Multipleksing og Tapsgjenoppretting

Introduksjon

En viktig oppgave for transportlaget er multipleksing. Mens nettverkslaget er ansvarlig for å levere pakker til et endesystem, er transportlaget ansvarlig for å levere disse pakkene til riktig prosess på endesystemet. Dette er imidlertid en forenkling. I mange tilfeller må en enkel prosess tilby mange forbindelser, og transportlaget må tilby multipleksing mellom alle disse forbindelsene i samme prosess. Et veldig typisk eksempel på dette er en webserver, som må utlevere filer til flere nettlesere samtidig.

Hvis transportlaget tilbyr pålitelige tjenester til laget over, kan det også være nødvendig å implementere pålitelighet hvis det underliggende nettverket ikke gjør det. I dagens Internett tilbyr TCP multipleksing og pålitelighet på toppen av nettverkslaget IP.

I denne oppgaven skal du bruke UDP-protokollen for å etterligne en pakkeorientert, forbindelsesløs, upålitelig nettverkslagprotokoll, og du lager ditt eget transportlag som implementerer multipleksing for flere samtidige forbindelser til samme prosess på en enkelt server.

Du skal implementere to applikasjoner: en klient og en server. Disse applikasjonene implementerer både funksjonalitet for transportlag og funksjonalitet for applikasjonslag.

Oppgaven

Denne oppgaven er inspirert av en protokoll som sjelden brukes i dag: filtjenesteprotokollen FSP. Den overfører store filer (for eksempel en operativsystemkjerne) over UDP og brukes noen ganger for å levere store filer til mange bladdatamaskiner i en beregningsklynge når klyngen slås på. Disse bladdatamaskinene har ingen permanent lagring; FSP brukes for å hente operativsystemene deres under oppstart. FSP ble utviklet fordi det er en veldig lettvektig protokoll på toppen av UDP, men dens rolle er svært begrenset fordi den ikke kan gi pålitelig dataoverføring. Hvis pakker går tapt, må FSP starte overføringen på nytt.

I denne oppgaven skal du opprette et nytt transportlag, RDP, Reliable Datagram Protocol, som tilbyr multipleksing og pålitelighet, en NewFSP-klient som henter filer fra en server, og en NewFSP-server som bruker RDP for å levere store filer til flere NewFSP-klienter pålitelig.

RDP

Dessverre kan du ikke implementere RDP direkte over IP fordi det krever rotprivilegier. I stedet skal du bruke UDP for å etterligne en forbindelsesløs, upålitelig nettverkslagprotokoll under transportprotokollen RDP. På grunn av begrensningene ved IFIs installasjon, må du støtte IPv4-adresser. Du kan også støtte IPv6-adresser.

Transportprotokollen RDP, som ligger på toppen av denne «nettverksprotokollen», skal tilby multipleksing og pålitelighet. RDP-implementeringen din vil bruke en enkelt UDP-port for å motta pakker fra alle klienter, og den vil også svare på dem over denne porten.

Pålitelighet: RDP-protokollen din er en forbindelsesorientert protokoll, fordi det er lettere å gi pålitelighet med forbindelsesorienterte protokoller enn forbindelsesløse. Applikasjonen din skal kalle funksjonen `rdp_accept()` for å sjekke om en ny forbindelsesforespørsel er kommet, og godta den hvis det er tilfelle. `rdp_accept()` returnerer en NULL-peker hvis det ikke er tilfelle, og en peker til **en dynamisk allokert datastruktur** som representerer en RDP-forbindelse hvis en forbindelse aksepteres. RDP er ikke så komplisert som TCP; den bruker en enkel stopp-og-vent-protokoll for å oppnå pålitelig dataoverføring. Hver pakke som sendes av en RDP-avsender, må kvitteres av mottakeren før den kan sende neste pakke (stopp-og-vent). Hvis ingen bekreftelse mottas innen 100 millisekunder, sender RDP pakken på nytt. RDP-avsenderen godtar ikke en annen pakke fra applikasjonen før den forrige pakken har blitt levert (det betyr at den returnerer 0 når applikasjonen kaller `rdp_write()`).

Multipleksing: Påkrevet multipleksing skal implementeres av deg i RDP-protokollen din. Du skal bruke forbindelses-ID-er som identifiserer riktig forbindelse på serveren og på klienten. Når en klient oppretter en forbindelse, velger den sin egen forbindelses-ID tilfeldig og inkluderer en forbindelses-ID 0 for serveren når den ber om en forbindelse. Hvis en annen klient med samme forbindelses-ID allerede er koblet til serveren, må den avvise den nye forbindelsesforespørselen ved å sende en pakke med `flags==0x20`.

NewFSP

Du skal implementere to programmer: en NewFSP-server og en NewFSP-klient.

NewFSP-klienten skal bruke RDP for å koble seg til en NewFSP-server og hente en fil. Klienten kjenner ikke navnet på filen; den vil ganske enkelt opprette en RDP-forbindelse til serveren og motta pakker. NewFSP-klienten lagrer disse pakkene i en lokal fil kalt `kernel-file-<tilfeldig tall>`. Når NewFSP-serveren indikerer at filoverføringen er fullført ved å sende en tom pakke, lukker NewFSP-klienten forbindelsen.

NewFSP-serveren skal bruke RDP for å motta forbindelsesforespørsler fra NewFSP-klienter og betjene dem. NewFSP-serveren er en prosess med kun én tråd, som kan betjene flere NewFSP-klienter samtidig. Den har en hovedløkke, der den

1. sjekker om en ny RDP-forbindelsesforespørsel er mottatt,
2. prøver å levere neste pakke (eller den siste tomme pakken) til hver tilkoblede RDP-klient,
3. sjekker om en RDP-forbindelse er stengt.
4. Hvis ingenting har skjedd i noen av disse tre tilfellene (ingen ny forbindelse, ingen forbindelse ble stengt, ingen pakke kunne sendes til noen forbindelse), må du vente en periode.
   Det er to implementasjonsalternativer:
   a. Å vente i ett sekund og så forsøke igjen
   b. Å bruke `select()` for å vente til en hendelse inntreffer

Når en klient har koblet seg til, vil NewFSP-serveren overføre en stor fil til klienten. Når den har overført hele filen, sender den en tom pakke for å indikere at overføringen er fullført.

Den sender den samme store filen til alle disse klientene. Du kan velge om NewFSP-serveren leser filen fra disken flere ganger eller holder den i minnet. Den bruker RDPs multipleksing for å oppnå dette, men det betyr at den må kunne godta nye tilkoblinger fra en NewFSP-klient innimellom mens den sender pakker til andre, allerede tilkoblede NewFSP-klienter.

Pakkene

RDP-protokollformatet for transportlaget ditt må være følgende:

Datatype	**Navn (forslagsvis)**	**Beskrivelse**
unsigned char	`flag`	Flagg som definerer forskjellige typer pakker.
• Hvis `flag==0x01`, så er denne pakken en forbindelsesforespørsel. • Hvis `flag==0x02`, så er denne pakken en forbindelsesavslutning. • Hvis `flag==0x04`, så inneholder denne pakken data. • Hvis `flag==0x08`, så er denne pakken en ACK. • Hvis `flag==0x10`, så aksepterer denne pakken en forbindelsesforespørsel. • Hvis `flag==0x20`, så avslår denne pakken en forbindelsesforespørsel.		
usignert char	`pktseq`	Sekvensnummer pakke
usignert char	`ackseq`	Sekvensnummer ACK-ed av pakke
usignert char	`unassigned`	Ubrukt, det vil si alltid 0
int	`senderid`	Avsenderens forbindelses-ID i nettverksbyterekkefølge
int	`recvid`	Mottakerens forbindelses-ID i nettverksbyterekkefølge

int	`metadata`	Heltallsverdi i nettverksbyterekkefølge der tolkningen avhenger av verdien av flagg
• Hvis `flag==0x04`, så gir `metadata` den totale lengden på pakken, inkludert nyttelasten, i bytes. • Hvis `flag==0x20`, så gir `metadata` en heltallsverdi som indikerer årsaken til å avslå forbindelsesforespørselen. Du velger betydningen av disse feilkodene.		
bytes	`payload`	Nyttelast, det vil si antall bytes som er angitt med forrige heltall, men aldri mer enn 1000 bytes. Nyttelast er bare inkludert i pakken hvis `flag==0x04`.

Merknader:

- I hver pakke kan bare én av bitene i `flags` settes til 1. Hvis du mottar en pakke der dette ikke stemmer, kast den.
- En pakke kan kun ha nyttelast hvis `flags==0x04`.
- Hvis `flags==0x01`, er serverens forbindelses-ID ukjent og må inneholde 0.
- Den første datapakken i en forbindelse har sekvensnummeret 0.
- En ACK-pakke inneholder sekvensnummeret til den siste pakken den har mottatt (ulikt TCP).

Programmet sender datagrammer som har en nyttelast mellom 0 og 1000 bytes (>=0 og <1000). Programmet er helt avhengig av RDP for å garantere at alle bytes kommer i riktig rekkefølge og at ingen går tapt. NewFSP-applikasjonen tolker at en datapakke uten nyttelast indikerer slutten på en filoverføring – for RDP er en tom datapakke bare en vanlig datapakke.

Klienten

Klienten skal godta følgende kommandolinjeargumenter:

- IP-adressen eller vertsnavnet til maskinen der serverapplikasjonen kjører.
- UDP-portnummeret som brukes av serveren.
- Tapssannsynligheten `prob` gitt som en flytende verdi mellom 0 og 1, hvor 0 ikke er noe tap og 1 alltid er tap.

Klienten kaller funksjonen `set_loss_probability(prob)` ved starten av programmet.

Klienten prøver å koble seg til serveren. Hvis serveren ikke svarer på forbindelsesforespørselen i løpet av ett sekund, skriver klienten ut en feilmelding og avslutter. Når klienten har koblet seg til, skriver den ut forbindelses-ID-ene til både klient og server på skjermen.

Når klienten har mottatt og lagret den komplette filen som er overført av serveren og RDP-forbindelseen er lukket, skriver den ut navnet på filen som den har skrevet (dvs. `kernel-file-<tilfeldig tall>`) til skjermen og avslutter.

Serveren

Serveren skal godta følgende kommandolinjeargumenter:

- UDP-portnummeret som serverapplikasjonen skal motta pakker på
- Filnavnet på filen som den sender til alle tilkoblede NewFSP-klienter
- Antallet *N* filer som serveren skal servere før den avslutter seg selv.
- Taps-sannsynligheten gitt som en flyt-verdi mellom 0 og 1, hvor 0 ikke er noe tap og 1 alltid er tap.

Serveren kaller funksjonen `set_loss_probability(prob)` ved starten av programmet.

Hver gang en klient kobler seg til serveren, svarer serveren med en pakke med `flagg==0x10` og skriver ut forbindelses-ID-ene til både serveren og klienten til skjermen: `CONNECTED <klient-ID> <server-ID>` . Men hvis serveren allerede sender eller har sendt den *N*-te filen, godtar den ikke forbindelsen og svarer med en pakke med `flagg==0x20` og skriver ut: `NOT CONNECTED <klient-ID> <server-ID>`

Når en klient kobler seg fra serveren, skriver serveren ut forbindelses-ID-ene til både serveren og klienten til skjermen: `DISCONNECTED <klient-ID> <server-ID>`

Serveren skal avslutte når den har levert *N* filer.

Pakketap

Du må også håndtere tap på stien fra klienten til serveren. Stien vil ha tap fordi du må bruke funksjonen `send_packet()` som erstatning for C-funksjonen `sendto()`. Det tar de samme argumentene i samme rekkefølge.

De medfølgende filene `send_packet.c` og `send_packet.h` kompilerer på påloggingsmaskinene på IFI uten noen `-std` kompileringsflagg. Det kompileres også med `-std=gnu11` (standard), selv med `-Wall -Wextra`. Andre C-standarder, spesielt `-std=c11`, vil gi en advarsel og funksjonen vil muligens ikke funke riktig.

Pakketapssannsynligheten er gitt som et tall mellom 0 og 1. Hvis du implementerer pakkeformatet som beskrevet i denne oppgaven, blir bare pakker som inneholder data (`flags==0x04`) eller ACKs (`flags==0x08`) forkastet. Dette er for enkelhets skyld, men du kan

fjerne den betingelsen hvis du også vil oppleve tap av forbindelsesforespørsler og forbindelsesavslutninger.

Både klienten og serveren din må sette tapssannsynligheten ved å kalle funksjonen `void set_loss_probability(float x)` som er gitt i `send_packet.c`.

Minnehåndtering

Vi forventer at det ikke er minnelekkasjer under en normal kjøring av applikasjonene. En normal kjøring er en kjøring der kommandolinjeargumenter og filnavn er gyldige. Alt dynamisk allokert minne må eksplisitt deallokeres. Alle åpnede fildeskriptorer må være eksplisitt lukket.

Vi forventer at Valgrind ikke viser noen advarsler under en normal kjøring av applikasjonene. Hvis du er overbevist om at en advarsel er falsk, noe som sjelden skjer, må du oppgi dette i koden eller i et eget dokument.

Ikke å oppfylle disse forventningene vil påvirke evalueringen av oppgaven din negativt.

Innlevering

Du må sende inn all koden din i et enkelt TAR-, TAR.GZ- eller ZIP-arkiv.

Hvis filen heter `<kandidatnummer>.tar` eller `<kandidatnummer>.tgz` eller `<candidatenumber>`.tar.gz, vil vi bruke kommandoen `tar` på login.ifi.uio.no for å pakke den ut. Hvis filen heter `<kandidatnummer>`.zip, vil vi bruke kommandoen `unzip` på login.ifi.uio.no for å pakke den ut. Forsikre deg om at dette fungerer før du laster opp filen. Det kan også være smart å laste ned og teste koden etter innlevering.

Arkivet må inneholde en `Makefile`, som skal ha minst disse alternativene:

- `make` - kompilerer begge programmene til kjørbare binærfiler `client` og `server`
- `make all` - gjør det samme som `make` uten parameter
- `make clean` - sletter kjørbare filer og eventuelle midlertidige filer (f.eks. `*.o`)

Om evalueringen

Hjemmeeksamen vil bli evaluert på datamaskinene i `login.ifi.uio.no`-klyngen. Programmene må kompileres og kjøres på disse datamaskinene. Vi vil prøve å kjøre NewFSP-klient og server på forskjellige datamaskiner i klyngen.

Hjemmeeksamen krever forståelse for ulike funksjoner som operativsystemer og nettverksprotokoller må tilby applikasjoner, spesielt minnehåndtering, filhåndtering og nettverk.

Det er lurt å løse deler av denne hjemmeeksamen i separate programmer først og kombinere dem til en klient og en server senere.

Vi foreslår å adressere deloppgavene som følger:

- Minnehåndtering
    a. Ikke glem å frigjøre alt minnet du har allokert. I operativsystemer og nettverk er du alltid personlig ansvarlig for minnehåndtering. Å vite nøyaktig hvilket minne du bruker, hvor lenge du bruker det og når du kan frigjøre det, er en av de viktigste oppgavene i operativsystemer og nettverk. Vi bruker C i dette kurset for å sikre at du forstår hvor vanskelig denne oppgaven er og hvordan du kan løse den. Å gjøre dette riktig (med hjelp fra Valgrind) er veldig viktig for hjemmeeksamen.
    b. Vi forventer ikke at serveren frigjør alt minne før du implementerer nettverksdeloppgave f, men vi forventer at du husker minnet som du har allokert.
    c. Vi forventer ikke at klienten og serveren frigjør alt minne i tilfelle krasj eller når du må trykke `Ctrl-C på` grunn av et annet problem.
- Nettverk
    a. Send UDP-pakker mellom klient og server ved hjelp av funksjonene `sendto()` og `recvfrom()`. Forsikre deg om at både klient og server frigjør alt minne før du avslutter det.
    b. Implementere pakkeformatet som kreves av oppgaven.
    c. Bytt ut funksjonen `sendto()` med funksjonen `send_packet()`og sett tapssannsynligheten på klientsiden til en verdi mellom 0 og 1.
    d. Pass på at gjensending av pakker fungerer.
    e. Forsikre deg om at multipleksing fungerer etter forbindelse med flere klienter samtidig. Du må bruke den medfølgende funksjonen `send_packet()` med en viss sannsynlighet for pakketap. Filoverføring kan være for raskt for å starte flere samtidige NewFSP-klienter hvis du ikke gjør det.
    f. Implementer «steng forbindelse»-funksjonaliteten og sørg for at serveren frigjør alt minne riktig før den avslutter.
- Fil på serveren
    a. Sjekk om filen som er angitt på kommandolinjen til NewFSP-serveren eksisterer. Avslutt med en feilmelding hvis filen ikke blir funnet.
    b. Du kan åpne filen for sending én gang per forbindelse fra NewFSP-klienten, men du kan også lese den en gang og holde den i minnet til NewFSP-serveren avsluttes.
- Fil på klienten
    a. Sjekk at filen `kernel-file-<tilfeldig tall>` ikke eksisterer, og at du kan åpne den.
    b. Skriv pakkene som klienten mottar til filen.
- Opprette en komplett klient og server
  Du må løse flere nettverksdeloppgaver og flere fildeloppgaver for å bestå

hjemmeeksamen. En slurvet løsning for mange deloppgaver er like verdifull som en veldig god løsning for noen få deloppgaver.

Nyttige og relevante funksjoner

- For *sockets*
  - `socket()`
  - `bind()`
  - `sendto()`
  - `recvfrom()`
  - `select()`
  - `perror()`
  - `getaddrinfo()`
- For filer
  - `fread()`
  - `fwrite()`
  - `fopen()`
  - `fclose()`
  - `basename()`
  - `lstat` or `fstat()`
- For mapper
  - `opendir()`
  - `readdir()`
  - `closedir()`