MEASUREMENT AND ANALYSIS OF BITTORRENT

A Thesis

by

VIDESH SADAFAL

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2008

Major Subject: Computer Science

MEASUREMENT AND ANALYSIS OF BITTORRENT

A Thesis

by

VIDESH SADAFAL

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Dmitri Loguinov
Committee Members,	Jennifer Welch
	Pierce Cantrell
Head of Department,	Valerie E. Taylor

August 2008

Major Subject: Computer Science

ABSTRACT

Measurement and Analysis of BitTorrent. (August 2008)

Videsh Sadafal, B.En. National Institute of Technology Karnataka, India

Chair of Advisory Committee: Dr. Dmitri Loguinov

BitTorrent is assumed and predicted to be the world's largest Peer to Peer (P2P) network. Previous studies of the protocol mainly focus on its file sharing algorithm, and many relevant aspects of the protocol remain untouched. In the thesis, we conduct a number of experiments to explore those untouched aspects. We implement a BitTorrent crawler to collect data from trackers and peers, and statistically analyze it to understand the characteristics and behaviors of the BitTorrent protocol better. We find that the expected lifetime of a peer in the BitTorrent is 56.6 minutes and the activity is diurnal. Peers show strong preference towards a limited number of torrents, and 10% of torrents are responsible for 67% of traffic. The US contributes maximum number of peers to the BitTorrent and μTorrent emerges as the favorite BitTorrent client. We measure the strength of Distributed Denial of Service (DDoS) attack using BitTorrent network and conclude that it is transient and weak. Finally we address and discuss the content locatability problem in BitTorrent and propose two solutions.

To my parents

ACKNOWLEDGMENTS

I am sincerely grateful to Dr. Dmitri Loguinov for agreeing to guide my Master's thesis and for allowing me to do research with him. This work would not have been possible without his constant motivation and guidance. Every bit of interaction with him has been a learning experience in some way or the other. His attitude and passion towards whatever he likes to do has always surprised and inspired me. I truly acknowledge Dr. Jennifer Welch and Dr. Pierce Cantrell for being the members of my advising committee. I also thank all the faculty members with whom I had a chance to interact and get educated.

I appreciate the help and company of all the members of the Internet Research Lab and friends in College Station. I am especially thankful to Xiaoming Wang for supporting and guiding me whenever I needed his help.

Last, but not least, I am indebted to my parents and family members for all their support and encouragement so far.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

With the increase in the connection speed of the Internet, online activities of users have also increased. One of the activities which has become prevalent in the Internet nowadays is files sharing. The Internet is the source of significant digital content. This digital content is spread worldwide, hosted on servers, personal computers and data centers. A basic and simple model for distributing files in the Internet is the client server model, where the content is hosted on servers, and downloaded by the clients. There are many known deficiencies of this model. The server can serve a limited number of clients depending on its capacity, the download rate is very slow, and the server is the central bottleneck for any failure. Multiple models have been proposed and developed to counter these problems with different pros and cons. Peer to Peer (P2P) sharing of the contents is one such solution that is distributed in nature and eliminates the need of a central storage system.

P2P networks have been receiving increasing demand from users and are now accepted as a standard way of distributing information because its architecture enables scalability, efficiency and performance as key concepts. The P2P model is a distributed model, and provides fault tolerance and robustness. The P2P applications use individual's computing power and resources, instead of powerful centralized servers and thus provide high content availability among peers. In P2P every node acts as a server and a client. If a client has a piece of the file, it acts as a server

The journal model is *IEEE/ACM Transactions on Networking*.

for other clients. It exploits the connectivity and bandwidth among different participants, in which a peer simultaneously downloads different pieces of the same file from many participants, which increases the downloading speed.

There are many versions of P2P systems, some of them are blend of the client-server and distributed models and some are completely distributed. For example, Napster, BitTorrent and OpenNAP fall in the first category, while Gnutella and Freenet in the second. Keeping aside the illegal sharing of the copyrighted contents and porn materials, P2P is a great breakthrough for sharing large files. Many Internet applications are evolving around P2P technologies, and it has become a hot field to research. Its use is not only limited to the sharing of the files, but it is also used for distribution of live video and audio streams, email, and anonymous and encrypted contents. Now the major portion of the Internet traffic is generated by P2P applications. A study of internet traffic conducted by *Cachelogic* reveals that in January 2006 P2P traffic accounted for 71% of all internet traffic, and 30% of that is caused by BitTorrent. BitTorrent is assumed and predicted to be the largest file sharing network in the world accounting for the large portion of internet traffic.

BitTorrent is a P2P protocol which provides effective solution for sharing and distributing large files over the Internet. The protocol was designed by Bram Cohen in April 2001, and it has now millions of users worldwide. Bittorrent has become prevalent nowadays. The reason is its simple architecture and robust file sharing algorithm that penalizes the selfish peers who only want to download without or contributing very less to others, and provides fast download to honest peers.

A Objective

The thesis aims to advance research in Bittorrent and derive meaningful conclusions by exploring the various aspects of the protocol. We collect and analyze real life data statistically to understand the user and protocol behavior better. We also strive to find the deficiencies or undesired behaviors in Bittorrent and propose modifications to make the Bittorrent more powerful. Most of the previous studies [1], [2], [3] mainly focus on Bittorrent file sharing algorithm to determine its efficiency, effectiveness and fairness. But there are many other aspects of the protocol which are still untouched and carry a great importance to characterize the protocol. We analyze seeds, non-seeds, population and popularity distribution for torrents, the residual lifetimes of the peers, their location distribution, usage distribution of the multiple Bittorrent clients and peers' activity pattern in the BitTorrent network. We also mount Distributed Denial of Service (DDoS) attack [4] exploiting BitTorrent platform and measure its strength and effectiveness.

Currently torrent files are hosted on the servers and peers are tracked by the trackers. There are a large number of Bittorrent trackers spread across the world and each has its own community of peers independent of each other. Nothing has been done to collaborate among these trackers, so that they can provide more effective solution of locating a file in the BitTorrent network. We propose some changes in the existing Bittorrent protocol with focus on the content locatability that eliminate the need of the servers to host the torrent files and integrate the trackers spread worldwide. Our integration makes the search more effective and helps users to locate the files in any Bittorrent network. A user need not to go to individual servers to find a file and the integrated platform enables the user to submit a query through any tracker and search the whole network.

B Contribution

We collect the statistical data for Bittorrent and shed light on some of the characteristics of the protocol. Our contribution can be summarized in following points:

- We design and implement a BitTorrent crawler that crawls a list of trackers and collects hashes for all the torrents managed by each tracker, and then for each hash obtains a list of peers contributing in the swarm.
- We analyze the seed and non-seed distribution among torrents, which reveals some interesting facts. The 48% of torrents have no seeds and 27% are without any downloader. Just 10% of torrents are responsible for major population of seeds and non-seeds in the network.
- Population distribution among torrents confirms that contribution of 90% of torrents to the peer population in the network is just 33% and remaining 10% torrents are responsible for 67% of the population
- We study the popularity of the torrents based on two metrics, number of times a torrent is downloaded and number of peers participating in the swarm. We see strong correlation between these two metrics.
- Geographic distribution of the peers at the country level reveals that maximum population of the peers in BitTorrent network originates from US.
- On the BitTorrent client side, users show strong preference towards μTorrent. The other popular clients are BitComet, Azureus and Bram's BitTorrent.
- We use a simple renewal-process based Residual Lifetime Method proposed in [5] to find the estimated lifetime of peers from the observed residuals. Our experiment shows that the expected lifetime of a peer is 56.6 minutes.

- We exploit a feature in BitTorrent protocol to mount the DDoS attack targeting a host in the Internet. The feature allows a peers to specify its address explicitly when it communicates through a proxy or when it is on the same local side of a NAT gateway as Tracker. We find that DDoS attack is not powerful in BitTorrent and lasts for a short duration of time.

- We record the peers activity in the network and conclude that their behavior is diurnal with users most active during noon and least active during midnight.

- Later we discuss on the problem of torrents locatability and propose two solutions, unstructured flood based approach and DHT based approach.

C Thesis Organization

We take a quick tour of BitTorrent protocol in Chapter II. Chapter III introduces us with the experimental setup and environment. We see the detailed study and analysis of various statistical measurements in Chapter IV to get an insight into BitTorrent protocol. Chapter V mentions implementation hurdles and high level organizations of various program architectures. We discuss the torrent locatability problem and propose some solutions in Chapter VI. Finally we conclude our thesis in Chapter VII.

CHAPTER II

BITTORRENT: AN OVERVIEW

In this chapter, we give the brief overview of the BitTorrent protocol [6], that is helpful for the readers who are new to the BitTorrent and provides them the basic foundation. We also explain some of features in detail, that we extensively used in our implementation. The chapter is also important to understand the following chapters comprehensively, since we make use of many terms defined here to describe our implementations and measurements.

Downloading large files from a server is often slow because there is a single source of the file. The Bittorrent protocol is a great breakthrough in the process of large size file sharing and distribution. We start our explanation of Bittorrent by first defining the main entities of the protocol as follows:

- Peer: It is responsible for sharing, downloading and uploading files in the network. When a peer has the complete file, it becomes a *Seed*. A seed uploads a file to others and does not download. A peer, who both downloads and uploads the file pieces is called a *Non-seed*. A peer who only downloads and does not upload the file to other peers is called a *Leecher*.

- Torrent: It is a file that contains the metadata information for a file to be shared. Bittorrent divides the shared file into multiple small size pieces and generates a torrent file, that contains an "announce" section that specifies the URL of the tracker who is going to assist the downloading of the torrent and an "info" section that contains the name of the file, its length, the piece length used, hash identifier for the shared file and a SHA-1 hash code for each piece, all of which is used by the clients to verify the integrity of the data they download.

- Tracker: It is a central unit which manages the swarms. A *swarm* is a group of peers collaborating among themselves to download a torrent. Tracker keeps a list of participating peers for each swarm with their downloading status (started|stopped|completed).

When a user wants to share a file, it generates the torrent file and publishes it on some web server or elsewhere, and registers with a tracker. Whenever a user wants to download a file from the BitTorrent network, it first obtains the torrent file through some web server and extracts the tracker's url from the torrent file. It then contacts the tracker to gather a list of participating peers in the swarm and starts downloading different pieces of the file in rarest first fashion from those peers. Simultaneously downloading multiple pieces from different peers not only makes the download fast but also provides robustness against failure. Such a group of peers connected to each other to share a torrent forms a swarm. While downloading different pieces, a peer also uploads the downloaded pieces to other peers, thus it contributes to the swarm.

A Encoding

Now we describe encoding scheme and some message formats in Bittorrent that we use in our implementation. Bencoding is an encoding scheme used in BitTorrent to specify and organize data in concise format. The following types are supported in Bittorrent:

- Byte string: it is encoded as follows: *<string length>:<string data>*
  Example: *6:videsh*

- Integer: it is delimited by *'i'* and *'e'* at the beginning and ending of the integer respectively. The format is *i<integer>e*

Example: *i256e*

- List: it is delimited by *'l'* and *'e'* at the beginning and ending respectively. It contains any bencoded type like integers, string, dictionary or another list. The format is *l<bencoded type>e*

  Example: *li34e6:videshe*

- Dictionary: it contains key-value pairs. Key must be bencoded string and value may be any bencoded type including integer, string, list or another dictionary. It is delimited by *'d'* and *'e'* at the beginning and ending. The format is *d<bencoded string><bencoded type>e*

  Example: *d4:name6:videsh6:coursei689ee*

B Message Format

The tracker is a HTTP/HTTPS service which responds to a HTTP GET request. The Bittorrent protocol specifies two ways by which a peer can communicate with a tracker. A peer needs tracker's URL appended with either "announce" or "scrape" to talk to the tracker. When the URL contains "announce", we call it *announce URL* and when it contains "scrape", we call it *scrape URL*. Both of these URLs have specific meanings, which we will describe shortly. Generally trackers do not publish their scrape URLs, but only announce URLs. A scrape URL of a tracker can be obtained by replacing "announce" with "scrape".

1. Scrape: it is used by the peer to find the status (number of seeds, peers and name of the shared file) for a particular torrent or for all torrents. For example: *http://example.com/announce -> http://example.com/scrape*

http://example.com/x/announce -> http://example.com/x/scrape

http://example.com/announce.php -> http://example.com/scrape.php

To collect the status of a particular torrent, the scrape URL is suffixed by the escape sequence of torrent's hash identifier. For example:

http://example.com/scrape.php?info_hash=aa...aa&info_hash=bb...bb

Many trackers do not support scrape convention but if a tracker does, then it responds to this HTTP GET request in a "text/plain" format or in gzip format consisting of the bencoded dictionaries. If no info_hash identifier is specified in the message, then the tracker returns information for all the hashes it maintains. The dictionary in a response contains following entries:

- files: this is a dictionary containing key/value pair for each torrent.
  - Key: A 20 byte info hash.
  - Value: It is the following dictionary
    - complete: number of seeders.
    - downloaded: number of times the tracker registered download complete event.
    - incomplete: number of downloading peers.
    - name: torrent's name as specified in info section of a .torrent file.

In the following example a torrent has 5 seeds, 10 downloading peers and it is completely downloaded 50 times. "...." represents a 20 byte info_hash.

d5:filesd20:....d8:completei5e10:downloadedi50e10:incompletei10eeee

2. announce: it is used to convey the status of a peer to the tracker and to obtain the address list of the participating peers about particular torrent. Thus the tracker keeps an updated statistics related to the torrent. The base announce

URL is of the form:

http://example.com/announce

http://example.com/x/announce

http://example.com/announce.php

The status parameters are added to the announce URL using standard CGI methods. All binary data in the URL must be properly escaped. The following parameters are used in the client request:

- info_hash: 20 byte SHA1 hash value of the torrent.
- peer_id: 20 byte unique client id.
- port: Port number the client is listening on.
- uploaded: Total number of bytes uploaded.
- downloaded: Total number of bytes downloaded.
- left: The number of bytes client still has to download.
- compact: It indicates if the client accepts compact response. In compact response each peer is represented by 6 bytes, first 4 bytes represent ip and next 2 port of the peer, all in network byte order.
- no_peer_id: It indicates if the peer wants the peer IDs also for other peers. If compact is enabled, it is ignored.
- event: It can be following one of the three or empty. Empty indicates that status will be sent at regular intervals.
  - started: The first request to the tracker.
  - stopped: When the client shuts down gracefully.
  - completed: When the download completes.

- ip: ip address of the client machine.
- numwant: Number of peers a client would like to receive from tracker. By default it is 50.
- key: It is an optional field, which is used by a client to prove its identity in case its ip changes.
- trackerid: (optional). If the previous announce contained a trackerid it should be included here.

Tracker response of the request consist of following fields in bencoded format:

- failure reason: It states the reason in case of failure in the human readable format. No other field is present if it is.
- warning message: Similar to the failure reason but included other fields also.
- interval: Interval between successive announce requests.
- min interval: (optional). If present the client must not re-announce more frequent than this.
- tracker id: It should be sent back by client in its next announcement. If absent and if the client used tracker id in its previous announcements the same id should be used in subsequent announcements.
- complete: Number of seeds.
- incomplete: number of downloading peers.
- peers: It can be either in dictionary mode or compact mode. In compact mode each peer is represented as described previously in the compact field. In dictionary mode following keys are present:

- peer id: 20 byte unique id of the peer.
- ip: Peer's IP address.
- port: Port number the peer is listening on.

Generally trackers randomly select peers to be included in the response, but it is completely onto them to use any intelligent algorithm.

CHAPTER III

EXPERIMENTAL SETUP

Our experiments require collecting statistics about trackers and peers in Bittorrent network. Peers are managed by the trackers, and trackers are distributed all over the world, so our first task is to collect trackers' announce URLs. We search through Internet and collect the trackers addresses. In our efforts, we find a good link [7] that contains a list of around 800 trackers and we include those also in our list of trackers. But unfortunately all of the listed trackers are not accessible. Many of them are either unresponsive or are private trackers who require an account and a passkey to access them, so we are able to access and collect the statistics only from the public trackers. There are around 35 public trackers, who reply to our requests. The next step in out experiment requires a large collection of peers and torrents information. We query trackers to find the hash identifiers of all the torrents they maintain, and then using those hashes, we obtain a list of peers belonging to the swarm of each torrent. To accomplice the mentioned tasks, we design and implement a BitTorrent Crawler and then according to the requirement of each experiment tweak it a little to perform the desired function.

A BitTorrent Crawler

Bittorrent crawler is a program, when given a list of the trackers crawls each tracker and collects hash identifier and swarm information (seeds, non-seeds and number of times the torrent is downloaded) for each torrent managed by a tracker, and then for each hash collects the peers' addresses (ip, port, id) participating in a swarm. The

```
GET /scrape.php HTTP/1.1\r\nConnection: close\r\n
Accept-Encoding: gzip\r\User-Agent: IRL-P2P-crawler\r\n
Host: torrent.unix-ag.uni-kl.de:6969\r\n\r\n
```

Fig. 1. Formatted string for scrape request.

crawler consists of two logical parts:

1 Hash Collection

We use the scrape convention described in the previous chapter to collect all the hashes managed by the trackers. Each hash corresponds to a unique torrent. We take the announce URL of a tracker, replace *announce* keyword with *scrape* and send the HTTP GET request to the tracker. For example, a typical scrape request for announce URL *http://torrent.unix-ag.uni-kl.de:6969/announce.php* looks as given in Figure 1, where we accept gzip encoding and advertise the user agent as IRL-P2P-crawler. Abbreviation IRL represents *Internet Research Laboratory* in Texas A&M university.

2 Peer Collection

We use announce method to obtain a list of peers in the swarm for each torrent. We make use of the hash identifiers collected in the previous step. When we send an announce request with *event=started* to a tracker, we get a list of peers but at the same time the tracker also registers our address and distributes it to the other peers in the swarm. The side effect of this appears in the form of inbound connections from other peers who try to contact us in search of the pieces they need. To avoid this undesirable situation, we send an announce request with *event=stopped* immediately after getting

```
GET /announce.php?info_hash=....&peer_id=....&port=6882&uploaded=0&
downloaded=0&left=100000000&event=started&compact=1&numwant=10000&
no_peer_id=1 HTTP/1.1\r\nConnection: keep-alive\r\n
Accept-Encoding: gzip\r\nUser-Agent: IRL-P2P-crawler\r\n
Host:torrent.unix-ag.uni-kl.de:6969\r\n\r\n
```

Fig. 2. Formatted string for started announce request.

a response from the tracker for the previous announce request. On the receipt, the tracker deregisters us from the swarm, that prevents spread of our address to other peers. For example, the formatted string of the started announce HTTP message for announce URL *http://torrent.unix-ag.uni-kl.de:6969/announce.php* appears as shown in Figure 2.

In Figure 2, "...." represents a 20 byte escaped hex sequence of the formatted string for *info_hash* and *peer_id*. Fields *uploaded* and *downloaded* are initialized as 0. An important parameter of the request is *left* and it must be initialized with some large value, otherwise setting it to 0 makes the tracker consider us as a seed, which may cause unnecessary security concerns and tracker authority may assume us sharing the illegal copyrighted contents. To make the peer address representation in the response small as explained in the previous chapter, we set *compact=1*. Field *numwant* is set to some large value so that we can collect as many peers from a swarm as possible, but a tracker ignores it if it is more than 50. When we don't need peer id, we set *no_peer_id=1*. Since we need to send the stopped announce request immediately after response, we keep the current TCP connection open with *Connection:keep-alive* and send the stopped message in the same connection. If a tracker closes the TCP connection after responding to started announce request, we

```
GET /announce.php?info_hash=....&peer_id=....&port=6882&uploaded=0&
downloaded=0&left=100000000&event=stopped&compact=1&numwant=10000&
no_peer_id=1 HTTP/1.1\r\nConnection: close\r\n
Accept-Encoding: gzip\r\nUser-Agent: IRL-P2P-crawler\r\n
Host:torrent.unix-ag.uni-kl.de:6969\r\n\r\n
```

Fig. 3. Formatted string for stopped announce request.

open a new TCP connection to send stopped announce request. The formatted string of the stopped announce HTTP message for the same announce URL appears as shown in Figure 3.

Initially we implement the hash and peer collection together in the crawler. During our experiments we realize that a fresh list of peers are required each time an experiment is run, but hashes remain same across experiments, so we separate the hash collection part from peer collection. The hash collection is required only once at the beginning of all experiments and the subsequent runs use the same hashes. Also it speeds up our experiments, since hash collection requires decoding a large bencoded file for each tracker that takes long time and we need not to wait for decoding of the bencoded files. We collect around 1 million hashes for 35 trackers. Since experiments require fresh list of peers each time they are run; the number of hashes we query and peers we collect, vary across experiments depending on the availability of the trackers and network condition at the time of the run.

CHAPTER IV

MEASUREMENTS

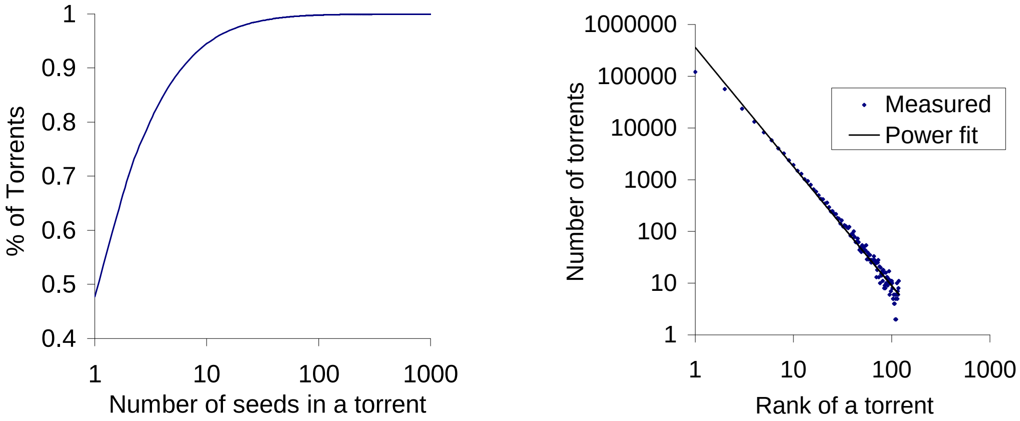
In this chapter, we conduct a series of experiments, collect the data and analyze it statistically. We obtain some interesting results about BitTorrent, which we will see shortly. These results help us to understand the behavior of the protocol and performance in real life situation.

A Seed Distribution

We start our analysis with the seeds distribution among the torrents. A seed is an important part of the swarm and makes sure that at least one complete copy of the shared file exists in the network.

We collect number of seeds available for all the torrents in a tracker using our BitTorrent Crawler. We collect the number of seeds, non-seeds and times a torrent is downloaded information for around 254,000 torrents. The same data we will be using to analyze non-seed, population and popularity distributions in the coming sections. Figure 4(a) shows the CDF distribution of seeds on torrent population in BitTorrent network, that is exponential in nature. In Figure 4(b), we draw the frequency of torrents verses their ranks on log-log scale. A torrent with the n seeds is provided a rank of $(n + 1)$, thus a torrent with no seed is given rank 1, with 1 seed rank 2 and so on. We draw this Figure till rank 120, after that power fit is not possible because there are no torrents for some of the ranks. Analyzing this data reveals following results:

1. Since there is no seed for a large fraction of the population, the fraction can not



(a) CDF of seeds on torrents.

(b) Rank-wise distribution of torrents.

Fig. 4. Seed distribution in BitTorrent network.

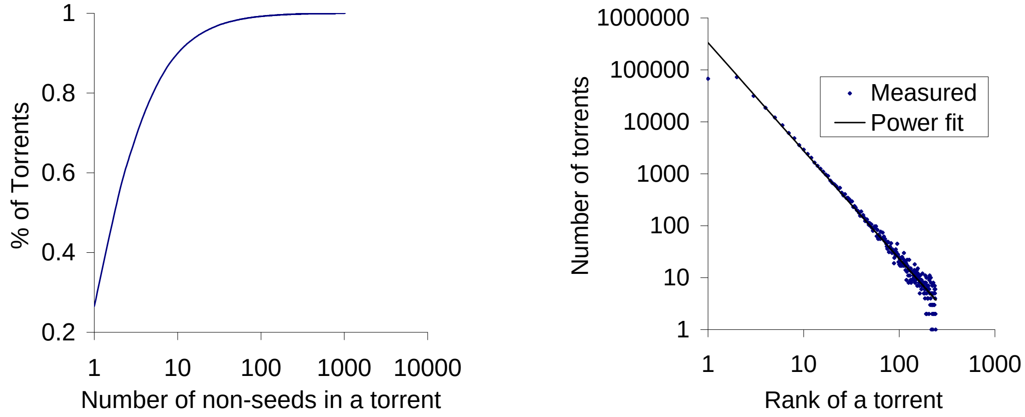
guarantee that there exists a complete copy of the shared item and may not be useful. This fraction is 48% of the total population.

2. Even 90% of the torrent population has less than 6 seeds, so a large population of seeds comes from just 10% of the torrents.

3. Close resemblance of the torrents distribution to the power fit on log-log scale in Figure 4(b) proves that the seed distribution closely follows Zipf distribution.

B Non-Seed Distribution

In this experiment, we do similar analysis as in the previous section but this time with the non-seeds. The experiment is useful to know how the non-seeds are distributed among torrents in BitTorrent network. CDF distribution of non-seeds on torrents in Figure 5(a) exhibits the exponential distribution. In Figure 5(b), we provide ranks to all the torrents based on the number of non-seeds they have and draw the frequency distribution for top 240 ranks. A torrent with n non-seeds is given a rank of $n + 1$.



(a) CDF of non-seeds on torrents.

(b) Rank-wise distribution of torrents.

Fig. 5. Non-seed distribution in BitTorrent network.

We find that

1. There is no downloader for 27% of torrents. Probably the fraction is an indicator of useless material in BitTorrent, either because there is no seed and download can not be completed, or the material is a mal-content.

2. Torrents with less than 6 downloaders constitute 83% of torrents population. It indicates that most of the downloaders are interested in just 17% of the torrents.

3. The rank-wise distribution of torrents in Figure 5(b) closely follows power fit, that implies the Zipf distribution.

C Population Distribution

Now we determine how the population is distributed over the BitTorrent network. In this distribution, we consider both seeds and non-seeds in a torrent to analyze the population distribution among torrents. There are two important aspects of this analysis, first to find how the peers' density is distributed over torrent population, and second how much is the contribution of those torrents to the total peer population in the network. Figure 6(a) displays the CDF distribution of peers on torrents. In Figure 6(b), we analyze the rank-wise frequency distribution of torrents with Zipf distribution, providing a rank of $n + 1$ to a torrent with n peers. The collected statistics shows that

1. There is neither a seed nor a non-seed for 7% of the torrents, so the fraction is a garbage.

2. The distribution of population is very sparse for a large fraction of torrents. This uneven distribution is visible from the fact that around 80% of the torrents have less than 8 peers and small fraction of the torrents is highly dense.

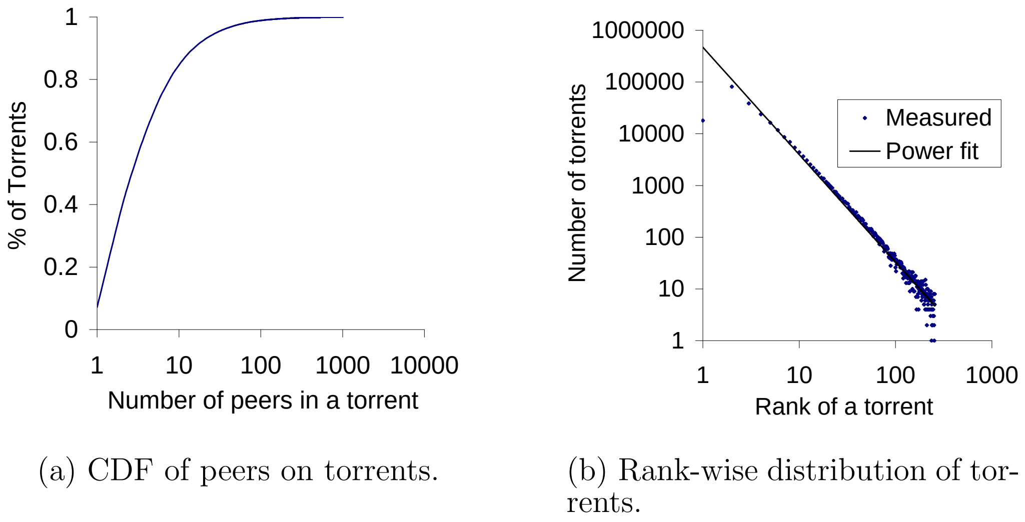
3. Only 10% of the torrents have more than 15 peers, which indicates that a large population of peers comes just from 10% of the torrents.

4. Figure 6(b) exhibits that rank wise frequency distribution of torrents closely follows the power fit, and that implies the Zipf distribution.

Figure 7 shows the relationship between the torrent population and their contribution to the peer population. Around 7% of the torrents do not contribute anything. The Figure concludes that contribution of the 90% of torrents to the peer population in the network is just 33% and remaining 10% torrents are responsible for 67% of the population. If we assume the download rate to be constant among all the peers, than we can conclude that 10% of the torrents cause 67% of the Bittorrent traffic in the Internet.

D Popularity Distribution

We use two metrics to measure popularity of the torrents



(a) CDF of peers on torrents.

(b) Rank-wise distribution of torrents.

Fig. 6. Peer distribution in BitTorrent network.

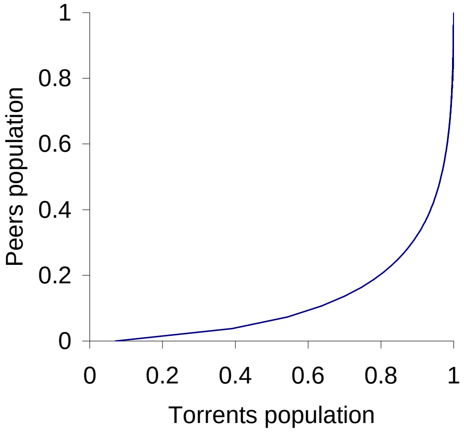
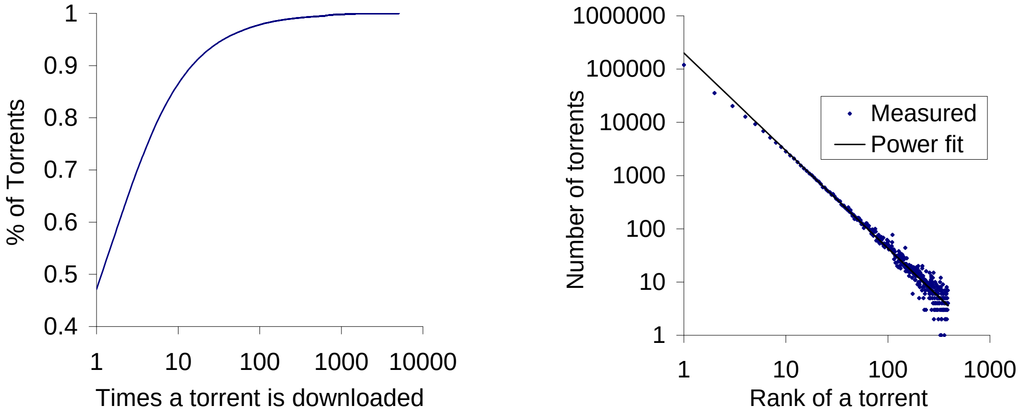


Fig. 7. Torrents contribution to the peer population.

1. Total number of peers in a torrent: This is the same metric used to analyze population distribtution. The number of peers in a torrent swarm is also an indicator of the torrent's popularity. The collected data shows that users are not at all interested in 7% torrents. Around 80% of the torrents have less than 8 peers. Torrents with more than 15 peers constitute only 10% of the population, so the users are most interested in mere 10% torrents.

2. Times a torrent is downloaded: The above metric does not take history of a torrent into account, but the times a torrent is downloaded is a good indicator of the torrent popularity till now including its past. Figure 8(a) shows the CDF distribution of torrents based on the number of times it is downloaded. The Figure reveals that 47% of torrents are not even downloaded once. Around 80% of the torrents are downloaded less than 6 times. Only 10% of torrents are downloaded more than 15 times. Thus very small population of torrents is really popular among downloaders. Figure 8(b) shows the frequency distribution of the torrents on the increasing order of their ranks. A torrent that is downloaded n times is given a rank of $n + 1$. The distribution closely follows the power fit and confirms the Zipf distribution.

Both the metrics follow Zipf distribution and exhibit linear correlation between them that indicates that either peer distribution or times-downloaded distribution can be used to obtain the popularity distribution for a tracker. The obtained results also conclude that even if the BitTorrent network seems so large sharing huge number of files, but peers are interested in very small fraction of the files contributing major portion of the traffic.



(a) CDF of download frequency on torrents.

(b) Rank-wise distribution of torrents.

Fig. 8. Popularity distribution in BitTorrent network based on the number of times a torrent is downloaded.

E Client Popularity Distribution

There are more than 75 BitTorrent clients, some are well known but many not even heard of. We take the study to find the users preference for different BitTorrent clients. Each peer in BitTorrent network is recognized by its ID, which contains the client ID, client version and a random number. We extract client ID from the peer ID to find which client a peer is using. There are two ways, we can collect client ID's of the peers; 1) directly from a tracker setting *compact=0* and *no_peer_id=0*, so that a tracker returns a peer ID also with its address, 2) We use BitTorrent crawler to get the peers currently participating in the BitTorrent network, and then we contact every peer to find its client ID. Problem with the first approach is that many trackers do not follow the convention and even after asking for peer id, they return the response in concise format without peer id, so we stick to the second approach in our implementation. From the collected 220,000 client IDs, we draw the distribution for top 17 clients in use as shown in Figure 9 and find that 95% of

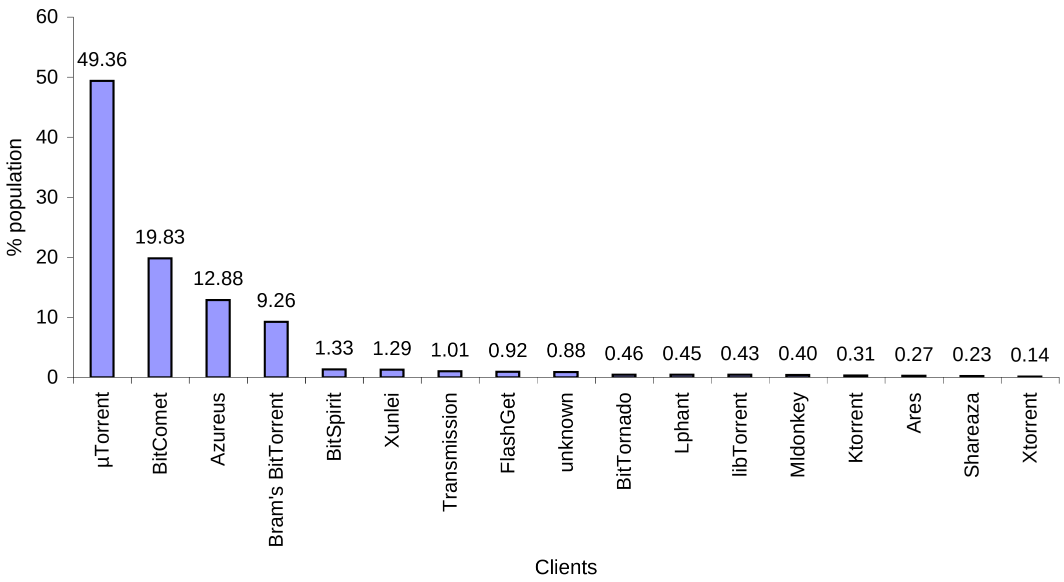


Fig. 9. Usage distribution of top 17 clients.

the users use only four BitTorrent clients, μTorrent, BitComet, Azureus and Bram's BitTorrent (it is the BitTorrent client devised by the inventor of BitTorrent "Bram Cohen"). Among them μTorrent emerges as a clear winner claiming 49.36% of the BitTorrent clients. BitComet follows next with 19.83% clients. Azureus claims 12.88% and Bram's BitTorrent 9.26% clients.

μTorrent is a freeware written in C++, that is designed to use minimal computer resources while offering all the major functionality including Mainline DHT. BitComet is also a C++ written windows client, while Azureus is written in Java. Bram's BitTorrent is the first BitTorrent client, that revolutionized file sharing.

F Geographic Distribution of Peers

Geographic disperse of the peers is useful to know the location preference of the users. To see the geographic spread of the peers, we first obtain unique IP addresses of the peers using BitTorrent crawler and then we identify their locations from their IP addresses.

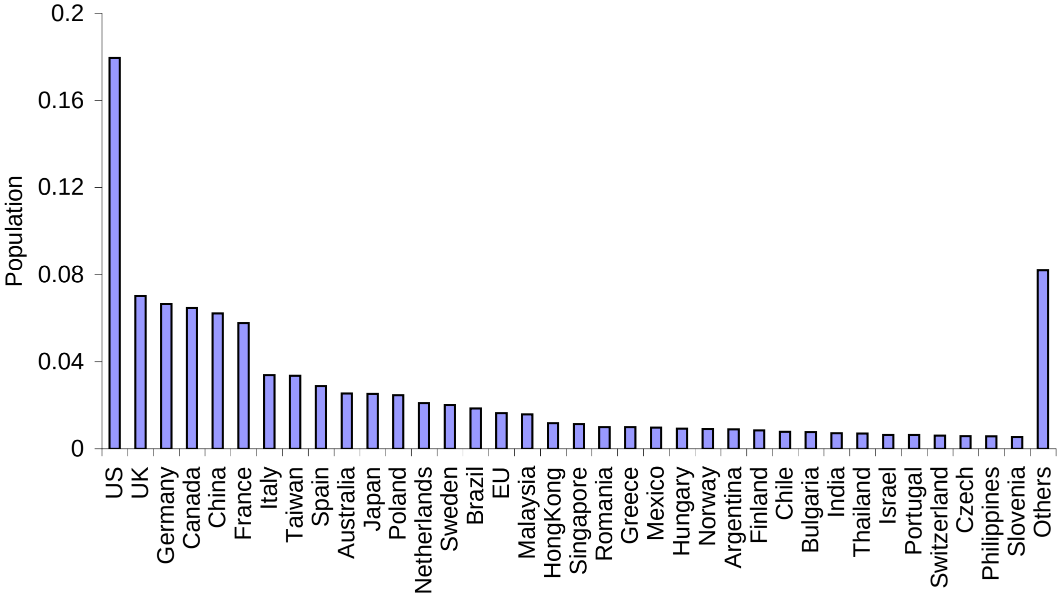


Fig. 10. Geographic distribution of peers.

To find the location of a peer, we use the WHOIS database. WHOIS is a TCP-based query/response protocol which is widely used for querying a WHOIS database in order to determine the owner of a domain name, an IP address, or an autonomous system number on the Internet. One WHOIS server stores the WHOIS information from all the registrars for the particular set of data. When we try to retrieve information related to a particular IP address, it searches the database to find which domain and ASN the IP address belongs to and returns its information. But WHOIS database is good to obtain location information of IP addresses, only when the granularity is high, i.e. country level. We use a WHOIS utility called NetCat to perform the mentioned task. NetCat takes a file containing IP addresses and retrieves the information from WHOIS server.

We collect close to 700,000 unique IP addresses and their country level distribution in Figure 10 shows that US has dominance over geographic preference of the peers with 18% peers population. Most of the European countries contribute almost equally with slightly higher proportion contributed by UK with 7%. In Asia, although China emerges as the main participant, but its share in the BitTorrent user population is around 6% less than half as compared to US.

G Expected Lifetime of a Peer

Expected lifetime of a peer in the P2P network is an important property to characterize the network. Peer lifetimes are important for understanding general user behaviors, their habits and application performance offered by the peers in the system. Many existing methods use *Create Based Method* (CBM) [8], which has many known deficiencies. It divides a given observation window into two halves and samples users every $\triangle$ time units until they die or the observation window ends. A small observation window or large $\triangle$ may lead to highly inaccurate lifetime distribution in CBM. A better accuracy can be achieved with a small $\triangle$, but it introduces significant overhead in the method. Thus the sampling method exhibits an inherent tradeoff between overhead and accuracy. Further the approach causes bias in the measurement because of two factors; inconsistent round-offs that is introduced by rounding up the lifetimes for some users and rounding down for others, and missed users who arrive and depart between $\triangle$ interval.

Papers [5], [9] and [10] create a novel analytical framework to understand and characterize bias in the network sampling. They show that any sampling method that directly attempts to measure user lifetime every $\triangle$ time units is biased as long as $\triangle > 0$ and the bias can not be removed regardless of the mathematical manipulation applied to the measured samples. To overcome these limitations of the direct sampling methods [5] proposes a simple renewal-process based *ResIDual-based Esti-*

mator (RIDE) to produce an unbiased version of the residual distribution $H(x)$. The method requires taking a snapshot of the entire network and tracing residual lifetime of each peer found in the snapshot until it dies or the observation window expires. The lifetime distribution $F(x)$ can be obtained from sampled residuals $H(x)$ using a simple mechanism based on renewal churn model of [9], [10] with a negligible amount of error.

1 RIDE

At time 0 RIDE takes the snapshot of the whole system and records all the alive users n found in the snapshot in a set S_0. It probes all the peers in S_0 for the subsequent interval j of $\triangle$ time units either until they die or the observation window T expires. RIDE defines two important properties: 1) no valid samples can be missed since only users who are alive at time $t = 0$ are valid measurements; 2) no samples can be inconsistently rounded off since all valid residual lifetimes start from the time of the first crawl.

The measured distribution can be used to obtain an unbiased estimator of the actual residual distribution:

$$P(R(t) \leq x_j) = \lim_{|S_0| \to \infty} \frac{N(x_j)}{|S_0|}, \tag{1}$$

where $N(x_j)$ denotes the number of samples in S_0 with measured residual lifetime smaller than or equal to time $x_j(x = j\triangle)$. At any time $t \gg 0$, the distribution of the remaining lifetime $R(t)$ of users present in the system is given by [10]:

$$H(x) = P(R(t) \leq x) = \frac{1}{\mu} \int_0^x (1 - F(u))\mathrm{d}u, \tag{2}$$

where $\mu = E[L]$ is the expected lifetime of a joining peer and system size n is sufficiently large. For residual lifetime sampling the following is an unbiased estimator of lifetime L:

$$E_R(x_j) = 1 - \frac{h(x_j)}{h(0)}, \tag{3}$$

where $h(x_j)$ is the Probability Density Function (PDF) of $R(t)$. Since $H(x)$ is obtained without bias, it is possible to numerically compute its derivative $h(x)$ using Taylor expansion with error bounded by $O(\triangle^k/k!)$, where $k = T/\triangle$ is the number of samples.

2 Weibull Distribution

The Weibull distribution is a continuous probability distribution and is often used in data analysis. It defines the CDF as follows:

$$P(X < x) = 1 - \exp^{-(x/\lambda)^k} \tag{4}$$

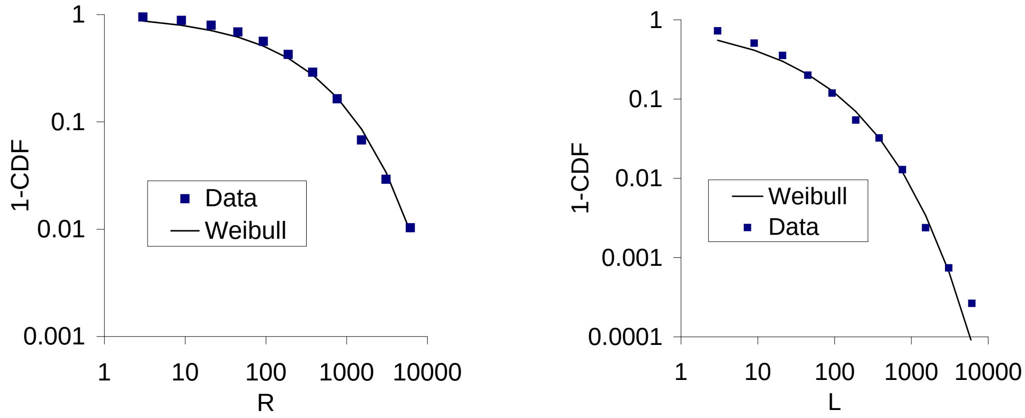
where $k > 0$ is the *shape parameter* and $\lambda > 0$ is the *scale parameter* of the distribution. The expected mean of the distribution is give as:

$$E(X) = \lambda\Gamma(1 + \frac{1}{k}) \tag{5}$$

where Γ represents the Gamma function.

3 Observation

In BitTorrent network, we first obtain all the hashes for the torrents from trackers, and then for each hash, we collect the peers. Since a tracker is a centralized entity, it does not entertain many connections from a peer and limits the number of connections a peer can establish at a time. It causes an elongated snapshot period, which may potentially harm and introduce bias in our measurement. To avoid this, we start probing each peer independently as soon as it is available. Further all the peers



(a) $k = 0.4612$ and $\lambda = 218.21$

(b) $k = 0.3618$ and $\lambda = 12.587$

Fig. 11. Complementary CDF ($S_0 = 3.75 \times 10^5, T = 7$ days, $\triangle = 3$ minutes).

collected from the trackers may not be alive or reachable, and since RIDE restricts the observation only for the alive peers from the snapshot, we include only those peers into set S_0 who reply to our first probe message.

We collect around one million unique peers from 250,000 hashes, but only 375,000 of them are reachable and constitute our initial set S_0. We observe the peers for one week ($T = 7$ days) and probe each peer every $\triangle = 3$ minutes to see if it is alive. Figure 11(a) shows complementary CDF of the residual lifetime distribution. It is clear from the Figure that the distribution closely follows the Weibull distribution with $k = 0.4612$ and $\lambda = 218.21$. We use two point derivative method [5] to obtain expected lifetime distribution of the peers using Equation 3 as shown in Figure 11(b). The resulting complementary CDF curve follows the Weibull distribution with $k = 0.3618$ and $\lambda = 12.587$, that gives expected lifetime of 56.6 minutes. Our results confirm the finding of [11] that the session length distribution is neither Poisson nor Pareto and is more accurately modeled by a Weibull distribution. For $\triangle = 3$ and $T = 7days$, the resulting error comes to 4.95×10^{-8789}, that can be considered 0 for all the practical

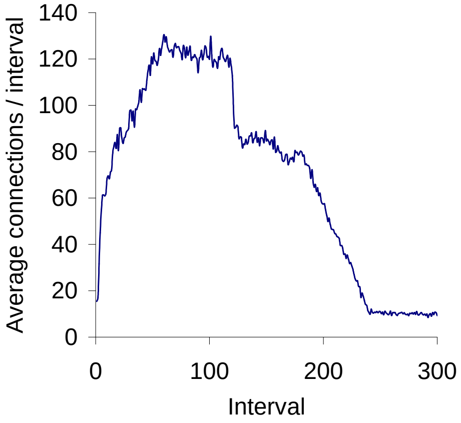


Fig. 12. Intensity of attack with time.

purposes.

H Distributed Denial of Service Attack

When a peer wants to download a torrent, it announces to the tracker of its presence. The tracker registers peer address and returns a random list of maximum 50 peers participating in the torrent swarm. The peer then contacts other peers in the swarm to download the torrent. Since its address is registered in the tracker, it is distributed to other peers as a result of their update announce request. As a result the peer starts receiving connections from the other peers.

The BitTorrent protocol allows a peer to specify its IP address and port number explicitly in the announce request to the tracker when the peer communicates through the proxy. This feature is also necessary when both the client and the tracker are on the same local side of a NAT gateway. The BitTorrent protocol does not enforce any security check to verify that the announced IP is same as that of the requester. We exploit this feature to attack a host in the Internet with the connection requests from the BitTorrent network.

In this experiment, we measure the strength of Distributed Denial of Service attack using the BitTorrent P2P network. We use two systems connected to the internet, one as an attacker and other as a target. We publish the target's address from the attacker for all the hashes collected during hash collection step of BitTorrent crawler and at the same time measure the number of connections being received at the target. We publish the target for 198,000 hashes in 45 minutes and monitor the number of incoming connections at target for 150 minutes. Figure 12 shows average number of connections per interval, where each interval is a window of 30 seconds. We measure the incoming connections for 150 minutes at the target. We observe maximum 164 inbound connections per second. Averaging it over every 30 seconds window gives maximum 130.59 average connections in an interval as shown in the Figure 12. For the observation period of 150 minutes, average connections per second comes to 70.36. We also observe that the attack diminishes with the time. The attack proves to be quite weak. The possible reasons are

1. Once a peer fail to connect to the target, it does not retry to connect.

2. Tracker removes a peer address from the list of available peers, if it does not receive the updated announce request from the peer within certain time from the previous announcement. The time is usually set 1.5 or 2 times of announce interval and the announce interval is generally 30 minutes. We see a decline in the attack intensity in the Figure 12 at around and after interval 120 (3600 seconds), that is the result of tracker deregistering our announced target. We see the sharp decline after 1 hour, that eases down later. The phenomena can be explained here. We have many small trackers and some large trackers and since we start advertising target to all the trackers together with the same intensity, small trackers finish fast and our attack continues with only some large trackers.

The result is high intensity of attack at the beginning that comes down with the time and obviously the same effect is observed when trackers start deregistering our target. We advertise the target address for all the hashes only once and then watch its effect with time without renewing the attack.

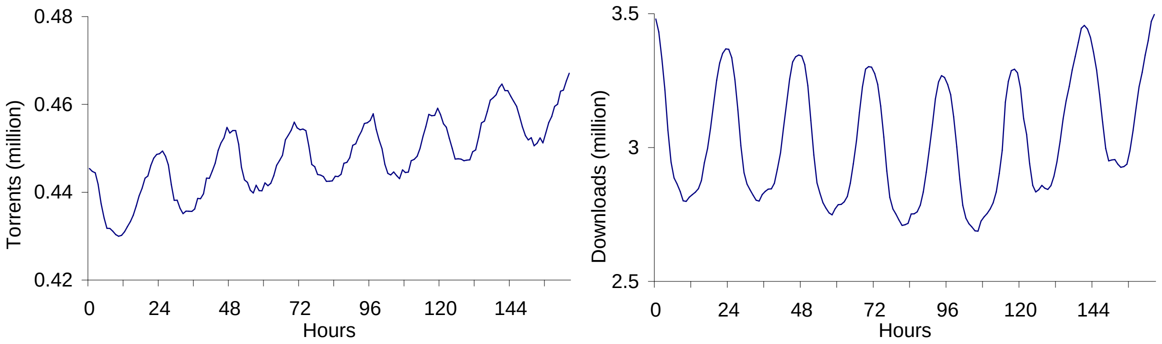
3. The strength of the attack is weaken by the fact that tracker is a central entity and does allow a limited number of connections from a peer. If we would have published all the hashes in 3 minutes rather than 45 minutes, then it would have caused much powerful attack, but again for a short period of time due to the above two reasons.

4. We have a small collection of 35 public trackers who respond to our request and sometimes not all of them are responsive at the same time. Collecting a huge number of trackers and then mounting attack will prove rather powerful.

I Activity in the Network

Peers do not remain online always, they join the network to download a file and then leave it. We take the study to find at what time users prefer in a day to download a file, and how is the arrival pattern of the users. To record the peer activity in the BitTorrent network, we select a tracker with a large user base and monitor it for 7 days recording user activity every hour in terms of total torrents and active downloads using scrape method. Statistical analysis of collected data shows that Activity in the BitTorrent network exhibits diurnal behavior and the users are most active around noon.

We identify a large tracker *a.tracker.thepiratebay.org* for this task and monitor it for one week. We start our experiment at 12:00 PM on 4/13/2008 and take snapshot of the network every hour. Figure 13(a) shows users upload behavior in terms of



(a) Users upload behavior.

(b) Users download behavior.

Fig. 13. Activity in the BitTorrent network. Experiment starts at 12:00 PM, Sunday, 4/13/2008.

number of torrents in the network. We observe that in a day users upload most torrents around noon and are least active during midnight. We also observe growth in tracker network, and total torrents at the end of our experiments (one week) are around 465,000 up 20 thousands from the start of experiments, so the tracker grows 4.5% during our experiment.

Figure 13(b) captures users download behavior. It plots number of active downloads every hour for 7 days. We observe similar diurnal behavior as in previous graph with users most active around noon and least active during midnight. We start our experiment on Sunday 12:00 PM, which registers relatively higher number of downloads compared to that of weekdays and it again goes high during next weekend. Probably users prefer downloading during weekends when they are free then weekdays.

CHAPTER V

IMPLEMENTATION

Initially we decide to use open source BitTorrent client *Arctic* to implement BitTorrent Crawler. Arctic is a simple implementation of BitTorrent client in C++ in windows platform with a simple GUI. We modify the source to use only the required portion that communicates with a tracker. But it proves very slow for our experiments. Arctic provides the thread based implementation where communication with each tracker is done separately in a thread. Each thread combines the hash collection and peer collection parts, which is not desired to perform variety of experiments for the same reason we explained in Chapter III. Since decoding is a very computation intensive operation and takes up the whole CPU, performing peer collection at the same time makes many requests to get timed out, that further worsens the experiment time. Issues faced during open source modifications lead us to write our own implementation from scratch. We write the code in C++ and make use of the open source zlib library for decompressing of data from the tracker. We separate out the hash and peer collection parts. We keep the hash collection part fairly simple, where we generate limited number of threads who contact the trackers, decode the hashes and store them for future use. Figure 14 shows the high level structure of hash collection program.

In the peer collection part, since we need to communicate with many trackers at the same time collecting peers for many hashes, we first experiment with simple thread based approach where we open one separate thread for each tracker and then each thread opens many connections with the assigned tracker to collect peers. Man-

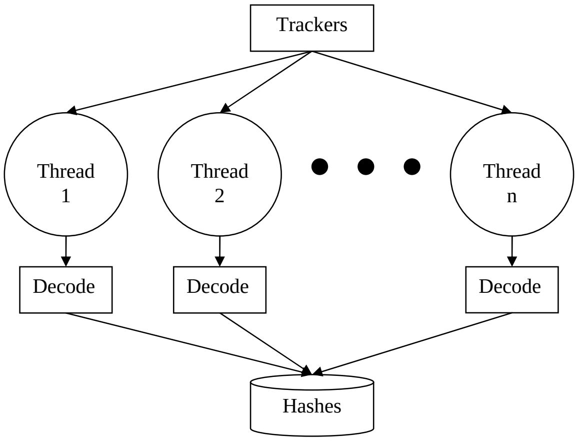


Fig. 14. High level organization of hash collection program.

aging many connections at a time can be accomplished with non-blocking sockets. The problem with this approach is that opening more than one thread per processor does not provide any efficiency improvement and non-blocking socket based implementation suffers from the serious drawback of polling.

The solution comes in the form of *Input-Output Completion Port* (IOCP), that provides an scalable solution to manage large number of connections in a windows machine, while still keeping the threads overhead at the low. IOCP is an API for performing multiple simultaneous asynchronous input/output operations. An IOCP port can be created and associated with a number of sockets or file handles. When I/O service is requested on the object, completion is indicated by a message queued to the I/O completion port. A process requesting I/O services is not notified of completion of the I/O services, but instead checks the I/O completion port's message queue to determine the status of its I/O requests.

Figure 15 shows the high level organization of peer collection program. We use two threads in our program. *Tracker thread* creates IOCP based sockets and generates asynchronous connections to the trackers to discover new peers. If connection to a

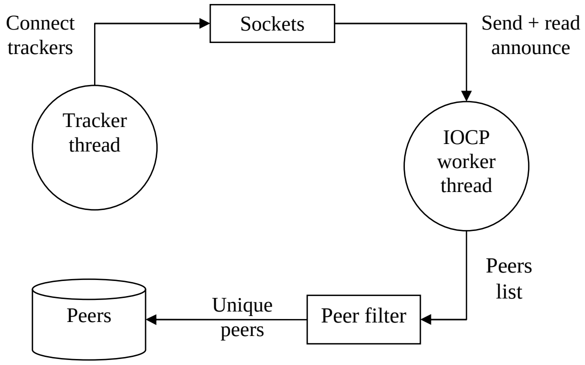


Fig. 15. High level organization of peer collection program.

tracker is successful, *IOCP worker thread* sends announce request for the torrent hash collected during hash collection phase to the tracker. On the receipt of the response from the tracker the worker thread filters already seen peers and puts the newly discovered peers in some file for latter processing. Although IOCP based implementation resolves issues with managing large number of connections efficiently, yet we can not open unlimited connections with a tracker. Tracker is a central entity and limits number of connections from a peer. To speed up the peer collection, we open 50000 connections simultaneously with a tracker, but it does no better than opening 200 connections at a time. Most of the connections get timed out and very few are successful, so we limit the number of connections with a tracker to 200.

The trickiest part of our experiments is to track every peer for its residual lifetime. As explained in the previous chapter, since our initial snapshot takes a long time, to mitigate its effect from the measurement, as soon as we discover a new peer we poke it to see if it is alive and if yes, we put it into our sample set, where they are tracked every $\triangle$ interval. Figure 16 explains the high level organization of the program for discovering new peers and tracking them for their residual lifetime, that we obtain by modifying our peer collection program. We introduce one more thread *peer thread*

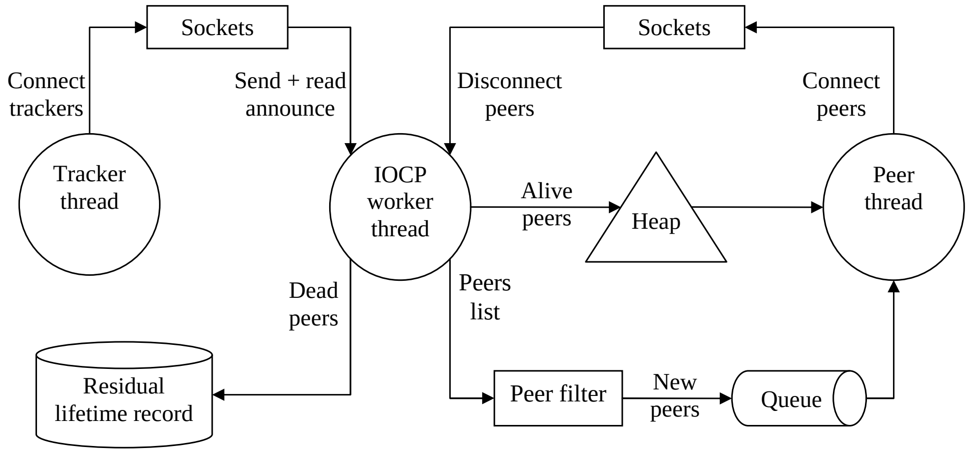


Fig. 16. High level organization of residual lifetime tracking program.

that tracks discovered peers for their residual lifetime. Here on the receipt of the response from the tracker the worker thread filters already seen peers and puts the newly discovered peers in the queue. Peer thread takes the new peers from the queue, creates IOCP based sockets and issues asynchronous connections to them. Connection indication is received by the worker thread and if successful then the peer is considered in our sample set and put on the heap, otherwise discarded. The peer thread tracks each peer on the heap every △ interval of its previous poke time until the peer dies or the observation time expires. When the connection to a peer in our sample set fails, the peer is considered dead and worker thread registers its residual lifetime.

CHAPTER VI

A BETTER SEARCH INFRASTRUCTURE FOR BITTORRENT NETWORKS

Bittorrent is one of the most successful Peer to Peer (P2P) protocols. Its success lies in its simplicity of organizing the peers sharing the same file together, so that they can find and help each other in downloading different pieces of the files simultaneously, and its algorithm which discourages free riders and rewards the uploaders. Although there is no doubt about the popularity of Bittorrent, but one thing which remains cumbersome and problematic in Bittorrent is how to locate the torrent of a particular file. A user can take advantage of Bittorrent only when it has torrent for the shared file, but where to obtain that torrent from? This is a big question and a big roadblock on the road to Bittorrent becoming almighty from the mighty.

The current implementation of Bittorrent requires you to get the torrent for the shared file from some external source before downloading can be initiated. Generally a user who wants to share some file in Bittorrent network generates the torrent for the shared file and uploads it to some web server. Whoever wants to download the file gets the torrent file from the web server and then starts downloading the shared file. But locating the server which contains the desired torrent is just a hit and trial approach and sometimes it can take long time; sometimes the users are so frustrated that they simply choose to give up. The web is huge and sometimes finding a particular shared file proves a gigantic task if not impossible.

In this chapter, we focus on the issue of torrent locatability and propose some solutions for the problem. We think that integration of search functionality of isolated trackers will resolve the issue and consider different approaches to achieve this. We explore the pros and cons of each solution, and compare their practicability, feasibility and effectiveness to solve the problem. We give here the brief description of the approaches we consider and will discuss each approach in detail in coming sections. We propose two solutions to search the torrent file in the Bittorrent network

1. Unstructured Flood based approach: The approach is inspired by the method used in Gnutella network [12] to find a particular file, which floods the network with the query and gets the distributed response from multiple nodes who share the same file.

2. Structured Distributed Hash Table (DHT) based approach: The approach is inspired by the DHT based approach adopted in the Kademlia P2P network [13], [14], [15], which stores the object to the node determined by the hash obtained from the key of the object and uses the same hash to retrieve the object back.

A The Current Situation

In Bittorrent, trackers manage downloading of shared files represented by the torrents and torrents are hosted by web servers. Generally a tracker has its own separate web server for hosting the torrent files, but it is not mandatory. The web servers provide users the search utility to find the torrents hosted in their servers. In the present scenario, a tracker and its torrent server are completely isolated entities and there is no cooperation or integration between them. Even there is no cooperation among trackers. Similarly all the torrent servers also operate in isolation. The web servers provide users the search utility to find the torrents hosted in their servers. If a user wishes to download a file, he first searches for the torrent visiting different servers and only after finding the torrent, starts downloading it. But if the user does not know the URL of the web server which hosts the torrent of the file, he simply can not download the file, and Considering the vastness of the web and popularity of BitTorrent this seems to be most often the case. Although there are some BitTorrent search sites which claim to provide efficient search results in BitTorrent network, but they only send the queries to some renowned torrent servers and display the results returned, yet there exist thousands of such torrent servers which are not covered. In present scenario, there does not exist an elegant solution that integrates these isolated torrent servers and provides an efficient and effective search with high accuracy. We take up the challenge to explore the various alternatives of the existing approach and propose two solutions in the coming sections. These solutions though are subjected to a detailed analysis and verification.

The existing system is also vulnerable to the failure of the torrent server. If the server goes down, even if the tracker is up and running, users can not locate the torrents and thus can not join the tracker and start downloading files. We also find that there is no need of a server at all for hosting the torrent files, what the user needs is just the info hash of the file, which is a 20 byte SHA-1 identifier of the file and the tracker address who manages this file. Once the info hash is available, the user can get a list of the peers involved in downloading the file by contacting the tracker that manages its swarm, and with the slight modification in the BitTorrent protocol, we can make the new peer to download torrent file directly from one of the downloading peers. This will completely eliminate the need of the torrent server and bring down the cost of storage and maintenance. In our proposal, we assume that each has two logical components; 1) a search utility to find the relevant torrents in the BitTorrent network and 2) a traditional management utility that maintains the swarms information. In our explanation, our reference to tracker includes search utility also until we explicitly say the other way.

B Solution

Our proposal is that if we integrate search utilities of the isolated trackers spread worldwide, we can provide better infrastructure for locating and sharing files. All the trackers are connected in a network and the managed torrents from a tracker is made available to other trackers for the purpose of search. When a user wants to search a shared file, he enters the query to the BitTorrent client. The client sends the query to any tracker it is connected to, the tracker forwards the query to the search utility, which then retrieves the information from the network and sends the result back to the client, which is then displayed to the user. Each result contains the meta info of the file like its name, category, info hash of the file and the announce URL of the tracker who manages this file. When the user clicks on a result to download the file, the client sends the announce request for the info hash of the torrent to the specified tracker in the result and the tracker replies with a random list of currently downloading peers and seeds. The client then retrieves the torrent file from one of the peers and starts participating in the swarm.

The first concern that arises with all this explanation is how the trackers are going to form the network. We rule out the presence of a central authority, which helps the trackers to get the addresses of other trackers. The central authority makes the design week and presents a single point of failure. On the contrary we adopt a novel approach, in which the trackers learn the other trackers' addresses from the peers who join its network. A peer downloads the file from multiple trackers in its lifetime, if each peer just stores the tracker's address from which it downloaded the last file and provides this information to the current tracker, then this information is more than sufficient for the tracker to build the knowledge base about other trackers. The tracker selects some entries from its knowledge base and connects to them to join the network of trackers.

Now the question is how the search utilities of the trackers are going to interact, such that they provide the efficient management and retrieval of information in the network of trackers. We examine two approaches to achieve this; first is unstructured flood based approach and second structured DHT based approach. We will explore each approach in details one by one.

1 Unstructured Flood Based Approach

This approach is inspired by the unstructured P2P network Gnutella, which floods the user query in the network and gets a distributed response from different peers. The querying peer forwards the query to all the neighboring nodes it is connected to. Each neighboring node then forwards the query to all the nodes it is connected to, and thus the query is propagated in the network until the number of hops visited reaches the *Time To Live* (TTL) value. The query is generated with some TTL which determines its life as the number of hops, after which the query is discarded. This is a very simple approach and it is very easy to implement. But flooding is not considered a good idea, because of the huge cost of communication and it does not scale well with the size of the network. To reduce the amount of traffic and provide more stability to the network, Gnutella is divided into two kinds of peers, Ultra peers and Leaf peers. Ultra peers are more stable peers and participate in the routing of the messages. An Ultra peer is connected to other Ultra peers and Leaf peers. Leaf peers connect to the Ultra peers. Since the TTL limits the life of the query, the search of the file in the network is unreliable. We can never be sure that a query will cover all the hops in the network, so there is no guarantee that if some file exists in the network then it will be found. The same query sometimes returns results and sometimes fails.

We can connect the trackers in a Gnutella like network, where trackers play the role of Ultra peers and normal peers the role of leaves. As stated before, trackers can easily collect the addresses of other trackers from peers and then connect to the k randomly chosen trackers. Thus the trackers will form a network, where each node has the degree k. Each tracker maintains an index of torrents that it maintains. When a peer requests a search, the query is sent to the trackers it knows. The trackers search the requested item in their index list and send the result back to the peer. At the same time they also forward the query with ip and port number of the requesting peer to all of its neighbor trackers, who carry out the same activity on receiving the query request. To limit the lifetime of the request we use the TTL field and populate it with the value which ensures a greater reach to the network.

Why should we consider the flooding approach when we know the obvious shortcomings of the approach? Definitely, with the increasing size of the network, the traffic becomes more severe in the network, but we should not forget that we are connecting trackers in Gnutella like network not the peers and number of trackers in the world are very few compared to number of the peers. Also the growth in the tracker network will be very slow. So the approach is worth considering even after its shortcomings because of the simplicity and robustness. Though one problem still remains unsolved is the accuracy of the result, since even if the network provides greater reach it does not ensure the complete reach.

2 Structured DHT Based Approach

This approach is inspired by the DHT based P2P protocols. We studied several DHT based P2P protocols, Chord, Tapestry and Kademlia, and realized that concepts of Kademlia DHT can be used to form trackers network. First we explain how Kademlia

[13] works and then we construct DHT based trackers network.

a Kademlia DHT

Kademlia is a Distributed Hash Table (DHT) based P2P network protocol, which uses the XOR metric to route the messages and locate the nodes and objects. A Kademlia node takes advantage of every message that it receives in updating the information about network and exploits this information to tolerate node failures by sending parallel and asynchronous query messages. Each node in the network is recognized by a 160 bit ID which is the SHA-1 hash. An object is identified by a (Key,Value) pair. To store an object in the network 160 bit hash is obtained from its key and then the (Key, Value) pair is stored in the K nodes closest to the hash obtained from the key. To route a message to the nearest node to a given ID, XOR metric is used, which has unidirectional property, i.e. all lookups for the same key converge along the same path irrespective of their originating nodes. Each node in Kadelmia maintains a routing table with the number of entries equal to the number of bits in its ID. Each entry in the table is called a K bucket, since it stores a list of K (IP, UDP Port, Node ID) triples, where each triple has the same prefix as the Node's ID till the i'th bit. Triples in a k-bucket are kept in the sorted order by the time last seen. A new node is inserted in the k-bucket only when it is not full or the old nodes leave the system, thus it prevents the denial of service attack in which the network is flooded with malicious nodes. When the look up request for a key comes to a node, it first obtains the hash for the key and then finds the K nearest nodes to the hash from its K buckets. The node then selects the α nearest nodes from the selected K, and forwards the search request to these α nodes, each receiving node then returns the K nearest nodes to the hash from their K buckets. After receiving the response from the α nodes, the node again finds the α nearest nodes for the hash and repeats the same procedure until the nearest existing node to the hash in the network is found. To determine the nearest node XOR metric is used. XOR of the node ID from the given ID gives the distance between node ID and given ID.

When a node wants to join the network, it must have knowledge of at least one participating node. The node then selects a random ID for self and then performs its own lookup against the node it knows. In the process, it populates its own K buckets and those of the traversing nodes.

Kademlia is used in file sharing networks. Filename searches in the Kademlia network are implemented using keywords. By making Kademlia keyword searches, one can find information in the file-sharing network so that it can be downloaded. Since there is no central instance to store an index of existing files, this task is divided evenly among all clients. The filename is divided into its constituent words. Each of these keywords is hashed and stored in the network, together with the corresponding meta-information of the file. A search involves choosing one of the keywords, contacting the node with an ID closest to that keyword hash, and retrieving the meta-information of the file that contain the keyword.

b BitTorrent

We connect the BitTorrent trackers using Kademlia protocol with some changes which we will explain as we move on in our description. Each tracker in the network plays the role of a node and each torrent the role of an object.

A tracker in the network is recognized by 160 bit SHA-1 obtained from its IP. Each tracker hosts many torrents and each torrent is recognized by the metadata which includes its name, category and some other attributes. Analogous to Kademlia, in the trackers network, a torrent is an object and it is published in the network by (key, value) pair by the tracker, where value consists of info hash, name, category and hosting tracker. Key is a keyword present in its metadata by which the object can be searched. An object may be associated with multiple keys if its metadata includes more than one keyword and it is published in the network for each key by the tracker. A peer can submit the search query for a torrent to a tracker. When a search query arrives to a tracker, it retrieves the keywords from the search string and calculates hashes for each key. The tracker then performs lookup for each hash in the trackers network, sorts the results in the order of their relevance and returns the results to the peer.

When a tracker first time joins the network, it publishes all the torrents it has, so that they can be accessed by other trackers also. At later point of time when the tracker leaves the network and then rejoins it, it just needs to refresh the torrent objects in the network that it manages. When a new torrent is submitted to a tracker, it is also published in the network.

A tracker refreshes all the torrent objects it stores periodically. The refresh period affects the periodic traffic in the network, so it should not be kept small. Also it should not be very large; otherwise it will affect the consistency of lookups in the network because nodes responsible for the keys may leave the network. If a node does not receive refresh message for a (key, value) pair within twice the refresh period, it removes the pair from its list. The refresh period we suggested is tentative and may be tuned to give better result and performance. This arrangement is required to get rid of stale information. When a node leaves the network, objects published by it become stale with the time they are removed from the network.

We need to determine the replication factor K also. Replication is needed to cop with the churn in the network. A node is responsible for the keys which are close to its ID. When a node leaves the network, lookups for the keys it handles are not affected because of the replication in the network. Our intuition is that we can keep the K small, since we are connecting trackers in the Kademlia network, who are lot more stable than the average peers.

We want to maintain the robustness of the system against failures, so a tracker should be functional even if it is cut off from the rest of trackers network. To achieve this, a tracker not only publishes the torrents it maintains, but also keeps a complete index of the torrents in its local BitTorrent network. In the presence of such extreme failure when a tracker is isolated, at least it will be able to satisfy the search requests for its own BitTorrent network. To further optimize the computation time for sorting the results and amount of traffic in the network, when an object lookup is performed for a key, the tracker sends all the keys found in the search string with the key lookup, so that the node responsible for handling the key retrieves the objects for the key based on the relevance of the objects with all the keys present in the search string. The node then sorts the result according to their relevance and sends the result directly to the querying tracker. It helps the querying node in sorting and arranging the results in the order of their relevance.

The changes suggested by us do not hinder or interfere with the existing functionality of BitTorrent protocol. They will only change the way torrents are located, managed and downloaded. Even in the extreme case when all the trackers are isolated, the system will continue working as it works now, only the users connected to a tracker will not be able to search the torrents of other trackers.

CHAPTER VII

CONCLUSION

We conduct a series of experiments and find the interesting results for BitTorrent. These results give us an idea of how the protocol performs and peers behave in real world. We find that a large portion of the torrents population is not usable, since there is no peer at all. Also we find that 67% of the traffic is contributed by just 10% of the torrents. Expected lifetime of a peer is 56.6 minutes in the network. Maximum peers originate from the US and μTorrent is the most favorite BitTorrent client. We also exploit a feature in BitTorrent to mount DDoS attack, though it does not prove powerful. Later we address the content locatibility issue in BitTorrent and propose two solutions. Our proposal opens a new field of research and is subject to improvement and verification.

We see immense potential in BitTorrent. Its use is not only limited to sharing the files, but it is showing its strength in other fields also, Video on Demand is one such field. We are hopeful to see BitTorrent as the most powerful file sharing platform, spreading its wings across other applications.

REFERENCES

[1] L. Guo, S. Chen, Z. Xiao, E. Tan, X. Ding and X. Zhang, “Measurements, analysis, and modeling of BitTorrent systems,” College of William and Mary, Tech Report, WM-CS-2005-08, July 2005.

[2] D. Qiu and R. Srikant, “Modeling and performance analysis of BitTorrent-like peer-to-peer networks,” in *Proc. of ACM SIGCOMM*, August 2004, pp. 367-378.

[3] A. R. Bharambe and C. Herley, “Analyzing and Improving BitTorrent Performance,” Microsoft Corporation, Tech Report, MSR-TR-2005-03, Feb. 2005.

[4] N. Naoumov and K.W. Ross, “Exploiting P2P Systems for DDoS Attacks,” International Workshop on Peer-to-Peer Information Management, 2006, vol. 152, article 47.

[5] X. Wang, Z. Yao and D. Loguinov, “Residual-Based Measurement of Peer and Link Lifetimes in Gnutella Networks,” *IEEE INFOCOM*, May 2007, pp. 391-399.

[6] Wikipedia, “Bittorrent Protocol Specification v1.0,” http://wiki.theory.org/BitTorrentSpecification.

[7] D3F, “The Offical Huge Tracker List,” http://forums.phoenixlabs.org/t85-the-official-huge-f-tracker-list.html, Phoenix Labs Forum, Sep 2005.

[8] D. Roselli, J. R. Lorch and T. E. Anderson, “A Comparison of File System Workloads,” in *Proc. USENIX Annual Technical Conference*, Jun. 2000, pp. 41-54.

[9] D. Leonard, V. Rai and D. Loguinov, "On Lifetime-Based Node Failure and Stochastic Resilience of Decentralized Peer-to-Peer Networks," in *Proc. ACM SIGMETRICS*, Jun. 2005, pp. 26-37.

[10] Z. Yao, D. Leonard, X. Wang and D. Loguinov, "Modeling Heterogeneous User Churn and Local Resilience of Unstructured P2P Networks," in *Proc. IEEE ICNP*, Nov. 2006, pp. 32-41.

[11] D. Stutzbach and R. Rejaie, "Understanding Churn in Peer-to-Peer Networks," in *Proc. ACM IMC*, Oct. 2006, pp. 189-202.

[12] D. Stutzbach, R. Rejaie and S. Sen, "Characterizing Unstructured Overlay Topologies in Modern P2P File-Sharing Systems," *IEEE/ACM Trans. Networking*, April 2008, pp. 267-280.

[13] P. Maymounkov and D. Mazieres, "Kademlia: A Peer-to-peer Information System Based on the XOR Metric," *IPTPS*, March 2002, vol. 2429, pp. 53-65.

[14] J. Falkner, M. Piatek, J. P. John, A. Krishnamurthy and T. Anderson, "Profiling a million user DHT," *ACM IMC*, 2007, pp. 129-134.

[15] I. Baumgart and S. Mies, "S/Kademlia: A Practicable Approach Towards Secure Key-Based Routing," *ICPADS*, Dec. 2007, pp. 1-8.

VITA

Videsh Sadafal received his Bachelor of Engineering degree in Information Technology in India from National Institute of Technology, Karnataka in May 2004. He received his Master of Science degree in Computer Science from Texas A&M University in August 2008. His research interests include P2P networks, distributed and large-scale systems, and computer networks. He can be reached at:

Videsh Sadafal
538, Trimurti Nagar, DamohNaka
Jabalpur, M.P.
India, 482002

The typist for this thesis was Videsh Sadafal.