μTP

Heze Yin, Laixian Wan, Peiqing Lu, Xiaoduan Chang

GitHub link: https://github.com/drcxd/CS655_GENI_Project_uTP
Project on GENI: `utp-experiment`

1 Introduction

One major problem that BitTorrent users experience is throttling of non-BitTorrent TCP connections. A BitTorrent client can simultaneously talk to many peers, each using multiple TCP connections. As a result, non-BitTorrent TCP connections are limited in speed, sometimes significantly when there are tens of BitTorrent TCP connections. This is because TCP protocol is designed to split bandwidth equally among all TCP connections. To overcome this problem, Micro Transport Protocol, also known as μTP or uTP is developed. It is an open-sourced UDP-based variant of the BitTorrent peer-to-peer file sharing protocol intends to mitigate poor latency and other congestion control problems found in conventional BitTorrent over TCP but still provide reliable, ordered delivery.

In this project, we are going to investigate into μTP's mechanism and also conduct experiments to evaluate how effective μTP can be, comparing to traditional BitTorrent file sharing protocol over TCP.

Through this experiment, we have learned about mechanism of μTP, more detail about how BitTorrent protocol works and also several tools which could help ease the process of experiment.

2 Experimental Methodology

First of all, we need to verify the problem of traditional BitTorrent file sharing problem over TCP, i.e. throttling of other non-BitTorrent TCP connections. For this purpose, we need to find a BitTorrent client which has not implement μTP. After some investigation, we decide to use `aria2`[1], which is well-documented and provides a easy to use command-line interface. This makes it a good choice to use on GENI virtual machines.

Then, to validate the effectiveness of μTP and compare with BitTorrent over TCP, we have to use another BitTorrent client which has implemented μTP. `qBittorrent` was our initial choice, however, it turns out to be difficult to use only under command-line interface. Thus, we choose `transmission`[2]. To be specific, its command-line variant `transmission-daemon` along with `transmission-remote` to control the daemon process.

2.1 Slice Topology

The topology of the GENI slice we use is shown in the following figure:

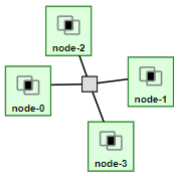

Figure 1: Topology

In this slice we have four nodes or hosts (we will use the term "node" and "host" interchangeably to refer to virtual machines in the network) connected by one link.

`node-0` and `node-1` are the seeding nodes. They possess the whole file in their local hard drive and transmit data to downloading nodes. `node-2` is the single downloading node. It will request file from the two seeding nodes using BitTorrent protocol. `node-3` is the testing node. It use `iperf` to establish a TCP connection with `node-2` and measure the throughput.

2.2 Limit Bandwidth

To get reasonable experiment result, we have to put some constraints on the bandwidth of network interfaces of these nodes.

First, to have enough time to operate, we have to either transmit a extremely large file, e.g. 20GB, or limit the speed of the BitTorrent client. We choose the later since it looks more flexible. To be specific, we limit the upload bandwidth of two seeding nodes to 1024KB/s.

Second, the BitTorrent client download bit rate should greater than 2/3 of the download bandwidth, otherwise, we may not observe the throttling of other TCP connections. In our case, there is only one other TCP connection established by `iperf` server and client. Assume upload bit rate is u, then the download bit rate d of the downloading node should be in the range of $(1u, 3u)$. Thus, we limit the download bandwidth of downloading node to 2048KB/s.

No constraints need to be put on the testing node.

2.3 Expected Results

Assume the download bandwidth is d, then when using BitTorrent over TCP, we expect the throughput measured by `iperf` be around $d/3$; when using μTP, we expect the throughput measured by `iperf` be at least $d/2$.

3 Results

3.1 Usage Instructions

Currently the four nodes are ready to conduct the experiment. We can login and follow the instructions below.

Since the configuration for the BitTorrent clients are local to users, we need to switch to user `xiaoduan` to execute the current commands correctly:

```
sudo su xiaoduan
```

3.1.1 Common Setup

1. Launch the Tracker

   Each BitTorrent client could communicate with each other with the help of a tracker service. The tracker is deployed on `node-0`. Before doing any of the two experiments, we have to launch the tracer first. On `node-0`, execute:

   ```
   # use another screen to run the tracker, it's like open another
   # instance of shell
   screen
   bttrack --port 6969 --dfile ~/.bttrack/dstate --logfile ~/.bttrack/tracker.log \
   --nat_check 0 --scrape_allowed full
   # The tracker launches and output nothing. Press Ctrl-A + d to switch
   # back to the original shell
   ```

2. Limit Bandwidth

   We also have to limit the bandwidth of `nodes-0`, `nodes-1` and `nodes-2` before any experiment.

   On `node-0` and `node-1` execute:

   ```
   # This limit the upload bandwidth to 1024KB/s
   cd ~
   sudo wondershaper/wondershaper -a eth1 -u 1024
   # To clear all constraints on an interface
   # sudo wondershaper/wondershaper -ca eth1
   ```

   On `node-2` execute:

   ```
   # This limit the download bandwidth to 2048KB/s
   cd ~
   sudo wondershaper/wondershaper -a eth1 -d 2048
   ```

3. Launch `iperf` Server on `node-2`

   On `node-2`:

   ```
   screen
   iperf -s
   # Use Ctrl-A + d to return to the original shell
   ```

3.1.2 Using `aria2` without μTP

Seeding on `node-0` and `node-1`:

```
cd ~
aria2c -V beethoven.torrent
```

Start downloading on `node-2`:

```
# make sure that the downloads directory is empty
cd ~
rm downloads/*
aria2c beethoven.torrent
```

Start `iperf` client on `node-3`:

```
# Set a longer time, so all three connections could reach stable state
# We may have to terminate it using Ctrl-C manually when the
# downloading finishes. Otherwise, the average throughput will count
# throuhput when there is no BitTorrent connections.
iperf -c 10.10.1.3 -t 180
```

We could stop one or both of the seeding processes to test the throughput when there is only 1 or 0 BitTorrent TCP connection.

3.1.3 Using `transmission` with μTP

Using `transmission` is more complicated, because we have to launch a daemon process and use another program to control it.

Make sure the tracker and `iperf` server are running and constraints are still on interfaces.

Seeding on `node-0` and `node-1`:

```
screen # run daemon in a new shell
transmission-daemon -f # force transmission-daemon in the foreground
# Use Ctrl-A + d to return to the original shell
cd ~;
transmission-remote -a beethoven.torrent
# show all torrents
# transmission-remote -l
# remove torrent with torrent-id, torrent-id is displayed in the
# output of transmission-remote -l
# transmission-remote -t torrent_id -r
```

One thing worth noting is that `transmission` remembers all the torrents once they are added, so if we have added a torrent once, we don't have to add it again. Also, if we want to download a torrent once again, we have to remove both the torrent from `transmission` and the local file (not the .torrent file).

Start downloading on `node-2`:

```
screen
transmission-daemon -f
# Ctrl-A + d return to original shell
cd ~;
rm downloads/*
# remove all torrents if necessary
# transmission-remote -t all -r
transmission-remote -a beethoven.torrent
# display downloading information, including speed
# transmission-remote -l
```

Start `iperf` client on `node-3`:

```
iperf -c 10.10.1.3 -t 180
```

We could test throughput when there is only 1 or 0 peer by closing one or both `transmission-daemon` processes on `node-0` and `node-1`.

3.1.4 Cleanup

Remember to stop all the `aria2c` processes, `transmission-daemon` processes, `screen` sessions, `iperf` server process and the tracker process. Also, clear all the constraints we put on the network interfaces.

3.2 Analysis

3.2.1 Using `aria2` without μTP

The output of `iperf` client detecting the throughput when there are two other BitTorrent TCP connections:

```
xiaoduan@node-3:~$ iperf -c 10.10.1.3 -t 180
------------------------------------------------------------
Client connecting to 10.10.1.3, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 10.10.1.4 port 37852 connected with 10.10.1.3 port 5001
^C[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-165.0 sec  13.6 MBytes   693 Kbits/sec
```

The output of `iperf` client detecting the throughput when there is one BitTorrent TCP connection:

```
xiaoduan@node-3:~$ iperf -c 10.10.1.3 -t 180
------------------------------------------------------------
Client connecting to 10.10.1.3, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 10.10.1.4 port 37854 connected with 10.10.1.3 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-181.1 sec  26.6 MBytes  1.23 Mbits/sec
```

The output of `iperf` client detecting the throughput when there is no other TCP connection:

```
xiaoduan@node-3:~$ iperf -c 10.10.1.3 -t 30
------------------------------------------------------------
Client connecting to 10.10.1.3, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 10.10.1.4 port 37856 connected with 10.10.1.3 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-30.9 sec  7.62 MBytes  2.07 Mbits/sec
```

The experiment result meets our expectation that when there are two BitTorrent TCP connections, the `iperf` TCP connection only have around 1/3 share of the total bandwidth.

3.2.2 Using `transmission` with μTP

The output of `iperf` client detecting the throughput when there is two peers connected:

```
xiaoduan@node-3:~$ iperf -c 10.10.1.3 -t 180
------------------------------------------------------------
Client connecting to 10.10.1.3, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 10.10.1.4 port 37862 connected with 10.10.1.3 port 5001
^C[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-83.0 sec  17.9 MBytes  1.81 Mbits/sec
```

When `iperf` is running, the output of `transmission-remote -l` on `node-2`:

```
xiaoduan@node-2:~$ transmission-remote -l
ID     Done       Have  ETA           Up    Down  Ratio  Status       Name
   2    94%   25.09 MB  12 sec       0.0    26.0    0.0  Downloading  Beethoven_Symphony_NO5_IV_Finale.mp3
Sum:          25.09 MB               0.0    26.0
```

From the result we can see that the μTP implementation of `transmission` reduces its own bit rate significantly when there is other TCP connections.

Here is also a figure 2 shows how the download speed changes. We launch `iperf` client for 30 seconds multiple time during the whole process. Every time the download speed drops drastically, we know that the `iperf` client is launched.

4 Conclusion

From our experiments, we confirm that BitTorrent protocol implemented over TCP does throttle other non-BitTorrent connections, especially when there are multiple BitTorrent TCP connections.

On the other hand, μTP implemented by `transimisson` over UDP is much less aggressive. In fact, it almost stop downloading entirely when there is other competing TCP connection. Anyway, we can see that μTP implemented over UDP

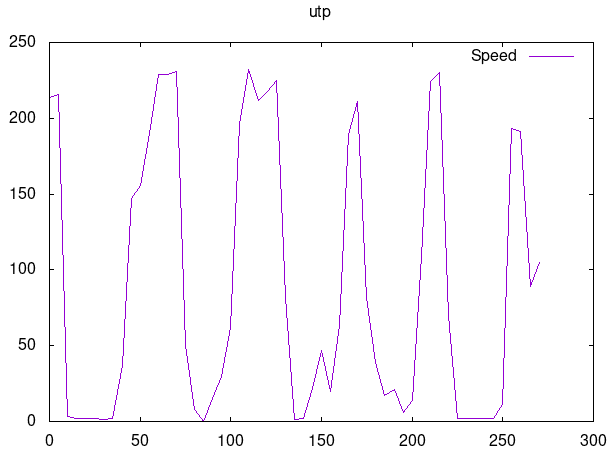

Figure 2: μTP

does address the issue while still provides reliable and ordered delivery. It does achieve the goal that BitTorrent clients using μTP do not disturb other internet connections while still utilizing the unused bandwidth fully. In our case, since `iperf` is a bandwidth measurement program and tries to take as much bandwidth as possible, the download speed of BitTorrent clients drops to almost 0.

To extend our experiment of μTP, we could also test its behavior when it works with other TCP connections which are not as aggressive as the `iperf` and see how it adjusts BitTorrent client's download speed instead of almost stopping at all.

5 Division of Labor

Xiaoduan Cheng: configured the environment, designed the experiment procedure, wrote the report.
Heze Yin: did the experiment, recorded the demo video, wrote the report.
Laixian Wan: did the experiment, wrote the report.
Peiqing Lu: did the experiment, wrote the report.

6 Reference

1. `aria2` https://aria2.github.io/
2. `transmission` https://transmissionbt.com/
3. Wikipedia https://en.wikipedia.org/wiki/Micro_Transport_Protocol
4. BEP-29 http://www.bittorrent.org/beps/bep_0029.html
5. RFC 6817 https://tools.ietf.org/html/rfc6817