distance:4; depth:1;
classtype:policy-violation; sid:1000067; rev:1;)
```

Snort Rule 1000067.

```
 alert tcp any any -> any any (msg:"LocalRule:  P2P eMule - Client to Client - Public Key";
content:"|C5|"; depth:1; content:"|85|"; distance:4; depth:1; classtype:policy-violation;
sid:1000068; rev:1;)
```

Snort Rule 1000068.

```
 alert tcp any any -> any any (msg:"LocalRule:  P2P eMule - Client to Client - Signature";
content:"|C5|"; depth:1; content:"|86|"; distance:4; depth:1; classtype:policy-violation;
sid:1000069; rev:1;)
```

Snort Rule 1000069.

```
 alert tcp any any -> any any (msg:"LocalRule:  P2P eMule - Client to Client -
Preview Request"; content:"|C5|"; depth:1; content:"|90|"; distance:4; depth:1;
classtype:policy-violation; sid:1000070; rev:1;)
```

Snort Rule 1000070.

```
 alert tcp any any -> any any (msg:"LocalRule:P2P eMule - Client to Client -
Preview Answer"; content:"|C5|"; depth:1; content:"|91|"; distance:4; depth:1;
classtype:policy-violation; sid:1000071; rev:1;)
```

Snort Rule 1000071.

A.5 Extended Client/Client UDP

```
alert udp any any -> any any (msg:"LocalRule: P2P eMule UDP - Client to Client
- Re-ask File"; content:"|C5|"; depth:1; content:"|90|"; distance:4; depth:1;
classtype:policy-violation; sid:1000072; rev:1;)
```

Snort Rule 1000072.

```
alert udp any any -> any any (msg:"LocalRule: P2P eMule UDP - Client to Client - Re-ask
File Ack - it is in the queue"; content:"|C5|"; depth:1; content:"|91|"; distance:4; depth:1;
classtype:policy-violation; sid:1000073; rev:1;)
```

Snort Rule 1000073.

```
alert udp any any -> any any (msg:"LocalRule: P2P eMule UDP - Client to Client - Re-ask
File Ack - file not found"; content:"|C5|"; depth:1; content:"|92|"; distance:4; depth:1;
classtype:policy-violation; sid:1000074; rev:1;)
```

Snort Rule 1000074.

```
alert udp any any -> any any (msg:"LocalRule: P2P eMule UDP - Client to Client
- Queue Full"; content:"|C5|"; depth:1; content:"|93|"; distance:4; depth:1;
classtype:policy-violation; sid:1000075; rev:1;)
```

Snort Rule 1000075.

A.6 KAD Client/Client UDP

For Kadu (Kad AdunanzA) rules, replace “E4” by”A4”.

```
alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD Bootstrap Request";
content:"|E4 00|"; depth:2; classtype:policy-violation; sid:1000080; rev:1;)
```

Snort Rule 1000080.

```
alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD2 UDP - KAD2 Bootstrap
Request"; content:"|E4 01|"; depth:2; classtype:policy-violation; sid:1000082; rev:1;)
```

Snort Rule 1000082.

```
alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD Bootstrap Response";
content:"|E4 08|"; depth:2; classtype:policy-violation; sid:1000084; rev:1;)
```

Snort Rule 1000084.

```
alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD2 Bootstrap
Response"; content:"|E4 09|"; depth:2; classtype:policy-violation;sid:1000086; rev:1;)
```

Snort Rule 1000086.

```
alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD Hello Request";
content:"|E4 10|"; depth:2; classtype:policy-violation; sid:1000088; rev:1;)
```

Snort Rule 1000088.

```
alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD2 Hello Request";
content:"|E4 11|"; depth:2; classtype:policy-violation; sid:1000090; rev:1;)
```

Snort Rule 1000090.

```
alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD Hello Response";
content:"|E4 18|"; depth:2; classtype:policy-violation; sid:1000092; rev:1;)
```

Snort Rule 1000092.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD2 Hello Response";
content:"|E4 19|"; depth:2; classtype:policy-violation; sid:1000094; rev:1;)
```

Snort Rule 1000094.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD Request";
content:"|E4 20|"; depth:2; classtype:policy-violation; sid:1000096; rev:1;)
```

Snort Rule 1000096.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD2 Request";
content:"|E4 21|"; depth:2; classtype:policy-violation; sid:1000098; rev:1;)
```

Snort Rule 1000098.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD Response";
content:"|E4 28|"; depth:2; classtype:policy-violation; sid:1000101; rev:1;)
```

Snort Rule 1000101.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD2 Response";
content:"|E4 29|"; depth:2; classtype:policy-violation; sid:1000103; rev:1;)
```

Snort Rule 1000103.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD Search Request";
content:"|E4 30|"; depth:2; classtype:policy-violation; sid:1000105; rev:1;)
```

Snort Rule 1000105.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD Search Notes
Request"; content:"|E4 32|"; depth:2; classtype:policy-violation; sid:1000107; rev:1;)
```

Snort Rule 1000107.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD2 Search Key
Request"; content:"|E4 33|"; depth:2; classtype:policy-violation; sid:1000109; rev:1;)
```

Snort Rule 1000109.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD2 Search Source
Request"; content:"|E4 34|"; depth:2; classtype:policy-violation; sid:1000111; rev:1;)
```

Snort Rule 1000111.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD2 Search Notes
Request"; content:"|E4 35|"; depth:2; classtype:policy-violation; sid:1000113; rev:1;)
```

Snort Rule 1000113.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD Search Response";
content:"|E4 38|"; depth:2; classtype:policy-violation; sid:1000115; rev:1;)
```

Snort Rule 1000115.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD Search Notes
Response"; content:"|E4 3A|"; depth:2; classtype:policy-violation;sid:1000117; rev:1;)
```

Snort Rule 1000117.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD2 Search Response";
content:"|E4 3B|"; depth:2; classtype:policy-violation; sid:1000119; rev:1;)
```

Snort Rule 1000119.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD Publish Request";
content:"|E4 40|"; depth:2; classtype:policy-violation; sid:1000121; rev:1;)
```

Snort Rule 1000121.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD Publish Notes
Request"; content:"|E4 42|"; depth:2; classtype:policy-violation; sid:1000123; rev:1;)
```

Snort Rule 1000123.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD2 Publish Key
Request"; content:"|E4 43|"; depth:2; classtype:policy-violation; sid:1000125; rev:1;)
```

Snort Rule 1000125.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD2 Publish Source
Request"; content:"|E4 44|"; depth:2; classtype:policy-violation; sid:1000127; rev:1;)
```

Snort Rule 1000127.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD2 Publish Notes
Request"; content:"|E4 45|"; depth:2; classtype:policy-violation; sid:1000129; rev:1;)
```

Snort Rule 1000129.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD Publish Response";
content:"|E4 48|"; depth:2; classtype:policy-violation; sid:1000131; rev:1;)
```

Snort Rule 1000131.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD Publish Notes
Response"; content:"|E4 4A|"; depth:2; classtype:policy-violation; sid:1000133; rev:1;)
```

Snort Rule 1000133.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD2 Publish Response";
content:"|E4 4B|"; depth:2; classtype:policy-violation; sid:1000135; rev:1;)
```

Snort Rule 1000135.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD Firewalled Request";
content:"|E4 50|"; depth:2; classtype:policy-violation; sid:1000137; rev:1;)
```

Snort Rule 1000137.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD FindBuddy Request";
content:"|E4 51|"; depth:2; classtype:policy-violation; sid:1000139; rev:1;)
```

Snort Rule 1000139.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD CallBack Request";
content:"|E4 52|"; depth:2; classtype:policy-violation; sid:1000141; rev:1;)
```

Snort Rule 1000141.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD Firewalled
Response"; content:"|E4 58|"; depth:2; classtype:policy-violation;sid:1000143; rev:1;)
```

Snort Rule 1000143.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD Firewalled Ack
Response"; content:"|E4 59|"; depth:2; classtype:policy-violation; sid:1000145; rev:1;)
```

Snort Rule 1000145.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD FindBuddy Response";
content:"|E4 5A|"; depth:2; classtype:policy-violation; sid:1000147; rev:1;)
```

Snort Rule 1000147.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD2 Ping"; content:"|E4
60|"; depth:2; classtype:policy-violation; sid:1000149; rev:1;)
```

Snort Rule 1000149.

```
 alert udp any any -> any any (msg:"LocalRule:  P2P eMule KAD UDP - KAD2 Pong"; content:"|E4
61|"; depth:2; classtype:policy-violation; sid:1000151; rev:1;)
```

Snort Rule 1000151.

Appendix B

Snort Rules for Gnutella

B.1 General Gnutella TCP

```
 alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P GnuTella Outgoing
- Connect Request (gnutella connect)"; flow:to_server,established; content:"GNUTELLA
CONNECT/"; nocase; depth:17; classtype:policy-violation; sid:1000201; rev:2;)
```

Snort Rule 1000201.

```
 alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P GnuTella Incoming
- Connect Request (gnutella connect)"; flow:from_client,established; content:"GNUTELLA
CONNECT/";nocase; depth:18; classtype:policy-violation; sid:1000202;rev:1;)
```

Snort Rule 1000202.

B.2 LimeWire TCP

```
 alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P LimeWire Outgoing
uri-res afinada"; flow:to_server,established; content:"GET /uri-res/n2r"; nocase; depth:16;
content:"urn:sha1:"; distance:1; content:"X-Gnutella-Content-URN";nocase; offset:124;
content:"urn:sha1:"; distance:1; classtype:policy-violation; sid:1000203; rev:2;)
```

Snort Rule 1000203.

```
 alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P LimeWire Incoming
uri-res afinada"; flow:to_server,established; content:"GET /uri-res/n2r"; nocase; depth:16;
content:"urn:sha1:"; distance:1;content:"X-Gnutella-Content-URN";nocase; offset:124;
content:"urn:sha1:"; distance:1; classtype:policy-violation; sid:1000204; rev:2;)
```

Snort Rule 1000204.

```
 alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P LimeWire Outgoing -
GET request (/get/)"; flow:to_server,established; content:"GET /get/"; nocase; depth:9;
content:"X-Gnutella-"; offset:9; nocase; classtype:policy-violation; sid:1000205; rev:1;)
```

Snort Rule 1000205.

B.3 LimeWire UDP

```
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P LimeWire UDP Outgoing -
GND"; content:"GND"; nocase; depth:3; classtype:policy-violation; sid:1000250; rev:1;)
```

Snort Rule 1000250.

```
 alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P LimeWire UDP Incoming -
GND"; content:"GND"; nocase; depth:3; classtype:policy-violation; sid:1000251; rev:1;)
```

Snort Rule 1000251.

```
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P LimeWire UDP Outgoing
- Gnutella"; content:"GNUTELLA"; nocase; depth:8; classtype:policy-violation; sid:1000252;
rev:1;)
```

Snort Rule 1000252.

```
 alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P LimeWire UDP Incoming
- Gnutella"; content:"GNUTELLA"; nocase; depth:8; classtype:policy-violation; sid:1000253;
rev:1;)
```

Snort Rule 1000253.

```
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P LimeWire UDP Outgoing
uri-resA UDP"; content:"GET /uri-resA"; nocase; offset:4; content:"/n2r"; nocase;
distance:6; content:"urn:sha1:";distance:1; classtype:policy-violation; sid:1000254; rev:2;
```

Snort Rule 1000254.

```
 alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P LimeWire UDP Incoming
uri-resA UDP"; content:"GET /uri-resA"; nocase; offset:4; content:"/n2r"; nocase;
distance:6; content:"urn:sha1:";distance:1;  classtype:policy-violation; sid:1000255; rev:2;)
```

Snort Rule 1000255.

```
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P LimeWire UDP
Outgoing X-Gnutella-Content-URN UDP"; content:!"GET /uri-resA"; nocase; offset:4;
content:"X-Gnutella-Content-URN:"; nocase;offset:124; content:"urn:sha1:";distance:1;
classtype:policy-violation; sid:1000256; rev:1;)
```

Snort Rule 1000256.

```
 alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P LimeWire UDP
Incoming X-Gnutella-Content-URN UDP"; content:!"GET /uri-resA";nocase;offset:4;
content:"X-Gnutella-Content-URN:";nocase;offset:124; content:"urn:sha1:";distance:1;
classtype:policy-violation; sid:1000257; rev:1;)
```

Snort Rule 1000257.

B.4 GTK-Gnutella UDP

```
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P Gtk-Gnutella
UDP Outgoing SCPA"; content:"|60 60|";offset:2; content:"SCPA"; offset:25; nocase;
content:"VCEGTKG";nocase;distance:2; classtype:policy-violation; sid:1000258; rev:1;)
```

Snort Rule 1000258.

```
 alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P Gtk-Gnutella
UDP Incoming DHTC"; content:"|60 60|";offset:2; content:"DHTC";offset:39;nocase;
classtype:policy-violation; sid:1000261; rev:1;)
```

Snort Rule 1000261.

```
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P Gtk-Gnutella UDP
Outgoing 60 60 offset 4"; content:"|C1 88|";depth:2; content:"|60 60|";distance:2;depth:2;
classtype:policy-violation; sid:1000264; rev:2;)
```

Snort Rule 1000264.

```
 alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P Gtk-Gnutella UDP
Incoming 60 60 offset 4"; content:"|C1 88|";depth:2; content:"|60 60|";distance:2;depth:2;
classtype:policy-violation; sid:1000265; rev:2;)
```

Snort Rule 1000265.

```
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P Gtk-Gnutella UDP
Outgoing 60 60 urn:sha1"; content:"|60 60|";offset:2; content:"urn:sha1:";offset:31;
classtype:policy-violation; sid:1000266; rev:1;)
```

Snort Rule 1000266.

```
 alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P Gtk-Gnutella UDP
Incoming 60 60 urn:sha1"; content:"|60 60|";offset:2; content:"urn:sha1:";offset:31;
classtype:policy-violation; sid:1000267; rev:1;)
```

Snort Rule 1000267.

Appendix C

Snort Rules for BitTorrent

C.1 General BitTorrent TCP

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P BitTorrent Outgoing
announce request"; flow:to_server,established; content:"GET"; offset:0;depth:4;
content:"/announce"; distance:1; content:"info_hash="; offset:4; content:"event=started";
offset:4; classtype:policy-violation; sid:1000301; rev:1;)
```

Snort Rule 1000301.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P BitTorrent Incoming
announce request"; flow:from_client,established; content:"GET"; offset:0; depth:4;
content:"/announce"; distance:1; content:"info_hash="; offset:4; content:"event=started";
offset:4; classtype:policy-violation; sid:1000302; rev:1;)
```

Snort Rule 1000302.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P BitTorrent Incoming data
transfer"; flow:to_server,established; content:"|13|BitTorrent protocol"; offset:0; depth:20;
classtype:policy-violation; sid:1000303; rev:1;)
```

Snort Rule 1000303.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule: P2P BitTorrent Outgoing
data transfer"; flow:from_client,established; content:"|13|BitTorrent protocol"; offset:0;
depth:20; classtype:policy-violation; sid:1000304; rev:1;)
```

Snort Rule 1000304.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule: P2P BitTorrent Outgoing
- tracker request"; flow:to_server,established; content:"GET"; offset:0;depth:4;
content:"/scrape"; distance:1; content:"info_hash="; offset:12; content:"User-Agent:";
offset:80;classtype:policy-violation; sid:1000305; rev:1;)
```

Snort Rule 1000305.

C.2 Vuze Plain Encryption TCP

```
 alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P Vuze Plain Encryption
Outgoing BitTorrent_Handshake"; flow:to_server; content:":BT_HANDSHAKE3:";nocase;
classtype:policy-violation; sid:1000314; rev:2;)
```

Snort Rule 1000314.

```
 alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P Vuze Plain Encryption
Incoming BitTorrent_Handshake"; flow:to_server; content:":BT_HANDSHAKE3:";nocase;
classtype:policy-violation; sid:1000315; rev:2;)
```

Snort Rule 1000315.

```
 alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P Vuze Plain
Encryption Outgoing Azureus_Handshake"; flow:to_server; content:"AZ_HANDSHAKE";
offset:8;depth:12;nocase;classtype:policy-violation; sid:1000316; rev:1;)
```

Snort Rule 1000316.

```
 alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P Vuze Plain
Encryption Incoming Azureus_Handshake"; flow:to_server; content:"AZ_HANDSHAKE";
offset:8;depth:12;nocase; classtype:policy-violation; sid:1000317; rev:1;)
```

Snort Rule 1000317.

C.3 External TCP Rules

By Chich Thierry, http://www.emergingthreats.net/rules/emerging-p2p.rules

```
 alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"ET P2P BitTorrent peer
sync"; flow:  established; content:"|0000000d0600|"; offset:  0; depth:  6;
reference:url,bitconjurer.org/BitTorrent/protocol.html; classtype:  policy-violation; sid:
2000334; rev:8;)
```

Snort Rule 2000334.

```
 alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"ET P2P BitTorrent Traffic";
flow:  established; content:"|0000400907000000|"; offset:  0; depth:  8;
reference:url,bitconjurer.org/BitTorrent/protocol.html; classtype:  policy-violation; sid:
2000357; rev:4;)
```

Snort Rule 2000357.

C.4 General BitTorrent UDP

```
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P BitTorrent UDP -
Outgoing DHT for trackerless comunication request (d1:ad2:id20)"; content:"d1:ad2:id20";
nocase; depth:11; classtype:  policy-violation; sid:1000306; rev:2;)
```

Snort Rule 1000306.

```
 alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P BitTorrent UDP -
Incoming DHT for trackerless comunication Response (d1:rd2:id20)"; content:"d1:rd2:id20";
depth:11; nocase; classtype:policy-violation; sid:1000307; rev:3;)
```

Snort Rule 1000307.

```
 alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P BitTorrent UDP -
Incoming DHT for trackerless comunication request (d1:ad2:id20)"; content:"d1:ad2:id20";
nocase; depth:11; classtype:policy-violation; sid:1000308; rev:3;)
```

Snort Rule 1000308.

```
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P BitTorrent UDP -
Outgoing DHT for trackerless comunication Response (d1:rd2:id20)"; content:"d1:rd2:id20";
nocase; depth:11; classtype:policy-violation; sid:1000309; rev:3;)
```

Snort Rule 1000309.

C.5 Vuze UDP

```
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P Vuze UDP - Outgoing DHT
"; content:"d1:c0:1:n0:1"; nocase; classtype:policy-violation; sid:1000310; rev:2;)
```

Snort Rule 1000310.

```
 alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"LocalRule:  P2P Vuze UDP - Incoming DHT
"; content:"d1:c0:1:n0:1"; nocase;classtype:policy-violation; sid:1000311; rev:2;)
```

Snort Rule 1000311.

C.6 External UDP Rules

By David Bianco, http://www.emergingthreats.net/rules/emerging-p2p.rules

```
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"ET P2P BitTorrent DHT
ping request"; content:"d1\:ad2\:id20\:"; depth:12; nocase; threshold:
type both, count 1, seconds 300, track by_src; classtype:policy-violation;
reference:url,wiki.theory.org/BitTorrentDraftDHTProtocol; sid:2008581; rev:1;)
```

Snort Rule 2008581.

```
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"ET P2P BitTorrent DHT find_node
request"; content:"d1\:ad2\:id20\:"; nocase; depth:12; content:"6\:target20\:";
nocase; distance:20; depth:11; content:"e1\:q9\:find_node1\:"; nocase; distance:20;
depth:17; content:"e1\:q9\:find_node1\:"; distance:20; depth:17; nocase; threshold:
type both, count 1, seconds 300, track by_src; classtype:policy-violation;
reference:url,wiki.theory.org/BitTorrentDraftDHTProtocol; sid:2008582; rev:1;)
```

Snort Rule 2008582.

```
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"ET P2P BitTorrent DHT nodes
reply"; content:"d1\:rd2\:id20\:"; nocase; depth:12; content:"5\:nodes"; nocase;
distance:20; depth:7; threshold:  type both, count 1, seconds 300, track by_src;
classtype:policy-violation; reference:url,wiki.theory.org/BitTorrentDraftDHTProtocol;
sid:2008583; rev:1;)
```

Snort Rule 2008583.

```
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"ET P2P BitTorrent DHT get_peers
request"; content:"d1\:ad2\:id20\:"; nocase; depth:12; content:"9\:info_hash20\:"; nocase;
distance:20; depth:14; content:"e1\:q9\:get_peers1\:"; nocase; distance:20; depth:17;
threshold:  type both, count 1, seconds 300, track by_src; classtype:policy-violation;
reference:url,wiki.theory.org/BitTorrentDraftDHTProtocol; sid:2008584; rev:1;)
```

Snort Rule 2008584.

```
 alert udp $HOME_NET any -> $EXTERNAL_NET any (msg:"ET P2P BitTorrent DHT
announce_peers request"; content:"d1\:ad2\:id20\:"; nocase; distance:20;
depth:14; content:"e1\:q13\:announce_peer1\:"; nocase; distance:55; threshold:
type both, count 1, seconds 300, track by_src; classtype:policy-violation;
reference:url,wiki.theory.org/BitTorrentDraftDHTProtocol; sid:2008585; rev:1;)
```

Snort Rule 2008585.

Appendix D

Snort Rules for Livestation

```
 alert tcp $EXTERNAL_NET 80 -> $HOME_NET any (msg:"LocalRule:  P2PTV Livestation Login
Successful"; flow:from_server,established; content:"<message xsi\:type=\"xsd\:string\"
>Login Successful</message>";offset:680; nocase; classtype:policy-violation; sid:1000401;
rev:2;)
```

Snort Rule 1000401.

```
 alert tcp $EXTERNAL_NET 80 -> $HOME_NET any (msg:"LocalRule:  P2PTV Livestation Login
Failed"; flow:from_server,established; content:"<message xsi\:type=\"xsd\:string\">Login
failed";offset:680; nocase; classtype:policy-violation; sid:1000402; rev:2;)
```

Snort Rule 1000402.

Appendix E

Snort Rules for TVU Player

E.1 TVU Player UDP

```
alert udp $HOME_NET any <> $EXTERNAL_NET any (msg:"LocalRule:  P2PTV UDP TVU Player |00
01|"; content:"|00 01|"; offset:2; depth:2; classtype:policy-violation; sid:1000410; rev:1;)
```

Snort Rule 1000410.

```
alert udp $HOME_NET any <> $EXTERNAL_NET any (msg:"LocalRule:  P2PTV UDP TVU Player |00
01|"; content:"|00 02|"; offset:2; depth:2; classtype:policy-violation; sid:1000411; rev:1;)
```

Snort Rule 1000411.

```
alert udp $HOME_NET any <> $EXTERNAL_NET any (msg:"LocalRule:  P2PTV UDP TVU Player |00
01|"; content:"|00 01|"; offset:2; depth:2; threshold:  type both, count 500, seconds 10,
track by_src; classtype:policy-violation; sid:1000412; rev:1;)
```

Snort Rule 1000412.

```
alert udp $HOME_NET any <> $EXTERNAL_NET any (msg:"LocalRule:  P2PTV UDP TVU Player |00
02|"; content:"|00 02|"; offset:2; depth:2; threshold:  type both, count 70, seconds 10, track
by_src; classtype:policy-violation; sid:1000413; rev:1;)
```

Snort Rule 1000413.

E.2 TVU Player TCP

```
 alert tcp $HOME_NET any -> $EXTERNAL_NET 80 (msg:"LocalRule:  P2P TVUPplayer
TCP 80 - contacting server"; content:"User-Agent:  TVUPlayer"; nocase; offset:23;
content:"tvunetworks.com";within:40; classtype:policy-violation; sid:1000420; rev:2;)
```

Snort Rule 1000420.

```
 alert tcp $EXTERNAL_NET 80 -> $HOME_NET any (msg:"LocalRule:  P2P TVUPplayer TCP 80
- response from server"; content:"<PRODUCT_CODE>TVUPlayer</PRODUCT_CODE>"; nocase;
offset:200; classtype:policy-violation; sid:1000421; rev:1;)
```

Snort Rule 1000421.

Appendix F

Snort Rules for Goalbit

F.1 Goabit Protocol

```
alert tcp $HOME_NET any <> $EXTERNAL_NET any (msg:"LocalRule:  P2PTV Goalbit Protocol";
content:"|10|GoalBit protocol"; depth:17; nocase;classtype:policy-violation; sid:1000440;
rev:1;)
```

Snort Rule 1000440.

```
alert tcp $HOME_NET any <> $EXTERNAL_NET any (msg:"LocalRule:  P2PTV Goalbit GET
/announce"; content:"GET"; content:"/announce"; distance:1; content:"protocol=goalbit";
distance:1; content:"User-Agent:"; offset:300; content:"Goalbit"; nocase; distance:1;
nocase;classtype:policy-violation; sid:1000441; rev:1;)
```

Snort Rule 1000441.

F.2 Goalbit - BitTorrent

Already listed for BitTorrent protocol.

```
 alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"LocalRule:  P2P BitTorrent
Outgoing announce request"; flow:to_server,established; content:"GET";
offset:0;depth:4; content:"/announce"; distance:1; content:"info_hash="; offset:4;
content:"event=started";offset:4; classtype:policy-violation; sid:1000301; rev:1;)
```

Snort Rule 1000301.

```
#http://www.emergingthreats.net/rules/emerging-p2p.rules
# By Chich Thierry
 alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"ET P2P BitTorrent peer
sync"; flow:  established; content:"|0000000d0600|"; offset:  0; depth:  6;
reference:url,bitconjurer.org/BitTorrent/protocol.html; classtype:  policy-violation; sid:
2000334; rev:8;)
```

Snort Rule 2000334; Obtained from [95].