Flytkontroll og håndtering av pakketap

Introduksjon

Flytkontroll er et konsept i nettverk som kontrollerer at hastigheten til en avsender ikke overskrider mottakerens kapasitet til å motta data så raskt den ankommer. Som et generelt konsept kan det finnes på flere lag i nettverkstacken, på lag 2 eller et hvilket som helst høyere lag. På lag 2 sikrer for eksempel flytkontroll at en node ikke sender raskere enn den direkte nabonoden kan motta, på lag 4 sørger flytkontroll for at et sendende endesystem ikke sender raskere enn et endesystem på mottakersiden kan motta og leverer til applikasjonslaget. Ikke alle protokoller inkluderer flytkontroll, lag N må implementere det hvis det tilbyr flytkontrollert tjeneste til lag N + 1.

“Loss recovery” er et annet konsept som en protokoll på lag 2 eller ethvert høyere lag kan implementere. Begrepet refererer til tapet av pakker som et lag N må korrigere hvis det tilbyr pålitelig service til det høyere laget N + 1. For å gjøre dette må protokollen et lag N implementere en reparasjonsmekanisme, og i mange tilfeller gjøres dette ved å oppdage tapet og sende den tapte pakken på nytt.

Flytkontroll og loss recovery er vanligvis veldig tett integrert med hverandre. I mange lærebøker beskrives faktisk overføring for loss recovery som en oppgave med flytkontroll.

På lag 4 av Internett tilbyr TCP en tjeneste med både flytkontroll og loss recovery, mens UDP tilbyr en tjeneste uten verken flytkontroll eller loss recovery.

Oppgaven

Det finnes applikasjoner som ikke er fornøyd med verken tjenestene til TCP eller UDP. I mange tilfeller leter de etter pålitelig, flytkontrollert tjeneste, men de krever at pakker blir levert til den mottakende applikasjonen nøyaktig slik den sendende applikasjonen sender dem. Av denne grunn legger de til et nytt lag som gir de manglende tjenestene på toppen av enten TCP eller UDP. Et eksempel er QUIC, som brukes for Google Chrome, og som implementerer en "bedre" transportlagstjeneste på toppen av transportlagstjenesten UDP.

I denne oppgaven skal du skrive et klient-serverapplikasjon som sammenligner bilder som er lagret på to endesystemer.

- Klienten leser flere bilder fra disken, åpner deretter en forbindelse til serveren og sender bildene til den. Hvert bilde overføres i en pakke. Bildene som følger med er små nok til å passe inn i en enkelt ethernet-ramme.

- Når serveren mottar en pakke fra klienten, trekker den ut bildet fra pakken og sammenligner den med alle bildene som er lagret lokalt på serveren. Den skriver navnet på filen som matcher bildet til en lokal fil. Denne filen kan deretter brukes til å sjekke at implementasjonen din fungerer som forventet.

Fordi serveren må gjøre mye arbeid for hvert bilde, er den selvfølgelig ikke klar til å motta nye pakker hele tiden. Flytkontroll er nødvendig for å forhindre at klienten sender for raskt.

Du må også håndtere tap på nettverksstien fra klienten til serveren. Stien vil ha tap fordi du må bruke vår funksjon send_packet() som erstatning for C-funksjonen send(). Tapssannsynligheten bør settes ved å kalle `set_loss_probability`().

Du må skrive et lag på toppen av UDP som gir flytkontroll og gjenoppretting av tap. I dette laget implementerer du skyvevindu-algoritmen (Sliding Window) med en maksimal vindusstørrelse på 7 pakker.

Pakkene

Protokollformatet til transportlaget ditt må være følgende:

- (int) pakningens totale lengde inkludert nyttelasten i byte
- (unsigned char) sekvensnummer for denne pakken
- (unsigned char) sekvensnummer for den siste mottatte pakken (ACK)
- (unsigned char) flagg
    - 0x1: 1 hvis denne pakken inneholder data
    - 0x2: 1 hvis denne pakken inneholder en ACK
    - 0x4: 1 hvis denne pakken lukker tilkoblingen
- Alle andre biter: 0
- (unsigned char) ubrukt, må alltid inneholde verdien 0x7f
- (bytes) nyttelast

Det skal ikke være noen data i mellom variablene i headeren, og mellom headeren og nyttelasten.

En pakke med flagget satt til 0x1 er en datapakke. Det skal ha nyttelast.
En pakke med flagget satt til 0x2 er en ACK. Det må ikke ha nyttelast.
En pakke med flagget satt til 0x4 er en avslutningspakke. Det må ikke ha nyttelast. Den brukes av klienten til å avslutte serveren, og den blir ikke ACKed.

Bare en av flaggene 0x1, 0x2 eller 0x4 kan være satt i en gyldig pakke.

Applikasjonen sender nyttelaster som inneholder følgende:

- Hvis applikasjonen sender en fil
    - (int) unikt nummer på forespørselen
    - (int) lengden på filnavnet i byte (inkludert endelig 0)
    - Filnavn (inkludert final 0)
    - Bytes av bildene
- ACK-pakker og termineringspakker har ikke -payload.

Filnavnet skal ikke inneholde kataloger. Du kan bruke funksjonen `basename`.

Klienten

Klienten skal godta følgende kommandolinjeargumenter:

- IP-adressen eller vertsnavnet til maskinen der serverprogrammet kjører.
- Portnummeret som serverprogrammet skal motta pakker til.
- Et filnavn som inneholder en liste over filnavn. En eksempelfil er gitt.
- Et tapsprosent. Dette bør være mellom 0 og 20.

Du kan ikke anta at noen av filnavnene (både de som er gitt og de i den gitte filen) er gyldige, og må derfor kontrollere at de er det. Hvis et filnavn er ugyldig, skal klienten skrive ut en passende melding og avslutte. Du kan anta at de andre kommandolinjeargumentene er riktige.

Nyttelast som inneholder en hel bildefil skal legges i én pakke og sendes til serveren. Den første pakken skal ha sekvensnummer 0 (dette er en forutsetning på serveren). Sekvensnummeret økes deretter med en for hver pakke; Den andre pakken har sekvens nummer 1, den tredje pakken har sekvens nummer 2, og så videre.

Pakketap er noe som ikke kan påvises pålitelig. Det beste vi kan gjøre er å konkludere med pakketap på grunn av mangel på en ACK. Klienten skal anse en pakke som tapt hvis den ikke har mottatt en bekreftelse på pakken innen 5 sekunder, bør den bruke en timeout for å oppdage dette. For enkelhets skyld kan du bare spore tiden for den eldste pakken (laveste sekvensnummer ikke mottatt bekreftelse for).

Hvis en pakke går tapt, må den sendes på nytt. Det er klientens jobb å gjøre det. Klienten må føre en koblet liste over sendte, men ikke kvitterte (ACKed) pakker for dette formålet. Du må allokere pakkene i denne listen i dynamisk minne. Hver oppføring i den koblede listen skal inneholde nok informasjon til å kunne sende pakken på nytt og fjerne pakker når en timeout eller mottak av kvittering skjer.

Hver gang en timeout oppstår, skal alle pakker i det nåværende vinduet sendes på nytt, og klokka på 5 sekunder skal tilbakestilles.

Hver gang en kvittering mottas, skal kvitteringsnummeret sammenlignes med sekvensnummeret til den eldste pakken. Hvis de stemmer overens, aksepteres kvitteringen; Pakken kan fjernes fra listen, og vinduet kan avanseres av en pakke.

Hvis de ikke stemmer overens, blir kvitteringen avvist; Klienten gjør ingenting. Kvitteringer er ikke kumulative i denne oppgaven, slik de er i TCP. Dette er ineffektivt, men det forenkler implementasjonen.

Når klienten har sendt alle sine bildefiler, skal den sende en termineringspakke til serveren. Denne pakken trenger ikke sendes pålitelig. Den tapte funksjonen vår (send_packet) kaster ikke termineringspakker. I det usannsynlige tilfellet at UDP dropper termineringspakken, må serveren avsluttes manuelt.

Serveren

Serveren skal godta følgende kommandolinjeargumenter:

- Portnummeret som serverprogrammet skal motta pakker til
- Et katalognavn som inneholder bildefiler.
- Filnavn for utskrift av treff

Serveren skal lytte til innkommende pakker på den angitte porten. Når serveren mottar en pakke, må den kontrollere at sekvensnummeret til den pakken stemmer overens med sekvensnummeret som forventes. Den første pakken antas å ha sekvensnummer 0.

Hver gang en pakke som har det forventede sekvensnummeret mottas, skal serveren gjøre følgende. Utfør trinnene i Go-back-N-protokollen for mottak av pakker og pass nyttelasten for behandling.

Nyttelasten inneholder et filnavn og et bilde. Bildet skal sammenlignes med innholdet i hver av filene i den gitte katalogen. Hvis en identisk fil blir funnet, skal serveren legge en streng som følgende til filen:

> “<nyttelasten sitt filnavn> <Server sitt filnavn>\n”

Hvis ingen treff blir funnet, bør serveren legge til en streng som følgende:

> “<nyttelasten sitt filnavn> UNKNOWN\n”

Du kan sammenligne bildene på to måter:

- enten lager du bildestrukturer av typen `struct Image` ved hjelp av funksjonen `Image_create` og sammenligner bildene med `Image_compare`. Ikke glemt å frigjøre minne med `Image_free`. Vi utleverer disse funksjonene i filene pgmread.c og pgmread.h.
- eller du sammenligner bildene byte-for-byte selv.

Serveren skal avslutte når den mottar en termineringspakke.

Minnehåndtering

Vi forventer at det ikke er noen minne-lekkasjer under en normal kjøring av applikasjonene. En normal kjøring er en kjøring der kommandolinjeargumenter og filnavn er gyldige, og UDP ikke har noe pakketap. Alt dynamisk tilordnet minne må eksplisitt frigjøres. Alle åpne fildeskriptorer må eksplisitt lukkes.

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.

Hvis du ikke oppfyller disse forventningene, vil det 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 din heter <kandidatnummer> .tar eller <kandidatnummer> .tgz eller <kandidatnummer> .tar.gz, bruker vi kommandoen tar på login.ifi.uio.no for å trekke den ut. Hvis filen din heter <kandidatnummer> .zip, bruker vi kommandoen unzip på login.ifi.uio.no for å trekke den ut. Forsikre deg om at dette fungerer før du laster opp filen.

Arkivet ditt må inneholde Makefile som har minst disse alternativene

make - kompilerer begge programmene dine, noe som resulterer i kjørbare binærprogrammer "client" og "server"

make all - gjør det samme som make uten noen parameter

make clean - sletter kjørbare filer og eventuelle midlertidige filer (f.eks. * .o)

Om evalueringen

Hjemmeeksamen krever forståelse av forskjellige funksjoner som operativsystemer og nettverksprotokoller må tilby til 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 ta opp deloppgavene som følger:

- Minnehåndtering

    a. Ikke glem å frigjøre alt minnet du har tildelt. 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 på dette kurset for å sikre at du forstår hvor vanskelig den 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 din frigjør alt minne før du implementerer Networking-underoppgaven (g) - men vi forventer at du ikke glemmer minnet du har tildelt.
    c. Vi forventer ikke at din klient og server frigjør alt minne i tilfelle krasjer eller når du må trykke på Ctrl-C på grunn av et annet problem.

- Nettverk

    a. Sende UDP-pakker fra en klient til server ved hjelp av funksjonene sendto og recvfrom, og sørg for at klienten frigjør alt minne før det avsluttes.
    b. Implementere pakkeformatet som kreves av oppgaven.
    c. Legg til stopp-og-vent-funksjonalitet med bare to sekvensnumre (du kan implementere Go-back-N direkte, men dette er tryggere).
    d. Erstatt funksjonen sendto med vår funksjon send_packet, bruk select for å implementere timeouts og sett taps-sannsynlighet på klientsiden til en prosentandel som ikke er null. Forsikre deg om at stop-and-wait fungerer.
    e. Bytt ut Stop-and-wait med Go-back-N med en maksimal vindusstørrelse på 7 pakker.
    f. Forsikre deg om at sendevinduet er implementert som en lenkeliste av strukturer som er allokert i dynamisk minne (sannsynligvis allerede gjort i (e)).
    g. Implementere funksjonen "tett forbindelse", og sørg for at serveren frigjør alt minnet riktig før det avsluttes.

- Filer

    a. Les en liste over filnavn fra en fil.
    b. Les en fullstendig fil i dynamisk minne.
    c. Sjekk om to buffere som er lagret i dynamisk minne, er identiske.
    d. Skriv en fil som inneholder to filnavn på hver linje hvis bufferne er identiske, eller ett filnavn og ordet UNKNOWN hvis bufferne ikke er identiske.

    e. Oppdag innholdet i en katalog (ved å bruke `readdir`), bestemme hvilke av disse som er filer (ved å bruke `lstat` eller `fstat`), og bruk listen som en liste over filnavn på serveren.

- Opprette en komplett klient og server.
  Du må løse flere underoppgaver for nettverk og flere underoppgaver for filer for å bestå hjemmeeksamen. En slurvete løsning for mange underoppgaver er like verdifull som en veldig god løsning for noen få underoppgaver.

    a. Det kan være en god ide å kombinere Networking (a) + (b) og Files (a) + (b) + (c) for en veldig enkel første versjon av klienten og serveren din.
    b. Det er en veldig god ide å lage en fungerende klient-server-løsning som kombinerer Networking (a) + (b) + (c) + (d) og Files (a) + (b) + (c) før du løser mer sub- oppgaver.
    c. Utvid dette ved å løse de gjenværende deloppgavene.

Nyttige og relevante funksjoner

- For sockets

    - `socket`
    - `bind`
    - `sendto`
    - `recvfrom`
    - `select`
    - `perror`
    - `getaddrinfo`

- For filer

    - `fread`
    - `fwrite`
    - `fopen`
    - `fclose`
    - `basename`
    - `lstat` eller `fstat`

- For kataloger

    - `opendir`
    - `readdir`
    - `Closedir`

Handouts

https://www.uio.no/studier/emner/matnat/ifi/IN2140/v20/undervisningsmaterial/h1-2020-handout-v1.tgz