



# DATA2410 Datanettverk og skytjenester

## DRPT/UDP Hjemme eksamen

Kandidatnummer: **311**  
Emnekode: DATA2410  
Emnenavn: Datanettverk og skytjenester  
Studieprogram: Dataingeniør  
Antall sider: 14  
Innleveringsfrist: 21.05.2024 kl. 12:00

OSLO METROPOLITAN UNIVERSITY  
STORBYUNIVERSITETET

## Introduksjon:

Dette prosjektet handler om å implementere en enkel protokoll for pålitelig dataoverføring med navn «DATA2410 Reliable Transport Protocol (DRTP)». Denne protokollen er designet for å sikre pålitelig og ordnet levering av data, uten manglende eller dupliserte pakker, på tvers av et nettverk. DRTP er bygget på toppen av UDP (User Datagram Protocol), en enkel, men mindre pålitelig transportprotokoll.

DRTP bruker Go-Back-N (GBN) metoden for å håndtere pakkeoverføringen. GBN er en viktig del av pålitelighetsfunksjonene som er implementert i denne applikasjonen. Ved hjelp av en skyvevindumekanisme tillater GBN flere pakker å være «i flyt» samtidig, noe som forbedrer effektiviteten av dataoverføringen over nettverket. Hvis en pakke går tapt eller kommer ut av rekkefølge, vil GBN metoden sørge for at pakken blir sendt på nytt.

I dette prosjektet er det utviklet et Python-program kalt «application.py», som håndterer filoverføring mellom to nettverksnoder. Programmet kan kjøre enten som server eller klient, alt etter behov. Under kjøring tar den flere argumenter, inkludert IP-adresse, portnummer, filsti, vindusstørrelse, og et testtilfelle for å hoppe over et sekvensnummer. Brukeren skal oppgi filstien til en jpg eller jpeg fil som klienten skal overføre til server via pakker. Serveren mottar disse pakkene og danner en ny jpg fil som skal være identisk med den opprinnelige filen klienten sendte.

Hele overføringsprosessen i DRTP er delt opp i tre hovedfaser: forbindelsesoppbygging, dataoverføring og forbindelsesavslutning. Forbindelsesoppbyggingen skjer ved hjelp av en SYN-SYN-ACK-ACK håndhilsing-prosedyre, mens avslutning av forbindelsen håndteres med en FIN-FIN-ACK prosedyre. Dette ligner svært mye på hvordan transportprotokollen TCP (Transmission Control Protocol) fungerer. Under dataoverføringen, bruker programmet teknikker som skyvevindu og retransmisjon for å sikre pålitelig overføring. Programmet håndterer også scenarier som pakketap, tidsavbrudd og pakker som kommer ut av rekkefølge.

## Implementasjon:

Programmet «application.py» kan startes og kjøres ordentlig ved at brukeren skriver inn de nødvendige kontrollinjeargumentene, sjekk README.md for mer grundig informasjon om hvordan dette kan gjøres.

```
application.py X
praksis_eksamen > application.py ...
567 def main():
568     # Kaller på funksjonen `parse_arguments` som returnerer argumentene som er spesifisert når programmet kjøres
569     args = parse_arguments()
570
571     # Sjekker om servermodus er spesifisert
572     if args.server:
573         # Oppretter en serverinstans med de gitte argumentene
574         server = Server(args.ip, args.port, args.file, args.window, args.discard)
575         # Starter serveren
576         server.start()
577     # Sjekker om klientmodus er spesifisert
578     elif args.client:
579         # Oppretter en klientinstans med de gitte argumentene
580         client = Client(args.ip, args.port, args.file, args.window, args.discard)
581         # Starter klienten
582         client.connect()
583     else:
584         # Hvis verken server- eller klientmodus er spesifisert, skrives en feilmelding ut og programmet avsluttes
585         print("Feil: Vennligst spesifiser enten servermodus (-s) eller klientmodus (-c).")
586         sys.exit(1)
587
588     # Sjekker om dette scriptet ble kjørt direkte (i stedet for å bli importert som en modul)
589     if __name__ == "__main__":
590         # Hvis scriptet ble kjørt direkte, kaller den på hovedfunksjonen
591         main()
592
593
```

Dersom kontrollinjeargumentene stemmer starter programmet «application.py», og det første som skjer når koden starter er at den sjekker om scriptet ble kjørt direkte og ikke importert som en modul. Dette sjekkes av if-setningen på linje 589. Dersom scriptet ble kjørt direkte, blir hovedfunksjonen main() kalt på og startet. Hovedfunksjonen til programmet, main(), kaller på en annen funksjon parse\_arguments(), som analyserer kommandolinjeargumentene oppgitt da «application.py» ble startet. Nedenfor kan du se hvordan funksjonen parse\_arguments() ser ut.

```
# Beskrivelse av funksjonen:
# Funksjon for å analysere kommandolinjeargumenter
# Funksjonen gjør:
# Definerer hva slags argumenter som kan tas inn fra kommandolinjen når programmet kjøres
# Argumenter:
# Ingen argumenter trengs for denne funksjonen siden den bruker argparse biblioteket for å håndtere kommandolinjeargumenter
# Retur: Returnerer argumentene som ble spesifisert når programmet kjørte
def parse_arguments():
    parser = argparse.ArgumentParser(description="DRTP filoverføringsapplikasjon")
    group = parser.add_mutually_exclusive_group(required=True)
    group.add_argument('-s', '--server', action='store_true', help="Aktiver servermodus")
    group.add_argument('-c', '--client', action='store_true', help="Aktiver klientmodus")
    parser.add_argument('-i', '--ip', type=valid_ip, default='127.0.0.1', help="IP-adressen til serveren (standard: 127.0.0.1)")
    parser.add_argument('-p', '--port', type=valid_port, default=8080, help="Portnummer (standard: 8080)")
    parser.add_argument('-f', '--file', type=valid_file, help="Filbane")
    parser.add_argument('-w', '--window', type=valid_window_size, default=3, help="Størrelsen på skyvevinduet (standard: 3)")
    parser.add_argument('-d', '--discard', type=int, help="Tilpasset testtilfelle for å hoppe over et sekvensnummer")
    return parser.parse_args()
```

Funksjonen legger alle argumentene oppgitt i kontrollinjen i «parser» variabelen som videre analyserer og sjekker om alt er som det skal være. Argumentet -c for klient og -s for server legges i en egen variabel «group» som videre blir parset slik at programmet sørger for at kun en av argumentene er valgt. Dette sørger for at programmet validerer kontrollinjeargumentene og kaster en feilmelding dersom både -c og -s blir oppgitt samtidig som kontrollinjeargumenter. En annen ting som er verdt å bemerke er at ip, port, filsti og vindu blir også validert med hver sin egen funksjon. Navnet på de funksjonene som skal validere de ulike kontrollinjeargumentene står i «type» variabelen for hvert argument. Siden ip og port må være oppgitt for at programmet skal kjøre, blir hver av disse sendt til funksjonene valid\_ip for ip og valid\_port for port.

```
19 def valid_ip(ip):
20     try:
21         ipaddress.ip_address(ip)
22         return ip
23     except ValueError:
24         raise argparse.ArgumentTypeError(f"Ugyldig IP adresse: {ip}")
25
```

```
34 def valid_port(port):
35     try:
36         port = int(port)
37         if 0 <= port <= 65535:
38             return port
39         else:
40             raise argparse.ArgumentTypeError(f"Ugyldig portnummer: {port}. Må være mellom 0 og 65535.")
41     except ValueError:
42         raise argparse.ArgumentTypeError(f"Ugyldig portnummer: {port}. Må være et tall mellom 0 og 65535.")
43
```

Ovenfor ser du en bildesnutt av funksjonene valid\_ip og valid\_port. Disse funksjonene tar inn ip- og portadressen gitt i kontrollinjeargumentene og sjekker om de er skrevet på riktig måte og om de er gyldige eller ikke. Dersom ip og/eller port ikke stemmer vil funksjonen returnere en feilmelding som brukeren mottar.

På samme måte som funksjonene valid\_ip og valid\_port, er funksjonene valid\_file og valid\_window\_size implementert i «application.py». Disse funksjonene har samme oppgave som de andre valideringsfunksjonene. valid\_file sjekker om filstien finnes og om filen er av riktig filtype, mens valid\_window\_size sørger for at dersom en vindu-størrelse er oppgitt, skal størrelsen være et positivt tall.

Nå som alle kontrollinjeargumentene er analysert og validert sjekker main() funksjonen med if-setninger om enten servermodus eller klientmodus er spesifisert.

```
application.py X application_reserve.py
praksis_eksamen > application.py > ...
566 # Hvis verken server- eller klientmodus er spesifisert, skriver funksjonen ut en feilmelding og avslutter programmet med en feilsetning
567 def main():
568     # Kaller på funksjonen 'parse_arguments' som returnerer argumentene som er spesifisert når programmet kjøres
569     args = parse_arguments()
570
571     # Sjekker om servermodus er spesifisert
572     if args.server:
573         # Oppretter en serverinstans med de gitte argumentene
574         server = Server(args.ip, args.port, args.file, args.window, args.discard)
575         # Starter serveren
576         server.start()
577     # Sjekker om klientmodus er spesifisert
578     elif args.client:
579         # Oppretter en klientinstans med de gitte argumentene
580         client = Client(args.ip, args.port, args.file, args.window, args.discard)
581         # Starter klienten
582         client.connect()
583     else:
584         # Hvis verken server- eller klientmodus er spesifisert, skrives en feilmelding ut og programmet avsluttes
585         print("Feil: Vennligst spesifiser enten servermodus (-s) eller klientmodus (-c).")
586         sys.exit(1)
587
```

Deretter oppretter funksjonen main() en instans basert på om enten server eller klient er valgt, og lagrer argumentene som trengs fra args i den såkalte instansen. Til slutt startes enten server eller klient avhengig av hvilken modus brukeren oppga som kontrollinjeargument.

La oss anta at brukeren har valgt å kjøre programmet i servermodus. Da vet vi at main() funksjonen oppretter en serverinstans med de nødvendige argumentene som trengs. Jeg har valgt å dele server og klient koden i hver sine egne klasser. Dette har jeg gjort for å kunne holde koden ryddig og oversiktlig, samtidig at det gir mulighet for å holde alle server funksjonene sammen og alle klient funksjonene sammen. Det gir også muligheten til å ekspandere koden på en forståelig måte dersom det trengs. På neste side kan du se et skjermskudd av klassen Server som inneholder en konstruktør def \_\_init\_\_. Konstruktøren tar argumentene/inngangsparameterne gitt fra funksjonen main() og initialiserer variablene for klassen. Samtidig validerer konstruktøren «file» og «window\_size» hvis de er oppgitt av brukeren i servermodus. Serveren skal ikke ta imot en filsti og en vinudu-størrelse, det er noe som skal oppgis i klientmodus.

Den siste setningen i konstruktøren kaller på funksjonen create\_socket() som oppretter en UDP socket og binder socketen til ip- og portadressen oppgitt av brukeren som argumenter. Funksjonen håndterer feil ved opprettelse av socket ved å lukke socketen, stoppe programmet og si ifra til brukeren. Neste funksjon i koden er start() som blir kalt på i main() funksjonen.

```
def start(self):
    # Starter serveren og venter på SYN-pakken
    print(f'\nServeren kjører med IP-adresse = {self.ip} og portadresse = {self.port}\n')
    #Kjører neste funksjon
    self.receive_syn_packet()
```

Denne funksjonen printer ut en setning til brukeren som bekrefter at serveren har startet, uten noen slags error fra validering av argumentene oppgitt i kontrollinjeargumentene. Funksjonen receive\_syn\_packet blir kalt og nå venter serveren på en SYN pakke fra klient slik at en kobling mellom server og klient kan dannes.

```

126 class Server:
141     def __init__(self, ip, port, file, window_size, discard):
142         # Validerer inngangsparameterne
143         if file:
144             print("-s valget kan ikke ta -f argument.")
145             sys.exit(1)
146         if window_size != 3:
147             print("Kun klient (-c) kan endre vindusstørrelsen")
148             sys.exit(1)

```

Bilde av serverklassen og konstruktøren def \_\_init\_\_

Når serveren mottar en SYN-pakke fra klient, vil funksjonen «recieve\_syn\_packet» håndtere og behandle dette.

```

200 def receive_syn_packet(self):
201     # Venter på og behandler en innkommende SYN-pakke
202     try:
203         packet, address = self.sock.recvfrom(1000)
204         seq, ack, flags = parse_header(packet)
205         if flags == SYN:
206             print("SYN-pakke mottatt")
207             #Kjører neste funksjon
208             self.send_syn_ack(seq, address)
209     #Feilhåndtering
210     except socket.timeout:
211         print("Socket timeout oppstod under venting på SYN-pakke.")
212     except Exception as e:
213         print("Feil ved mottakelse av SYN-pakke:", e)
214

```

Ovenfor ser vi at funksjonen tolker pakkehodet(header) til den mottatte pakken ved hjelp av funksjonen «parse\_header». Dersom flagget er satt til SYN, skrives det ut en setning og videre kalles funksjonen «send\_syn\_ack» som skal sende en SYN-ACK pakke til klienten. Dersom det oppstår en timeout eller feil under behandlingen av SYN-pakken, blir dette håndtert av funksjonen.

```

225 def send_syn_ack(self, seq, address):
226     # Sender en SYN-ACK-pakke tilbake til klienten
227     self.sock.sendto(create_packet(0, seq+1, SYN | ACK), address)
228     print("SYN-ACK-pakke sendt")
229     #Kjører neste funksjon
230     self.receive_ack()
231

```

Funksjonen «send\_syn\_ack» tar imot sekvensnummeret og adressen til klienten, og lager en pakke med flagget satt til SYN | ACK. Denne pakken sendes til klienten og en setning for å bekrefte dette skrives til brukeren. Til slutt kalles funksjonen «receive\_ack». Denne funksjonen ligner veldig på funksjonen «recieve\_syn\_packet», kun at if-setningen sjekker om pakken er lik ACK istedenfor SYN. Hvis dette stemmer sendes det en bekreftelsessetning til brukeren og funksjonen «receive\_data\_packets» blir kalt. Dersom det oppstår en timeout eller feil under behandlingen av ACK-pakken, blir dette håndtert av funksjonen.



```

126 class Server:
265
266     def receive_data_packets(self, address, discard):
267         # Får stien til den nåværende mappen
268         current_dir = os.path.dirname(os.path.abspath(__file__))
269         # Legger til filnavnet til stien
270         filename = os.path.join(current_dir, 'Photo_received.jpg')
271         # Oppretter en ny fil i den nåværende mappen for å lagre mottatt data
272         file = open(filename, 'wb')
273         # Initialiserer variabler for å beregne gjennomstrømningen
274         start_time = time.time()
275         total_bytes = 0
276         # Forventet sekvensnummer
277         expected_seq = 1
278         # Siste sekvens skrevet til filen
279         last_written_seq = 0
280         discard_done = False

```

Ovenfor ser du starten av funksjonen «receive\_data\_packets». Først finner funksjonen filstien til den nåværende mappen, slik at den kan lagre den nye filen «Photo\_received.jpg» på riktig sted. Deretter åpnes denne filen i den riktige plasseringen med fil-modusen «wb». W betyr at filen er åpen for skriving, og b betyr at dataene skal skrives i binærmodus. Binærmodus passer i dette tilfellet siden det brukes for eksempel når du arbeider med ting som bilder eller utførbare filer. Deretter initialiseres det variabler for å blant annet beregne gjennomstrømningen, forventede sekvensnummere, siste sekvens skrevet og en boolsk variabel som endrer sannhetsverdi til true dersom det blir utført en forkastning av pakke.

```

281         # Hovedserverløkke
282         while True:
283             try:
284                 # Mottar data fra socket, henter avsenderens adresse og parserer pakkeheaderen for sekvensnummer, bekreftelsesflagg og flagg
285                 data, address = self.sock.recvfrom(1000)
286                 seq, ack, flags = parse_header(data[:6])
287

```

Ovenfor kan du se koden «data, address = self.sock.recvfrom(1000)» som mottar data fra gjennom socketen «self.sock», med en bufferstørrelse på 1000 bytes, og lagrer de innkommende dataene og avsenderens adresse i «data» og «address» variablene. Deretter kaller koden «seq, ack, flags = parse\_header(data[:6])» funksjonen «parse\_header» med de første 6 bytes av de mottatte dataene, som representerer pakkeheaderen, for å trekke ut sekvensnummeret, bekreftelsesnummeret og flaggene for å håndtere pålitelig dataoverføring.

```

# Hvis sekvensnummeret er det vi forventer
if seq == expected_seq:
    # Hvis sekvensnummeret er lik self.discard og vi ikke allerede har forkastet en pakke
    if seq == self.discard and not discard_done:
        # Informer om at pakken er forkastet
        print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- Pakke {seq} ble forkastet")
        discard_done = True # Oppdater flagget for å indikere at vi har forkastet en pakke

        continue # Fortsett til neste iterasjon av løkken uten å behandle den forkastede pakken

# Hvis sekvensnummeret ikke er lik self.discard, behandles pakken som vanlig
print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- Pakke {seq} er mottatt")
file.write(data[6:])
total_bytes += len(data) - 6
last_written_seq = seq # Oppdaterer den siste sekvens som ble skrevet
expected_seq += 1

print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- Sender ack for den mottatte {last_written_seq}")
self.sock.sendto(create_packet(0, expected_seq, ACK), address)

# Hvis pakken har et høyere sekvensnummer enn forventet
elif seq > expected_seq:
    print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- Uordnet pakke {seq} er mottatt")
    continue # Fortsett til neste iterasjon av løkken uten å behandle den ut-av-rekkefølge-pakken

```

Dette er hoveddelen av serveren der pakkene som blir sendt av klienten mottas og håndteres. Dersom sekvensnummeret stemmer skrives det ut en setning som forteller tiden og at serveren har mottatt pakken med det forventede sekvensnummeret. Deretter skrives denne pakken i filen «Photo\_received.jpg», og sender en ACK for den mottatte pakken til klienten slik at klienten får verifikasjon. Slik kan både klient og server holde styr på pakkene og klient kan da sende neste pakke. Dersom sekvensnummeret er lik forkastningsverdien oppgitt av brukeren og den boolske variabelen «discard\_done» er satt til verdien false, forkastes denne pakken. Da vil den boolske variabelen endres til true. Siden en pakke blir forkastet, vil serveren motta pakker som ikke er i den forventede ordnede rekkefølgen. Dette blir brukeren varslet om, før serveren fortsetter til neste iterasjon av løkken for å motta den forkastede pakken på nytt. Det er klienten som håndterer den fortapte pakken.

```

        if flags == FIN:
            print("\nFIN-pakke mottatt")
            #Kjører neste funksjon
            self.send_fin_ack(seq, address)
            break
        except socket.timeout:
            print("Socket timeout oppstod under venting på en pakke.")
        except Exception as e:
            print("Feil ved mottakelse av pakke:", e)
            sys.exit(1)
# Beregner og skriver ut gjennomstrømningen
throughput = total_bytes / (time.time() - start_time) * 8 / 1e6
print(f"Gjennomstrømningen er {throughput:.2f} Mbps")
# Lukker tilkoblingen og avslutter programmet
self.sock.close()
print("Forbindelsen er avsluttet")
sys.exit(1)

```

Når pakkeoverføringen er ferdig, vil serveren motta en FIN-pakke fra klienten. Når dette skjer, sender serveren en FIN-ACK pakke med funksjonen «send\_fin\_ack». Dette er slik en kobling avsluttes med transportprotokollen TCP. Det er lagt til validering try-metoden dersom det oppstår en timeout eller en feil under mottak av pakker. Til slutt skrives gjennomstrømningen i megabytes per sekund og en setning som bekrefter at forbindelsen er avsluttet til brukeren.

```

362 def __init__(self, ip, port, file, window_size, discard):
363     # Sjekker om filbanen er gitt
364     if not file:
365         print("Feil: Filbane er nødvendig i klientmodus.")
366         sys.exit(1)
367     # Sjekker om discard-argumentet er gitt
368     if discard:
369         print("Feil: -c valget kan ikke ta -d argumentet.")
370         sys.exit(1)
371     # Setter objektvariablene
372     self.ip = ip
373     self.port = port
374     self.file = file
375     self.window_size = window_size
376     self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
377     self.sock.settimeout(0.5)

```

Dette er konstruktøren «\_\_init\_\_» til klassen klient. Konstruktøren tar argumentene/inngangsparameterne gitt fra funksjonen main() og initialiserer variablene for klassen. Samtidig validerer konstruktøren «file» og «discard». Filstien må være oppgitt i klientmodus siden «application.py» skal sende over en fil fra klient til server. I klientmodus skal ikke brukeren oppgi -d (forkast argumentet) siden pakken skal forkastes av serveren og ikke klienten. Som et argument for klienten kan brukeren endre vindusstørrelse dersom den ønsker det, ellers er den standard størrelsen på 3. Konstruktøren lager en UDP socket og lagrer det i den

initialiserte variablene «self.sock». Her blir også timeout verdien satt til 500 millisekunder, som beskrevet i oppgaveteksten.

```
386     def connect(self):
387         # Prøver å koble til serveren
388         try:
389             self.sock.connect((self.ip, self.port))
390             print("\nFase for etablering av forbindelse:\n")
391             # Sender en SYN-pakke til serveren for å initiere forbindelse
392             self.send_syn_packet()
393         except ConnectionRefusedError:
394             print(f"Feil: Tilkobling nektet. Serveren er ikke tilgjengelig på {self.ip}:{self.port}")
395             self.sock.close()
396         except Exception as e:
397             print("Feil ved oppretting av klient socket:", e)
398             self.sock.close()
399
```

Dette er funksjonen «connect()» som blir kalt på fra main() funksjonen etter at den har opprettet en klientinstans med argumentene fra args. Funksjonen «connect()» kobler socketen til den ip- og portadressen oppgitt av brukeren, som må være det samme som den serveren kjører på for at programmet skal fungere. Dersom ip- og portadressen stemmer, kalles det på funksjonen send\_syn\_packet(). Dersom socketen ikke får tilgang til serveren eller at en feil oppstår, skrives det ut en feilmelding til brukeren med nødvendig informasjon.

```
407     def send_syn_packet(self):
408         self.sock.sendto(create_packet(0, 0, SYN), (self.ip, self.port))
409         print("SYN-pakke er sendt")
410         # Mottar en SYN-ACK-pakke fra serveren
411         self.receive_syn_ack()
412
```

Ovenfor ser du funksjonen «send\_syn\_packet()» som blir kalt på av «connect()» dersom alt går som planlagt. Denne funksjonen lager en pakke med flagget SYN og sender den til serveren på riktig ip- og portadresse. Det blir også skrevet ut en bekreftelse på dette til brukeren.

Til slutt kalles funksjonen «receive\_syn\_ack()» for videre håndtering. «Receive\_syn\_ack()» sjekker om flagget til den mottatte pakken fra klienten er satt til SYN-ACK. Dersom dette stemmer kalles funksjonen «send\_ack\_packet», og hvis noe går galt med håndteringen av SYN-ACK pakken varsles brukeren om dette. Funksjonen «send\_ack\_packet» sender en ACK pakke til serveren og bekrefter til brukeren at en forbindelse er dannet. Oppbyggingen til denne funksjonen er ganske lik «send\_syn\_packet», bare at flagget er satt til ACK denne gangen. Nå som håndhilsing-fasen er etablert kalles funksjonen «transfer\_file», som er hovedfunksjonen til klient klassen.

```
458     def transfer_file(self):
459         print("Dataoverføring:\n")
460         # Initialiserer skyvevinduet og sekvensnummeret
461         window = []
462         seq = 1
463         sent_time = {} # Holder styr på når hver pakke ble sendt
464         packet_data = {} # Lagrer data for hver pakke
465         self.timeout = 0.5 # Setter timeout-verdien
466
```

Her er starten av funksjonen «transfer\_file». Her kan vi se at det opprettes en array for vindu variabelen. Dette er fordi koden har implementert Go-Back-N strategien der vi bruker en vindusstørrelse på 3. Det initialiseres også andre variabler som for eksempel seq, sent\_time, packet\_data og self.timeout.



```

458 :f transfer_file(self):
459     # Åpner filen og leser dataene
460     with open(self.file, 'rb') as f:
461         while True:
462             # Hvis vinduet ikke er fullt, leses mer data fra filen og sendes
463             while len(window) < self.window_size:
464                 data = f.read(994)
465                 if not data:
466                     break
467                 self.sock.sendto(create_packet(seq, 0, 0) + data, (self.ip, self.port))
468                 sent_time[seq] = time.time() # Lagrer tiden pakken ble sendt
469                 packet_data[seq] = data # Lagrer data for pakken
470                 window.append(seq)
471                 print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- Pakke med sekvensnummer = {seq} er sendt, skyvevindu = {window}")
472                 seq += 1

```

Videre åpner funksjonen filen som skal sendes i lesemodus, som er spesifisert med «rb». Så lenge vinduet er mindre enn størrelsen satt for vinduet, legges det inn flere sekvensnummere i vinduet. Når vinduet har 3 sekvensnummere i seg vil den starte å oppdatere ettersom klienten sender pakken og mottar ACK fra server med det samme sekvensnummeret. Klienten skriver ut når en pakke ble sendt med pakkens sekvensnummer. Samtidig oppdateres vinduet.

```

if not window:
    print("Dataoverføring fullført\n")
    print("Nedbryting av forbindelse:\n")
    self.send_fin_packet()
    break

```

Det sjekkes også om vinduet er tomt, og dersom vinduet står tomt etter pakkeoverføringen, antar koden at dataoverføringen er fullført og iverksetter prosessen for nedbryting av forbindelsen. Da kalles det på funksjonen «send\_fin\_packet()».

```

487 try:
488     # Venter på en ACK-pakke fra serverens
489     data, server = self.sock.recvfrom(1000)
490     _, ack, flags = parse_header(data[:6])
491     if flags == ACK:
492         # Hvis en ACK-pakke mottas, fjernes det tilsvarende sekvensnummeret fra vinduet
493         if ack - 1 in window:
494             print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- ACK for pakke = {ack - 1} er mottatt")
495             window.remove(ack - 1)
496     except socket.timeout:
497         print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- Timeout oppstod, sender alle pakker i vinduet på nytt")
498         print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- RTO oppstod")
499         for seq_num in window:
500             # Sjekker om det har gått nok tid for en retransmisjon
501             if time.time() - sent_time[seq_num] > self.timeout:
502                 print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- Sender pakke med sekvensnummer = {seq_num} på nytt")
503                 self.sock.sendto(create_packet(seq_num, 0, 0) + packet_data[seq_num], (self.ip, self.port)) # Sender korrekt data på nytt
504                 sent_time[seq_num] = time.time() # Oppdaterer sendetiden for pakken
505

```

Når klienten sender pakker med unike sekvensnummere for hver pakke, forventer den å motta ACK for hvert sekvensnummer. Når dette skjer vil koden skrive ut til brukeren at den har mottatt ACK for pakken x og fjerner det tilsvarende sekvensnummeret x fra vinduet. Dersom det oppstår en timeout på 500 millisekunder eller 0.5 sekunder håndterer koden det med prinsipper fra Go-Back-N strategien. Først vil brukeren bli varslet om at det har oppstått en timeout, før koden sender alle pakkene i vinduet på nytt. Timeout-en oppstår fordi klienten ikke mottar ACK for en pakke fra serveren.

Da vil klienten stoppe opp og ikke sende noen nye pakker. Den vil sende pakkene i vinduet på nytt, helt fram til den mottar ACK fra serveren for den forventede pakken.

Etter å ha sendt alle pakkene i vinduet på nytt venter klienten på en ACK for den pakken i vinduet med det laveste sekvensnummeret. Når den forventede ACK-en kommer fra serveren, fortsetter pakkeoverføringen som normalt. Dette er hvordan koden håndterer timeout og uordnede pakker.

Når pakkeoverføringen er ferdig kalles det på funksjonen «send\_fin\_packet()». Denne funksjonen sender en pakke med FIN-flagget til serveren, og kaller deretter på funksjonen «receive\_fin\_ack». Denne funksjonen kan du se nedenfor:

```
def receive_fin_ack(self):
    # Prøver å motta en FIN ACK-pakke fra serveren
    try:
        data, server = self.sock.recvfrom(1000)
        _, ack, flags = parse_header(data[:6])
        if flags == ACK:
            print("FIN ACK pakke er mottatt")
            # Avslutter forbindelsen
            self.close_connection()
    except socket.timeout:
        print("Timeout oppstod, mislykket nedbryting av forbindelsen")
        self.sock.close()
    except Exception as e:
        print("Feil ved mottak av FIN ACK pakke fra serveren:", e)
        self.sock.close()

# Beskrivelse av funksjonen:
# Avslutter forbindelsen til serveren
# Argumenter:
# self: Referanse til det aktuelle Client-objektet
# Funksjonen gjør:
# Lukker socketforbindelsen og avslutter programmet
# Retur: Ingen returverdi for denne funksjonen
def close_connection(self):
    print("Forbindelse avsluttet")
    self.sock.close()
    sys.exit(1)
```

Funksjonen «receive\_fin\_ack» sjekker om den har mottatt en ACK-flagg som skal være FIN-ACK flagget fra serveren. Dersom dette stemmer avsluttes forbindelsen ved å kalle på funksjonen «close\_connection». Dersom det oppstår en timeout eller feil håndteres dette av funksjonen «receive\_fin\_ack».

Dette er hvordan programmet «application.py» skal fungere i praksis.

### Diskusjon:

1. Jeg har laget en tabell med resultatene slik at det blir mer presist og oversiktlig.

Vindusstørrelse	RTT	Gjennomstrømning
3	100(standard)	0.23 mbps
5	100(standard)	0.38 mbps
10	100(standard)	0.75 mbps

Som vi kan se øker gjennomstrømningen for hver gang vindusstørrelsen øker. Gjennomstrømningen øker fordi et større vindu tillater mer data å være i transitt på samme tid. Vindusstørrelsen bestemmer antall pakker som kan sendes før klienten trenger en bekreftelse (ACK) fra serveren. Ved å øke vindusstørrelsen, kan klienten sende flere pakker uten å vente på en bekreftelse, noe som utnytter nettverksbåndbredden mer effektivt og reduserer ventetiden. Dermed, når vindusstørrelsen øker fra 3 til 5 og til 10, tillates mer data å være i transitt, noe som resulterer i høyere gjennomstrømning.

2. Her er en tabell med alle resultatene

Vindusstørrelse	RTT	Gjennomstrømning
3	50	0.45 mbps
5	50	0.75 mbps
10	50	1.45 mbps
3	200	0.12 mbps
5	200	0.19 mbps
10	200	0.38 mbps

Det første jeg legger merke til er at gjennomstrømmningen er mindre jo mer RTT øker. La oss bruke vindusstørrelsen 5 som et eksempel. Med RTT 50 er gjennomstrømmningen 0.75 mbps, RTT 100 er den 0.38 mbps og med RTT på 200 er gjennomstrømmningen på 0.19 mbps. Det ser ut som at gjennomstrømmningen nesten halveres for hver gang RTT økes dobles. RTT refererer til tiden det tar for en pakke å reise fra senderen til mottakeren og tilbake igjen. En lengre RTT betyr at det tar lenger tid for senderen å motta bekreftelsen for en sendt pakke, noe som kan redusere gjennomstrømmningen fordi senderen, altså klienten, må vente lenger før den kan sende flere pakker. Dette kan forklare hvorfor gjennomstrømmningen nesten halveres når ventetiden for klienten dobles.

3. Her bruker jeg -d 240 som et av kontrollinjeargumentene slik at serveren forkaster pakken med sekvensnummeret 240. Nedenfor kan du se hvordan både server og klient håndterte dette.

```

17:05:43.933881 -- Pakke med sekvensnummer = 239 er sendt, s
kyvevindu = [237, 238, 239]
17:05:43.986224 -- ACK for pakke = 237 er mottatt
17:05:43.988274 -- Pakke med sekvensnummer = 240 er sendt, s
kyvevindu = [238, 239, 240]
17:05:43.992452 -- ACK for pakke = 238 er mottatt
17:05:43.997139 -- Pakke med sekvensnummer = 241 er sendt, s
kyvevindu = [239, 240, 241]
17:05:44.041492 -- ACK for pakke = 239 er mottatt
17:05:44.543375 -- Timeout oppstod, sender alle pakker i vin
duet på nytt
17:05:44.543496 -- RTO oppstod
17:05:44.543603 -- Sender pakke med sekvensnummer = 240 på n
ytt
17:05:44.543984 -- Sender pakke med sekvensnummer = 241 på n
ytt
17:05:44.647248 -- ACK for pakke = 240 er mottatt
17:05:44.647508 -- ACK for pakke = 241 er mottatt
Dataoverføring fullført

Nedbryting av forbindelse:
FIN-pakke er sendt
FIN ACK pakke er mottatt
Forbindelse avsluttet
root@Ubuntu:/media/sf_Datanettverk_og_skytjenester_DATA2410
/ubuntuv-shared#
root@Ubuntu:/media/sf_Datanettverk_og_skytjenester_DATA2410/ubuntuv-shared#

```

Her kan vi se hvordan klienten håndterte den forkastede pakken 240. Klienten hadde sendt pakkene 239, 240 og 241. Etter 0.5 sek mottok den ikke ACK for den neste forventede pakken med sekvensnummer 240, som førte til en timeout eller/og RTO (retransmission timeout) error. Da sendte klienten pakkene med sekvensnummer 240 og 241 på nytt, siden pakke 239 er den siste pakken klienten hadde mottatt en ACK for. Dette er hvordan Go-Back-N strategien fungerer i praksis. Til slutt mottok klienten ACK for alle pakkene og avsluttet forbindelsen. Her er hvordan serveren håndterte forkastningen av pakke med sekvensnummer 240.

```

17:05:43.933881 -- Pakke 239 er mottatt
17:05:43.939838 -- Sender ack for den mottatte 236
17:05:43.980347 -- Pakke 237 er mottatt
17:05:43.985618 -- Sender ack for den mottatte 237
17:05:43.990669 -- Pakke 238 er mottatt
17:05:43.991432 -- Sender ack for den mottatte 238
17:05:44.041141 -- Pakke 239 er mottatt
17:05:44.041285 -- Sender ack for den mottatte 239
17:05:44.089573 -- Pakke 240 ble forkastet
17:05:44.097402 -- Uordnet pakke 241 er mottatt
17:05:44.645974 -- Pakke 240 er mottatt
17:05:44.646091 -- Sender ack for den mottatte 240
17:05:44.646868 -- Pakke 241 er mottatt
17:05:44.646936 -- Sender ack for den mottatte 241

FIN-pakke mottatt
FIN ACK-pakke sendt

Gjennomstrømmningen er 0.21 Mbps
Forbindelsen er avsluttet
root@Ubuntu:/media/sf_Datanettverk_og_skytjenester_DATA2410/ubuntuv-shared#

```

Her kan vi se at serveren forkastet pakke 240 og mottok en uordnet pakke 241. Deretter mottok serveren pakke 240 og 241 på nytt, og sendte en ACK for hver av pakkene til klienten.

Lengere ned på skjermbildet ovenfor kan vi se håndhilsing-prosedyren, gjennomstrømningen for pakkeoverføringen og at forbindelsen avsluttes.

4. For å ikke gjøre rapporten altfor lang skal jeg ta med ett bildesnutt av hvordan server og klient oppførte seg med tc-netem for å simulere pakketap på både 2% og 5%.

```
ies XTerm 17 Mai 17:41
"Node: h1"
17:38:59.756058 -- Pakke med sekvensnummer = 71 er sendt, sk
yvevindu = [69, 70, 71]
17:38:59.857713 -- ACK for pakke = 69 er mottatt
17:38:59.857946 -- Pakke med sekvensnummer = 72 er sendt, sk
yvevindu = [70, 71, 72]
17:38:59.858039 -- ACK for pakke = 70 er mottatt
17:38:59.858319 -- Pakke med sekvensnummer = 73 er sendt, sk
yvevindu = [71, 72, 73]
17:39:00.359749 -- Timeout oppstod, sender alle pakker i vin
duet på nytt
17:39:00.359871 -- RT0 oppstod
17:39:00.359966 -- Sender pakke med sekvensnummer = 71 på ny
tt
17:39:00.360308 -- Sender pakke med sekvensnummer = 72 på ny
tt
17:39:00.360608 -- Sender pakke med sekvensnummer = 73 på ny
tt
17:39:00.461384 -- ACK for pakke = 71 er mottatt
17:39:00.461817 -- Pakke med sekvensnummer = 74 er sendt, sk
yvevindu = [72, 73, 74]
17:39:00.461947 -- ACK for pakke = 72 er mottatt
17:39:00.462198 -- Pakke med sekvensnummer = 75 er sendt, sk
yvevindu = [73, 74, 75]
17:39:00.462954 -- ACK for pakke = 73 er mottatt
17:39:00.463195 -- Pakke med sekvensnummer = 76 er sendt, sk
yvevindu = [74, 75, 76]
17:39:00.562761 -- ACK for pakke = 74 er mottatt
17:39:00.562938 -- Pakke med sekvensnummer = 77 er sendt, sk
yvevindu = [75, 76, 77]
```

Dette er klient med pakketap på 2%.

```
ies XTerm 17 Mai 17:44
"Node: h2"
17:38:59.652513 -- Pakke 64 er mottatt
17:38:59.652560 -- Sender ack for den mottatte 64
17:38:59.652624 -- Pakke 65 er mottatt
17:38:59.654066 -- Sender ack for den mottatte 65
17:38:59.753908 -- Pakke 66 er mottatt
17:38:59.754009 -- Sender ack for den mottatte 66
17:38:59.754114 -- Pakke 67 er mottatt
17:38:59.754176 -- Sender ack for den mottatte 67
17:38:59.755324 -- Pakke 68 er mottatt
17:38:59.755376 -- Sender ack for den mottatte 68
17:38:59.856563 -- Pakke 69 er mottatt
17:38:59.857450 -- Sender ack for den mottatte 69
17:38:59.857573 -- Pakke 70 er mottatt
17:38:59.857620 -- Sender ack for den mottatte 70
17:38:59.958520 -- Uordnet pakke 72 er mottat
17:38:59.958655 -- Uordnet pakke 73 er mottat
17:39:00.460793 -- Pakke 71 er mottatt
17:39:00.460814 -- Sender ack for den mottatte 71
17:39:00.461118 -- Pakke 72 er mottatt
17:39:00.461170 -- Sender ack for den mottatte 72
17:39:00.461234 -- Pakke 73 er mottatt
17:39:00.462854 -- Sender ack for den mottatte 73
17:39:00.562396 -- Pakke 74 er mottatt
```

Dette er server med samme pakketap på 2%

Her kan vi se at både server og klient håndterer uforutsigbart pakketap akkurat som de skal. Klienten gjør som den skal ved å sende alle pakkene i vinduet på nytt og serveren varsler om uordnede pakker før den mottar de på nytt og sender en ACK for hver av dem.



```
ies XTerm 17 Mai 17:53
"Node: h1"
17:50:49.164748 -- Pakke med sekvensnummer = 39 er sendt, sk
ygevindu = [37, 38, 39]
17:50:49.665860 -- Timeout oppstod, sender alle pakker i vin
duet på nytt
17:50:49.665985 -- RT0 oppstod
17:50:49.666101 -- Sender pakke med sekvensnummer = 37 på ny
tt
17:50:49.666395 -- Sender pakke med sekvensnummer = 38 på ny
tt
17:50:49.666618 -- Sender pakke med sekvensnummer = 39 på ny
tt
17:50:49.772500 -- ACK for pakke = 37 er mottatt
17:50:49.772737 -- Pakke med sekvensnummer = 40 er sendt, sk
ygevindu = [38, 39, 40]
17:50:49.772885 -- ACK for pakke = 38 er mottatt
17:50:49.773092 -- Pakke med sekvensnummer = 41 er sendt, sk
ygevindu = [39, 40, 41]
17:50:50.274085 -- Timeout oppstod, sender alle pakker i vin
duet på nytt
17:50:50.274176 -- RT0 oppstod
17:50:50.274252 -- Sender pakke med sekvensnummer = 39 på ny
tt
17:50:50.274511 -- Sender pakke med sekvensnummer = 40 på ny
tt
17:50:50.274732 -- Sender pakke med sekvensnummer = 41 på ny
tt
17:50:50.375778 -- ACK for pakke = 39 er mottatt
17:50:50.376022 -- Pakke med sekvensnummer = 42 er sendt, sk
ygevindu = [40, 41, 42]
17:50:50.877436 -- Timeout oppstod, sender alle pakker i vin
duet på nytt
17:50:50.877591 -- RT0 oppstod
17:50:50.877768 -- Sender pakke med sekvensnummer = 40 på ny
tt
17:50:50.878204 -- Sender pakke med sekvensnummer = 41 på ny
tt
17:50:50.878488 -- Sender pakke med sekvensnummer = 42 på ny
tt
17:50:50.980754 -- ACK for pakke = 40 er mottatt
17:50:50.981328 -- Pakke med sekvensnummer = 43 er sendt, sk
```

Her er klient med pakketap på 5%. Pakketap forekom oftere sammenlignet med pakketap på 2%, men koden håndterte det godt ved å bruke prinsippene fra Go-back-N strategien.

```
ies XTerm 17 Mai 18:00
"Node: h2"
17:50:49.056431 -- Sender ack for den mottatte 32
17:50:49.060936 -- Pakke 33 er mottatt
17:50:49.061755 -- Sender ack for den mottatte 33
17:50:49.062080 -- Pakke 34 er mottatt
17:50:49.062128 -- Sender ack for den mottatte 34
17:50:49.158067 -- Pakke 35 er mottatt
17:50:49.158193 -- Sender ack for den mottatte 35
17:50:49.164269 -- Pakke 36 er mottatt
17:50:49.164363 -- Sender ack for den mottatte 36
17:50:49.259725 -- Uordnet pakke 38 er mottat
17:50:49.264960 -- Uordnet pakke 39 er mottat
17:50:49.767232 -- Pakke 37 er mottatt
17:50:49.772184 -- Sender ack for den mottatte 37
17:50:49.772347 -- Pakke 38 er mottatt
17:50:49.772397 -- Sender ack for den mottatte 38
17:50:49.873521 -- Uordnet pakke 40 er mottat
17:50:49.874194 -- Uordnet pakke 41 er mottat
17:50:50.375193 -- Pakke 39 er mottatt
17:50:50.375307 -- Sender ack for den mottatte 39
17:50:50.375423 -- Uordnet pakke 41 er mottat
17:50:50.476442 -- Uordnet pakke 42 er mottat
17:50:50.980286 -- Pakke 40 er mottatt
17:50:50.980428 -- Sender ack for den mottatte 40
17:50:50.980579 -- Pakke 41 er mottatt
17:50:50.981794 -- Sender ack for den mottatte 41
17:50:50.981794 -- Pakke 42 er mottatt
17:50:50.981794 -- Sender ack for den mottatte 42
17:50:51.082295 -- Pakke 43 er mottatt
17:50:51.082444 -- Sender ack for den mottatte 43
17:50:51.082617 -- Pakke 44 er mottatt
17:50:51.082707 -- Sender ack for den mottatte 44
17:50:51.083912 -- Pakke 45 er mottatt
17:50:51.084677 -- Sender ack for den mottatte 45
```

Her er samme pakketap på 5% fra serverens side. Koden håndterte pakketapene godt, som forventet.

**Kilder:**

Mine obligatoriske oppgaver, oblig 1 og oblig 2

Github til dette faget DATA2410 laget av lærer Safiqul:

Islam, S. 2410, GitHub. <https://github.com/safiqul/2410>

Moduler for DATA2410 i Canvas: <https://oslomet.instructure.com/courses/27894>

Pensumbok:

Ross, K., & Kurose, J. F. (2022). Computer networking: a top-down approach (8. ed.). Pearson Education; Pearson Education.