CSEE4119 Computer Networking HW2

Tong Wu, tw2906

Question 1

Part a

According to the question content, connection between client and the server uses non-persistent HTTP with a single connection TCP and will establish the connection via three-way handshake, the link is $3 \times 10^7\ m$ and connection link has bandwidth $10^6\ bits/sec$. Each data packet has $10^5\ bits$ long and each control or request message has $2000\ bits$ long.

For the **TCP handshake phase**, the total time of communication will be the sum of the time that:

1. client SYN upload: $\frac{2000\ bits}{1000000\ bits/sec} = 0.002\ sec$
2. client SYN transmit: $\frac{3\times10^7\ m}{3\times10^8\ m/s} = 0.1\ sec$
3. server download client SYN: $\frac{2000\ bits}{1000000\ bits/sec} = 0.002\ sec$, which is covered in transmit and upload time
4. server ACK upload: $\frac{2000\ bits}{1000000\ bits/sec} = 0.002\ sec$
5. server ACK transmit: $\frac{3\times10^7\ m}{3\times10^8\ m/s} = 0.1\ sec$
6. client download server ACK: $\frac{2000\ bits}{1000000\ bits/sec} = 0.002\ sec$, which is covered in transmit and upload time
7. client ack upload: $\frac{2000\ bits}{1000000\ bits/sec} = 0.002\ sec$
8. client ack transmit: $\frac{3\times10^7\ m}{3\times10^8\ m/s} = 0.1\ sec$
9. server download client ack: $\frac{2000\ bits}{1000000\ bits/sec} = 0.002\ sec$, which is covered in transmit and upload time

The total time will be:

$$0.1\ s \times 3 + 0.002 \times 3 = 0.306\ sec$$

For the **single object transmission phase**, the total time of communication will be the sum of the time that:

1. client request upload: $\frac{2000\ bits}{1000000\ bits/sec} = 0.002\ sec$
2. client request transmit: $\frac{3\times10^7\ m}{3\times10^8\ m/s} = 0.1\ sec$
3. server download client request: $\frac{2000\ bits}{1000000\ bits/sec} = 0.002\ sec$, which is covered in transmit and upload time
4. server DATA upload: $\frac{100000\ bits}{1000000\ bits/sec} = 0.1\ sec$
5. server DATA transmit: $\frac{3\times10^7\ m}{3\times10^8\ m/s} = 0.1\ sec$
6. client download server DATA: $\frac{100000\ bits}{1000000\ bits/sec} = 0.1\ sec$, which is covered in transmit and upload time

The total time will be:

$$0.1\ s \times 3 + 0.002 = 0.302\ sec$$

The whole 9 objects that client need to get include 1 initial page and 8 referenced objects, which have the same packet size. So the total process for the client to get 9 objects is 9 TCP handshake phase and 9 object transmission phase. The total time will be:

$$9 \times (0.306\ s + 0.302\ s) = 5.472\ sec$$

Part b

According to the question content, the connection now changes to enough parallel TCP connections to allow all requests. So the process to receive 9 objects are the same, while for the initial page it should be a single link and 8 referenced objects uses 8 parallel links.

According to the calculation in part a, for the TCP handshaking, the total time is $0.306\ s$

For the initial page getting, the total time is $0.302\ s$

So for the initial page transmission, the total time is $0.306\ s + 0.302\ s = 0.608\ sec$

For the referenced page getting, since the link is divided into 8 parallel connections each has $\frac{1}{8}$ of the total bandwidth, which should be $1000000\ bits/sec \times \frac{1}{8} = 125000\ bits/sec$. So according to the process, for the **TCP handshaking phase**, the processes are:

1. client SYN upload: $\frac{2000\ bits}{125000\ bits/sec} = 0.016\ sec$
2. client SYN transmit: $\frac{3\times10^7\ m}{3\times10^8\ m/s} = 0.1\ sec$
3. server download client SYN: $\frac{2000\ bits}{125000\ bits/sec} = 0.016\ sec$, which is covered in transmit and upload time
4. server ACK upload: $\frac{2000\ bits}{125000\ bits/sec} = 0.016\ sec$
5. server ACK transmit: $\frac{3\times10^7\ m}{3\times10^8\ m/s} = 0.1\ sec$
6. client download server ACK: $\frac{2000\ bits}{125000\ bits/sec} = 0.016\ sec$, which is covered in transmit and upload time
7. client ack upload: $\frac{2000\ bits}{125000\ bits/sec} = 0.016\ sec$
8. client ack transmit: $\frac{3\times10^7\ m}{3\times10^8\ m/s} = 0.1\ sec$
9. server download client ack: $\frac{2000\ bits}{125000\ bits/sec} = 0.016\ sec$, which is covered in transmit and upload time

The total time will be:

$$0.1\ s \times 3 + 0.016\ s \times 3 = 0.348\ sec$$

For the **parallel object transmission phase**, the total time of communication will be the sum of the time that:

1. client request upload: $\frac{2000\ bits}{125000\ bits/sec} = 0.016\ sec$

2. client request transmit: $\frac{3\times10^7\ m}{3\times10^8\ m/s} = 0.1\ sec$
3. server download client request: $\frac{2000\ bits}{125000\ bits/sec} = 0.016\ sec$, which is covered in transmit and upload time
4. server DATA upload: $\frac{100000\ bits}{125000\ bits/sec} = 0.8\ sec$
5. server DATA transmit: $\frac{3\times10^7\ m}{3\times10^8\ m/s} = 0.1\ sec$
6. client download server Data: $\frac{100000\ bits}{125000\ bits/sec} = 0.8\ sec$, which is covered in transmit and upload time

The total time will be:

$$0.1\ s \times 2 + 0.8\ s + 0.016 = 1.016\ sec$$

The whole 9 objects transmission will use total time:

$$0.608\ s + 0.348\ s + 1.016\ s = 1.972\ sec$$

Part c

For the persistent HTTP with a single TCP connection, the TCP handshaking will be proceed only once during the whole transmission. According to the calculation in part a, the TCP handshaking time for single TCP connection is $0.306\ sec$, and the object transmission time for single TCP connection is $0.302\ sec$. The total transmission time for 9 objects is then:

$$0.306 + 9 \times 0.302 = 3.024\ sec$$

Part d

The speedup factor for the persistent HTTP using single TCP connection against non-persistent HTTP using single TCP connection is:

$$\frac{5.472-3.024}{5.472} \approx 0.4474 = 44.74\%$$

Overall, the persistent HTTP does increase the speed of transmission since only one TCP handshaking is needed, while it cannot be said as significant because of the speedup factor for the parallel TCP connections will be larger, which should be $\frac{5.472-1.972}{5.472} \approx 0.6396 = 63.96\%$

Part e

If the link length changes to 3×10^{10}, which is 1000 times than previous, so the packet transmission time will become $100\ sec$. By roughly calculating, the gain will be approximately 660 since the transmission time in link is much greater than the uploading and downloading time, and the main gain is from object transmission phase, which is now need approximately $200\ sec$, so compared with the $0.302\ sec$, the gain is 660.

More specific calculation:

According to the calculation in part a. The total time of TCP handshaking phase will now be $100\ s \times 3 + 0.002 \times 3 = 300.006\ sec$, and the total time of object transmission phase will now be $100\ s \times 2 + 0.002 + 0.1 = 200.102\ sec$.

The total time for 9 object transmission will be:

$$300.006 + 9 \times 200.102 = 2100.924$$

The total gain is:

$$\frac{2100.924}{3.024} \approx 694.75$$

Part f

If the HTTP/2 Server Push is used, then the TCP handshaking is not change, still be $0.306\ sec$, and for the initial page transmission, the server will push initial page and the referenced objects to the client without client's further request. So the total time in this phase will be:

1. client request upload: $\frac{2000\ bits}{1000000\ bits/sec} = 0.002\ sec$
2. client request transmit: $\frac{3 \times 10^7\ m}{3 \times 10^8\ m/s} = 0.1\ sec$
3. server download client request: $\frac{2000\ bits}{1000000\ bits/sec} = 0.002\ sec$, which is covered in transmit and upload time
4. server DATA upload (1 initial page + 8 referenced objects): $\frac{900000\ bits}{1000000\ bits/sec} = 0.9\ sec$
5. server DATA transmit: $\frac{3 \times 10^7\ m}{3 \times 10^8\ m/s} = 0.1\ sec$
6. client download server DATA: $\frac{900000\ bits}{1000000\ bits/sec} = 0.9\ sec$, which is covered in transmit and upload time

$$0.002 + 0.1 \times 2 + 0.9 = 1.102\ sec$$

Total time of whole transmission will be:

$$0.306s + 1.102s = 1.408\ sec$$

Part g

For the e-commerce web page, it has 1 initial page and 8 referenced object, client use HTTP/2 server push with multiple parallel TCP connection. The total time of TCP handshaking establishment is not change, according to the calculation in part a which is $0.306\ sec$, and the time for transmission phase should be:

1. client request upload: $\frac{2000\ bits}{1000000\ bits/sec} = 0.002\ sec$
2. client request transmit: $\frac{3 \times 10^7\ m}{3 \times 10^8\ m/s} = 0.1\ sec$
3. server download client request: $\frac{2000\ bits}{1000000\ bits/sec} = 0.002\ sec$, which is covered in transmit and upload time
4. server DATA upload (1 initial page + 8 referenced objects): $\frac{900000\ bits}{1000000\ bits/sec} = 0.9\ sec$
5. server DATA transmit: $\frac{3 \times 10^7\ m}{3 \times 10^8\ m/s} = 0.1\ sec$
6. client download server DATA: $\frac{900000\ bits}{1000000\ bits/sec} = 0.9\ sec$, which is covered in transmit and upload time

The total time of whole transmission and TCP handshaking should be $1.408\ sec$

After that, the client needs extra $0.3\ sec$ to execute two JS script before to perform the image, so the total time for the images to appear on the user's screen should be:

$$1.408\ s + 0.3\ s = 1.708\ sec$$

Part h

According to the question content, the CSS file need to be loaded first before display contents, which should be put on the first place to transmit, also the initial HTML page also at the first place since it contains the text and referenced objects place. The external sync JS containing critical logic which need to be run before rendering, so it should be transmit on the second place. The visible image should be at the third place to transmit. Then for external async JS and 2 invisible images should be placed at the end. So the final prioritisation policy should be:

HTML = CSS > sync JS > 3 visible images > async JS = 2 invisible images

The TCP handshaking time is $0.306\ sec$, and for the transmission phase, the time until client download visible images is:

1. client request upload: $\frac{2000\ bits}{1000000\ bits/sec} = 0.002\ sec$
2. client request transmit: $\frac{3\times10^7\ m}{3\times10^8\ m/s} = 0.1\ sec$
3. server download client request: $\frac{2000\ bits}{1000000\ bits/sec} = 0.002\ sec$, which is covered in transmit and upload time
4. server DATA upload (initial HTML -> CSS -> sync JS -> 3 visible images): $\frac{600000\ bits}{1000000\ bits/sec} = 0.6\ sec$
5. server DATA transmit: $\frac{3\times10^7\ m}{3\times10^8\ m/s} = 0.1\ sec$
6. client download server DATA: $\frac{600000\ bits}{1000000\ bits/sec} = 0.6\ sec$, which is covered in transmit and upload time

It is worth to mention that after the client received sync JS, there are still 3 images need to be transmitted, which costs $0.3\ sec$ can just cover the parallel execution of sync JS, which is also $0.3\ sec$.

So the total time to show images on client's screen should be:

$$0.306\ s + 0.002\ s + 0.1\ s \times 2 + 0.6\ s = 1.108\ sec$$

Part i

The length of a HTTP Frame header is fixed at 9-octet (9 bytes) [1] [2].

Part j

In section 4 HTTP Frames with subsection 4.1 Frame Format [1].

Part k

For the HTTP/2, sender and receiver both sent WINDOW_UPDATE frame to each other to indicate what size of window that is allowed to receive [1] [2]. For some cases, the single HTTP link may has multiple hosts, e.g. proxy, each hosts has different flow control and both receiver and sender should follow it [2]. The flow control cannot be disabled [1], while the both sides can set WINDOW_UPDATE to the maximum ($2^{31} - 1$) in order to disable the flow control, while they still follow the maximum number [2].

Part l

The setting of the priority for a stream does not guarantee any transmission order for the stream, which means the priority is only a suggestion [1].

Question 2

Part a

Because:

1. UDP need not to establish a connection, since it may increase the delay.
2. The UDP has no congestion control, which allows UDP to deliver message as fast as possible.

Part b

A record

```
$ nslookup -type=a www.columbia.edu
Server:         127.0.0.53
Address:        127.0.0.53#53

Non-authoritative answer:
www.columbia.edu        canonical name = www.a.columbia.edu.
www.a.columbia.edu      canonical name = source.failover.cc.columbia.edu.
Name:   source.failover.cc.columbia.edu
Address: 128.59.105.24
```

AAAA record

```
$ nslookup -type=aaaa www.columbia.edu
Server:         127.0.0.53
Address:        127.0.0.53#53

Non-authoritative answer:
www.columbia.edu        canonical name = www.a.columbia.edu.
www.a.columbia.edu      canonical name = source.failover.cc.columbia.edu.
```

CNAME record

```
$ nslookup -type=cname www.columbia.edu
Server:         127.0.0.53
Address:        127.0.0.53#53

Non-authoritative answer:
www.columbia.edu        canonical name = www.a.columbia.edu.

Authoritative answers can be found from:
failover.cc.columbia.edu        nameserver = ns-1911.awsdns-46.co.uk.
failover.cc.columbia.edu        nameserver = ns-814.awsdns-37.net.
failover.cc.columbia.edu        nameserver = ns-1053.awsdns-03.org.
failover.cc.columbia.edu        nameserver = ns-89.awsdns-11.com.
source.failover.cc.columbia.edu internet address = 128.59.105.24
www.a.columbia.edu      canonical name = source.failover.cc.columbia.edu.
```

NS record

```
$ nslookup -type=ns columbia.edu
Server:         127.0.0.53
Address:        127.0.0.53#53

Non-authoritative answer:
columbia.edu    nameserver = auth5.dns.cogentco.com.
columbia.edu    nameserver = auth4.dns.cogentco.com.
columbia.edu    nameserver = auth1.dns.cogentco.com.
columbia.edu    nameserver = ext-ns1.columbia.edu.
columbia.edu    nameserver = dns2.itd.umich.edu.
columbia.edu    nameserver = auth2.dns.cogentco.com.
columbia.edu    nameserver = ns1.lse.ac.uk.

Authoritative answers can be found from:
ext-ns1.columbia.edu    has AAAA address 2001:18d8:1::1:1
ext-ns1.columbia.edu    has AAAA address 2001:18d8::35
ext-ns1.columbia.edu    internet address = 128.59.1.1
```

MX record

```
$ nslookup -type=mx columbia.edu
Server:         127.0.0.53
Address:        127.0.0.53#53

Non-authoritative answer:
columbia.edu    mail exchanger = 10 mxa-00364e01.gslb.pphosted.com.
columbia.edu    mail exchanger = 10 mxb-00364e01.gslb.pphosted.com.

Authoritative answers can be found from:
columbia.edu    nameserver = auth2.dns.cogentco.com.
columbia.edu    nameserver = ns1.lse.ac.uk.
columbia.edu    nameserver = ext-ns1.columbia.edu.
columbia.edu    nameserver = auth4.dns.cogentco.com.
columbia.edu    nameserver = auth5.dns.cogentco.com.
columbia.edu    nameserver = dns2.itd.umich.edu.
columbia.edu    nameserver = auth1.dns.cogentco.com.
ext-ns1.columbia.edu    has AAAA address 2001:18d8:1::1:1
ext-ns1.columbia.edu    has AAAA address 2001:18d8::35
ext-ns1.columbia.edu    internet address = 128.59.1.1
```

Part c

The AAAA records in DNS system can map a domain name to an IPv6 address [3]. According to the nslookup command used in part a, when querying AAAA records for domain name 'www.columbia.edu', the output shows no IPv6 address is mapped to the domain. So the domain name 'www.columbia.edu' does not support IPv6.

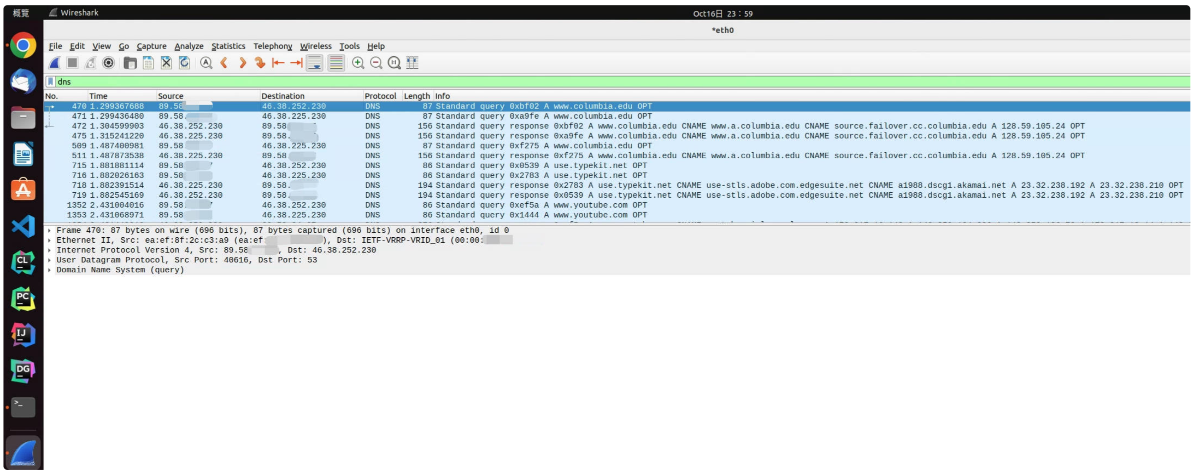
Part d

By using the nsloopup command to find the DNS resolver:

```
$ nslookup .
Server:      127.0.0.53
Address:     127.0.0.53#53
```

Which is a local IP address.

Use wireshark to capture the packet sent to the server when the broswer is entering 'www.columbia.edu'.


```
*eth0
dns
No.  Time        Source           Destination      Protocol Length Info
 470 1.299367688 89.58[redacted]  46.38.252.230    DNS       87 Standard query 0xbf02 A www.columbia.edu OPT
 471 1.299436480 89.58.[redacted] 46.38.225.230    DNS       87 Standard query 0xa9fe A www.columbia.edu OPT
 472 1.304599903 46.38.252.230    89.58[redacted]  DNS      156 Standard query response 0xbf02 A www.columbia.edu CNAME www.a.columbia.edu CNAME source.failover.cc.columbia.edu A 128.59.105.24 OPT
 475 1.315241220 46.38.225.230    89.58.[redacted] DNS      156 Standard query response 0xa9fe A www.columbia.edu CNAME www.a.columbia.edu CNAME source.failover.cc.columbia.edu A 128.59.105.24 OPT
 509 1.487400981 89.58[redacted]  46.38.225.230    DNS       87 Standard query 0xf275 A www.columbia.edu OPT
 511 1.487873538 46.38.225.230    89.58[redacted]  DNS      156 Standard query response 0xf275 A www.columbia.edu CNAME www.a.columbia.edu CNAME source.failover.cc.columbia.edu A 128.59.105.24 OPT
 715 1.881881114 89.58[redacted]  46.38.252.230    DNS       86 Standard query 0x0539 A use.typekit.net OPT
 716 1.882026163 89.58[redacted]  46.38.225.230    DNS       86 Standard query 0x2783 A use.typekit.net OPT
 718 1.882391514 46.38.225.230    89.58.[redacted] DNS      194 Standard query response 0x2783 A use.typekit.net CNAME use-stls.adobe.com.edgesuite.net CNAME a1988.dscg1.akamai.net A 23.32.238.192 A 23.32.238.210 OPT
 719 1.882545169 46.38.252.230    89.58.[redacted] DNS      194 Standard query response 0x0539 A use.typekit.net CNAME use-stls.adobe.com.edgesuite.net CNAME a1988.dscg1.akamai.net A 23.32.238.192 A 23.32.238.210 OPT
1352 2.431004016 89.58.[redacted] 46.38.252.230    DNS       86 Standard query 0xef5a A www.youtube.com OPT
1353 2.431068971 89.58[redacted]  46.38.225.230    DNS       86 Standard query 0x1444 A www.youtube.com OPT

› Frame 470: 87 bytes on wire (696 bits), 87 bytes captured (696 bits) on interface eth0, id 0
› Ethernet II, Src: ea:ef:8f:2c:c3:a9 (ea:ef[redacted]), Dst: IETF-VRRP-VRID_01 (00:00:[redacted]
› Internet Protocol Version 4, Src: 89.58[redacted], Dst: 46.38.252.230
› User Datagram Protocol, Src Port: 40616, Dst Port: 53
› Domain Name System (query)
```
Wireshark
Oct16日 23 : 59
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

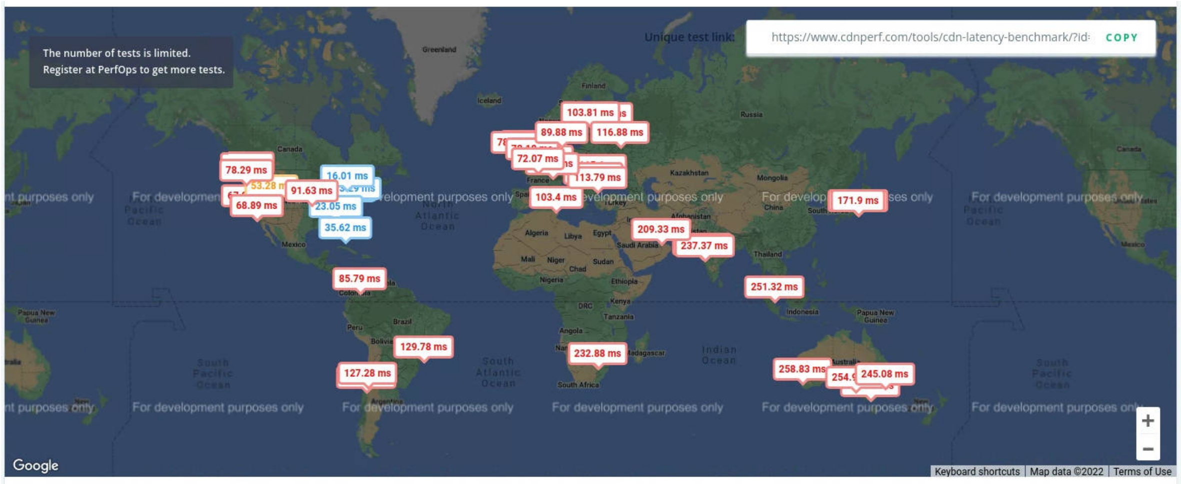
```
470 1.299367688 89.58.*.*    46.38.252.230    DNS 87   Standard query 0xbf02 A
www.columbia.edu OPT
```

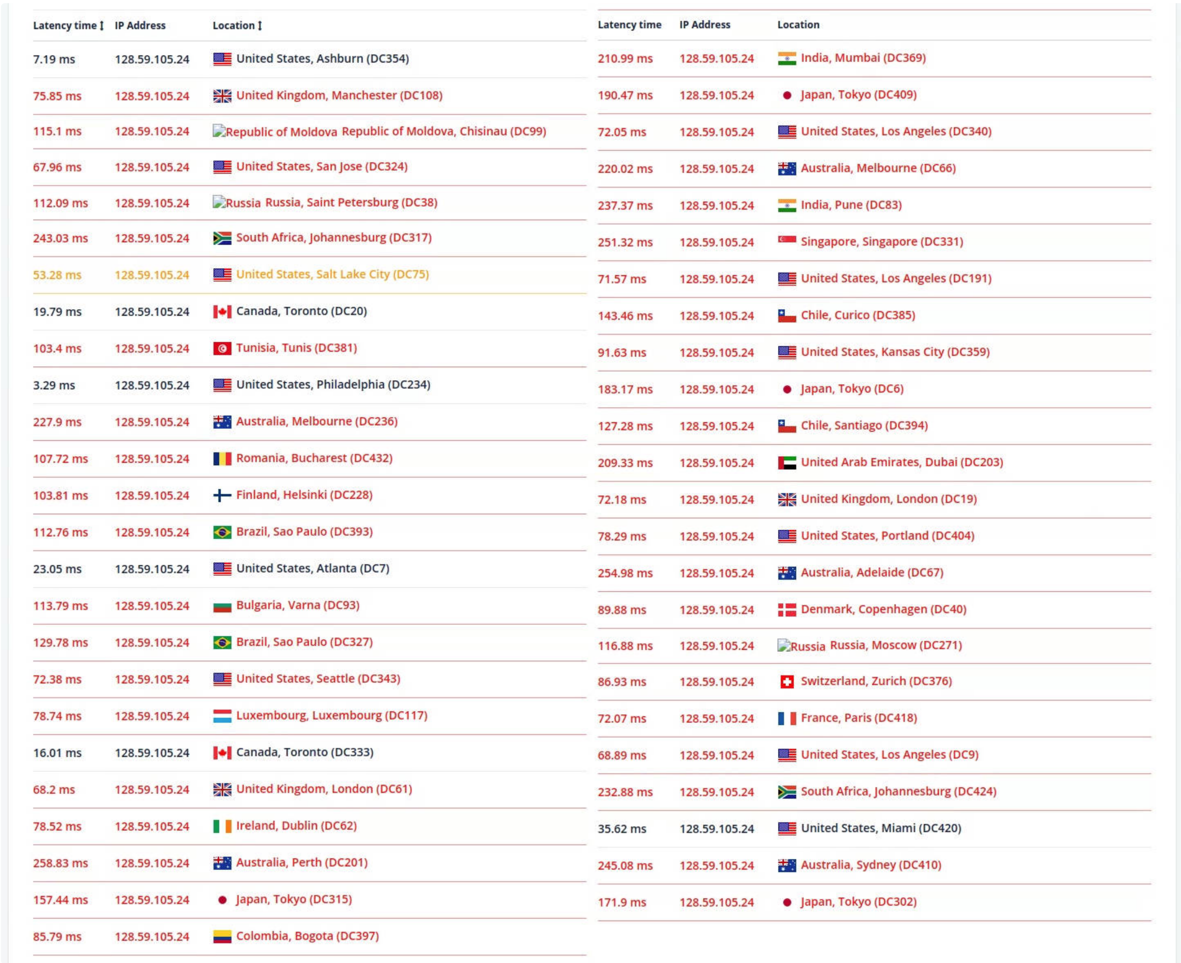
Filter to the DNS protocol and can found that the local machine with public IP 89.58.*.* sent query to the DNS resolver with IP address 46.38.252.230, which can be found is a local DNS server maintained by netcup.

```
$ nslookup 46.38.252.230
230.252.38.46.in-addr.arpa      name = dns2.dnsserver2.de.

Authoritative answers can be found from:
252.38.46.in-addr.arpa  nameserver = second-dns.netcup.net.
252.38.46.in-addr.arpa  nameserver = third-dns.netcup.net.
252.38.46.in-addr.arpa  nameserver = root-dns.netcup.net.
```

Part e

By using CDNPerf to ping 'www.columbia.edu', the result is attached as following:



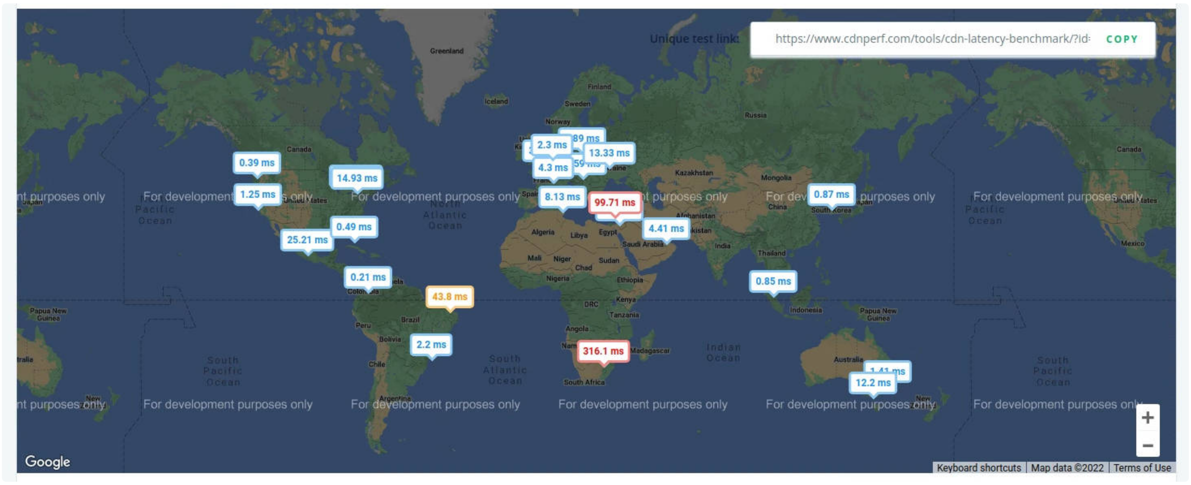


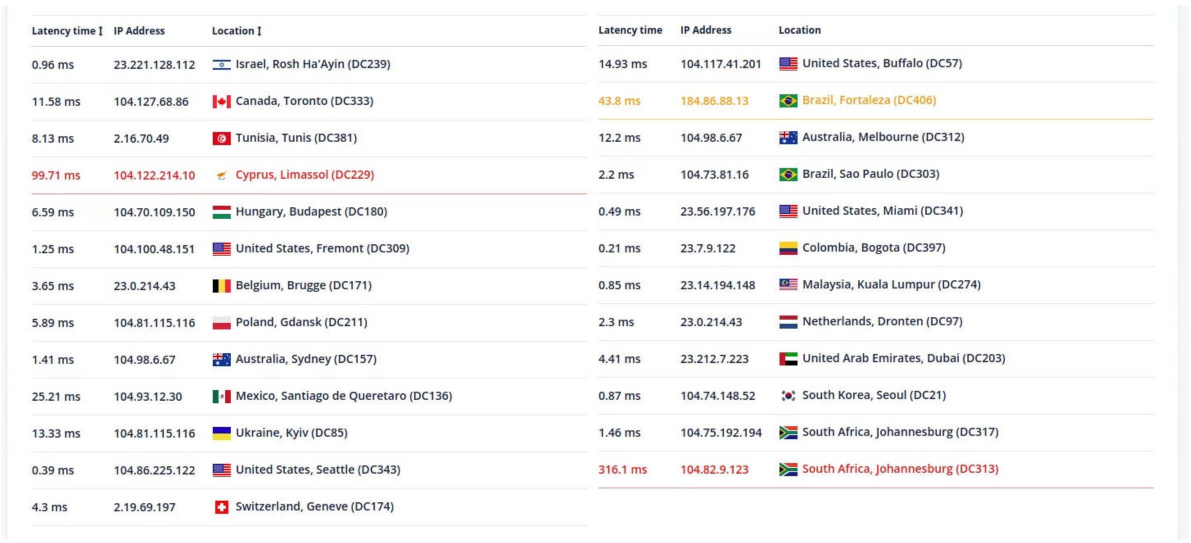
Latency time	IP Address	Location
7.19 ms	128.59.105.24	United States, Ashburn (DC354)
75.85 ms	128.59.105.24	United Kingdom, Manchester (DC108)
115.1 ms	128.59.105.24	Republic of Moldova Republic of Moldova, Chisinau (DC99)
67.96 ms	128.59.105.24	United States, San Jose (DC324)
112.09 ms	128.59.105.24	Russia Russia, Saint Petersburg (DC38)
243.03 ms	128.59.105.24	South Africa, Johannesburg (DC317)
53.28 ms	128.59.105.24	United States, Salt Lake City (DC75)
19.79 ms	128.59.105.24	Canada, Toronto (DC20)
103.4 ms	128.59.105.24	Tunisia, Tunis (DC381)
3.29 ms	128.59.105.24	United States, Philadelphia (DC234)
227.9 ms	128.59.105.24	Australia, Melbourne (DC236)
107.72 ms	128.59.105.24	Romania, Bucharest (DC432)
103.81 ms	128.59.105.24	Finland, Helsinki (DC228)
112.76 ms	128.59.105.24	Brazil, Sao Paulo (DC393)
23.05 ms	128.59.105.24	United States, Atlanta (DC7)
113.79 ms	128.59.105.24	Bulgaria, Varna (DC93)
129.78 ms	128.59.105.24	Brazil, Sao Paulo (DC327)
72.38 ms	128.59.105.24	United States, Seattle (DC343)
78.74 ms	128.59.105.24	Luxembourg, Luxembourg (DC117)
16.01 ms	128.59.105.24	Canada, Toronto (DC333)
68.2 ms	128.59.105.24	United Kingdom, London (DC61)
78.52 ms	128.59.105.24	Ireland, Dublin (DC62)
258.83 ms	128.59.105.24	Australia, Perth (DC201)
157.44 ms	128.59.105.24	Japan, Tokyo (DC315)
85.79 ms	128.59.105.24	Colombia, Bogota (DC397)
210.99 ms	128.59.105.24	India, Mumbai (DC369)
190.47 ms	128.59.105.24	Japan, Tokyo (DC409)
72.05 ms	128.59.105.24	United States, Los Angeles (DC340)
220.02 ms	128.59.105.24	Australia, Melbourne (DC66)
237.37 ms	128.59.105.24	India, Pune (DC83)
251.32 ms	128.59.105.24	Singapore, Singapore (DC331)
71.57 ms	128.59.105.24	United States, Los Angeles (DC191)
143.46 ms	128.59.105.24	Chile, Curico (DC385)
91.63 ms	128.59.105.24	United States, Kansas City (DC359)
183.17 ms	128.59.105.24	Japan, Tokyo (DC6)
127.28 ms	128.59.105.24	Chile, Santiago (DC394)
209.33 ms	128.59.105.24	United Arab Emirates, Dubai (DC203)
72.18 ms	128.59.105.24	United Kingdom, London (DC19)
78.29 ms	128.59.105.24	United States, Portland (DC404)
254.98 ms	128.59.105.24	Australia, Adelaide (DC67)
89.88 ms	128.59.105.24	Denmark, Copenhagen (DC40)
116.88 ms	128.59.105.24	Russia Russia, Moscow (DC271)
86.93 ms	128.59.105.24	Switzerland, Zurich (DC376)
72.07 ms	128.59.105.24	France, Paris (DC418)
68.89 ms	128.59.105.24	United States, Los Angeles (DC9)
232.88 ms	128.59.105.24	South Africa, Johannesburg (DC424)
35.62 ms	128.59.105.24	United States, Miami (DC420)
245.08 ms	128.59.105.24	Australia, Sydney (DC410)
171.9 ms	128.59.105.24	Japan, Tokyo (DC302)

From the result can found that the east coast has latency around 25ms, and west coast has around 70ms. By measuring the rough distance from east to west coast, which is 4073km, and assume the light speed in fibre which is $2 \times 10^8 \, m/s$ [4]. Can calculate that the optimal latency is $\frac{4073 \times 1000}{2 \times 10^8} \approx 0.0204 \, sec = 20.4 \, ms$. So it can be summarised that the direct latency should be 20.4ms. By adding extra condition such as several hups and extra latency made by router and possible proxy, the 70ms latency can be used as evidence that users on the west coast are accessing the same physical servers as users on the east coast.

Part f

By using CDNPerf to ping ‘www.akamai.com’, the result is attached as following:



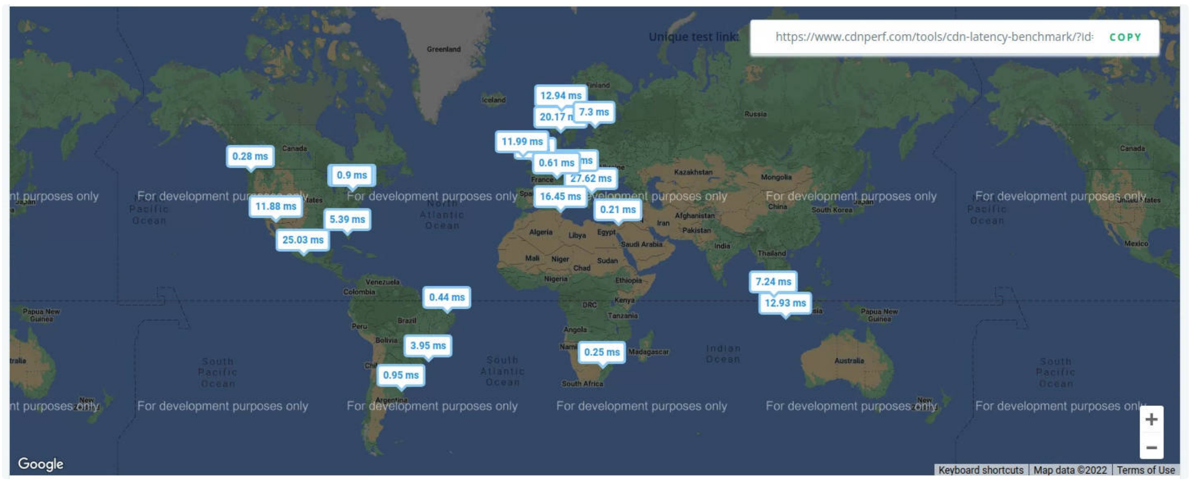


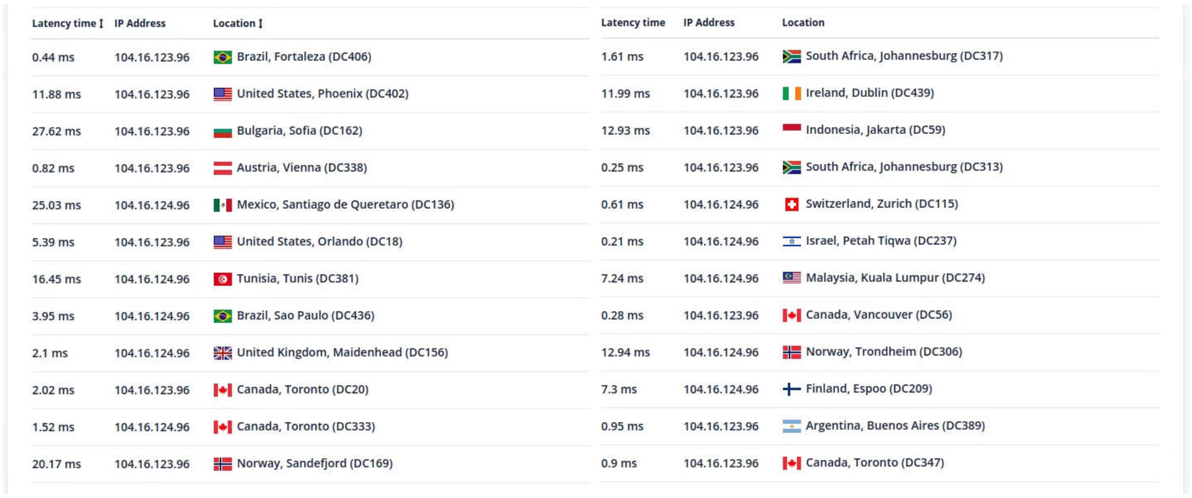
Latency time	IP Address	Location
0.96 ms	23.221.128.112	Israel, Rosh Ha'Ayin (DC239)
11.58 ms	104.127.68.86	Canada, Toronto (DC333)
8.13 ms	2.16.70.49	Tunisia, Tunis (DC381)
99.71 ms	104.122.214.10	Cyprus, Limassol (DC229)
6.59 ms	104.70.109.150	Hungary, Budapest (DC180)
1.25 ms	104.100.48.151	United States, Fremont (DC309)
3.65 ms	23.0.214.43	Belgium, Brugge (DC171)
5.89 ms	104.81.115.116	Poland, Gdansk (DC211)
1.41 ms	104.98.6.67	Australia, Sydney (DC157)
25.21 ms	104.93.12.30	Mexico, Santiago de Queretaro (DC136)
13.33 ms	104.81.115.116	Ukraine, Kyiv (DC85)
0.39 ms	104.86.225.122	United States, Seattle (DC343)
4.3 ms	2.19.69.197	Switzerland, Geneve (DC174)
14.93 ms	104.117.41.201	United States, Buffalo (DC57)
43.8 ms	184.86.88.13	Brazil, Fortaleza (DC406)
12.2 ms	104.98.6.67	Australia, Melbourne (DC312)
2.2 ms	104.73.81.16	Brazil, Sao Paulo (DC303)
0.49 ms	23.56.197.176	United States, Miami (DC341)
0.21 ms	23.7.9.122	Colombia, Bogota (DC397)
0.85 ms	23.14.194.148	Malaysia, Kuala Lumpur (DC274)
2.3 ms	23.0.214.43	Netherlands, Dronten (DC97)
4.41 ms	23.212.7.223	United Arab Emirates, Dubai (DC203)
0.87 ms	104.74.148.52	South Korea, Seoul (DC21)
1.46 ms	104.75.192.194	South Africa, Johannesburg (DC317)
316.1 ms	104.82.9.123	South Africa, Johannesburg (DC313)

Where the latency from east and west coast are both around 1ms. It can summarise that user will visit different physical server when visiting ‘www.akamai.com’. Akamai uses DNS redirection CDN since the IP addresses that different node visit are different. DNS redirection CDN can let each server has own IP address and use CDN’s DNS to give user a IP address that is the closest server, which can react quickly.

Part g

By using CDNPerf to ping ‘www.cloudflare.com’, the result is attached as following:





Latency time	IP Address	Location	Latency time	IP Address	Location
0.44 ms	104.16.123.96	Brazil, Fortaleza (DC406)	1.61 ms	104.16.123.96	South Africa, Johannesburg (DC317)
11.88 ms	104.16.123.96	United States, Phoenix (DC402)	11.99 ms	104.16.123.96	Ireland, Dublin (DC439)
27.62 ms	104.16.123.96	Bulgaria, Sofia (DC162)	12.93 ms	104.16.123.96	Indonesia, Jakarta (DC59)
0.82 ms	104.16.123.96	Austria, Vienna (DC338)	0.25 ms	104.16.123.96	South Africa, Johannesburg (DC313)
25.03 ms	104.16.124.96	Mexico, Santiago de Queretaro (DC136)	0.61 ms	104.16.124.96	Switzerland, Zurich (DC115)
5.39 ms	104.16.123.96	United States, Orlando (DC18)	0.21 ms	104.16.124.96	Israel, Petah Tiqwa (DC237)
16.45 ms	104.16.124.96	Tunisia, Tunis (DC381)	7.24 ms	104.16.124.96	Malaysia, Kuala Lumpur (DC274)
3.95 ms	104.16.124.96	Brazil, Sao Paulo (DC436)	0.28 ms	104.16.123.96	Canada, Vancouver (DC56)
2.1 ms	104.16.124.96	United Kingdom, Maidenhead (DC156)	12.94 ms	104.16.124.96	Norway, Trondheim (DC306)
2.02 ms	104.16.123.96	Canada, Toronto (DC20)	7.3 ms	104.16.124.96	Finland, Espoo (DC209)
1.52 ms	104.16.124.96	Canada, Toronto (DC333)	0.95 ms	104.16.123.96	Argentina, Buenos Aires (DC389)
20.17 ms	104.16.123.96	Norway, Sandefjord (DC169)	0.9 ms	104.16.123.96	Canada, Toronto (DC347)

Where the latency from east and west coast are both around 1ms. It can summarise that user will visit different physical server when visiting ‘www.cloudflare.com’. Cloudflare uses Anycast CDN since the IP addresses that different node visit are the same. Anycast CDN will choose physical server by using BGP routing, which is simple to deploy and can defence DDOS.

Part h

For the content delivery approach in part e, which is single physical server without CDN. The advantage is costing less. The disadvantage is bringing a bad user experience for users far away to the physical server.

For the content delivery approach in part f, which is DNS redirection CDN. The advantage is that it can react quickly when the load of traffic is changing. The disadvantage is that it is not easy to operate.

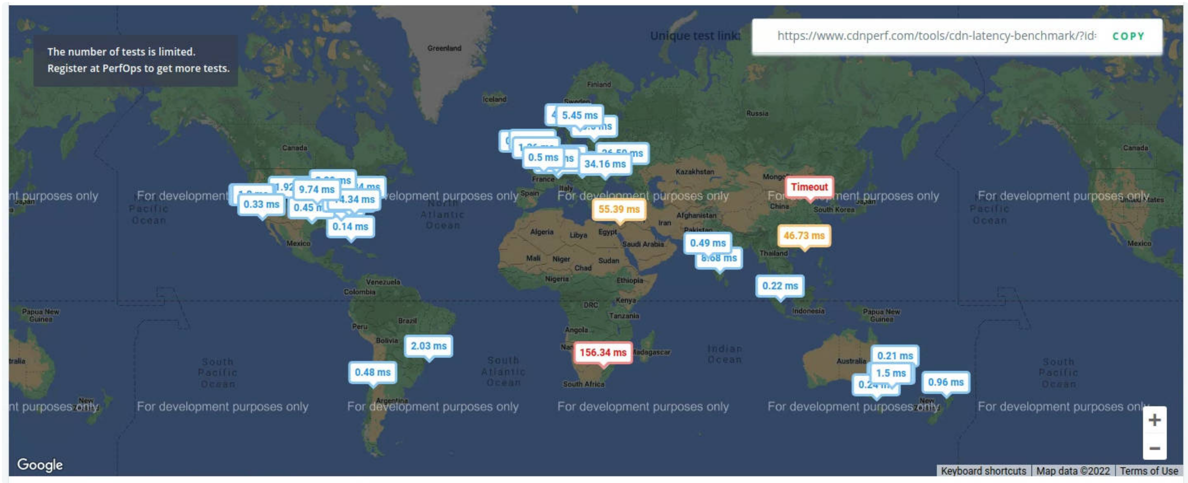
For the content delivery approach in part f, which is Anycast CDN. The advantage is that it can defence DDOS attack naturally. The disadvantage is that it may not be able to choose the fastest server.

Part i

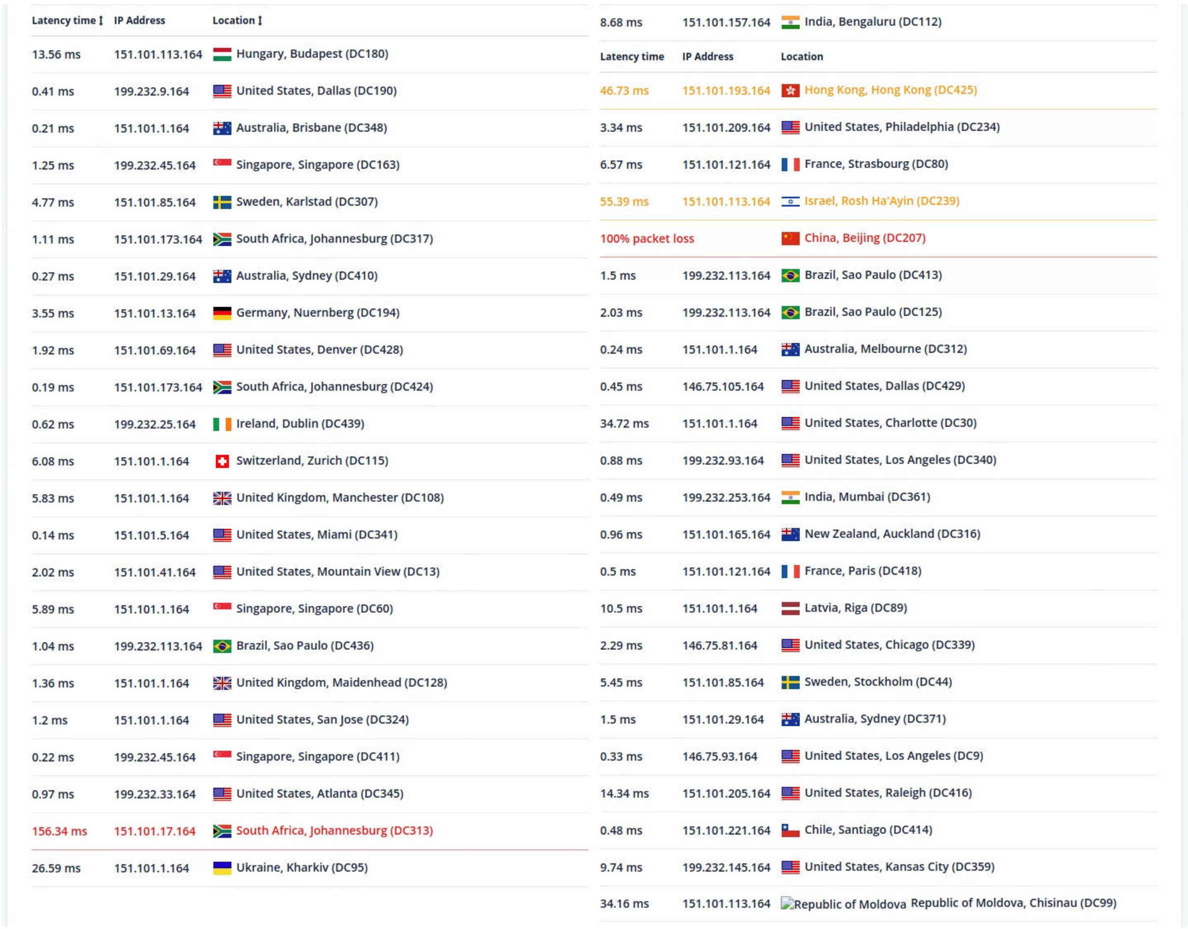
Check name servers that are used by New York Times by nslookup:

```
$ nslookup -type=ns www.nytimes.com
Server:         127.0.0.53
Address:        127.0.0.53#53

Non-authoritative answer:
www.nytimes.com canonical name = www.prd.map.nytimes.com.
www.prd.map.nytimes.com canonical name = www.prd.map.nytimes.xovr.nyt.net.
www.prd.map.nytimes.xovr.nyt.net        canonical name =
nytimes.map.fastly.net.

Authoritative answers can be found from:
fastly.net      nameserver = ns2.fastly.net.
fastly.net      nameserver = ns4.fastly.net.
fastly.net      nameserver = ns1.fastly.net.
fastly.net      nameserver = ns3.fastly.net.
fastly.net
        origin = ns1.fastly.net
        mail addr = hostmaster.fastly.com
        serial = 2017052201
        refresh = 3600
        retry = 600
        expire = 604800
        minimum = 30
nytimes.map.fastly.net  internet address = 151.101.209.164
```

Can found that name servers are held by Fastly. Use CDNPerf to check latency accessed from the world:


The number of tests is limited.
Register at PerfOps to get more tests.

Unique test link: https://www.cdnperf.com/tools/cdn-latency-benchmark/?id



Latency time	IP Address	Location
13.56 ms	151.101.113.164	Hungary, Budapest (DC180)
0.41 ms	199.232.9.164	United States, Dallas (DC190)
0.21 ms	151.101.1.164	Australia, Brisbane (DC348)
1.25 ms	199.232.45.164	Singapore, Singapore (DC163)
4.77 ms	151.101.85.164	Sweden, Karlstad (DC307)
1.11 ms	151.101.173.164	South Africa, Johannesburg (DC317)
0.27 ms	151.101.29.164	Australia, Sydney (DC410)
3.55 ms	151.101.13.164	Germany, Nuernberg (DC194)
1.92 ms	151.101.69.164	United States, Denver (DC428)
0.19 ms	151.101.173.164	South Africa, Johannesburg (DC424)
0.62 ms	199.232.25.164	Ireland, Dublin (DC439)
6.08 ms	151.101.1.164	Switzerland, Zurich (DC115)
5.83 ms	151.101.1.164	United Kingdom, Manchester (DC108)
0.14 ms	151.101.5.164	United States, Miami (DC341)
2.02 ms	151.101.41.164	United States, Mountain View (DC13)
5.89 ms	151.101.1.164	Singapore, Singapore (DC60)
1.04 ms	199.232.113.164	Brazil, Sao Paulo (DC436)
1.36 ms	151.101.1.164	United Kingdom, Maidenhead (DC128)
1.2 ms	151.101.1.164	United States, San Jose (DC324)
0.22 ms	199.232.45.164	Singapore, Singapore (DC411)
0.97 ms	199.232.33.164	United States, Atlanta (DC345)
156.34 ms	151.101.17.164	South Africa, Johannesburg (DC313)
26.59 ms	151.101.1.164	Ukraine, Kharkiv (DC95)
8.68 ms	151.101.157.164	India, Bengaluru (DC112)
46.73 ms	151.101.193.164	Hong Kong, Hong Kong (DC425)
3.34 ms	151.101.209.164	United States, Philadelphia (DC234)
6.57 ms	151.101.121.164	France, Strasbourg (DC80)
55.39 ms	151.101.113.164	Israel, Rosh Ha'Ayin (DC239)
100% packet loss		China, Beijing (DC207)
1.5 ms	199.232.113.164	Brazil, Sao Paulo (DC413)
2.03 ms	199.232.113.164	Brazil, Sao Paulo (DC125)
0.24 ms	151.101.1.164	Australia, Melbourne (DC312)
0.45 ms	146.75.105.164	United States, Dallas (DC429)
34.72 ms	151.101.1.164	United States, Charlotte (DC30)
0.88 ms	199.232.93.164	United States, Los Angeles (DC340)
0.49 ms	199.232.253.164	India, Mumbai (DC361)
0.96 ms	151.101.165.164	New Zealand, Auckland (DC316)
0.5 ms	151.101.121.164	France, Paris (DC418)
10.5 ms	151.101.1.164	Latvia, Riga (DC89)
2.29 ms	146.75.81.164	United States, Chicago (DC339)
5.45 ms	151.101.85.164	Sweden, Stockholm (DC44)
1.5 ms	151.101.29.164	Australia, Sydney (DC371)
0.33 ms	146.75.93.164	United States, Los Angeles (DC9)
14.34 ms	151.101.205.164	United States, Raleigh (DC416)
0.48 ms	151.101.221.164	Chile, Santiago (DC414)
9.74 ms	199.232.145.164	United States, Kansas City (DC359)
34.16 ms	151.101.113.164	Republic of Moldova Republic of Moldova, Chisinau (DC99)

Can found that New York Time website may use DNS redirection CDN, since the IP address that different node visit are different.

Part j

Section i

For the domain name ‘ipv4-c001-lga001-nysernet-isp.1.oca.nflxvideo.net’, the mapped IP address can be found by using nslookup command to find the A records and AAAA records which map the IPv4 and IPv6 addresses:

```
$ nslookup -type=a ipv4-c001-lga001-nysernet-isp.1.oca.nflxvideo.net
Server:         127.0.0.53
Address:        127.0.0.53#53

Non-authoritative answer:
Name:   ipv4-c001-lga001-nysernet-isp.1.oca.nflxvideo.net
Address: 199.109.94.18
```

```
$ nslookup -type=aaaa ipv4-c001-lga001-nysernet-isp.1.oca.nflxvideo.net
Server:         127.0.0.53
Address:        127.0.0.53#53

Non-authoritative answer:
*** Can't find ipv4-c001-lga001-nysernet-isp.1.oca.nflxvideo.net: No answer
```

Can found that this domain name has one IPv4 address ‘199.109.94.18’ and no IPv6 address.

Section ii

For the IP address ‘199.109.94.18’, using nslookup command to find PTR address:

```
$ nslookup 199.109.94.18
18.94.109.199.in-addr.arpa      name = netflix-c001.lga001.nysernet.net.
```

Where can found that the PTR address is ‘18.94.109.199.in-addr.arpa’, and using Wireshark dig command to find full PTR record:

```
$ dig @1.1.1.1 18.94.109.199.in-addr.arpa. PTR

; <<>> DiG 9.18.1-1ubuntu1.2-Ubuntu <<>> @1.1.1.1 18.94.109.199.in-addr.arpa. PTR
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 31259
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
;; QUESTION SECTION:
;18.94.109.199.in-addr.arpa.    IN      PTR

```

```
;; ANSWER SECTION:
18.94.109.199.in-addr.arpa. 86400 IN    PTR     netflix-
c001.lga001.nysernet.net.

;; Query time: 428 msec
;; SERVER: 1.1.1.1#53(1.1.1.1) (UDP)
;; WHEN: Mon Oct 17 13:56:49 EDT 2022
;; MSG SIZE  rcvd: 101
```

The value of PTR record is ‘netflix-c001.lga001.nysernet.net’

Section iii

The PTR (pointer record) provide a mapping from IP address to domain name, rather than A record is mapping from domain name to IP address [5]. PTR will be useful when an IP address is mapped to several domain name. From the information in section ii, the value of PTR address is not same as ‘ipv4-c001-lga001-nysernet-isp.1.oca.nflxvideo.net’. Using dig command for the PTR value ‘netflix-c001.lga001.nysernet.net’:

```
$ dig @8.8.4.4 netflix-c001.lga001.nysernet.net. PTR

; <<>> DiG 9.18.1-1ubuntu1.2-Ubuntu <<>> @8.8.4.4 netflix-
c001.lga001.nysernet.net. PTR
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 3956
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;netflix-c001.lga001.nysernet.net. IN   PTR

;; AUTHORITY SECTION:
nysernet.net.           1800    IN      SOA     dns01.nysernet.org.
noc.nysernet.org. 2022052500 86400 14400 864000 86400

;; Query time: 292 msec
;; SERVER: 8.8.4.4#53(8.8.4.4) (UDP)
;; WHEN: Mon Oct 17 14:22:12 EDT 2022
;; MSG SIZE  rcvd: 119
```

Which shows that it is come from nysernet, which is an non-profit ISP in NY [6]. The reason may that, the IP address is not owned by Netflix but rent from the ISP, and the PTR record is managed by ISP but not the domain owner [6], so the PTR record for Netflix’s domain has different value.

Section iv

As mentioned in section iii, can conclude that the domain name ‘ipv4-c001-lga001-nysernet-isp.1.oca.nflxvideo.net’ is located on a off-net server which is included in Netflix Open Connect program, since the PTR record is not match with the domain address.

Question 3

Part a

For a 1.5 hour (90 minutes = 5400 sec) video file, low quality has file size 70 MB. The bitrate can be calculated by

$$\frac{file\ \ size(bit)}{time(sec)} = bitrate\ \ (bit/sec)$$

For a file size with 70MB, the file size into megabits (Mb) should be $70 \times 8 = 560Mb$, then the bitrate can be calculated by:

$$\frac{560 \times 10^6\ bit}{5400\ sec} \approx 103703.7\ bps = 103.7037\ Kbps$$

Where the low-quality video has bitrate $103.7307\ bps$

Part b

For high-quality video with 2 GB file size, which transform to terabits should be $2 \times 8 = 16\ Gb$

According to the equation in part a, the bitrate can be calculated by:

$$\frac{16 \times 10^9\ bit}{5400\,\mathrm{sec}} \approx 2962963\ bps = 2.963\ Mbps$$

Part c

DASH will choose low-quality when the current access rate is $256\ KBps = 2.048Mbps$. Since the access rate is not sufficient for the high-quality video, in order to provide best experience the DASH will choose low-quality.

Part d

If a user is watching YouTube video, the application layer protocols will be used are DNS and HTTP. User’s browser send DNS query to the server in order to distribute a fastest physical server from CDN system, and use HTTP to establish a connection between local machine and web server.

Part e

The reason is that TCP has feature of congestion control, which can cause less congestion problem in order to deliver high resolution video. The TCP protocol has reliable delivery to ensure the unbroken packets. Also, TCP protocol can easy to monitor the bandwidth from the user, which can be used to adjust the quality of video chunk that is requested form user browser.

Part f

QUIC is a multiplexed transport protocol that make a fusion of TCP, TLS and HTTP/2 [7]. Some improvements in QUIC will be mentioned and one of them is the handshaking process. TCP handshaking needs three way and extra 3 way for TLS encryption. QUIC combines the handshake with TCP and TLS, so in QUIC a three-way handshaking can complete connection between machines and encryption, which provides better performance and always encrypted connection [8]. Another improvement is head-of-line blocking. From previous question can found that if browser want to perform a webpage, CSS and referenced objects need to be requested by established multiple TCP connections, and they are influenced by packet loss, 'even if the data that was lost only concerned a single request' [8]. While the parallel connections in QUIC can share the connection state of QUIC which means they do not need to establish a new connection, and also the QUIC streams are deliver independently, which means packet loss of one stream will not affect others [8]. Based on improvements of latency, performance and packet losing, streaming services are more likely to use QUIC.

Part g

Under server-client architecture, the time to distribute a file with file size F to N client, each client has download speed d and upload speed u, server upload speed u_s. The equation of calculating distribution time is:

$$D_{c-s} \geq max\{NF/u_s, F/d_{min}\}$$

Which is given on the lecture slide. Where NF/u_s shows the total time of N copies of files with size F under server uploading speed u_s, and F/d shows the download time for each user. And $d_{min} = d$ since all clients' download speed are the same. The equation of the distribution time under optimal situation is:

$$min(D_{c-s}) = max\{NF/u_s, F/d\}$$

Part h

Under P2P architecture, the time to distribute a file with file size F to N client, each client has download speed d and upload speed u, server upload speed u_s. The equation of calculating distribution time is:

$$D_{P2P} \geq max\{F/u_s, F/d_{min}, NF/(u_s + \sum u_i)\}$$

Which is given on the lecture slide. Where F/u_s shows the single file upload time by server. F/d shows the download time of each user for single file. $NF/(u_s + Nu)$ shows the total time that server and peers upload N copies of file. And $d_{min} = d$ since all clients' download speed are the same, and the sum of peers' upload can be simplified since the upload speed are the same. The equation of the distribution time under optimal situation is:

$$min(D_{P2P}) = max\{F/u_s, F/d, NF/(u_s + Nu)\}$$

Part i

For CDN situation, the variable $N = kM$, where k is positive integer and M is the number of CDN nodes. The time of distribution should has equation:

$$min(D_{CDN}) = max\{kF/u_s,\ F/d\}$$

Where kF/u_s is the time of each CDN node upload number of k of the file copies, where assume that each CDN has number of k users, since each CDN node has dedicated maximum upload speed u_s so number of CDN node M has eliminated. F/d shows the download time of each user for a single file.

Part j

For several situations, the minimum distribution time for P2P architecture will less than CDN system, for example:

- The user upload speed u is equal or higher than the server upload speed u_s, where $u \geq u_s$, assume that the download time for user F/d will not affect the maximum selection, and the number of CDN node $M \geq 1$.
  In this case, since the user upload speed u just influence the minimum time for P2P architecture, so increase the user upload speed will have no affect to CDN system. Once $u \geq u_s$, the total upload time $NF/(u_s + Nu)$ will smaller than server upload time F/u_s, and according to the assumption, k must greater than 1, so the minimum distribution time of P2P architecture will smaller than CDN's.

For situations that CDN system has faster distribution time than P2P, for example:

- The number of CDN $M \geq 2$ and the user upload speed $u < \frac{1}{N}u_s$, assume that the download time for user F/d will not affect the maximum selection.
  In this case, the term $NF/(u_s + Nu)$ will bigger than kF/u_s, since the user upload speed $u < \frac{1}{N}u_s$. This will cause $NF/(u_s + Nu) > \frac{kNF}{u_s}$ since $k \geq 2$. Also the term $NF/(u_s + Nu) > F/u_s$. So the minimum distribution time for CDN must greater than P2P's.

Part k

Reasons that YouTube use CDN network rather than P2P architecture:

1. State of peers in P2P architecture always changing, peers may has different network congestion and some peers may hit-and-run (only download but not seeding), which cannot provide a stable bandwidth for video streaming.
2. CDN system provide load distribution and less latency to user, brings better experience to user.

Part l

For the advantage to using chunks rather than continuous ABR is that, chunks methods has far less connections during 30 seconds than continuous ABR. Since the continuous ABR need to receive N packets in 1 seconds by using UDP, which means N connections need to be established in 1 second and will cost more performance of router and machine. Also, the chunks method seems to sent encoded video chunks and continuous ABR sent packets that needs to be encoded on local machine, which will cost performance and need support by player, which need to be developed by software team.

Part m

For the disadvantage to using chucks rather than continuous ABR is that, the chunks method can only change resolution after playing the present chuck, which has 30 seconds. Also the user need to wait until the 30 seconds chunk download finished, which will both brings bad experience to users. While continuous ABR can play and buffer the video for each second, and also available to change quality each second, which is more flexible than chunks method.

Question 4

Part a

Section 1

Before the input of sentence, client side established a TCP connection with server side. When the sentence is finished and the enter key is put, the client machine will send the message in binary through the specific port number to server machine. After that the server machine program capitalise all the letter and return these to the client machine. Finally the client show the original message with all capital letter sent from server side and the socket is closed.

Section 2

When run TCPClient file without running TCPServer before, an error message popped out says ‘connection refused’. This is because that the server side has not start to listen the port yet. If the connection request to this port is raised, it will be refused by the system since no process is listening this port.

```
$ python ./TCPClient.py
Traceback (most recent call last):
  File "/home/tedwu/4119/HW2/./TCPClient.py", line 7, in <module>
    clientSocket.connect((serverName,serverPort))
ConnectionRefusedError: [Errno 111] Connection refused
```

Section 3

The terminal raise a same error with in section 2, where the connection is refused by the system since there is no process is listening the port that client side need to access.

Section 4

When client and server side is not at the same port, then the server side will not receive the message sent from client side, since the port number is unmatched, and the client will not connect to the port since no process is listening on this port. The TCP socket send message to port M while the server side can only receive the message in port N.

Part b

Section 1

When the sentence is finished and the enter key is put, the client machine will send the message in binary through the specific port number to server machine after establishing a UDP connection using specific server name and port number. After that the server machine program receive and capitalise all the letter and return these to the client machine. Finally the client show the original message with all capital letter sent from server side and the socket is closed.

Section 2

UDPClient.py file is successfully opened and the sentence is inputted since the program establish an UDP connection after the input message. While the server side will not receive the message since the UDP connection is established at the moment of finishing the input message, so the server side will not received the message that is sent before the server starts to listen the port.

Section 3

The message will be received by server side and return capital letters to the client side. The reason is mentioned in section 2, which is that the client side will open a UDP socket until the input message is finished.

Section 4

The message will not deliver to the server side because the port number in two program is not matched. The UDP socket send message to port M while the server side can only receive the message in port N.

Part c

Section 1

By running TCPServer.py, a welcome socket has been created by the code:

```
serverSocket = socket(AF_INET,SOCK_STREAM) ## LISTENING SOCKET
```

Which is a welcome socket in order to handshake with the client side.

Section 2

The maximum number of sockets created by server side is 2. One is the welcome socket as mentioned in section 1, and the second is the socket that is created to transmitting message with client side after the server side contacted with client in welcome socket.

Section 3

If n TCP client connects to the server side at the same time, the maximum number of concurrent socket number will be $n+1$, since 1 welcome TCP socket plus, each TCP connection from a new client needs a new TCP socket created by server side, the TCP socket need to let port number, IP address to be unique.

Section 4

By running UDPServer.py, an UDP socket has been created by the code:

```
serverSocket = socket(AF_INET, SOCK_DGRAM)
```

Where it create a UDP socket to transmit the message.

Section 5

The maximum number of UDP socket will be 1 since the UDP protocol does not need handshaking, all message deliver is running on one socket.

Section 6

If n UDP client connects to the server at the same time, the maximum number of concurrent socket number will still be 1, since the UDP protocol does not require handshaking so no welcome socket, and the UDP protocol does not require the uniqueness of port number. So although different client transmit message to server at different port, the server only use 1 UDP socket to receive all the message from different ports.

Part d

Section 1

In order to receives 10 connections at t=0, it can use threading to create 10 threads to create 10 TCP sockets. Write a function to contain all the codes about create socket, receive message, capitalise and send message. Then in the main function, create 10 threads to run the function, by using different port number. The specific code for server side is shown below, which is not been tested however. It is worth to mention that part of the code is referenced from project 1 preliminary stage submission which is written by myself.

```python
import socket
import threading

def tcpServer(serverPort):
    while True:
        try:
            # Create TCP socket
            serverSocket = socket(AF_INET,SOCK_STREAM)
            serverSocket.bind(('',serverPort))
            serverSocket.listen(1)
            # Waiting for the client connection
            connectionSocket, addr = serverSocket.accept()
            isExit = False
            while not isExit:
                # Recv client message
                clientMessage = connectionSocket.recv(2048)
                print("From receiver: " + clientMessage.decode())
                # If client disconnect, then disconnect and wait new
                if not clientMessage:
                    isExit = True
                # Send capitalised message
                serverMessage = message.decode().upper()
                print("Sent to client: " + serverMessage)
                connectionSocket.send(serverMessage.encode())
            # Restart socket if has query
            if isExit:
                try:
                    client_connectionSocket, addr = client2proxySocket.accept()
                    isExit = False
                except client2proxySocket.timeout:
                    isExit = True
            # Close sockets
            serverSocket.close()
            connectionSocket.close()
        except KeyboardInterrupt:
            # print("From proxy: Ctrl+C detected, closing the socket...")
            serverSocket.close()
            connectionSocket.close()
            exit()

if __name__ == '__main__':
    # initiate 10 TCP sockets to recv client connection
    threads = []
    for i in range(10)
        portNumber = 12300+i
        t = threading.Thread(target=tcpServer,args=(portNumber,))
        threads.append(t)
        # t.setDaemon(True)
    # Start threads
    for t in threads:
        t.start()
```

Section 2

The content of the question can be understand as the client need to send a object of Notes class which has 3 attributes. The implementation is that, create two classes in client and server side with same name and attributes. The client side response to save the user input of the Notes and save it to class as object. Once the client side receive all attributes of the Notes class, it then send all attributes value to server each by each. Since the value of note content may exceed buffer limit, so it is needed to detect the EOM. The specific code is shown below including client and server side, which is not been tested however. It is worth to mention that part of the code is referenced from project 1 preliminary stage submission which is written by myself.

Client side:

```
import socket
# import datetime

# Create Notes class
class Notes:
    noteCount = 0
    serverName = 'localhost'
    serverPort = 12300

    # Three attributes
    def __init__(self, date, content, author):
        self.creationDate = date
        self.content = content
```

```python
        self.author = author
        Notes.noteCount += 1

    # Send notes objects to server
    def sendNote(self):
        # Create socket
        clientSocket = socket(AF_INET, SOCK_STREAM)
        clientSocket.connect((serverName,serverPort))
        # Send date
        clientSocket.send(str(self.creationDate).encode())
        ACK = clientSocket.recv(2048).decode()
        # Receive server message to ACK
        if ACK == 'ACK':
            pass
        else:
            print("Failed")
            break
        # Send large mount of note content
        clientSocket.sendall(str(self.content).encode())
        ACK = clientSocket.recv(2048).decode()
        # Receive server message to ACK
        if ACK == 'ACK':
            pass
        else:
            print("Failed")
            break
        # Send author
        clientSocket.send(str(self.author).encode())
        ACK = clientSocket.recv(2048).decode()
        # Receive server message to ACK
        if ACK == 'ACK':
            pass
        else:
            print("Failed")
            break
        clientSocket.close()
        print("Sent")

if __name__ == '__main__':
    # date = datetime.now()
    # content = input("Type something for note content: ")
    # author = input("Type your name: ")
    note1 = Notes("Note 1", "10/17/2022", "itsforevera", "Vera")
    note1.sendNote()
```

Server side:

```python
import socket

tmp=0

# Create Notes class with same attributes as client side
class Notes:
    noteCount = 0

    def __init__(self, date, content, author):
        self.creationDate = date
        self.content = content
        self.author = author
        Notes.noteCount += 1

def serverTCP():
    serverPort = 12300
    while True:
        try:
            # Create TCP socket
            serverSocket = socket(AF_INET,SOCK_STREAM)
            serverSocket.bind(('',serverPort))
            serverSocket.listen(1)
            # Waiting for the client connection
            connectionSocket, addr = serverSocket.accept()
            isExit = False
            # Start recving
            # Date
            while not isExit:
                serverMessage = ''
                isContinue = True
                while isContinue:
                    clientMessage_tmp =
connectionSocket.recv(2048).decode()
                    date += clientMessage_tmp
                    # If client disconnect
                    if not serverMessage:
                        isExit = True
                        isContinue = False
                    # Detect EOM
                    if b'\n' in date.encode():
                        isContinue = False
                        clientMessage_tmp = ''
                        ACK = "ACK"
                        connectionSocket.send(ACK.encode())
            # Content
            while not isExit:
                serverMessage = ''
                isContinue = True
                while isContinue:
                    clientMessage_tmp =
connectionSocket.recv(2048).decode()
                    content += clientMessage_tmp
                    # If client disconnect
                    if not serverMessage:
                        isExit = True
                        isContinue = False
                    # Detect EOM
                    if b'\n' in content.encode():
                        isContinue = False
                        clientMessage_tmp = ''
                        ACK = "ACK"
                        connectionSocket.send(ACK.encode())
            # Author
            while not isExit:
                serverMessage = ''
                isContinue = True
                while isContinue:
                    clientMessage_tmp = connectionSocket.recv(2048).decode()
                    author += clientMessage_tmp
                    # If client disconnect
                    if not serverMessage:
                        isExit = True
                        isContinue = False
                    # Detect EOM
                    if b'\n' in author.encode():
                        isContinue = False
                        clientMessage_tmp = ''
                        ACK = "ACK"
                        connectionSocket.send(ACK.encode())
            # Insert values into class
            Note[tmp] = Notes(tmp, date, content, author)
            tmp += 1
            # Restart socket if has query
            if isExit:
                try:
                    client_connectionSocket, addr = client2proxySocket.accept()
                    isExit = False
                except client2proxySocket.timeout:
                    isExit = True
            # Close sockets
            serverSocket.close()
            connectionSocket.close()
        except KeyboardInterrupt:
            # print("From proxy: Ctrl+C detected, closing the socket...")
            serverSocket.close()
            connectionSocket.close()
            exit()

if __name__ == '__main__':
    serverTCP()
```

Section 3

In order to handle with the exception threw by the socket, using ‘try-except’ method is a way to solve it. The program will check the except condition, for example, whether the socket throw a timeout error or other error messages. In this case, the program will not run the code in ‘try’ area but will run code in specific ‘except’ area, so some code can be put here to ask user whether to have a retry. The specific client side code is shown below, which is not been tested however. It is worth to mention that part of the code is referenced from project 1 preliminary stage submission which is written by myself.

```python
import socket

if __name__ == '__main__':
    isRetry = False
    i = 0
    while isRetry:
        try:
            isRetry = False
            serverName = 'localhost'
            serverPort = 12300
            message = "itsforevera"
            # Create socket
            clientSocket = socket(AF_INET, SOCK_STREAM)
            clientSocket.connect((serverName,serverPort))
            # Send date
            clientSocket.send(message.encode())
        # Exception: the socket throw a error message
        except clientSocket.error as msg:
            # Auto retry
            print("Socket error: " + msg + "\nPlease retry")
            isRetry = True
            # Until retry 3 times
            if i == 3:
                print("Timeout")
                isRetry = False
                break
            i += 1
        # Exception: connection timeout
        except clientSocket.timeout:
            # For timeout, ask user whether to retry or edit the configure
            ans = input("Connection timeout, do you want to reconnect rather than edit port and server name?(y/N)")
            # Default for N
            if ans == "y":
                isRetry = True
            else:
                isRetry = False
```

References

[1] Belshe, M., Peon, R., & Thomson, M. (2015). *Hypertext Transfer Protocol Version 2 (HTTP/2)* (Request for Comments RFC 7540). Internet Engineering Task Force. https://doi.org/10.17487/RFC7540

[2] *11. HTTP/2 [RFC 7540]* . (n.d.). Retrieved 16 October 2022, from https://sunyunqiang.com/blog/http2_rfc7540/#112-http2-frame

[3] *DNS AAAA record* . (n.d.). Cloudflare. Retrieved 16 October 2022, from https://www.cloudflare.com/learning/dns/dns-records/dns-aaaa-record/

[4] *Speed of Light in Fiber—The First Building Block of a Low-Latency Trading Infrastructure* . (n.d.). Retrieved 17 October 2022, from https://www.blog.adva.com/en/speed-light-fiber-first-building-block-low-latency-trading-infrastructure

[5] *What is a DNS PTR record?* (n.d.). Cloudflare. Retrieved 17 October 2022, from https://www.cloudflare.com/learning/dns/dns-records/dns-ptr-record/

[6] *Reverse DNS - Hetzner Docs* . (n.d.). Retrieved 17 October 2022, from https://docs.hetzner.com/dns-console/dns/general/reverse-dns/#if-i-set-up-reverse-entries-ptr-for-my-ips-on-my-name-server-why-are-these-not-accepted

[7] *QUIC, a multiplexed transport over UDP* . (n.d.). Retrieved 17 October 2022, from https://www.chromium.org/quic/

[8] *The Road to QUIC* . (2018, July 26). The Cloudflare Blog. http://blog.cloudflare.com/the-road-to-quic/