PROTOCOL STACK - FILOVERFØRSEL

TCP/IP SOCKET PROGRAMMING

Deltagere

201509378 – Niklas Meyer Møller Sørensen

201507686 – Jeppe Traberg Sørensen

20101979 – Henrik Søby Jørgensen

16. MAJ 2017

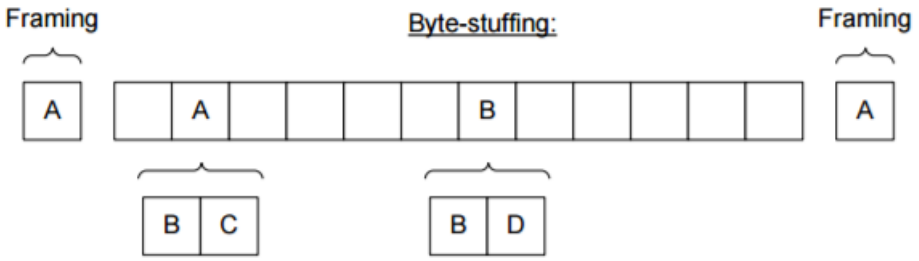
Problemformulering:

Der skal udvikles en mulighed for at overføre en fil vha. den serielle kommunikationsport på en virtuel maskine. Systemet skal kunne overføre en fil af vilkårlig størrelse fra en virtuel maskine til en anden. Dette skal gøres via to applikationer en server og en client.

Clienten skal fortælle serveren hvad fil som skal overføres og serveren skal sende den pågældende fil tilbage med 1000bytes af gangen.

Datalag - Link lag

Linklaget skal implementeres som en SLIP protokol. Karakteren 'A' vil bliver brugt som start og stop karakter, også kaldet Framing. Dette betyder at når et rigtig 'A' sendes skal dette erstattes med BC og 'B' skal erstattes med BD.



Figur 1. SLIP-protokol i linklaget. Kilde:

I linklagets send funktion bliver et Byte[] kaldet sendBuff. Dette bruge stil at gemme det nye kodede byte array som skal sendes. Den buffer som ønskes sendt bliver løbet et foreach og de pågældende chars bliver udskiftet. Sidst bliver det nye array sendt via en SerialPort.

```
public void send (byte[] buf, int size)
       {
            int i = 1;
            byte[] sendBuf = new byte[size * 2 + 2];
            sendBuf[0] = DELIMITER;
                  for (int k = 0; k < size; k++)
                  {
                      if (buf[k] == (byte)'A')
                      {
                          sendBuf[i] = (byte)'B';
                          i++;
                          sendBuf[i] = (byte)'C';
                      }
                      else if (buf[k] == (byte)'B')
                      {
                          sendBuf[i] = (byte)'B';
                          i++;
                          sendBuf[i] = (byte)'D';
                      }
                      else
                      {
                          sendBuf[i] = buf[k];
                      }
                      i++;
                  }
            sendBuf[i] = DELIMITER;
                     serialPort.Write (sendBuf, 0, sendBuf.Length);
        }
```

Figur 2. Send-funktion i linklaget.

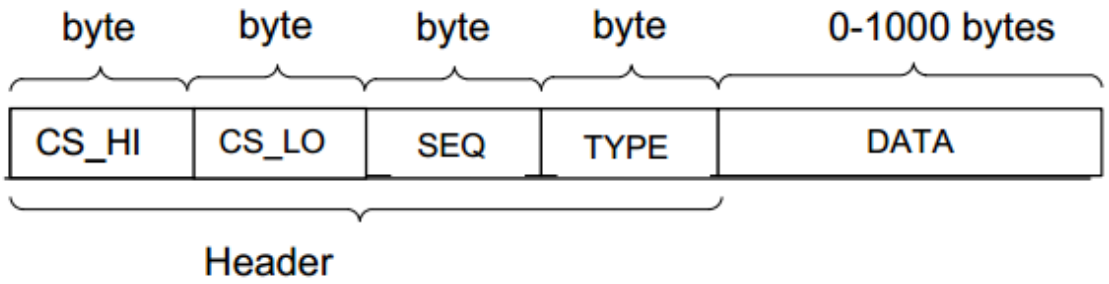
Recieve() i linklaget vil modtage en kodet buffer som skal dekodes. Her er et udsnit af det loop som læser af den medtagede buffer. Delimiter er A og er allerede taget ud af bufferen. Dette gør at der nu kan løbes igennem hele bufferen indtil det næste A bliver fundet. Under dette gennemløb bliver bufferen bliver dekodet og gemt i et byte[] temp.

```
while (buffer[0] != DELIMITER)
{
    if (buffer[0] == B)
    {
        serialPort.Read(buffer, 0, 1);
        if (buffer[0] == C)
        {
            temp.Add(DELIMITER);
        }
        else if (buffer[0] == D)
        {
            temp.Add(B);
        }
        else{
          throw new Exception("Recieved message not formatted correctly");
        }
    }
    else
    {
        temp.Add(buffer[0]);
    }
```

Figur 3. Receive-funktion i linklaget.

Transportlag

I transportlaget skal der implementeres en stop and wait protokol. Denne skal indeholde en 16 bit internet checksum så det kan detekteres hvis der opstår fejl. Der skal opsættes en payload på 1000 bytes og ligeledes overholde header formaterne:



Figur 4. Tilføjelse af header i transportlaget. Kilde: Slides fra "Oplæg til Øvelse 12 2. del"

Her er CS_HI og CS_LO henholdsvis MSB og LSB af checksumsberegningen.
SEQ er et sekvensnummer på det afsendte segment og typen indeholder enten et 1 eller 0. 1 for en ACK besked og 0 for data. Data, den payload der ønskes sendes, er på 1000 bytes.

Send-metode

```
public void send(byte[] buf, int size)
{
    do
    {
        buffer[(int)TransCHKSUM.SEQNO] = (byte)seqNo;
        buffer[(int)TransCHKSUM.TYPE] = (byte)TransType.DATA;
        Array.Copy(buf, 0, buffer, 4, size);

        checksum.calcChecksum(ref buffer, size);
        /*
        if (++errorCount == 2)
        {
            buffer[1]++;
            Console.WriteLine("NOISE! - Byte #2 is damaged in the first trans-mission!");
        }
        */
        link.send(buffer, size + 4);
    } while (!receiveAck());

    old_seqNo = DEFAULT_SEQNO;
}
```

Figur 5. Send-funktionen i Transport layer.

Vores send-metode arbejder på et buffer-array. Dette array er globalt defineret i transportlaget. Vi bruger denne til at indsætte vores sekvens, type og checksumsnummer i. Herefter kopierer vi data fra buf-parameteren over i den globalt defineret buffer. Først indsættes sekvensnummer på index 2, herefter datatype på plads 3. Vi kopierer nu vores payload (buf) over i bufferen. Payloaden indsættes i bufferen fra index 4 og frem med Array.Copy(). Herefter bruger vi calcChecksum i Checksum-klassen til at beregne checksummen for bufferen. Denne beregnes ud fra sekvensnummer og datatypen og indsættes på index 0 og 1 i bufferen.

Vi har nu fået tilføjet vores header og sender den med linklagets send-funktion. Hele dette er pakket ind i en do-while løkke, der fortsætter så længe resultatet fra receiveAck-metoden er false. Dette medfører at løkken først stopper, når der modtages et gyldigt acknowledge fra modtageren.

Receive-metode

Receive funktionen i transportlaget skal modtage data og vurdere om transmitteringen er OK. Dette realiseres ved at den modtager data ved hjælp af linklaget indtil checksum og sekvensnummer er som forventet.

Det gamle sekvensnummer (fra forrige pakke) gemmes, således der kan tjekkes at det ikke er dette sekvensnummer der modtages i den aktuelle pakke. Hvis dette er tilfældet er der nemlig gået noget galt i transmitteringen, og der sendes derfor et NACK. Disse transmitteringsfejl simuleres fra receivesiden i sendAck funktionen, ved at 2. byte bliver beskadiget.

En vigtig detalje i slutningen af funktionen er, at der kun sendes den betydende del af dataen tilbage, dvs. uden de 4 første bytes som indeholder sekvensnummer, checksum og type.

```
public int receive(ref byte[] buf)
        {
            bool status = false;
            do
            {
                int size = link.receive(ref buffer);
                status = checksum.checkChecksum(buffer, size-4);

                sendAck(status);

                if (status == true)
                {
                    Array.Copy(buffer, 4, buf, 0, size - 4);
                    return size;
                }

            } while (status == false);

            return 1;
        }
```

Figur 6. Receive-metode i transportlaget.

Applikationslag

Applikationslaget er meget udledt af den tidligere opgave fra øvelse 7/8. Dog gøres der nu brug af transportlaget i både FileServer og FileClient.

FileServer:

I FileServeren ligger forskellen i at et Transport object bliver oprettet og dennes send() metode bliver brugt til at overføre vores bytes.

```
        private void sendFile(String fileName, long fileSize, Transport transport)
                {
              using (FileStream SourceStream = File.Open(fileName, FileMode.Open))
              {
                  long sentBytes = 0;
                  byte[] data = new byte[BUFSIZE];
                  int count = 0;

                  while (fileSize > sentBytes)
                  {
                      count = SourceStream.Read(data, 0, data.Length);
                      transport.send(data,count);
                      sentBytes += count;
                      Console.WriteLine(sentBytes);
                  }
                  Console.WriteLine("File sent! Bytes sent: {0}", sentBytes);
              }
                }
```

Figur 7. FileServer i applikationslaget.

FileClient:

I FileClient ligger ændringen ligeledes i at et Transport object bliver oprettet. Dennes recieve() metode bliver brugt til modtagelse af filen.

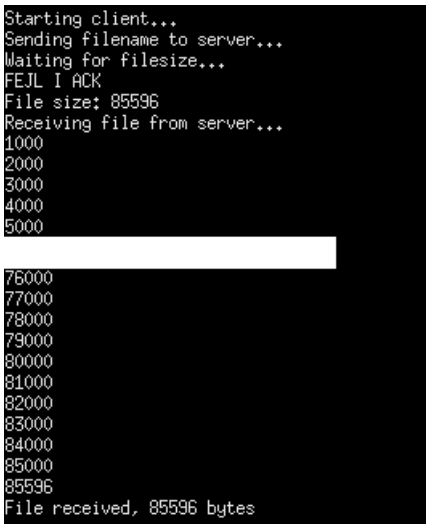
```
private void receiveFile (String fileName, Transport transport)
                {
                     Console.WriteLine("Receiving file from server...");
              FileStream fs = new FileStream(fileName, FileMode.Create,
FileAccess.Write);

              long receivedBytes = 0;
                    long fileSizeLong = Int32.Parse(fileSize);
              int count = 0;
              byte[] data = new byte[BUFSIZE];

              while (fileSizeLong > receivedBytes)
              {
                  count = transport.receive(ref data);
                  fs.Write(data, 0, count);
                  receivedBytes += count;
                  Console.WriteLine(receivedBytes);
              }
          }
```

Figur 8. FileClient i applikationslaget.

Test

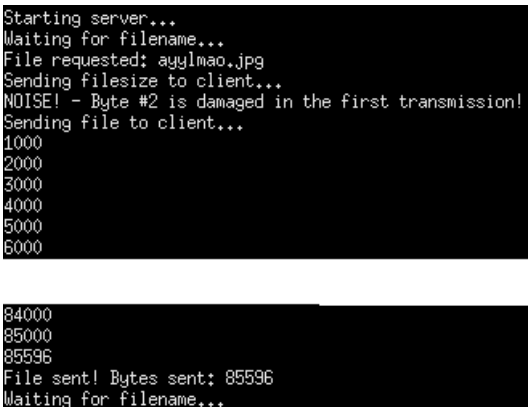
Vi har gennemtvunget nogle fejl for at kunne teste når der sker i fejl i Acknowledge og bitfejl i transportlaget. Fejlene er blevet introduceret ved hjælp af det udleverede materiale[1]. Skal simulere disse forskellige fejl.

Først fra FileClient vil vi sende et filnavn til serveren, her tvinge en enkelt ACK fejl og derefter fortsætte som forventet.



Figur 9 - Test af FileClient

Ligeledes på FileServer fremtvinges en bitfejl for at teste funktionalitet, og derefter forsætter overførslen af filen som forventet.



Figur 10 - Test af FileServer

[1] https://bb.au.dk/bbcswebdav/pid-755684-dt-content-rid-2002039_1/xid-2002039_1

Konklusion

Der er igennem opgaven opnået en dybdegående forståelse for, hvordan en protokolstack opbygges. Vi har i opgaven formået at opbygge et applikations-, transport- og linklag.

Vi har i linklaget implementeret en SLIP protokol, der fungerer som kravene i opgaven.

I transportlaget har vi implementeret en stop-and-wait protokol, der anvender en 16-bit internet-checksum til fejldetektering. Vi har valgt ikke at implementere timeout-funktionaliteten. Den maksimale payload er sat til 1000 bytes.

I applikationslaget er lavet et program, der tester og bekræfter at vores protokolstack virker. Laget ligner meget øvelse 7/8. Dette skyldes, at vi i denne opgave har ændret lagene under applikationslaget i forhold til øvelse 7/8, derfor bør de to applikationslag også ligne hinanden, da interfacet til lagene under er det samme.