NGK Journal 1

Contains journal for exercises 3, 4, 5, 6 and 7

Group members

Name	Student id	email
Marcin	202009418	201908195@post.au.dk
Martin	201908195	201908195@post.au.dk
Mathias Birk Olsen	202008722	202008722@post.au.dk

Exercise 3

Question 1

The time from a ping between H1 and H2 is measured with the command

> ping -c 1 10.0.0.2 on H1 to ping H2

```
$ ping -c 1 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.324 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.324/0.324/0.324/0.000 ms
```

the time is 0.324ms

Question 2

The same aproach is used, as above, this time with the command

> ping -c 10 10.0.0.2

```
ase@ubuntu:~$ ping -c 10 10.0.02
PING 10.0.02 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.599 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=1.01 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=1.06 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=1.06 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=1.03 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.428 ms
```

```
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=1.11 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.722 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=0.653 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=0.707 ms

--- 10.0.02 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9070ms
rtt min/avg/max/mdev = 0.428/0.837/1.105/0.229 ms
```

> Min = 0.428 ms
>
> avg = 0.837 ms
>
> max = 1.105 ms

Question 3

Lets now ping something on the world wide web. And our target will be non other then one of the moby dicks of the internet.

google.com

A signal ping first

```
ase@ubuntu:~$ ping -c 1 www.google.com
PING www.google.com (142.250.179.164) 56(84) bytes of data.
64 bytes from ams15s41-in-f4.1e100.net (142.250.179.164): icmp_seq=1 ttl=128
time=29.7 ms

--- www.google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 29.685/29.685/29.685/0.000 ms
```

The ping took 29.685 ms, which is alot slower then when we tested the neighbours H1 and H2.

Question 4

Now lets check the consistensy of google with 10 consecutive pings ping -c 10 www.google.com is called

```
PING www.google.com (142.250.179.164) 56(84) bytes of data.
64 bytes from ams15s41-in-f4.1e100.net (142.250.179.164): icmp_seq=1 ttl=128
time=30.5 ms
64 bytes from ams15s41-in-f4.1e100.net (142.250.179.164): icmp_seq=2 ttl=128
time=37.3 ms
64 bytes from ams15s41-in-f4.1e100.net (142.250.179.164): icmp_seq=3 ttl=128
time=30.3 ms
64 bytes from ams15s41-in-f4.1e100.net (142.250.179.164): icmp_seq=4 ttl=128
time=30.4 ms
64 bytes from ams15s41-in-f4.1e100.net (142.250.179.164): icmp_seq=5 ttl=128
```

```
time=29.7 ms
64 bytes from ams15s41-in-f4.1e100.net (142.250.179.164): icmp_seq=6 ttl=128
time=36.8 ms
64 bytes from ams15s41-in-f4.1e100.net (142.250.179.164): icmp_seq=7 ttl=128
time=36.3 ms
64 bytes from ams15s41-in-f4.1e100.net (142.250.179.164): icmp_seq=8 ttl=128
time=38.8 ms
64 bytes from ams15s41-in-f4.1e100.net (142.250.179.164): icmp_seq=9 ttl=128
time=37.1 ms
64 bytes from ams15s41-in-f4.1e100.net (142.250.179.164): icmp_seq=10 ttl=128
time=30.4 ms

--- www.google.com ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9018ms
rtt min/avg/max/mdev = 29.660/33.754/38.808/3.559 ms
```

> the minimum 29.669
>
> The avg time is 33.747
>
> The max time is 38.808

Question 5

Lets try pinging something that can't be pinged.

5.1 Pinging

```
ase@ubuntu:/etc/apache2$ ping -c 10 www.tv2.dk
PING aws-https-redirect-prod.tv2net.dk (3.123.202.164) 56(84) bytes of data.

^C
--- aws-https-redirect-prod.tv2net.dk ping statistics ---
10 packets transmitted, 0 received, 100% packet loss, time 9199ms
```

It is then confirmed that tv2.dk does not liked to be pinged.

5.2

Lets now measure a pingable site, but with wireshark instead. First, lets find our own IP with host -H

```
ase@ubuntu:~$ hostname -I
192.168.44.128 10.0.0.1
```

Then the three way handshake is used as indicator of the response. Identifing our SYN and the SYC, ACH, and the time between them.

```
7   0.698303972 192.168.44.128  10.83.252.23    TCP 74  36480 → 443 [SYN] Seq=0
Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=2515601345 TSecr=0 WS=128
8   0.701008276 10.83.252.23    192.168.44.128  TCP 60  443 → 36480 [SYN, ACK]
Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
9   0.701034020 192.168.44.128  10.83.252.23    TCP 54  36480 → 443 [ACK] Seq=1
Ack=1 Win=64240 Len=0
```

the time between them is $$ [SYC, ACK] - [ACK] = 0.701 - 0.698 = 0.002705304 $$

The three way handshake is also cool, and illustrated with the SYN from us, the SYN, ACK from AU, and our ACK to AU.

Confirming that we are speaking with AU

```
ase@ubuntu:~$ host 10.83.252.23
23.252.83.10.in-addr.arpa domain name pointer auinstallation48v6.cs.au.dk.
23.252.83.10.in-addr.arpa domain name pointer typo3.au.dk.
23.252.83.10.in-addr.arpa domain name pointer cepdisc.au.dk.
23.252.83.10.in-addr.arpa domain name pointer ced.au.dk.
```

We are

Question 6

now, lets travel the world. Lets wireshark the Australien goverment.

From the terminal we find the IPs of www.australia.gov.au.

```
ase@ubuntu:~$ host www.australia.gov.au
www.australia.gov.au is an alias for cdn.prod65.dta.adobecqms.net.
cdn.prod65.dta.adobecqms.net has address 18.64.103.78
cdn.prod65.dta.adobecqms.net has address 18.64.103.66
cdn.prod65.dta.adobecqms.net has address 18.64.103.43
cdn.prod65.dta.adobecqms.net has address 18.64.103.14
```

In wireshark we can find the three-way hand shake.

```
7   0.182429071 192.168.44.128  18.64.103.43    TCP 74  57228 → 443 [SYN] Seq=0
Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=3899355100 TSecr=0 WS=128
9   0.217609102 18.64.103.43    192.168.44.128  TCP 60  443 → 57228 [SYN, ACK]
Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
10  0.217711228 192.168.44.128  18.64.103.43    TCP 54  57228 → 443 [ACK] Seq=1
Ack=1 Win=64240 Len=0
```

Here we can see the time from our request to and the respose from Ausstrila.gov.au. $$ 0.217 - 0.182 = 0.035180031 $$

questions 7

The difrence between the time alculated in 5 og 6. $$ 0.035180031 - 0.002705304 = 0,032474727 $$ Not very much slower to go to Australia, only like a mili second.

Exercise 4: HTTP Client/server

1. Hamlet on HTTP

This task will examine loading the website: http://i4prj.ase.au.dk/I4IKN. This examination will be done with wireshark, Where wireshark will be set to recored before loading the page in the browser. Then wiresharks filter will be used to isolate the relavant packages. First DNS and then the http.

DNS query and response

The dns query

```
13  1.573834603 10.211.55.7 10.211.55.1 DNS 75  Standard query 0xfeb8 AAAA
i4prj.ase.au.dk
```

And the DNS respose

```
17  1.601046855 10.211.55.1 10.211.55.7 DNS 141 Standard query response 0xfeb8
AAAA i4prj.ase.au.dk SOA uniinfobloxintern02.uni.au.dk
```

Time delay

The time delay for the DNS query and respose was $$ 1.601046855 - 1.573834603 = 0.027212251999999992 $$

The HTTP request header

below the header recieved from the website can be seen. Where the content is between the HTTP and the finishing "\r\n"

```
    HTTP/1.1 304 Not Modified\r\n
    Cache-Control: private\r\n
    Server: Microsoft-IIS/8.0\r\n
    X-AspNet-Version: 4.0.30319\r\n
    X-Powered-By: ASP.NET\r\n
    Date: Sat, 12 Mar 2022 17:54:17 GMT\r\n
    \r\n
```

```
        [HTTP response 1/2]
        [Time since request: 0.026490211 seconds]
        [Request in frame: 56]
        [Next request in frame: 394]
        [Next response in frame: 445]
        [Request URI: http://i4prj.ase.au.dk/I4IKN/bundles/modernizr?
    v=inCVuEFe6J4Q07A0AcRsbJic_UE5MwpRMNGcOtk94TE1]
```

The HTTP request body

Following the header, the body of the website is delivered. From the HTTP, some information about the package is first recieved, followed by the line-based text data, where the content of the site actually resite

Here is the information as listed in wireshark.

```
        HTTP/1.1 200 OK\r\n
        Cache-Control: private\r\n
        Content-Type: text/html; charset=utf-8\r\n
        Server: Microsoft-IIS/8.0\r\n
        X-AspNetMvc-Version: 5.2\r\n
        X-AspNet-Version: 4.0.30319\r\n
        X-Powered-By: ASP.NET\r\n
        Date: Sat, 12 Mar 2022 17:54:17 GMT\r\n
        Content-Length: 255303\r\n
        \r\n
        [HTTP response 1/2]
        [Time since request: 0.213257541 seconds]
        [Request in frame: 18]
        [Next request in frame: 391]
        [Next response in frame: 397]
        [Request URI: http://i4prj.ase.au.dk/I4IKN]
        File Data: 255303 bytes
    Line-based text data: text/html (10220 lines)
```

Where the last line, Line-based text data: text/html (10220 lines), is the 10220 lines of hamlet, formatted with html.

2. Installing apache2

The installation of the apache2 server went smoothly, and after it was done, it was checked if the apache server was running with the command

> sudo systemctl status apache2.service

This confirmed that it was loaded and active, with the output containing:

```
         Loaded: loaded (/lib/systemd/system/apache2.service; enabled; vendor preset:
    enabled)
         Active: active (running) since Sat 2022-03-12 19:29:24 CET; 4min 0s ago
```

3. Establising a lan conection

The connection to the server was then tested with the command telnet.

```
ase@ubuntu:~$ telnet 10.0.0.1 80
Trying 10.0.0.1...
Connected to 10.0.0.1.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.1 200 OK
Date: Sun, 13 Mar 2022 10:25:41 GMT
Server: Apache/2.4.41 (Ubuntu)
Last-Modified: Sat, 12 Mar 2022 18:29:22 GMT
ETag: "2aa6-5da09a048cb63"
Accept-Ranges: bytes
Content-Length: 10918
Vary: Accept-Encoding
Connection: close
Content-Type: text/html
```

Following the snippet seen above, the content of the standard index file was printet.

4. Connecting to the server from the client.

The command telnet can be used to establish the connections via the local area network.

> telnet <address> <port>

When the connection is established, a prombt to enter a command for the server will appear. In our case, GET is our friend.

> GET <path/to/file> HTTP/1.X

> GET / HTTP/1.0 host: 10.0.0.2

> GET / HTTP/1.1 host: 10.0.0.2

Where, at least in 1.1 case, will add a host. The name / address of the client is used.

HTTP 1.0

The commulication between the server and the client when getting with HTTP 1.0 is seen here:

```
No. Time    Source  Destination Protocol    Length  Info
16  3.242932398     10.0.0.2    10.0.0.1    TCP 74  36744 → 80 [SYN] Seq=0
Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=1871154045 TSecr=0 WS=128
17  3.242960021     10.0.0.1    10.0.0.2    TCP 74  80 → 36744 [SYN, ACK] Seq=0
```

```
Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=313594699 TSecr=1871154045 WS=128
18  3.243205239     10.0.0.2    10.0.0.1    TCP 66  36744 → 80 [ACK] Seq=1 Ack=1
Win=64256 Len=0 TSval=1871154046 TSecr=313594699
46  13.353596113    10.0.0.2    10.0.0.1    TCP 82  36744 → 80 [PSH, ACK] Seq=1
Ack=1 Win=64256 Len=16 TSval=1871164156 TSecr=313594699 [TCP segment of a
reassembled PDU]
47  13.353632645    10.0.0.1    10.0.0.2    TCP 66  80 → 36744 [ACK] Seq=1 Ack=17
Win=65152 Len=0 TSval=313604809 TSecr=1871164156
57  18.413083289    10.0.0.2    10.0.0.1    TCP 82  36744 → 80 [PSH, ACK] Seq=17
Ack=1 Win=64256 Len=16 TSval=1871169216 TSecr=313604809 [TCP segment of a
reassembled PDU]
58  18.413105605    10.0.0.1    10.0.0.2    TCP 66  80 → 36744 [ACK] Seq=1 Ack=33
Win=65152 Len=0 TSval=313609869 TSecr=1871169216
61  19.564063730    10.0.0.2    10.0.0.1    HTTP    68  GET / HTTP/1.0
62  19.564080406    10.0.0.1    10.0.0.2    TCP 66  80 → 36744 [ACK] Seq=1 Ack=35
Win=65152 Len=0 TSval=313611020 TSecr=1871170367
63  19.564276223    10.0.0.1    10.0.0.2    TCP 2962    80 → 36744 [PSH, ACK]
Seq=1 Ack=35 Win=65152 Len=2896 TSval=313611020 TSecr=1871170367 [TCP segment of a
reassembled PDU]
64  19.564300455    10.0.0.1    10.0.0.2    TCP 2962    80 → 36744 [PSH, ACK]
Seq=2897 Ack=35 Win=65152 Len=2896 TSval=313611020 TSecr=1871170367 [TCP segment
of a reassembled PDU]
65  19.564369838    10.0.0.1    10.0.0.2    TCP 2962    80 → 36744 [PSH, ACK]
Seq=5793 Ack=35 Win=65152 Len=2896 TSval=313611020 TSecr=1871170367 [TCP segment
of a reassembled PDU]
66  19.564392873    10.0.0.1    10.0.0.2    HTTP    2570    HTTP/1.1 200 OK
(text/html)
67  19.564518744    10.0.0.1    10.0.0.2    TCP 66  80 → 36744 [FIN, ACK]
Seq=11193 Ack=35 Win=65152 Len=0 TSval=313611020 TSecr=1871170367
68  19.564559132    10.0.0.2    10.0.0.1    TCP 66  36744 → 80 [ACK] Seq=35
Ack=2897 Win=63488 Len=0 TSval=1871170367 TSecr=313611020
69  19.564559167    10.0.0.2    10.0.0.1    TCP 66  36744 → 80 [ACK] Seq=35
Ack=5793 Win=61568 Len=0 TSval=1871170367 TSecr=313611020
70  19.564724576    10.0.0.2    10.0.0.1    TCP 66  36744 → 80 [ACK] Seq=35
Ack=8689 Win=63488 Len=0 TSval=1871170367 TSecr=313611020
71  19.564724641    10.0.0.2    10.0.0.1    TCP 66  36744 → 80 [ACK] Seq=35
Ack=11193 Win=61568 Len=0 TSval=1871170367 TSecr=313611020
72  19.565069600    10.0.0.2    10.0.0.1    TCP 66  36744 → 80 [FIN, ACK] Seq=35
Ack=11194 Win=64128 Len=0 TSval=1871170368 TSecr=313611020
73  19.565093189    10.0.0.1    10.0.0.2    TCP 66  80 → 36744 [ACK] Seq=11194
Ack=36 Win=65152 Len=0 TSval=313611021 TSecr=1871170368
```

The HTTP1.1 Number 16 til 18 are the three-way handshake between the client and server.

At number 61, the HTTP get instruction arrive and it can be seen that the protocol to be used is 1.0. Then the follwing 4 packecs are the transfer of the html file. Then a HTTP ok and [FIN, ACK] is recieved and the transfer is finnished.

After this, the clients sends ackknowledgements for the packets and then the connection is closed immediatly. Or alt least within a micro second.

Closed by who?

The connection is closed by the server.

Version of apache ?

Yes, this can be seen in the HTTP OK message at 66. If it is unfolded, it can be seen that the version is 2.4.41.

HTTP 1.1

```
No. Time    Source  Destination Protocol    Length  Info
1   0.000000000     10.0.0.2    10.0.0.1    TCP 74  36740 → 80 [SYN] Seq=0
Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=1870786059 TSecr=0 WS=128
2   0.000028144     10.0.0.1    10.0.0.2    TCP 74  80 → 36740 [SYN, ACK] Seq=0
Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=313226713 TSecr=1870786059 WS=128
3   0.000238935     10.0.0.2    10.0.0.1    TCP 66  36740 → 80 [ACK] Seq=1 Ack=1
Win=64256 Len=0 TSval=1870786059 TSecr=313226713
21  14.257228804    10.0.0.2    10.0.0.1    TCP 82  36740 → 80 [PSH, ACK] Seq=1
Ack=1 Win=64256 Len=16 TSval=1870800316 TSecr=313226713 [TCP segment of a
reassembled PDU]
22  14.257270236    10.0.0.1    10.0.0.2    TCP 66  80 → 36740 [ACK] Seq=1 Ack=17
Win=65152 Len=0 TSval=313240970 TSecr=1870800316
63  22.528849980    10.0.0.2    10.0.0.1    TCP 82  36740 → 80 [PSH, ACK] Seq=17
Ack=1 Win=64256 Len=16 TSval=1870808588 TSecr=313240970 [TCP segment of a
reassembled PDU]
64  22.528869488    10.0.0.1    10.0.0.2    TCP 66  80 → 36740 [ACK] Seq=1 Ack=33
Win=65152 Len=0 TSval=313249242 TSecr=1870808588
69  25.398128406    10.0.0.2    10.0.0.1    HTTP    68  GET / HTTP/1.1
70  25.398152490    10.0.0.1    10.0.0.2    TCP 66  80 → 36740 [ACK] Seq=1 Ack=35
Win=65152 Len=0 TSval=313252111 TSecr=1870811457
71  25.398365099    10.0.0.1    10.0.0.2    TCP 2962    80 → 36740 [PSH, ACK]
Seq=1 Ack=35 Win=65152 Len=2896 TSval=313252112 TSecr=1870811457 [TCP segment of a
reassembled PDU]
72  25.398389806    10.0.0.1    10.0.0.2    TCP 2962    80 → 36740 [PSH, ACK]
Seq=2897 Ack=35 Win=65152 Len=2896 TSval=313252112 TSecr=1870811457 [TCP segment
of a reassembled PDU]
73  25.398453403    10.0.0.1    10.0.0.2    TCP 2962    80 → 36740 [PSH, ACK]
Seq=5793 Ack=35 Win=65152 Len=2896 TSval=313252112 TSecr=1870811457 [TCP segment
of a reassembled PDU]
74  25.398477270    10.0.0.1    10.0.0.2    HTTP    2551    HTTP/1.1 200 OK
(text/html)
75  25.398671528    10.0.0.2    10.0.0.1    TCP 66  36740 → 80 [ACK] Seq=35
Ack=2897 Win=63488 Len=0 TSval=1870811458 TSecr=313252112
76  25.398671594    10.0.0.2    10.0.0.1    TCP 66  36740 → 80 [ACK] Seq=35
Ack=5793 Win=61568 Len=0 TSval=1870811458 TSecr=313252112
77  25.398732035    10.0.0.2    10.0.0.1    TCP 66  36740 → 80 [ACK] Seq=35
Ack=11174 Win=57600 Len=0 TSval=1870811458 TSecr=313252112
80  30.404531733    10.0.0.1    10.0.0.2    TCP 66  80 → 36740 [FIN, ACK]
Seq=11174 Ack=35 Win=65152 Len=0 TSval=313257118 TSecr=1870811458
81  30.405217993    10.0.0.2    10.0.0.1    TCP 66  36740 → 80 [FIN, ACK] Seq=35
Ack=11175 Win=64128 Len=0 TSval=1870816464 TSecr=313257118
82  30.405268076    10.0.0.1    10.0.0.2    TCP 66  80 → 36740 [ACK] Seq=11175
Ack=36 Win=65152 Len=0 TSval=313257118 TSecr=1870816464
```

From what can be observed here, The HTTP1.1 protocol starts out, transmits and finnishes the transmistion in the same way that the HTTP1.0 does. THe only execption is that the closing of the connection happens 5 seconds later. This is a sympton of the pipelining introduces with the 1.1 protocol, Allowing for handeling multiple request at a time. The delay of closing opens the window for recieveing and handeling more requests, where the 1.0 protocal only allows time for one connection.

The connection is closed by the server, In packed 74 HTTP/1.1 200 OK the version of appache was transferd aswell.

5. Sending our own html file.

The html page:

This html page is naivly develeped with three divs and 3 imgages. The images are supplyed by this website: pixabay.com

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Birks NGK TEST</title>
</head>
<body>
   <div><img src="img/img3.jpeg" alt="birds in the sky"></div>
   <div><img src="img/img2.jpeg" alt="fog in the hills"></div>
   <div><img src="img/img1.jpeg" alt="lake in the mountains"></div>
</body>
```

Examination of the new site

Wireshark is then used to examine how the new website is transferred. Wireshark is stareted on server, and then the website is refreshed on the client.

```
No. Time    Source  Destination Protocol    Length  Info
1   0.000000000 10.0.0.2    10.0.0.1    TCP 74  35178 → 80 [SYN] Seq=0 Win=64240
Len=0 MSS=1460 SACK_PERM=1 TSval=4020241694 TSecr=0 WS=128
2   0.000027566 10.0.0.1    10.0.0.2    TCP 74  80 → 35178 [SYN, ACK] Seq=0 Ack=1
Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=1297789268 TSecr=4020241694 WS=128
3   0.000369113 10.0.0.2    10.0.0.1    TCP 66  35178 → 80 [ACK] Seq=1 Ack=1
Win=64256 Len=0 TSval=4020241694 TSecr=1297789268
```

Above we see the three way handshake establishing the connection.

```
4   0.000395130 10.0.0.2    10.0.0.1    HTTP    530 GET /birk.html HTTP/1.1
5   0.000414948 10.0.0.1    10.0.0.2    TCP 66  80 → 35178 [ACK] Seq=1 Ack=465
Win=64768 Len=0 TSval=1297789268 TSecr=4020241694
6   0.000811457 10.0.0.1    10.0.0.2    HTTP    685 HTTP/1.1 200 OK  (text/html)
```

After the connection is established, H2 request the header and H2 ansers with it.

```
7   0.001151914 10.0.0.2    10.0.0.1    TCP 66  35178 → 80 [ACK] Seq=465 Ack=620
Win=64128 Len=0 TSval=4020241695 TSecr=1297789268
8   0.031909052 10.0.0.2    10.0.0.1    HTTP    477 GET /img/img3.jpeg HTTP/1.1
9   0.031927519 10.0.0.1    10.0.0.2    TCP 66  80 → 35178 [ACK] Seq=620 Ack=876
Win=64384 Len=0 TSval=1297789299 TSecr=4020241725
10  0.032193148 10.0.0.1    10.0.0.2    HTTP    248 HTTP/1.1 304 Not Modified
```

Then the clients asks for the first images on a new tcp connection, witch the server gladly supllies. This is then followed by what apears to be two simuntaniorsly 3-three way handshake. This is a proper exsample of pipelineing.

```
11  0.032364183 10.0.0.2    10.0.0.1    TCP 74  35180 → 80 [SYN] Seq=0 Win=64240
Len=0 MSS=1460 SACK_PERM=1 TSval=4020241726 TSecr=0 WS=128
12  0.032364274 10.0.0.2    10.0.0.1    TCP 74  35182 → 80 [SYN] Seq=0 Win=64240
Len=0 MSS=1460 SACK_PERM=1 TSval=4020241726 TSecr=0 WS=128
13  0.032383583 10.0.0.1    10.0.0.2    TCP 74  80 → 35180 [SYN, ACK] Seq=0 Ack=1
Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=1297789300 TSecr=4020241726 WS=128
14  0.032416726 10.0.0.1    10.0.0.2    TCP 74  80 → 35182 [SYN, ACK] Seq=0 Ack=1
Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=1297789300 TSecr=4020241726 WS=128
15  0.032568040 10.0.0.2    10.0.0.1    TCP 66  35178 → 80 [ACK] Seq=876 Ack=802
Win=64128 Len=0 TSval=4020241726 TSecr=1297789300
16  0.032638317 10.0.0.2    10.0.0.1    TCP 66  35180 → 80 [ACK] Seq=1 Ack=1
Win=64256 Len=0 TSval=4020241726 TSecr=1297789300
17  0.032638373 10.0.0.2    10.0.0.1    TCP 66  35182 → 80 [ACK] Seq=1 Ack=1
Win=64256 Len=0 TSval=4020241726 TSecr=1297789300
```

Then the two remaining pictures are transferd by the same means.

```
18  0.033298943 10.0.0.2    10.0.0.1    HTTP    477 GET /img/img2.jpeg HTTP/1.1
19  0.033306183 10.0.0.1    10.0.0.2    TCP 66  80 → 35178 [ACK] Seq=802 Ack=1287
Win=64128 Len=0 TSval=1297789301 TSecr=4020241727
20  0.033443563 10.0.0.2    10.0.0.1    HTTP    477 GET /img/img1.jpeg HTTP/1.1
21  0.033462780 10.0.0.1    10.0.0.2    TCP 66  80 → 35182 [ACK] Seq=1 Ack=412
Win=64768 Len=0 TSval=1297789301 TSecr=4020241727
22  0.033598658 10.0.0.1    10.0.0.2    HTTP    248 HTTP/1.1 304 Not Modified
23  0.033655496 10.0.0.1    10.0.0.2    HTTP    249 HTTP/1.1 304 Not Modified
```

And as the last part of the site have been transferd, all the tcp connections are closed with ACK and FIN, ARKS.

```
24  0.033917042 10.0.0.2    10.0.0.1    TCP 66  35178 → 80 [ACK] Seq=1287 Ack=984
Win=64128 Len=0 TSval=4020241728 TSecr=1297789301
25  0.033917090 10.0.0.2    10.0.0.1    TCP 66  35182 → 80 [ACK] Seq=412 Ack=184
Win=64128 Len=0 TSval=4020241728 TSecr=1297789301
26  5.007173688 10.0.0.1    10.0.0.2    TCP 66  80 → 35178 [FIN, ACK] Seq=984
Ack=1287 Win=64128 Len=0 TSval=1297794275 TSecr=4020241728
27  5.008244518 10.0.0.2    10.0.0.1    TCP 66  35178 → 80 [FIN, ACK] Seq=1287
Ack=985 Win=64128 Len=0 TSval=4020246701 TSecr=1297794275
28  5.008320955 10.0.0.1    10.0.0.2    TCP 66  80 → 35178 [ACK] Seq=985 Ack=1288
Win=64128 Len=0 TSval=1297794276 TSecr=4020246701
29  5.037869969 10.0.0.1    10.0.0.2    TCP 66  80 → 35182 [FIN, ACK] Seq=184
Ack=412 Win=64768 Len=0 TSval=1297794305 TSecr=4020241728
30  5.038944833 10.0.0.2    10.0.0.1    TCP 66  35182 → 80 [FIN, ACK] Seq=412
Ack=185 Win=64128 Len=0 TSval=4020246732 TSecr=1297794305
31  5.038981795 10.0.0.1    10.0.0.2    TCP 66  80 → 35182 [ACK] Seq=185 Ack=413
Win=64768 Len=0 TSval=1297794306 TSecr=4020246732
32  6.007081031 10.0.0.2    10.0.0.1    TCP 66  35180 → 80 [FIN, ACK] Seq=1 Ack=1
Win=64256 Len=0 TSval=4020247700 TSecr=1297789300
33  6.007262341 10.0.0.1    10.0.0.2    TCP 66  80 → 35180 [FIN, ACK] Seq=1 Ack=2
Win=65280 Len=0 TSval=1297795275 TSecr=4020247700
34  6.007603475 10.0.0.2    10.0.0.1    TCP 66  35180 → 80 [ACK] Seq=2 Ack=2
Win=64256 Len=0 TSval=4020247701 TSecr=1297795275
```

The content of the request header

In the request header, can bee seen from the keyword GET to the escape sequence "\r\n" in the snippet below. The GET key word is followed by the requested page. After the escape sequence the statistics of the packeds transfer.

```
Hypertext Transfer Protocol
    GET /birk.html HTTP/1.1\r\n
    Host: 10.0.0.1\r\n
    User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:97.0) Gecko/20100101
Firefox/97.0\r\n
    Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=
0.8\r\n
    Accept-Language: en-US,en;q=0.5\r\n
    Accept-Encoding: gzip, deflate\r\n
    Connection: keep-alive\r\n
    Upgrade-Insecure-Requests: 1\r\n
    If-Modified-Since: Sun, 13 Mar 2022 07:27:10 GMT\r\n
    If-None-Match: "1cf-5da147deb8c32-gzip"\r\n
    Cache-Control: max-age=0\r\n
    \r\n
    [Full request URI: http://10.0.0.1/birk.html]
    [HTTP request 1/3]
    [Response in frame: 6]
    [Next request in frame: 8]
```

The content of the responce header

The Response can be seen from HTTP keyword to the escape sequence "\r\n". Here it can see that its the first of three responces.

```
Hypertext Transfer Protocol
    HTTP/1.1 200 OK\r\n
    Date: Sun, 13 Mar 2022 07:36:27 GMT\r\n
    Server: Apache/2.4.41 (Ubuntu)\r\n
    Last-Modified: Sun, 13 Mar 2022 07:27:10 GMT\r\n
    ETag: "1cf-5da147deb8c32-gzip"\r\n
    Accept-Ranges: bytes\r\n
    Vary: Accept-Encoding\r\n
    Content-Encoding: gzip\r\n
    Content-Length: 282\r\n
    Keep-Alive: timeout=5, max=100\r\n
    Connection: Keep-Alive\r\n
    Content-Type: text/html\r\n
    \r\n
    [HTTP response 1/3]
    [Time since request: 0.000416327 seconds]
    [Request in frame: 4]
    [Next request in frame: 8]
    [Next response in frame: 10]
    [Request URI: http://10.0.0.1/birk.html]
    Content-encoded entity body (gzip): 282 bytes -> 463 bytes
    File Data: 463 bytes
Line-based text data: text/html (17 lines)
```

Exercise 3

In this Exercise we will examine the DNS-protocol with the cmd HOST. (The command nslookup can be used on non-unix systems, with at the time of writing where not avaible).

Just running host in the terminal will display the options that the host cmd can take. This is illustrted here: Use only the

```
ase@ubuntu:~/NGK_Repo$ host
Usage: host [-aCdilrTvVw] [-c class] [-N ndots] [-t type] [-W time]
            [-R number] [-m flag] hostname [server]
       -a is equivalent to -v -t ANY
       -A is like -a but omits RRSIG, NSEC, NSEC3
       -c specifies query class for non-IN data
       -C compares SOA records on authoritative nameservers
       -d is equivalent to -v
       -l lists all hosts in a domain, using AXFR
       -m set memory debugging flag (trace|record|usage)
       -N changes the number of dots allowed before root lookup is done
```

```
        -r disables recursive processing
        -R specifies number of retries for UDP packets
        -s a SERVFAIL response should stop query
        -t specifies the query type
        -T enables TCP/IP mode
        -U enables UDP mode
        -v enables verbose output
        -V print version number and exit
        -w specifies to wait forever for a reply
        -W specifies how long to wait for a reply
        -4 use IPv4 query transport only
        -6 use IPv6 query transport only

```

The settings used to test different DNS protocols have been chosen by one part randomeness and one part "i belive i have heard about this before".

The host cmd will be ran with different flags

- -4 : Use only the ipv4 for query transport
- -6 : Use only the ipv6 for query transport
- -a : "all"
- -d : printing debugging traceings

three websites are tested www.google.com, www.tv2.dk and www.australia.gov.au.

Conclusion from the test

noflags

Google returns a simple ip when the host is called vanilla on www.google.com. Where the two others returns their DNS alias and a series of ips for that alias. A best guess is that google has thier own DNS server and dont have a need for alias.

-a

All three websits returned "host <website> not found: 4(notimp)".

-4

-4 returns the exact same result as host with no flags. This makes sense since we are looking up the IP4 connected to the domain in the DNS.

-6

The connection to all websites timed out when IPV6 was used to transmit the request. This probably indicates something we will learn later in the couse.

-d

Gives a long detailed answer, that, agian, we might be able to interprete later in the course.

The tests

The test have been automated with the following shell script

```
#!/bin/bash
function code {
    echo "$@"
    echo '```'
    $@
    echo '```'
}
echo "## Running host test on "$1""  > "$2"
echo '### No flags' >> "$2"
echo No flags test.
code host "$1" >> "$2"
echo '### -a flag' >> "$2"
echo -a flag test
code host -a "$1" >> "$2"
echo '### -4 flags' >> "$2"
echo -4 flag test
code host -4 "$1" >> "$2"
echo '### -6 flags' >> "$2"
echo -6 flag test
code host -6 "$1" >> "$2"
echo '### -d flags' >> "$2"
echo -d flag test
code host -d "$1" >> "$2"
```

Running host test on www.google.com

No flags

host www.google.com

```
www.google.com has address 142.250.179.164
```

-a flag

host -a www.google.com

```
Trying "www.google.com"
Host www.google.com not found: 4(NOTIMP)
Received 32 bytes from 127.0.0.53#53 in 24 ms
```

-4 flags

host -4 www.google.com

```
www.google.com has address 142.250.179.164
```

-6 flags

host -6 www.google.com

```
;; connection timed out; no servers could be reached

```

-d flags

host -d www.google.com

```
Trying "www.google.com"
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 45542
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;www.google.com.            IN  A

;; ANSWER SECTION:
www.google.com.     74  IN  A   142.250.179.164

Received 48 bytes from 127.0.0.53#53 in 0 ms
Trying "www.google.com"
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 22139
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;www.google.com.            IN  AAAA

Received 32 bytes from 127.0.0.53#53 in 12 ms
Trying "www.google.com"
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 10348
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;www.google.com.            IN  MX

Received 32 bytes from 127.0.0.53#53 in 48 ms
```

Running host test on www.australia.gov.au

No flags

host www.australia.gov.au

```
www.australia.gov.au is an alias for cdn.prod65.dta.adobecqms.net.
cdn.prod65.dta.adobecqms.net has address 18.64.103.66
cdn.prod65.dta.adobecqms.net has address 18.64.103.43
cdn.prod65.dta.adobecqms.net has address 18.64.103.14
cdn.prod65.dta.adobecqms.net has address 18.64.103.78
```

-a flag

host -a www.australia.gov.au

```
Trying "www.australia.gov.au"
Host www.australia.gov.au not found: 4(NOTIMP)
Received 38 bytes from 127.0.0.53#53 in 24 ms
```

-4 flags

host -4 www.australia.gov.au

```
www.australia.gov.au is an alias for cdn.prod65.dta.adobecqms.net.
cdn.prod65.dta.adobecqms.net has address 18.64.103.66
cdn.prod65.dta.adobecqms.net has address 18.64.103.43
cdn.prod65.dta.adobecqms.net has address 18.64.103.14
cdn.prod65.dta.adobecqms.net has address 18.64.103.78
```

-6 flags

host -6 www.australia.gov.au

```
;; connection timed out; no servers could be reached
```

-d flags

host -d www.australia.gov.au

```
Trying "www.australia.gov.au"
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 33623
;; flags: qr rd ra; QUERY: 1, ANSWER: 5, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
```

```
;www.australia.gov.au.      IN  A

;; ANSWER SECTION:
www.australia.gov.au.   49  IN  CNAME   cdn.prod65.dta.adobecqms.net.
cdn.prod65.dta.adobecqms.net. 49 IN A   18.64.103.66
cdn.prod65.dta.adobecqms.net. 49 IN A   18.64.103.43
cdn.prod65.dta.adobecqms.net. 49 IN A   18.64.103.14
cdn.prod65.dta.adobecqms.net. 49 IN A   18.64.103.78

Received 144 bytes from 127.0.0.53#53 in 0 ms
Trying "cdn.prod65.dta.adobecqms.net"
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 11854
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;cdn.prod65.dta.adobecqms.net.  IN  AAAA

Received 46 bytes from 127.0.0.53#53 in 28 ms
Trying "cdn.prod65.dta.adobecqms.net"
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 42663
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;cdn.prod65.dta.adobecqms.net.  IN  MX

Received 46 bytes from 127.0.0.53#53 in 28 ms
```

Running host test on www.tv2.dk

No flags

host www.tv2.dk

```
www.tv2.dk is an alias for aws-https-redirect-prod.tv2net.dk.
aws-https-redirect-prod.tv2net.dk has address 3.123.214.150
aws-https-redirect-prod.tv2net.dk has address 3.123.214.120
aws-https-redirect-prod.tv2net.dk has address 3.123.202.164
```

-a flag

host -a www.tv2.dk

```
Trying "www.tv2.dk"
Host www.tv2.dk not found: 4(NOTIMP)
Received 28 bytes from 127.0.0.53#53 in 24 ms
```

-4 flags

host -4 www.tv2.dk

```
www.tv2.dk is an alias for aws-https-redirect-prod.tv2net.dk.
aws-https-redirect-prod.tv2net.dk has address 3.123.214.150
aws-https-redirect-prod.tv2net.dk has address 3.123.214.120
aws-https-redirect-prod.tv2net.dk has address 3.123.202.164
```

-6 flags

host -6 www.tv2.dk

```
;; connection timed out; no servers could be reached
```

-d flags

host -d www.tv2.dk

```
Trying "www.tv2.dk"
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 26528
;; flags: qr rd ra; QUERY: 1, ANSWER: 4, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;www.tv2.dk.            IN   A

;; ANSWER SECTION:
www.tv2.dk.     28  IN  CNAME   aws-https-redirect-prod.tv2net.dk.
aws-https-redirect-prod.tv2net.dk. 247 IN A 3.123.214.150
aws-https-redirect-prod.tv2net.dk. 247 IN A 3.123.214.120
aws-https-redirect-prod.tv2net.dk. 247 IN A 3.123.202.164

Received 121 bytes from 127.0.0.53#53 in 0 ms
Trying "aws-https-redirect-prod.tv2net.dk"
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 10672
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;aws-https-redirect-prod.tv2net.dk. IN  AAAA

Received 51 bytes from 127.0.0.53#53 in 64 ms
Trying "aws-https-redirect-prod.tv2net.dk"
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 26478
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
```

```
;aws-https-redirect-prod.tv2net.dk. IN  MX

Received 51 bytes from 127.0.0.53#53 in 12 ms
```

Exercise 6

TCP-Client/Server

In this exercise we develop a TCP Server that can be used to download a file via a TCP client, which we also develop. The programming language used in this exercise is C++.

Server app

To create a TCP server we use linux's socket.h API. In our solution we opted to create a TCP server class, which contains the necessary data and most functions for setting up the server.

socket.h abstracts a fair deal of both TCP and UDP socket functionality into simple functions. A TCP server needs to have a socket which listens to incoming connections. After such a socket has been created, bound and set to listening mode, a client can attempt to connect with a server.

```
// Create a socket used to accept incoming connections
requestSocket = socket(AF_INET, SOCK_STREAM, 0);

// Bind the socket to this IP address and port 9000
if(bind(requestSocket, (struct sockaddr *)&thisAddress, sizeof(address)) < 0 ){
    throw "Unable to bind \n";
}
// Listen to max 3 connections simultaneously through this socket
if(listen(requestSocket, 3) < 0){
    throw "Listen error \n";
}
```

the struct sockaddr_in, cast into sockaddr* , is used to store port number, IP address and the family of the address, such as IP4, IP6 or even a non-internet based family. This server uses port 9000, IP 10.0.0.1 and IP4 family and is set up in the constructor.

The server then waits for requests to connect (the 3 way TCP handshake) and accepts the first incoming request, creating a new socket via accept() function.

```
accSocket = accept(requestSocket, (struct sockaddr *)&address,
(socklen_t*)&addrLen);
```

Then it reads a string from the socket and checks whether the string matches a directory and a filename on the machine, and if so, attempts to send the file.

Sending the file

To send the file the server opens a file, reads 1024 bytes of data into the buffer, then sends it via socket via send() command. Reading and sending repeated up to filesize/bufferSize times, and the last time the server sends filesize % bufferSize bytes. This ensures that the correct amount of bytes is sent at all times. In our solution we opted to respond to each 1024 data chunk transfered on the client, so the server waits for the client to save the data.

```
// Send file loop
for(int i = 0; i < num_loops; i++){
    sent = read(fd, &sendBuffer, BUF_SIZE);
    actuallySent += send(outToClient, &sendBuffer, BUF_SIZE, 0);
    read(outToClient, &conf, 4);
}
```

Server client

Unlike the server we opted not to create a whole class for a client which is not supposed to run several times. The client side of a TCP connection needs only to create a socket and request a connection via connect() function to be able send and receive messages.

```
// Create a socket to connect to the server
if((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0){
...
// Connect to the server
if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
...
```

The string to send to the server is input upon calling the program. Then the client waits to receive either filesize or an error message, and if succesful, prepares to download the file.

Downloading the file

To properly download a file the client must create a new file with write privileges enabled, read the correct number of bytes from the socket and write the data read into the new file. The number of loops is again filesize/bufferSize, and after the for loop the client must read filesize % bufferSize bytes.

```
for(int i = 0; i < num_loops; i++){
    // Read from the stream
    int readBytes = read(fd, buffer, BUF_SIZE);
    totalBytes += readBytes;
    // Print download status periodically
    if(i%500 == 0){
        printf("\33[2K\r");     // Overwrites the previous text line
        cout << "    Downloading progress : " << totalBytes << " bytes out of " <<
fileSize << " ";
    }
    cout << "\r ";
    // Save to file
    write(outfile, buffer, readBytes);
    // Send confirmation msg
    send(fd, &msgCmf, 4, 0);
}
```

To print the download status the client keeps track of total no of bytes downloaded, and ideally keep printing it on the same line.

The results of this exercise can be watched in the attached TCPtest.wav video.

Exercise 7

UDP-Client/Server

In this exercise we develop a UDP Server that can be used to send data read from diffent files to a UDP client that asks for this. The programming language used in this exercise is C++.

Server app

A UDP server needs to have a socket which listens to incoming connections. Client will be sending telegrams towards the server requesting something, and then the server will send back a response.

In the code below you see the the "main" action of the server program.

```
int udp_serv::run_serv(){
bool done = false;
int recvBytes;

cout << "Starting server\n";
while(!done){
    recvBytes = recvfrom(udp_socket, &recvBuf, sizeof(recvBuf), 0, (sockaddr*)
&sother, &sother_len);
    if(recvBytes < 0){
        perror("error reading recv\n");
        done = true;
    }
    else{
        // Do something
        if(checkCmd()){
            std::cout << "Sent data\n";
```

```
                clearBufs();
            }
            else{
                std::cout << "Unrecognized command.\n";
            }
        }
    }
    return 0;
```

In this snippet below, you see the functonality for handling and responding to commands read from the incoming socket.

```
bool udp_serv::checkCmd(){
    std::cout << "cmd : " << recvBuf << "\n";
    for(int i = 0; i < 4; i++){
        if(strcoll(accCmd[i], recvBuf) == 0){
            if(accCmd[i] == "U" || accCmd[i] == "u"){
                cpyUptime();
                sendto(udp_socket, &sendBuf, sizeof(sendBuf), 0, (sockaddr*)
&sother, sother_len);
                return true;
            }
            else if(accCmd[i] == "L" || accCmd[i] == "l"){
                cpyLoadAvg();
                sendto(udp_socket, &sendBuf, sizeof(sendBuf), 0, (sockaddr*)
&sother, sother_len);
                return true;
            }
        }
    }
    return false;
}
```

Below is a code snippet showing one of the file reading functions used to complete the forementioned functionality

```
int udp_serv::cpyLoadAvg(){
    int fd = open("/proc/loadavg", O_RDONLY);
    int readBytes = read(fd, &sendBuf, sizeof(sendBuf));
    if(readBytes < 0){
        perror("Couldn't read from loadavg\n");
        return -1;
    }
    close(fd);
    return 0;
}
```

Client App

The client app is also very similar interms of the setup stuff, where again the socket is set to SOCK_DGRAM. Furthermore there is now no need to do the binding manualy, since the use of sendto() automatically binds the socket.

Results

the results of this exercise can be watched in the attached UDPtest.wav video.