IKN Journal 2: Øvelse 11
Gruppe: 50
I4IKN
15-05-2018

Navn	Studienummer
Fatima Kodro	201609565
Søren Bech	201604784
Daniel Pat Hansen	201601915

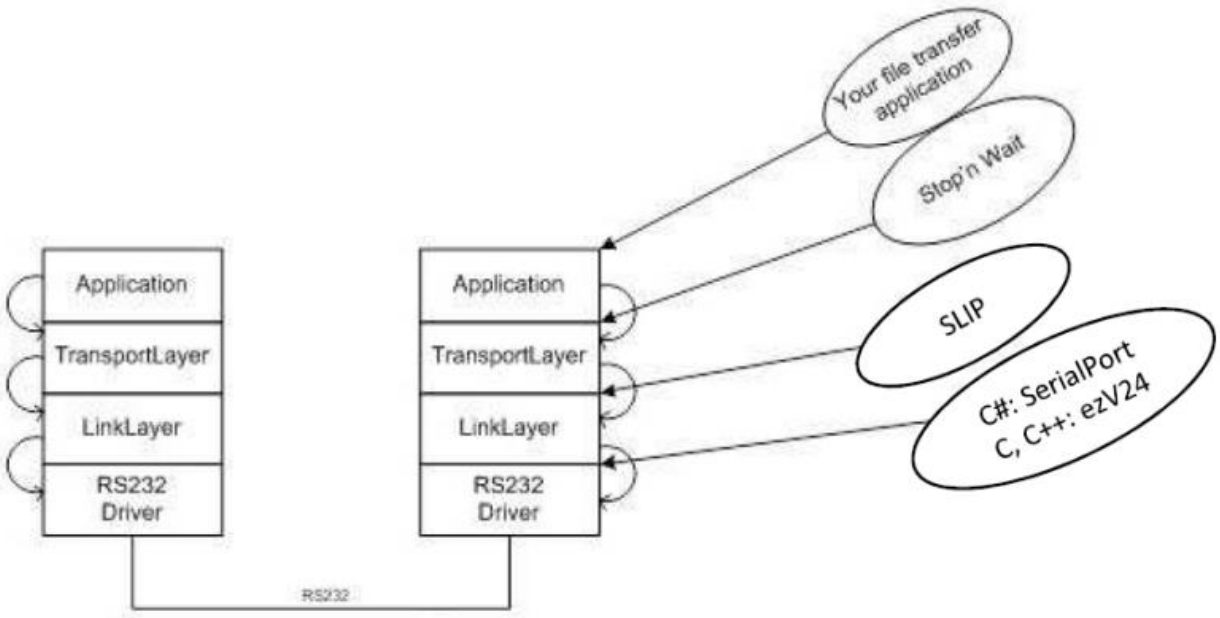
Indhold

Opgaveformulering

Der skal i denne øvelse designes og implementeres mulighed for at overføre en fil vha. den serielle kommunikations-port i en virtuel maskine. Der skal her udvikles en protokolstak.

Der skal kunne overføres en fil af en vilkårlig type og størrelse i en mappe fra en virtuel computer (H1) til en anden virtuel computer (H2). Disse virtuelle computere skal anvendes som server og klient, hvor klienten skal meddele om en vilkårlig fil, og serveren skal sende denne fil til klienten, hvis den findes, i pakkestørrelser på 1000 bytes payload ad gangen. klienten skal herefter gemme disse pakker i en fil.

Linklaget skal ikke være pålideligt. Det skal udelukkende kunne 'frame' segmenterne fra transportlaget. Transportlaget skal dog være pålideligt dvs. kunne detektere og håndtere fejl. Figur 1 illustrerer protokolstakken.



Figur 1 Lagdelt model for en protokolstak

Udviklingsforløb

For at kunne isolere de forskellige lag er det fysiske lag blevet testet først. Tty og loop-back portene, som simulerer null modem, blev testet. Idét loopback portene som var givet i forvejen ikke fungerede ordentligt, bruges et kernemodul ved navn tty0tty. Denne fil skal rettes til at indeholde en dependency (linux/sched/signal.h) hvorefter filen kan *makes* igen. Dette modul kan herefter indsættes med *insmod* hver gang computeren startes. Loopback portene som herefter kan anvendes er /dev/tnt0 og /dev/tnt1. Disse testes ved hjælp af minicom. Hele udviklingen af protokolstakken er blevet skrevet på én virtuel computer og testet med loopback-portene.

Link-laget udvikles herefter, som skal indramme hver datapakke og sende dette over den serielle port. Dette lag skulle ikke kunne håndtere fejl. Når dette fungerede fint blev transport-laget udviklet som skal kunne håndtere datafejl og ackfejl. Dvs. at hver gang en besked sendes skal en en checksum udregnes, samt skal et sekvensnummer tilføjes. Hvis der er fejl i en checksum skal der kunne sendes en NACK tilbage fra klienten. Modtages alt korrekt, sendes et ACK.

Note: Dog under test af protokolstakken over tty portene blev der konstateret at link-laget kræver at kunne håndtere fejl. Serveren sender hurtigere end klienten kan nå at modtage. dvs. at sidste del af pakken modtages uden den første delimiter. For at kunne komme uden om dette skal klienten sende en NACK tilbage og vente 200 ms, hvorefter den skal receive igen. Dette får transmissionen til at blive meget langsom over null-modemet.

Applikationslaget udvikles til sidst som skal bruge transportlaget til at sende en fil.

Lag

Fysisk lag

Det fysiske lag, også kaldt medielaget er hvor der fysisk tilsluttes systemet. Vi har tilsluttet vores system med virtuelle RS232 forbindelser som giver os en serial forbindelse mellem de to PC'er. Vi har også installeret noget der hedder tty0tty der skal compileres og tilsættes til driverene for at få en loopback funktion til at tilkalde vores egen indre drivere, til brug af tests.

Portene er TNT0 og TNT1.

Linklag

Linklaget er et lag hvor man adressere pakkerne som bliver linket til selve PCerne, man er stadigvæk i medielaget bare at man nu tilsætter man adressering og laver knude til knude data transmissionen. Den linker sig mellem de 2 computere og gør det i stand til at de kan kommunikere med hinanden. Der implementeres en modificeres SLIP protokol.

Den definerer flow control hvor den sørger for at senderen ikke sender for hurtigt til en langsom modtager.

Link laget skal "frame" datapakkerne. Delimiteren defineres som et 'A', dvs. alle A'er i datapakken bliver ændret til et AB. Alle B'er bliver ændret til BD. Dette gør at pakken bliver indrammet. Hvis pakken er modtaget forkert så vil den blive sendt igen.

Kodeforklaring

Send()

Følgende kodeudsnit er en del af *Send()* af link-laget som indrammer data ved at tilføje delimiters. A og B bliver erstattet hvorefter det skrives til den serielle port.

```
if (buf [i] == 'A')
{
    data.Add ((byte)'B');
    data.Add ((byte)'C');
}
else if (buf [i] == 'B')
{
    data.Add ((byte)'B');
    data.Add((byte)'D');
}
else
{
    data.Add (buf [i]);
}
```

Receive()

Receive skal derimod de-SLIP data, dvs. at det første og sidste index er et A. Forekommer der et B skal der indsættes et A eller B baseret på den næste karakter som enten kan være C eller D. Næste kodeudsnit viser dette. Desuden er der tilføjet en ekstra håndtering når der ikke modtages en delimiter. Her returneres der en værdi på -1.

```
if (_buffer [i] == 'B')
{
    // Must check on next index to insert A or B
    switch (_buffer [i + 1])
    {
    case (byte)'C':
        buf [bufIndex++] = (byte)'A';
        i++;
        break;
    case (byte)'D':
        buf [bufIndex++] = (byte)'B';
        i++;
        break;
    }
}
else if (_buffer [i] == DELIMITER)
    break;
else
    buf [bufIndex++] = _buffer [i];
```

Transportlag

Transportlageret håndterer dataerne: hvornår de skal sendes og hvornår dataerne som er modtaget er færdige. Det betyder at sekvenserne af data bliver segmenteret i mindre bidder for at så gå igennem en kvalitetskontrol med nack og ack.

Der skal så implementeres et transportlager hvor der håndteres data efter hvilke fejl og mangler der kunne være i pakkerne alt efter om de kom sikkert igennem.

I sekvensdiagrammet som er tillagt opgaverne til transportlageret er opgaverne følgende:

1. Et antal errors (i denne aflevering er dette 10) før senderen stopper med at prøve at sende
2. Timeout er i denne aflevering 500 ms
3. Hvis identiteten af pakken er det samme, så skal receiveren ignorere pakken og requeste en ny pakke
4. Hvis der er fejl i pakken så skal transmitteren gensende pakken, indtil kravet på opgave 1 er opfyldt eller der lykkes at sende pakken.

Kodeforklaring

For at kunne få koden til at fungere er *receiveAck()* metoden lavet om. En variabel *_ackSeqNo* (førhen kaldt oldSeqNo) er tilføjet. Denne bliver sat inde i receiveAck(). Herefter kan man sammenligne seqNo med denne variabel.

```
private void receiveAck()
{
    _recvSize = link.Receive(ref _buffer);
    _dataReceived = true;

    if (_recvSize == (int)TransSize.ACKSIZE) {
        _dataReceived = false;
        if (!checksum.CheckChecksum (_buffer, (int)TransSize.ACKSIZE)
            || _buffer [(int)TransCHKSUM.TYPE] != (int)TransType.ACK)
        {
            Console.WriteLine("Error in ack checksum or type!");
            _ackSeqNo = (byte) ((_buffer[(int)TransCHKSUM.SEQNO] + 1) % 2); // Increment current ack_seq
        }
        else
        {
            _ackSeqNo = (byte) _buffer[(int)TransCHKSUM.SEQNO]; // No incrementing since already incremented
        }
    }
}
```

Send()

Koden til transportlagets *Send* metode består i at resette bufferen, tilføje sekvensnummeret og datatypen. Bufferen bliver fyldt med data hvorefter checksummet beregnes. Dette ses på det næste kodeudsnit.

```
do
{
    for (int i = 0; i < _buffer.Length; i++)
    {
        _buffer [i] = 0;
    }

    //Seq
    _buffer [(int)TransCHKSUM.SEQNO] = _seqNo;
    //Type
    _buffer [(int)TransCHKSUM.TYPE] = (int)TransType.DATA;

    Array.Copy(buf, 0, _buffer, 4, size);

    //Tilføjer de to første "bytes" på buf
    checksum.CalcChecksum (ref _buffer, _buffer.Length);
```

Næste kodeudsnit tæller antallet af pakker som er overført. Når denne counter rammer 3, tilføjes der en fejl i datapakken, som receiveren skal reagere på. Ack-nummeret fra receiveren sættes lig sekvensnummeret.

```
Console.WriteLine($"TRANSMIT #{++_transmitCount}");

if(_transmitCount == 3) // Simulate noise
{
    _buffer[1]++; // Important: Only spoil a checksum-field (buffer[0] or buffer[1])
    Console.WriteLine($"Noise! - pack #{_transmitCount} is spoiled");
}

if (_transmitCount == 5)
    _transmitCount = 0;

_ackSeqNo = _seqNo;
```

Næste del af koden består i at beskeden sendes hvorefter der ventes på svar fra receiveren. Modtages en korrekt ack, dvs. seqNo = ackSeqNo er beskeden modtaget korrekt. Er dette ikke tilfældet, vil senderen gensende beskeden og vente på svar igen. Modtages der intet svar, dvs. der forekommer et timeout, vil en exception blive kastet, som derefter gribes. Senderen skal gensende beskeden og vente på svar. Dette kan ses i den vedlagte kode.

Ved fejl bliver en fejltæller talt op. Er denne counter lig med 10, stopper sendingen.

Er pakken korrekt sendt, vil den sidete del af koden opdatere sekvensnummeret:

```
// Update seqNo
_seqNo = (byte)((_seqNo + 1) % 2);
_errorCount = 0;
_ackSeqNo = DEFAULT_SEQNO;
```

Receive()

Receive vil læse indtil der modtages noget fra senderen. Hvis der returneres et -1 fra link-laget, vil der sendes en NACK samt ventes 200 ms. Der ventes, ellers vil der modtages resten af datapakken fra transmitteren, hvilket der ikke ønskes. Derfor skal der ventes så pakken er overført og transmitteren venter på et ACK/NACK. En exception kastes når der ikke modtages noget på receiveren, derfor try/catch blokken:

```
_recvSize = 0;
// Will time out while waiting, so must catch
while (_recvSize == 0 || _recvSize == -1)
{
    try
    {
        _recvSize = link.Receive (ref _buffer);
        if (_recvSize == -1)
        {
            // Send NACK and wait for message t
            sendAck(false);
            System.Threading.Thread.Sleep(200);
        }
    }
    catch (Exception)
    {
    }
}
```

Følgende kodeudsnit beskriver håndteringen af data. Er checksummen ikke korrekt skal der blot sendes en NACK tilbage. Er denne korrekt, skal seqNo opdateres og bufferen returneres. oldSeqNo opdateres og der sendes et ACK tilbage.

```
// If check is not right, send NACK, else do this
if (checksum.CheckChecksum (_buffer, _recvSize))
{
    Console.WriteLine ("Data pack OK.");

    // Update seqNo
    _seqNo = _buffer [(int)TransCHKSUM.SEQNO];

    // Return the received data
    Array.Copy (_buffer, (int)TransSize.ACKSIZE, buf, 0, _recvSize - (int)TransSize.ACKSIZE);

    // If identical package received, ignore
    if (_seqNo == _oldSeqNo)
        Console.WriteLine ("\tReceived identical package. Ignore");

    // Update oldSeqNo and return ACK
    _oldSeqNo = _seqNo;
    sendAck (true);
    return _recvSize - 4;
}

Console.WriteLine ("Error in data pack. Sending NACK.");
sendAck (false);
```

Der kan introduceres fejl i ack-pakkerne ved at tilføje den udkommenterede linje i næste kodeudsnit:

```
private void sendAck (bool ackType)
{
    byte[] ackBuf = new byte[(int)TransSize.ACKSIZE];
    ackBuf [(int)TransCHKSUM.SEQNO] =
        (byte)(ackType ? (byte)_buffer [(int)TransCHKSUM.SEQNO]
            : (byte)(_buffer [(int)TransCHKSUM.SEQNO] + 1) % 2);
    ackBuf [(int)TransCHKSUM.TYPE] = (byte)(int)TransType.ACK;
    checksum.CalcChecksum (ref ackBuf, (int)TransSize.ACKSIZE);

    if(_transmitCount == 1) // Simulate noise
    {
        ackBuf[1]++; // Important: Only spoil a checksum-field (ackBuf[0] or a
        Console.WriteLine($"Noise! ack #{_transmitCount} checksum is spoiled ;
    }

    if (_transmitCount == 10)
        _transmitCount = 0;

    Console.WriteLine ($"Ack sends seqNo #{ackBuf[(int)TransCHKSUM.SEQNO]}");

    link.Send(ackBuf, (int)TransSize.ACKSIZE);
}
```

Dog vil dette medføre fejl som ikke er omfattet af opgaven. Hvis der modtages en fejl i checksummen af en ACK pakke vil senderen se dette og gensende sin pakke. Dog kan en fejl forekomme hvis fejlen findes i den sidste pakke i en transmission. Dvs. dette kan ske hvis der tilføjes en ack-fejl på den første pakke som sender filstørrelsen til klienten eller på den sidste pakke som er en del af filen. I tilfældet ved filstørrelsen vil klienten ikke modtage filstørrelsen og vil gensende sin pakke. Serveren har modtaget filnavnet korrekt og begynder derfor at sende filen, men klienten venter på en filstørrelse som vil resultere i en fejl. Der ses et eksempel på dette under *resultater*.

Applikationslag

Applikationslageret er det der kommer tættest op af et brugerinterface hvor her brugeren kan tilgå og skrive kommandoer til programmet som eks. Hente en specifik fil fra serveren af.

Kodeforklaring

Server

Server koden består i at modtage en fil, tjekke at filen findes og hvis filen ikke findes skal der forespørges om en ny fil. Når et gyldigt filnavn er modtaget, sendes filen. Følgende kodeudsnit viser den første del af serveren. Resten af koden er sendingen af filen men dette er næsten det samme som i øvelse 6 og derfor ikke vist her. Kan dog ses i den vedlagte kode.

```
// Check file exist
long fileSize = LIB.check_File_Exists(LIB.ToString(filename).Substring(0, size));

Console.WriteLine (LIB.ToString (filename).Substring (0, size));

// If it does not exist, ask for another filename
while (fileSize == 0)
{
    string errorMsg = "File '" + LIB.ToString(filename) + "' not found \n";

    Console.WriteLine(errorMsg);

    // Send filesize back
    _transport.Send(LIB.ToBytes(fileSize.ToString()), LIB.ToBytes(fileSize.ToString()).Length);

    size = _transport.Receive(ref filename);

    Console.WriteLine (LIB.ToString (filename));

    // Check if file exist - must do with substring to get exact path
    fileSize = LIB.check_File_Exists(LIB.ToString(filename).Substring(0, size));
}

Console.WriteLine("File is found with size " + fileSize);

_transport.Send(LIB.ToBytes(fileSize.ToString()), LIB.ToBytes(fileSize.ToString()).Length);

sendFile (LIB.ToString (filename).Substring(0, size), fileSize, _transport);
// TO DO Your own code
```

Client

Følgende kode viser et udsnit af klient-koden. Her sendes filnavnet til serveren hvorefter der modtages en filstørrelse. Filstørrelsen tjekkes. Er størrelsen 0 vil der forespørges om et nyt filnavn fra serveren. Når denne ikke længere er 0, vil filen blive modtaget.

```
// Send file name
_transport.Send(LIB.ToBytes(filename), LIB.ToBytes(filename).Length);

// Filesize in bytes
byte[] fileSize = new byte[BUFSIZE];

// Receive file size
int size = _transport.Receive(ref fileSize);

// If the file size is equal to 0, resend a new file name
while ((LIB.ToString(fileSize).Substring(0, size) == "0"))
{
    Console.WriteLine($"File {filename} not found. Input a valid file");
    filename = Console.ReadLine();

    Console.WriteLine($"Requesting filename '{filename}'");

    _transport.Send (LIB.ToBytes (filename), LIB.ToBytes (filename).Length);

    _transport.Receive(ref fileSize);
}

Console.WriteLine ("File size: " + LIB.ToString(fileSize));

// Receive the file with the file size
receiveFile (filename, long.Parse (LIB.ToString (fileSize)), _transport);
```

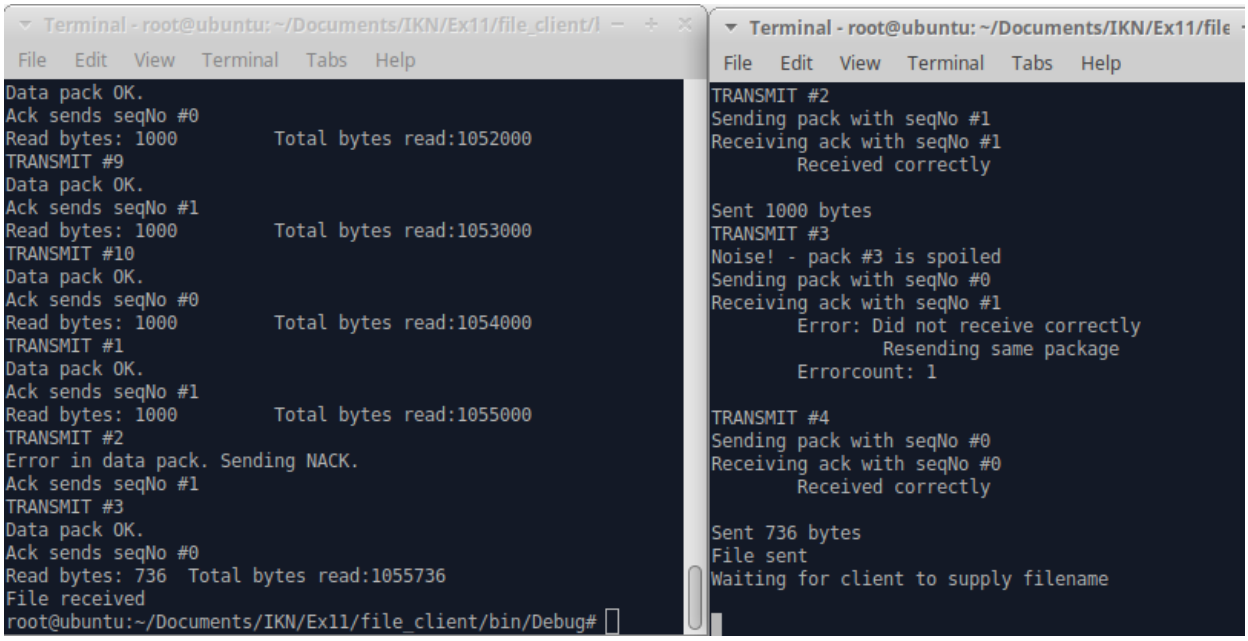
Læsningen af filen er næsten det samme som i øvelse 6, dog her bruges transportlagets receive metode i stedet. Kan ses i den vedlagte kode.

Resultater

Der er foretaget to former for test: nogle på release (dvs. tty) og nogle på debug (tnt). Dette er gjort da programmet opfører sig forskelligt baseret på dette.

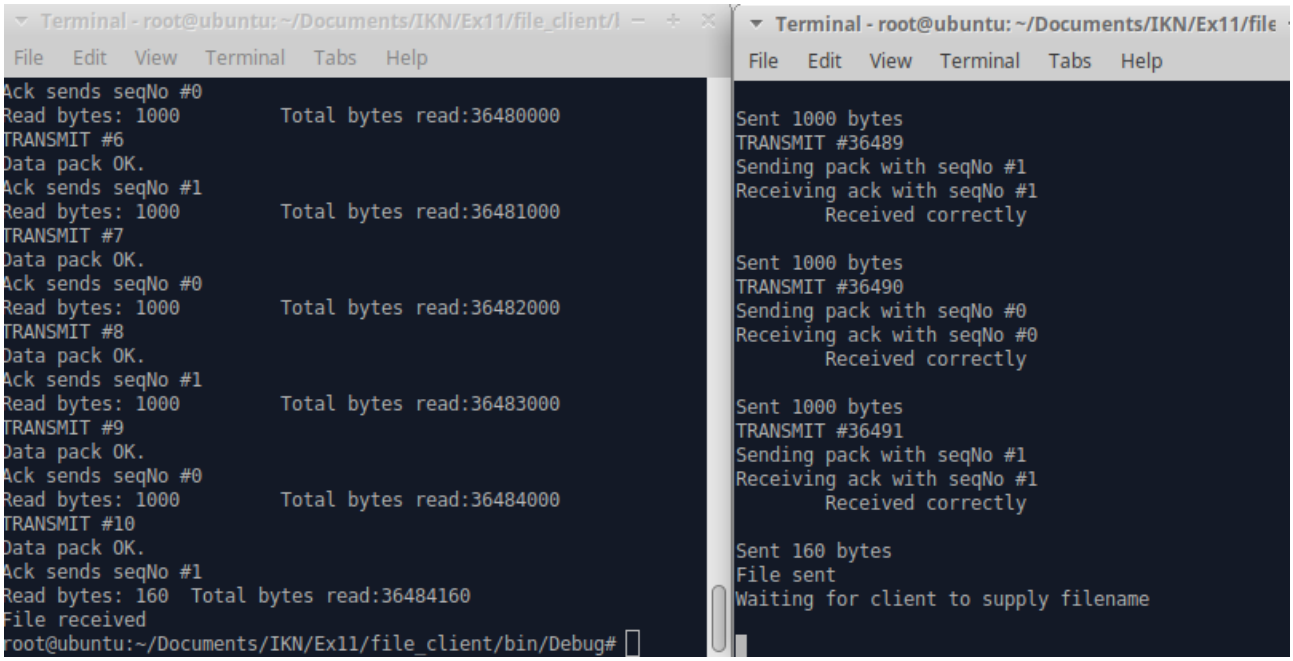
Debug

På Figur 2 ses der at en fil på 1 mb modtages fint og hurtigt.



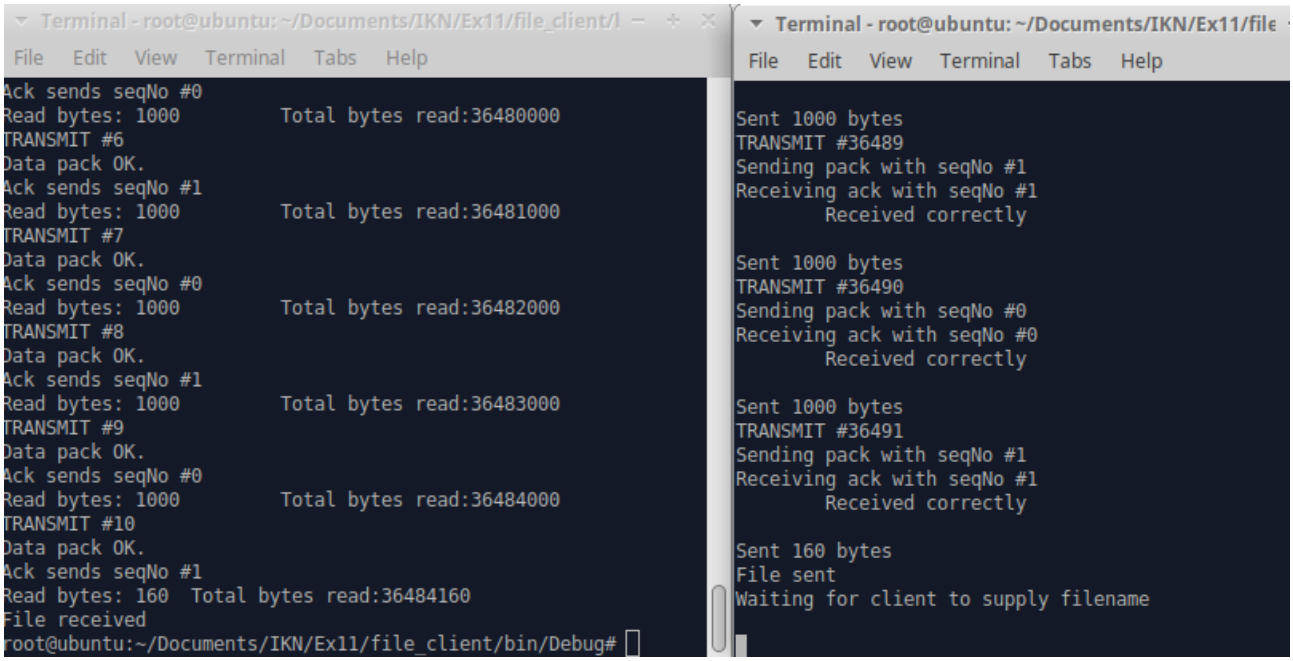
Figur 2 Client (venstre) modtager filen som server (højre) har sendt på en størrelse af ca. 1 mb

For at teste overførsel af større filer, sendes en fil på ca. 36 mb. Denne overføres også fint og relativt hurtigt. Se Figur 3.



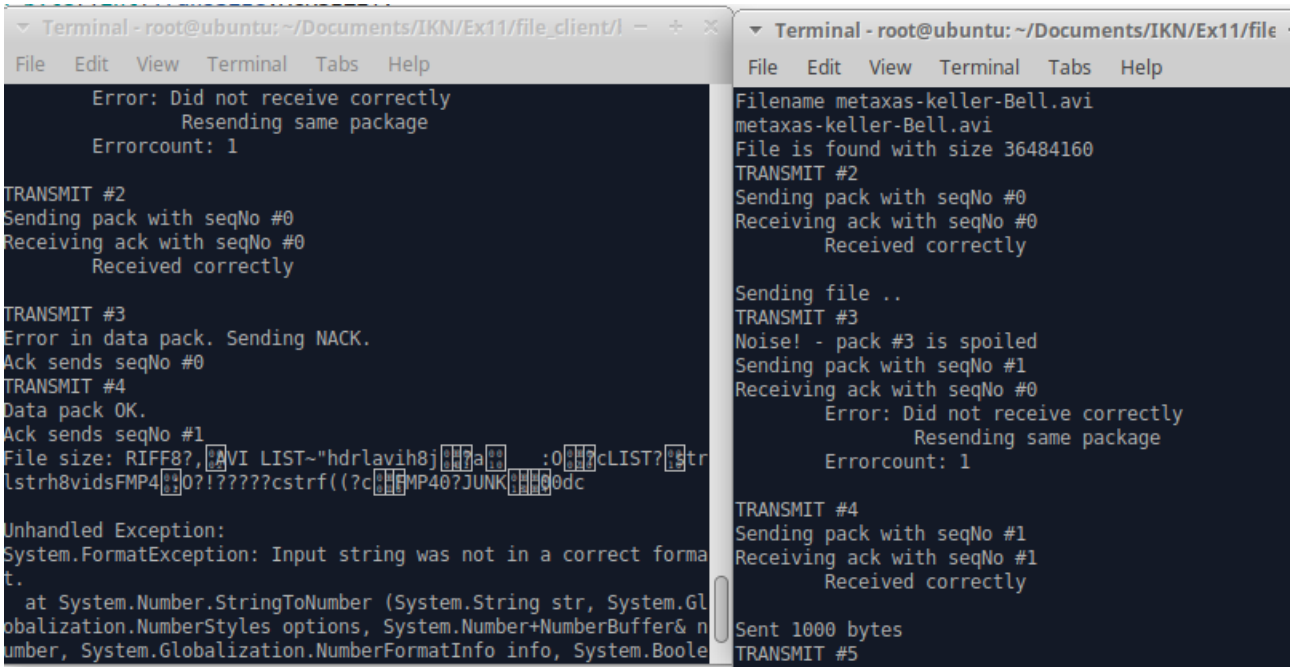
Figur 3 Overførsel af en fil på 36 mb: fungerer fint

Hvis der tilføjes både ack (på 2, transmission) og datafejl, fungerer det stadig fint. Se Figur 4.



Figur 4 Overførsel af fil på 36 mb med både ack og datafejl. Fungerer fint.

Ændres ack-fejlen til at forekomme på den første transmission, sker der en fejl. Se

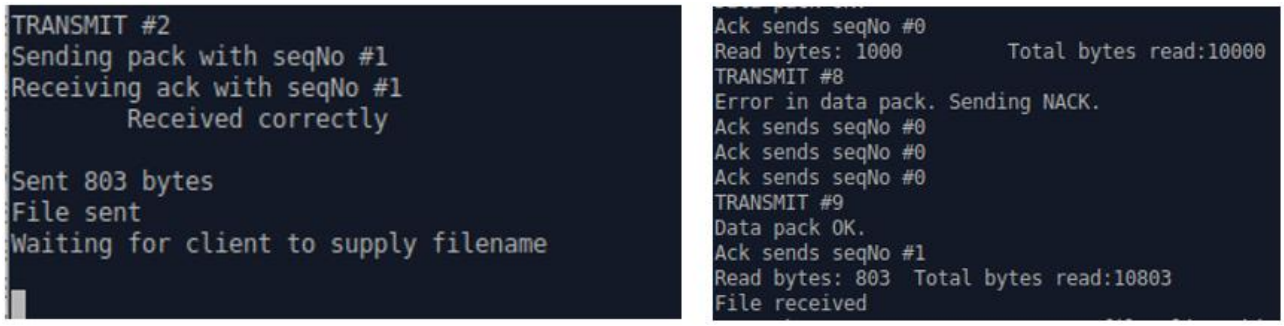


Figur 5 Transmission med ack-fejl på første transmission. Fejl sker.

Ack sker ved transmit #1. Der sker en exception idét at serveren sender den rigtige filstørrelse men klienten modtager med en ack fejl og venter herefter på en ny datapakke. Serveren er i gang med at sende den næste pakke som er filen selv. Dette modtager klienten som ikke er korrekt. Dette er valgt ikke at blive håndteret, så der er ikke indsat ack-fejl (udkommenteret) da denne fejl kan forekomme når der sendes en ack-fejl på den sidste pakke i hver transmission.

Release

Der testes med at overføre en .jpeg med en størrelse på 10803 kB. Se Figur 6.



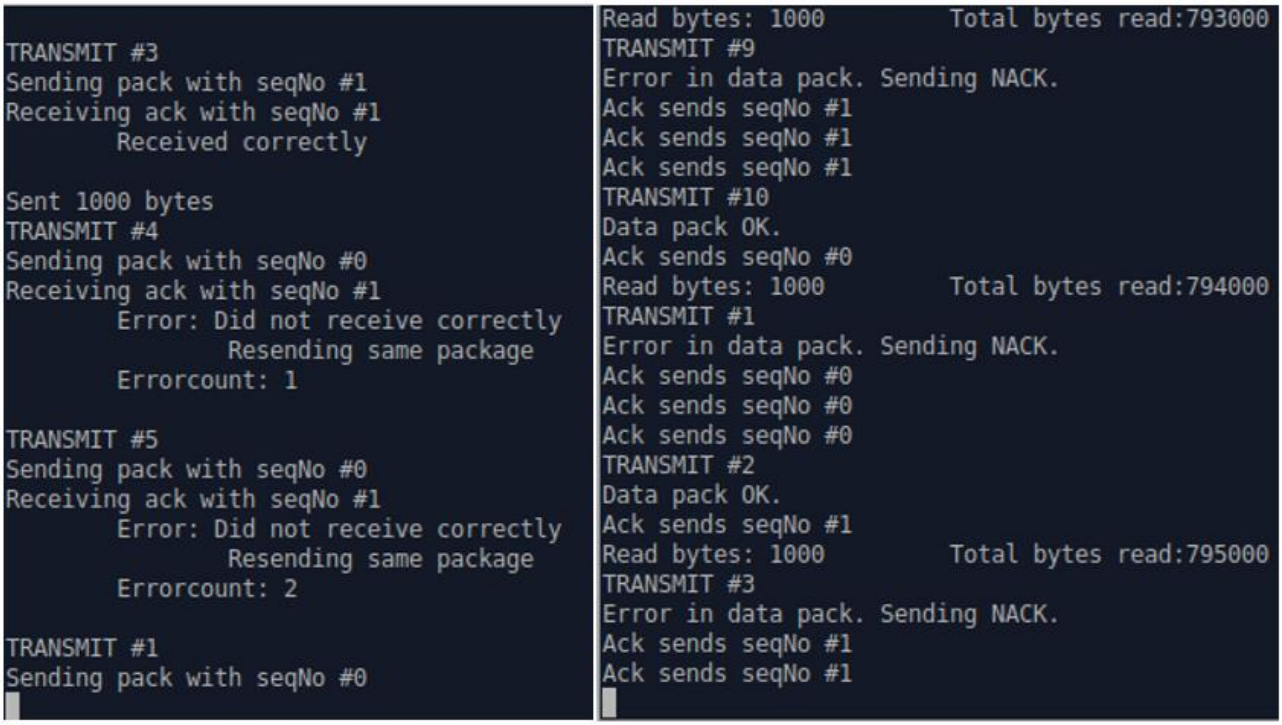
Figur 6 Server (venstre side) har sendt filen. Klienten (højre side) har modtaget filen på 10803 kb.

Filen sendes fint men langsomt. Filen fra serveren overføres til klienten og der kan ses at de to filer er ens på Figur 7.

```
root@ubuntu:~/Desktop/Ex11_TransmittedFiles# diff -s pic.jpeg ~/Documents/IKN/Ex11/pic.jpeg
Files pic.jpeg and /root/Documents/IKN/Ex11/pic.jpeg are identical
root@ubuntu:~/Desktop/Ex11_TransmittedFiles#
```

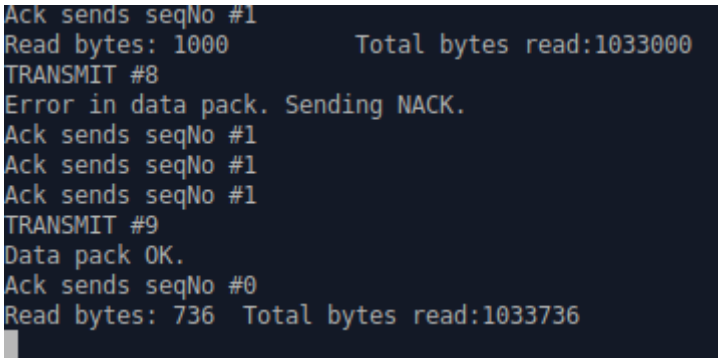
Figur 7 De to filer er ens

Figur 8 viser overførsel er af en større fil på 1mb. Her er det tydeligt at se at der forekommer mange fejl som der skal håndteres. Dette gør sendingen langsom, især med ekstra data-fejl som er påført i nogle datapakker.



Figur 8 Server (venstre) sender filen men der forekommer en error count op til 2. Klienten (højre) sender tit fejl pga. timingsproblemer.

Når serveren er færdig med at overføre har klienten ikke modtaget hele filen, som der ses på Figur 9. Der mistes en del pakker når der skal håndteres større filer. Dog fungerer dette helt fint på loop-back portene som der blev set i forrige afsnit.



Figur 9 Client modtager ikke hele filen: pakker er mistet.

Konklusion

Det er muligt at sende en vilkårlig fil fra klient til server på debug-mode. Dette er ikke helt muligt på release, siden pakker mistes og der forekommer timingsfejl, da senderen sender hurtigere end modtageren kan modtage.

Linklageret havde en uventet fejl. Den håndterede i starten ikke nogle former for flow control, det betyder at link lageret ikke tager højde for at det er muligvis en hurtig sender, men langsom modtager. Så dataerne bliver gensendt for hurtigt og laver fejl, der får senderen til at timeout før modtageren har modtaget filerne. Dette blev håndteret på bedste vis.

Det er muligt at håndtere både data- og ackfejl, men ackfejl kan ikke altid håndteres i nogle situationer. Håndteringen af denne situation er ikke implementeret da opgaven ikke omfatter dette.