TECHNICAL UNIVERSITY OF KOŠICE
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS

Novel approaches to P2P traffic optimization

Ivan KLIMEK

DIPLOMA THESIS

2010

TECHNICAL UNIVERSITY OF KOŠICE
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS
Department of Computers and Informatics

Novel approaches to P2P traffic optimization

DIPLOMA THESIS

Ivan Klimek

Supervisor: Associate Professor Frantisek Jakab, PhD.

Consultant: Associate Professor Frantisek Jakab, PhD.

Košice 2010

Metadata Sheet

Author:	Ivan Klimek
Thesis title:	Novel approaches to P2P traffic optimization
Language:	English
Type of Thesis:	Diploma Thesis
Number of Pages:	88
Degree:	Engineer
University:	Technical University of Košice
Faculty:	Faculty of Electrical Engineering and Informatics (FEI)
Department:	Department of Computers and Informatics (DCI)
Study Specialization:	Informatics
Study Programme:	Informatics
Town:	Košice
Supervisor:	Associate Professor František Jakab, PhD.
Consultant:	Associate Professor František Jakab, PhD.
Date of submission:	7. 5. 2010
Date of defence:	27. 5. 2010
Keywords:	Peer-to-Peer Networks, Optimization, Man-in-the-middle, Deep packet inspection, Redundancy, Botnets
Category Conspectus:	(In Slovak only) Výpočtová technika; Počítačové siete
Thesis Citation:	Ivan Klimek: Novel approaches to P2P traffic optimization. Diploma Thesis. Košice: Technical University of Košice, Faculty of Electrical Engineering and Informatics. 2010. 88 pages
Title SK:	Nové prístupy k optimalizácii P2P dátových prenosov
Keywords SK:	Peer-to-Peer siete, Optimalizácia, Man-in-the-middle, Deep packet inspection, Redundancia, Botnety

Abstract in English

This thesis instead of focusing on all the different Internet's main problems, aims at the single factor they have all in common. Redundancy can be found everywhere; but it is especially visible in Peer-to-Peer networks. Using calculations based on global data, clear and undeniable statistical evidence of extreme redundancy rates will be presented. Protocols used by millions cannot be easily replaced; therefore a transparent optimization approach utilizing automated hacking techniques was developed. These techniques enable to monitor the network traffic and optimize it in real time. Redundancy is not only about unnecessary data - Spam, Malware, Denial-of-Service attacks or even Botnet coordination; all of them can be identified by redundant patterns they generate. To address these threats a novel 3 layer fallback system called Protect/Warn/Disable was created.

Abstract in Slovak

Táto práca sa nesnaží riešit rôzne problémy Internetu, zameriava sa však na ich hlavný spoločný faktor. Redundanciu môžme nájsť všade, avšak práve v Peer-to-Peer sieťach je najviditeľnejšia. Použítím výpočtov podložených globálnymi dátami, budú prezentované jasné a nevyvrátitelné dôkazy o extrémnych úrovniach redundancie. Protokoly s miliónmi aktívnych používateľov sa nedajú jednoducho nahradiť, preto bol vytvorený transparentný optimalizačný postup využívajúci automatizované útočné techniky. Tieto techniky dovolujú monitorovať dátový tok a v reálnom čase ho optimalizovať. Redundancia nie je iba o nepotrebných dátach - Spam, Malware, Denial-of-Service útoky alebo aj Botnet koordinácia, všetky môžu byť identifikované redundantnými znakmi ktoré generujú. Na zamedzenie týchto hrozieb bol vyvinutý nový 3 vrstvový systém nazvaný Protect/Warn/Disable.

TECHNICAL UNIVERSITY OF KOŠICE

Faculty of Electrical Engineering and Informatics

Department of Computers and Informatics

DIPLOMA THESIS

Student: **Ivan Klimek, Bc.**

Branch of study: **Informatics**

Academic year: 2009/2010

Name of the thesis in English and in Slovak:

Novel approaches to P2P traffic optimization

Nové prístupy k optimalizácii P2P dátových prenosov

Instructions for elaboration:

1. Theoretical analysis of Peer-to-Peer networks
2. Propose novel optimization approaches
3. Experimentally determine the functionality and effectivity of the proposed approaches
4. Analyze possible extensions and different usage scenarios of the proposed approaches
5. Write documentation according to standards of the department

Diploma thesis supervisor: Doc. Ing. František Jakab, PhD.

Diploma thesis consultant: Doc. Ing. František Jakab, PhD.

Deadline for submission of the diploma thesis: 7.5.2010

prof. Ing. Ján Kollár, CSc.
head of the department

TECHNICKÁ UNIVERZITA V KOŠICIACH FAKULTA ELEKTROTECHNIKY A INFORMATIKY -11-

prof. Ing. Liberios Vokorokos, PhD.
dean of the faculty

In Košice on 31.10.2009

Declaration

I hereby declare that this thesis is my own work and effort. Where other sources of information have been used, they have been acknowledged.

Košice, 7. 5. 2010

....

signature

Acknowledgement

In the first place I would like to thank Associate Professor František Jakab, PhD. for his constant support and valuable advices.

I am glad that my dream of a specialized P2P workgroup came true during the work on this thesis. We have created a strong team that doesn't need to be afraid of any technical challenge.

One of the most important things in everyone's life is family, mine provided me everything they could and I am and always will be thankful for that.

Thank you.

Preface

Peer-to-Peer (P2P) networks are a fascinating phenomenon, no other technology ever before affected the Internet so greatly. Not only due to its impact on the network - massive increase of Internet's traffic - but also because of the sociological evolution they caused. Before P2P, Internet was used primarily as a communication and information sharing tool, nowadays most of the Internet-enabled population knows and uses it as the primary tool for accessing multimedia content. Disregarding the possibilities and the mainstream acceptance of P2P, it is still understood as something negative. Either because of the copyrighted material that it uses to distribute or because of the high network usage it generates. Internet Service Providers tend to shape P2P traffic or even block it completely, the reasons why they do this is to save resources and/or costs. But there is another solution that addresses the problem cause directly and not only its symptoms. This thesis will show that redundancy is the primary problem and that it can be almost completely avoided thru transparent and cost effective dynamic caching of P2P content. Further, it will be shown that the presented approach enables total user anonymity and by that protects user privacy. During the work on this thesis it was discovered that redundancy is the common cause of most of the contemporary Internet's main problems as spam, malware or even botnets. They all generate some form of measurable redundancy that distinguishes them from human generated traffic. The last part of this thesis presents how the techniques developed for transparent P2P caching can be used to fight against the mentioned problems by completely novel means.

There are no great men, only great challenges that ordinary men are forced by circumstances to meet. (Admiral William Halsey)

Table of Contents

List of Figures **10**
List of Symbols and Abbreviations **11**
List of Terms **14**
INTRODUCTION **15**
1 BITTORRENT ARCHITECTURE **16**
1.1. Torrent (MetaInfo) Format 18
1.1.1 Bencoding 18
1.1.2 Torrent (MetaInfo) File Structure 19
1.2 Tracker HTTP/HTTPS Protocol 21
1.3 Tracker 'scrape' Convention 26
1.4 Peer wire protocol 28
1.4.1 Handshake 29
1.4.2 Messages 30
1.5 Distributed Hash Table 32
1.5.1 Overview 32
1.5.2 Routing Table 33
1.5.3 BitTorrent Protocol Extension 33
1.5.4 Torrent File Extensions 33
1.5.5 KRPC Protocol 34
1.5.6 DHT Queries 35
1.6 Local Peer Discovery 39
1.7 Magnet link 39
2. P2P POPULARITY DISTRIBUTION **40**
3. CONTROLING THE FLOW **43**
3.1 Client-Tracker Communication 44
3.2 Distributed Hash Table 45
3.3 Local Peer Discovery 46
4. PEER-TO-PEER PROXY/CACHE **47**
4.1. Intercepting Tracker HTTP/HTTPS protocol 49
4.2. The Man-in-the-middle Attack 50
4.3. The logic behind 51

4.4. Finding content 52
4.5. HW requirements 55
4.6. Software architecture – first prototype 56
4.7. Software architecture – current state 57
4.7.1 OpenSolaris 58
4.7.2 Automated interception/recognition/MiTM engine 58
4.7.3 The Downloader 59
4.7.4 Support massive parallelism 60
4.7.5 Proxy/Cache Communication Protocol – PCP 61
4.7.6 Proxy/Cache Database 63
4.7.7 Data Visualization and Analysis GUI 63
4.7.8 Coordinator 64
4.7.9 Analyzer 64
5. PERFORMANCE ANALYSIS 65
5.1 The Interceptor 65
5.2 Download performance 66
5.3 Upload performance 67
5.4 Test summary 68
6 LEGAL ISSUES 69
7 OTHER POSSIBLE ADDITIONS 70
7.1 Flash/generic HTTP proxy/caching 70
7.2 Cleaning the Internet from Botnets 72
7.2.1 Protect the user from becoming infected 73
7.2.2 Recognize and block the Botnet coordination 75
7.2.3 Block spam and DDoS at its source 77
7.2.4 Prototype architecture 78
Conclusion 84
Bibliography 85
Appendices 88

List of Figures

Fig. 1 Protocol type distribution ... 15
Fig. 2 P2P protocol distribution by volume ... 15
Fig. 3 The publishing problem ... 16
Fig. 4 The BitTorrent solution ... 17
Fig. 5 BitTorrent protocol architecture ... 17
Fig. 6 Distribution of torrents based on number of peers ... 40
Fig. 7 Detailed distribution with logarithmic vertical axis ... 40
Fig. 8 Traffic generated per torrent ... 41
Fig. 9 Total traffic generated per torrents ... 41
Fig. 10 Cache size ... 42
Fig. 11 Out-of-Band vs. In-Band ... 45
Fig. 12 Example P2P Proxy/Cache topology ... 48
Fig. 13 Example of a Hierarchical P2P Proxy/Cache topology ... 48
Fig. 14 Man-in-the-middle attack ... 50
Fig. 15 Original prototype simplified application logic ... 54
Fig. 16 Proposed storage subsystem ... 55
Fig. 17 Simplified first prototype P2P Proxy/Cache software architecture ... 57
Fig. 18 Schematic view of the current software architecture ... 58
Fig. 19 Interceptor schematic overview ... 59
Fig. 20 CUDA ... 60
Fig. 21 Example of Proxy/Cache data visualization ... 63
Fig. 22 Core logic ... 64
Fig. 23 HTTP requests per minute ... 65
Fig. 24 BitTorrent tracker requests ... 65
Fig. 25 Proxy/Cache Downloader vs. uTorrent ... 66
Fig. 26 Download performance ... 67
Fig. 27 Upload performance ... 67
Fig. 28 Phishing example ... 77
Fig. 29 AV miss rate on drive-by downloads ... 80
Fig. 30 Applications targeted by drive-by exploits ... 81
Fig. 31 BotHunter infection dialog set ... 82
Fig. 32 BotHunter Infection Lifecycle ... 82
Fig. 33 Protect/Warn/Disable schematic overview ... 83

List of Symbols and Abbreviations

API	Application Programming Interface
AV	Anti Virus
C&C	Command and Control
CA	Certificate Authority
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DDoS	Distributed Denial of Service
DHT	Distributed Hash Table
DLL	Dynamic-link Library
DMCA	Digital Millennium Copyright Act
DNS	Domain Name System
DNSSEC	Domain Name System Security Extensions
DoS	Denial of Service
EXE	Executable
FTP	File Transfer Protocol
FUP	Fair Use Policy
GB	Giga Byte
Gbps	Giga Bits per Second
GIF	Graphics Interchange Format
GLD	Generic LAN Driver
GNU	GNU's Not Unix! (recursive acronym)
GPGPU	General-purpose Graphics Processing Unit
GPL	General Public License
GUI	Graphic User Interface
HDD	Hard Disk Drive
HTTP	Hypertext Transport Protocol
HTTPS	Hypertext Transport Protocol Secure
I/O	Input Output

ICO	Microsoft Windows Icons Image Format
ID	Identifier
IDS	Intrusion Detection System
IE	Internet Explorer
IM	Instant Messaging
IP	Internet Protocol
IPRED	Intellectual Property Rights Enforcement Directive 2004/48/EC
IPS	Intrusion Prevention System
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IRC	Internet Relay Chat
ISP	Internet Service Provider
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
KB	Kilo Byte
KRPC	Kademlia Remote Procedure Call
LAN	Local Area Network
MB	Mega Byte
Mbps	Mega Bits per Second
MD5	Message-Digest algorithm 5
MiTM	Man-in-the-middle
NAT	Network Address Translation
NIC	Network Interface Card
OS	Operating System
P2P	Peer-to-Peer
PCP	Proxy/Cache Communication Protocol
PDF	Portable Document Format
PE	Portable Executable
PNG	Portable Network Graphics

PWD	Protect/Warn/Disable
QC	Quad Core
RAM	Random Access Memory
Regexp	Regular expression
RFC	Request for Comments
RPC	Remote Procedure Call
RPM	Revolutions per Minute
SHA	Secure Hash Algorithm
SMTP	Simple Mail Transport Protocol
SSD	Solid-state Drive
SSDP	Simple Service Discovery Protocol
TB	Tera Byte
TCP	Transmission Control Protocol
TFlop	Tera Floating Point Operations per Second
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
URL	Universal Resource Locator
USB	Universal Serial Bus
UTF	Unicode Transformation Format
VOD	Video-on-Demand
VoIP	Voice over IP
ZFS	Zettabyte File System

List of Terms

VoIP is a technology that enables transferring real-time voice communication over an IP Data Network.

DoS attack is an attempt to make a computer resource unavailable to its intended users.

DDoS is the distributed form of DoS.

URI is a string of characters used to identify a name or a resource on the Internet.

INTRODUCTION

P2P networks are based on the idea of decentralization, sharing of resources across large number of users. This decentralization enables the network to outperform all other content delivery technologies. In all of the parameters like total download amounts, speed, availability, scalability and cost - P2P superiority is unmatched. There are many P2P implementations currently available; however the most popular is the BitTorrent protocol with 60-90 percent of total P2P traffic [1, 2]. BitTorrent is a very flexible protocol that can be used to deliver any kind of content. For example it is ideal for distributing Video-on-Demand (VoD) and other high bandwidth applications. Because of the mentioned facts this study will focus on BitTorrent. To be able to fully exploit the possibilities of P2P networks the negative side effects of decentralization must be solved first. How can it be that the total P2P traffic is by more than 75 percent redundant? [3] This means the network communication is not effective and creates a lot of unnecessary overhead.

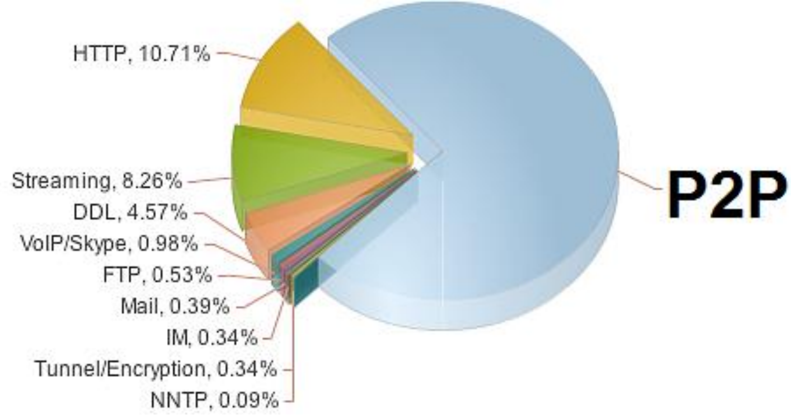


Figure 1: Protocol Type Distribution - Germany 2007, P2P represents 73.79 %

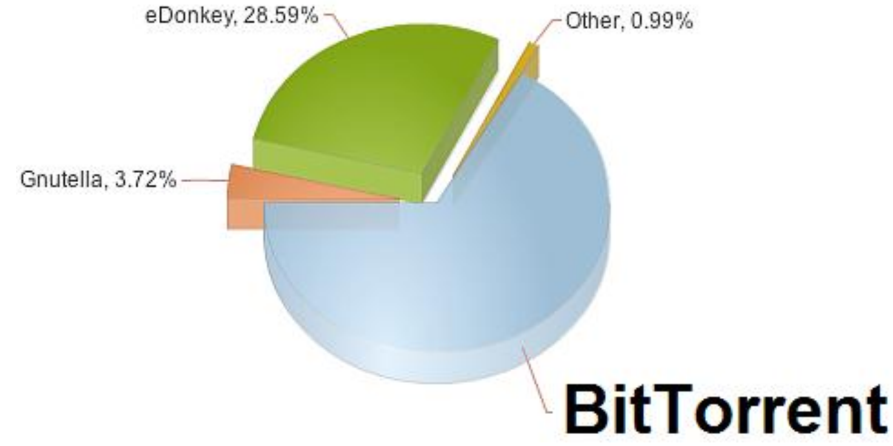


Figure 2: P2P Protocol Distribution by Volume - Germany 2007, BitTorrent represents 66.7 %

1 BITTORRENT ARCHITECTURE

In contrast with most other P2P protocols BitTorrent provides only a content transportation mechanism without any integrated search facility. This means that the user needs to find what he wants to download using some 3rd party search engine that is not part of the BitTorrent network. However these engines search thru the databases of the BitTorrent trackers which are a crucial part of the technology. Trackers are dedicated servers that are used for coordinating the network. After the client finds a metainfo .torrent file which contains details about the download, he loads the .torrent into a BitTorrent client that initializes communication with the tracker specified in it. The tracker then provides information about other peers which the client contacts and downloads from resp. uploads to. According to the BitTorrent specification the tracker does not need to hold the .torrent files nor provide any search capability. But in reality most trackers did exactly that which resulted into increasing legal pressure against them. This pressure caused the protocol to evolve and gradually replace the functionalities usually provided by trackers by other more decentralized means. The peer look-up role of the tracker is nowadays completely replaced by Distributed Hash Table which is not fully distributed either but eliminates the need for a tracker. Also the client does not need to search for the .torrent file anymore, the only thing he needs is a special plain text link called the magnet link. With this two technologies universally accepted by the user base there is no more single point of failure, no tracker that can be taken down, no single entity in control, only the users themselves.

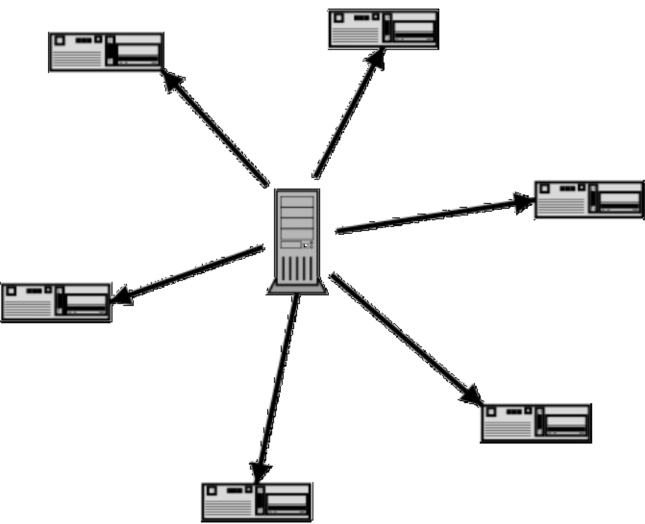


Figure 3: The Problem with Publishing: More users require more bandwidth

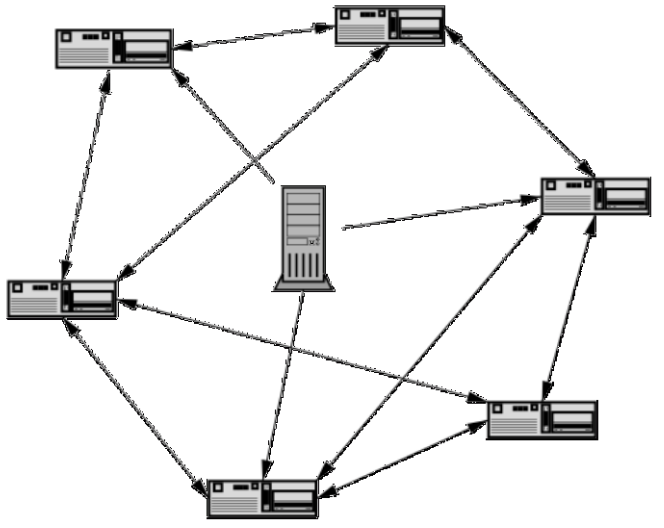


Figure 4: The BitTorrent solution: Users cooperate in the distribution

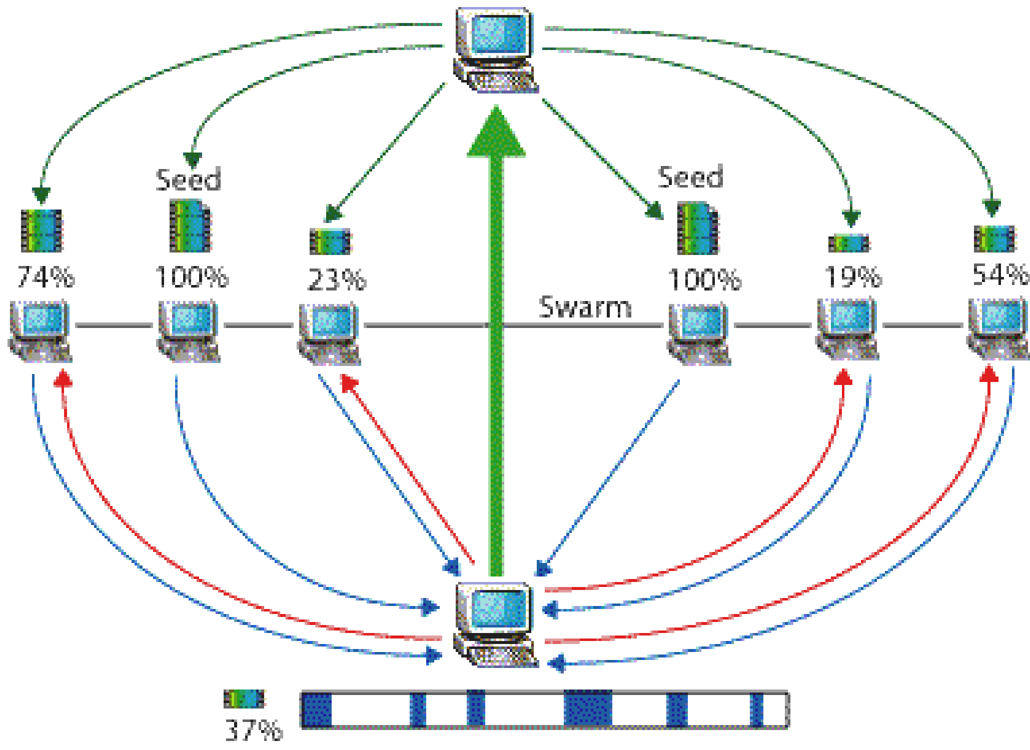


Figure 5: A graphical explanaition of the BitTorrent protocol architecture from HowStuffWorks.com

1.1. Torrent (MetaInfo) Format

The .torrent file contains informations about the file(s) that will be downloaded and peer discovery methods to be used. The information is stored in a special format called Bencode.

1.1.1 Bencoding

Bencoding is a way to specify and organize data in a terse format. It supports the following types: byte strings, integers, lists, and dictionaries.

Type	Description	Format	Example
Strings	Normal Strings [series of continuous characters]	*<string length>***:***<string data>*	*7:network* Represents the string network.
Integers	Normal integers	**i***<integer>***e**	***i3e*** Represents 3.
Lists	They are lists of types [strings, integers, lists, dictionaries]	**l***<bencoded type>***e**	***l****8:advanced7:network****e*** Represents the list of two strings: ["advanced", "network"].
Dictionaries	They are a mapping of keys to values	**d***<bencoded string><bencoded element>***e** **Note:** The keys are bencoded **strings**	**d**9:publisher3:bob17:publisher-webpage15:www.example.com18:publisher.location4:home**e** Represents {"publisher"=>"bob", "publisher-webpage"=>"www.example.com", "publisher.location"=>"home"}.

1.1.2 Torrent (MetaInfo) File Structure

The content of a metainfo file is a bencoded dictionary, containing the keys listed below. All character string values are UTF-8 encoded. Specification for single file torrents:

Key	Description
Info	A dictionary that describes the files.
- length	Length of file in bytes (integer).
- md5sum (optional)	A 32 character hexadecimal string corresponding to the MD5 sum of the file.
- name	The filename of a string (string).
- piece length	Number of bytes in each piece (integer).
- pieces	String consisting of the concatenation of all 20-byte SHA1 hash values, one per piece (raw binary encoded).
- private (optional)	If it is set to "1", the client MUST publish its presence to get other peers ONLY via the trackers explicitly described in the metainfo file. If this field is set to "0" or is not present, the client may obtain peer from other means, e.g. PEX peer exchange, DHT (integer).
Announce	The announce URL of the tracker.
Announce-list (optional)	This is an extension to the official specification, which is also backwards compatible. This key is used to implement lists of backup trackers.
Creation date (optional)	The creation time of the torrent, in standard Unix epoch format (integer seconds since 1-Jan-1970 00:00:00 UTC).

Comment (optional)	Free form text comments (string).
Created by (optional)	Name and version of the program used to create.
Encoding (optional)	The string encoding format used to generate the pieces part of the info dictionary in the .torrent metafile (string).

For the case of the multi-file mode, the info dictionary contains the following structure:

Key	**Description**
Info	A dictionary that describes the files.
- files	A list of dictionaries, one for each file. Each dictionary in this list contains the following keys.
- length	Length of file in bytes (integer).
- md5sum (optional)	A 32 character hexadecimal string corresponding to the MD5 sum of the file.
- path	A list containing one or more string elements that together represent the path and filename. Each element in the list corresponds to either a directory name or (in the case of the final element) the filename. For example, the file "dir1/dir2/file.ext" would consist of three string elements: "dir1", "dir2", and "file.ext".
name	the name of the top-most directory in the structure -- the directory which contains all of the files listed in the above **files** list (string).
piece length	Number of bytes in each piece (integer).

pieces	String consisting of the concatenation of all 20-byte SHA1 hash values, one per piece (raw binary encoded).

The piece length specifies the nominal piece size, and is usually a power of 2. The piece size is typically chosen based on the total amount of file data in the torrent, and is constrained by the fact that too-large piece sizes cause inefficiency, and too-small piece sizes cause large .torrent metadata file. Historically, piece size was chosen to result in a .torrent file no greater than approx. 50 - 75 KB (presumably to ease the load on the server hosting the torrent files). The most common sizes are 256 KB, 512 KB, and 1 MB.

Every piece is of equal length except for the final piece, which is irregular. The number of pieces is thus determined by *ceil (total length / piece size)*.

For the purposes of piece boundaries in the multi-file case, the file data is considered as one long continuous stream composed of the concatenation of each file in the order listed in the files list. The number of pieces and their boundaries are then determined in the same manner as the case of a single file. Pieces may overlap file boundaries.

Each piece has a corresponding SHA1 hash of the data contained within that piece. This hash is not a list but a single string. Which length must be a multiple of 20. [4, 5]

1.2 Tracker HTTP/HTTPS Protocol

The tracker is an HTTP/HTTPS service which responds to HTTP GET requests. The requests include metrics from clients that help the tracker keep overall statistics about the torrent. The response includes a peer list that helps the client participate in the network. The base URL consists of the "announce URL" as defined in the metadata (.torrent) file. The parameters are then added to this URL, using standard CGI methods (i.e. a '?' after the announce URL, followed by 'param=value' sequences separated by '&').

All binary data in the URL (particularly info_hash and peer_id) have to be properly escaped. This means any byte not in the set 0-9, a-z, A-Z, '.', '-', '_' and '~', must be encoded using the "%nn" format, where nn is the hexadecimal value of the byte. (As defined in RFC1738) The parameters used in the client => tracker GET request are:

Parameter	Description
info_hash	20-byte SHA1 hash of the *value* of the **info** key from the Metainfo file. The *value* will be a bencoded dictionary.
peer_id	20-byte string used as a unique ID for the client, generated by the client at startup.
port	The port number that the client is listening on. Ports reserved for BitTorrent are typically 6881-6889. Clients may choose to give up if unable to establish a port within this range.
uploaded	The total amount uploaded bytes since the client sent the 'started' event to the tracker, encoded in base ten ASCII.
downloaded	The total amount downloaded bytes since the client sent the 'started' event to the tracker, encoded in base ten ASCII.
left	The number of bytes the client still has to download, encoded in base ten ASCII.
compact	Setting this to 1 indicates that the client accepts a compact response. The peers list is replaced by a peers string with 6 bytes per peer. The first four bytes are the host (in network byte order), the last two bytes are the port (also in network byte order). Some trackers only support compact responses (for saving bandwidth) and either refuse requests without "compact=1" or simply send a compact response unless the request contains "compact=0" (in which case they will refuse the request).

no_peer_id	Indicates that the tracker can omit peer id field in peers dictionary. This option is ignored if compact is enabled.
event	If specified, must be one of **started**, **completed**, or **stopped**. If not specified, then this request is one performed at regular intervals.
- started	The first request to the tracker **must** include the event key with the **started** value.
- stopped	Must be sent to the tracker if the client is shutting down gracefully.
- completed	Must be sent to the tracker when the download completes. However, must not be sent if the download was already 100% complete when the client started. Presumably, this is to allow the tracker to increment the "completed downloads" metric based solely on this event
Ip (optional)	The true IP address of the client machine, in dotted quad format or RFC3513 defined hexed IPv6 address. *Notes: In general this parameter is not necessary as the address of the client can be determined from the IP address from which the HTTP request came. The parameter is only needed in the case where the IP address that the request came in on is not the IP address of the client. This happens if the client is communicating to the tracker through a proxy (or a transparent web*

	Proxy/Cache.) It also is necessary when both the client and the tracker are on the same local side of a NAT gateway. The reason for this is that otherwise the tracker would give out the internal (RFC1918) address of the client, which is not routeable. Therefore the client must explicitly state its (external, routeable) IP address to be given out to external peers. Various trackers treat this parameter differently. Some only honor it only if the IP address that the request came in on is in RFC1918 space. Others honor it unconditionally, while others ignore it completely. In case of IPv6 address (e.g.: 2001:db8:1:2::100) it indicates only that client can communicate via IPv6.
numwant (optional)	Number of peers that the client would like to receive from the tracker. This value is permitted to be zero. If omitted, typically defaults to 50 peers.
key (optional)	An additional identification that is not shared with any users. It is intended to allow a client to prove their identity should their IP address change.
trackerid (optional)	If a previous announce contained a tracker id, it should be set here.

The tracker replies with a standard HTTP response containing a "text/plain" document consisting of a bencoded dictionary with the following keys:

Key	Description
failure reason	If present, then no other keys may be present. The value is a human-readable error message as to why the request failed (string).
warning message (optional)	Similar to failure reason, but the response still gets processed normally. The warning message is shown just like an error.
interval	Interval in seconds that the client should wait between sending regular requests to the tracker.
min interval (optional)	Minimum announce interval. If present clients must not reannounce more frequently than this.
tracker id (optional)	A string that the client should send back on its next announcements. If absent and a previous announce sent a tracker id, the client should keep using the old one.
complete	Number of peers with the entire file, i.e. seeders (integer).
incomplete	Number of non-seeder peers, aka "leechers" (integer).
peers	The value is a list of dictionaries, each with the following keys. (dictionary model)
- peer id	Peer's self-selected ID (string).
- ip	Peer's IP address either IPv6 (hexed) or IPv4 (dotted quad) or DNS name (string)

- port	Peer's port number (integer)
peers	Instead of using the dictionary model described above, the peers value may be a string consisting of multiples of 6 bytes. First 4 bytes are the IP address and last 2 bytes are the port number. All in network (big endian) notation (binary model).

As already mentioned, the list of peers is length 50 by default. If there are fewer peers in the torrent, then the list will be smaller. Otherwise, the tracker randomly selects peers to include in the response. The tracker may choose to implement a more intelligent mechanism for peer selection when responding to a request. For instance, reporting seeds to other seeders could be avoided.

Clients may send a request to the tracker more often than the specified interval, if an event occurs (i.e. stopped or completed) or if the client needs to learn about more peers. However, it is considered bad practice to "hammer" on a tracker to get multiple peers. If a client wants a large peer list in the response, then it should specify the **numwant** parameter. [4, 5]

1.3 Tracker 'scrape' Convention

By convention most trackers support another form of request, which queries the state of a given torrent (or all torrents) that the tracker is managing. This is referred to as the "scrape page" because it automates the otherwise tedious process of "screen scraping" the tracker's stats page. The scrape URL is also a HTTP GET method as the Tracker HTTP/HTTPS request.

Examples: (announce URL => scrape URL):

```
http://spam.com/announce            -> http://spam.com/scrape
http://spam.com/x/announce          -> http://spam.com/x/scrape
http://spam.com/announce.php        -> http://spam.com/scrape.php
http://spam.com/a                   -> (scrape not supported)
http://spam.com/announce?x=2%0644 -> http://spam.com/scrape?x=2%0644
http://spam.com/announce?x=2/4      -> (scrape not supported)
http://spam.com/x%064announce       -> (scrape not supported)
```

To determine if the tracker supports scrape convention the following steps are taken:

- Find the last '/' in the announce URL.
- If the text immediately following the found '/' isn't 'announce' it will be taken as a sign that that tracker doesn't support the scrape convention.
- If it does, 'scrape' is substituted for 'announce' to find the scrape page.

The scrape URL may be supplemented by the optional parameter info_hash. This restricts the tracker's report to that particular torrent. Otherwise stats for all torrents that the tracker is managing are returned. The response of this HTTP GET method is a "text/plain" document consisting of a bencoded dictionary, containing the following keys:

Key	**Description**
failure reason (optional)	Human-readable error message as to why the request failed (string).
flags (optional)	A dictionary containing miscellaneous flags. For example min_request_interval.
files	A dictionary containing one key/value pair for each torrent for which there are stats. If *info_hash* was supplied and was valid, this dictionary will contain a single key/value. Each key consists of a 20-byte binary *info_hash*. The value of each entry is another dictionary containing the following:
- complete	Number of peers with the entire file, i.e. seeders (integer).
- downloaded	Total number of times the tracker has registered a completion (integer) ("event=complete", i.e. a client finished downloading the torrent).

- incomplete	Number of non-seeder peers, aka "leechers" (integer).
- name (optional)	The torrent's internal name, as specified by the "name" file in the info section of the .torrent file.

The scrape response has three levels of dictionary nesting. For example:

d5:files**d**20:....................**d**8:complete**i**5**e**10:downloaded**i**50**e**10:incomplete**i**10**eeee**

Where .... is the 20 byte info_hash and there are 5 seeders, 10 leechers, and 50 complete downloads. [4, 5]

1.4 Peer wire protocol

The peer protocol facilitates the exchange of pieces as described in the 'metainfo file.

The original specification also used the term "piece" when describing the peer protocol, but as a different term than "piece" in the metainfo file. For that reason, the term "block" will be used to describe the data that is exchanged between peers over the wire. A block is typically lesser in size than the piece.

Every client must maintain state information for each connection that it has with a remote peer:

choked: Whether or not the remote peer has choked this client. When a peer chokes the client, it is a notification that no requests will be answered until the client is unchoked. The client should not attempt to send requests for blocks, and it should consider all pending (unanswered) requests to be discarded by the remote peer.

interested: Whether or not the remote peer is interested in something this client has to offer. This is a notification that the remote peer will begin requesting blocks when the client unchokes them.

This also implies that the client will also need to keep track of whether or not it is interested in the remote peer, and if it has the remote peer choked or unchoked. So, the real list looks something like this:

- **am_choking**: this client is choking the peer, starts on 1
- **am_interested**: this client is interested in the peer, starts on 0
- **peer_choking**: peer is choking this client, starts on 1
- **peer_interested**: peer is interested in this client, starts on 0

A block is downloaded by the client when the client is interested in a peer, and that peer is not choking the client. A block is uploaded by a client when the client is not choking a peer, and that peer is interested in the client. It is important for the client to keep its peers informed as to whether or not it is interested in them. This state information should be kept up-to-date with each peer even when the client is choked. This will allow peers to know if the client will begin downloading when it is unchoked (and vice-versa). [4, 5]

1.4.1 Handshake

The handshake is a required message and must be the first message transmitted by the client. It is (49+len(pstr)) bytes long.

handshake: <pstrlen><pstr><reserved><info_hash><peer_id>

- **pstrlen**: string length of <pstr>, as a single raw byte
- **pstr**: string identifier of the protocol
- **reserved**: eight (8) reserved bytes. All current implementations use all zeroes. Each bit in these bytes can be used to change the behavior of the protocol. An email from Bram suggests that trailing bits should be used first, so that leading bits may be used to change the meaning of trailing bits.
- **info_hash**: 20-byte SHA1 hash of the info key in the metainfo file. This is the same info_hash that is transmitted in tracker requests.

- **peer_id**: 20-byte string used as a unique ID for the client. This is usually the same peer_id that is transmitted in tracker requests.

In version 1.0 of the BitTorrent protocol, pstrlen = 19, and pstr = "BitTorrent protocol". If everything matches the receiver responds with handshake message with peer_id field modified to its own ID. In case of any exception the connection is dropped. [4, 5]

1.4.2 Messages

All of the remaining messages in the protocol are in the following form:

<length prefix><message ID><payload>

The length prefix is a four byte big-endian value. The message ID is a single decimal byte. The payload is message dependent. [4, 5]

Message	Description
keep-alive: <len=0000>	The **keep-alive** message is a message with zero bytes, specified with the length prefix set to zero. There is no message ID and no payload. Peers may close a connection if they receive no messages (**keep-alive** or any other message) for a certain period of time, so a keep-alive message must be sent to maintain the connection *alive* if no command has been sent for a given amount of time (generally two minutes).
choke: <len=0001><id=0>	The **choke** message is fixed-length and has no payload.
unchoke: <len=0001><id=1>	The **unchoke** message is fixed-length and has no payload.

interested: <len=0001><id=2>	The **interested** message is fixed-length and has no payload.
not interested: <len=0001><id=3>	The **not interested** message is fixed-length and has no payload.
have: <len=0005><id=4><piece index>	The **have** message is fixed length. The payload is the zero-based index of a piece that has just been successfully downloaded and verified via the hash.
bitfield: <len=0001+X><id=5><bitfield>	The **bitfield** message may only be sent immediately after the handshaking sequence is completed, and before any other messages are sent. It is **optional**, and need not be sent if a client has no pieces.
request: <len=0013><id=6><index><begin><length>	The **request** message is fixed length, and is used to request a block. The payload contains the following information: • **index**: integer specifying the zero-based piece index • **begin**: integer specifying the zero-based byte offset within the piece • **length**: integer specifying the requested length.
piece: <len=0009+X><id=7><index><begin><block>	The **piece** message is variable length, where X is the length of the block. The payload contains the following information: • **index**: integer specifying the zero-based piece index • **begin**: integer specifying the zero-based byte offset within the piece • **block**: block of data, which is a subset of the piece specified by index.

cancel: **<len=0013><id=8><index><begin><length>**	The **cancel** message is fixed length, and is used to cancel block requests. The payload is identical to that of the "request" message.
port: <len=0003><id=9><listen-port>	The **port** message is sent by newer versions of the Mainline that implements a DHT tracker. The listen port is the port this peer's DHT node is listening on. This peer should be inserted in the local routing table (if DHT tracker is supported).

1.5 Distributed Hash Table

BitTorrent uses a "distributed sloppy hash table" (DHT) for storing peer contact information for "trackerless" torrents. In effect, each peer becomes a tracker. The protocol is based on Kademila [ref na kademlia] and is implemented over UDP. The DHT is composed of nodes and stores the location of peers. Where a "node" is a client/server listening on a UDP port implementing the distributed hash table protocol and a "peer" is a client/server listening on a TCP port that implements the BitTorrent protocol. BitTorrent clients include a DHT node, which is used to contact other nodes in the DHT to get the location of peers to download from using the BitTorrent protocol. [6]

1.5.1 Overview

Each node has a globally unique identifier known as the "node ID." Node IDs are chosen at random from the same 160-bit space as BitTorrent infohashes. A "distance metric" is used to compare two node IDs or a node ID and an infohash for "closeness." Nodes must maintain a routing table containing the contact information for a small number of other nodes. The routing table becomes more detailed as IDs get closer to the node's own ID. Nodes know about many other nodes in the DHT that have IDs that are "close" to their own but have only a handful of contacts with IDs that are very far away from their own. In Kademlia, the distance metric is XOR and the result is interpreted as an unsigned integer. *distance(A,B) = |A xor B|* Smaller values are closer.

1.5.2 Routing Table

Every node maintains a routing table of known good nodes. The nodes in the routing table are used as starting points for queries in the DHT. Nodes from the routing table are returned in response to queries from other nodes.

Not all nodes that we learn about are equal. Some are "good" and some are not. Many nodes using the DHT are able to send queries and receive responses, but are not able to respond to queries from other nodes. It is important that each node's routing table must contain only known good nodes. A good node is a node has responded to one of our queries within the last 15 minutes. A node is also good if it has ever responded to one of our queries and has sent us a query within the last 15 minutes. After 15 minutes of inactivity, a node becomes questionable. Nodes become bad when they fail to respond to multiple queries in a row. Nodes that we know are good are given priority over nodes with unknown status.

1.5.3 BitTorrent Protocol Extension

Peers supporting the DHT set the last bit of the 8-byte reserved flags exchanged in the BitTorrent protocol handshake. Peer receiving a handshake indicating the remote peer supports the DHT should send a PORT message. It begins with byte 0x09 and has a two byte payload containing the UDP port of the DHT node in network byte order. Peers that receive this message should attempt to ping the node on the received port and IP address of the remote peer. If a response to the ping is received, the node should attempt to insert the new contact information into their routing table according to the usual rules.

1.5.4 Torrent File Extensions

A trackerless torrent dictionary does not have an "announce" key. Instead, a trackerless torrent has a "nodes" key. This key should be set to the K closest nodes in the torrent generating client's routing table. Alternatively, the key could be set to a known good node such as one operated by the person generating the torrent.

nodes *= [["<host>", <port>], ["<host>", <port>], ...]*

nodes *= [["127.0.0.1", 6881], ["your.router.node", 4804]]*

1.5.5 KRPC Protocol

The KRPC protocol is a simple RPC mechanism consisting of bencoded dictionaries sent over UDP. A single query packet is sent out and a single packet is sent in response. There is no retry. There are three message types: query, response, and error. For the DHT protocol, there are four queries: ping, find_node, get_peers, and announce_peer. A KRPC message is a single dictionary with two keys common to every message and additional keys depending on the type of message. Every message has a key "t" with a string value representing a transaction ID. This transaction ID is generated by the querying node and is echoed in the response, so responses may be correlated with multiple queries to the same node. The transaction ID should be encoded as a short string of binary numbers, typically 2 characters are enough as they cover 2^16 outstanding queries. The other key contained in every KRPC message is "y" with a single character value describing the type of message. The value of the "y" key is one of "q" for query, "r" for response, or "e" for error.

1.5.5.1 Contact Encoding

Contact information for peers is encoded as a 6-byte string. Also known as "Compact IP-address/port info" the 4-byte IP address is in network byte order with the 2 byte port in network byte order concatenated onto the end.

Contact information for nodes is encoded as a 26-byte string. Also known as "Compact node info" the 20-byte Node ID in network byte order has the compact IP-address/port info concatenated to the end.

1.5.5.2 Queries

Queries, or KRPC message dictionaries with a "y" value of "q", contain two additional keys; "q" and "a". Key "q" has a string value containing the method name of the query. Key "a" has a dictionary value containing named arguments to the query.

1.5.5.3 Responses

Responses, or KRPC message dictionaries with a "y" value of "r", contain one additional key "r". The value of "r" is a dictionary containing named return values. Response messages are sent upon successful completion of a query.

1.5.5.4 Errors

Errors, or KRPC message dictionaries with a "y" value of "e", contain one additional key "e". The value of "e" is a list. The first element is an integer representing the error code. The second element is a string containing the error message. Errors are sent when a query cannot be fulfilled. The following table describes the possible error codes:

Error Code	Description
201	Generic Error.
202	Server Error.
203	Protocol Error, such as a malformed packet, invalid arguments, or bad token.
204	Method Unknown.

Example Error Packets:

generic error = *{"t":"aa", "y":"e", "e":[201, "A Generic Error Ocurred"]}*

bencoded = *d1:eli201e23:A Generic Error Ocurrede1:t2:aa1:y1:ee*

1.5.6 DHT Queries

All queries have an "id" key and value containing the node ID of the querying node. All responses have an "id" key and value containing the node ID of the responding node.

1.5.6.1 Ping

The most basic query is a ping. "q" = "ping" A ping query has a single argument, "id" the value is a 20-byte string containing the senders node ID in network byte order. The appropriate response to a ping has a single key "id" containing the node ID of the responding node.

arguments: *{"id" : "<querying nodes id>"}*

response: *{"id" : "<queried nodes id>"}*

Example packets:

ping Query *= {"t":"aa", "y":"q", "q":"ping", "a":{"id":"abcdefghij0123456789"}}*

bencoded *= d1:ad2:id20:abcdefghij0123456789e1:q4:ping1:t2:aa1:y1:qe*

Response *= {"t":"aa", "y":"r", "r": {"id":"mnopqrstuvwxyz123456"}}*

bencoded *= d1:rd2:id20:mnopqrstuvwxyz123456e1:t2:aa1:y1:re*

1.5.6.2 find_node

Find node is used to find the contact information for a node given its ID. "q" == "find_node" A find_node query has two arguments, "id" containing the node ID of the querying node, and "target" containing the ID of the node sought by the queryer. When a node receives a find_node query, it should respond with a key "nodes" and value of a string containing the compact node info for the target node or the K (8) closest good nodes in its own routing table.

arguments: *{"id" : "<querying nodes id>", "target" : "<id of target node>"}*

response: *{"id" : "<queried nodes id>", "nodes" : "<compact node info>"}*

Example Packets:

find_node Query *= {"t":"aa", "y":"q", "q":"find_node", "a": {"id":"abcdefghij0123456789", "target":"mnopqrstuvwxyz123456"}}*

bencoded *= d1:ad2:id20:abcdefghij01234567896:target20:mnopqrstuvwxyz123456e1 :q9: find_node1:t2:aa1:y1:qe*

Response *= {"t":"aa", "y":"r", "r": {"id":"0123456789abcdefghij", "nodes": "def456..."}}*

bencoded *= d1:rd2:id20:0123456789abcdefghij5:nodes9:def456...e1:t2:aa1:y1:re*

1.5.6.3 get_peers

Get peers associated with a torrent infohash. "q" = "get_peers" A get_peers query has two arguments, "id" containing the node ID of the querying node, and "info_hash"

containing the infohash of the torrent. If the queried node has peers for the infohash, they are returned in a key "values" as a list of strings. Each string containing "compact" format peer information for a single peer. If the queried node has no peers for the infohash, a key "nodes" is returned containing the K nodes in the queried nodes routing table closest to the infohash supplied in the query. In either case a "token" key is also included in the return value. The token value is a required argument for a future announce_peer query. The token value should be a short binary string.

arguments: *{"id" : "<querying nodes id>", "info_hash" : "<20-byte infohash of target torrent>"}*

response: *{"id" : "<queried nodes id>", "token" :"<opaque write token>", "values" : ["<peer 1 info string>", "<peer 2 info string>"]}*

or: *{"id" : "<queried nodes id>", "token" :"<opaque write token>", "nodes" : "<compact node info>"}*

Example Packets:

get_peers Query *= {"t":"aa", "y":"q", "q":"get_peers", "a": {"id":"abcdefghij0123456789", "info_hash":"mnopqrstuvwxyz123456"}}*

bencoded = d1:ad2:id20:abcdefghij01234567899:info_hash20: mnopqrstuvwxyz123456e1:q9:get_peers1:t2:aa1:y1:qe

Response with peers *= {"t":"aa", "y":"r", "r": {"id":"abcdefghij0123456789", "token":"aoeusnth", "values": ["axje.u", "idhtnm"]}}*

bencoded = d1:rd2:id20:abcdefghij01234567895:token8:aoeusnth6:valuesl6: axje.u6:idhtnmee1:t2:aa1:y1:re

Response with closest nodes *= {"t":"aa", "y":"r", "r": {"id":"abcdefghij0123456789", "token":"aoeusnth", "nodes": "def456..."}}*

bencoded *= d1:rd2:id20:abcdefghij01234567895:nodes9:def456...5:token8: aoeusnthe1:t2:aa1:y1:re*

1.5.6.4 announce_peer

Announce that the peer, controlling the querying node, is downloading a torrent on a port. announce_peer has four arguments: "id" containing the node ID of the querying node, "info_hash" containing the infohash of the torrent, "port" containing the port as an integer, and the "token" received in response to a previous get_peers query. The queried node must verify that the token was previously sent to the same IP address as the querying node. Then the queried node should store the IP address of the querying node and the supplied port number under the infohash in its store of peer contact information. [6]

arguments: *{"id" : "<querying nodes id>", "info_hash" : "<20-byte infohash of target torrent>", "port" : <port number>, "token" : "<opaque token>"}*

response: *{"id" : "<queried nodes id>"}*

Example Packets:

announce_peers Query *= {"t":"aa", "y":"q", "q":"announce_peer", "a": {"id":"abcdefghij0123456789", "info_hash":"mnopqrstuvwxyz123456", "port": 6881, "token": "aoeusnth"}}*

bencoded *= d1:ad2:id20:abcdefghij01234567899:info_hash20:
*

mnopqrstuvwxyz1234564:porti6881e5:token8:aoeusnthe1:q13:announce_peer1:t2:aa1: y1:qe

Response *= {"t":"aa", "y":"r", "r": {"id":"mnopqrstuvwxyz123456"}}*

bencoded *= d1:rd2:id20:mnopqrstuvwxyz123456e1:t2:aa1:y1:re*

1.6 Local Peer Discovery

The Local Peer Discovery protocol is a BitTorrent protocol extension. It is designed to support the discovery of local BitTorrent peers, aiming to minimize the traffic through the Internet Service Provider's channel and maximize use of higher-bandwidth local networks. It utilizes IPv4 multicasts for sending regular (every 5 minutes) SSDP messages to the multicast group 239.192.152.143:6771. The message format can be best illustrated by the following code block from libtorret [7, 8]:

```
snprintf(msg, 200,
  "BT-SEARCH * HTTP/1.1\r\n"
  "Host: 239.192.152.143:6771\r\n"
  "Port: %d\r\n"
  "Infohash: %s\r\n"
  "\r\n\r\n", listen_port, ih_hex);
```

Where the listen port is the port on which the peer listens for incomming BitTorrent connections and ih_hex is the hex encoded info_hash of the propagated torrent.

1.7 Magnet link

This extension instead allows clients to download the metadata from peers. It makes it possible to support magnet links, a link on a web page only containing enough information to join the swarm (the info hash).

The magnet URI format is:

magnet:?xt=urn:btih:<info-hash>&dn=<name>&tr=<tracker-url>

xt is the only mandatory parameter. dn is the display name that may be used by the client to display while waiting for metadata. tr is a tracker url, if there is one. If there are multiple trackers, multiple tr entries may be included. Both dn and tr are optional. If no tracker is specified, the client should use the DHT to acquire peers. [9]

2. P2P POPULARITY DISTRIBUTION

To study the popularity distribution in P2P networks, specifically in the BitTorrent network, a spider that went thru the most popular BitTorrent search engine – ThePirateBay.org and collected the necessary data was created. Even if the site claims to have more than 2.5 million torrents indexed only 1 066 000 were found of which only 524 288 were active. Where active was defined as every torrent having at least one seeder and one leecher. The cumulated active BitTorrent content was calculated to be about 700 TB.

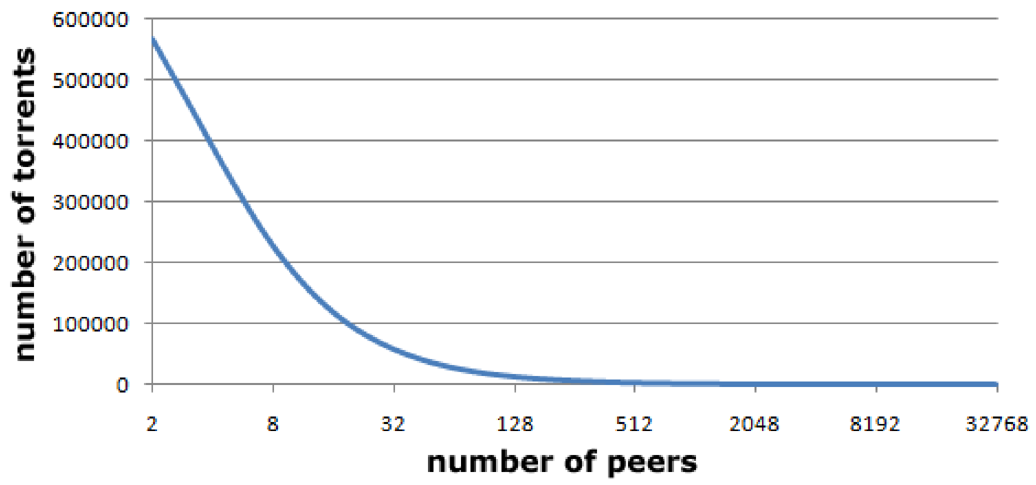


Figure 6: Distribution of torrents based on number of peers.

First we measured distribution of torrents based on number of peers. It is clear that only a fraction of torrents has more than 200 peers.

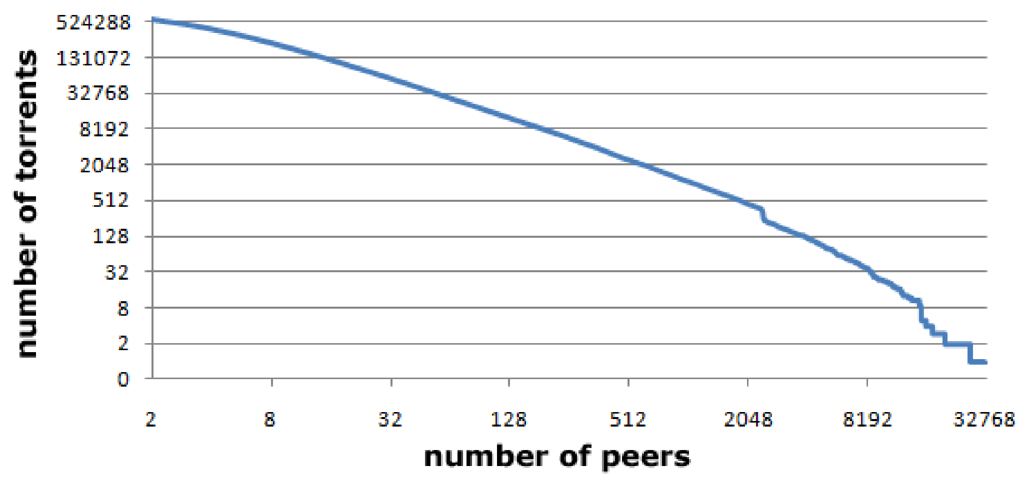


Figure 7: Detailed distribution with logarithmic vertical axis.

In the Figure 7, detailed graph of distribution, vertical axis represents number of torrents with more than x peers.

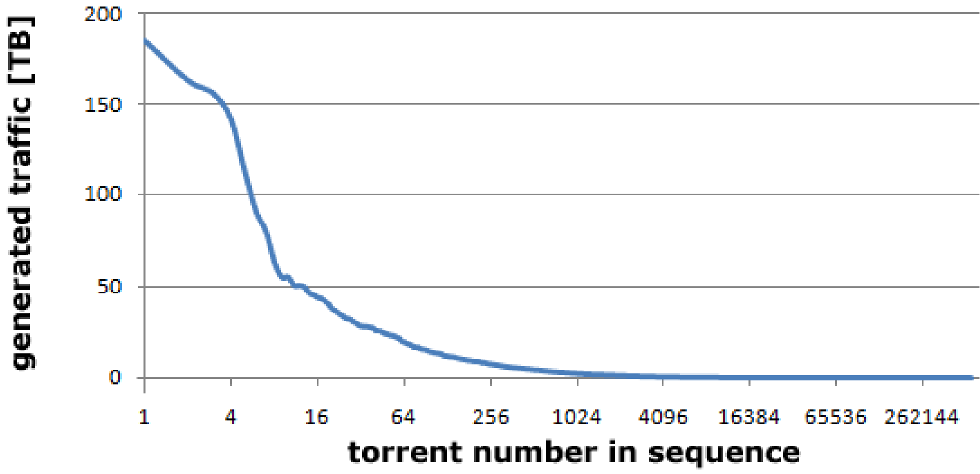


Figure 8: Traffic generated per torrent.

The horizontal axis represents the torrents sorted by their traffic while the vertical axis represents the corresponding traffic generated by the torrent. Figure 8 shows that just a few torrents are responsible for majority of the BitTorrent traffic.

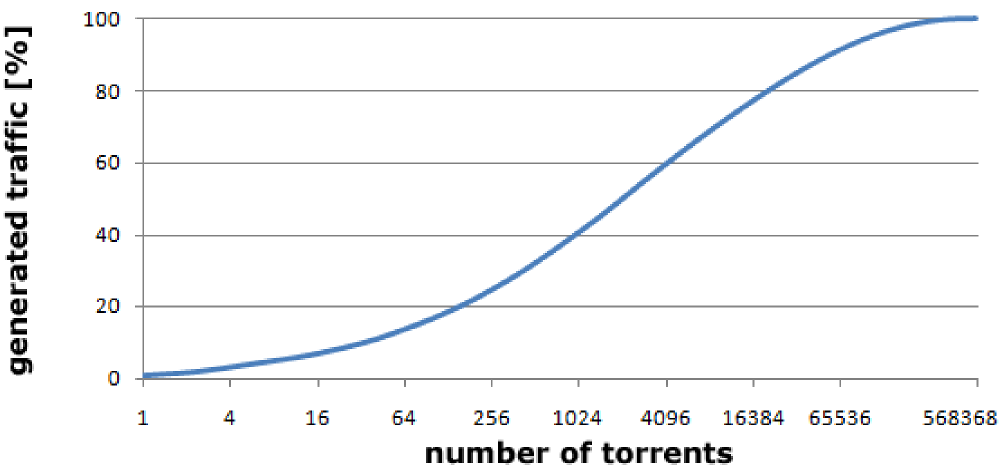


Figure 9: Total traffic generated per torrents.

Figure 9 demonstrates that just by caching of the 150 most popular torrents it is possible to save up to 20% of the BitTorrent traffic. This means that a Proxy/Cache with only approximately 500 GB of storage can save up to 8% of the ISP's downlink.

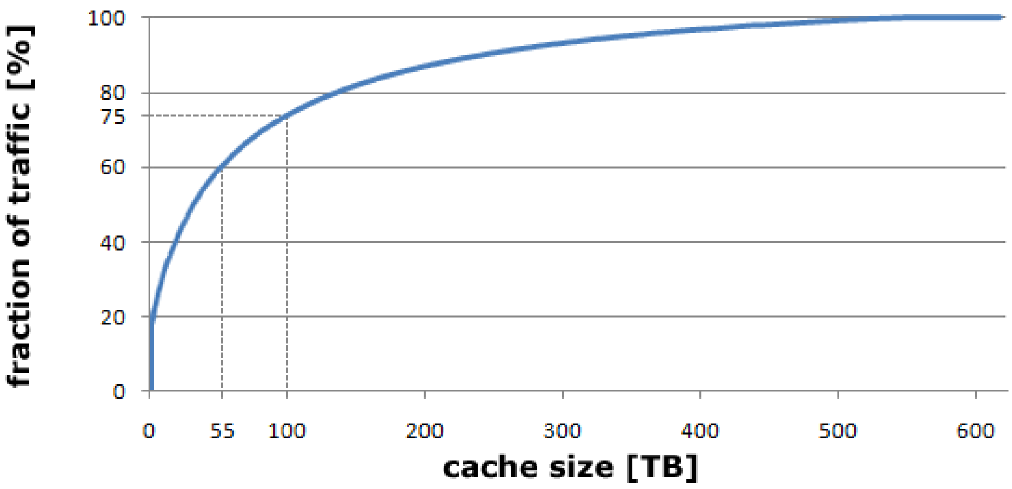


Figure 10: Cache size.

The cache disk space can be represented by the Pareto principle also called the 80-20 rule. By caching 20 percent of the active BitTorrent content up to 80 percent of hit probability can be achieved. According to our observations, optimal cache size for active torrents is between 55 TB and 100 TB. [10]

3. CONTROLING THE FLOW

As practically every existing protocol, BitTorrent is vulnerable to man-in-the-middle (MiTM) attacks. The only question is how effectively can such attack be performed?

It was already shown that BitTorrent generates immense amounts of data; clearly controlling this flow wouldn't be an easy task. Therefore the plain text client-tracker communication was selected to be the primary attack vector. Instead of monitoring the inter-peer communication that transports the actual data, only the peer look-up process is attacked. Classic MiTM scenarios require the attacker to be in-band e.g. to forward the victims data as a gateway for example. In such case the whole victim's traffic is flowing thru the attacker and all of it needs to be processed. Probably the easiest way how to set-up this scenario is to use Iptables port redirecting to a local port. Iptables is able to redirect any communication going thru the local node to any port set. An example rule would be:

iptables -A PREROUTING -t nat -i eth0 -p tcp --dport 80 -j REDIRECT --to-port 8080

This command takes all TCP traffic to any destination host to port 80 and redirects it to the local node's incoming interface port 8080. To further optimize the command a string matching option can be added for example with: *-m string --algo bm --string "info_hash="*, the effect would be that only packets containing "info_hash=" would be redirected.

Trackers don't always use port 80, what about other ports? One possibility would be to add more rules with different destination ports. Or take it from the other side and specify a list of "known" trackers and redirect all communication with them to the attacker, not exactly the most generic approach. None of this solution is optimal, because:

- Any in-band device represents in most cases a single point of failure
- Either the need to have many rules covering all the used ports
- Or specify a pre-generated list of "known" trackers

A better solution would be to use an out-of-band approach. Out-of-band means that the attacker listens to the network traffic and takes actions only when it is necessary without processing/forwarding the "uninteresting" traffic. In the case of client-tracker communication and as it will be shown also in most other cases, it is completely sufficient to listen only to the segment uplink e.g. the traffic flowing from the local segment to the Internet. Generally in all other than server hosting networks the uploaded data is several times smaller than the downloaded. The most simple out-of-band MiTM attack would be to modify the local DNS entries for a list of pre-determined most used trackers and direct this traffic to the attacker. Several problems exist in this scenario, first of all - DNSSEC. Domain Name System Security Extensions works by digitally signing answers to DNS lookups using public-key cryptography. Theoretically it should protect from MiTM attacks but due to its build-in backwards compatibility trivial downgrade attack is possible if the network administrator is willing to use "unsecure" DNS in the local segment. A more complex attack would be to perform the protocol downgrade only on requests to specific trackers. Thus the security level for "normal" DNS requests wouldn't be lowered. Anyway, DNS MiTM won't be ever generic. As there is no way to know which host is a tracker or not from the DNS request itself. The only truly universal BitTorrent tracker MiTM attack has to work by string matching the whole segment uplink for specific patterns and launch a TCP session hijack on a hit. To optimize the load only minimum traffic has to flow thru the hijacked TCP session. [11]

3.1 Client-Tracker Communication

In the client-tracker communication a simple HTTP moved permanently redirect to the attacker's IP address seems to be optimal. According to RFC 2616 the Location field contains the URI to which the client is being redirected. The original request containing all the parameters is parsed to this new URI and the original request Host: port is added to the end of it as an additional parameter. This way no information is being lost and all client-tracker communication is being intercepted.

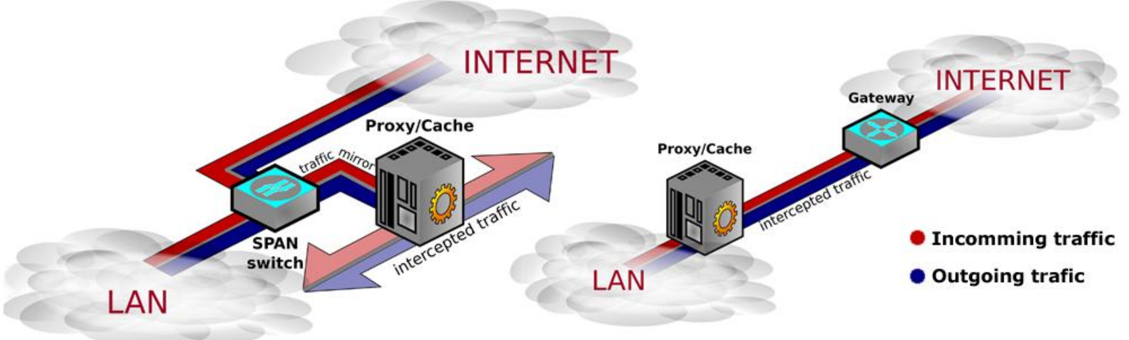


Figure 11: Out-of-Band vs. In-Band.

3.2 Distributed Hash Table

To be able to fully optimize the BitTorrent traffic all peer discovery methods have to be controlled. Theoretically DHT is decentralized, but practically not so much. The router node specified in the .torrent file is mostly one of only a few well known “always-up” nodes: router.bittorrent.com or router.utorrent.com. This router node is contacted if the peer doesn’t have any other peers in his peerlist, defacto it is used for bootstrapping. Blocking this node would stop the peer gathering process only for nodes without any other peers. A situation that normally doesn’t happen. Therefore another form of control over DHT has to be developed. Even if DHT uses UDP, it is not completely trivial to perform a MiTM attack on it because:

Every message has a key "t" with a string value representing a transaction ID. This transaction ID is generated by the querying node and is echoed in the response, so responses may be correlated with multiple queries to the same node. [6]

If the node receives a reply with an ID that is not in its request list it will ignore it. So, the attacker needs to perform a DHT session hijack. First step is to capture a DHT query and reply with its ID and hijacked ip:port. There two possible approaches:

- Reply and DHT error to all queries, effectively completely blocking off the whole DHT protocol
- Reply the attacker as the only node in all find_node and get_peers replies

Forcing the attacker as the only node seems as a better option due to less required processing/packets to than to block DHT.

3.3 Local Peer Discovery

As already presented, Local Peer Discovery uses multicasts. This effectively limits its reach to the nodes in the same multicasts domain. Communication between nodes in the same segment is usually not an issue that would need to be addressed.

4. PEER-TO-PEER PROXY/CACHE

Controlling BitTorrent data flow can be done as showed in the previous text. But what could be the motivation behind doing something like that? The possibility to create a transparent BitTorrent Proxy/Cache server. Such device would provide the full control over the user downloads in the network segment served by the proxy. It would be possible to collect detailed statistical information on all downloads; select content to be cached and then transparently served to other users requesting the same content thus eliminating redundancy of data being redownloaded.

Benefits of such approach:

- Possibility to completely eleminate BitTorrent uplink from clients in the given segment
- Massively reduce amounts of downloaded data due to reducing redundancy
- If content is already cached, clients download it at full speed of their Internet connection (link) from the very first byte to the end, this is in contrast to classic P2P behaviour where downloads start very slow and then by negotiating with more users gain speed
- If content is not yet cached and there is a high possibility more users will be interested in the same content the proxy actively supports the client's download thus greatly enhances the client download speeds while parallely caching the content
- Specific content can be kept accessible longer than on classic P2P networks

Because the device is completely transparent it can be placed on different layers of the network, creating a hierarchical structure that maximizes the mentioned benefits even more. Using proxies a hybrid P2P network architecture can be created, hybrid because the content is no longer completely decentralized, the data flows are no longer uncontrolled and by that the negative side-effects of decentralization are solved while the benefits of it remain. The final effect of proxy-ing depends on fine tuning of the "all data - stored data" ratio. More disk space will enable more data being served from local cache. But when the content will be provided from the local cache, the clients won't

need to upload. It will be kept available by the proxy cache itself, so from the global perspective its availability will not be reduced. It is important to mention that because of the transparency of this solution neither the client software nor any other part of the currently used technology needs to be changed.

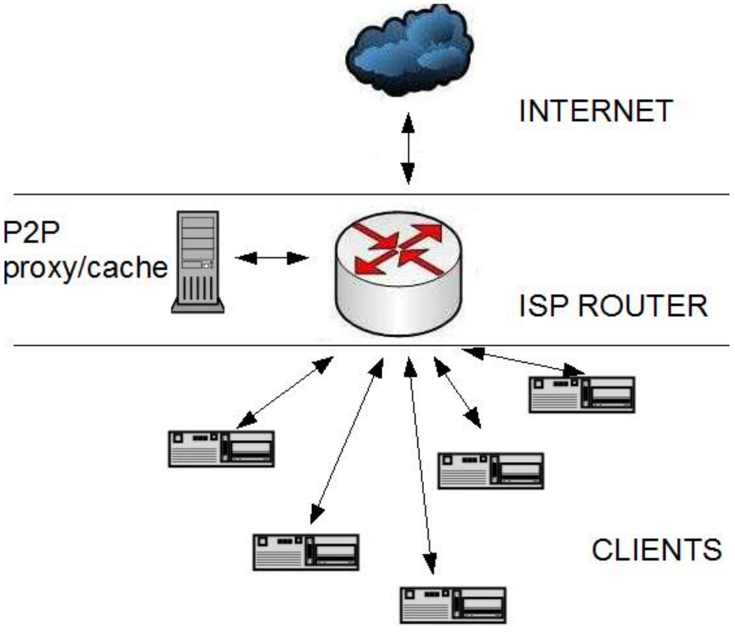


Figure 12: Example P2P Proxy/Cache topology.

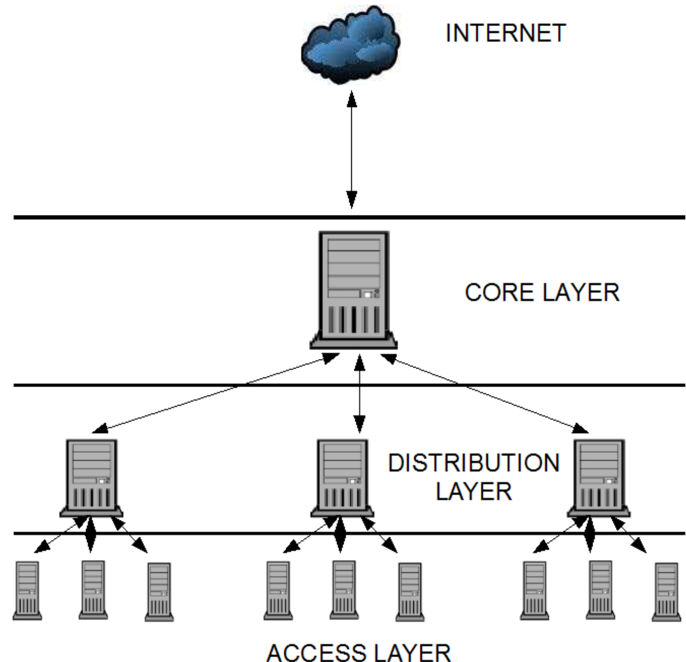


Figure 13: Example of a Hierarchical P2P Proxy/Cache topology.

This technology can be used not only in standard wired or wireless ISP's networks but it also opens new possibilities for satellite Internet providers. Great majority of Earth's

population has no access to broadband Internet. The latest generation of satellites is able to provide download speeds of up to 155Mbps with 6Mbps upload per user with relatively small dish size diameter of 45 centimeters. The problem with this kind of Internet connection is the extremely slow latency generally about 500-900ms [12, 13] (for comparison dial-up has latency around 150-200ms). The only way around this problem is to use on-orbit caching, thus reducing the latency for content provided from the cache to theoretical limit of 233ms (for the orbit height of 70,000 km). As shown with the use of extensive multi-protocol caching it would be possible to reduce latency to dial-up levels for almost all non-real-time traffic.

4.1. Intercepting Tracker HTTP/HTTPS protocol

Tracker requests are plain HTTP requests with a specific message format. For example some traffic filtering techniques use tracker responses to deny just the Peer-wire protocol (client-client communication). This is possible because they contain a clear text list of peers with their specific IP addresses and ports. As a reaction, some trackers started to encrypt this part of the response. The proxy recognizes and intercepts the requests, and then the actual man-in-the-middle (MiTM) attack begins. Most trackers still use HTTP or at least support it for compatibility reasons; however it is possible to intercept HTTPS in most cases too. This is because many trackers use self-signed certificates which are vulnerable to MiTM. When this is not the case then MiTM on HTTPS can be still successful as majority of BitTorrent clients does not verify the server certificate. Another factor that guarantees that the proxy will be able to intercept the user request is that it is a de-facto standard to use always more than one tracker - multiplying the chance of interception. The only case in which this interception mechanism would not work would require that only one tracker is specified and that uses a CA signed certificate (or all specified trackers use HTTPS and have a CA signed certificate) simultaneously the BitTorrent client verifies the certificate and then actively refuses the connection because it detects a MiTM attack. [11, 14]

4.2. The Man-in-the-middle Attack

The attack consists of three parts:

1) Identifying the BitTorrent Tracker request
2) Hijacking the TCP session
3) Sending a HTTP redirect to the Proxy/Cache

There are several approaches how to identify the Tracker request, for example:

1) Using iptables string matching – not a very flexible solution
2) Using pcap library – widely used but creates too much overhead
3) Read raw data directly from kernel and parse it thru some regexp – most flexible solution without any indirect overhead

The protocol defines clearly what argument and in what format shall be in the request therefore identifying it is pretty straightforward. Further performance boost could be achieved by parallelization of this comparison/parsing. We will look at this topic further in the chapter 4.7.4.

The TCP hijack is performed by identifying the session numbers and artificially generating appropriate values for the spoofed reply. The reply is a standard HTTP moved permanently response telling the user where he should resend the request, in our case to the Proxy/Cache. An automated hijacking tool has been developed especially for this purpose [14].

.

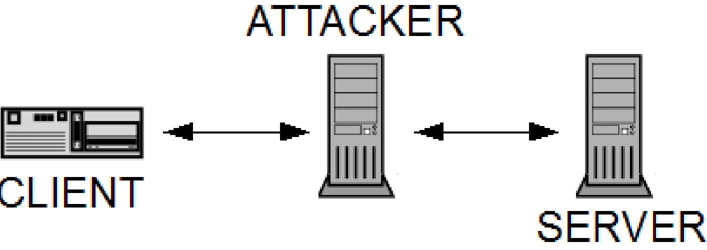


Figure 14: Man-in-the-middle attack.

4.3. The logic behind

After the client resends the request to the Location mentioned in the redirect he got a pseudo-tracker response is sent to him. This response contains only one peer and that is the Proxy/Cache downloader. If the content is not yet cached, Proxy/Cache reacts pro-actively and immediately starts downloading the given content. Of course disk space is not unlimited, yet it is important to use the disk space available as much as possible. So the strategy is to basically download everything but a cleaner process is running in the background that checks the popularity of all the content downloaded and if the free disk space drops below a critical point it deletes the at least popular files. Except of having as much as possible content already cached, clients benefit from pro-active approach too as the Proxy/Cache will always download faster than them because of several factors:

- It is on a faster link
- It can accept incoming connections (firewalled client's speeds are lower because their ports are blocked)
- It updates its list of trackers using all possible trackers that can be found on search engines having the content to be downloaded registered
- It uses its own Downloader which is superior to any commonly available BitTorrent client, more on this in the following chapter

All this factors result in massively enhancing client’s download speeds as the content is seeded parallel to downloading it.

4.4. Finding content

Two methods of caching content without the need to dump or "record" any network communication have been developed. The default method is based on BitTorrent search engines and metafile storage servers. The client sends the infohash parameter in its requests, the proxy uses it to search for that specific torrent metafile and then download it into the cache when needed. Using this method, it is possible to find most of the torrent metafiles, because the popular trackers are all well indexed. If the torrent file cannot be found but the at least one tracker has the infohash registered a magnet link is used. However, not everything can be found on public trackers, for example if the content is served by some private tracker. Private trackers don't allow DHT so magnet links won't work. When that happens, the backup method is used. This method is by far more complicated than the default one, but is able to reconstruct all the information needed to download the torrent content just from the client requests and some protocol hacking e.g. without the metafile itself. The problem is that the user starts the download based on information from the metafile which the proxy doesn't have, so if it wants to download the content it needs to get the missing information somehow. The most important thing is to get the piece length, without this parameter it would be impossible to reconstruct the received data. Luckily, from information sent by other peers via the Peer-wire protocol it is easy to calculate the piece length parameter from the bitfield length. The Downloader only needs the infohash, full length of the content and the tracker's address (one and more) parameters. All this information can be parsed from the client's request. The Downloader then starts downloading the torrent with default piece length. It listens to Peer-wire protocol for the bitfields and using a special algorithm verifies if the default piece length is the right one. If not, then the download is restarted with the correct value.

The piece length calculation algorithm (C code representation):

```
size_t GetRealPieceLength(size_t bitfieldNBytes)
{
size_t pieceLength = argm_file_size / (bitfieldNBytes * 8);
return (log2(pieceLength)>int(log2(pieceLength))) ? 2^(int(log2(pieceLength))+1):
2^(int(log2(pieceLength)));
}
```

Where:

- bitfieldNBytes is the length of the bitfield
- argm_file_size is the full content size
- pieceLength is what is needed to be computed

According to the BitTorrent protocol specification: *"A bitfield of the wrong length is considered an error. Clients should drop the connection if they receive bitfields that are not of the correct size, or if the bitfield has any of the spare bits set."* [6] That is why it is almost sure if few same bitfield lengths from different sources are detected, that this is the correct length. The Bitfield identifies which block has the client already downloaded. Piece length is usually power of 2 [4, 5], so it is calculated by simply looking for the nearest bigger power of 2 to the division of full content length by number of bits in the bitfield. The problem is that the spare bits on end of the bitfield are set to zero, so its length does not need to be always the correct argument. That means the maximal error can be 7 bits, a very small probability exists this piece length calculation could provide a bad result. Anyway, this is solved by the Downloader, where specific errors can be produced only by the piece length miscalculation; the piece length is then automatically divided by two. This should provide 100 percent correct setting of the piece length parameter, thus make Proxy/Cache able to download any content just by listening to the wire. [11]

The search engine method is the default one because it creates less overhead. Several BitTorrent metafile storage servers exist and almost every new .torrent file is uploaded to them. A simple downloader script tries to retrieve the metafile with very short timeout of 2 seconds, the first server that replies the file is used. The tracker look-up process uses BitTorrent search engines, currently it is primary torrentz.com.

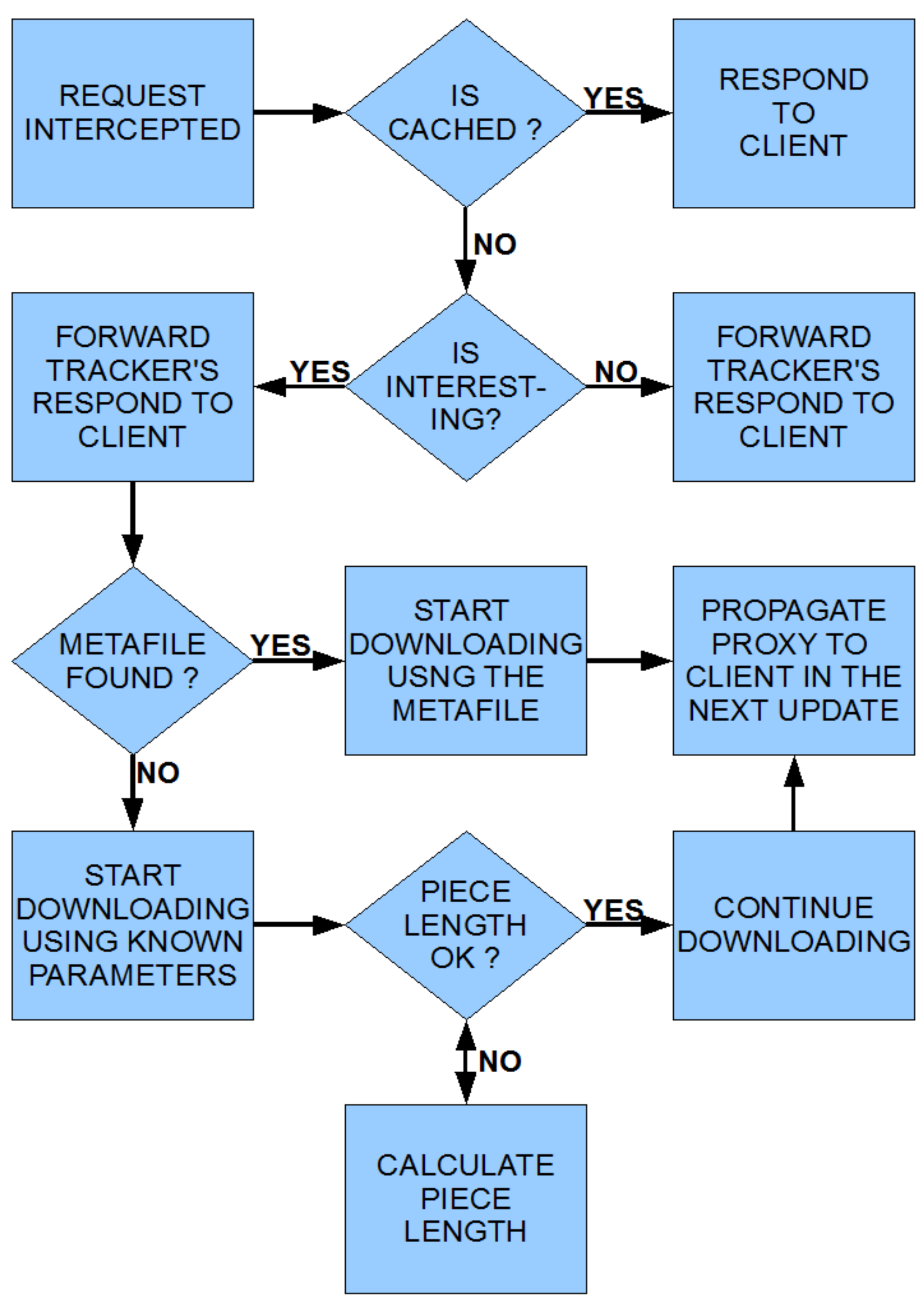


Figure 15: Original prototype simplified application logic.

4.5. HW requirements

The presented solution does not require special hardware; its requirements can be compared to normal file server for the given bandwidth. Although minor modifications consequent from the specifics of the protocol would greatly enhance its performance. The most heavily stressed component of a proxy is always the storage subsystem. Because the content will be mostly downloaded just once and then multiple times read the ideal approach would be to combine cheap high-capacity storage (using classic disks) with a extremely fast read buffer (using SSD technology). The downloading of content is a relatively slow process, thus it can be cached directly onto the high-capacity storage. From there it can be on-demand copied into the read buffer. The idea is to serve all requests from the read buffer. Because it has limited capacity only the currently most requested content should be held available on it. [11]

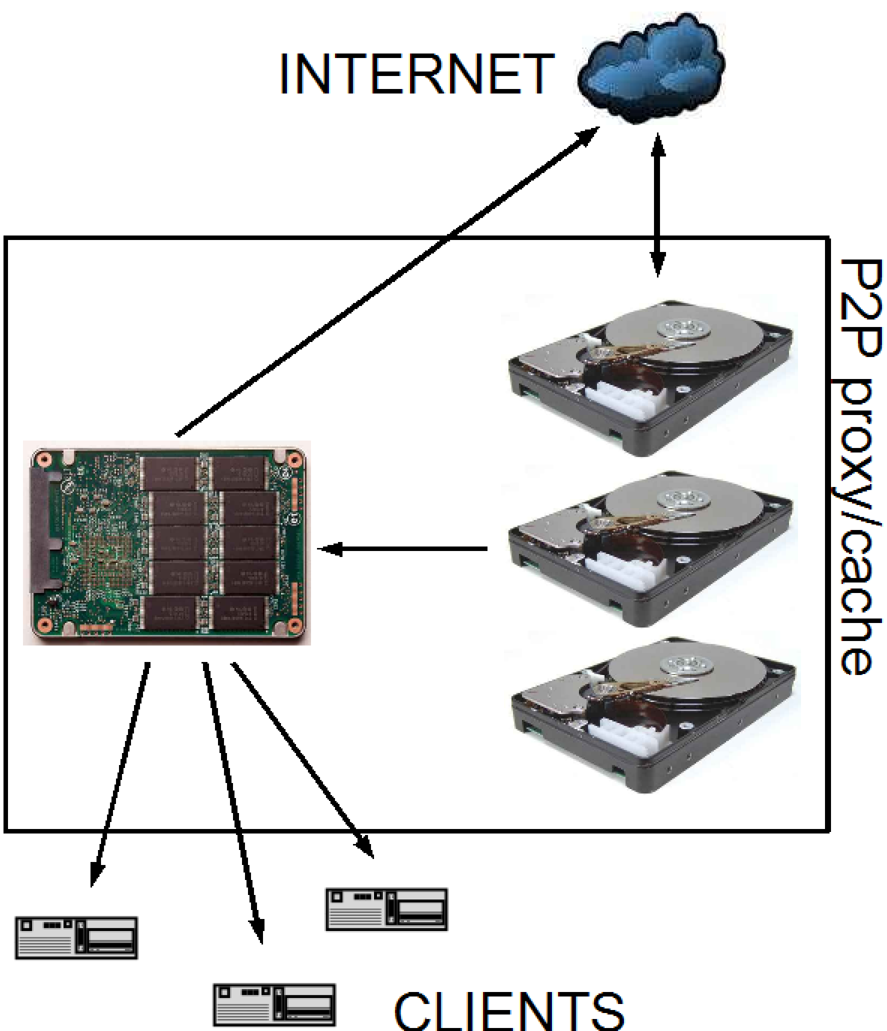


Figure 16: Schematic representation of the proposed storage subsystem, using SSDs for holding most frequently requested content.

4.6. Software architecture – first prototype

The first prototype used to intercept the tracker HTTP requests using iptables [15] string matching rules. Then they were redirected to a virtual interface where tshark [16] with another much more strict string filter made sure that only the real Tracker requests were intercepted not just requests containing the "right" key words. Tshark also parsed the needed information for the MiTM attack. These arguments were then passed to Nemesis [17] which initialized the session hijack. From that point on the Proxy/Cache was the Tracker for the client. The situation was basically the same for HTTPS, only the initial iptables matching couldn't be done using string filters as it was not possible to see into the HTTPS payload. Instead it was necessary to intercept requests based on "known trackers IP addresses". Modified Ctorrent [18] (a minimalistic console BitTorrent client) was used for downloading the content. Every instance of Ctorrent was running on a separate port, this made it easy to differentiate what content is available on what port. Ctorrent was modified to be able to download the content of a torrent without the metafile (this approach is needed only when the metafile cannot be found using BitTorrent search engines).

Another problem with downloading torrents without the initial metafile is the process of checking for bad downloaded pieces, as the proxy does not have the SHA hashes to check the downloaded data with. This was solved by another small modification of Ctorrent, as the proxy was the only peer the client could download the content from (if it was already cached), when it provided a bad downloaded piece the client would ask for it again and again as the client normally has the metafile and recognizes the pieces as damaged. This behavior can be easily recognized and the identified piece re-downloaded from the network and provided to the client. Another solution would be to add more sources (clients) into the tracker reply except of the Proxy/Cache, so when the data cannot be downloaded from the Proxy/Cache it will be downloaded from a different source. As usually only a small percentage of blocks are bad the overhead created by adding more clients into the reply should be minimal.

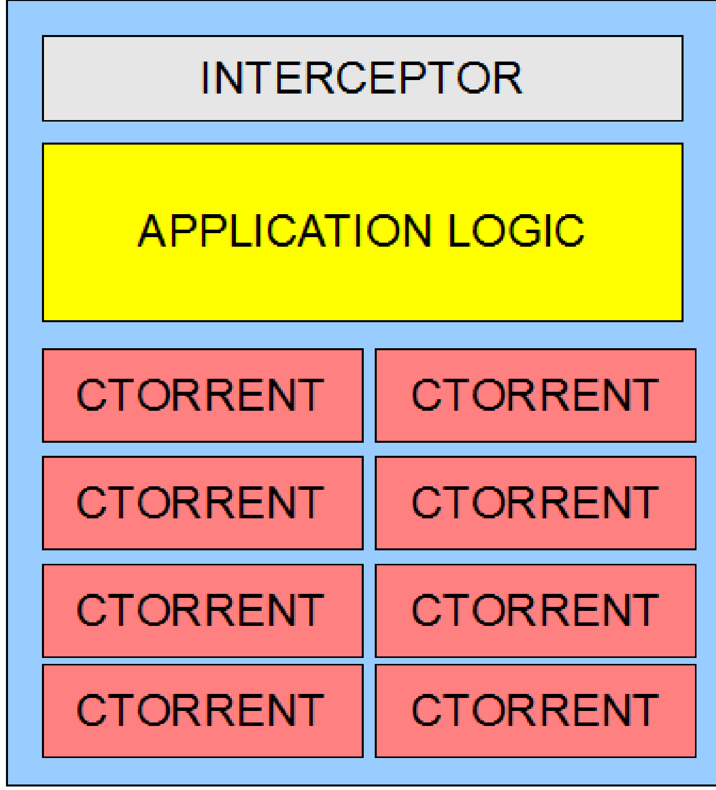


Figure 17: Simplified schematic representation of P2P Proxy/Cache first prototype software architecture.

4.7. Software architecture – current state

The first prototype did what it was supposed to do, but it was not scalable enough. Therefore it has been decided to optimize the whole solution by:

- Using OpenSolaris as the OS
- Develop own automated interception/recognition/MiTM engine
- Develop own Downloader
- Support massive parallelism

Except of optimization, it was necessary to develop a communication protocol between the Proxy/Cache components that would allow the design to be flexible. A design that would enable the components to be distributed across network on several different machines, add or remove components at any time etc. Also it was decided that because of the enormous volumes of collected data a graphical and analytical user interface needed to be developed.

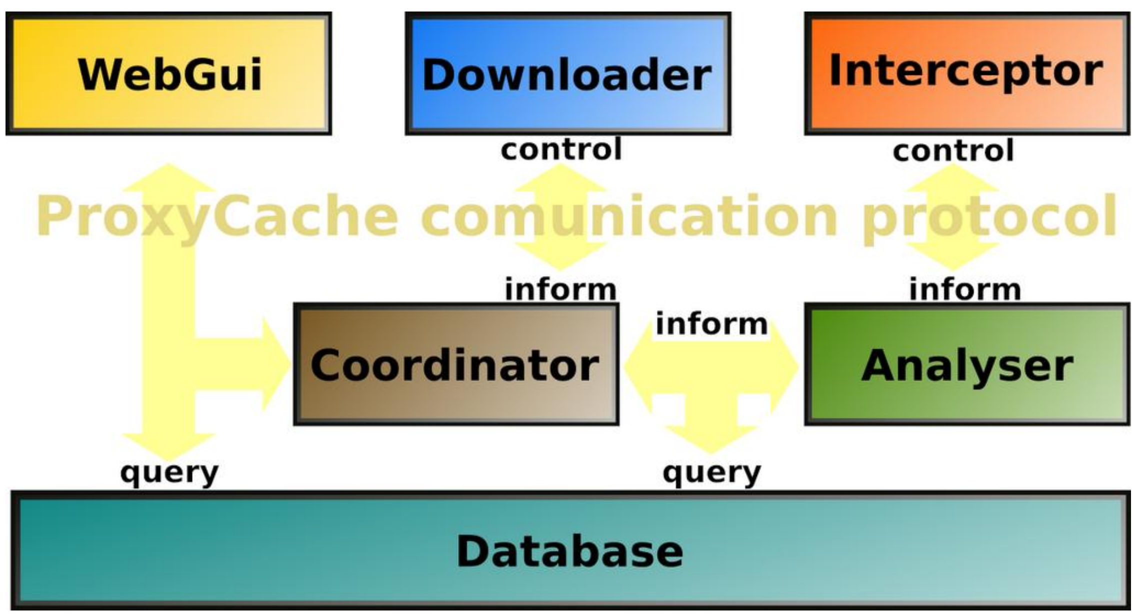


Figure 18: Schematic view of the current software architecture.

4.7.1 OpenSolaris

OpenSolaris is a free and open UNIX system; it is based on Solaris 10. Except of enterprise grade stability and performance it provides many interesting features:

- ZFS – supports transparent combining different disks into "pools" in which SSDs are used as fast buffers what results in massive (20x +) I/O performance, also it eliminates redundancy on the block level, self-healing and self-managing [19]
- Nemo (a.k.a. GLDv3) framework – greatly improved performance due to direct function calls and packet chaining between IP and device driver, lower CPU utilization, use of advanced NIC features such as stateless offloading [20]
- CUDA support – GPGPU can be used for parallel computations
- DTrace – detailed performance analysis of every detail in the system

4.7.2 Automated interception/recognition/MiTM engine

Using the features of the Nemo framework we succeeded in building a dedicated sniffing kernel module. This module is called kceptor because it is the kernel part of the Interceptor Proxy/Cache module. It dumps all traffic on a given interface and using shared memory mapping provides it to user space where uceptor (u – stands for user space) matches it against given rules. This matching is done using regular expressions.

After a match is detected the MiTM attack using the data from the matched request is commenced and a TCP session hijack injects the HTTP redirect. This topic is covered in-depth in another publication from our team. [14]

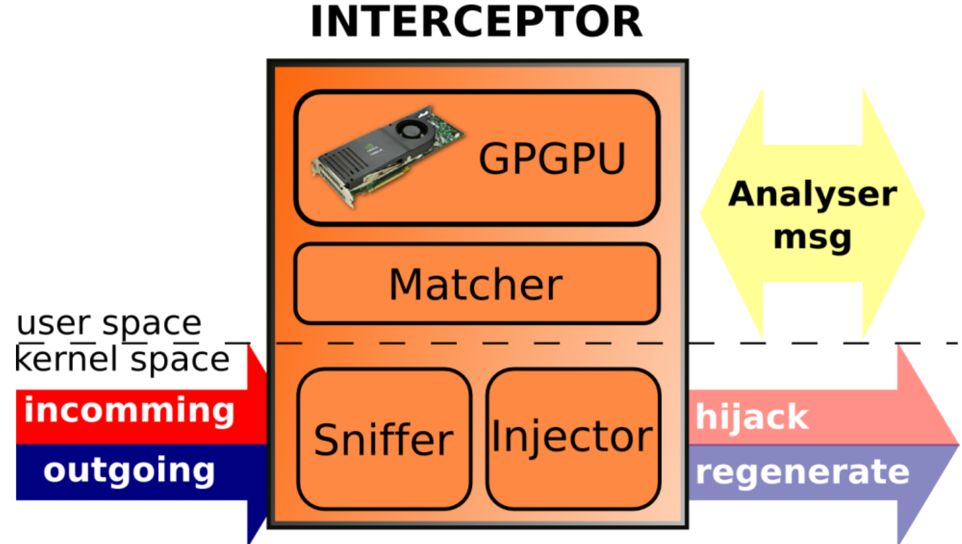


Figure 19: Interceptor schematic overview.

4.7.3 The Downloader

The downloader is build using the Rakshasa Libtorrent API, mainly because it's superior performance. The library differentiates itself from other implementations by transferring data directly between file pages mapped to memory by the mmap() function and the network stack. On high-bandwidth connections, it is claimed to be able to seed at 3 times the speed of the official client. [8] To support hundreds of simultaneous downloads hash checking is by default disabled to save CPU time. Because every download gets the list of all known trackers and its max_peer count is unlimited it can create thousands of connections, with hundreds of downloads the TCP stack and system settings need to be tweaked to avoid creating a performance bottleneck. For this reason the system net_buffer_size and receive_buffer_size variables are set to their maximal values. Also the number of incoming connection is set using the system variable tcp_conn_req_max_q to its maximal value; that same is done for the max TCP half-open connetion limit - tcp_conn_req_max_q0. All ulimit settings are set to unlimited too. Of course all this settings tuning is done at the expense of the RAM usage.

4.7.4 Support massive parallelism

The recognition and analysis of large amounts of data using a few simple algorithms is the ideal usage scenario for GPGPU. For example the regexp check in the tracker request recognition could be implemented using parallel computing very easily. The analysis/data-mining of the Proxy/Cache database could be also performed on a GPU. This would not only free the CPU resources for other tasks but also greatly enhance the performance of the whole system as for example on a Nvidia GeForce 8800 which is a two years old graphics card it is possible to string match 16Gbps of traffic. Current high-end graphics cards have more than 4,6 TFlops, that's almost 9 times more than the mentioned GeForce 8800. The Surricata IPS/IDS GPGPU string matching implementation can be used to achieve this goal. [21]. But until now it was not required as the current CPU-only regexp implementation does not generate substantial CPU load (under 2 percent).

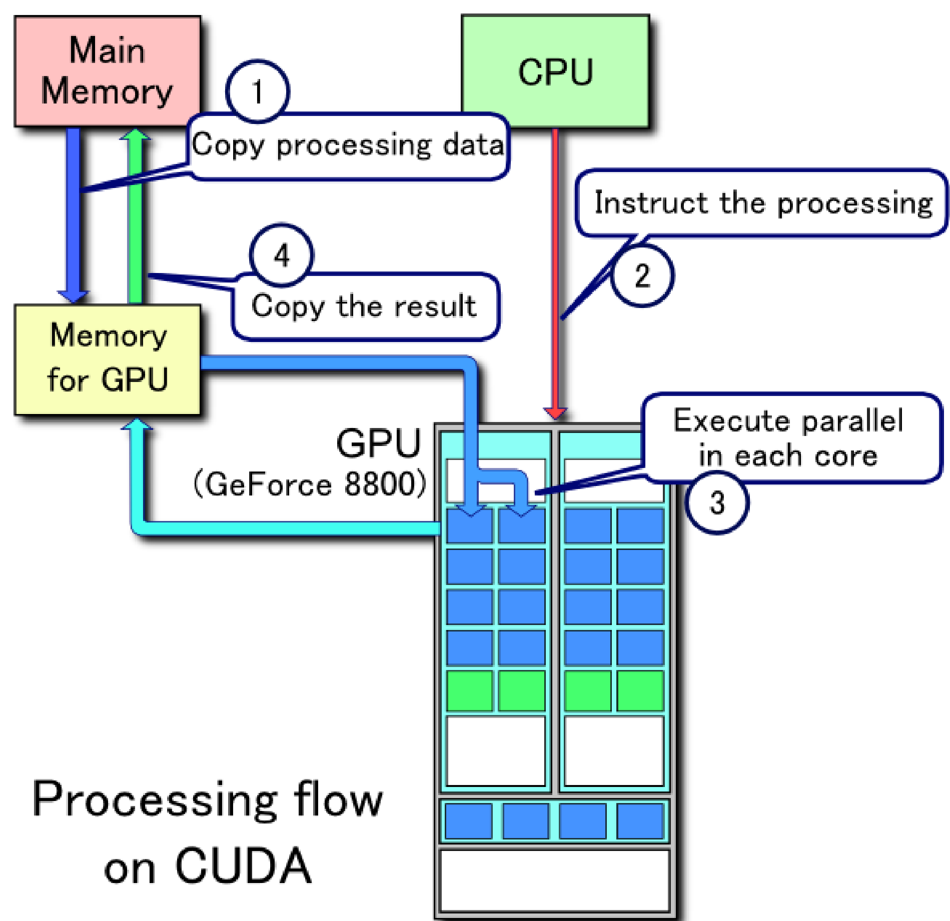


Figure 20: CUDA.

4.7.5 Proxy/Cache Communication Protocol – PCP

Proxy/Cache has to support unlimited number of components that can be distributed over network, PCP was designed appropriately. It uses UDP to avoid TCP overhead and its own simple transport control mechanism to guarantee message delivery. The PCP messages are in JSON format [22].

Structure of an example message:

```
{
    "header" : {
        "messageID" : "e9307a6601c8d8e",
        "responseID" : "abe95cb779384cc",
        "senderID" : "interceptor_5AE2",
        "listener" : "147.232.103.100:7810",
        "confirmation" : "147.232.103.100:6350"
},
    "message" : {
        "protocol" : "bittorrent",
        "event" : "intercepted",
        "parameters" : {
            "infohash" : "69f5e4bc6570ddd81477880d0e1a53da8653012d",
            "request" : "GET...",
            "client" : "147.232.145.178:31945"
        }
    }
}
```

An Application Programming Interface has been developed to make implementing PCP into Proxy/Cache components easier. Example of the API usage:

```
CommunicationProtocol::AddRecipient("Coordinator", "147.232.103.101", 6666);
CommunicationProtocolMessage * pMessage;
pMessage = new CommunicationProtocolMessage(7810, "abe95cb779384cc");
    (*pMessage)["protocol"] = "bittorrent";
    (*pMessage)["event"] = "intercepted";
    (*pMessage)["parameters"]["infohash"] =
    "69f5e4bc6570ddd81477880d0e1a53da8653012d";
    (*pMessage)["parameters"]["request"] = "GET...";
    (*pMessage)["parameters"]["client"] = "147.232.145.178:31945";
CommunicationProtocol::Send("Coordinator", pMessage, 6350);
```

A control mechanism has been developed that ensures every message is received. PCP acknowledges every message. If the sender doesn't receive an acknowledgement during the defined time frame it automatically resends the message. The sender retries only a defined number of times. Acknowledgement message format:


```
{
    "recieved" : {
        "messageID" : "abe95cb779384cc",
        "recieverID" : "coordinator_8D2A",
        "confirmation" : "147.232.103.100:6350"
    }
}
```


The timeouts and number of retries are not statically given instead they are calculated based on the protocol statistics for each component. Statistics collected: Latency, LatencyAvg, LatencyPeek, LatencyLast, MessageLoss. The meaning of the variables should be clear from their name; the last one MessageLoss collects the percentage of lost messages.

4.7.6 Proxy/Cache Database

Proxy/Cache can perform optimally only when it will analyze all the information it has and react on the changes instantly. An example has been already mentioned, the disk space cleaner process which keeps the disk space reasonably full and keeps only the most popular content in it. This can be possible only if all the statistical data is nonstop being analyzed. Not only popularity is statistically important but all other parameters too as one day Proxy/Cache will be able to real-time decide what actions will result in the best cost effectiveness. For example it will be able to prioritize certain content in terms of network or disk capacity etc. The full structure of the database can be found in the Appendix.

Currently Proxy/Cache is facing a problem with enormous database size; a two database model has been proposed and currently is being developed. One database would contain only the short-term entries of all parameters logged. Parallel GPGPU processing would analyze this data for correlations/trends and everything abnormal. The necessary algorithms already exist and are broadly used in the financial sector, open source CUDA implementations are available too [23]. Just the long term statistically important data would be then saved into the second long term database. Such approach reduces the database size without losing relevancy.

4.7.7 Data Visualization and Analysis GUI

Even if we succeed in reducing the database size it still will be impossible to get the real view of situation without effective means of visualization. Therefore a specialized GUI is being developed, because another thesis covers this topic we won't focus on it. [24]

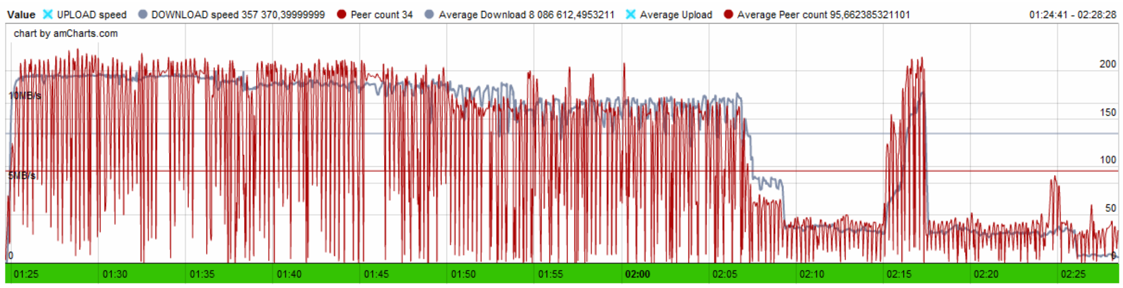


Figure 21: Example of Proxy/Cache data vizualization.

4.7.8 Coordinator

Coordinator provides the core application logic of the whole Proxy/Cache system. It receives the redirected tracker requests and replies with a list of peers containing the Downloader(s) and other peers if necessary. This enables the Coordinator to assign the download to any available Downloader(s) and thus load-balance the traffic.

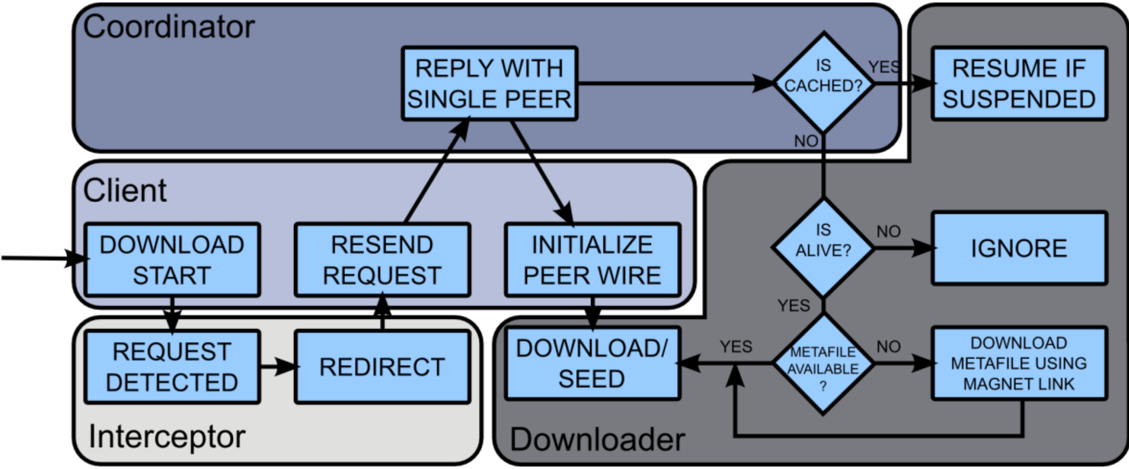


Figure 22: Core logic.

4.7.9 Analyzer

Analyzer is the component that does all the other functions that cannot be implemented in other components. For example the background disk cleaning e.g. removing of unpopular content on low disk space or selective pre-caching even before a request for that particular content is detected based on global popularity trends etc. All the other functionalities that will be mentioned in the last chapter are implemented in the Analyzer.

5. PERFORMANCE ANALYSIS

To measure the practical performance limits of the presented solution a series of tests in a controlled environment has been done. The P2P Proxy/Cache performance can be divided into three parts: the Interceptor which intercepts the Tracker requests, the download and the upload performance. The test hardware was a commodity PC with the following configuration: Intel Core2 Duo QC@2.4Ghz, 4GB RAM, 250GB 7200rpm HDD.

5.1 The Interceptor

The Tracker requests always use HTTP GET requests in a specific format. HTTP GET requests create only a fraction of the whole HTTP traffic, so to string filter them against a specific pattern is not a demanding task. In the performed tests it was not even possible to determine its performance impact as it was so negligible.

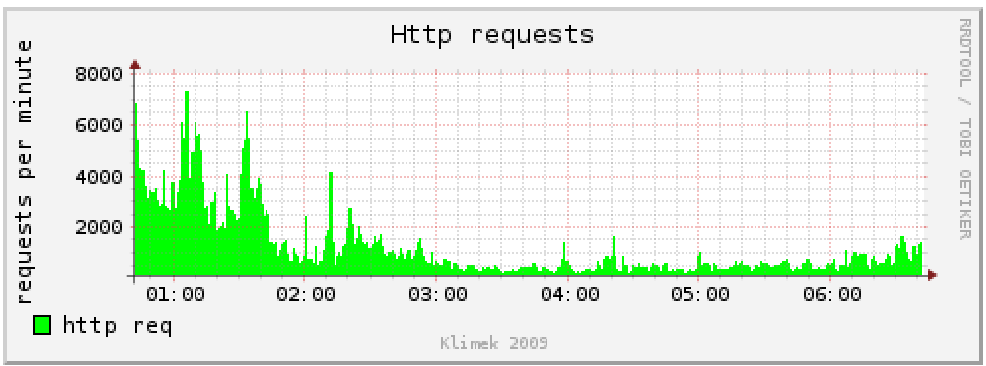


Figure 23: HTTP requests per minute on our campus network.

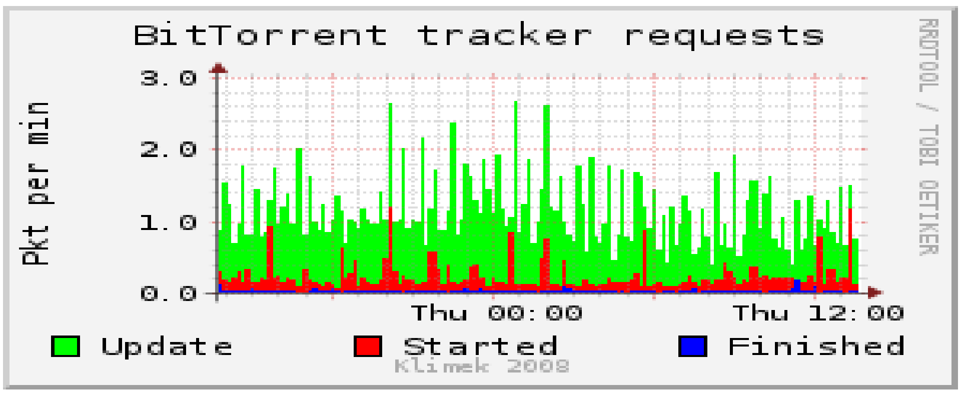
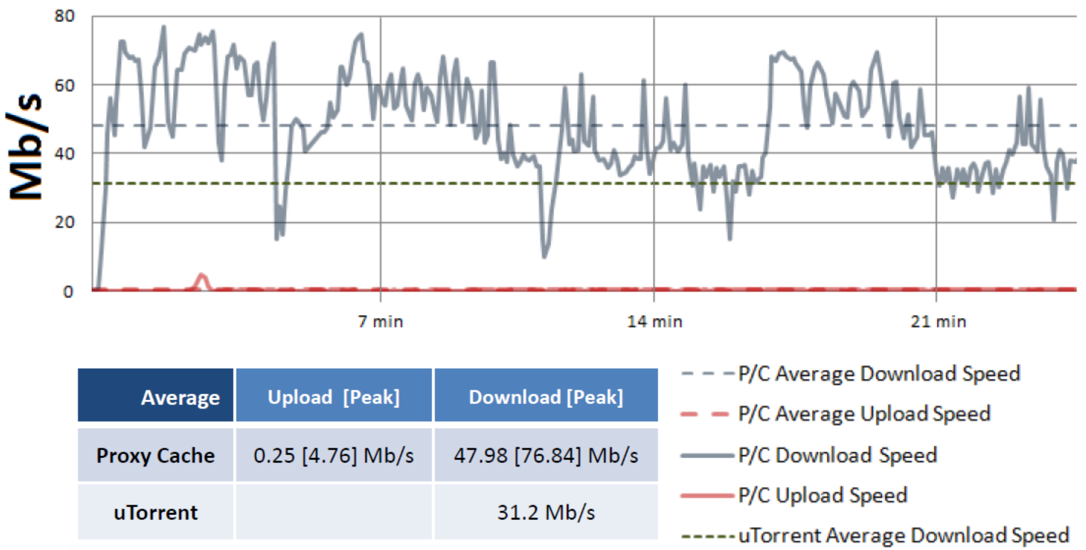


Figure 24: Monitoring of our university campus during 24 hours with approximately 3000 active users.

5.2 Download performance

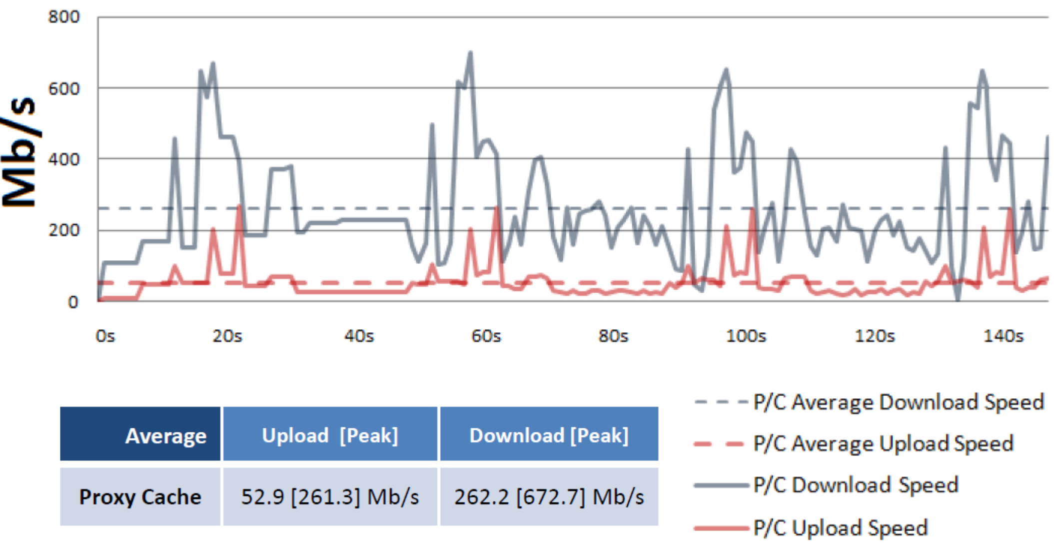
The Proxy/Cache needs to download the content as fast as possible; some already mentioned features are used to maximize its download speeds. For example when comparing with the currently fastest BitTorrent client uTorrent, Proxy/Cache downloads faster by 65%. The test has been made on the same machine, the same network connection and without any link aggregation.



Average	Upload [Peak]	Download [Peak]
Proxy Cache	0.25 [4.76] Mb/s	47.98 [76.84] Mb/s
uTorrent		31.2 Mb/s

Figure 25: Proxy/Cache Downloader is 65 percent faster than the fastest common BitTorrent client – uTorrent.

To determine the limits of the test server, an experiment on a 1Gbps Internet uplink was performed. The practical maximal download speed that can be achieved on a 1Gbps link is about 750~800Mbps. During the test runs the peak download speed was 672.7Mbps with average speed of 262.2Mbps. The upload speed was not limited in any way. These numbers are so high because of the used allocation mode – write first everything as it comes to limit the I/O operations and let the drive write linear where it has the best speed. The results were limited only by the disk used, which was a commodity 7200rpm HDD. After the download is finished and the load is lower, the pieces are placed into the correct order.

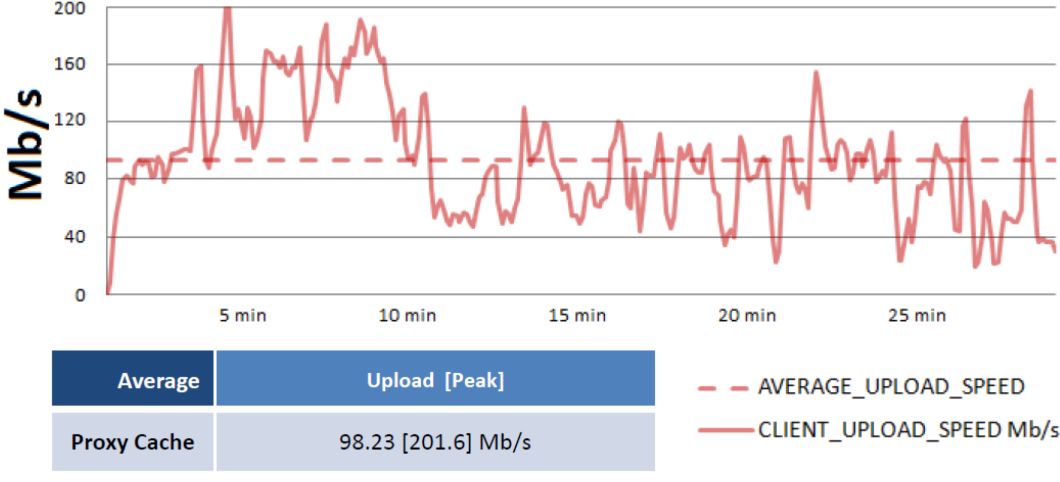


Average	Upload [Peak]	Download [Peak]
Proxy Cache	52.9 [261.3] Mb/s	262.2 [672.7] Mb/s

Figure 26: Download performance.

5.3 Upload performance

The seeding test was performed on a network with 15 active users, where each of them had 31 active downloads. That totals 465 torrent downloads. The results clearly show that it starts to be too much I/O operations for an HDD. The performance could be massively enhanced by using a solid state drive disk as an I/O buffer. If the last mile uplink is not a problem, the users in the same network segment can be allowed to download from each other too thus enhancing their download speeds even more.



Average	Upload [Peak]
Proxy Cache	98.23 [201.6] Mb/s

Figure 27: Upload performance.

5.4 Test summary

The test results show that an ordinary desktop PC is able to function as a P2P Proxy/Cache in a network with tens to hundreds of active users with an uplink of up to 1Gbps. The results also clearly demonstrate that the most important factor when speaking about performance is the amount of RAM and the speed of the disk subsystem. [25]

6 LEGAL ISSUES

Peer-to-Peer networks clearly represent a legal issue, numerous trials with the users of P2P systems or people behind the BitTorrent trackers are everyday in the news.

Let's summarize the facts:

- Caching content as defined by the DMCA [26] is not illegal, even storing and relaying copyrighted content for the purposes of caching is not illegal [27].
- The article 6 of IPRED gives the power to the interested party to apply for evidence regarding an infringement that lies in the hands of the other party to be presented. The only requirement is for that party to present "reasonably available evidence sufficient to support its claim" to courts. [28]

Thus, according to this interpretation using caching as described it is possible to legally defeat current monitoring mechanism and by that protect P2P users as their activity can no longer be monitored.

7 OTHER POSSIBLE ADDITIONS

P2P Proxy/Cache opens completely new possibilities because of its transparent automated Man-in-the-middle approach. With only minor modifications it could provide Flash/generic HTTP proxy/caching too. Also, it could revolutionize the techniques used to fight botnets.

7.1 Flash/generic HTTP proxy/caching

The growing popularity of audio and video streaming is undeniable, statistics show that it is taking over P2P users [1, 2]. Adobe's Flash platform, that is used to embed multimedia content into Web pages, is the uncontested number one. It makes up between 60 and 83 percent of all streaming traffic. Currently there is only one open source project that is to certain level able to cache Flash traffic – specifically YouTube.

"Squid is a caching proxy for the Web supporting HTTP, HTTPS, FTP, and more. It reduces bandwidth and improves response times by caching and reusing frequently-requested web pages. Squid has extensive access controls and makes a great server accelerator. It runs on most available operating systems, including Windows and is licensed under the GNU GPL." [29]

Current Squid YouTube caching mechanism is based on the fact that every YouTube video has an identifier that can be clearly parsed out of every user´s YouTube request. With this identifier it is easy to download the video then to the local cache and provide it when requested again. As the proxy sees the entire HTTP communication, it can parse the session token too and use it directly to download the video by using a simple HTTP GET request on www.youtube.com, for example like this one:

/get_video?video_id=wnD5FGnLyHE&t=vjVQa1PpcFMPn_dyBPjk_oc-mJyncuATPobUOezVUjM

Marked with red is the video identifier and blue is the session token that allows multiple downloads per one session token.

As already mentioned this is a very easy mechanism, but what about all the other video sites on the Internet? What about duplicate content with different URL? What about file storage services that use session strict session protection? The possibilities of Squid end

here. The only possible generic flash/http caching cannot be build upon URL checking – the classic mentioned approach – checking the user´s requests but instead checking the server´s reply can clearly identify all content at all times. This is a fundamental thinking shift but the Proxy/Cache is able to do it right out of box. The logic could look like:

- Check the parameters in the server´s HTTP reply:
  - Content-Type
  - Content-Length
  - Hash a few random blocks of the data section

All the other HTTP reply parameters could be artificially forged to avoid caching, Content-Type and Content-Length are necessary for all the media players so this two parameters can be used as identifiers. Further, random blocks of the data section will be selected; their addresses and a hash of a few bytes will be saved. So when there will be content with the same Type and Length, only the few hashes will be compared and if there will be a match then this has to be identical content. The false positive ratio is only a factor of the selection of the random blocks, with sufficient hashes of random blocks all content can be identified. GPGPU could be used for this logic as it is a simple algorithm that needs to be run parallel on a great number of threads. Until it will be decided if it is redundant data, the server´s reply will be postponed. If decided that this content is already cached, the content will be provided transparently from the cache. If not already cached the server´s reply will be allowed to pass to the client.

Flash players often embed further information and content into the video, for example logos, commercials, subtitles etc. This content cannot be cached because it is mostly dynamically generated and it would be contra productive to the content provider to cache it. Also, this content is very often connected to a precise position in the video, which means the player has to update the server during the playback. All this is not a problem to handle for the Proxy/Cache because the MiTM attack targets only the video data TCP session all the other sessions stay untouched. If there would be a situation that the player would send an unrecognized request in the hijacked session, for example after the video loading process is completed it may request some embedded information

like links to other videos, then the Proxy/Cache transparently sends the request to the original server in the hijacked session and resends the reply transparently back to the client. This requires keeping the hijacked session up all the time the client downloads data from Proxy/Cache even between the proxy and the original content provider but it guarantees 100 percent compatibility and transparency.

7.2 Cleaning the Internet from Botnets

Botnets exploit the fundamental design flaws of Internet. They utilize great number of bots (infected PCs) to send Spam e-mail messages or perform Distributed Denial of Service Attacks (DDoS). Both this actions use simple brute force sending of millions e-mail messages (SMTP protocol) or various DDoS techniques (SYN flood, etc.) The protections against these threads are various spam filters either directly at the e-mail server or at the e-mail client and special networking equipment that supports reducing the impacts of DDoS. All this solutions have one thing in common; they are placed near the “attack victim”. They don’t fix the problem cause; they only limit its effects. Proxy/Cache can change that.

Intrusion Protection Systems / Intrusion Detection Systems (IPS/IDS) are systems designed to monitor the network flow and catch attack attempts by recognizing various patterns and taking appropriate actions. This technology is widely used in the enterprise sector for years now, but could it help against Botnets?

Proxy/Cache is an automated transparent detection and MiTM system, it can detect any kind of “interesting” communication and take actions, similarly to an IPS/IDS. What is different is the placement of the system, Proxy/Cache is to be placed where the users are to reduce the redundancy and thus save capacity/enhance user experience. But the users are also the source of the mentioned problems with botnets. Proxy/Cache can therefore fight against this problem on three levels:

- Protect the user from becoming infected
- Recognize and block the Botnet coordination effectively crippling it
- Block spam and DDoS at its source

These three levels are also fallback options, if the Proxy/Cache doesn't succeed in protecting the user from becoming infected, it still can block the botnet coordination. Without coordination a botnet is useless as the infectors mostly don't contain any attack logic, they are as small and inconspicuous as possible. After they infect the machine they download additional components that realize the attack. Therefore infectors are also called droppers. If the dropper won't be able to get instructions from the botnet network and download additional components it won't be any threat. If Proxy/Cache doesn't succeed in recognizing and blocking this botnet communication, it will block its main purpose as it is really simple to recognize a Spam sender or DDoS attack bot due to the abnormal levels of redundancy which is our system designed to monitor.

7.2.1 Protect the user from becoming infected

First of all it is important to try to protect the users from becoming infected. The most common infection vectors are [30, 31, 32]:

- Unpatched operating systems
- Unpatched applications – like the recent IE6 hole used to hack Google [
- Infected downloads
- User benevolence – run dangerous ActiveX components etc.

All the mentioned vectors have a very important feature in common, they either directly run an infected binary or use some mechanism to download it and then run it. Both ways Proxy/Cache will detect it. Further, in most cases it needs to be directly executable, this is because the user needs to be able to run it, as for example a download or a mail attachment. In the case of an exploit a minimalistic shell code is injected thru the OS/App vulnerability that downloads additional components, again in most cases it needs to be able to run the executable directly. [30, 31, 32] We are using the wording "in most cases" because a layer of obfuscation/encryption can be implemented, of course this would make the task harder but not impossible. Still this is only the first fallback option.

How would such a download then look like? What protocol would be most probably used?

- User download´s a infected file from the web – HTTP or P2P
- User benevolence – ActiveX component installs/downloads malware – HTTP
- OS/App exploit – needs to use the OS/App resources to download the infector – HTTP

It's clear that in most attacks the initial infector will need to go thru one of the already by Proxy/Cache monitored protocols. A security layer that would check the monitored data flows on suspicious downloads could be easily added.

Security layer model:

- Monitor the data flow, search for executable code (in any form, not only PE)
  - Can be done thru:
    - Checking the URL
    - Checking content-type parameter in HTTP responds
    - Checking the first bytes of the data section in the HTTP respond to see if the header is correct and adequate to the content-type defined
    - Checking the content of the .torrent metafile
- If a executable is found
  - Extract it and send for analysis to a dedicated component
  - Keep the session up until analysis finishes
  - Most executables are not big files, depending on the resources available a size limit can be dynamically determined so that the analysis speed takes a reasonable amount of time
- If not infected then provide it to the client
- If infected use the approach StopBadware uses e.g. warn the user by redirecting to a site with a warning and options to continue to the file or to let it be [33]. If it

is a BitTorrent download then place an explanationary readme instead of the infected file

All the needed technology is already implemented in the p2p and generic http/flash caching process. The analysis would need to be a dedicated machine running antivirus software; scanning of small files is a fast process and can be speeded up by using GPGPU processing. Antivirus'es that support GPGPU processing already exist [34]. Also an Cloud AV can be used.

7.2.2 Recognize and block the Botnet coordination

Botnets can be classified according to the protocol used for command and control (C&C) as:

- IRC based
- HTTP based
- DNS based
- P2P based

The logic is always the same: *"After initial infection, in secondary injection phase, the infected hosts execute a script known as shell-code. The shell-code fetches the image of the actual bot binary from the specific location via FTP, HTTP, or P2P. The bot binary installs itself on the target machine. Once the bot program is installed, the victim computer turns to a "Zombie" and runs the malicious code. The bot application starts automatically each time the zombie is rebooted. In connection phase, the bot program establishes a command and control (C&C) channel, and connects the zombie to the command and control (C&C) server. Upon the establishment of C&C channel, the zombie becomes a part of attacker's botnet army. After connection phase, the actual botnet command and control activities will be started. The botmaster uses the C&C channel to disseminate commands to his bot army. Bot programs receive and execute commands sent by botmaster. The C&C channel enables the botmaster to remotely control the action of large number of bots to conduct various illicit activities."* [30]

If the first fallback option didn't block the fetching of the actual bot binary then it will be necessary to block the C&C traffic. Even if it uses different protocols, it still has some common characteristics, most researchers focus on finding correlations between various factors and complicated statistical analysis [35]. But in fact this is not necessary. Redundancy will always reveal it.

As in every automated system the actions taken by the C&C protocol will be repetitive and thus create redundancy that can be detected by the already presented caching mechanisms. Of course this is true only if protocols supported by the Proxy/Cache will be used for C&C. The protocols currently unsupported like IRC and DNS can be monitored easily too because IRC popularity is hitting historic minimums so basically every IRC communication is suspicious. Passive DNS redundancy monitoring support can be added anytime as it is only an anomaly detection mechanism e.g. too many DNS requests or too many name errors. Where "too many" is dynamically defined, based on the normal levels detected in the given network.

For example the Bobax botnet uses HTTP for C&C, it periodicaly connects to the C&C server with an URL such as

http://hostname/reg?u=[8-digit-hex-id]&v=114

The server then sends commands via a HTTP reply. Commands can tell the bot to send spam or to update its binary etc. This process happens very often, some reports say up to every 5 seconds [36]. This means the level of detected HTTP reply redundancy will be extreme. The generic http caching proposed earlier is not looking only at the content parameters in the http header but also at the payload, it reads the payload header and makes hashes of block at random positions so that redundancy can be clearly identified even if it comes from different sources – servers. This is particulary handy when speaking about botnets which use decentralization and techniques like fast-flux more and more often [32].

7.2.3 Block spam and DDoS at its source

The main motivations behind Botnets are money that is made on sending spam or providing DDoS attacks. These actions are distributed amongst the infected PCs. If these PCs wouldn't be able to perform the given tasks it would become harder to make profit and thus the motivation behind Botnets would cease to exist.

If both mentioned fallback options fail, this third line of defense has to work. As already mentioned, sending spam or performing a DoS attack are highly repetitive actions that can be detected by simple anomaly detection mechanisms. Botnets profit because right now there are no such mechanism implemented in the ISPs network, the ISPs just don't care if their customers are part of a botnet or not as the botnet traffic doesn't negatively affect the source ISP's network in any way. If no other, there is one reason why ISPs should care, phishing. Phishing attacks are one of the most dangerous, for example let's take a look at this screenshot:

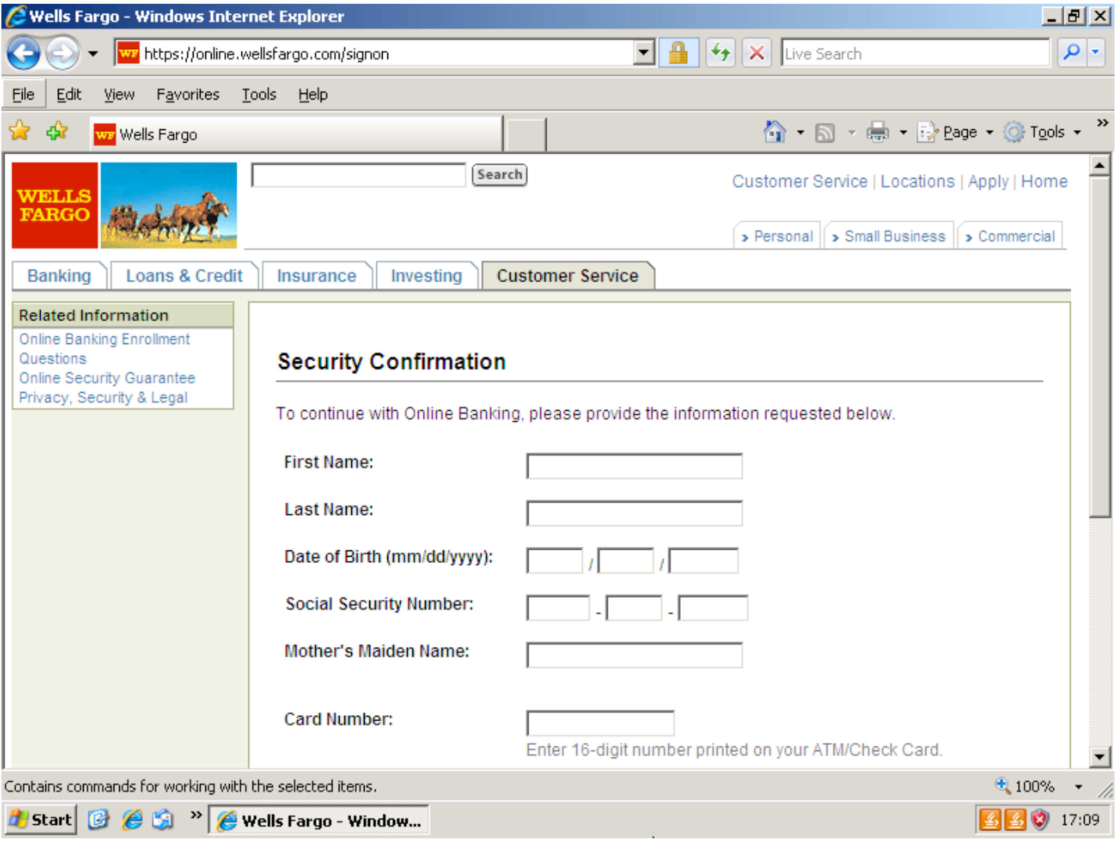


Figure 28: Phishing example.

The whole page seems completely trustworthy, it even uses https. There is just no way of knowing what danger awaits there. It is a phishing attack. The local PC is infected and when it detects that the browser moves to a login site it captures all the data – unencrypted and then sends to the control server. Yes, this is one of the functionalities of a botnet too. It's called Man-in-the-Browser attack. So the infected clients are not only senders of spam or DDoS attackers, where the ISP doesn't need to care but also in direct danger of losing their bank account / credit card credentials. This is a real threat for everyone.

Technically a simple anomaly detection mechanism will be enough to provide this final line of defense. If a greater than normal number of possible spam/DoS actions are detected block the used protocol for the client. Inform the client what just happened and that he is most probably infected and in great risk. Also, block all SMTP relaying in the segment as there is no point in running a SMTP server on a desktop machine – only botnets do that. This could well be integrated into the ISP's Fair Usage Policies – anomalies will be blocked and the client will be warned of the danger.

Basically the effect of such blocking would be the decrease of the usability of infected PCs to a level where it wouldn't be recognizable from normal activity. And thus greatly limiting the usability and profitability of botnets even if all three fallback options fail.

7.2.4 Prototype architecture

Based on the current state of P2P Proxy/Cache a prototype of the mentioned 3-layer fallback system was designed. It was named Protect/Warn/Disable (PWD).

7.2.4.1 Protect

To protect the users from becoming infected a transparent network antivirus would need to be created. A device that would protect from downloading malicious code in any of its forms. This prototype uses two methods how to achieve this goal, first of all it monitors all HTTP GET requests, redirects the user to a temporary "placeholder" port and analyses the respond. Redirecting all GET requests wouldn't be optimal as it would degrade the user's web experience. Thus only potentially dangerous GET requests are being redirected. In this prototype what is potentially dangerous is determined by the requested domain in the first place. It is not very likely that a trusted company domain

would distribute malware. A few examples of trusted companies could be microsoft.com, facebook.com, google.com etc. Due to the upcoming DNSSEC it will be even less likely as DNS hijacks will become harder to perform. How much requests need to be checked then? To answer this question the Interceptor module was modified to record all HTTP GET requests from our university campus. The recording started on 15th April and ended on 28th April totaling in 241 647 718 capture GET requests with a log size of 26GB. The analysis of this log showed 80 (92 resp.) percent of all GET requests went to only 0,15 (2 resp.) percent of the requested domains. Only Facebook accounted for over 25 percent of all requests. This results show that a dynamic whitelisting approach should be able to avoid the need to check about 80 to 90 percent of all requests. Where a domain would be whitelisted after reaching certain level popularity without any positive malware detection. All whitelisted domains would be from time to time rechecked to make sure they are still malware free. The next optimization step is to control only the requested files that can be potentially dangerous. All[1] requests going to non-whitelisted domains are redirected to a temporary port and the requested link with the original user-agent is downloaded by wget. The default HTTP timeout provides more than enough time for a scan. As URLs cannot be trusted the content-type and file header in the respond need to be checked. Most requests will be only harmless picture files as (JPEG, GIF, PNG, ICO) or HTTP files which are mostly safe for the user. The user is then redirected back to the original link as it seems safe to do so. But if a .exe/.dll file is detected then it is automatically uploaded to a Cloud AV - the current implementation uses the Jotti's free online malware scanner that supports uploading files via POST method and scans it using 20 different top AV products. If the Cloud AV returns at least one positive result the user is redirected to a warning site that contains a link to the Cloud AV scan result. If the user still decides to download the file, it is provided from the Proxy/Cache to avoid any possible looping. A hash of every file together with the scan result is stored in the database to avoid redundant requests.

Even if every .exe/.dll file from untrustworthy domains is checked by 20 top AV scanners it still does not protect the user completely. The AV miss ratio in binary downloads initialized from exploits is about 72 percent! [37] Every binary download

[1] except of session protected URLs – current Interceptor limitation

from every exploit is extremely dangerous and even if the AV doesn't detect it as an infected executable it has to be blocked. But how could an exploit be detected by a network AV? Currently it is done by deep packet inspection and matching against known patterns which is not an optimal solution neither by its performance nor effectiveness. A better solution would be to look instead for the effects of an exploit and that is the activation of an download where it shouldn't be. It's called a drive-by download. As of the day of writing this thesis there is only one research project focusing on the detection of drive-by downloads - blade defender. Blade defender uses a specialized kernel module to detect and block hidden downloads/execution. The statistics clearly show that exploits use pdf, java, flash or IE security holes. This means that if a pdf/java/flash/activeX content from an untrustworthy domain is detected it should be analyzed by blade defender for possible drive-by download exploits. As blade defender needs to run on a Windows machine preferably of the same OS version as the user uses and the link should be opened in the same browser as the user uses a cloud of windows virtual machines would be required. Because of the XEN support in OpenSolaris, Proxy/Cache is able to run this cloud locally. In the case of a drive-by download exploit detection the user is redirected to a warning page exactly as in the case of an .exe/.dll infection.

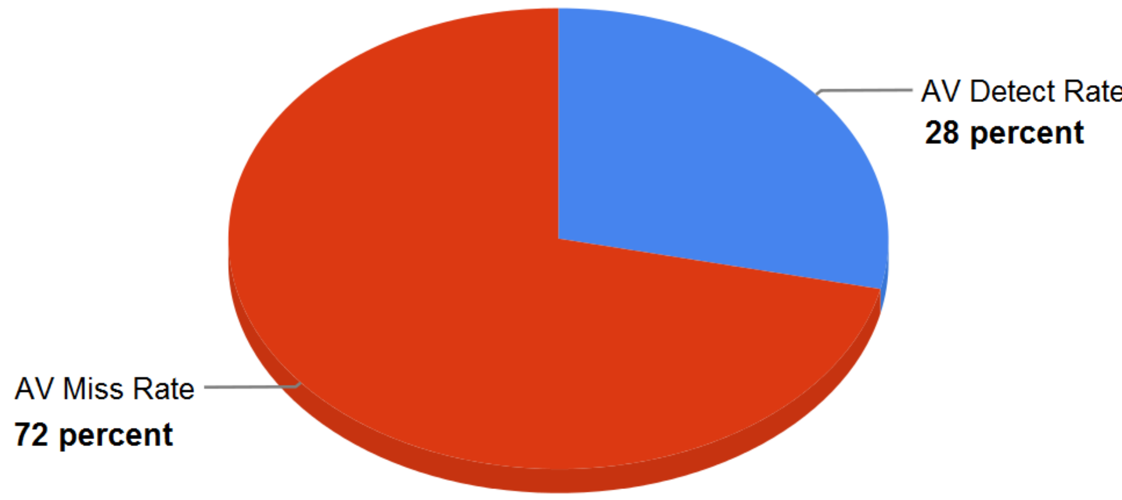


Figure 29: AV miss rate on drive-by downloads. [37]

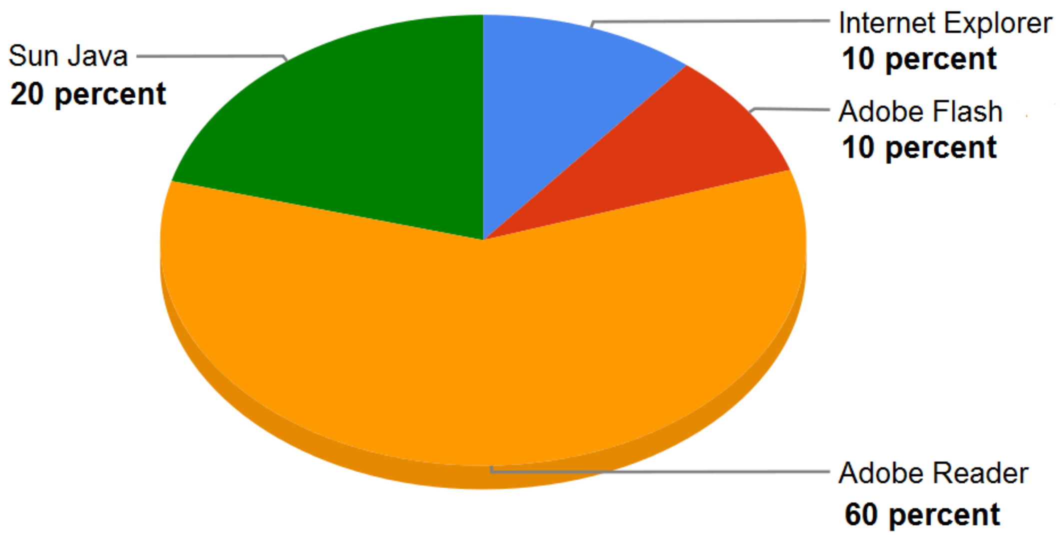


Figure 30: Applications targeted by drive-by exploits. [37]

7.2.4.2 Warn

No protection is perfect; there will always be some way how the user can get infected. But the fight is not over when that happens, as already mentioned blocking the droper of downloading further components will greatly reduce its usability. Infected hosts tend to generate specific traffic as they try to contact the C&C servers, tools to detect these patterns already exist. The most specialized is BotHunter which analyzes network flows and based on what it sees models an infection sequence as a composition of participants and a loosely ordered sequence of network dialog exchanges:

$$Infection\ I = \langle A, V, E, C, P, V', \{D\} \rangle$$

where A = attacker, V = victim, E = egg download location, C = C&C server, P = peer-to-peer coordination points, and V' = the victim's next propagation targets. {D} represents a set of dialog sequences composed of bidirectional flows that cross the egress boundary. [38]

Dialog Class	Dialog Event	Dialog Icon	English Description
E1[bh/rb]	Inbound Scan		<Host> was the target of an inbound scan
E2[rb] .	Inbound Attack	IN	<Host> was the target of an inbound infection attempt
E2[dns]	DNS Lookup to Client Exploit		<Host> performed a DNS query to a malicious host associated with client-side exploits
E3[rb]	Egg Download		<Host> downloaded a binary executable from an external source
E4[rb]	Malware C&C		<Host> appears to have exchanged command and control message with an external malware controller
E4[nbr]	Russian Business Net Connection		<Host> attempted to connect to a monitored Russian Business Network site
E4[dns]	DNS Lookup to Botnet C&C		<Host> performed a DNS query to a known malware control site
E5[rb]	Outbound Attack Propagation	OUT	<Host> is conducting outbound attacks, possibly to propagate a malware infection
E5[bh]	Outbound Scan		<Host> may be conducting an outbound IP address sweep
E6[rb]	Attack Preparation		<Host> is conducting activities associated with preparing to launch an attack
E7[rb]	Peer to Peer Coordination		<Host> is engaged in P2P communications associated with Malware coordination
E8[bh]	Malicious Outbound Scan		<Host> is conducting an IP address sweep using network ports associated with malware propagation
E8[rb]	Outbound to Malware Site		<Host> has connected to a known malware control site

Figure 31: BotHunter's current infection dialog set {D} detection coverage. [38]

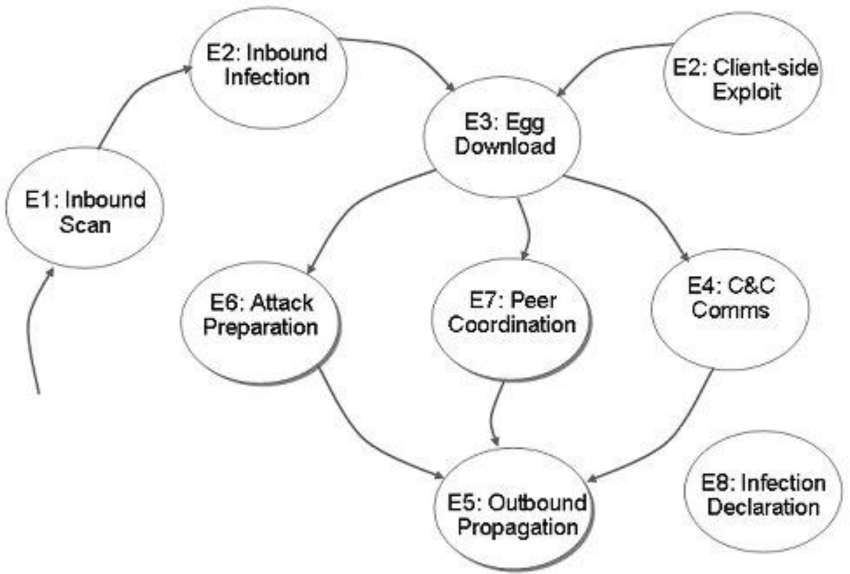


Figure 32: BotHunter Infection Lifecycle. [38]

From the BotHunter Infection Lifecycle it is clear that the presented Protect layer focuses on the Egg Download phase and on the Client-side exploit phase. For full protection a firewall is required to block any incoming vulnerability scans. Except of the mentioned most common infection vectors, the user can get infected from offline sources too (for example USB autorun infectors).

In any case, if BotHunter detects an infection, the user is warned about this fact by redirecting his HTTP sessions to a warning page. Until the user cleans the infection the HTTP redirect to warning page will occur more and more often to push the user for his own good into taking adequate actions.

7.2.4.3 Disable

If the Proxy/Cache won't be able to protect the user nor the warnings won't have any effect the last resort is to block all client's communication to avoid any possibility of the misuse of the infection for sending spam or participating in DDoS attacks etc. This can be done by implementing a PCP enabled daemon at the firewall to which Proxy/Cache can send a request for blocking a client. The client will need to take actions when his Internet connection will be reduced to seeing the infection waning page on every HTTP request with all other communication blocked.

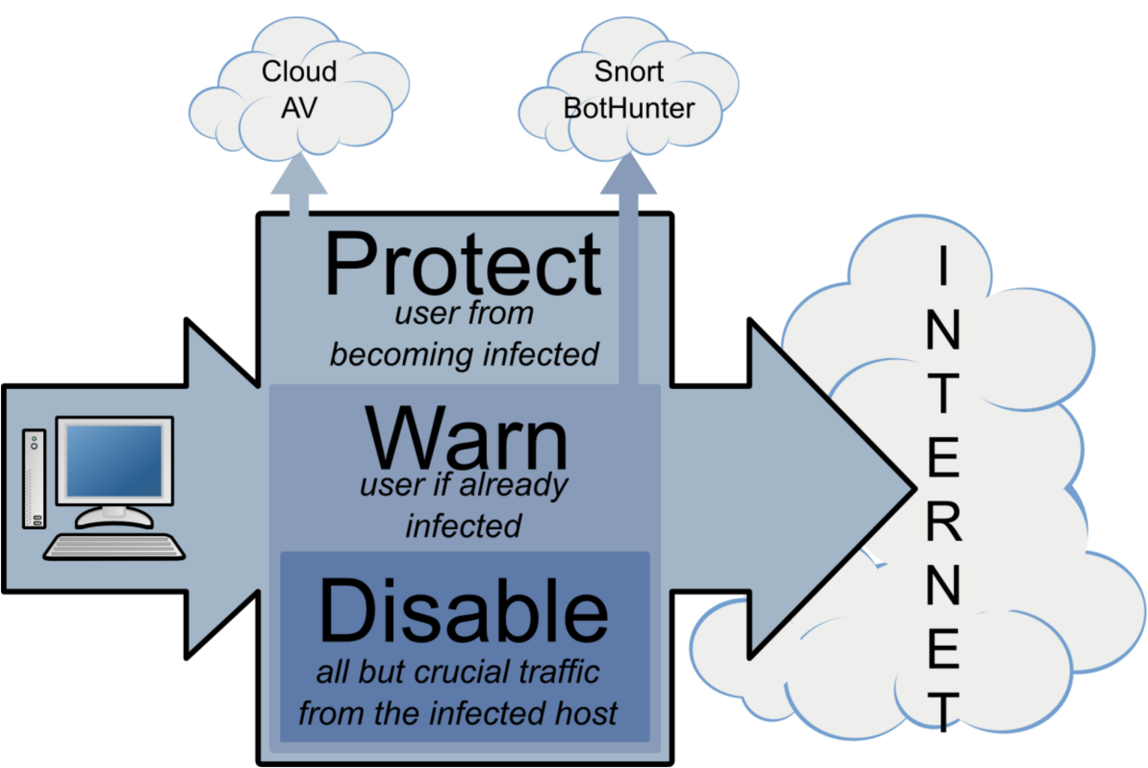


Figure 33: Protect/Warn/Disable schematic overview.

Conclusion

The presented solution by exploiting a found attack vector in the currently most popular and perspective P2P protocol - BitTorrent, enables the ISPs to stop fighting P2P and instead cooperate with the community on creating a more effective hybrid P2P network model. Based on existing studies an assumption that caching could be the solution for the increasing P2P networks traffic was made. To confirm this assumption calculations based on global data were performed and the results clearly showed that in fact a great level of redundancy exists. Therefore a novel transparent optimization approach was proposed and in several prototype stages demonstrated. Final tests were performed on the TUKE campus network and the results confirm that P2P Proxy/Cache is a valid solution for the P2P traffic problem. As presented redundancy also affects streamed video, the proposed generic HTTP/Flash caching provides novel means of optimization of this type content with until now unprecedented effectiveness. Even if Protect/Warn/Disable is only in prototype stage the author believes that its simple and generic logic will provide a beneficial added protection layer for network clients and that if it would be deployed on a global scale it could solve this kind of problems once and for all.

You know you've achieved perfection in design, not when you have nothing more to add, but when you have nothing more to take away. (de Saint-Exupéry)

Bibliography

[1] Schulze, H., Mochalski, K.: Internet Study 2007. ipoque, November 2007.

[2] Schulze, H., Mochalski, K.: Internet Study 2008/2009. ipoque, November 2009.

[3] PeerApp, “UltraBand Family overview”, 2007, [Online; accessed 23-November-2008], [Online], Available: http://www.peerapp.com/products-ultraband.aspx

[4] TheoryOrg, “Bittorrent Protocol Specification v1.0”, 2010, [Online; accessed 3-May-2010], [Online], Available: http://wiki.theory.org/BitTorrentSpecification

[5] BitTorrent.org, “The BitTorrent Protocol Specification”, 2008, [Online; accessed 3-May-2010], [Online], Available: http://www.bittorrent.org/beps/bep_0003.html

[6] BitTorrent.org, “DHT Protocol”, 2008, [Online; accessed 3-May-2010], [Online], Available: http://www.bittorrent.org/beps/bep_0005.html

[7] Wikipedia, "Local Peer Discovery", 2010, [Online; accessed 3-May-2010], [Online], Available: http://en.wikipedia.org/wiki/Local_Peer_Discovery

[8] http://libtorrent.rakshasa.no

[9] BitTorrent.org, “Extension for Peers to Send Metadata Files”, 2008, [Online; accessed 3-May-2010], [Online], Available: http://www.bittorrent.org/beps/bep_0009.html

[10] Klimek, I., Korenko, T., Keltika, M.: Project Proxy/Cache: Eliminating Redundancy - Internet’s Number One Enemy. IIT.SRC 2010, Bratislava, April 21, 2010, pp. 333-340.

[11] Klimek, I.: P2P Proxy/Cache. In Proc. of of the Third International Conference on Internet Technologies and Applications (ITA 09), Wrexham, UK, September 2009. [Online; accessed 3-May-2010], [Online], Available: http://s.cnl.sk/~klimek/ita09.pdf

[12] JAXA, “Overview of the KIZUNA (WINDS)”, 2008, [Online; accessed 23-November-2008], [Online], Available: http://www.jaxa.jp/countdown/f14/overview/kizuna_e.html

[13] Wikipedia, “Satellite Internet access”, 2008, [Online; accessed 23-November-2008], [Online], Available: http://en.wikipedia.org/wiki/Satellite_Internet_access

[14] Keltika, M.: Analýza a riadenie P2P prevádzky. Diploma Thesis, Technical University of Kosice, 2010.

[15] http://www.netfilter.org/

[16] http://www.wireshark.org

[17] http://nemesis.sourceforge.net

[18] http://www.rahul.net/dholmes/ctorrent/

[19] Sun Microsystems, Inc., “Solaris™ ZFS™ Enables Hybrid Storage Pools — Shatters Economic and Performance Barriers”, 2008, [Online; accessed 3-May-2010], [Online], Available: http://download.intel.com/design/flash/nand/SolarisZFS_SolutionBrief.pdf

[20] Sun Microsystems, Inc., “Project nemo: Nemo: A Framework for High-Performance Networking”, 2009, [Online; accessed 3-May-2010], [Online], Available: http://hub.opensolaris.org/bin/view/Project+nemo/

[21] http://www.openinfosecfoundation.org/

[22] Wikipedia, “JSON”, 2010, [Online; accessed 3-May-2010], [Online], Available: http://en.wikipedia.org/wiki/JSON

[23] Preis, T., Virnau, P., Paul, W., Schneider, J.: Accelerated fluctuation analysis by graphic cards and complex pattern formation in financial markets. New Journal of Physics 11, September 2009.

[24] Chalupka, M.: Grafická reprezentácia a analýza dát v prostredí P2P sietí. Bachelor Thesis, Technical University of Kosice, 2010.

[25] Klimek, I., Korenko, T., Keltika, M., Jakab, F.: P2P PROXY/CACHE, A NOVEL HYBRID P2P/CDN APPROACH TO HIGH QUALITY CONTENT DELIVERY. 7th Int. Conference on Emerging eLearning Technologies and Applications, The High Tatras, Slovakia, November 19-20, 2009. Accepted to appear.

[26] BitLaw, "17 USC 512, Limitations on liability relating to material online", 2005, [Online; accessed 12-June-2009], [Online], Available: http://www.bitlaw.com/source/17usc/512.html

[27] Wikipedia, "Field v. Google", 2009, [Online; accessed 12-June-2009], [Online], Available: http://en.wikipedia.org/wiki/Field_v._Google

[28] Europa.eu, "DIRECTIVE 2004/48/EC OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 29 April 2004 on the enforcement of intellectual property rights", 2004, [Online; accessed 12-June-2009], [Online], Available: http://eur-lex.europa.eu/pri/en/oj/dat/2004/l_195/l_19520040602en00160025.pdf

[29] http://www.squid-cache.org/

[30] Feily, M.; Shahrestani, A.; Ramadass, S.; A Survey of Botnet and Botnet Detection. In Proc of Emerging Security Information, Systems and Technologies, 2009. SECURWARE '09. Third International Conference on, 18-23 June 2009 Page(s):268 - 273, 2009.

[31] Sofat, S.; Bansal, D.; Gupta, M.: BOTNET - A Network of Compromised Systems. In Proc. of the Second National Conference on Challenges and Opportunities in Information Technology, 2009.

[32] Stone-Gross, B., Cova, M., Cavallaro, L., Gilbert, B., Szydlowski, M., Kemmerer, R., Kruegel, C., and Vigna, G.: Your botnet is my botnet: analysis of a botnet takeover. In Proc. of the 16th ACM Conference on Computer and Communications Security (Chicago, Illinois, USA, November 09 - 13, 2009). CCS '09. ACM, New York, NY, 635-647. 2009.

[33] http://stopbadware.org/

[34] Kaspersky Lab., "Kaspersky Lab utilizes NVIDIA technologies to enhance protection", 2009, [Online; accessed 27-January-2009], [Online], Available: http://www.kaspersky.com/news?id=207575979

[35] Chang, S. and Daniels, T. E.: P2P botnet detection using behavior clustering & statistical tests. In Proc. of the 2nd ACM Workshop on Security and Artificial intelligence (Chicago, Illinois, USA, November 09 - 09, 2009). AISec '09. ACM, New York, NY, 23-30. 2009.

[36] Ramachandran, A. and Feamster, N.: Understanding the network-level behavior of spammers. SIGCOMM Comput. Commun. Rev. 36, 4 (Aug. 2006), 291-302. 2006.

[37] http://www.blade-defender.org/

[38] http://www.bothunter.net/

Appendices

Appendix A CD medium – containing the Diploma thesis in its electronic form and P2P Proxy/Cache static binaries
Appendix B P2P Proxy/Cache Administration Manual
Appendix C Published papers