README FIL - 155116

Hvordan å kjøre programmet mitt:

- Ha en terminal for server
- Ha én eller flere terminaler for servere
- Make all
- Kjør server og klient uten argumenter
    - Dette er for å få kommandolinjene
- Kjør igjen, med riktige argumenter

NB!

- Når klienten skal skrive til fil, så fikke jeg ikke til å konkatinere 2 strenger, slik at jeg kunne lage filen "kernel-fil-<tilfeldig tall>"
- Så løsningen jeg brukte var bare å skrive filen som et tilfeldig tall
- Oppgaven sier ikke dette, så denne løsningen er en stor ulempe
- Ellers fungerer alt som det skal

Hva er mitt program?

Programmet er en enkel server-client kommunikasjons system, der serveren skal kunne sende pakker til klienten ved bruk av en enkel socket. Serveren skal kunne håndtere flere klienter samtidig, i tillegg til å håndtere pakketap.

I oppgaven står det at hoved loopen skal:

1. Sjekke om en ny RDP-forbindelse er mottat
2. Prøve å levere neste pakke til hver tilkoblede klient
3. Sjekke om en tilkobling skal bli sendt
4. Hvis ingenting har skjedd, vent en periode, eller til noe skjer på socketen

Jeg har valgt å utføre denne loopen ved å se på hver enkel pakke som serveren mottar. Hver gang serveren mottar en pakke, så sjekker jeg om pakken er en forespørsel, terminasjon, eller en ack. Fordelen med dette er at jeg ikke må kalle på RDP metodene mine oftere enn jeg må. Ulempen blir nok at jeg bare håndterer en og en pakke, med unntak av noe som skal forklares senere.

Min loop:

Min loop går slik:

1. Mottar pakke på socket
2. Sjekker om pakken er request
    a. Hvis ja, så aksepterer vi pakken, dersom det tillater seg
3. Sjekker om pakken er en terminasjon

    a. Hvis ja, terminer pakken
    b. Hvis alle klienter er terminert, avslutt server
4. Sjekker om det er en ack
    a. Hvis ja, sender neste pakke
5. Vent 1 sek

Stop and wait protocol():

Med min loop så tenkte jeg at det var enkelt å implementere stop and wait. Dersom pakken vi mottok var en ack, så vet vi at vi kan sende neste pakke. Dette betyr at serverer kommer til å vente på en ack, før den i det hele tatt sender neste pakke. Samme gjelder for klient siden. En klient sender ikke acks uten videre.

Utfordringen av å bruke én socket:

Stopp og vent protokollen, multipleksing, samt håndtering av pakketap er noe som man enkelt kan implementere ved bruk av TCP. I tcp så kan vi bruke en socket for hver eneste klient som er tilkoblet, og videre sjekke om vi har mottatt meldinger på en socket, og deretter svare på den socketen.

Med UDP, derimot, så bruker vi kun én socket. Dette betyr at hver klient som er tilkoblet vil sende meldinger på samme socket. Utfordringen blir da å kunne skille alle klientene.

Jeg løste dette ved å lage en struct i rdp protokollen. Dette var en struct som representerte koblinger mellom server og klient. Hver kobling inneholdte ID til server, ID til klient, noe som sier om den er lukket eller i timeout, en tid, antall pakker som klienten har mottatt, hvilken pakke som ble sendt sist og adressen til klienten. Dette er noe som er greit å ha, fordi da får jeg tilgang til riktig adresse til hver klient, og hvilke pakker som har gått tapt. I tilleg til å holde styr på hvor mange pakker som har blitt mottatt

Dette løste utfordringen, fordi ved recvid til pakkene så kunne jeg få tilgang til riktig kobling, og dermed få tilgang til riktig adresse som pakkene skulle sendes til. Dette er også en av fordelene ved å håndtere en og en pakke. Fordi da kan jeg håndtere klienten som sendte pakken(ack, term).

RDP:

Min rdp protocol inneholder flere funksjoner som jeg mener er nødvendige for å kunne løse denne oppgaven.

Dette er funksjonene:

- rdp_connect(), som kobler en klient til en server
- rdp_accept(), som aksepterer koblingen
- rdp_write(), som sender pakker til en gitt adresse
- rdp_read(), som leser pakker, og sender ack
- rdp_terminate(), som lukker en kobling

Dette er metoder jeg synes ga mening, fordi det er slike metoder vi kan bruke i tcp for å sende pakker. Dette gjør det også enklere å vite når serveren eller klienten gjør noe. Og hva de gjør

Pakketap:

Å håndtere pakketap var en av utfordringene i denne oppgaven. Tidligere nevnte jeg at jeg håndterte en og en pakke individuelt. Men dersom vi sender en pakke som går tapt, så vil ikke serveren motta en ack. Det kan hende at serveren ikke mottar en ack, enten fordi pakken gikk tapt, eller fordi acken gikk tapt.

Server:
På server siden er det slik at hvis pakkene går tapt, så vil ikke socketen få en melding. Dersom det er flere klienter, så vil ikke socketen få en melding når alle klientene har tapte pakke. Da vil serveren vente i 1 sekund. Dersom ingenting skjer, så vil serveren gå gjennom alle klientene, se hvilken pakke som ble sendt sist, og deretter sende den pakken.

Her vil serveren håndtere alle tilkoblede klienter i samme iterasjon.

Klient:
På klient siden så vet vi at en pakke har gått tapt dersom den mottar samme pakke flere ganger. Dette løser jeg ved å holde styr på hvilken pktseq som kom før, og deretter sende samme ack hvis den nye pakken har samme pktseq som den før

Multipleksing:

Multipleksingen løste jeg ved bruk av structen jeg lagde i min RDP protocol.

Jeg lagde en liste som inneholdte hver connection, slik at jeg kunne håndtere hver klient på tvers av connections, istedenfor på tvers av socket. I motsetning til tcp.

Uvanlige hendelser med min kode:

Koden min kan følge med noen uvanlige metoder å løse ting på, med tanke på de ulike nettverkslagene.

RDP_read():

- Denne metoden tar inn 3 ekstra variabler, for at serveren skal kunne ha styr på tidligere pakker som har blitt sendt, antall pakker som har blitt sendt, og for å legge den mottatte pakken i en liste til klienten.

Pktseq og ackseq:

- I stopp og vent protokollen må disse variere mellom 0 og 1
- Måten jeg har utført dette på, er ved å settte %2 på rekkefølgen hver pakke blir sendt i
- Hvis en server skal sende N pakker, så vil pakkene bli sendt i en rekkefølge fra 0 til N
- Da bruker jeg %2 på denne rekkefølgen slik at partall er 0 og oddetall er 1.

Timer i koblinger:

- Hver kobling har en timer som blir resettet hver gang en pakke sendes
- En ny timer blir satt når serveren mottar en ack fra klienten
- Disse tidene blir sammenlignet, for å sjekke om serveren mottok acken i løpet av 100 milli sekunder