

Aarhus University

4. Semesterprojekt

IKN OPGAVE 12

Author:

Søren Landgrebe (201508295)
Stinus Lykke Skovgaard
(201401682)
Daniel Tøttrup (201509520)

Supervisor:

Torben Gregersen

May 16, 2017

TABLE OF CONTENTS

1	Opgaveformulering	1
2	Link::send	3
3	LINK::recieve	4
4	Transport::send	5
5	Transport::recieve	6
6	Applikationlaget::client	8
7	Applikation::server	10
8	Samlet	12

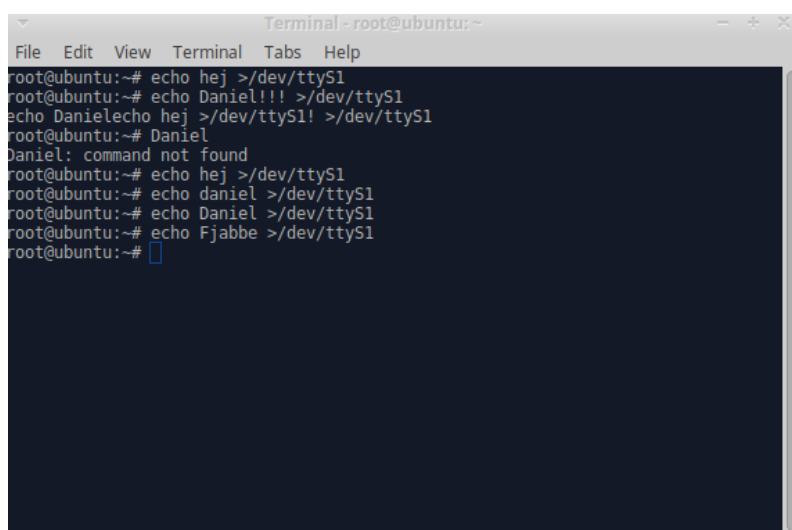
LIST OF FIGURES

1.1	Test vha minicom v. maskine1	1
1.2	Test vha minicom v. maskine2	2
5.1	Header tansportlag	6
6.1	Client modtager fil	9
7.1	Server	11
7.2	Server korrupt data	11
7.3	Server korrupt ack	11

OPGAVEFORMULERING

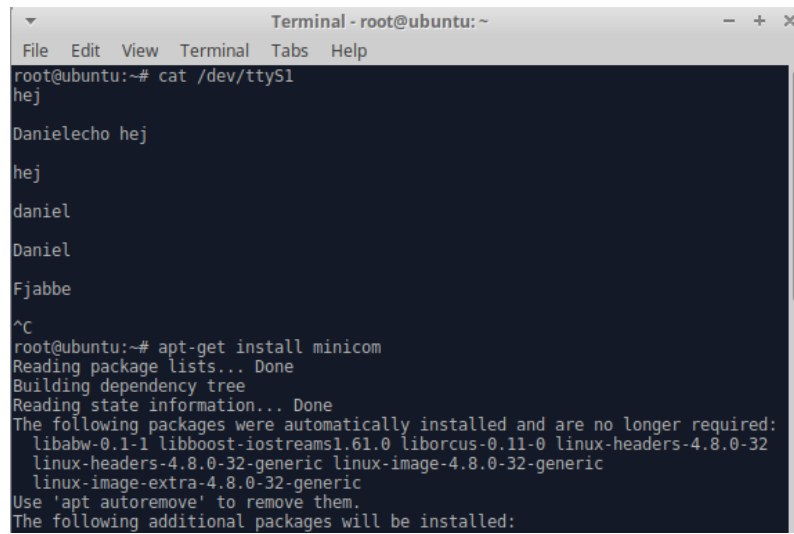
I denne opgave skal vi udvikle en protokol stack, som kan overføre en fil vha. den serielle port i vores virtuelle maskine. Vi bruger til øvelsen RS-232, som er en seriel digital datakommunikation. I opgaven, skal vi udvikle både en client og en server som hhv. skal stå for at sende og modtage en bestemt fil. Vi skal opbygge hele protokollen og derved hele laget, som en TCP-protokol. Dette indebærer linklaget, transportlaget og applikationslaget, som skal implementeres på både client og server. Hertil skal vores link lag, gøre brug af SLIP-protokollen. Vi bruger opgave 7 som grundlag for, at lave et applikationslag, som vi kun behøver at modificere for at kunne bruge til denne opgave.

Vi tester først vores serielle port, for at sikre os vi har opsat porten korrekt og at vi kan skabe en forbindelse mellem de to virtuelle maskiner.



```
Terminal - root@ubuntu:~
File Edit View Terminal Tabs Help
root@ubuntu:~# echo hej >/dev/ttyS1
root@ubuntu:~# echo Daniel!!! >/dev/ttyS1
echo Danielecho hej >/dev/ttyS1! >/dev/ttyS1
root@ubuntu:~# Daniel
Daniel: command not found
root@ubuntu:~# echo hej >/dev/ttyS1
root@ubuntu:~# echo daniel >/dev/ttyS1
root@ubuntu:~# echo Daniel >/dev/ttyS1
root@ubuntu:~# echo Fjebbe >/dev/ttyS1
root@ubuntu:~#
```

Figure 1.1: Test vha minicom v. maskine1



```
Terminal - root@ubuntu: ~
File Edit View Terminal Tabs Help
root@ubuntu:~# cat /dev/ttyS1
hej
Danielecho hej
hej
daniel
Daniel
Fjabbe
^C
root@ubuntu:~# apt-get install minicom
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  libabw-0.1-1 libboost-iostreams1.61.0 liborcus-0.11-0 linux-headers-4.8.0-32
  linux-headers-4.8.0-32-generic linux-image-4.8.0-32-generic
  linux-image-extra-4.8.0-32-generic
Use 'apt autoremove' to remove them.
The following additional packages will be installed:
```

Figure 1.2: Test vha minicom v. maskine2

LINK::SEND

I vores link send, vil vi implementere SLIP-protokollen som går ud på at vi starter og slutter vores data, altså fram'er hele dataprotokollen med A, for at undgå støj, og derved sikre at vi kender et start og slut punkt for data-bufferen. Vi vælger først at sætte den første plads i vores nye data-array til 'A', så vi ved vi starter med et A. Herefter erstatter vi alle steder vi har et A, med et BC, og et B med et BD. Vi bliver dog nød til at flytte det hele en plads i arrayet, for at få et ekstra karakter ind, dette gøres ved at tælle bufferen op. Alt dette gør vi, da vi kun vil bruge A som start og slut bit, og derfor bliver nød til at erstatte det fra inputbufferen. Hvis vi får et 0 fra inputbufferen, er besked slut, og vi tilføjer derfor et A til bufferen. Vi sikre dog den ikke tilføjer uendelige A, ved at lave en Acount, som bliver brugt som et flag, der bliver sat højt, når vi har lavet den sidste bit om til et A. Hvis vi modtager andet end fra vores inputbuffer end A eller B, bliver det placeret i den nye buffer.

```
void Link::send(const char buf[], short size)
{
    int j = 0;
    buffer[0] = 'A';
    int Acount = 0;

    for(int i = 0; i < size-1; ++i)
    {
        ++j;
        if(buf[i] == 'A')
        {
            buffer[j] = 'B';
            ++j;
            buffer[j] = 'C';
        }
        else if(buf[i] == 'B')
        {
            buffer[j] = 'B';
            ++j;
            buffer[j] = 'D';
        }
        else if (buf[i] == 0 && Acount == 0)
        {
            buffer[j] = 'A';
            Acount = 1;
        }
        else
            buffer[j] = buf[i];
    }
    v24Write (serialPort, (unsigned char *)buffer, strlen(buffer));
}
```

LINK::RECEIVE

I vores link send, vil vi implementere SLIP-protokollen som går ud på at vi starter og slutter vores data, altså fram'er hele dataprotokollen med A, for at undgå støj, og derved sikre at vi kender et start og slut punkt for data-bufferen. Vi vælger først at sætte den første plads i vores nye data-array til 'A', så vi ved vi starter med et A. Herefter erstatter vi alle steder vi har et A, med et BC, og et B med et BD. Vi bliver dog nød til at flytte det hele en plads i arrayet, for at få et ekstra karakter ind, dette gøres ved at tælle bufferen op. Alt dette gør vi, da vi kun vil bruge A som start og slut bit, og derfor bliver nød til at erstatte det fra inputbufferen. Hvis vi får et 0 fra inputbufferen, er besked slut, og vi tilføjer derfor et A til bufferen. Vi sikre dog den ikke tilføjer uendelige A, ved at lave en Acount, som bliver brugt som et flag, der bliver sat højt, når vi har lavet den sidste bit om til et A. Hvis vi modtager andet end fra vores inputbuffer end A eller B, bliver det placeret i den nye buffer.

```
void Link::send(const char buf[], short size)
{
    int j = 0;
    buffer[0] = 'A';
    int Acount = 0;

    for(int i = 0; i < size-1; ++i)
    {
        ++j;
        if(buf[i] == 'A')
        {
            buffer[j] = 'B';
            ++j;
            buffer[j] = 'C';
        }
        else if(buf[i] == 'B')
        {
            buffer[j] = 'B';
            ++j;
            buffer[j] = 'D';
        }
        else if (buf[i] == 0 && Acount == 0)
        {
            buffer[j] = 'A';
            Acount = 1;
        }
        else
            buffer[j] = buf[i];
    }
    v24Write (serialPort, (unsigned char *)buffer, strlen(buffer));
}
```


TRANSPORT::SEND

I vores `send()` funktion i transportlaget starter vi med at modtage filen

TRANSPORT::RECIEVE

I teansport recieve skal vi modtage bufferen fra linklaget, samtidig med at vi skal transportere bufferen videre til applikationslaget. ifølge koden kører vi først en do, som kører så længe vores modtaget ack, er false. Denne ack, kommer fra checkChecksum, som validerer vores header, som består af 4 værdier.

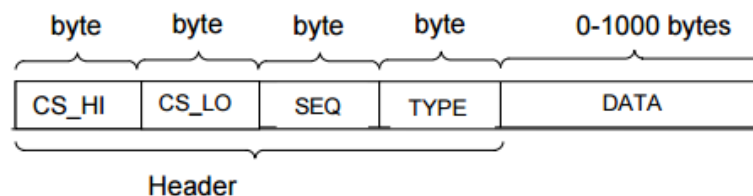


Figure 5.1: Header tansportlag

Til at starte med opretter vi først en lokal variable, som gemmer værdien fra link::recieve. Hvis Enable debuggeren er true, udskriver vi bufferen. Herefter opretter vi ack, som ifølge overstående bliver brugt som validering. Derfor sender vi også en sendAck(false), hvis vores modtaget ack er false. Vi laver også en errorhandling hvis vi modtager den samme pakke, vi sætter ack til false, da while løkken skal køre igen, men sender et true til sendAck(), da den derved får afvide vi gerne vil have den skal sende igen.

Hvis/når ack'en bliver true, sender vi et true til sendAck(), dette gøres så den ved at den gerne må sende næste buffer. Herefter overskriver vi den gamle sekvens nummer med den nye. og til sidste ligger vi vores modtaget buffer, over i buf, som derved kan sendes videre til applikationslaget.

```

short Transport::receive(char buf[], short sizeApp)
{
    int currentSize = 0;
    bool ACK_;
    int i;

    do{
        currentSize = link->receive(buffer, sizeApp+ACKSIZE);

        ACK_ = checksum->checkChecksum (buffer, currentSize);

        if(ACK_ == false)
            sendAck(false);

        if(buffer[SEQNO] == old_seqNo)
            //Registrere hvis vi modtager den samme pakke igen.
  
```

```
    {  
        ACK_ = false; //Koer loop igen  
        sendAck(true);  
    }  
  
    }while(ACK_ == false);  
  
    sendAck(true);  
  
    old_seqNo = buffer[SEQNO];  
  
    for(i = 0; i < sizeApp && i < (currentSize-ACKSIZE); i++)  
    {  
        buf[i] = buffer[i+ACKSIZE];  
    }  
  
    return i;  
}
```

APPLIKATIONSLAGET::CLIENT

i applikationslaget bruger vi vores hovedsagelig den viden vi har lært fra opgave 7. Det vil sige vi har taget det meste af koden fra opgave 7 og modificeret den.

Det første vi gør i koden, er at vi modtager filstørrelsen, denne bruger vi senere i koden, og bruger den itl at tjekke om filen eksisterer, dette gør vi på den måde at hvis file size er 0, eksisterer filen ikke.

Herefter laver vi en file descriptor, som peger på filen, den bruger vi til at kunne åbne filen. Hvis file descriptor'en er -1, har vi modtaget noget corrupt, og vi kører derfor en error på den.

Hvis der ikke er overført korrump data, modtager vi herefter filen, med 100 bytes af gangen, når der ikke er mere end 1000 bytes tilbage sender vi en "last bytes recieved" og udskrifter størrelsen på arrayet.

```
void file_client::receiveFile (std::string fileName, Transport::Transport *myTransport)
{
    char buffer[BUFSIZE] = {0};
    int n, recievedSize = 0;
    long fsize;
    int fd;

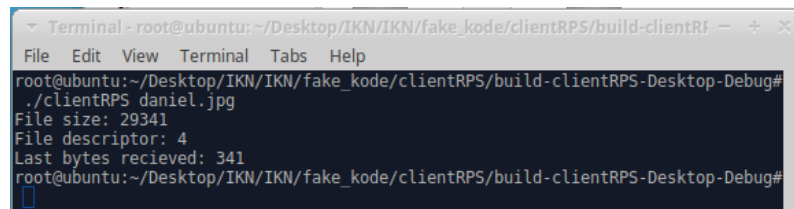
    /*Filstørrelse*/
    myTransport->receive(buffer,BUFSIZE);
    fsize = atol(buffer);
    printf("Fil størrelse: %d \n", fsize);

    if (fsize == 0)
    {
        fprintf(stderr,"ERROR: Filen findes ikke. \n");
        exit(0);
    }

    /*Laver file descriptor*/
    fd = open(fileName.c_str(), O_WRONLY | O_CREAT,S_IXGRP);
    if (fd == -1) error ("Kunne ikke aabne fil");
    printf("File descriptor: %d \n", fd);

    /*Fil fra serveren*/
    for (int i = 0; i<fsize/BUFSIZE+1; i++)
    {
        if (i<fsize/BUFSIZE) //Hvis der er mere end 1000bit tilbage
        {
            recievedSize = myTransport->receive(buffer,BUFSIZE);
            n = write(fd, buffer, BUFSIZE); /*fil bliver lagt over i fd*/
        }
    }
}
```

```
    }  
  
    else  
    {  
        recievedSize = myTransport->receive(buffer, fsize%BUFSIZE);  
        n = write(fd, buffer, fsize%BUFSIZE); /*fil bliver lagt over i fd*/  
  
        printf("Sidste bit er modtaget: %d \n", recievedSize);  
    }  
  
}  
  
close(fd);  
}
```



A terminal window titled "Terminal - root@ubuntu: ~/Desktop/IKN/IKN/fake_kode/clientRPS/build-clientRPS" with a menu bar (File, Edit, View, Terminal, Tabs, Help). The prompt is "root@ubuntu:~/Desktop/IKN/IKN/fake_kode/clientRPS/build-clientRPS-Desktop-Debug#". The user enters the command ". /clientRPS daniel.jpg". The output shows: "File size: 29341", "File descriptor: 4", and "Last bytes recieved: 341". The prompt returns to "root@ubuntu:~/Desktop/IKN/IKN/fake_kode/clientRPS/build-clientRPS-Desktop-Debug#".

Figure 6.1: Client modtager fil

APPLIKATION::SERVER

Ligesom med clienten er koden taget fra øvelse 7 og tilpasset vores nye system.

Serveren modtager et navn på en fil og sender den med det samme tilbage til clienten.

```
file_server::file_server ()
{
    Transport::Transport * myTransport = new Transport::Transport(BUFSIZE);
    char buffer[BUFSIZE] = {0};
    long fsize;

    CORRUPT_DATA = 2;
    CORRUPT_ACK = 2;

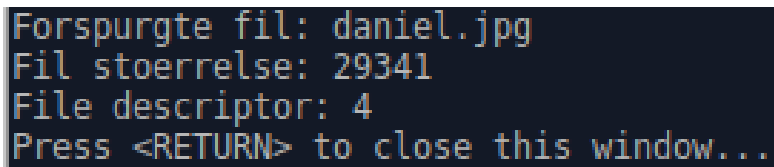
    /*Modtager filnavn*/
    myTransport->receive(buffer,BUFSIZE);
    std::cout << "Forspurgte fil: " << buffer << std::endl;
    string fileName_(buffer);

    /*Soeger efter filen*/
    struct stat sts;
    if ((stat (buffer, &sts)) == -1)
    {
        fprintf(stderr,"Fejl: Filen findes ikke. \n");
        fsize = 0;
    }
    else
    {
        fsize = sts.st_size;
        printf("Fil stoerrelse: %d \n", fsize);
    }

    sprintf(buffer,"%d",fsize);
    myTransport->send(buffer, sizeof(buffer));

    if(fsize == 0)
    {
        exit(0);
    }

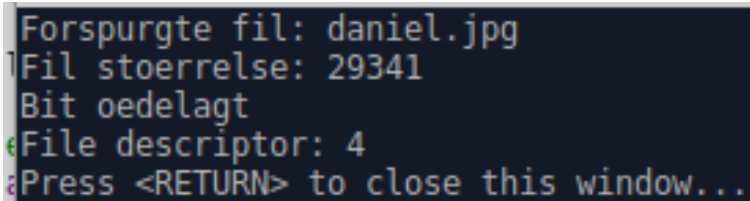
    /*Sender fil*/
    sendFile(fileName_,fsize,myTransport);
}
```

A terminal window with a dark background and light-colored text. The text is as follows:

```
Forspurgte fil: daniel.jpg
Fil stoerrelse: 29341
File descriptor: 4
Press <RETURN> to close this window...
```

A blue cursor is visible at the end of the last line.

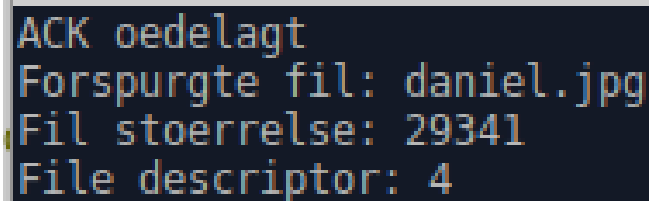
Figure 7.1: Server

A terminal window with a dark background and light-colored text. The text is as follows:

```
Forspurgte fil: daniel.jpg
Fil stoerrelse: 29341
Bit oedelagt
File descriptor: 4
Press <RETURN> to close this window...
```

A blue cursor is visible at the end of the last line.

Figure 7.2: Server korrupt data

A terminal window with a dark background and light-colored text. The text is as follows:

```
ACK oedelagt
Forspurgte fil: daniel.jpg
Fil stoerrelse: 29341
File descriptor: 4
```

A blue cursor is visible at the end of the last line.

Figure 7.3: Server korrupt ack

SAMLET