Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Elio Gubser

Measuring Explicit Congestion Negotiation (ECN) support based on P2P networks

Semester Thesis SA-2015-FS
February 2015 to May 2015

Tutors: Brian Trammell & Mirja Kühlewind
Supervisor: Prof. Dr. Bernhard Plattner

Abstract

Explicit Congestion Negotiation (ECN) is a TCP/IP protocol extension, which allows routers to signal congestion. The goal of ECN is to reduce packet loss, latency and jitter. Unfortunately, this extension is disabled by default in operating systems because old and faulty hardware breaks connectivity.

We want to find an answer to how good ECN support is in the Internet. A previous study assessed ECN support of webservers. A software has been written for measuring a million websites from multiple vantage points simultaneously.

This semester thesis investigated ECN support by using a similar approach in the BitTorrent network, in which most participants are end users at home. In contrast to websites, however, there is no easily obtainable list of hosts in the BitTorrent network. Therefore the existing software has been modified and extended with an efficient algorithm for collecting IP and port number of participants in the BitTorrent network. In addition, the whole measurement process has been automated and is communicating over the mPlane protocol.

Using this new software, a measurement has been performed on one million hosts from five vantage points. The majority of hosts in the BitTorrent network were ECN safe. About two hosts per thousand were unreachable when ECN negotiation was enabled. The fact that this figure is within the same order of magnitude as in the webserver measurements shows that ECN related connectivity problems not only occur at webservers but also close to the end user.

Contents

Chapter 1

Introduction

Conventionally, a TCP/IP connection endpoint interprets packet drops as congestion. Explicit Congestion Negotiation (ECN) is a TCP/IP protocol extension allowing routers to notify the sender of the fact that there is a congestion, before having the router forced to drop packets[2]. If the sender receives such a notification, it should reduce its congestion window preventively to avoid a loss of packets resulting in better latency and less jitter. Unfortunately, this extension is disabled by default in most operating systems today, because there are still some old or faulty devices which break connectivity.

1.1 Motivation

We would like to know: How good is ECN support in the Internet?

To answer this question, we need to quantify the deployment of ECN in the Internet by performing large scale measurements. In fall 2014, a measurement study conducted from commercial grade servers assessed ECN support of a million popular websites using the HTTP protocol[1].

Based on this approach, this thesis focuses on the BitTorrent network, where most participants are end users at home. BitTorrent has been chosen because it is the largest peer to peer network today. In contrast to websites, there is no list of hosts which is easily obtainable in the BitTorrent network. Thus an algorithm has to be developed to extract such a list.

1.2 Goals

This leads to the following tasks:

- Get familiar with existing tools and results (ecnspider, qof, measurement study).
- Find and assess feasible solutions on how to get ip addresses from the BitTorrent network (Section 3.1).
- Integrate chosen variant into ecnspider2 (Section 3.2.2).
- Implement mPlane support into ecnspider2 (Section 3.2.3).
- Write a tool to conduct measurements from multiple vantage points at the same time (Section 3.3).
- Perform measurement and evaluate results (Chapter 4).

1.3 Code Archive

Accompanying this thesis document is a zip file containing the complete ecnspider2, a patch for mPlane (Section 3.2.3) and the code for testing method 2 *Custom Software* (Section 3.1.2).

Chapter 2

Background

The ECN measurement method presented in section 2.2 has been used in a previous study about ECN support of websites using HTTP[1]. At that time, a program called `ecnspider` has been developed. Its reimplementation, `ecnspider2`, is described in section 2.3.

2.1 ECN Safety and Negotiation

Before ECN is actually used in a connection, both hosts have to negotiate ECN in the TCP handshake. An ECN safe connection is where the connection succeeds whether or not ECN is negotiated. An ECN broken connection is where the connection fails only because the initiating host shows willingness to negotiate ECN.

2.2 ECN Measurement Method

To check ECN support for a given IP address/port pair (called *endpoint address* from now on), two consecutive connection attempts are made (Figure 2.1). In order to determine whether the endpoint is reachable at all, ECN negotiation is disabled for the first connection attempt. After that, ECN negotiation is enabled and the second connection attempt is performed. Based on failure or success of the two connection attempts, the endpoint is categorized into *ECN supported*, *ECN broken*, *nobody home* or *transient/other* according to table 2.1.

So far, it is impossible to find out if the failure happened at the endpoint or on the path. To gain more insight into this matter, the same measurement is simultaneously conducted from geographically distant vantage points in order to have the connections routed via different paths (Figure 2.2). If ECN-negotiating connection attempts fail only on some paths, the connectivity is

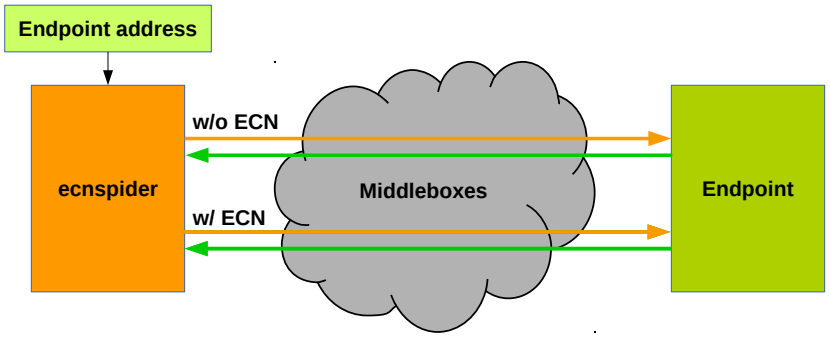

Figure 2.1: Procedure for testing a single endpoint. Two connection attempts are made using a machine, running the measurement software *ecnspider2*, to the endpoint through the Internet (middleboxes).

Connection	w/o ECN	w/ ECN
ECN safe	success	success
ECN broken	success	failure
nobody home	failure	failure
transient/other	failure	success

Table 2.1: Characterization of ECN safety based on the results of two connection attempts.

considered *path-dependent* and the problem most probably lies within a middlebox on the path. On the other hand, if all paths fail, it is called *site-dependent* and the problem most probably lies near the endpoint.

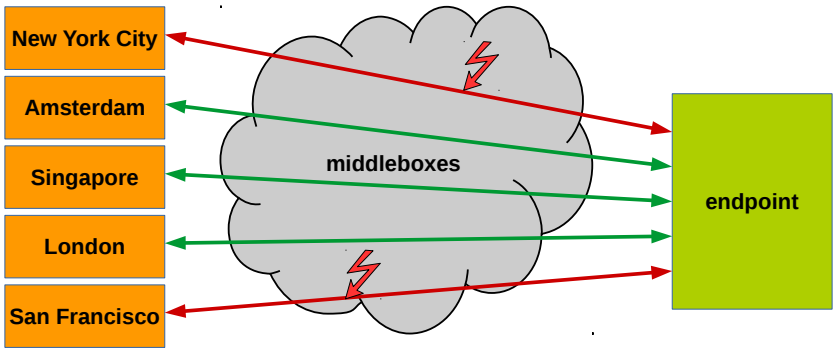

Figure 2.2: When some vantage points fail to establish an ECN-negotiating connection, the connectivity is considered path-dependent.

2.3 Structure of ecnspider2

To automate the measurement presented in the last section, ecnspider has been developed by Damiano Boppart in 2014[1]. It has been recently reimplemented by Brian Trammell and is organized as shown in figure 2.3.

A measurement is initiated by calling `add_job()` with an endpoint address. Any number of jobs can be added and will eventually be executed. Several hundred jobs are executed in parallel by worker threads. At the beginning, ECN negotiation is turned off in the operating system by the configurator. Each worker tries to establish a connection to the endpoint specified in the job. After all workers have finished connecting, ECN negotiation is turned on by the configurator. Now each worker tries to establish a second connection to the endpoint. Once all workers have finished connecting, the results of both connection attempts are stored in the `Result Queue` and all open connections are closed.

At the same time, a program called Quality of Flow[4] (QoF) observes the network traffic and stores detailed information, such as TCP/IP flags and TTL values into the `Flow Queue`.

In the end, items from the `Flow Queue` and `Result Queue` are combined by comparing endpoint IP and local port[1]. Then the user-specified callback function `result_sink()` is called with the complete result.

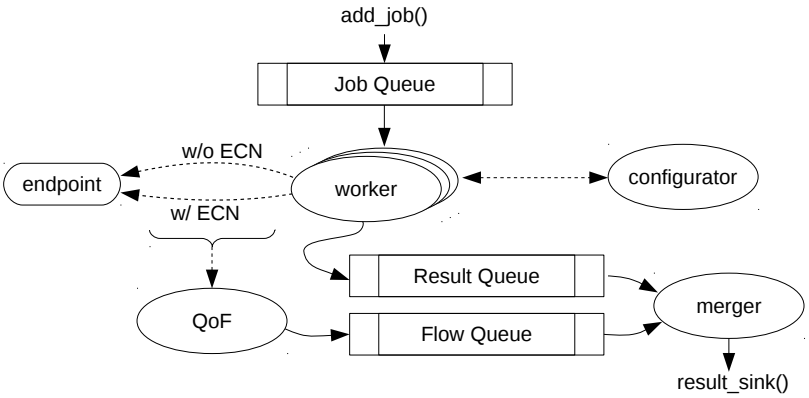

Figure 2.3: Simplified structure of ecnspider2. Each one of several hundred worker threads performs a measurement on a different endpoint, while the configurator changes the operating system's ECN negotiation setting. Detailed TCP flow information is acquired using QoF. The merger combines the connection results and flow observation into a measurement result.

[1] It is not sufficient to compare only by endpoint IP, because there are two connections which are being initiated. Furthermore, to ensure that the local port is not the same for both connection attempts, the first connection is kept open while connecting for the second time.

Chapter 3

Methodology

The practical part of this thesis consists of enhancing `ecnspider2` to perform measurements using endpoints connected to the BitTorrent network. For extracting endpoints from the network, three methods are discussed in section 3.1. The expansion of `ecnspider2` is explained in section 3.2. The setup used to perform the measurement is presented in 3.3. Finally, incorporation of `ecnspider2` into `mPlane`[3] is described in section 3.2.3.

BitTorrent clients transfer data using either TCP or uTP[6]. uTP is a BitTorrent transport protocol working over UDP. It has been developed to improve latency and congestion issues compared to traditional BitTorrent transport over TCP. However, this measurement study only investigates the TCP protocol, because uTP doesn't have ECN functionality.

Note: Differences in Measuring ECN in the BitTorrent protocol compared to HTTP

In contrast to an HTTP server, a BitTorrent client immediately drops the connection if it realizes that it can't serve the request. Only a TCP handshake can be observed. ECN signalling is disabled for TCP control packets. This prevents measurement of ECN support in the IP layer and limits the observation to ECN negotiation in the TCP layer. Fixing this deficiency is a task that still has to be conducted (Chapter 6).

3.1 Acquisition of IP Address/Port Pairs

As there is no list of running BitTorrent clients readily available, a method for collecting endpoint addresses is needed. Three concepts to achieve this are presented in the following subsections, concluding with the one which has actually been implemented.

3.1.1 Method 1: Passive Monitoring

An unmodified BitTorrent client is started and several popular torrents are added. The client will connect to other clients and download the data as usual. During this operation, the network traffic is passively observed and every client's IP and port is recorded.

Comment

This method is considered easy to implement. But unfortunately, it isn't suitable for collecting thousands of endpoint addresses because the client only needs to connect to a couple of clients to be able to download at full speed.

3.1.2 Method 2: Custom Software

To speed up the address collection process, the software is customized so that it will connect to many more endpoint addresses. At the same time, instead of monitoring the network traffic, the addresses should be extracted directly from the BitTorrent client through API calls.

One way to achieve this is by adding firewall rules (e.g. `iptables`) in order to block traffic to all learned endpoint addresses. The client is then forced to ask for more clients to download. Another way is to use blacklisting if it is supported by the software.

For a quick test of the feasibility of the method, a small python script has been written[1]. It uses libtorrent[9] to do the heavy lifting and maintains an IP-filter (blacklist) obeyed by libtorrent to disallow connections to already known clients. This forces libtorrent to repeatedly ask for more clients. After a certain number of endpoint addresses have been collected, the script writes them into a file and shuts down the client.

Comment

Using a custom software is a lot faster than the previous method. With libtorrent it is particularly easy to automate and extract endpoint addresses without needing to observe network traffic. Drawback is that the code is hacky and inelegant. To enforce the blocklist, one has to pause the client, update blocklist and resume client. This generates a lot of overhead and may result in unstable performance.

3.1.3 Method 3: Exploiting BitTorrent Protocols

Another approach is to look directly at the underlying BitTorrent protocols. More specifically, the protocols used by the client to learn new endpoint addresses. There are two major protocols in use today: the tracker protocol[5] and the distributed hash table (DHT) protocol[7]. The former needs a central address server, called tracker, to which every client has to connect and request information about other clients. The latter protocol doesn't need a server, because each client is assigned to maintain a part of the address list, forming a decentralized database. The DHT protocol also supports IPv6 by an extension[8].

One may assume that tracker servers impose limitations on the number of requests made by the client, effectively limiting the speed of address collection. Therefore, this thesis concentrates on the decentralized DHT protocol.

The DHT protocol runs over UDP using the same port number as the BitTorrent protocol, which uses TCP. In this protocol, a client is referred to as a *node*. Each node has a unique 160 bit identifier called *node ID*. It is randomly generated from the same number space as BitTorrent infohashes used for identifying torrents.

To start downloading or uploading a torrent, a node needs to find out which nodes are actually serving the torrent. The DHT requires every node to store torrent infohashes which are close to its own node ID.

Each node maintains a routing table of node ID and endpoint addresses with a preference for nodes having a small distance. Where the distance is defined as $d = A \; xor \; B$.

To find the endpoint address of a particular node in the network, the node closest to it in the local routing table is contacted. The contacted node then returns a list of endpoint addresses of nodes yet closer to the particular node. Repeating this process, one comes closer and closer to the node in question and finally acquires its endpoint address. This functionality is called `find_node` in the protocol specification. Figure 3.1 illustrates it with an example.

By using the `find_node` functionality of the DHT protocol, acquiring endpoint addresses is straightforward. As shown in Figure 3.2, an algorithm sends random requests to nodes, which in turn responds with a list of nodes. These nodes are then also contacted using random requests. The process is repeated until the desired number of endpoint addresses is reached.

[1] Located in the code archive inside `method2/`

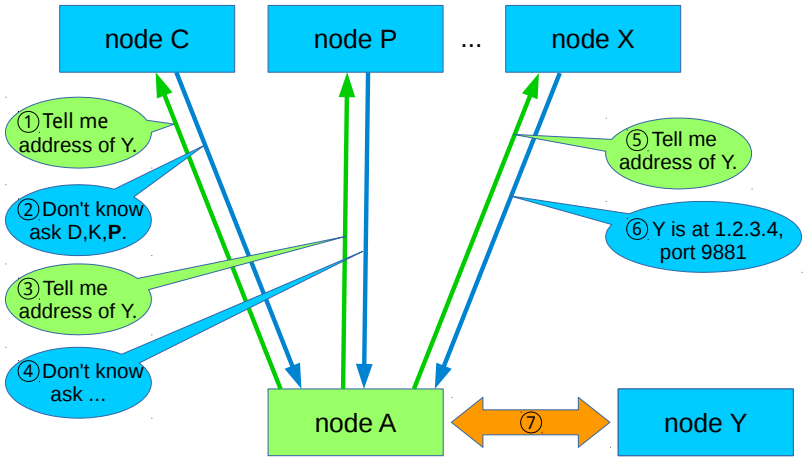

Figure 3.1: A network of nodes, where A, C, P... indicate node IDs. In this example, node A wants to know the endpoint address of node Y. ① First it requests this information from the known node C closest to Y. ② But C doesn't know Y, it only knows some nodes closer to Y. ③ Node A asks node P and ④ still gets no satisfying answer. This process is repeated until it arrives at node X ⑤, which returns the endpoint address of Y ⑥. Finally, node A is able to connect to node Y ⑦.

Comment

Repeating this process quickly yields a massive amount of possible targets with a minimal effort. It is superior to the previous methods and therefore chosen for implementation.

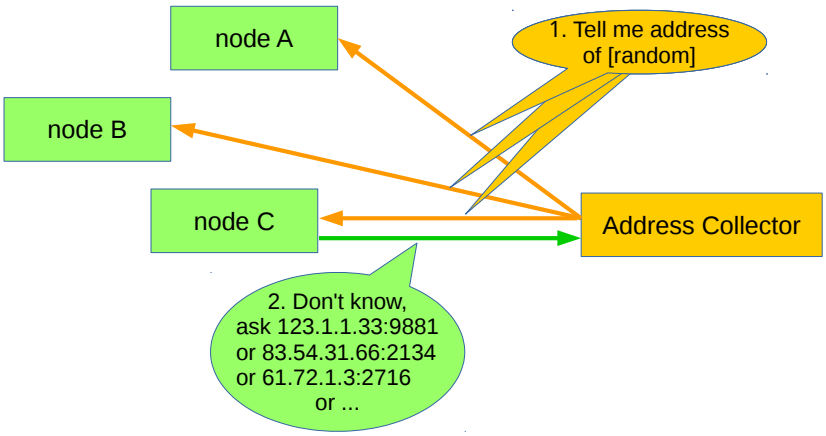

Figure 3.2: Concept for collecting addresses from the BitTorrent DHT network.

3.2 Implementation

This section explains the building blocks of the measurement setup as illustrated in figure 3.4.

Among other changes, ecnspider2 has been modified to be easily extendable for supporting more protocols and operating systems (Section 3.2.1).

The address collector described in section 3.1.3 has been worked out as illustrated in figure 3.3 and described in section 3.2.2. The Implementation centers around a Python class, called `BtDhtSpider`, following the generator pattern. This means, getting endpoint addresses from the DHT network is as easy as iterating over a list (Listing 3.1).

In section 3.2.3, the mPlane components for the address collector and ecnspider2 are presented. Utilising these components, an mPlane client application has been written to perform automated measurements (Section 3.3).

The final analysis has been done separately (Section 3.4).

The complete ecnspider2 - including address collector algorithm and mPlane code - is located in the code archive inside `pathtools/`.

3.2.1 Amendments to ecnspider2

The following is a summary of the changes that have been made to ecnspider2.

Initially, ecnspider2 only supported HTTP connections. For each HTTP connection a `GET /` request is performed. To add support for other protocols, such as BitTorrent, the HTTP code is stripped away from class `EcnSpider2`, now measuring TCP handshake only. HTTP related code is moved into a new subclass `EcnSpider2Http`. Other protocols can be easily added to the ecnspider2 suite by overriding `connect()`, `post_connect()`, `ignore_flow()` of class `EcnSpider2`.

As explained in the introduction of this chapter 3, only observing the TCP handshake is possible when contacting a BitTorrent client. For that reason, the `EcnSpider2` class is used for measuring BitTorrent clients.

Other additions:

- Automatically select ECN configuration code, depending on the operating system. Currently Darwin/OS X and Linux are supported, but this can be extended easily.
- After investigating spurious missing measurement data, it was found out that to actually close a TCP connection immediately `socket.shutdown(socket.SHUT_RDWR)` has to be called before socket.close().
- For supporting the mPlane reference protocol implementation's interrupt pattern, a thread called interrupter has been added to qofspider, which repeatedly calls a user-supplied function and performs a clean teardown when it returns `True`.

3.2.2 Address Collector

Following the general idea of section 3.1.3, the detailed workflow is presented here. Before the start, the `Nodes to Ask` queue is filled with known endpoint addresses called bootstrap nodes ① (See figure 3.3). Afterwards, the sender thread draws endpoint addresses from this queue and sends requests to nodes at these addresses.

Unfortunately, if the request rate is too high, the kernel will *silently* drop packets before even sending them out to the network interface. Therefore, rate limiting is implemented into the algorithm as described in the next paragraph.

To limit the amount of running requests and bandwidth, the sender will issue a request only if the following conditions are met ②:

- `Output Queue` size falls under a pre-defined threshold.

- Count of `Running Requests` is below a maximum number of running requests.
- The average number of bytes sent in a time interval is below a maximum bandwidth limit. The average is calculated with the help of the `Bytes Sent` queue. For each sent packet, a pair `(count of bytes sent, current time)` is added to the queue. Entries older than a specific time, called `slot time`, are deleted from the queue. The average bandwidth is then calculated as the sum of bytes divided by `slot time`.

If these conditions are met, the request is sent and a request state, including the current time, is added to the list of `Running Requests` ③.

The sender thread scans this list and deletes entries older than a specific time (timeout) ④.

Assuming the node answers to our request in time, the receiver removes the corresponding request entry in `Running Requests` ⑤ and analyzes the response data. It adds the newly learned endpoint addresses to the `Nodes to Ask` queue ⑥. If the queue is full, it removes older items and replaces them with new ones.

Then the receiver puts all learned endpoint addresses in the `Output Queue` ⑦. Optionally, the receiver maintains a set of already encountered endpoint addresses in `Unique Set` to ensure that each endpoint address is put only once into the `Output Queue`.

Finally, endpoint addresses can be obtained from `BtDhtSpider` by simply iterating over the object as shown in listing 3.1.

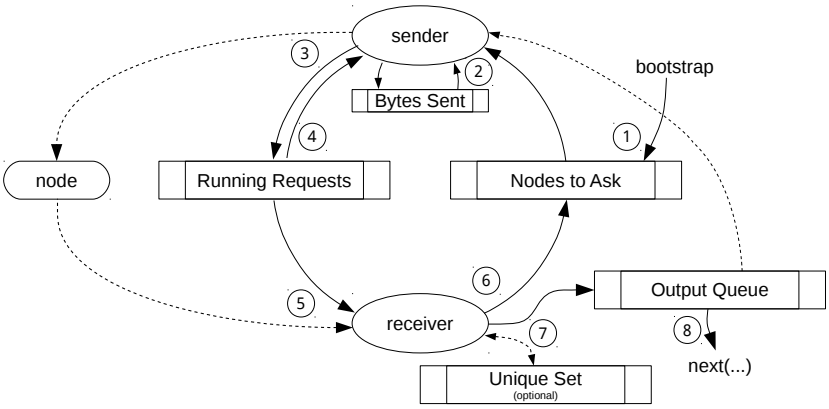

Figure 3.3: Overview of the address collector. A sender and a receiver thread collect endpoint addresses with the help of various lists and queues. A detailed description is found in section 3.2.2.

```
1  dht = BtDhtSpider()
2  dht.start()
3  for idx, addr in enumerate(dht):
4    print(dht)
5    if idx >= 100:
6      break
7  dht.stop()
```

Listing 3.1: Usage example of `BtDhtSpider`. Prints every learned address to standard output.

3.2.3 Integration into mPlane

For supporting mPlane, two components and a client application have been written. The client application, called master, controls the measurement process. It retrieves endpoint addresses from the address collector component, called `btdhtspider`. Then it sends them to the `ecnspider` component. The detailed setup is explained in section 3.3. The terminology used in this section is declared in [3].

Name	Type	Description
btdhtspider.nodeid	string	BitTorrent Distributed Hash Table node identifier (in hex)
btdhtspider.unique	boolean	Assertion (or negation) that only unique addresses should be returned
btdhtspider.count	natural	Count of addresses to collect
ecnspider.ecnstate	boolean	Assertion (or negation) that ECN negotiation is enabled in the OS.
ecnspider.initflags.fwd	natural	TCP initialization flags in forward direction.
ecnspider.synflags.fwd	natural	TCP SYN flags in forward direction.
ecnspider.unionflags.fwd	natural	TCP flags in forward direction.
ecnspider.initflags.rev	natural	TCP initialization flags in reverse direction.
ecnspider.synflags.rev	natural	TCP SYN flags in reverse direction.
ecnspider.unionflags.rev	natural	TCP flags in reverse direction.
ecnspider.ttl.rev.min	natural	Minimum TTL seen in reverse direction.

Table 3.1: New registry entries used by ecnspider in addition to the default registry.

```
[module_ecnspider]
module = ecnspider2.mp_component
# custom registry url
reguri = https://n.ethz.ch/~egubser/ecnregistry.json
# count of worker threads
worker_count = 200
# timeout for measurement connections
connection_timeout = 4
# libtrace URI which QoF should monitor.
interface_uri = ring:wlan0
# port number for communication with QoF
qof_port = 54739

# other optional arguments:
# ip4addr = 0.0.0.0   # bind ecnspider to this IPv4 address
# ip6addr = ::        # bind ecnspider to this IPv6 address
# btdhtport4 = 9881   # bind address collector to this IPv4 address
# btdhtport6 = 9882   # bind address collector to this IPv6 address
```

Listing 3.2: ecnspider2 relevant section in the component configuration file.

Listed in table 3.1 are additional ecnspider-specific elements in the mPlane registry. The constraint `[*]` declares a multivalue element. Multivalue parameters are an experimental feature of the mPlane protocol. At the time of writing, this feature was broken but a patch has been written as part of this thesis[2].

Table 3.2 shows the capabilities of the ecnspider component and table 3.3 shows the capabilities of the address collector component.

Accompanying the component is a configuration file `mp_component.conf`. The relevant section is shown in listing 3.2. The client also has a configuration file `mp_client.conf`, shown in listing 3.3.

[2]Located in the code archive inside `mplane/`

Parameter	Constraint
destination.ip4 / .ip6	[*]
destination.port	[*]

Result
source.port
destination.ip4 / .ip6
destination.port
connectivity.ip
ecnspider.ecnstate
ecnspider.initflags.fwd
ecnspider.synflags.fwd
ecnspider.unionflags.fwd
ecnspider.initflags.rev
ecnspider.synflags.rev
ecnspider.unionflags.rev
ecnspider.ttl.rev.min

Table 3.2: ecnspider-ip4 / ecnspider-ip6: Capability for TCP handshake-only measurement.

Metadata	Value
source.ip4 / .ip6	(defined in config)
source.port	(defined in config)

Parameter	Constraint
btdhtspider.count	*
btdhtspider.unique	*

Result
destination.ip4 / .ip6
destination.port
btdhtspider.nodeid

Table 3.3: btdhtspider-ip4 / btdhtspider-ip6: Capability for BitTorrent address collection.

```
[ecnspider]
# list of vantage points, keyword = host:port
# the keyword is used in the results file as vantage point identifier.
ams = path-ams.corvid.ch:18888
lon = path-lon.corvid.ch:18888
nyc = path-nyc.corvid.ch:18888
sfo = path-sfo.corvid.ch:18888
sin = path-sin.corvid.ch:18888

[client]
# url for address collector.
# either btdhtspider-ip4 or btdhtspider-ip6 is allowed.
btdhtspider-ip4 = path-ams.corvid.ch:18888
# how many measurements should be done in a job.
chunk_size = 10000
```

Listing 3.3: ecnspider2 relevant sections in the client configuration file.

3.3 Measurement Setup

The measurement is conducted from five vantage points as shown in figure 3.4. Endpoint addresses are collected from the BitTorrent network and sent to the master. The master formulates jobs and sends them to each vantage point, which in turn perform the measurement. After that, the results are reported back to the master, which saves all results into a csv file for later analysis (Section 3.4).

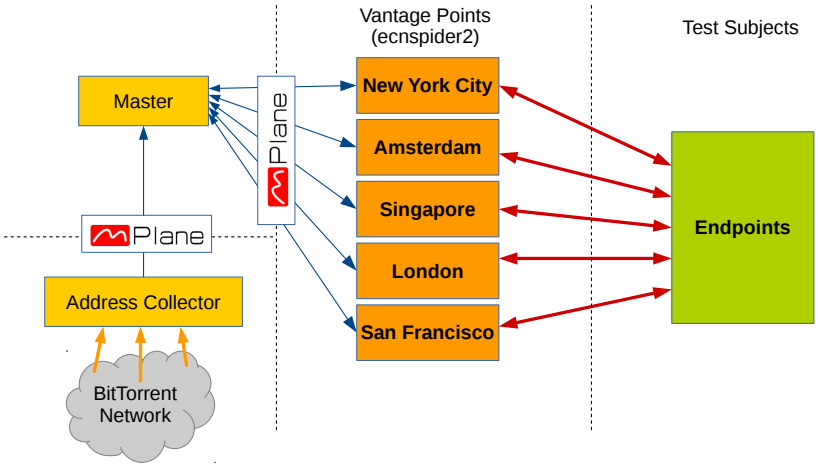

Figure 3.4: Measurement setup consisting of master, address collector, vantage points and test subjects. The address collector retrieves endpoint addresses from the BitTorrent network and gives them to the master, who orders the vantage points to perform measurement on the endpoints. The results from all vantage points are finally collected and analyzed at the master.

3.4 Analysis

The analysis is inspired by the former measurement study[1], but it has been written from scratch[3].

[3]Located in the code archive under `pathtools/ecnspider2/evaluation.ipynb`

Chapter 4

Results

The measurement was performed on 11 Mai 2015 targetting 1 million endpoints from five commercial grade servers rented from DigitalOcean: New York City, Amsterdam, Singapore, London and San Francisco. Of all contacted endpoints, 687089 (68.71%) were online, i.e. reachable by at least one vantage point.

Table 4.1 shows the key figures obtained by combining the results (Table 2.1) from all vantage points. The vast majority (92.34%) of all online endpoints were reported *ECN safe* from all vantage points, which is slightly lower compared to the HTTP study[1] (95.86%). On the other hand, 0.21% were reported as *ECN broken*, implying site-dependent connectivity. This value is better than in the HTTP study (0.38%). In the third case (1.45%), some paths have failed, but it wasn't because of ECN. The last case with a percentage of 6.01% is for measurements which didn't fit into the other three cases. It contains transients and endpoints with potentially path dependent connectivity.

Hosts	pct	Description
634426	92.34%	ECN safe on all paths
1441	0.21%	ECN broken on all paths
9936	1.45%	Connection failure on some paths, not ECN related
41286	6.01%	Transient / Potentially path dependent
687089	**100.00%**	**Total online endpoints**

Table 4.1: Connectivity statistics, of 687089 IPv4 hosts, all vantage points, 11 Mai 2015

Looking at the results from the London vantage point yields more information about the participants of the BitTorrent network. According to the TTL spectrum (Figure 4.1), there are no endpoints with TTL over 128 (Solaris or Google). Endpoints with TTL > 64 are considered Windows machines and TTL ≤ 64 are considered Unix derivatives, such as Linux and Mac OS X. Not surprisingly, the majority of BitTorrent users have Windows machines (70.44%).

Heterogenous Linux machines

Regarding ECN safety, there is an unexpected high percentage of Transient / Potentially path dependent connectivity (14.40%) for Linux / OS X machines (Table 4.2).

This may be due to the strong heterogenous nature of Linux operating systems. Three of many possible causes could be: *1) Embedded Systems* like NAS, routers and media centers can have integrated BitTorrent clients. These systems normally run a highly customized Linux operating system. *2) Seedboxes* are rented servers which run BitTorrent clients within a specialized datacenter network. There may be some accellerator device or load balancing in place. *3)* There are BitTorrent apps for Android (and also iOS) smartphones. If such a mobile device is on a train or in a car, it often has to switch its base station, resulting in unstable connectivity.

Linux / OS X	pct	Windows	pct	Description
167595	82.64%	466831	96.61%	ECN safe on all paths
557	0.27%	884	0.18%	ECN broken on all paths
5453	2.69%	4483	0.93%	ECN unrelated connection failure on some paths
29202	14.40%	11024	2.28%	Transient / Potentially path dependent
202807	**100.00%**	**483222**	**100.00%**	**Total records**

Table 4.2: Connectivity statistics, of 686029 IPv4 hosts, London, 11 Mai 2015

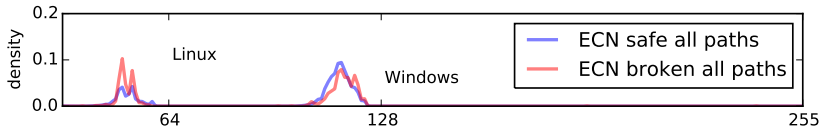

Figure 4.1: TTL spectrum of connectivity, of 686029 IPv4 hosts, London, 11 Mai 2015

ECN Negotiation

Almost all Windows machines do not negotiate ECN (Table 4.3, Figure 4.2). This corresponds with the fact, that ECN negotiation is disabled by default in all Windows versions except Windows Server 2008 and newer. It seems that some people (5781 hosts, 1.24%) are running a BitTorrent client in a Windows Server corporate infrastructure.

The ECN negotiation rate of Linux machines (68.28%) is almost the same as experienced in the HTTP study (69.73%).

Overall negotiation rate (19%) is much lower compared to the HTTP study (56%). This is simply because most of the clients are Windows machines.

ECN safe on all paths	Linux	pct	Windows	pct	Both	pct	HTTP
negotiated	114432	18%	5781	1%	120213	19%	56%
not negotiated	53163	8%	461050	73%	514213	81%	44%

Table 4.3: ECN negotiation, of 634426 IPv4 hosts, London, 11 Mai 2015

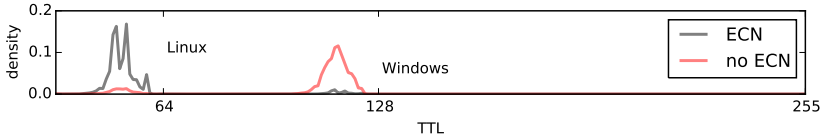

Figure 4.2: TTL spectrum of ECN negotiation, of 634426 IPv4 hosts, London, 11 Mai 2015

Chapter 5

Conclusion

The existing measurement software ecnspider2 has been modified to be easily extendable for supporting more protocols and operating systems (Section 3.2.1). An efficient algorithm for collecting IPv4 and IPv6 addresess from the BitTorrent network has been written (Section 3.2.2). A set of tools have been developed to facilitate automated measurement and analysis (Section 3.3 and 3.4). All communication is done over the mPlane protocol (Section 3.2.3).

By measuring BitTorrent clients from commercial grade servers, valuable information has been collected about ECN support in the downstream. A large measurement of 1 million endpoints was performed on 11 Mai 2015 from five vantage points. The fact that most BitTorrent clients ran on Windows desktop machines hardens the assumption that we are dealing with end users at home.

The majority of hosts in the BitTorrent network were ECN safe (92.34%), which is slightly worse compared to webservers (95.86%, HTTP study[1]). Unfortunately, when ECN negotiation was enabled, a small fraction of hosts (0.21%) was not reachable at all. Altough this value is better than in the HTTP study (0.38%), it is still within the same order of magnitude.

Apparently, ECN related connectivity problems not only occur at webservers but also near the end user. In my personal experience, my internet access at home breaks connectivity when ECN negotiation is enabled.

Chapter 6

Outlook

- A notable proportion (6.01%) of measured endpoints showed Transient / Potentially path dependent characteristics (Table 4.1). Further investigation could lead to interesting insights. For example, one could find out the proportion of stable path dependency. To determine that a particular endpoint exhibits path-dependent connectivity, the measurement needs to be repeated over time and the result should always be the same.
- A similar task would be to find out which categories of Linux devices contribute to the 14.40% in table 4.2.
- As explained in the introduction of chapter 3, only observing the TCP handshake is possible when contacting a BitTorrent client. To perform actual data transfer, the request from ecnspider2 needs to contain a valid torrent infohash, which has to be obtained from outside the DHT network. For example, by acquiring torrent infohashes by digesting RSS feeds from BitTorrent trackers and then looking for nodes seeding those torrents using the DHT protocol. When performing the measurement, the corresponding torrent infohash is sent as part of the BitTorrent handshake. The disadvantage here is that it adds a lot of complexity to the address collector algorithm and ecnspider2.

List of Figures

List of Tables

Bibliography

[1] B. Trammell, M. Kühlewind, D. Boppart, I. Learmonth, G. Fairhurst, and R. Scheffenegger: Enabling Internet-Wide Deployment of Explicit Congestion Notification, Proceedings of the 2015 Passive and Active Measurement Conference, New York (March 2015)

[2] K. Ramakrishnan, S. Floyd, D. Black: The Addition of Explicit Congestion Notification (ECN) to IP, RFC 3168, IETF (September 2001)

[3] B. Trammell (ed), mPlane Architecture Specification, mPlane Public Deliverable 1.4, October 2014.

[4] B. Trammell: Quality of Flow: IPFIX flow meter based on YAF, focused on passive TCP performance measurement, September 2014
https://github.com/britram/qof/tree/develop
(Retrieved: 04.05.2015, commit 32699377c5)

[5] B. Cohen: The BitTorrent Protocol Specification, BEP 3 (January 2008)
http://www.bittorrent.org/beps/bep_0003.html
(Retrieved: 27.04.2015)

[6] A. Norberg: uTorrent Transport Protocol, BEP 29 (June 2009)
http://www.bittorrent.org/beps/bep_0029.html
(Retrieved: 27.04.2015)

[7] A. Loewenstern, A. Norberg: DHT Protocol, BEP 5 (January 2008)
http://www.bittorrent.org/beps/bep_0005.html
(Retrieved: 27.04.2015)

[8] J. Chroboczek: BitTorrent DHT Extensions for IPv6, BEP 32 (October 2009)
http://www.bittorrent.org/beps/bep_0032.html
(Retrieved: 11.05.2015)

[9] A. Norberg: libtorrent python binding
http://www.libtorrent.org/python_binding.html
(Retrieved: 07.05.2015)

Problem Statement & Declaration of Originality

Measuring Explicit Congestion Negotiation (ECN) support based on P2P networks

Master / semester thesis

Background

Explicit Congestion Notification (ECN) [1] is a TCP/IP extension that allows congestion signaling without packet loss and therefore can greatly increase the performance on the Internet. Even though ECN was standardized in 2001, and it is widely implemented in end systems, it is barely deployed. This is due to a history of problems with severely broken middleboxes shortly after standardization, which led to connectivity failure and guidance to leave ECN disabled. Recent measuremnet studies [2, 3] have show an increasing support of ECN on websevers of up to 50% which is the first step for ECN deployment on the Internet. Further on-going activities in research and standardization aim to make the usage of ECN more beneficial. Therefore it is import to assess the marginal risk of enabling ECN negotiation by default on client end-systems.

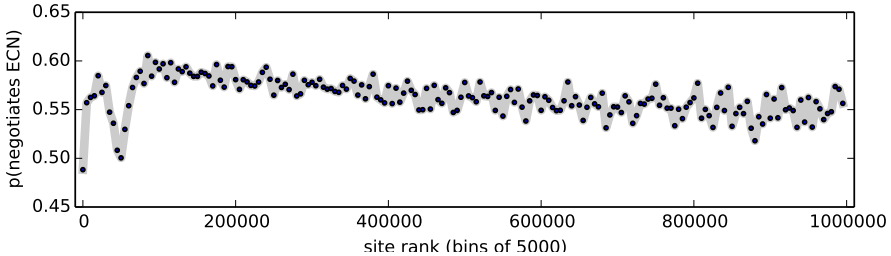

Figure 1: Proportion of websites negotiating ECN by Alexa top 1 mio rank

Thesis Goals

This measurement study aims to further assess the deployment status of ECN support utilizing Peer-to-Peer (P2P) networks. Based on popular content a set of IP addresses should be identified and probed for ECN support. The measurement methodoloy will be based on the same approach as used in [3]. Therefore a measurement tool called ECN-Spider is available that must be adapted to the P2P enviroment.

This leads to the following tasks:

1. Creation of a target IP address list based on popular P2P content
2. Adaptation of ECN-Spider to utilize an exsiting P2P client or perfrom TCP connection tests
3. Evaluation of measurement results to assess ECN support and detect connectivity problems
4. Interactive reprensentaion of the results on http://ecn.ethz.ch

Contact: Brian Trammell, trammell@tik.ee.ethz.ch, ETZ G93
Mirja Kühlewind, mirja.kuehlewind@tik.ee.ethz.ch , ETZ G93

Professor: Prof. Dr. Bernhard Plattner

References:

1. Ramakrishnan, K., Floyd, S., Black, D.: The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, IETF (September 2001)
2. Kühlewind, M., Neuner, S., Trammell, B.: On the State of ECN and TCP Options in the Internet. In: Proc. Passive and Active Measurement 2013, Hong Kong (March 2013)
3. http://ecn.ethz.ch, November 2014