IKN Øvelse 13 – Journal

Indholdsfortegnelse

UDVIKLING AF PROTOCOL STACK - FILOVERFØRSEL VIA RS-232 NULL-MODEM

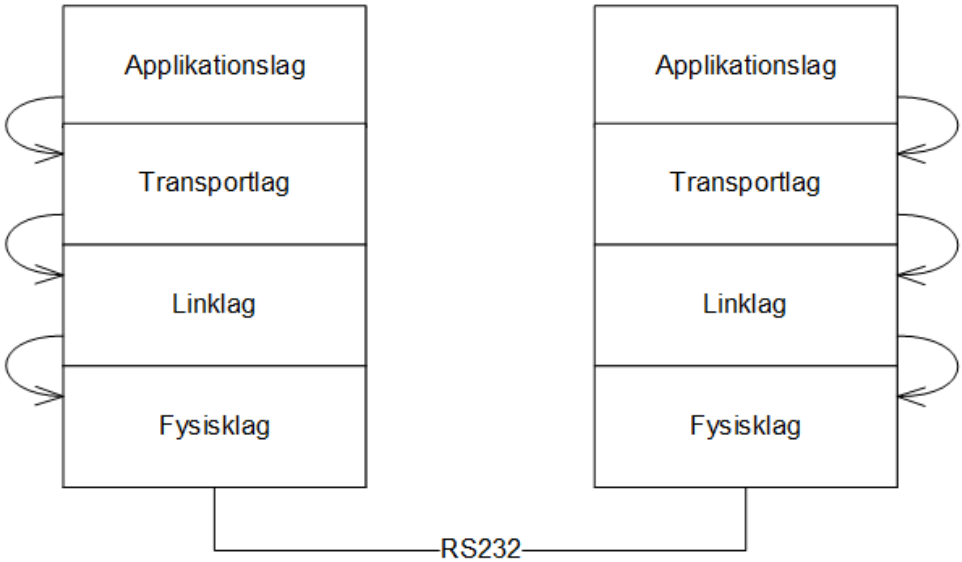
I denne øvelse skal designes og implementeres mulighed for at overføre en fil vha. den serielle kommunikations-port i en virtuel maskine. Det serielle interface er i denne øvelse et RS-232 interface. Microcontroller-baseret embedded udstyr har ofte kun mulighed for at kommunikere med omverdenen via et serielt interface. Derfor er problematikken i denne øvelse relevant. Det er yderligere yderst lærerigt at udvikle en protocol stack, hvilken er hovedformålet med øvelsen. Systemet skal give mulighed for at der fra en virtuel computer (H1) kan overføres en fil af en vilkårlig type/størrelse til en anden virtuel computer (H2).

Der skal designes, implementeres og testes to applikationer, en client og en server. Førstnævnte kan anbringes i H1, sidstnævnte kan anbringes i H2.
Den ene applikation (client) skal meddele serveren hvilken fil (evt. incl. sti-angivelse) der skal hentes. Den anden applikation (server) skal læse og sende filen til klienten i pakkestørrelser på 1000 bytes payload ad gangen. Client skal modtage disse pakker og gemme dem i en fil.
Såvel server som client er nemme at realisere, idet disse "applikationslags- applikationer" allerede er udviklet i øvelse 8 (TCP-baseret client/server).

I øvelse 8 blev der vha. anvendelse af socket-API udført kald som etablerede en connection, overførte filnavn, filstørrelse og fil mellem client og server. Disse kald skal i denne øvelse udskiftes med kald til et transportlaget i en protocol-stack, som I selv udvikler. Transportlaget skal være pålideligt med en funktionalitet, som svarer til rdt v.2.2 i lærebog/slides.
Client og server kan også have samme brugerflade som client-/server-applikationerne i øvelse 8 (TCP/IP-baseret filoverførsel). Kilde: Forsøgsbeskrivelsen

Fysisk lag – RS232

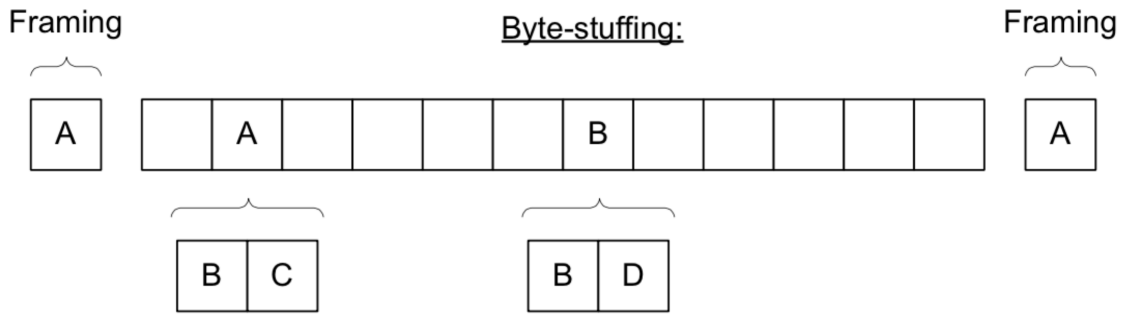
Det fysiske lag i protokol stacken er selve serielforbindelsen (RS232) imellem de 2 enheder der er forbundet.
I vores tilfælde 2 virtuelle Linux maskiner forbundet med hver deres fysiske USB-Serial konverter og et krydset NULL-MODEM kabel imellem, da Mac OSX og Parallels Desktop ikke kan håndtere virtuelle serielporte særlig godt. (Sidenote: husk at disconnecte kablerne i Parallels Desktop inden usb-kablet trækkes ud af computeren, ellers crasher operativsystemet, og lad vær med at bruge Straight RS232 kabler)



Figur 1 - Protokolstack opbygning

Link lag - SLIP protokol

SLIP protokollen skal implementeres i link laget. Protokollen skal sørge for at start og stop karakteren er 'A', kaldes frames. Når 'A' optræder i telegrammet skal det erstattes at de to karakterer 'B' efterfulgt af 'C' og hvis 'B' optræder i telegrammet erstattes denne også at to karakterer 'B' efterfulgt af 'D', dettes kaldes Byte Stuffing.



Figur 2 - Byte stuffing

Implementering af SLIP protokollen

Linklaget består af en send og receive funktioner, som skulle implementeres med byte stuffing, dvs. at send funktionen skal pakke data fra transportlaget ind i en frame med karakteren 'A' i begge ender, så receive funktionen kan genkende data når det modtages.

Send funktion – Nedpakning vha. SLIP

Først sendes der et 'A' ud på serielporten

```
. . .
unsigned char begin[1] = {'A'} ;
int n = v24Write(serialPort,begin,1);
        if(n != 1) {
                std::cout       << "ERROR("
                                << v24QueryErrno
                                << "): Failed to write to serialPort, bytes written: "
                                << n
                                << std::endl;
        }
. . .
```

Derefter pakkes den modtaget buffer ned så A = BC og B = BD

```
. . .
for(int i = 0; i < size; i++) {
        if (buf[i] == 'A') {
                buffer[cnt] = 'B';
                cnt++;
                buffer[cnt] = 'C';
                cnt++;
        }
        else if (buf[i] == 'B') {
                buffer[cnt] = 'B';
                cnt++;
                buffer[cnt] = 'D';
                cnt++;
        }
        else {
                buffer[cnt] = buf[i];
                cnt++;
        }
}
. . .
```

buffer sendes ud på serielporten

```
. . .
n = v24Write(serialPort,buffer,cnt);
        if(n != cnt) {
                std::cout       << "ERROR("
                                << v24QueryErrno
                                << "): Failed to write to serialPort, bytes written: "
                                << n
                                << std::endl;
        }
. . .
```

til sidst sendes det sidste 'A' ud på serielporten, på samme måde som det første 'A'.

Receive funktion – Udpakning vha. SLIP

Char arrayet der bruges skal være 2006 Bytes. 2*BUFSIZE+HEADER, headeren kan i værste tilfælde være 6 Bytes hvis checksummen er AA.

```
unsigned char tempBuf[size*2+6];
```

Først læses der på serielporten indtil første 'A' karakter mødes

```
. . .
while(v24Getc(serialPort) != 'A');
. . .
```

derefter læses én Byte ad gangen til tempBuf og counter incrementeres indtil en 'A' karakter læses. Dette medfører at der nu ligger det rå data i tempBuf uden 'A' termineringer.

```
. . .
while(1) {
        tempBuf[counter] = v24Getc(serialPort);
        if(tempBuf[counter] == 'A')
                break;
        else
                counter++;
        }
. . .
```

derefter læses den reelle data størrelse ud ved at tælle alle karakterer der ikke er 'B'

```
. . .
for(int i=0 ; i < counter ; i++) {
        if(tempBuf[i] != 'B')
                realSize++;
}
. . .
```

nu kan det reelle data pakkes ud ved at loope i en løkke realSize gange, og konvertere BC og BD til hhv. 'A' og 'B' karakterer. Den udpakkede data ligges direkte i det modtagede buf array som er en pointer givet af transportlaget

```
. . .
counter = 0;
for(int i=0 ; i < realSize ; i++) {
        if(tempBuf[counter] == 'B' && tempBuf[counter+1] == 'C') {
                buf[i] = 'A';
                counter+=2;
        }
        else if(tempBuf[counter] == 'B' && tempBuf[counter+1] == 'D') {
                buf[i] = 'B';
                counter+=2;
        }
        else {
                buf[i] = tempBuf[counter];
                counter++;
        }
}
. . .
```

til sidst returneres realSize

```
. . .
return realSize;
```

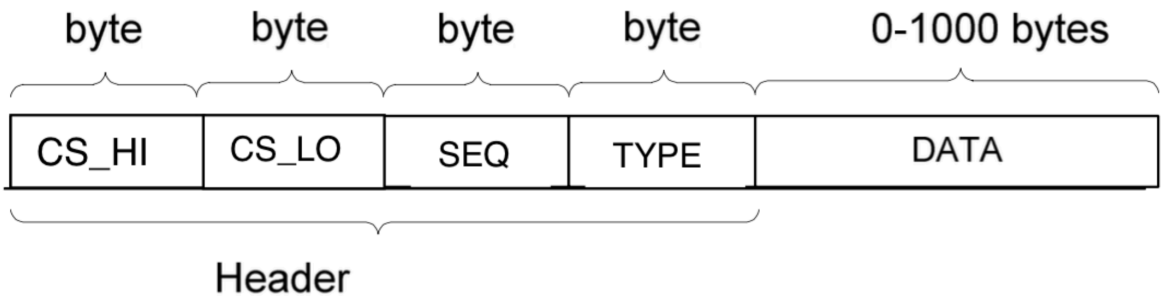
Transportlag og Checksum

Transportlagets opgave er at være et mellemled imellem applikationslaget og linklaget, det sørger for sikker data overførsel ved at tilføje en header til dataen med en checksum, et sekvensnummer og en typegenkendelse. Det sørger også for at sende og modtage ACK mellem client og server.

Implementering af Transportlaget

Transportlaget har ligeledes en send og en receive funktion som skulle implementeres. Send funktionen skal sørge for at header og data kobles sammen og sendes videre til linklaget og derefter modtage en ACK for korrekt dataoverførsel. Receive funktionen skal modtage data fra linklaget, tjekke om dataen er valid iht. checksummen og sende en ACK hvis dette er tilfældet.

Headeren skal ved DATA overførsel bestå af et 1 tal i TYPE, et SEKVENS nummer og en kalkuleret checksum på 2 Bytes.



Figur 3 - Transportlags header+data

Send funktion – header tilførsel

Send funktionen er bygget op i en do-while løkke der tager den modtaget buf data og ligger i transport klassens egen buffer, men først fra plads 4. Da der på plads 2 og 3 hhv. placeres sekvensnummer og typen DATA(1). Derefter tages den i forvejen implementeret calcChecksum() funktion i brug og regner en checksum ud der placeres på plads 0 og 1 i buffer arrayet. Alt dette data sendes videre til linklaget indtil der modtages en ACK.

```
. . .
 do {
        for(int i = 0 ; i < size ; i++) {       // Add data from buffer[4]
                buffer[i+4] = buf[i];
        }

        buffer[SEQNO] = seqNo; // Tilføj sekvensnummer
        buffer[TYPE] = DATA;      // Tilføj DATA byte <0>

        checksum->calcChecksum(buffer,size+4); // Calculate checksum

        link->send(buffer,size+4);
}while(!receiveAck());
. . .
```

Receive funktion – checkChecksum

Modtager funktionen er opbygget i en while(1) løkke, der som det første modtager data til klassens buffer array. Derefter tjekkes checksummen og hvis den er god sendes der en ACK, men der tjekkes også på om sekvensnummeret er korrekt og hvis det er korrekt ændres det og bufferen kopieres over i det modtaget buf pointer array fra applikationslaget, derefter returneres den reelle datastørrelse.

Hvis sekvens nummeret ikke passer startes løkken forfra.

```
. . .
while(1) {
        short n = 0;

        n = link->receive(buffer,size+4);

        while(!checksum->checkChecksum(buffer,n)) {
                sendAck(false);
                n = link->receive(buffer,size+4);
                printf("[Transport.receive] Checksum error\n");
        }

        sendAck(true);

        if(seqNo == buffer[SEQNO]) {
                seqNo = (buffer[SEQNO] + 1) % 2;
                for(int i=0 ; i < n-4; i++) {
                        buf[i] = buffer[i+4];
                }
                return n-4; // data byte size minus header
        }
}
. . .
```

Applikationslaget – Klient og Server

Applikationslaget har en klientside og en serverside. Klientsiden skal lave et forespørgsel på en fil, hvorefter en filstørrelse returneres efterfulgt af selve filen. Serversiden skal kigge på om filen eksisterer, sende filstørrelsen og derefter sende filen i 1000 Byte bider.

Implementering af applikationslaget

File_client

File_client er en constructor kaldet fra main, i vores klient skal der modtages et argument i form af filnavnet der ønskes hentet og dette laves der fejltjek på i main

```
. . .
if (argc < 2) {  // Syntax check
        error("[App.client] The server needs 1 argument <filename>\n");
}
new file_client(argc, argv);
. . .
```

det modtaget argument bruges i file_client(). I file_client() kaldes 2 funktioner og oprettes et Transport objekt

```
. . .
Transport::Transport transport(BUFSIZE);

/* Write fileName to Server */
transport.send(argv[1],strlen(argv[1]));

// Get the damn file
receiveFile(argv[1], &transport);
. . .
```

ReceiveFile

Det første der skal ske i recieveFile() funktionen er at fil størrelsen i Bytes skal findes. Der kaldes en receive til buffer, som konverteres om til en long

```
. . .
char buffer[BUFSIZE];
long fileSize;
. . .
int n = transport->receive(buffer,BUFSIZE);  // Get size of file to transfer
fileSize = atol(buffer);
. . .
```

Hvis filstørrelsen ikke er 0 eller derunder konverteres filnavnet til en const char* og der oprettes en fil pointer fp med filnavnet som der kan skrives til.
Der modtages nu data i en do-while løkke som skrives i filen med fwrite() indtil den samlede filstørrelse er på størrelse med den modtaget filstørrelse fra serveren og derefter lukkes filpointeren.

```
. . .
if(fileSize <= 0)
        error("[App.client] No such file on Server\n");
else {
        cout << "[App.client] Filesize: " << fileSize << " bytes" << endl;

        fileName = extractFileName(fileName); // Extract filename from path
        cout << "[App.client] Receiving <" << fileName << "> from Server" << endl;
        const char* fileReceiveName = fileName.c_str(); // Parse fileName to const char *ptr

        FILE * fp = fopen(fileReceiveName,"wb"); // Write new file or overwrite file

        if(fp == NULL)
                error("[App.client] File cannot be opened.\n");
        else {
                bzero(buffer, BUFSIZE); // Clear buffer

                // Get the file
                int n;
                long n_size = 0;

                do{
                        n = transport->receive(buffer,BUFSIZE); // Read block of data from stream
                        fwrite(buffer,1,n,fp);        // Write blocks from buffer 1 byte n times to fp
                        bzero(buffer,BUFSIZE);    // Clear buffer
                        cout << "[App.client] transfer size: " << n << " Byte(s)" << endl;
                        n_size += n;
                }while(n_size != fileSize);

                cout << endl << "[App.client] Ok received from server!" << endl;
                fclose(fp);     // Close file
        }
}
. . .
```

File_server

File_server er en constructor kaldet fra main(). Det første den skal er at læse filnavnet sendt fra klientsiden.

```
. . .
Transport::Transport transport(BUFSIZE);
char buffer[BUFSIZE];

// Read filename
bzero(buffer,BUFSIZE);
cout << "[App.server] waiting for filename" << endl;
int n = transport.receive(buffer,BUFSIZE);
. . .
```

herefter tjekkes der på om filen eksisterer med den allerede implementeret check_File_Exist() funktion som returnere filens størrelse

```
. . .
string fileName = string(buffer);
fileName = extractFileName(fileName);

// Check file exist
long fileSize = check_File_Exists(fileName);
. . .
```

så konverteres filstørrelsen til char bufferen og sendes til klienten som karakterer

```
. . .
sprintf(buffer, "%ld", fileSize);
transport.send(buffer,strlen(buffer));
. . .
```

og til sidst sendes filen med sendFile()

```
. . .
sendFile(fileName,fileSize,&transport);
. . .
```

SendFile

SendFile funktionen skal sende filen til klienten i bider af max 1000 bytes. Først oprettes en const char* fileSendName som bliver tildelt filnavnet derefter oprettes en fil pointer fp som åbner filen og gør klar til at læsning på denne.

I en while løkke læses og sendes filindholdet så længe der er data der kan læses fra filen. Derefter lukkes filpointeren.

```
. . .
const char* fileSendName = fileName.c_str(); // Parse to const char *ptr
char buffer[BUFSIZE];                                    // Send buffer
FILE *fp = fopen(fileSendName, "rb");              // Make filepointer rb = read binary
if(fp == NULL)
        error("[App.server] ERROR: No such file on Server\n");

cout << "[App.server] Sending <" << fileName << "> to the Client..." << endl;
bzero(buffer, BUFSIZE); // Clear buffer

int n; // Counter
while((n = fread(buffer, 1, BUFSIZE, fp))>0)
{
        transport->send(buffer, n);       // Send file in 1000 byte blocks
        cout << "[App.server] transfer size: " << n << " Bytes" << endl;
        bzero(buffer, BUFSIZE); // Clear buffer
}

fclose(fp);
cout << "[App.server] Ok sent to client!" << endl;
. . .
```

Test

Det lykkedes fejlfrit at overføre forskellige billedefiler på få kilobytes fra server til klient.

Vi lavedes også en stresstest på systemet med en overførsel på 13 MB(12996104 Bytes), det tog et sted imellem 15 og 20 minutter at overføre filen. Dette passer udmærket med en baud-rate på 115200.

$$\frac{115200\ bit/s}{8\ bit} = 14{,}4 kiloBytes/s$$

$$\frac{12996104\ Bytes}{14400\ Bytes/s} = 902{,}5\ s \cong 15\ min$$

Der blev lavet 12.996 godkendte overførsler af 1000 Bytes + en på 104 Bytes og der var 97 fejl undervejs dette giver en fejlrate på 0,746 % så det må siges at være en meget pålidelig dataoverførsel.

Konklusion

Det har været meget udfordrende at skulle implementere sin egen protokol stack. Vi har dog lært enormt meget ved at selv skulle skrive stort set alt koden. Men ikke uden at det fra tid til anden har været lidt frustrerende når der opstod uforklarlige fejl.

Vi savner at det ikke kun er underviseren der er tilstede for at hjælpe på dage hvor vi har kunnet få hjælp. Men ellers har det været brugbart når der endelig var folk til at give hints til hvad der kunne være galt.

Yderst god opgave stillet af underviser. Vi har bare brugt ALT for meget unødigt tid på den, som kunne være blevet brugt andre steder i studiet.

Bilag

Vi har valgt at inkludere kildekoden i deres respektive filer sammen med denne rapport