Ingeniørhøjskolen Aarhus

Introduktion til kommunikationsnetværk

Exercise 8 og 9

4. semester

Alexander Dahl Bennedsen *201310498*
Lasse Stenhøj *201407500*
Sarah Overby *201407410*
Simon Hansen *201403871*

29. marts 2016

Indholdsfortegnelse

TCP server 1

1.1 Opgaveformulering

Der skal udvikles en server med support for en client ad gangen, som kan modtage en tekststreng fra en client. Serveren skal køre i en virtuel Linux-maskine. Tekststrengen skal indeholde et filnavn, eventuel ledsaget af en stiangivelse. Tilsammen skal informationen i tekststrengen udpege en fil af en vilkårlig type/størrelse i serveren, som en tilsluttet client ønsker at hente fra serveren. Hvis filen ikke findes skal serveren returnere en fejlmelding til client'en. Hvis filen findes skal den overføres fra server til client i segmenter på 1000 bytes ad gangen – indtil filen er overført fuldstændigt. Serverens portnummer skal være 9000. Serverapplikationen skal kunne startes fra en terminal med kommandoen:

#./file_server (for C/C++ applikationers vedkommende)
#./file_server.exe (for C# applikationers vedkommende)
#python file_server.py (for Python applikationers vedkommende)

Serveren skal være iterativ, dvs. den skal ikke lukke ned når den har sendt en fil til en client. Den skal, efter endt filoverførsel, kunne håndtere en ny forespørgsel fra en client (samme client eller en anden client). Serveren skal kun kunne håndtere en client ad gangen.

1.2 TCP server

TCP er en protokol som er pålidelig, da den opretter en fast forbindelse mellem server og klient, ved et såkaldt "handshake", før en overførsel initieres. Det vil sige, at så snart en TCP forbindelse er oprettet, vil pakkerne der sendes ikke længere have brug for en modtager-adresse, men kan blot "dumpes"ned i den TCP-strøm som løber imellem klient og server. TCP benytter desuden også ordnet levering dvs. pakkerne ankommer til modtageren i samme rækkefølge som de blev afsendt.

Vi starter med at oprette en socket for at kunne kommunikere med clienten.

```
1 sock = socket(AF_INET, SOCK_STREAM, 0);
```

Vi binder serveren adresse og portnummer til socket'en. Vi benytter et fast portnummer som er 9000, for at gøre det lettere.

Herefter venter serveren på at clienten tager kontakt til serveren ved hjælp af TCP. Når clienten tager kontakt til severen oprettes en socket til den indkommende forbindelse.

```
1 newsock = accept(sock, (struct sockaddr *) &cli_addr, &client_size);
```

Herefter afventer serveren at clienten sender filnavnet som clienten ønsker at hente fra serveren. Serveren læser filnavnet fra clinten ved hjælp af funktionen *readTextTCP()* fra det udleverede bibliotek LIB.

Hvis filen eksisterer kaldes funktionen *sendFile()* ellers sender serveren en fejlmeddelse til clienten og lukker forbindelsen.

I det efterfølgende vil vi beskrive funktionen *sendFile*.

Når serveren skal sende den efterspurgte fil. Starter vi med at sende filstørrelsen til clienten med TCP, således at clienten ved hvor mange bit den skal modtage.

Vi opretter en stream og åbner herefter den efterspurgte fil. Open funktionen tager to parametre et char array med filnavnet og flags som specificere om det er input eller output og hvilken operation vi benytter.

```
1 std::ifstream FileIn;
2 FileIn.open(fileName.c_str(), std::ios::in | std::ios::binary);
```

Da vores filnvn er en string benytter vi c++ funktinen *c_ str()* til konvertere strengen om til et char array. Vi har angivet at det er et input og at filen skal sendes i binære mode, hvilket var et krav til opgaven.

Vi tjekker for om filen eksisterer og om den er blevet åbnet korrekt. Hvis dette er tilfældet begynder vi at sende data til clienten. Dette gør vi ved at oprette en variabel kaldet rest som indeholder filstørrelsen. I et while-loop tester vi for om rest er større end 0 dvs. om der er mere data der skal sendes. Vi læser herefter 1000 bytes fra filen ved hjælp af funktionen *redsome()*. Denne funktion returnerer hvor mange bits der er blevet læst, hvilket vi benytter så vi ved hvor mange bits vi skal sende til clienten. Vi kan nemlig være sikker på at der altid vil blive læst til 1000 bytes. Herefter tester vi for om vi har læst nogen data, hvis dette ikke er tilfældet forsøger vi igen. Hvis det er tilfældet sender vi dataerne vi har læst til clienten og tæller rest ned.

```
1  long rest = fileSize;
2  while (rest > 0)
3  {
4          int count = FileIn.readsome(buffer,bufferSize); // Reading file
5          if(count >= 0)
6          {
7                  write(outToClient, buffer, count); // sending file
8                  rest -= count;
9          }
10 }
```

Simon sætter billede ind og skriver tekst til billedet. :-)

TCP client 2

2.1 Opgaveformulering

Der skal udvikles en client kørende i en anden virtuel Linux-maskine. Denne client skal kunne hente en fil fra den ovenfor beskrevne server. Client'en sender indledningsvis en tekststreng, som er indtastet af operatøren, til serveren. Tekststrengen skal indeholde et filnavn + en eventuel stiangivelse til en fil i serveren. Client'en skal modtage den ønskede fil fejlfrit fra serveren – eller udskrive en fejlmelding hvis filen ikke findes i serveren. Client-applikationen skal kunne startes fra en terminal med kommandoen:

#./file_client <file_server's ip-adr.> <[sti] + filnavn> (for C/C++ applikationers vedkommende)
#./file_client.exe <file_server's ip-adr.> <[sti] + filnavn> (for C# applikationers vedkommende)
#python file_client.py <file_server's ip-adr.> <[sti] + filnavn> (for Python applikationers vedkommende)

2.2 TCP client

TCP-serveren forventer at clienten sender et eventuelt stinavn, samt navnet på filen som ønskes overført. For at kunne eksekvere filen, kræves det ligeledes at der sendes en ip-adresse som der ønskes at hente filen fra.

Syntaks for eksekvering af beskeden bliver dermed som ønsket:
./file_server <IP-ADRESSE><FIL-NAVN>

Da alle tre kommandoer tæller som et argument, sørger klienten allerførst for at verificere at der er et korrekt antal argumente:

```
1        if(argc>3)
2        {
3        cout<<argv[1]<<argv[2]<<endl;
4        error("Invalid amount of arguments");
5        }
```

Herefter opretter vi socket, laver opsætning og giver en port, som igen naturligvis er 9000.

For at benytte IP-adressen som skrives i kommandovinduet, indlæses anden plads i arrayet til serveradressen:

```
1  server = gethostbyname(argv[1]);
2  if(server == NULL)
3  {
```

```
4 error("Host does not exist");
5 }
```

Herpå forbindes der til serveren, med dertil indbygget fejlsikring. Hvis der ikke kan oprettes forbindelse, lukkes clienten:

```
1         if(connect(clientsocket,(struct sockaddr *) &serv_addr, sizeof(serv_addr))==-1)
2         {
3         error("Fejl ved forbindelse til server");
4         }
```

Herfra anmodes der om at modtage en fil vha *recieveFile()*.

Da klienten skal anmode om en fil, og derfra modtage filen i 1000 bytes af gangen, er processen inddelt i flere dele.

- Spørg serveren om filen findes
- Hvis filen findes, åbnes en fil som den overførte data kan skrives i
- Serveren fortæller hvor stor en fil der skal overføres
- Størrelsen af filen bruges som reference ift. den overførte mængde data, som nu kan loopes i mængder af 1000 bytes af gangen.
- Ved endt transmission, lukkes filen

Når først filen er overført, termineres TCP-forbindelsen.

For at udføre første ovennævnte punkt, uddnyttes udleverede LIB-funktion *writeTextTCP*. Den ansøger om et filnavn, som sendes igennem socket til server. Herefter kan et svar ventes, som læses med *getFileSizeTCP*. Denne returnerer værdien af filstørrelsen.

Herpå benyttes standard read/write funktioner til at læse dataen og skrive til en fil. Returværdien fra *read* benyttes dog til at holde styr på hvor mange bytes der reelt set er sendt.

Koden for TCP-klienten er vedlagt.

Resultat 3

Som kvalitetskontrol for client/server systemet skal den overførte fil kunne sammenlignes med den oprindelige fil vha. terminal-kommandoen: cmp <afsendt fil> < modtaget fil> eller diff –s <afsendt fil> < modtaget fil> <afsendt fil> er overført til client vha. af email, ftp eller anden pålidelig, ikke proprietær overføringsmetode. Der må ikke være forskel mellem filerne, hverken mht. til størrelse eller mht. indhold.

UDP Server 4

4.1 Opgaveformulering

Skriv en iterativ UDP-server med support for en client ad gangen, som kan modtage en forespørgsel fra en client. Forespørgslen fra client til server kan være en af to muligheder: "U" eller "L". Om bogstaverne er lower case eller upper case skal være ligegyldigt.

- Hvis serveren modtager et "U" skal informationen i filen /proc/uptime returneres til client. Denne fil indeholder aktuel information om den samlede tid serveren har været kørende siden opstart.

- Hvis serveren modtager et "L" skal informationen i filen /proc/loadavg returneres til client. Denne fil indeholder aktuel information om serverens CPU-load.

4.2 UDP Server

UDP er en protokol som ikke giver garanti for at data kommer frem dvs. den er ustabil. UDP benytter ikke ordnet levering dvs. pakkerne ankommer ikke nødvendigvis i den rigtige rækkefølge til modtageren.

Ligesom vi gjorde ved TCP starter vi med at sætte en socket op. Vi benytter stadigvæk portnummer 9000 og binder dette til socket'en.

```
1 sock = socket(AF_INET, SOCK_DGRAM, 0);
```

Når vi opretter socket'en bruger vi parameteren *SOCK_DGRAM* i stedet for *SOCK_STREAM*, da vi nu benytter UDP i stedet for TCP.

Vi venter indtil vi modtager en besked fra clienten. Dette gøres med funktionen *recvfrom*.

```
1 n = recvfrom(sock,buffer,bufferSize,0, (struct sockaddr *) &cli_addr, &client_size);
```

Herefter laver vi en stream, som vi bruger til at åbne vores fil.

Hvis vores besked indeholder U eller u åbner vi filen /proc/uptime hvorimod hvis beskeden indeholder L eller l åbner vi filen /proc/loadavg. Dette gøres ved at benytte en switch.

Hvis beskeden ikke indeholder U/u/L/l lukker serveren ned.

Vi læser vores fil med funktionen

```
1 getline(FileIn, file);
```

FileIn er vores stream og file er den variable vi ønsker at ligge vores file over i.

Når dette er gjort sender vi vores file til client men funktionen.

```
1         n = sendto(sock, file.c_str(), file.length()+1, 0, (struct sockaddr *) &cli_addr,
              client_size);
```

Vi laver vores string om til en char array med funktionen *c_ str()* og grunden til at vi +1 til file.length() skyldes at vi skal have 0 terminering med.

Herefter lukker vi stream og vores socket.

UDP Client 5

5.1 Opgaveformulering

Skriv en UDP-client kørende i en anden laptop eller virtuel maskine, som kan sende et kommando i form af et bogstav "U", "u", "L", "l" som indtastes af operatøren til UDP-serveren. Når svaret fra UDP-serveren (beskrevet i punkt 1) modtages, udskrives dette svar til UDP-client's bruger.

5.2 UDP Client

File_client for UDP-protokollen kræver ikke at IP-adresse sendes direkte fra kommandolinjen, så som et designvalg, startes klienten bare og beder herpå om en IP-adresse som overførslen skal foregå fra.

Opsætningen af socket og serveradresse foregår derfra som tidligere.

Brugeren bedes derfra om at taste karakteren for den ønskede information der skal overføres.

Den indlæste karakter lægges i et char-array, som sendes med funktionen *sendto*:

```
1       sendto(clientsocket,msg,BUFFLENGTH,0,(struct sockaddr*)&serv_addr, serv_length);
```

Herpå afventes serverens svar ved funktionen *recvfrom*:

```
1       recvfrom(clientsocket,msg,BUFFLENGTH,0,(struct sockaddr*) &serv_addr, &serv_length);
```

Herpå udskrives den modtagne besked og klienten lukker.

Koden for UDP-klienten er vedlagt

Resultat 6