

# I4IKN- Øvelse 12

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

Jonas Lind	201507296
Tais Hjortshøj	201509128

## Indhold

Problemformulering .....	3
Protokolstakken .....	3
Linklaget .....	5
Send-metode .....	5
Recieve-metode.....	6
Test .....	7
Transportlaget .....	9
Send-metode .....	9
Recieve-metode.....	10
Applikationslaget .....	10
Konklusion .....	10

## Problemformulering

I denne øvelse skal der designes og implementeres muligheden for at overføre en fil vha. den serielle kommunikationsport i en virtuel maskine. Det serielle interface er 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 desuden yderst lærerigt at udvikle en protocol stack, hvilket er hovedformålet med øvelsen.

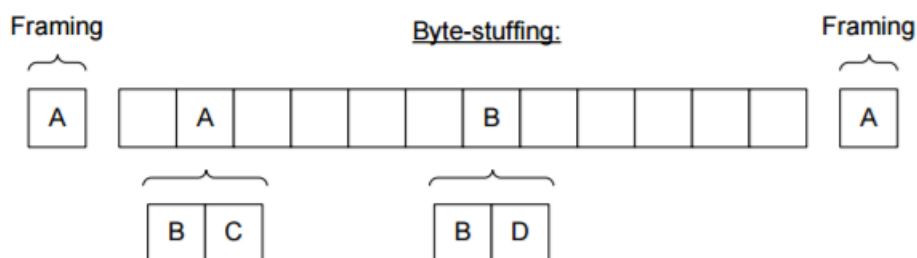
Systemet skal give mulighed for at der fra en virtuel computer (H1) kan overføres en fil af en vilkårlig type og størrelse til en anden virtuel computer (H2). Der skal designes, implementeres og testes to applikationer, en server og en client. Førstnævnte anbringes i H1, sidstnævnte kan anbringes i H2. Client applikationen skal meddele serveren hvilken fil der skal hentes. Server applikationen skal læse og sende filen til clienten i pakkestørrelser på 1000 bytes payload ad gangen. Client skal modtage disse pakker og gemme dem i en fil.

Server og client applikationerne er udviklet i I4IKN Øvelse 7 (TCP-baseret client/server). I Øvelse 7 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 lærebogen. Client og server kan også have samme brugerflade som client-/server-applikationerne i øvelse 7 (TCP/IP-baseret filoverførsel).

## Protokolstakken

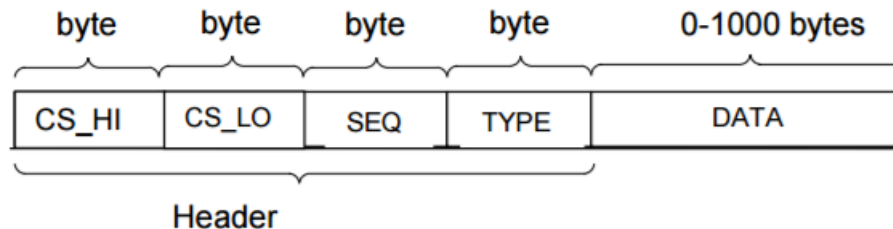
**Det fysiske lag:** Dette lag er givet ved seriel kommunikation via RS-232 porte og via et null-modem (/dev/ttyS1).

**Datalinklag:** Til dette lag skal I implementere en modificeret SLIP protokol. Protokollen skal implementeres således: Som start og stop karakter benyttes 'A'. Hvis karakteren 'A' forekommer i frame erstattes det med de to tegn 'B' og 'C', og hvis tegnet 'B' forekommer, erstattes det med de to tegn 'B' og 'D'.



**Netværkslag:** Dette lag findes ikke, da vi kun anvender point-to-point kommunikation (dvs. kommunikation uden routing).

**Transportlag:** I skal implementere en stop-and-wait protokol. Protokollen skal være pålidelig og skal anvende en 16 bit Internet-checksum til fejldetektering. Det er et krav at protokollen skal kunne håndtere ødelagte data (f.eks. pga. EMC). Det er ikke et absolut krav at protokollen skal kunne håndtere mistede data (timeout, rdt v. 3.0 ifølge lærebogen) men implementer timeout-funktionaliteten hvis tiden er til det - få stop-and-wait protokollen til at fungere først. Den maksimale længde af transportlagets data (payload) sættes til 1000 bytes.



<b>CS_HI</b>	Den mest betydende del af checksums-beregningen.
<b>CS_LO</b>	Den mindst betydende del af checksums-beregningen.
<b>SEQ</b>	Sekvensnummer på det afsendte segment.
<b>TYPE</b>	0 (DATA)
	1 (ACK)
<b>DATA</b>	Payload på mellem 0-1000 bytes. Skal altid være af længden 0 ved ACK.

**Applikationslaget:** Her skal client sende et filnavn (defineret af brugeren) til server – og server skal returnere filstørrelse og selve filen til client via transportlaget.

## Linklaget

### Send-metode

Et *for-loop* oprettes, den looper så længe  $i < \text{size}$ . For at udskifte karakteren 'A' med 'BC' oprettes et *if-loop*. For at udskifte karakteren 'B' med 'BD' oprettes et *else-if-loop*. Hvis buf index  $i = 'A'$ , så bliver buffer-arrayet  $k$  til 'B' og 'C' samt buffer  $k$  tælles op. Hvis ikke ovenstående *if-loop* er tilfældet, så tjekkes om buf index  $i = 'B'$ . Hvis dette er tilfældet, så sættes buffer-arrayet  $k$  til 'B' og 'D' samt buffer tælles op. Til sidst oprettes et *else-loop*. Hvis ingen af de ovenstående scenarier er tilfældet, så sættes buffer index  $k$  til at være lig med buf index  $i$  samt bufferen tælles op. For at sætte start og stop karakter sættes bytearrayet *buffer* på index 0 til karakteren 'A' og til slut sættes bytearrayet *buffer* på index  $k$  til karakteren 'A'.

```
void Link::send(const char buf[], short size)
{
    //TO DO Your own code
    int i, k = 0, rc;
    std::cout << "LINK: Incomming buf to link: "<< buf << " with size: "<< size <<
std::endl;

    buffer[k] = 'A';
    k++;

    for(i = 0; i < size; i++)
    {
        std::cout << "LINK: k == " << k << ". i == " << i << std::endl;
        if(buf[i] == 'A')
        {
            buffer[k] = 'B';
            k++;
            buffer[k] = 'C';
            k++;
        }
        else if(buf[i] == 'B')
        {
            buffer[k] = 'B';
            k++;
            buffer[k] = 'D';
            k++;
        }
        else
        {
            buffer[k] = buf[i];
            k++;
        }
    }

    buffer[k] = 'A';

    short bufferlength = k;

    rc = v24Puts(serialPort, buffer);
    if (rc < sizeof(buffer))
    {
        fputs("LINK: error: v24Puts failed.\n",stderr);
    }
    else
        std::cout << "LINK: Outgoing buffer: " << buffer << " with size: " <<
bufferlength << std::endl;
}
```

Snippet 1 – Send-metode i Linklaget.

## Recieve-metode

Et *do-while-loop* oprettes, den looper indtil vi modtager startkarakteren 'A'. Endnu et *do-while-loop* oprettes og looper indtil vi har fundet stopkarakteren 'A'. Et *for-loop* oprettes og der tjekkes hvilken karakter der er i buffer-arrayet *i*. Et *else-if-loop* oprettes hvor buffer index *i* = 'B' tjekkes og buffer-arrayet *i* bliver talt op. Et *if-loop* oprettes hvor den næste karakter i buffer-arrayet *i* tjekkes for om det er karakteren 'C' eller 'D'. Hvis den næste karakter er 'C' skrives karakteren 'A' til buf-arrayet *k* samt buf *k* tælles op. Hvis den næste karakter er 'D' skrives karakteren 'B' til buf-arrayet *k* samt buf *k* tælles op. Til sidst oprettes et *else-loop*. Hvis ingen af de ovenstående scenarier er tilfældet, så sættes buf index *k* til at være lig med buffer index *i* samt buf-arrayet *k* tælles op.

```
short Link::receive(char buf[], short size)
{
    //TO DO Your own code
    int i, k = 0, rc, ch, n = 0;

    do
    {
        ch = v24Getc(serialPort);
    }
    while(ch != 'A'); // Find first A

    do
    {
        ch = v24Getc(serialPort);
        buffer[n] = ch;
        n++;
    }
    while(ch != 'A');

    for(i = 0; i < size; i++)
    {
        if(buffer[i] == 'A')
        {
            //i++;
        }
        else if(buffer[i] == 'B')
        {
            ++i;

            if(buffer[i] == 'C')
            {
                buf[k] = 'A';
                k++;
            }
            else //(buffer[i] == 'D')
            {
                buf[k] = 'B';
                k++;
            }
        }
        else
        {
            buf[k] = buffer[i];
            k++;
        }
    }

    return k;
}
```

Snippet 2 – Recieve-metode i Linklaget.

## Test

SLIP protokollen er implementeret på Rx og Tx applikationerne. Linklaget er på client applikationen blevet testet med Snippet 3 – Test kode client. Testen er foretaget for at sikre at SLIP protokollen fungerer som forventet. På Figur 1 omdannes en tekststreng "HEJ JONAS BROTHER" til "AHEJ JONBCS BDROTHERA" i Linklaget. Der er oprettes et bytearray med 26 pladser.

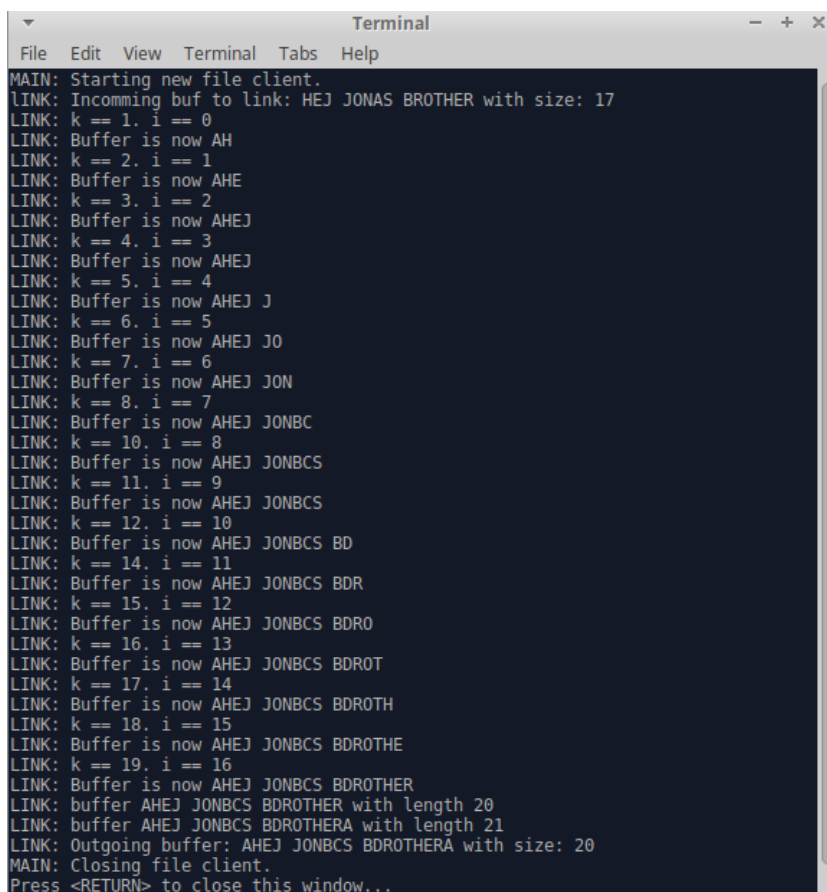
```
int main(int argc, char** argv) // Client
{
    std::cout << "\nMAIN: Starting new file client.\n";

    //new file_client(argc, argv);
    int size = 100;
    //const char msg[] = "HEJ JONAS BROTHER.";
    Link::Link conn(size);
    conn.send("HEJ JONAS BROTHER", strlen("HEJ JONAS BROTHER"));

    std::cout << "MAIN: Closing file client.\n";

    return 0;
}
```

Snippet 3 – Test kode client.



```
Terminal
File Edit View Terminal Tabs Help
MAIN: Starting new file client.
LINK: Incoming buf to link: HEJ JONAS BROTHER with size: 17
LINK: k == 1. i == 0
LINK: Buffer is now AH
LINK: k == 2. i == 1
LINK: Buffer is now AHE
LINK: k == 3. i == 2
LINK: Buffer is now AHEJ
LINK: k == 4. i == 3
LINK: Buffer is now AHEJ
LINK: k == 5. i == 4
LINK: Buffer is now AHEJ J
LINK: k == 6. i == 5
LINK: Buffer is now AHEJ JO
LINK: k == 7. i == 6
LINK: Buffer is now AHEJ JON
LINK: k == 8. i == 7
LINK: Buffer is now AHEJ JONBC
LINK: k == 10. i == 8
LINK: Buffer is now AHEJ JONBCS
LINK: k == 11. i == 9
LINK: Buffer is now AHEJ JONBCS
LINK: k == 12. i == 10
LINK: Buffer is now AHEJ JONBCS BD
LINK: k == 14. i == 11
LINK: Buffer is now AHEJ JONBCS BDR
LINK: k == 15. i == 12
LINK: Buffer is now AHEJ JONBCS BDRO
LINK: k == 16. i == 13
LINK: Buffer is now AHEJ JONBCS BDROT
LINK: k == 17. i == 14
LINK: Buffer is now AHEJ JONBCS BDROTH
LINK: k == 18. i == 15
LINK: Buffer is now AHEJ JONBCS BDROTHER
LINK: k == 19. i == 16
LINK: Buffer is now AHEJ JONBCS BDROTHER
LINK: buffer AHEJ JONBCS BDROTHER with length 20
LINK: buffer AHEJ JONBCS BDROTHERA with length 21
LINK: Outgoing buffer: AHEJ JONBCS BDROTHERA with size: 20
MAIN: Closing file client.
Press <RETURN> to close this window...
```

Figur 1 – Test af Send-metode i Linklag.

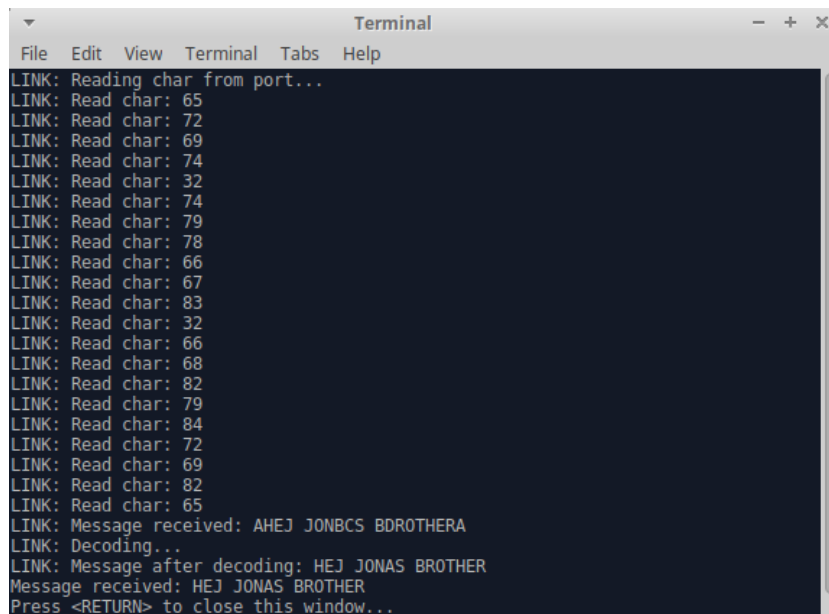
Linklaget er på server applikationen blevet testet med Snippet 4 – Test kode server. På Figur 2 testes senderen i Linklaget. Modtageren står og lytter. Herefter startes programmet i senderen og "AHEJ JONBCS BDROTHERA" ankommer hos modtageren. Den modtagne tekststreng er blevet decoded til "HEJ JONAS BROTHER" efter at have været igennem receive-metoden i linklaget.

```
int main(int argc, char **argv) // Server
{
    //new file_server();
    int size = 100;
    Link::Link conn(size);
    char buff[size];
    conn.receive(buff, size);

    std::cout << "Message received: " << buff << std::endl;

    return 0;
}
```

Snippet 4 – Test kode server.



```
LINK: Reading char from port...
LINK: Read char: 65
LINK: Read char: 72
LINK: Read char: 69
LINK: Read char: 74
LINK: Read char: 32
LINK: Read char: 74
LINK: Read char: 79
LINK: Read char: 78
LINK: Read char: 66
LINK: Read char: 67
LINK: Read char: 83
LINK: Read char: 32
LINK: Read char: 66
LINK: Read char: 68
LINK: Read char: 82
LINK: Read char: 79
LINK: Read char: 84
LINK: Read char: 72
LINK: Read char: 69
LINK: Read char: 82
LINK: Read char: 65
LINK: Message received: AHEJ JONBCS BDROTHERA
LINK: Decoding...
LINK: Message after decoding: HEJ JONAS BROTHER
Message received: HEJ JONAS BROTHER
Press <RETURN> to close this window...
```

Figur 2 – Test af Recieve-metode i Linklag.



## Transportlaget

Det vigtige i transportlaget er at sørge for at dataen er kommet frem. Dette gøres ved en checksum som verificerer dataen, samt et sekvensnummer som sikrer at vi ikke godkender en forkert pakke, hvis en anden pakke er blevet sendt dobbelt. Når man har modtaget en data pakke skal man returnere med en acknowledge pakke som fortæller om dataen er nået korrekt frem og man er klar til næste pakke.

### Send-metode

De 4 første pladser udgør Transportlagets header. Der bliver brugt funktionen *memcpy* til at flytte dataen fra det ene array til det andet. I buffer-arrayet sættes plads 2 til at være lig med *seqNo*. Plads 3 sættes til at være lig med 0, da vi ønsker at sende data. Checksum funktionen udregner checksummen og sætter denne på plads 0 og 1.

Et *do-while-loop* oprettes og sender data så længe der ikke er modtaget en ACK fra modtageren.

```
void Transport::send(const char buf[], short size)
{
    // TO DO Your own code
    char array[size+4] = {0};
    memcpy(array+4, buf, size);

    memcpy(array+4, buf, size);

    buffer[SEQNO] = seqNo;
    buffer[TYPE] = DATA;    // Send data

    checksum->calcChecksum (array, size+2);    // Checksum of header

    do
    {
        link->send(array, size+4);
        std::cout << "TRANSPORT: Waiting for ACK\n";
    }
    while(!receiveAck()); // Send till we get an acknowledge

    old_seqNo = DEFAULT_SEQNO;
}
```

Snippet 5 – Send-metode i Transportlaget.

### Recieve-metode

Et *do-while-loop* oprettes. Pakkerne der er blevet sendt afsted sammenlignes med de pakker, der er blevet modtaget. Dette gøres med en sammenligning udført af checksum. Dette tjek gemmes i variablen *res*. Vi forlader først *do-while* loopet, når både checksummen er korrekt samt det gamle sekvensnummer og det nye ikke er det samme.

Vi bruger *memcpy* til at kopiere dataen. Vi skal ikke længere bruge Transportlags headeren på 4 bytes og der tages højde for dette.

```
short Transport::receive(char buf[], short size)
{
    // TO DO Your own code
    int counter, res;

    do
    {
        counter = link->receive(buffer, size);
        res = checksum->checkChecksum(buffer, counter-2);
        sendAck(res);
    }
    while(!res || !(old_seqNo == buffer[SEQNO]));

    memcpy(buf, buffer+4, counter-4);

    old_seqNo = buffer[SEQNO];

    return (counter - 4);
}
```

Snippet 6 – Recieve-metode i Transportlaget.

### Applikationslaget

Applikationslaget er gennemgået i øvelse 7/8 og der vil derfor ikke blive gået i detaljer her. Det er i dette lag at filhåndteringen hører til. Serveren står og venter på at der bliver taget kontakt til denne. Klienten tager kontakt til serveren med en anmodning om en fil. Serveren undersøger om filen eksisterer og håndterer at udlæse BUFSIZE antal bytes som sendes i pakker til transportlaget. Klienten sørger for at skrive pakkerne ned i en fil på den anden side.

### Konklusion

Der i denne øvelse arbejdet med rigtig meget af det kode og de ideer som ligger bag at det store internet virker. Vi har prøvet at implementere nogle af de metoder bag TCP som skal sikre kvaliteten og forstået hvor kompliceret det hele er. Der er i kommunikationsprotokoller mange ting der kan være med til at give brugerne en god oplevelse uden at de kender til det. Til forskellige protokoller er der forskellige krav idet der kan være fordele såvel som ulemper ved mange funktionaliteter. Der er udtænkt mange meget avancerede teknikker, hvor vi nu har arbejdet med nogle, og kun hørt og læst teoretisk om andre.