Lab1: Reliable Data Transmission Under High Latency And High Packet Loss Rate Network

Due: **2016.6.5**
Start early! This lab is not easy!

Goal

In this lab, you need to implement a reliable data transmission protocol like TCP. This lab will emulate a high latency and high packet loss rate network environment, and your goal is to maximise the transfer speed under the network.

How this lab works

Overview

In short, the data is transferred with the following steps:

1. A program tries to open a **TCP** connection to `127.0.0.1:7000` , and sends **HTTP GET** request
2. Your program **Alice** listened on `127.0.0.1:7000` , accepted the connection, then send data to `172.19.0.1` . It's the IP address of a created TUN device **tunA**
3. The transfer program (provided by this lab) transfers your packet to **tunB**, and the destination IP is change to `172.20.0.1`
4. Your program **Bob** listened on `172.20.0.1` , get the request from **Alice**, then proxy the TCP connection to `127.0.0.1:8000`
5. There is a HTTP server (or whatever) accepts connection at `127.0.0.1:8000`, replies to the HTTP request.

Then, **Bob** should deliver the data from `172.20.0.1` to `172.20.0.2` on **tunB**. The transfer program transforms the packet, from `172.19.0.1` to `172.19.0.2`. **Alice** will get the packet on **tunA**, then she need to write data back to the origin TCP connection.

You need to implement the **Alice** and **Bob**. They should be the same program launched with different arguments:

```
./lab1 --alice
./lab1 --bob
```

How TUN works

TUN and **TAP** are virtual network kernel devices. It's a network interface emulated by software. The program could read and write packets on the interface.

In this lab, the transfer program will create two virtual interface **tunA** and **tunB**:

tunA: self ip `172.19.0.2`, peer ip `172.19.0.1`
tunB: self ip `172.20.0.1`, peer ip `172.20.0.2`

To explain how the transfer program works, you can try to ping `172.19.0.1`.

```
$ ping 172.19.0.1
PING 172.19.0.1 (172.19.0.1): 56 data bytes
64 bytes from 172.19.0.1: icmp_seq=0 ttl=64 time=0.603 ms
```

If you do a **tcpdump** on **tunA**, you'll get

```
$ sudo tcpdump -i tunA -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on tun8, link-type NULL (BSD loopback), capture size 262144
bytes
22:35:38.209997 IP 172.19.0.2 > 172.19.0.1: ICMP echo request, id 47671,
seq 0, length 64
22:35:38.210310 IP 172.19.0.1 > 172.19.0.2: ICMP echo reply, id 47671,
seq 0, length 64
```

And on **tunB**:

```
$ sudo tcpdump -i tun9  -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
```

```
listening on tun9, link-type NULL (BSD loopback), capture size 262144
bytes
22:35:38.210106 IP 172.20.0.2 > 172.20.0.1: ICMP echo request, id 47671,
seq 0, length 64
22:35:38.210183 IP 172.20.0.1 > 172.20.0.2: ICMP echo reply, id 47671,
seq 0, length 64
```

You can sort up the four packets by their send time:

```
172.19.0.2 > 172.19.0.1, ICMP request on tunA

< transfer from tunA to tunB >

172.20.0.2 > 172.20.0.1, ICMP request on tunB

< system replies to ICMP ping >

172.20.0.1 > 172.20.0.2, ICMP reply on tunB

< transfer from tunB to tunA >

172.19.0.1 > 172.19.0.2, ICMP reply on tunB
```

In this lab, **Alice** should communicate with **tunA** only, and **Bob** should communicate with **tunB** only. They are not allowed to communicate directly. Direct connection will be restricted on testing by TA. They can use any transfer layer protocol on **tunA** and **tunB** such as **TCP**, **UDP**, **ICMP**. The transfer program will emulate latency and packet loss on **tunA** and **tunB**.

Environment setting up

You can finish this lab on **Linux**, or **OSX**, or **Windows**. Just choose the platform you want.

First, fetch the material from:

```
https://git.oschina.net/nullmdr/network-pj1
```

Or you can

```
git clone https://git.oschina.net/nullmdr/network-pj1.git
```

This lab provides the transfer program with a **CMakeFiles**. It's a cross platform build script. I've built

the binary for you in `Release` Folder. However some extra steps are needed for some platforms.

Linux

Launch the binary with `sudo`:

```
$ sudo ./Release/transfer-linux -h
```

Mac OSX

You need to install **tuntaposx** from:

```
http://tuntaposx.sourceforge.net/download.xhtml
```

Then, launch the binary with `sudo`:

```
$ sudo ./Release/transfer-osx -t 10
```

If you have issue about root permission, see:

```
http://osxdaily.com/2015/10/05/disable-rootless-system-integrity-
protection-mac-os-x/
```

Windows

You need to install **tap-windows**. In `tap-win64` folder, run `addtap.bat`. Then, go to `Control Pannel - Network and Sharing Center - Change adapter settings`, rename the names of the two added **TAP-win32** adapter: rename **#2** to **tunA**, rename the other to **tunB**.

Then, launch the admin shell via `Release/windows-run.bat`. Launch **transfer** in the admin shell:

```
> transfer -l 100
```

Tips:

The **tap-windows** is a component of **OPENVPN**. You can download it (not required for this lab) on

```
https://openvpn.net/index.php/open-source/downloads.html
```

tap-windows install binary is at the bottom of the page. **Notice**: This lab requires two tun devices, so you need to manually install one in addiction after the installation if you want to install it from the **OPENVPN** website.

How to start this lab

Run download test

First, you need a HTTP server for speed test. You can use nginx. In fact you can use any http server you like. I suggest nginx with the following configuration

```
worker_processes  4;

events {
    worker_connections  1024;
}

http {
    include       mime.types;
    default_type  application/octet-stream;
    sendfile      on;
    keepalive_timeout  65;

    server {
        # Change it to 127.0.0.1:8000 after finishing this lab
        listen       8000;

        location / {
            # Change it to your test dir
            root   html;
        }

    }
}
```

Then, in the test dir, generate a 100M test file (Windows user can use **Cygwin** or **MinGW**)

```
$ dd if=/dev/urandom of=100M bs=1M count=100
```

For Mac OSX, you need to change `bs=1M` to `bs=1m`.

After **nginx** and **transfer** running up, you can test the transfer bridge

```
$ wget 172.19.0.1:8000/100M -O 100M.test
$ diff 100M 100M.test
```

After you finishing up this lab, you can test it in several ways

```
$ dd if=/dev/urandom of=1G bs=1M count=1024
$ wget 127.0.0.1:7000/1G -O 1G.test
$ diff 1G 1G.test

Test for multi thread downloading
$ axel -n 64 127.0.0.1:7000/1G -O 1G.test
$ mwget -n=128 127.0.0.1:7000/1G -O 1G.test
```

TCP Connection Proxy

This lab requires you to implement a TCP connection proxy in this lab. **Alice** should listen on 127.0.0.1:7000, and **Bob** should proxy all the connection to 127.0.0.1:8000. They communicate via tunA and tunB, or, **Alice** should only listen on 172.19.0.2 and send packet to 172.19.0.1, while **Bob** should only listen on 172.20.0.1 and send packet to 172.20.0.2. The address in the packet will be translated by **transfer**. Afterward **transfer** will send the transformed packet to the other tun interface. You can assume that for **Alice**, **Bob** is on 172.19.0.1; for **Bob**, **Alice** is on 172.20.0.2.

They *should* work correctly under multithread downloading. Use **diff** to check the correctness of data transfer.

TCP Congestion Control Algorithms

transfer can emulate a high latency, high packet loss rate network. As you can see, the system's default congestion control algorithm doesn't work well in this situation. **You need to implement a congestion control algorithm in this lab.** Maybe you can start up here:

```
https://en.wikipedia.org/wiki/TCP_congestion_control#Algorithms
```

You can try different algorithms, benchmark them, choose the best one, or implement one yourself.

Coding

This lab doesn't limit the platform or programming language you use. Please write down how to build your program in README. Add your test platform, compiler version and computer configuration to README, too.
Describe how do you implement this lab (e.g. which algorithm to use, why) in README.

MarkDown or PDF format is preferred for README. Please submit all of your source code and your README doc to ftp before due. Make a tarball or zip file like `13302010023_何天成.tar.gz` before submission.

Testing and Grading

30%

Sanity test. Your code should work correctly for single-thread and multi-thread downloading. Download speed should reach and keep at least 75% of the limit speed (option `-l`) with no delay and packet loss.

70%

Performance test. Your code will be tested under vary of configurations. Limit: delay will not exceed 300ms (rtt ~600ms), speed limit will not exceed 100Mbps, delay thrashing will not exceed 50%, packet loss rate: ???SECRET
The one wins the best performance will get full score in this part. The others' scores are based on their code's performance and the best performance of all.

How to build the transfer program

If you have trouble running the binary, or maybe you are interested in how to build the binary, here is the instruction:

Linux

Install **cmake**. If you are using **Ubuntu**:

```
$ sudo apt-get install cmake
```

Then, you can build the program

```
$ cmake .
$ make
```

Mac OSX

Install build toolchain:

```
$ xcode-select --install
```

Install **cmake** via **homebrew**:

```
$ /usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
$ brew install cmake
```

Then, build:

```
$ cmake .
$ make
```

Windows

Install **cmake**

```
https://cmake.org/
```

Install **Visual Studio 2015**

```
http://mvls.fudan.edu.cn/develope%20tools/SW_DVD9_NTRL_Visual_Studio_Ent
_2015_ChnSimp_FPP_VL_MLF_X20-29937.ISO
```

Build in **VS2015 MSBuild Command-line**

```
> cmake .
> msbuild ALL_BUILD.vcxproj /p:Configuration=Release
```