Kandidatnummer: 168

DATA2410 - Reliable Transport Protocol (DRTP)

Innledning

Jeg har i denne oppgaven laget en filoverførings-applikasjon som sender «JPEG»-filer over transport-protokollen UDP. For å sikre pålitelighet (*reliability*) er det på toppen av UDP implementert en transport-nivå protokoll som kalles *DRTP* (DATA2410 Reliable Transport Protocol). Dette for å sikre at all dataen som sendes fra klienten kommer frem til serveren, og at denne dataen kommer frem i korrekt rekkefølge og uten duplikater.

UDP tilbyr ingen form for bekreftelse for at en tilkobling mellom sender og mottaker er vellykket, og det er derfor implementert en såkalt *three-way handshake* med ACK- og SYN-pakker for å sikre dette. På samme måte er det også implementert kommunikasjon for å «*gracefully*» lukke tilkoblingen ved hjelp av FIN- og ACK-pakker når filoverføringen er fullført.

For å sikre at dataen som sendes kommer frem, og at den kommer frem i korrekt rekkefølge og uten duplikater er det implementert en «Go-Back-N»-funksjon med en standard «sliding window» størrelse på tre pakker. Dersom sender ikke mottar en bekreftelse (ACK-pakke) på en pakke med data sendt innen en standard tidsfrist på 500ms er alle tidligere pakker å anse som tapt, og alle pakker uten bekreftelse sendes på nytt til mottaker. Dersom det mottas en bekreftelse (ACK-pakke) på en allerede bekreftet pakke med data forkastes denne. Dersom mottaker får en pakke med et sekvensnummer ute av den forventede rekkefølgen forkastes pakken.

Dokumentasjon

CLI-kommando

Applikasjonen jeg har laget er som sagt en filoverførings-applikasjon som sender filer fra en sender til en mottaker over UDP, men med en egenimplementert protokoll, DRTP, som sikrer pålitelig filoverføring på toppen av dette.

Applikasjonen kjøres direkte i et CLI med python/python3. Det er påkrevd å enten kjøre applikasjonen i server-modus eller klient-modus, dette bestemmes av opsjonene -c for klient-modus og -s for server-modus.

I tillegg finnes det andre valgfrie opsjoner som kan brukes. Opsjonen -i kan brukes i begge modus og lar brukeren bestemme en egendefinert IP-adresse serveren skal kjøre på, eller en IP-adresse klienten skal koble seg til på. Opsjonen -p kan også brukes i begge modus og bestemmer hvilken port serveren skal godta tilkoblinger på, eller hvilken port klienten skal koble seg til på.

Den valgfrie opsjonen -w for størrelse på «sliding window», som tar et heltall, brukes kun i klient-modus for å bestemme hvor mange pakker som samtidig skal sendes, hvor standard størrelse er 3. Opsjonen -f tar inn en filbane og er eksplisitt brukt i klient-modus for å spesifisere hvilken fil som skal overføres. Denne opsjonen er i utgangspunktet også valgfri, men applikasjonen har ingen nytte dersom dette ikke benyttes.

I server-modus kan i tillegg den valgfrie opsjonen -d brukes for å spesifisere at en pakke med det gitte sekvens-nummeret forkastes. Denne opsjonen kan være nyttig for å se hvordan applikasjonen håndterer pakke-tap på et nettverk som ellers ikke vil ha noe tap.

Eksempler på bruk av applikasjonen:

```
python3 application.py -c -i 127.0.0.1 -p 12000 -f iceland_safiqul.jpg -w 5
```

Klient-modus med spesifisert IP-adresse, port-nummer, filbane til fil som skal overføres og størrelse på vindu

```
python3 application.py -s -i 127.0.0.1 -p 12000 -d 10
```

Server-modus med spesifisert IP-adresse, portnummer og hvilken pakke som skal forkastes

application.py

Filen application.py styrer kjøringen av applikasjonen. Her blir blant annet CLI-kommandoene parset og registrert ved hjelp av argparse-modulen i Python:

```
def get_args():
    parser = argparse.ArgumentParser(description="application args")

    group = parser.add_mutually_exclusive_group(required=True)
    group.add_argument('-s', '--server', action='store_true', help='run as server')
    group.add_argument('-c', '--client', action='store_true', help='run as client')

    parser.add_argument('-i', '--ip', type=str, default='10.0.1.2', help='port to use')
    ...
```

I tillegg til dette gjøres det også i denne filen feilhåndtering for å sikre at den oppgitte IPv4-adressen og portnummeret er på riktig format og for å sikre at det ikke er konflikt mellom brukte opsjoner. Når applikasjonen skal kjøre i enten server- eller klient-modus blir den respektive koden kjørt ut ifra hvilket opsjoner som ble gitt i kommandoen.

For å sikre bedre lesbarhet og generell struktur i prosjektet har jeg valgt å legge koden for server- og klient-modus ut i hver sine filer. Her er den nødvendige logikken implementert for at hver av modusene skal kjøre som de skal.

client.py

I client.py har jeg en funksjon med navn client_mode(args) som tar inn argumentene fra argparse. I denne funksjonen opprettes socket-objektet og korrekt timeout blir satt for dette objektet. Deretter blir tilkobling, filoverføring og teardown håndtert i hver sine respektive funksjoner:

```
def client_mode(args):
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    # Sets the timeout for each socket-operation to 500ms
    client_socket.settimeout(0.5)

    establish_conn(client_socket, (args.ip, args.port))

    send_data(client_socket, (args.ip, args.port), args.file, args.window)

    close_conn(client_socket, (args.ip, args.port))
```

Tilkobling- og teardown-funksjonene er ganske rett frem og jeg spesifiserer derfor ikke dette nærmere her. Funksjonen som sender dataen, send_data(…) er derimot noe mer kompleks, og jeg skal gå litt mer i dybden på denne.

Først åpner vi den angitte filen og leser innholdet av den. Deretter defineres noen viktige variabler; da særlig base_seq for å ha kontroll på den tidligste pakken som enda ikke er bekreftet og next_seq_num for å vite hva sekvensnummeret til den neste pakken skal være. Det opprettes også en dictionary, som fungerer som en buffer, for å kunne lagre de pakkene som enda ikke er bekreftet, slik at de kan sendes på nytt dersom de går tapt.

Vi går så videre til en dobbel while-løkke som kontrollerer sending av pakkene:

```
while i < len(file_data):
        while next_seq_num <  base_seq + window_size:
                ...
```

Den ytterste løkken sikrer at vi sender pakker så lenge det fortsatt finnes data igjen i filen. Den innerste løkken passer derimot på at vi til enhver tid under sendingen ikke overskrider det angitte vinduet. Når vi går ut av den innerste løkken ønsker vi å motta en ACK-pakke fra serveren, slik at vi kan flytte vinduet vårt lenger.

```
        try:
            base_seq = recv_ack(client_socket, packets)
        except TimeoutError:
            handle_rto(client_socket, packets, addr)
```

Her gjøres det feilhåndtering for å ta høyde for eventuelt tap i nettverket. Funksjonen recv_ack(…) tar imot en ACK-pakke og flytter vinduet i henhold til hvilken pakke som blir bekreftet. For eksempel, om vi har vinduet {10, 11, 12} og mottar ACK for pakke 10 vil vinduet etter dette bli {11, 12, 13}. Om vi mottar ACK for pakke 11 i samme eksempel vil vinduet bli {12, 13, 14}. Dette er fordi om vi mottar ACK for en pakke lenger frem i vinduet, kan vi ta for gitt at også de tidligere pakkene er bekreftet, men hvor ACK-pakken fra serveren har gått tapt.

Til slutt passer vi på at vi mottar ACK-pakker for pakkene som er under transport før vi går ut av send_data(…) funksjonen:

```
while len(packets) > 0:
        try:
            base_seq = recv_ack(client_socket, packets)
        except TimeoutError:
            handle_rto(client_socket, packets, addr)
```

server.py

I server.py filen kjøres server_mode(args) funksjonen når applikasjonen skal kjøre i server-modus. Her opprettes det et socket-objekt som så binder seg til den angitte IP-adressen og det angitte portnummeret, slik at den kan ta imot tilkoblinger der.

Siden serveren skal avsluttes når filoverføringen er ferdig er det kun laget en enkelt while-løkke som går så lenge filoverføringen skjer. Logikken her er ganske simpel, da den kun tar imot pakker dersom sekvensnummeret stemmer overens med hva serveren forventer at neste sekvensnummer skal være. Dersom sekvensnummeret er lik det som ble spesifisert i kommandoen med opsjonen -d eller pakken er ute av rekkefølge gjøres ingenting med dataen som kommer, og programmet fortsetter.

Når serveren til slutt mottar en pakke med FIN-flagget satt vil den på nytt gå inn i handle_conn(…) funksjonen for å avslutte tilkoblingen. Først vil derimot dataen som er mottatt bli skrevet til en fil og throughput skrives ut i terminalen.

```
if flags & FIN:
                with open('file.jpg', 'wb') as f:
                    f.write(file_data)

                throughput = total_bytes / 1000 / 1000 / (time.time() - start_time)
                print(f'\nThe throughput is {round(throughput, 2)} Mbps\n')
                handle_conn(server_socket, addr, packet)
                break
```

utils.py og constants.py

For å unngå duplikat-kode i client.py og server.py la jeg funksjonene som håndterer ned- og oppakking av pakker i en felles fil utils.py. Denne filen importeres inne i de andre filene og benyttes som vanlig der. I tillegg har jeg lagget en fil constants.py som inneholder alle konstantene som brukes på begge sider av applikasjonen.

Diskusjon

1.

Med vindu-størrelse 3 og 100ms RTT får vi en throughput på 0.03Mbps.

Med vindu-størrelse 5 og 100ms RTT får vi en throughput på 0.05Mbps.

Med vindu-størrelse 10 og 100ms RTT får vi en throughput på 0.09Mbps.

Ut ifra disse resultatene kan det se ut til at overføringshastigheten (throughput) bare vil øke så lenge vi øker størrelsen på vinduet. Dette vil stemme helt frem til vi møter på punktet hvor nettverket ikke lenger klarer å håndtere all dataen som sendes, hvor vi da vil starte å miste flere og flere pakker og som igjen vil gjøre at klienten på retransmittere pakkene som har gått tapt. Dette vil da resultere i en lavere overføringshastighet, og vi ser da det må være en balanse i vindu-størrelsen for å best mulig optimalisere det nettverket dataen sendes over. Å kontrollere størrelsen på vinduet er en effektiv måte å kontrollere opphopningen (congestion) i nettverket, og i for eksempel TCP-protokollen er dette implementert som en del av protokollen, da en god del mer avansert enn hva som er gjort i denne oppgaven (Kurose & Ross, 2022, s. 293).

2.

Window-size | RTT	50ms	200ms
3	0.05 Mbps	< 0.01 Mbps
5	0.09 Mbps	0.02 Mbps
10	0.18 Mbps	0.05 Mbps

Tabell: Forhold mellom vindu-størrelse og RTT

RTT, eller *Round Trip Time,* er den tiden det tar for en pakke å fraktes til mottaker, og tilbake til senderen igjen (Kurose & Ross, 2022, s. 129). Her ser vi at en lavere RTT fører til en høyere overføringshastighet, og at en høyere RTT på samme måte fører til en lavere overføringshastighet. Dette er ganske naturlig da det ved en lav RTT vil ta kortere tid før klienten mottar bekreftelse på pakkene den har sendt, og at den dermed kan sende ut pakker hyppigere. Når RTT er høy vil det ta lengre tid før den får en respons på mottatt pakke, og må derfor vente lenger før de nye pakkene kan sendes. I en virkelig situasjon kan vi sammenligne dette med *propagation delay*, som er den forsinkelsen som er et resultat av den fysiske distansen pakkene må sendes og i hvilket medium de blir sendt i (Kurose & Ross, 2022, s.67).

3.

Vi kjører applikasjonen i server-modus med -d flagget satt til 6 for å overse pakke med sekvensnummer 6:

```
h1$ 'python3 application.py -c -i 10.0.1.2 -p 8080 -f iceland_safiqul.jpg -w 5'

SYN packet sent
SYN-ACK packet recieved
ACK packet sent
Connection established

Data transfer:

13:37:22.966853 -- packet with seq = 1 is sent, sliding window = {1}
13:37:22.966923 -- packet with seq = 2 is sent, sliding window = {1, 2}
13:37:22.966939 -- packet with seq = 3 is sent, sliding window = {1, 2, 3}
13:37:22.966963 -- packet with seq = 4 is sent, sliding window = {1, 2, 3, 4}
13:37:22.966977 -- packet with seq = 5 is sent, sliding window = {1, 2, 3, 4, 5}
13:37:23.068221 -- ACK for packet = 1 is recieved
13:37:23.068299 -- packet with seq = 6 is sent, sliding window = {2, 3, 4, 5, 6}
13:37:23.068317 -- ACK for packet = 2 is recieved
13:37:23.068489 -- packet with seq = 7 is sent, sliding window = {3, 4, 5, 6, 7}
13:37:23.068517 -- ACK for packet = 3 is recieved
13:37:23.068550 -- packet with seq = 8 is sent, sliding window = {4, 5, 6, 7, 8}
13:37:23.068563 -- ACK for packet = 4 is recieved
13:37:23.068585 -- packet with seq = 9 is sent, sliding window = {5, 6, 7, 8, 9}
13:37:23.068596 -- ACK for packet = 5 is recieved
13:37:23.068625 -- packet with seq = 10 is sent, sliding window = {6, 7, 8, 9, 10}
13:37:23.571098 -- RTO occured
13:37:23.571161 -- retransmitting packet with seq 6
13:37:23.571234 -- retransmitting packet with seq 7
13:37:23.571253 -- retransmitting packet with seq 8
13:37:23.571269 -- retransmitting packet with seq 9
13:37:23.571284 -- retransmitting packet with seq 10
13:37:23.672736 -- ACK for packet = 6 is recieved
13:37:23.672878 -- packet with seq = 11 is sent, sliding window = {7, 8, 9, 10, 11}
13:37:23.672897 -- ACK for packet = 7 is recieved
...
...
```

```
h2$ 'python3 application.py -s -i 10.0.1.2 -p 8080 -d 6'

Server listening on port 8080
New connection from ('10.0.0.1', 47561)
SYN packet recieved
SYN ACK packet sent
ACK packet recieved
Connection established

13:37:23.068013 -- packet 1 is recieved
13:37:23.068116 -- sending ACK for the recieved packet 1
13:37:23.068143 -- packet 2 is recieved
13:37:23.068185 -- sending ACK for the recieved packet 2
13:37:23.068197 -- packet 3 is recieved
13:37:23.068215 -- sending ACK for the recieved packet 3
13:37:23.068230 -- packet 4 is recieved
13:37:23.068248 -- sending ACK for the recieved packet 4
13:37:23.068257 -- packet 5 is recieved
13:37:23.068273 -- sending ACK for the recieved packet 5
13:37:23.169363 -- out of order packet 7 is recieved
13:37:23.169394 -- out of order packet 8 is recieved
13:37:23.169405 -- out of order packet 9 is recieved
13:37:23.169415 -- out of order packet 10 is recieved
13:37:23.671860 -- packet 6 is recieved
13:37:23.672269 -- sending ACK for the recieved packet 6
13:37:23.672300 -- packet 7 is recieved
13:37:23.672359 -- sending ACK for the recieved packet 7
...
...
```

Vi ser at serveren mottar pakke 5 og sender ACK for denne som normalt. Siden vi har valgt å overse pakke 6 vil pakkene som kommer etter være ute av rekkefølge og dermed forkastes.

På grunn av logikken som er implementert i koden vil det fremkomme en *TimeoutError* på klient-siden etter det har gått 500ms uten at det har vært noen aktivitet i nettverket. Den vil da anse alle pakker som enda ikke har mottatt noen ACK som tapt, og sender dermed disse på nytt.

4.

```
h1$ 'python3 application.py -c -i 10.0.1.2 -p 8080 -f iceland_safiqul.jpg -w 5'

SYN packet sent
SYN-ACK packet recieved
ACK packet sent
Connection established

Data transfer:

11:55:34.369603 -- packet with seq = 1 is sent, sliding window = {1}
11:55:34.369667 -- packet with seq = 2 is sent, sliding window = {1, 2}
11:55:34.369683 -- packet with seq = 3 is sent, sliding window = {1, 2, 3}
11:55:34.369707 -- packet with seq = 4 is sent, sliding window = {1, 2, 3, 4}
11:55:34.369720 -- packet with seq = 5 is sent, sliding window = {1, 2, 3, 4, 5}
11:55:34.479508 -- ACK for packet = 1 is recieved
11:55:34.479627 -- packet with seq = 6 is sent, sliding window = {2, 3, 4, 5, 6}
11:55:34.479649 -- ACK for packet = 2 is recieved
11:55:34.479667 -- packet with seq = 7 is sent, sliding window = {3, 4, 5, 6, 7}
11:55:34.479676 -- ACK for packet = 3 is recieved
11:55:34.479691 -- packet with seq = 8 is sent, sliding window = {4, 5, 6, 7, 8}
11:55:34.479698 -- ACK for packet = 4 is recieved
11:55:34.479711 -- packet with seq = 9 is sent, sliding window = {5, 6, 7, 8, 9}
11:55:34.479717 -- ACK for packet = 5 is recieved
11:55:34.479735 -- packet with seq = 10 is sent, sliding window = {6, 7, 8, 9, 10}
11:55:34.582938 -- ACK for packet = 6 is recieved
11:55:34.583052 -- packet with seq = 11 is sent, sliding window = {7, 8, 9, 10, 11}
11:55:34.583070 -- ACK for packet = 7 is recieved
11:55:34.583096 -- packet with seq = 12 is sent, sliding window = {8, 9, 10, 11, 12}
11:55:35.083755 -- RTO occured
11:55:35.083808 -- retransmitting packet with seq 8
11:55:35.083864 -- retransmitting packet with seq 9
11:55:35.083878 -- retransmitting packet with seq 10
11:55:35.083887 -- retransmitting packet with seq 11
11:55:35.083897 -- retransmitting packet with seq 12
11:55:35.184652 -- ACK for packet = 8 is recieved
11:55:35.184738 -- packet with seq = 13 is sent, sliding window = {9, 10, 11, 12, 13}
11:55:35.184757 -- ACK for packet = 9 is recieved
11:55:35.184784 -- packet with seq = 14 is sent, sliding window = {10, 11, 12, 13, 14}
11:55:35.184796 -- ACK for packet = 10 is recieved
11:55:35.184816 -- packet with seq = 15 is sent, sliding window = {11, 12, 13, 14, 15}
...
...
11:55:35.694996 -- packet with seq = 41 is sent, sliding window = {37, 38, 39, 40, 41}
11:55:35.695008 -- ACK for packet = 37 is recieved
11:55:35.695029 -- packet with seq = 42 is sent, sliding window = {38, 39, 40, 41, 42}
11:55:36.195259 -- RTO occured
11:55:36.195301 -- retransmitting packet with seq 38
11:55:36.195456 -- retransmitting packet with seq 39
11:55:36.195474 -- retransmitting packet with seq 40
11:55:36.195483 -- retransmitting packet with seq 41
11:55:36.195491 -- retransmitting packet with seq 42
11:55:36.297992 -- ACK for packet = 38 is recieved
11:55:36.298111 -- packet with seq = 43 is sent, sliding window = {39, 40, 41, 42, 43}
11:55:36.298130 -- ACK for packet = 39 is recieved
11:55:36.298157 -- packet with seq = 44 is sent, sliding window = {40, 41, 42, 43, 44}
11:55:36.298169 -- ACK for packet = 40 is recieved
...
...
```

```
h2$ 'python3 application.py -s -i 10.0.1.2 -p 8080'

Server listening on port 8080
New connection from ('10.0.0.1', 53070)
SYN packet recieved
SYN ACK packet sent
ACK packet recieved

Connection established

11:55:34.479112 -- packet 1 is recieved
11:55:34.479326 -- sending ACK for the recieved packet 1
11:55:34.479345 -- packet 2 is recieved
11:55:34.479380 -- sending ACK for the recieved packet 2
11:55:34.479388 -- packet 3 is recieved
11:55:34.479399 -- sending ACK for the recieved packet 3
11:55:34.479412 -- packet 4 is recieved
11:55:34.479424 -- sending ACK for the recieved packet 4
11:55:34.479431 -- packet 5 is recieved
11:55:34.479441 -- sending ACK for the recieved packet 5
11:55:34.582332 -- packet 6 is recieved
11:55:34.582734 -- sending ACK for the recieved packet 6
11:55:34.582790 -- packet 7 is recieved
11:55:34.582845 -- sending ACK for the recieved packet 7
11:55:34.582866 -- out of order packet 9 is recieved
11:55:34.582876 -- out of order packet 10 is recieved
11:55:34.686702 -- out of order packet 11 is recieved
11:55:34.687110 -- out of order packet 12 is recieved
11:55:35.184281 -- packet 8 is recieved
11:55:35.184470 -- sending ACK for the recieved packet 8
11:55:35.184514 -- packet 9 is recieved
11:55:35.184545 -- sending ACK for the recieved packet 9
...
...
11:55:35.694483 -- sending ACK for the recieved packet 35
11:55:35.694492 -- packet 36 is recieved
11:55:35.694510 -- sending ACK for the recieved packet 36
11:55:35.694519 -- packet 37 is recieved
11:55:35.694537 -- sending ACK for the recieved packet 37
11:55:35.795954 -- out of order packet 39 is recieved
11:55:35.796015 -- out of order packet 40 is recieved
11:55:35.796025 -- out of order packet 41 is recieved
11:55:35.796251 -- out of order packet 42 is recieved
11:55:36.297394 -- packet 38 is recieved
11:55:36.297676 -- sending ACK for the recieved packet 38
11:55:36.297707 -- packet 39 is recieved
11:55:36.297750 -- sending ACK for the recieved packet 39
...
...
```

Utdrag av resultat med 2% loss rate

```
...
...
12:53:55.880824 -- packet with seq = 1388 is sent, sliding window = {1384, 1385, 1386, 1387, 1388}
12:53:55.881536 -- ACK for packet = 1384 is recieved
12:53:55.881568 -- packet with seq = 1389 is sent, sliding window = {1385, 1386, 1387, 1388, 1389}
12:53:56.385439 -- RTO occured
12:53:56.385497 -- retransmitting packet with seq 1385
12:53:56.385571 -- retransmitting packet with seq 1386
12:53:56.385591 -- retransmitting packet with seq 1387
12:53:56.385606 -- retransmitting packet with seq 1388
12:53:56.385620 -- retransmitting packet with seq 1389
12:53:56.919776 -- RTO occured
12:53:56.919829 -- retransmitting packet with seq 1385
12:53:56.919904 -- retransmitting packet with seq 1386
12:53:56.919925 -- retransmitting packet with seq 1387
12:53:56.919940 -- retransmitting packet with seq 1388
12:53:56.919954 -- retransmitting packet with seq 1389
12:53:57.433963 -- RTO occured
12:53:57.434024 -- retransmitting packet with seq 1385
12:53:57.434100 -- retransmitting packet with seq 1386
12:53:57.434127 -- retransmitting packet with seq 1387
12:53:57.434145 -- retransmitting packet with seq 1388
12:53:57.434161 -- retransmitting packet with seq 1389
12:53:57.935946 -- RTO occured
12:53:57.936001 -- retransmitting packet with seq 1385
12:53:57.936075 -- retransmitting packet with seq 1386
12:53:57.936097 -- retransmitting packet with seq 1387
12:53:57.936110 -- retransmitting packet with seq 1388
12:53:57.936120 -- retransmitting packet with seq 1389
12:53:58.046118 -- ACK for packet = 1385 is recieved
12:53:58.046208 -- packet with seq = 1390 is sent, sliding window = {1386, 1387, 1388, 1389, 1390}
12:53:58.047193 -- ACK for packet = 1386 is recieved
12:53:58.047229 -- packet with seq = 1391 is sent, sliding window = {1387, 1388, 1389, 1390, 1391}
12:53:58.048379 -- ACK for packet = 1387 is recieved
...
...
```

```
...
...
12:53:55.879974 -- packet 1383 is recieved
12:53:55.880752 -- sending ACK for the recieved packet 1383
12:53:55.880769 -- packet 1384 is recieved
12:53:55.881512 -- sending ACK for the recieved packet 1384
12:53:55.983024 -- out of order packet 1386 is recieved
12:53:55.983080 -- out of order packet 1387 is recieved
12:53:55.983093 -- out of order packet 1388 is recieved
12:53:55.983106 -- out of order packet 1389 is recieved
12:53:56.493159 -- out of order packet 1386 is recieved
12:53:56.493216 -- out of order packet 1387 is recieved
12:53:56.493226 -- out of order packet 1388 is recieved
12:53:56.493235 -- out of order packet 1389 is recieved
12:53:57.020442 -- out of order packet 1386 is recieved
12:53:57.020500 -- out of order packet 1387 is recieved
12:53:57.020511 -- out of order packet 1388 is recieved
12:53:57.020519 -- out of order packet 1389 is recieved
12:53:57.541873 -- out of order packet 1386 is recieved
12:53:57.541933 -- out of order packet 1387 is recieved
12:53:57.541944 -- out of order packet 1388 is recieved
12:53:57.541955 -- out of order packet 1389 is recieved
12:53:58.044299 -- packet 1385 is recieved
12:53:58.046040 -- sending ACK for the recieved packet 1385
12:53:58.046071 -- packet 1386 is recieved
12:53:58.047184 -- sending ACK for the recieved packet 1386
...
...
```

Utdrag av resultat med 5% loss rate:

Den tydeligste forskjellen mellom å kjøre applikasjon med 2% loss rate og 5% er naturligvis antall forekomster av RTO (*Recovery Time Objective)*. Dette skjer hver gang det har gått 500ms uten at det har vært noen aktivitet på nettverket (sending eller mottak av pakker).

Selv om løsningen som er implementert i denne oppgaven gjør det den skal, sørger for at pakker kommer frem og at de kommer frem i korrekt rekkefølge, er den ikke spesielt effektiv. Vi kan øke effektiviteten ved å sette et vindu på en størrelse som på best mulig måte tar nytte av den båndbredden som er tilgjengelig i et gitt nettverk, men det finnes andre aspekter som gjør at koden uansett ikke er optimal. For eksempel så er det ineffektivt for en mottaker å bare forkaste alle pakker som er ute av rekkefølge, kontra å ha en buffer med pakker som kan brukes til å komplettere etter hvert som korrekte pakker ankommer. I den forbindelse har vi også et problem på sender-siden, som ved en *TimeoutError* sender alle pakker i det gjeldende vinduet på nytt. Om hyppigheten av feil øker, og vinduet er av en betydelig størrelse, vil dette gjør at datapipelinen blir tungt belastet av alle disse retransmitterte pakkene. Dette er et av problemene ved å benytte seg av en Go-Back-N funksjon (Kurose & Ross, 2022, s. 250). Om denne løsningen skulle blitt videre utviklet for å bli av produksjons-kvalitet hadde det vært aktuelt å implementere mer avanserte og situasjonsavhengige løsninger for blant annet *congestion-control* og generell behandling av data på både klient- og server-siden.

Litteraturliste

Kurose, J. F. & Ross, K. W. (2022). *Computer Networking: A Top-Down Approach*. Pearson Education Limited.