Syddansk Universitet

Semesterprojekt i mobile robotsystemer, (e18) - RB-PRO3

Civ. Robotteknologi 3. semester - Gruppe 2

Deadline for aflevering: 17/12-2018

SoundBox

Studerende:
Andreas Risskov Sørensen
asoer09@student.sdu.dk
15/03-1988
Eks.nr.: 456885

Studerende:
Dan Nykjær Jakobsen
dajak17@student.sdu.dk
30/04-1998
Eks.nr.: 455913

Studerende:
Joakim Lykke Stein
joste17@student.sdu.dk
14/08-1997
Eks.nr.: 456129

Studerende:
Lars Gylding Lindved
lalin17@student.sdu.dk
09/12-1987
Eks.nr.: 443299

Studerende:
Mirzet Traljesic
mtral17@student.sdu.dk
29/05-1996
Eks.nr.: 113419323

Studerende:
Troels Zink Kristensen
tkris17@student.sdu.dk
30/04-1997
Eks.nr.: 455427

Vejleder:
Ib Refer
refer@mci.sdu.dk

Abstract

The purpose of this project is to communicate between two computers through DTMF (Dual-Tone Multiple-Frequency) tones. The communication is realized with a program written in C++ using the IDE, Visual Studio. The project looks at transferring files from a client to a server and vice versa. This will be done as reliably as possible to make sure that all the data is transferred and received correctly.

The program is divided into three layers: the upper one being the application layer; the middle one is the datalink layer; and the lower one being the physical layer. Four buffers are located between these three layers, which assures the transfer from the lower to the upper layer and vice versa through the datalink layer.

The application layer consists of a client and a server. The interface of the client is used by users to upload, download and remove files from the server. It is furthermore possible to receive a list of all the files on the server. That is, the client requests something and the server replies.

It is the datalink layer's responsibility to split up files from the client or server and to encapsulate them into frames. These frames are sent to either client or server, where they are decapsulated and checked for errors. The frames are then unified and sent to the application layer, and an acknowledgement is sent back to the sender.

In the physical layer, the frames are converted to DTMF tones and transmitted through speakers. Another device records the transmission, where the DTMF tones are filtered and analyzed by means of FFT (Fast Fourier Transform). Afterwards, the signals are decoded.

The project concludes with a safe and reliable transfer of files between the three layers to another device through a simple user interface.

Indholdsfortegnelse

1 Forord

Rapporten er udarbejdet af Andreas Risskov Sørensen, Dan Nykjær Jakobsen, Joakim Lykke Stein, Lars Gylding Lindved, Mirzet Traljesic og Troels Zink Kristensen.

Projektet (RB-PRO3) tilhører uddannelsen, Civilingeniør i Robotteknologi, på 3. semester 2018 v. Syddansk Universitet i Odense. Projektet dækker 10 ECTS point, hvor de to kurser, RB-MSI3 og RB-COK3, anvendes som supplement til at udføre projektet.

Projektet blev udleveret d. 17. september 2018 og afsluttes med aflevering af rapport senest d. 17. december 2018. Der er givet en arbejdsperiode på 13 uger.

Vi vil gerne takke vores vejleder, Ib Refer, for at have vejledt os.

Læsevejledning
Der skal gøres opmærksom på, at kilder refereres i rapporten med tal-eksponenter placeret i firkantede parenteser, således: [2]. Disse referencer kan ses under sektionen: *Litteraturliste*. I visse sektioner er der benyttet generel viden fra lærebøger fra semestrets undervisning, hvilket er beskrevet i *Litteraturliste*.

I det elektroniske bilag er journaler, flowcharts, kode mm. tilgængelige. Oversigten over det elektroniske bilag ses under sektionen: *Oversigt over elektronisk bilag*.

2 Indledning

Projektet har en eksperimental tilgang, da det er op til den enkelte gruppe, hvordan projektet skal udformes og afvikles. Fagene matematik, digital signalbehandling, datakommunikation og computersystemer, henholdsvis indeholdt i kurserne RB-MSI3 og RB-COK3, skal som et krav inkorporeres i projektet.

Selvom det er op til den enkelte gruppe, hvordan projektet udformes, skal disse krav overholdes:

- Bærbare computere skal kommunikere med hinanden, eller evt. et embedded system, ved udveksling af lyd.
- Der skal anvendes DTMF-toner, og der skal designes en kommunikationsprotokol.
- Hertil skal der udvikles en distribueret applikation i C++.
- Der skal anvendes en lagdelt softwarearkitektur.
- Arkitekturen kunne være client/server med f.eks. tykke klienter.

Projektet omhandler i stor grad DTMF-toner, da disse toner anvendes som signalelementer til at overføre data fra afsender til modtager og vice versa.
DTMF-toner står for **D**ual **T**one **M**ultiple **F**requency og anvendes i telefoner med trykfunktion; de giver en karakteristisk "dut" -lyd ved hver knap. Tonerne anvendes til at transmittere det indtastede telefonnummer ind til telefoncentralen. Der er maksimalt 16 DTMF-toner, og hver enkelt knap har hver sin lyd. Denne lyd er en kombination af to frekvenser ud af otte i alt, som kan give op til 16 DTMF-toner.[1]
Figur 1 viser en oversigt over de 16 forskellige DTMF-toner, med deres tilhørende kombination af frekvenser.

	1209 Hz	1336 Hz	1477 Hz	1633 Hz
697 Hz	1	2	3	(A)
770 Hz	4	5	6	(B)
852 Hz	7	8	9	(C)
941 Hz	*	0	#	(D)

Figur 1: Oversigt over DTMF-tonerne.

3 Problemformulering

Dette projekt handler om at lave en mobil filserver, som kan tilgås af klienter vha. lyd. Både server- og klientenheder er computere. Idéen er baseret på Dropbox. En serverenhed skal indeholde filer med begrænset datastørrelse, som kan tilgås og hentes af klienter. Ydermere skal klienterne være i stand til at tilføje og fjerne filer. Kommunikationen mellem to enheder skal kunne foregå under forskellige støj- og distanceforhold. Af denne årsag lægges fokus på filtrering og pålidelig kommunikation i stedet for hurtig overførsel.

Problemformuleringen udformes herunder. Arbejdsområderne beskrives med de tilhørende problemstillinger.

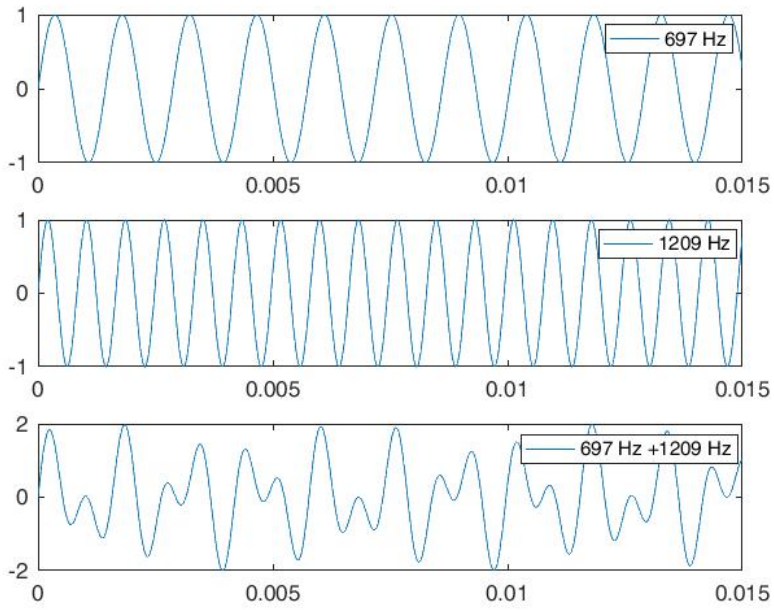
- Softwaren skal opbygges som lagdelt arkitektur, hvilket gør det nemmere at debugge og ændre funktionaliteter i forskellige lag uden at påvirke andre lag
  - Samtidig håndtering af processer i softwaren uden fejl i dataet.
  - Systemlås ved dataoverførsel mellem de forskellige lag.
- Klienten skal kunne tilgå indholdet på serveren vha. en simpel brugergrænseflade
  - Serveren og klienten skal være i stand til at læse og skrive til en fil.
  - Håndtering af forskellige filtyper i C++.
- Oprettelse af en dialog mellem server og klient
  - Identificering af forskellige enheder.
  - Opbygning af pakker og deres egenskaber.
  - Fejltjek af modtagne pakker.
  - Tab af pakker grundet overbelastning ved overførsel.
- Kommunikationen skal foregå vha. DTMF-toner
  - Håndtering af lyde i C++.
  - Højtalerens og mikrofonens håndtering af DTMF-frekvenserne.
  - Omgivelsernes påvirkning af signalet fra forskellige former for støj, f.eks. rumklang, hardwarestøj og generel støj, samt distance.
  - Identificering af modtagne signaler.

4 Lyd

I 1700-tallet fremlagde filosoffen George Berkeley spørgsmålet om, hvorvidt det larmer, når et træ falder i skoven, og der ikke er nogen til at høre det? Dette projekt omhandler ikke filosofi eller socialkonstruktionisme, men fra et videnskabeligt perspektiv kan man sige at lyd, som vi opfatter det, fremkommer af sinussvingninger, og at disse ikke larmer, så længe de ikke bliver hørt. Mennesker hører lyd ved, at trommehinderne opfanger svingningerne, der af hjernen bliver opfattet som lyd. Dette er ikke just tilfældet med en C++ kode på en computer, men her kan en mikrofon erstatte trommehinderne, og computerens regnekraft opfatte og analysere signalet fra mikrofonen.
Lyd er længdebølger, som opstår ved, at luftmolekylerne svinger langs med bølgens udbredelse. Disse bølger kan beskrives til tiden t vha. harmoniske sinussvingninger på formlen:[2]

$$f(t) = A \cdot \sin(2\pi \cdot f \cdot t)$$

hvor A angiver amplituden og f er frekvensen.
Hvis en sådan svingning består af blot én frekvens, kaldes den en ren tone. Rene toner kan vekselvirke gennem superpositionsprincippet og tilsammen danne sammensatte toner, hvilket ses på figur 2.



Figur 2: 697 Hz og 1209 Hz vekselvirker og danner DTMF-tonen "1".

Mennesker kan ud fra sammensatte toner f.eks. høre forskel på de forskellige DTMF-toner. Denne funktionalitet er ikke naturlig for en computer, men den kan gøre det via Fouriertransformation. Mere herom i afsnit 8.

5 Overordnet opbygning af programmet

I projektoplægget dikteres det, at softwaren skal programmeres i C++, og at opbygningen heraf skal være lagdelt. Det betyder, at hvert lag skal være en enhed i sig selv, og at de uden større problemer skal kunne tages fra og erstattes af nye. Det blev hurtigt besluttet, at en naturlig opdeling af programmet er, at have et applikationslag, et datalink-lag og et fysisk lag.

- Applikationslaget tager sig af brugergrænsefladen, filhåndtering og hvad der skal sendes og modtages. Dette lag er forskelligt hos klient og server, da de to applikationer har forskellige formål og funktioner.
- Datalink-laget står for at dele filer op i mindre pakker og forberede dem til at blive sendt. Laget står desuden for udpakning, fejltjek og flowkontrol, og sørger for, at der er en dialog mellem klient og server.
- Det fysiske lag sørger for at omdanne 4-bit størrelser til DTMF-toner og transmittere dem, samt i høj grad at opfange tonerne og ud fra frekvenserne aflæse den rette bitsekvens, på trods af støj fra omgivelserne.

Den eneste måde, hvorpå disse lag snakker sammen er gennem fire buffere placeret i en buffer-klasse. Denne klasse er oprettet som en singleton, hvilket vil sige, at der kun eksisterer én instans af klassen, og når buffere skal tilgås, kaldes en pointer til instansens adresse i hukommelsen.

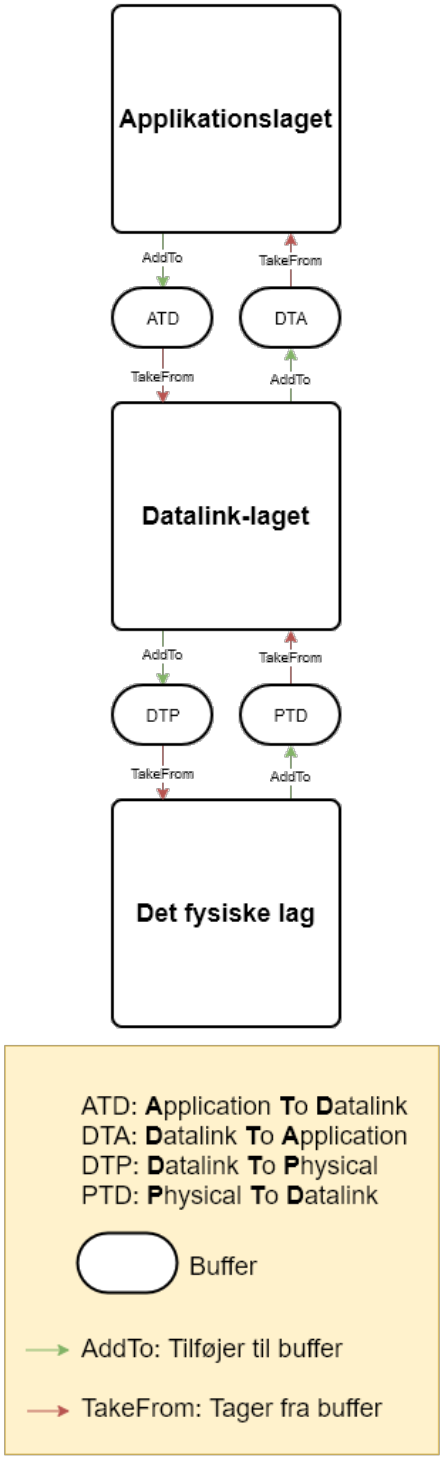
Mellem applikationen og datalink-laget ligger to buffere, som sender data mellem de to. Mellem datalink-laget og det fysiske lag ligger ligeledes to buffere, som på samme måde bruges til data, der skal sendes til det fysiske lag og modtagne data, der skal sendes til datalink-laget.

Hele programmet styres af forskellige tråde, som tager sig af forskellige dele af programmet, som skal køres samtidig.

I det elektroniske bilag under *Designklassediagrammer* ligger et samlet diagram over, hvordan klasserne i de tre forskellige lag hænger sammen, og hvordan disse tilgår bufferen, og derved hinanden.

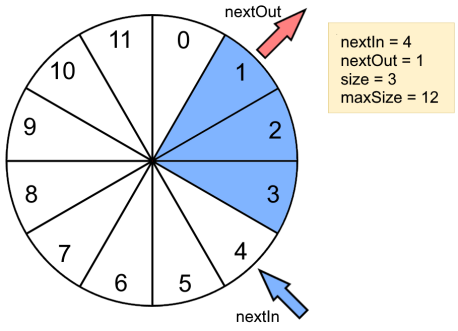
5.1 Buffere

Figur 3 viser et blokdiagram over de fire buffere og deres relation til de tre lag. Elementerne, der lægges i bufferne, skal anvendes i samme rækkefølge, de bliver lagt ind, dvs. First-In-First-Out. Benyttes en almindelig buffer til dette formål, skal alle elementer rykkes en plads ned, hver gang der tages ét ud, hvilket naturligvis koster ressourcer. Hvis det helt udelades at rykke elementerne, vil bufferens hukommelsesadresser på et tidspunkt ramme nogle adresser, som ikke er tilladte, og programmet vil give fejl. For at undgå dette anvendes i stedet ringbuffere bestående af et array, to pointere og to heltalsvariable. De to pointere peger på henholdsvis den næste plads, der skal tages et element fra, og den næste plads, der skal indsættes et element på, mens variablene henholdsvis angiver antallet af nuværende elementer i bufferen og den maksimale størrelse, som ligger fast for hver buffer. Figur 4 viser et generelt eksempel på tanken bag ringbuffere, og eksempler på, hvad de forskellige værdier betyder.



Figur 3: Blokdiagram over de fire buffere mellem lagene.

Hver gang der skal indsættes et element i en buffer, tjekkes det, om den pågældende buffer er fuld, altså om antallet af elementer, *size*, er lig størrelsen, *maxSize*. Hvis dette er tilfældet, ventes der indtil et element bliver udtaget. Er bufferen ikke fuld, placeres elementet på positionen angivet af pointeren *nextIn*, pointeren rykkes en plads videre og *size* inkrementeres.
Når der skal tages et element ud af bufferen, tages elementet på pladsen angivet af pointeren *nextOut* ud, *nextOut* inkrementeres og *size* dekrementeres. Eftersom *size* benyttes af begge funktioner, der styres af to forskellige tråde, er der anvendt en mutex lock for at undgå race condition. For kode-eksempel, se figur 5.



Figur 4: Eksempel på ringbuffer med tilsvarende værdier.

```
//AddToBuffer(newElement)
if (size == maxSize)
    throw "wait";

buffer[nextIn] = newElement;
nextIn = nextIn++ % maxSize;

mutex.lock();
    size++;
mutex.unlock();
```

```
//TakeFromBuffer()
if (size == 0)
    throw "wait";

element = buffer[nextOut];
nextOut = nextOut++ % maxSize;

mutex.lock();
    size--;
mutex.unlock();

return element;
```

Figur 5: De to buffer-funktioner som generel kode.

De to buffere mellem applikationslaget og datalink-laget består af arrays, der indeholder strenge af komplette filer eller kommandoer. Da der ikke er behov for at sende så mange elementer til denne buffer på én gang, er denne bufferstørrelse sat til 7.
Mellem datalink-laget og det fysiske lag består bufferen af bitsets med størrelser på 4. Her er der tale om meget mindre og flere værdier, så her er bufferstørrelsen sat til 20. Bufferstørrelsen betyder dog ikke så meget, da trådene tjekker om der er plads inden noget indsættes, og venter, hvis der ikke er plads.

5.2 Tråde

For at kunne få kommunikationen og programmet til at køre, styres forskellige dele af programmet af tråde. Det er nødvendigt at benytte flere tråde, da det tillader et program at håndtere flere funktioner på én gang. Dette er nødvendigt, da programmet kontinuerligt skal optage lyd og behandle det, stadig kunne opfange den næste tone og samtidig køre resten af programmet. Benyttes kun én tråd vil programmet køre langsommere og bruge mere hukommelse, da alt fra én proces skal gemmes, inden næste kan starte, og dermed vil man kun optage lyd i sektioner, inden der kan foretages andet.

Hver tråd har sine egne registre og stack i hukommelsen, men kan tilgå hele koden, hvis det er nødvendigt, og manipulere med det samme data. Det er derfor vigtigt at sørge for, at tråde ikke ændrer det samme data på samme tid, da det kan forårsage forkerte resultater for trådene. Dette undgås ved at indsætte mutex locks; når en tråd tilgår dataet, låses sektionen, og andre tråde kan ikke tilgå den, før den første tråd er færdig, og har låst sektionen op igen. Som det ses i figur 5 anvendes mutex lock i bufferne, når antallet af elementer skal tælles op eller ned, da denne værdi ændres, når en tråd sætter ind og en anden tråd tager ud.

I programmet benyttes f.eks. to tråde i datalink-laget, når der sendes beskeder afsted. Én tråd står for at sætte frames i bufferen til det fysiske lag, og derefter vente på acknowledgements. Den anden tråd står for at lytte efter beskeder, afkode dem og sætte ACK-flaget, hvis et sådant er modtaget, hvilket uddybes i afsnit 7.2.

Beskeder på tværs af tråde

Ved anvendelse af tråde, kan det være svært at håndtere kodesekvenser, som anvender ressourcer fra andre tråde. Derfor er der lavet et simpelt beskedsystem, som fungerer ligesom et statusregister. Dette er meget simple beskeder, men det tillader tråde at tjekke på de ønskede flag.
I dette projekt er der lavet følgende flag, der som standard er 0:

0	1	2	3	4	5	6	7	8	9
ST	SR	SEN	PAK	TXC	KOMF	DOS	MDK	FN2	FN1

Hvis sat sker følgende:

- ST: Stopper programmet.
- SR: Stopper med at optage.
- SEN: Angiver at der er stoppet med at modtage og må sende.
- PAK: Angiver at der er en pakke klar fra datalink-laget til det fysiske lag.
- TXC: Angiver at overførslen er færdig.
- KOMF: Angiver at der er sket en fejl ved overførsel.
- DOS: Angiver at download er påbegyndt.
- MDK: Angiver at der ikke er nogen DTMF-tone på mediet i mindst to vinduer.
- FN2:FN1 Bruges til at sætte filterniveauet.

FN2	FN1	Filterniveau
0	0	Ingen filtre
0	1	Første filterniveau
1	0	Andet filterniveau
1	1	Reserveret

6 Applikationslaget

Applikationslaget består af en klient og en server. Disse to dele af laget er opdelt for at skelne mellem "request" og "reply", henholdsvis forespørgsel og besvarelse. Klienten foretager en forespørgsel, og serveren besvarer denne forespørgsel. Klienten kan foretage ændringer på serveren; f.eks. i form af tilføjelse af en fil til serveren. Serveren er altså det sted, hvor alle filer føjes til, opbevares på og fjernes fra. Kommunikationen mellem klienten og serveren forekommer under datalink-laget. Dette lag anvendes som en port fra klient til server og fra server til klient.
Der er overordnet set fire forskellige metoder i filhåndteringen. Vha. disse metoder kan man oprette, hente og fjerne filer, samt tilgå listen af eksisterende filer på serveren.

Opret fil

Klienten ønsker at oprette en fil på serveren; dette er en forespørgsel, som fuldføres vha. *upload*-funktionen. En streng tildeles adresserne på server og klient, samt kommandoen 1 (oprettelse af fil) og derefter det data, som skal tilføjes til filen. Dette tilføjes til bufferen *appToDatalink*.
Serveren besvarer forespørgslen med funktionen *constructFile*, som opretter en ny fil på serveren, og har som paremetre: navnet på filen, adressen på filen, root-adressen hvorpå filen skal gemmes, filtypen og det data, som skal tilføjes til filen. Funktionen kaldes kun, når serveren modtager kommando 1.

Hent fil

Klienten ønsker at hente en fil fra serveren; dette er en forespørgsel, som fuldføres vha. *download*-funktionen. Kommando 2 anvendes for at hente en fil fra serveren, og når denne vælges, oprettes en fil via *constructFile*-funktionen hos klienten, når dataet modtages.
Serveren besvarer forespørgslen med funktionen *readListedFile*. Denne funktion læser den ønskede fil fra serveren, og sender en streng på samme vis, som i *upload*-funktionen, bortset fra, at kommandoen er 2. Dette tilføjes til bufferen *appToDatalink*.

Fjern fil

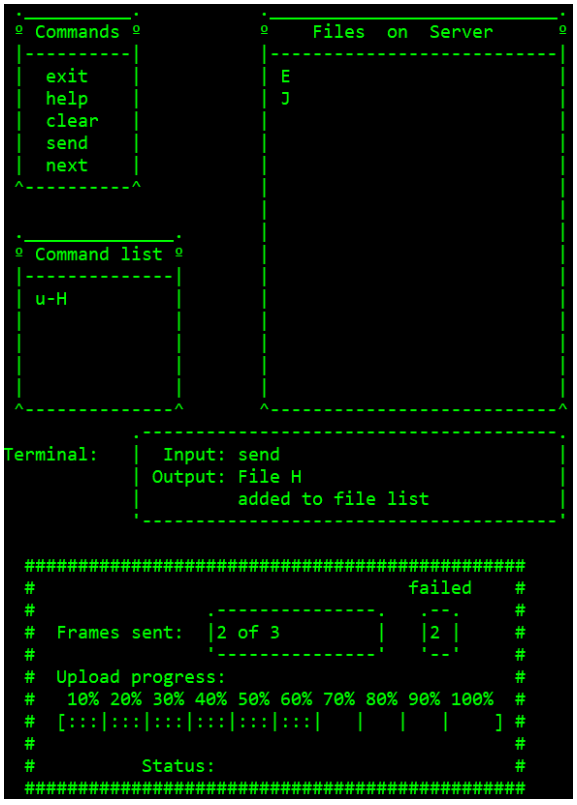
Hvis klienten i stedet ønsker at fjerne en fil fra serveren, realiseres dette med en forespørgsel med *upload*-funktionen. Klienten uploader en forespørgsel til serveren ved at sende en streng med adresserne på server og klient samt kommandoen 3, og derefter filens navn til bufferen, *appToDatalink*.
Herefter besvarer serveren forespørgslen med funktionen *removeFile*. Filen fjernes fra vektoren *files*, og den kan derfor ikke tilgås længere.

Liste af filer

Klienten kan også ønske at tilgå listen over alle filer på serveren. Der afsendes en forespørgsel med kommando 0 til bufferen *appToDatalink* vha. *upload*-funktionen.
Serveren foretager en besvarelse ved at sende kommando 0 samt listen over alle filer tilbage til klienten via *appToDatalink*-bufferen.
Herefter udtager klienten hver fil på listen, og katalogiserer dem på hver sin række på en vektor-streng vha. *download*-funktionen. Listen vises i Kommandoprompt hos klienten.

Brugergrænsefladen

Figur 6 og 7 viser den simple brugergrænseflade for klienten i Kommandoprompt. Figur 6 viser kommandoen *u-H*, som uploader filen *H* til serveren. I bunden er processen visualiseret med en procesbar. To frames er modtaget succesfuldt, dog har programmet i dette eksempel mislykket to gange under transmissionen af et frame. Output-feltet viser, at filen *H* er tilføjet til klienten; den er ikke tilføjet til serveren før transmissionen er 100 % fuldført. Figur 7 visualiserer en fuldført transmission og listen over alle filer på serveren.



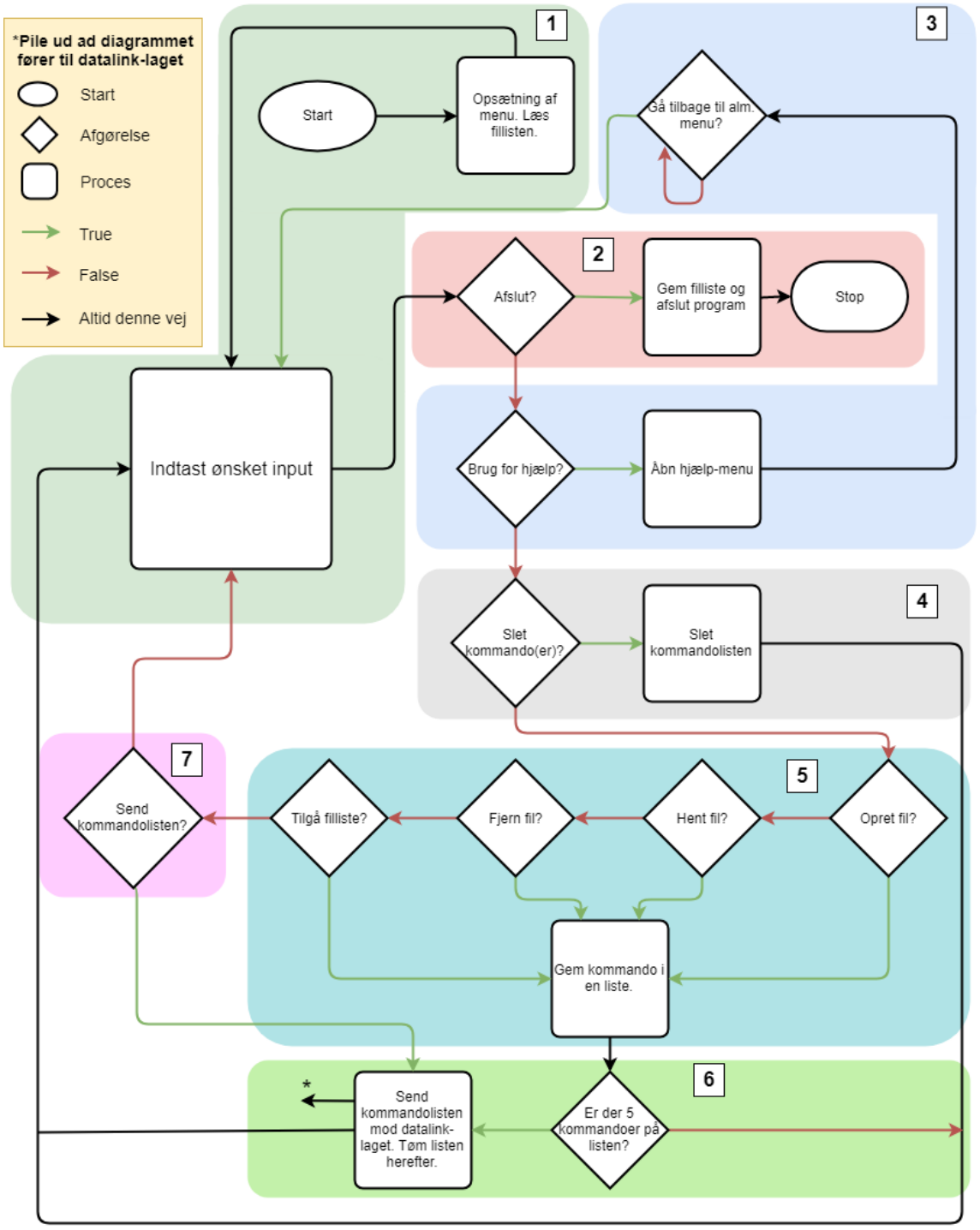
Figur 6: Kommando u-H, uploader filen H til serveren.

Figur 7: Brugergrænsefladen, hvor tre filer er uploadet til serveren.

I det elektroniske bilag (*SoundBox - Klientens brugergrænseflade*) kan en udvidet oversigt over brugergrænsefladen og sine funktioner ses.

6.1 Klient

Herunder ses flowchartet for klienten. Flowet er opdelt i syv blokke, hvor hver blok gennemgås på næste side. Flowchartet kan også ses under *Flowchart* i det elektroniske bilag.



Figur 8: Flowchart over klienten (forklaring næste side).

1. Når klienten opstartes, initialiseres start-feltet. Menuen opsættes i Kommandoprompt og fillisten indlæses, hvis den tidligere er blevet hentet. Herefter initialiseres markøren i inputfeltet; det felt, hvor kommandoer indtastes. Her vælger man mellem menuens otte forskellige kommandoer.

2. Hvis brugeren vælger at afslutte programmet, gemmes den eventuelt oprettede filliste, før programmet afsluttes. Herefter stoppes hele programmet, og man skal initialisere start-feltet igen.

3. Hvis brugeren i stedet ønsker at tilgå hjælp, åbnes en hjælp-menu. Denne menu forbliver, indtil man ønsker at gå tilbage til den tidligere menu igen.

4. Det er også muligt at slette de indtastede kommandoer, hvilke er vist under "Command List"-ruden, hvis indtastning fortrydes.

5. Brugeren kan også vælge at oprette, hente og fjerne en fil samt tilgå fillisten. Når kommandoen for fillisten eksekveres, hentes filkataloget fra serveren til klienten og oplistes i "Files on Server"-ruden. Disse kommandoer gemmes i kommandolisten.

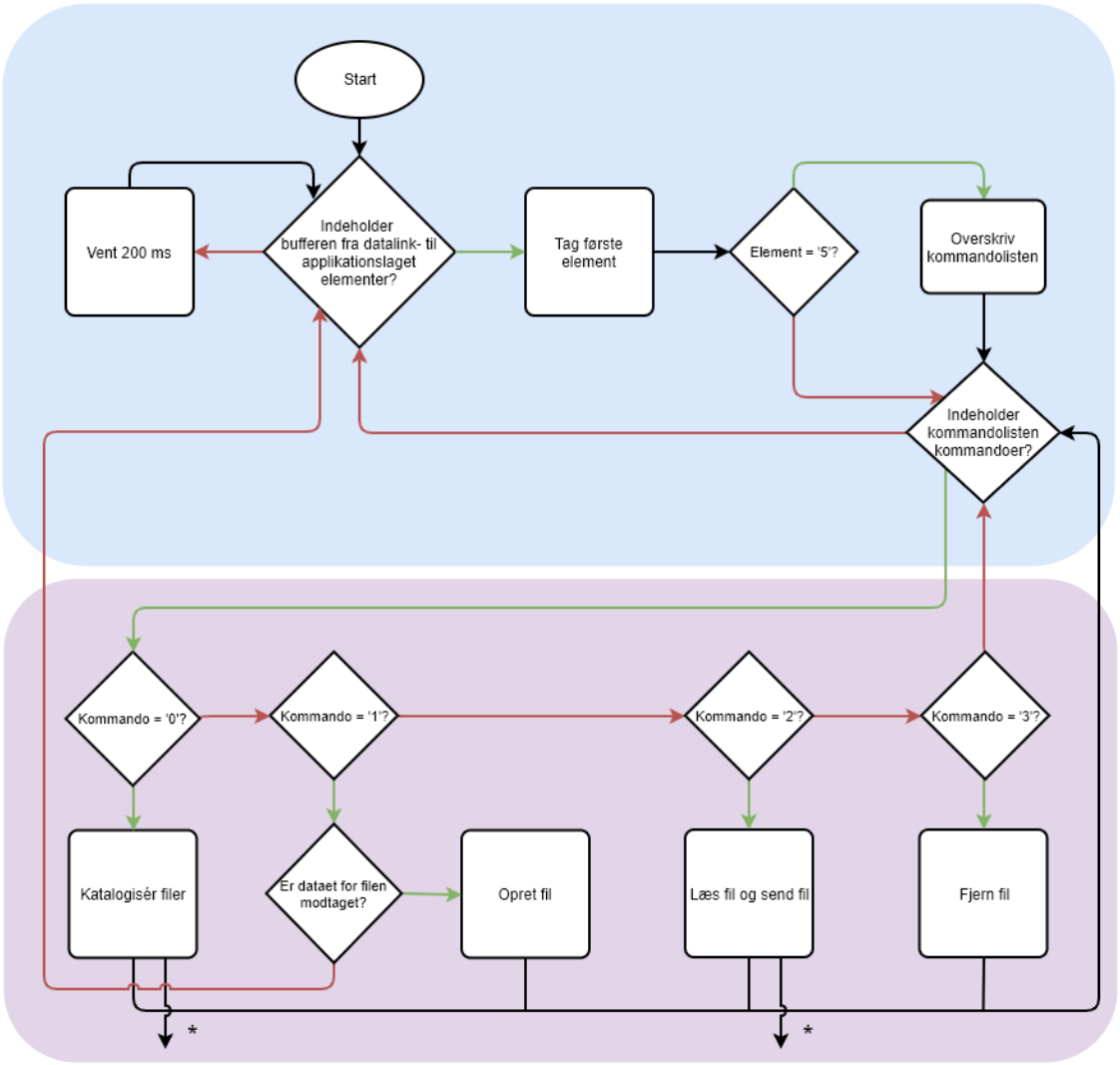
6. Hvis der på et tidspunkt er fem kommandoer på kommandolisten, sendes disse automatisk mod datalink-laget, hvorefter listen tømmes. Hvis der endnu ikke er fem på listen, føres programmet tilbage til "Indtast ønsket input".

7. Som en sidste mulighed kan man også sende kommandolisten, før der er indtastet fem kommandoer. Hvis dette er tilfældet, sendes disse mod datalink-laget. Listen tømmes herefter.

Både ved sletning af kommandolisten og afsendelse af kommandolisten føres programmet tilbage til "Indtast ønsket input".

6.2 Server

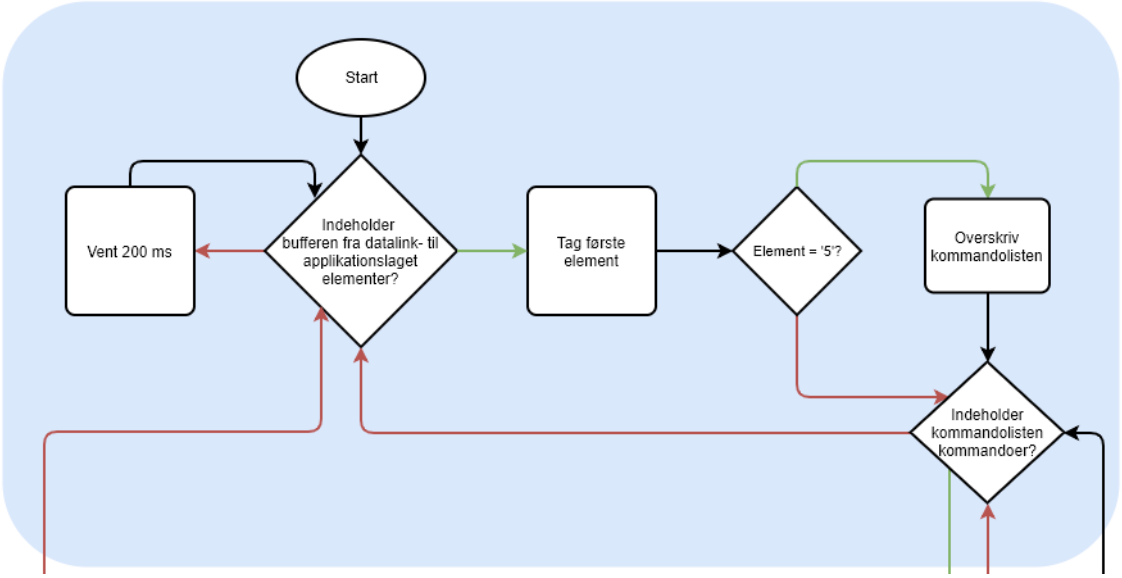
Herunder ses flowchartet for serveren. Flowchartet kan også ses under *Flowchart* i det elektroniske bilag.



Figur 9: Flowchart over serveren.

Der er to overordnede funktioner for serveren i applikationslaget: Tjek af bufferen og kommandolisten, samt håndtering af kommandoer, hvilket er illustreret på figur 9.

6.2.1 Tjek af bufferen samt kommandolisten



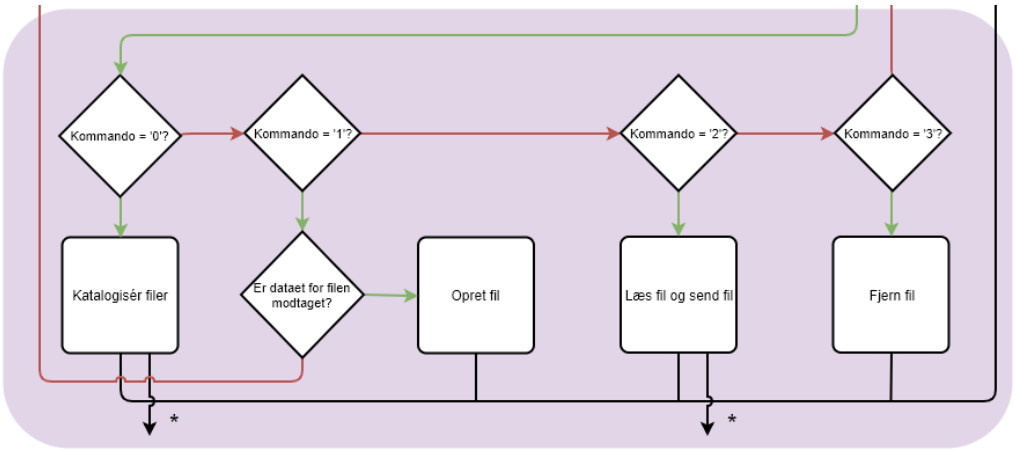
Figur 10: Blå sektion - Bufferen og kommandolisten.

Når serveren opstartes, initialiseres start-feltet. Herefter tjekker serveren om der er modtaget data i bufferen fra datalink-laget til applikationslaget. Hvis bufferen endnu ikke indeholder elementer, og derved er tom, venter serveren 200 millisekunder, før den tjekker igen.

Hvis bufferen derimod indeholder elementer, tjekker serveren første element på bufferen. Er position 8 for elementet lig med 5, overskrives kommandolisten med de kommandoer, som er tilsendt. Elementet er placeret på plads 8, da de første otte karakterer indeholder adressen på henholdsvis modtager og afsender. Herefter tjekkes der, om kommandolisten indeholder reelle kommandoer. Hvis dette ikke er tilfældet, føres programmet tilbage til bufferen igen. Indeholder listen reelle kommandoer, fortsættes til "lilla sektion".

Elementet kan også være forskelligt fra 5. Dette forekommer kun, når programmet har været mindst én gang igennem flowchartet. Situationen forekommer og forklares i "lilla sektion".

6.2.2 Kommandoer



Figur 11: Lilla sektion - Kommandoer.

Hvis kommandolisten indeholder kommandoer, føres programmet til denne sektion. Er kommandoen lig med enten 0, 1, 2 eller 3, forekommer ét af fire scenarier:

- Hvis karakteren er lig med 0, katalogiseres de eksisterende filer på serveren og afsendes mod datalink-laget og videre til klienten.
- Hvis karakteren er lig med 1, ønsker klienten at oprette en bestemt fil. Dog føres programmet tilbage til bufferen i "blå sektion", hvis dataet for filen endnu ikke er ankommet; kommandoen kommer før selve dataet. Herefter ankommes til situationen fra "blå sektion", hvor programmet returnerer false ved "Element = '5'?". Når dataet ankommer, er elementet ikke længere lig med 5, men indeholder derimod data, og derfor føres programmet videre til næste true/false-statement.
- Hvis karakteren er lig med 2, ønsker klienten at tilgå en fil, som allerede er oprettet på serveren. Denne fil læses først af serveren, og derefter sendes den videre mod datalink-laget og ud til klienten.
- Hvis karakteren er lig med 3, ønsker klienten at fjerne en fil fra serveren. Filen fjernes dermed fra fillisten og serveren.

Hvis første element på kommandolisten, hverken er lig med 0, 1, 2 eller 3, vender programmet tilbage til tjek af kommandoer på kommandolisten i "blå sektion". Programmet vil efter en fuldført kommando gå tilbage til "blå sektion" for at tjekke efter kommandoer på kommandolisten. Når første element er tjekket igennem, slettes denne, så næste element bliver første element på bufferen.

De fire overordnede funktioner, som er beskrevet under applikationslaget, kræver kommunikation mellem klienten og serveren, hvilket realiseres vha. datalink-laget.

7 Datalink-laget

Datalink-laget har to funktioner: At splitte filer fra applikationen og pakke dem ind i frames, så det sikres at en fejlfri transmission finder sted, samt at sørge for udpakning, fejlkontrol og flowkontrol gennem dialog mellem afsender og modtager.

7.1 Framing

Framingen er inspireret af HDLC-protokollen, som er bygget op omkring tre forskellige frame-typer; et I-frame til almindelige databeskeder, et dataløst S-frame til kontrolbeskeder og et U-frame til kontrolinformationer mellem enheder. Alle frames er opbygget således:

Flag	Adresser	Kontrol	Data	Fejltjek	Flag	Ekstra 0'ere
8 bit	8 bit	8 bit	n bit	8 bit	8 bit	0-7 bit

Tabel 1: Opbygning af frames.

S-frames har et tomt datafelt. Felterne gennemgås senere i dette afsnit. Frame-typerne og deres virkemåder bestemmes i kontrolfeltet, der er opbygget således:

	0	1	2	3	4	5	6	7
I-frame	0	N(S)			P/F	N(R)		
S-frame	1	0	Kode		P/F	N(R)		
U-frame	1	1	Kode		P/F	Kode		

Tabel 2: Kontrolfeltets opbygning.

N(S) angiver framenummeret, N(R) angiver acknowledgement-nummeret, som i I-frames kun benyttes i tilfælde af piggybacking, P/F angiver om der er tale om det sidste frame i en besked, og "Kode" angiver typekoden for det pågældende frame. I S-frames kunne det f.eks. betyde fejlfri modtagelse eller afvist frame, og i U-frames kunne det betyde forskellige opsætningsindstillinger.

Denne protokol blev valgt, da den er nem at anvende, og fordi den giver mulighed for forskellige frame-typer. I begyndelsen var tanken, at U-frames skulle benyttes til at oprette og afbryde forbindelsen mellem server og klient, og S-frames skulle desuden bruges til forskellige typer respons: F.eks. fejlfri modtagelse, fejl i adressering, fejl i transmission eller noget helt fjerde.
I sidste ende blev mange af disse funktioner dog ikke implementeret. Adressering blev overladt til applikationslaget, så brugen af U-frames blev overflødig. De to frametyper, der findes i den nuværende iteration, er I-frames til dataoverførsel og én type S-frame til acknowledgements for fejlfri modtagelse.

7.1.1 Indledende flow i framing

En tråd oprettes i initialiseringsklassen *KOMinit*, som tjekker hvert 20. ms, om der er lagt noget data i bufferen *appToDatalink*. Når bufferen indeholder data, kaldes klassen *InframeArchive* med dataet som parameter. Dataet består af en streng af ASCII-karakterer, hvoraf de første otte består af 0'er og 1'ere svarende til modtager- og afsenderadresser.

Først gemmes de otte karakterer for adresserne som en attribut i klassen, hvorefter strengen deles op i definerede datastørrelser på 16 bytes, som hver især skal pakkes ind i frames. Dette gøres ved at kalde *Inframe*-klassen for hver delstreng. Denne klasse benytter en række parametre: data, frametype, framenummer, slut-frame og adresser. Slut-frame er en værdi, der angiver, om det pågældende frame er det sidste frame i beskeden. Denne værdi er normalt '0', medmindre det er det sidste frame i den pågældende fil, hvor det i så fald er '1'. De færdige frames lægges i en vektor, som, sammen med de aflæste adresser, sendes til *Send*-klassen.

I *Inframe*-klassen gennemgås en række metoder, som tilføjer forskellige elementer til framet, der skal sikre en sikker transmission. Som eksempel gennemgås ASCII-strengen "OÃ", som kunne sendes fra klienten foruden de otte adressebit, som først omskrives til en streng af binær-værdier, hvorefter den kommer til at hedde:

"0100 1111 1100 0111"

7.1.2 Kontrolfelt

Som det første tilføjes kontrolfeltet, der er opbygget på to forskellige måder afhængig af, hvilken frametype, der er tale om:

0	1	2	3	4	5	6	7
Type	Reserveret			Slut	Framenr.		

Tabel 3: Kontrolfeltets opbygning.

I-frame (almindeligt data):

- **Bit 0:** Definerer typen, her "0".
- **Bit 1-3:** Reserveret.
- **Bit 4:** Angiver om framet er det sidste i den aktuelle fil.
- **Bit 5-7:** Angiver framenummeret.

S-frame (acknowledgement):

- **Bit 0:** Definerer typen, her "1".
- **Bit 1-3:** Reserveret.
- **Bit 4:** Angiver om det modtagne frame var det sidste i den aktuelle fil.
- **Bit 5-7:** Angiver framenummeret på det næste ønskede frame.

Eksemplet er et I-frame, og da beskeden er under 17 bytes, er framet det første og sidste i beskeden. Det kommer derfor til at hedde:

"00001000 0100111111000111"

- Bit 0 = "0"(I-frame)
- Bit 4 = "1"(slut-frame)
- Bit 1-3 = "000"(reserveret)
- Bit 5-7 = "000"(framenummer = 0)

7.1.3 Adresser

Det næste, der tilføjes er to 4-bit adresser, modtageradressen og afsenderadressen, som angives af applikationslaget i beskedens første otte karakterer. Serveren har en fast adresse, "0000". Det var tiltænkt, at serveren skulle generere en tilfældig adresse til klienten, når denne logger på serveren. Serveren skulle da modtage logon-requesten og enten acceptere den tilfældigt genererede adresse eller tildele klienten en ny. I stedet har klienten altid adressen "1111".
Hvis den tiltænkte adressering blev implementeret, kunne klientens tilfældige adresse eksempelvis hedde "0110", hvormed framet nu vil se således ud:

"00000110 00001000 0100111111000111"

7.1.4 Fejldetektering

For at detektere eventuelle fejl opstået under transmissionen, benyttes Cyclic Redundancy Check, CRC. CRC vælges, da det er en udbredt og effektiv metode til fejldetektering. Da framesene ikke er særligt store, og transmissionen er relativt langsom, vurderes det, at CRC-8 er passende, da det dermed kun tilføjer to ekstra toner til hvert frame.
CRC virker ved, at der tilføjes et kodeord for enden af dataet; ved CRC-8 er kodeordet 8 bit langt. Kodeordet bestemmes ved at lave en kopi af dataet, og tilføje en rest á otte 0'er for enden. En velkendt divisor på 9 bit (for CRC-8) tjekker først MSB i bitstrengen. Hvis MSB=1, XOR's bitstrengen med divisoren. Dernæst tjekkes den næste bit, og hvis denne også er lig 1, XOR's bitstrengen igen med divisoren, startende fra den pågældende bit. Dette gøres hele vejen gennem dataet, indtil alle bit deri er 0, og der i rest er opstået en kode, som tilføjes til enden af det oprindelige data. For eksempel, se figur 12.

Divisorer kan konstrueres ud fra en lang række kriterier for at opnå et tilfredsstillende fejltjek. I dette projekt er en standard divisor valgt, "100000111", for ikke at spilde tid på at konstruere én selv af formentlig ringere kvalitet.
Med det fundne kodeord indeholder framet nu:

"00000110 00001000 0100111111000111 11100110"

```
Frame:   00000110000010000100111111000111
Divisor: 100000111

00000110000010000100111111000111 00000000
    100000111
     100001010001001111111000111 00000000
     100000111
          11010010011111111000111 00000000
          100000111
           1010001111111111000111 00000000
           100000111
             100000011111111000111 00000000
             100000111
                  100111000111 00000000
                  100000111
                    111111111 00000000
                    100000111
                     11111000 00000000
                     10000011 1
                      1111011 10000000
                      1000001 11
                       111010 01000000
                       100000 111
                        11010 10100000
                        10000 0111
                         1010 11010000
                         1000 00111
                           10 11101000
                           10 0000111
                   Remainder:    11100110
```

Figur 12: Generering af CRC-8-kodeord for eksemplet.

7.1.5 Flag

Hvert frame starter og slutter med et flag på 8 bit. Der anvendes "01111110", da dette er et meget almindeligt mønster i mange protokoller, og det er ret simpelt at foretage bitstuffing. Med flagene ser framet således ud nu:

"01111110 00000110
00001000 0100111111000111 11100110 01111110"

7.1.6 Bitstuffing

Da der kan opstå en række af bit svarende til et flag inde midt i et frame, anvendes bitstuffing, så disse ikke fejlagtigt opfattes som flag. Hver gang der findes en bitsekvens på "011111", som ikke er flagene, tilføjes efterfølgende et ekstra 0, som illustreret:

"01111110 00000110 00001000 01001111 11000111 11100110 01111110"

⇓

"01111110 00000110 00001000 01001111 10100011 11010110 01111110"

Der vælges bitstuffing frem for bytestuffing for at minimere frame-størrelsen, men da strengen af 1'er og 0'er skal komprimeres til ASCII-karakterer, skal antallet af bit gå op i otte. Derfor tilføjes et antal 0'er bag det sidste flag, så antallet går op.

"01111110 00000110 00001000 01001111 10100011 11101001 10011111 10000000"

Hvis der blev benyttet bytestuffing, skulle der tilføjes otte bit, for hver gang et flag dukkede op, men her kan der bitstuffes op til otte gange, hvor kun ét byte tilføjes.

Når alt er tilføjet og framet er færdigt, tilføjes framet til en vektor i *InframeArchive*. Først skal det dog komprimeres, da hver bit er en karakter, og dermed fylder et helt byte, altså otte gange mere end nødvendigt. Derfor omskrives framet som nævnt til ASCII-karakterer

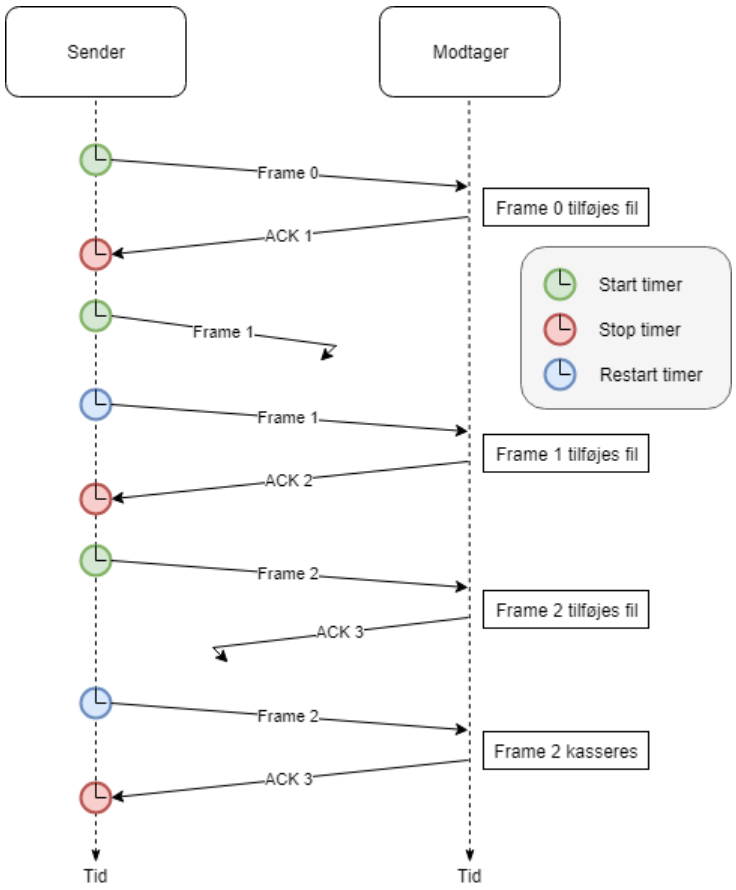
ved at tage otte elementer ad gangen, lave det om til et bitset med en størrelse på otte, omdanne bitsettet til en karakter og tilføje denne til en ny streng.

”01111110	00000110	00001000	01001111	10100011	11101001	10011111	10000000”
126	6	8	79	163	233	237	128
~	<ACK>	<bspc>	O	ú	Ú	*f*	Ç

Hukommelsesmæssigt ville det være ideelt kun at arbejde i bitset, men da bitsets skal have en konstant størrelse, når programmet kompileres, og da de derfor også er besværlige at manipulere med, blev det besluttet at arbejde med strenge af bit-værdier og komprimere dem til ASCII.

7.2 Dialog mellem enheder

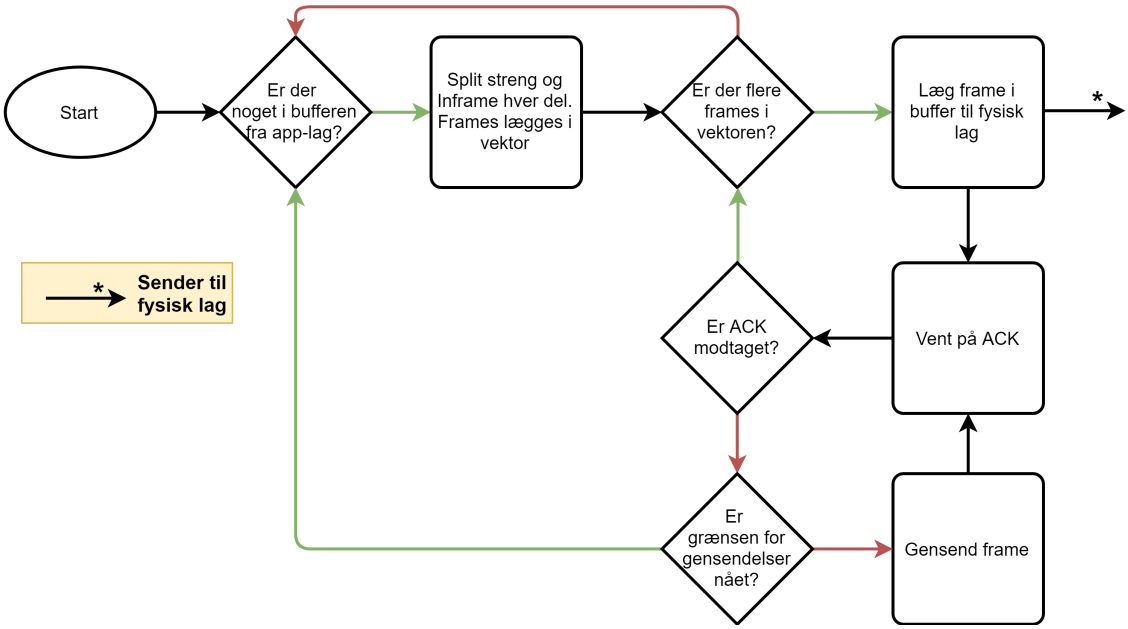
Selve dialogen mellem klienten og serveren er baseret på Stop-and-Wait-protokollen, som kan ses på figur 13, hvor afsenderen sender et frame afsted, og derefter venter på en kvittering, inden det næste frame afsendes. Denne protokol blev valgt, da den er simpel, og afsender og modtager ikke transmitterer samtidig, dvs. half-duplex-kommunikation. Fordi mediet er lyd, vil det være mere komplekst at anvende full-duplex-kommunikation, da transmissionerne vil interferere med hinanden og gøre kommunikationen mindre pålidelig.



Figur 13: Stop-and-Wait-protokol.

Der benyttes sekvensnummerering fra 0 til 7, men egentlig havde det været tilstrækkeligt at benytte en enkel bit til formålet. 3-bit-nummereringen blev valgt for at holde muligheden åben, for at kunne opgradere til en modificeret 'Go-Back-N'- eller 'Selective-Repeat'-protokol, men grundet den generelt langsomme sendetid og ustabile kommunikation, blev det besluttet at beholde Stop-and-Wait-metoden.

7.2.1 Senderside



Figur 14: Flowchart over afsendelse af frames.

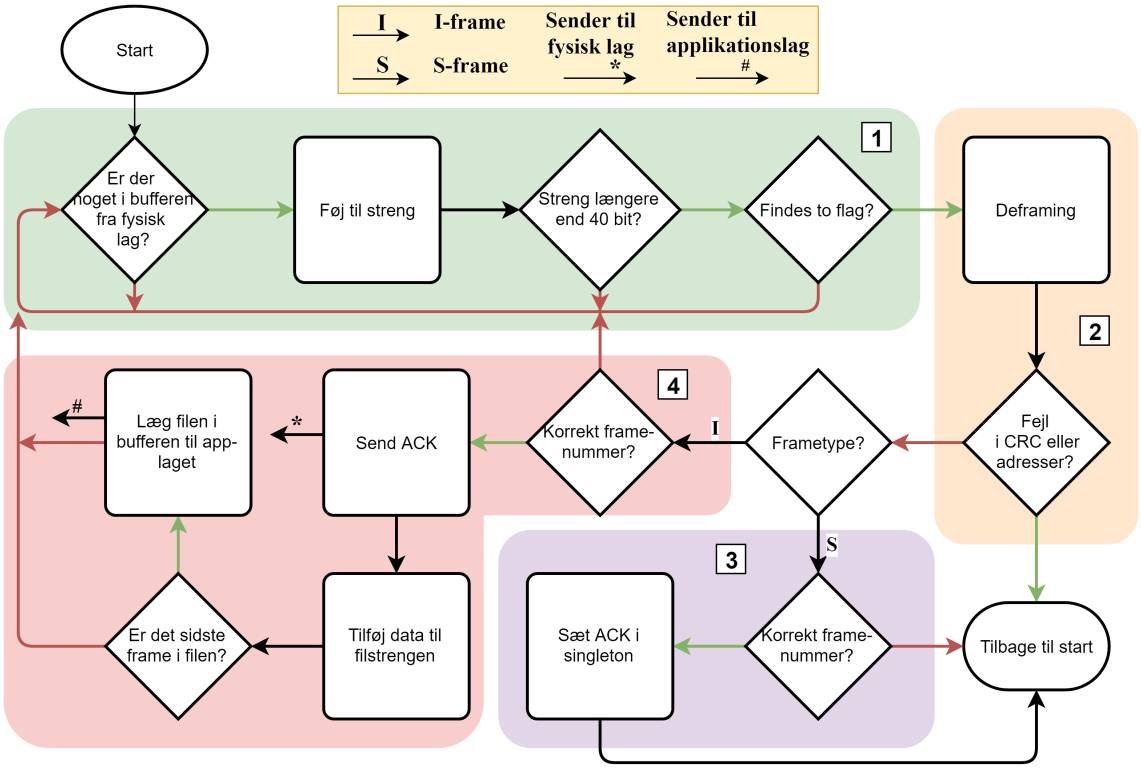
Når hele strengen af data har været igennem *Inframe*-processen og er tilføjet til en vektor i *InframeArchive*, kaldes metoden *sendFrames* i *Send*-klassen. Denne metode tager frame-vektoren og adresserne fra applikationslaget som parameter.
For at lægge det meste af ansvaret for adresse-håndtering over på applikationslaget, gemmes adresserne i singleton-klassen *Addresses*. Klassen oprettes som singleton, fordi adresserne, og nogle enkelte andre værdier placeret i klassen, skal tilgås, ændres og tjekkes på forskellige tidspunkter fra forskellige klasser. I den nuværende iteration er dette trin dog lidt overflødigt, da klientens adresse altid er "1111".

Derefter gennemgås frame-vektoren med en frame-counter, der starter ved 0. Frame nr. 0 sættes ind i bufferen *datalinkToSound* ved at oversætte strengen til bitsets á 4 bit, hvis der er plads. Hvis ikke, ventes der i 10 millisekunder, inden der prøves igen.
Når et helt frame er sat ind i bufferen, gemmes framecounteren i singleton-klassen *Addresses*, og ACK-flaget sættes til 0, hvilket betyder, at der nu ventes på et acknowledgement. Der oprettes en countdown-timer på 12, og hvert sekund tjekkes ACK-flaget.

Hvis der ikke er modtaget et acknowlegdement endnu, og ACK-flaget stadig er 0, dekrementeres timeren, og der tjekkes igen. Hvis timeren rammer 0, inkrementeres counteren *failedAttempts*, timeren nulstilles, og framet sendes igen. Hvis *failedAttempts* rammer et maksimum, sat til 7, afbrydes afsendelsen af beskeden.
Hvis ACK-flaget undervejs bliver sat til 1, nulstilles *failedAttempts*, framecounteren tælles

op, og det næste frame sendes til bufferen. Dette gentages, indtil alle frames er sendt, og et acknowledgement er modtaget, eller at transmissionen mislykkes. For flowchart, se figur 14.

7.2.2 Modtagerside



Figur 15: Flowchart over modtagelse af frames.

Flowet over modtagelsen af frames er opdelt i fire blokke, som beskrives herunder:

1. En tråd i *KOMinit*-klassen undersøger løbende modtagerbufferen *physicalToDatalink* for indhold. Hver gang der bliver fundet et nibble, sendes dette videre til *receiveBits*-metoden i *Receive*-klassen, hvor data lægges sammen. Når længden af modtaget data når en størrelse på 40 bit, undersøges det, om den indeholder to flag. Der ventes til denne størrelse, da frames ikke kan blive mindre end 40 bit, så der tjekkes ikke efter flag inden da. Hvis to flag findes, er et helt frame fundet, og dette sendes til *interpret*-metoden, som starter med at kalde *Deframe*-klassen med dataet.

2. Under deframing fjernes først de to 8-bit flag fra henholdsvis starten og slutningen af framet. Derefter bliver eventuelle ekstra nuller fra bitstuffing fundet ved at søge efter strengen "111110" og fjerne den efterfølgende bit. For at tjekke om der er sket fejl under transmissionen anvendes igen CRC-8 på samme måde som under inframing,

og hvis rest er nul, antages det, at framet er korrekt. Hvis ingen fejl bliver fundet, aflæses adresserne fra framet og sammenlignes med de gemte adresser i *Addresses*.

Når det er bekræftet, at pakken er ankommet til det rette sted, aflæses kontrolfeltet for frametype, framenummer og hvorvidt framet er det sidste i den pågældende fil. Hvis typen er et I-frame, er indholdet data, som skal konverteres fra bit til ASCII-karakterer. Dette gøres ved at tilføje strengen af bit til en stream, udtage en byte ad gangen ved hjælp af et bitset og konvertere det til en ASCII-karakter.

3. Informationerne fundet under deframing anvendes under *interpret*-metoden for at afgøre den videre håndtering. Hvis der blev modtaget et S-frame, er der tale om et acknowledgement hos den oprindelige afsender, og det undersøges, om det inkluderede framenummer er én højere end nummeret på det sidst afsendte frame. Er dette tilfældet, sættes ACK-flaget i singleton-klassen til 1, således at *sendFrames*-metoden kan fortsætte.

4. Er der derimod modtaget et I-frame med det forventede framenummer hos modtageren, tilføjes dataet til en streng, som senere skal blive til den endelige fil. Filen samles, når der modtages et frame, hvor slut-frame-bitten er sat. Der kvitteres for hver pakke ved at sende et acknowledgement tilbage til afsenderen. Dette sker ved at kalde *Inframe*-klassen med en tom streng, samt informationerne fra det modtagne frame, hvor framenummeret er inkrementeret tilsvarende nummeret på det næste frame, der ønskes. Dette acknowledgement-frame opdeles i nibbles, som lægges i bufferen til det fysiske lag.

8 Det fysiske lag

Det fysiske lags formål er at oversætte en bitstrøm til lyd, samt analysere og afkode lydsignaler, for at muliggøre kommunikation mellem to enheder via lyd. Det er meget oplagt at tildele DTMF-tonerne værdier fra 0 til 15, svarende til en nibble pr. tone.

8.1 Implementering af kommunikation i C++

DTMF-tonerne skal afspilles som sinuskurver gennem C++, hvilket som udgangspunkt ikke er muligt, da standardbibliotekerne giver begrænset adgang til PC'ens lydkort, og kun kan afspille rene toner.

8.1.1 Lydbibliotek

Dette kan dog løses ved at tilføje et lydbibliotek bestående af DLL-filer (Dynamic Link Library), som udvider C++'s funktionalitet. Disse DLL-filer kan give C++ større adgang til PC'ens lydkort, og tillader at anvende mere komplicerede lyde, som f.eks. sammensatte toner.

Der blev betragtet to potentielle udbydere af lydbiblioteker: SFML & SDL. Begge biblioteker kan afspille og optage sammensatte toner. SFML kan f.eks. gøre det i "real-tid" i bidder, som man selv bestemmer længden af. Altså kan der optages lyd i f.eks. 10 ms og løbende sende det videre til en anden del af programmet. Det kan SDL muligvis også, men grundet uoverskuelig dokumentation er denne funktionalitet aldrig blevet fundet. SFML har sin egen meget omfattende og logisk opbyggede dokumentation med mange eksempler, som er inddelt efter overordnede kategorier ift. grafik og lyd. Dette savnes ved SDL, som kun tilbyder en Wiki, hvor det ikke er ret nemt at finde rundt. Begge biblioteker ville formentlig kunne bruges i dette projekt, men SFML vælges pga. dets overskuelighed.

For at benytte SFML's Audio-bibliotek skal DLL-filerne tilføjes i C++; for nærmere info herom, se evt. *SFML guide* i det elektroniske bilag.

Med kommunikation menes afspilning og optagelse. Til at komme i mål med dette, blev der brugt en trinvis fremgang, som beskrives herunder.

8.1.2 Generering og afspilning af sammensatte toner i C++

I første omgang blev sammensatte toner genereret og afspillet i C++ på én computer og optaget i MATLAB på en anden. Der blev optaget i MATLAB, da det var ligetil at verificere signalet via den indbyggede *fft*-funktion (Fast Fourier Transformation), som på daværende tidspunkt endnu ikke var skrevet i C++.

Til at generere og afspille tonerne blev klassen *PlayDTMF* skrevet. Figur 16 viser humlen af klassens virkemåde.

```
for (int i = 0; i<sampleRate; i++)
{
    signal[i] = amp * ((sin(f1 * two_Pi) + sin(f2 * two_Pi)));
    f1 += f1inc; f2 += f2inc;
}
```

Figur 16: For-loop til generering af signal.

hvor *sampleRate* er samplingsfrekvensen og *amp* er amplituden, og begge er konstante værdier. *f1inc* og *f2inc* er konstante i forhold til hver enkelt DTMF-frekvens, med f_L værende den rene tone med lav frekvens og f_H den rene tone med høj frekvens. De bliver udregnet som:

$$f1inc = \frac{f_L}{samplerate} \qquad f2inc = \frac{f_H}{samplerate}$$

f1 og *f2* er variable værdier, som starter på 0 og stiger med *f1inc* eller *f2inc*.
signal[i] bliver et array med længden af *sampleRate*, og indeholder superpositionen af de to rene toner.
Så i forhold til superpositionen af to harmoniske svingninger:

$$y(t) = A\left(\sin\left(\omega_1 \cdot t\right) + \sin\left(\omega_2 \cdot t\right)\right) = A\left(\sin\left(2\pi \cdot f_1 \cdot t\right) + \sin\left(2\pi \cdot f_2 \cdot t\right)\right)$$

Der omskrives for de rene toner f_L og f_H:

$$signal(t) = amp \cdot \left(\sin\left(2\pi \cdot f_L \cdot t\right) + \sin\left(2\pi \cdot f_H \cdot t\right)\right)$$

Der normaliseres i forhold til samplingfrekvensen, og tiden t erstattes med element nummer n:

$$signal(n) = amp \cdot \left(\sin\left(2\pi \cdot \frac{f_L}{sampleRate} \cdot n\right) + \sin\left(2\pi \cdot \frac{f_H}{sampleRate} \cdot n\right)\right)$$

der vha. summer og *f1inc* og *f2inc* kan omskrives til:

$$signal(i) = amp \cdot \left(\sin\left(2\pi \cdot \sum_{n=0}^{i}(f1inc)\right) + \sin\left(2\pi \cdot \sum_{n=0}^{i}(f2inc)\right)\right)$$

hvor summerne netop kan udtrykkes som henholdsvis *f1(i)* og *f2(i)*:

$$signal(i) = amp \cdot \left(\sin\left(2\pi \cdot f1(i)\right) + \sin\left(2\pi \cdot f2(i)\right)\right)$$

hvilket repræsenterer skrivemåden, der bruges i C++.

8.1.3 Optagelse i C++

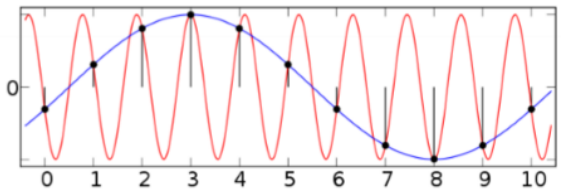
Næste trin var at optage i C++ og gemme dette som en .txt-fil, der kunne kontrolleres i MATLAB.
Til dette bruges klassen *StreamRecorder*, som arver fra SFML's indbyggede klasse *SoundRecorder*. *StreamRecorder* optager i bidder af en tidslængde, som defineres i den indbyggede SFML-metode *onStart*. I SFML-metoden *onProcessSamples* defineres, hvad der skal gøres løbende med hver bid af data optaget i forhold til tiden angivet i *onStart*. I denne tidlige iteration samler metoden *onProcessSamples* en vektor, som efter endt optagelse skrives til en .txt-fil, der analyseres i MATLAB.

8.1.4 Samtidig afspilning og optagelse i C++ gennem to tråde

I dette deltrin blev der implementeret en funktionalitet, hvor der på samme PC i én tråd løbende blev optaget til en vektor i klassen *StreamRecorder*, mens der i en anden tråd blev afspillet DTMF-toner vha. *PlayDTMF*-klassen. Efter endt optagelse og afspilning blev vektoren skrevet til en .txt-fil, der kunne kontrolleres i MATLAB.
I det endelige projekt skriver *StreamRecorder* ikke til en vektor, som gemmes, men derimod til en buffer *DTMF_SlicesBufferRecieve* i *Buffer*-klassen.

8.2 Domæneskift og frekvensdetektering

Når lydsignalet er optaget skal det analyseres, hvor det kan være fordelagtigt at kigge på signalet i frekvensdomænet. Da DTMF-toner består af to rene toner, giver det mening at lave frekvensanalyse af signalet og finde frem til, hvilke rene toner, der findes i signalet.
Når et signal konverteres fra analogt til digitalt, skal man være opmærksom på, at der kan være flere forskellige frekvenser, som kan danne den samme kurve. Se figur 17.



Figur 17: To forskellige frekvenser, som passer til de samme digitale punkter.

Derfor skal man, ifølge Nyquist og Shannon, kun kigge på frekvenser, som er under den halve samplingfrekvens. Når man har et signal, der kan genkendes som en trigonometrisk funktion, kan frekvenserne let aflæses. Hvis ikke det er muligt, kan man anvende Fourier-transformation.

8.2.1 Fourier-transformation

Alle signaler, selv firkantsignaler, kan udtrykkes som serier af trigonometriske funktioner. En tydelig udfordring ved Fourier er, at sinus og cosinus er periodiske, hvilket signalet ikke nødvendigvis er. Dette er ikke noget problem, da man altid kan udvide signalet så det bliver periodisk. Fourier-approksimationen $f_f(t)$ kan udtrykkes på følgende måde, hvor $f(t)$ er den funktion, som ønskes transformeret:

$$f_f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty}(a_n \cos(n\omega t) + b_n \sin(n\omega t))$$

hvor

$$a_n = \frac{2}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)\cos(n\omega t)\,dt \quad , \quad b_n = \frac{2}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)\sin(n\omega t)\,dt$$

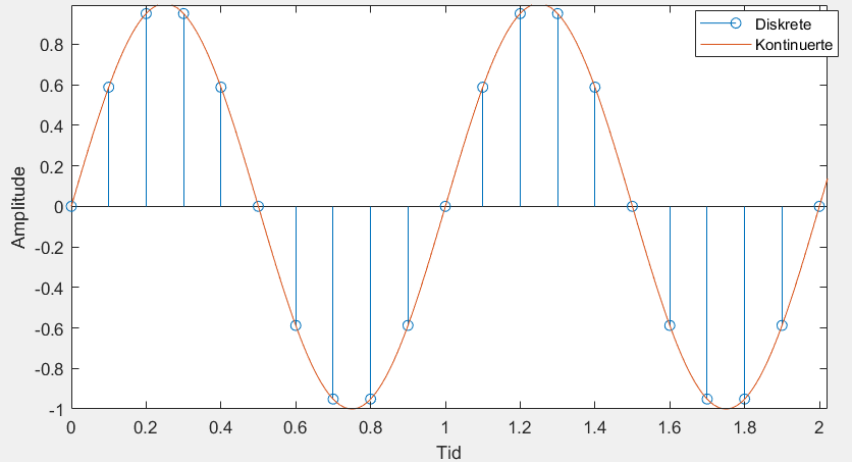
Eftersom sinus og cosinus kan udtrykkes på Eulers form, kan Fourier-transformation også skrives på følgende måde:

$$f_f(t) = \sum_{m=-\infty}^{\infty}(c_m e^{(jm\frac{2\pi}{T}t)})$$

hvor det gælder at $j^2 = -1$ og

$$c_m = \frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)e^{(-jm\frac{2\pi}{T}t)}\,dt$$

Man skal være opmærksom på, at der skelnes mellem to forskellige domæner, når man snakker Fourier. Enten tages der udgangspunkt i det kontinuerte tidsdomæne eller det diskrete tidsdomæne. Det kontinuerte tidsdomæne kan kun findes i den analoge verden, hvor alle punkter i signalet er sammenhængende. Det diskrete tidsdomæne findes i digitale systemer, hvor der er en afstand mellem hvert punkt, hvilket betyder, at man også snakker om opløsning i signalet. Jo større samplingfrekvens, desto bedre opløsning.



Figur 18: Forskellen mellem det kontinuerte og det diskrete tidsdomæne.

Når Fourier-transformation skal bruges til frekvensanalyse, bruges kun c_m. Derudover vil Fourier-transformation blive brugt i forhold til det diskrete tidsdomæne, hvor c_m beskrives som:

$$c_m = \frac{1}{N} \sum_{n=0}^{N-1} f[n]\, e^{-\frac{2jm\pi n}{N}}$$

hvor N er antal samples.
Eftersom c_m er et komplekst tal, angiver den både amplitude og faseforskydning i forhold til frekvensen.
Flere samples giver en bedre opløsning ved brug af Fourier. Hvis man f.eks. har to samples med en samplingfrekvens på 10 Hz, så har c_m én værdi ved henholdsvis 0 Hz og 10 Hz. Hvis man derimod har fire samples, er der værdier ved frekvenserne: 0 Hz, 3,33 Hz 6,67 Hz og 10 Hz. Man kan beskrive opløsningen som:

$$\frac{f_s \cdot i}{N-1} \quad , \quad 0 \leq i \leq N$$

hvor N er antal samples og f_s er samplingfrekvensen.
Når der vælges, hvor mange samples, som skal analyseres ad gangen, skal man være opmærksom på, hvor stort et spring der er mellem de forskellige søgte frekvenser.
Hvis man bruger formlen for c_m direkte, skal der foretages N^2 operationer, hvilket godt kan tage lang tid, især hvis man skal bearbejde rigtig mange samples. Der findes dog en algoritme, som kan bruges i stedet for, kaldet "Fast Fourier-transformation" (FFT).

Når FFT bruges skal antallet af samples være en potens af 2, men derimod skal der kun foretages $\frac{N}{2} \cdot \log_2(N)$ operationer. Det er dog ikke altid, at antallet af samples går op. Løsningen herpå er at tilføje 0'er indtil antallet går op, hvilket kaldes "zero-padding". Dette tilføjer ingen ekstra informationer til signalet, men det forbedrer opløsningen.
Fast Fourier-transformation kan enten laves med decimering i frekvens eller decimering i tid. Hvis man bruger decimering i frekvens halveres antallet af udregninger. Formlerne er som følger:

$$X(2m) = \sum_{n=0}^{N/2-1} a(n) \cdot W_{N/2}^{mn}$$

$$X(2m+1) = \sum_{n=0}^{N/2-1} b(n) \cdot W_N^n \cdot W_{N/2}^{mn}$$

hvor

$$a(n) = x(n) + x(n + \tfrac{N}{2}) \quad , \quad 0 \leq n \leq \left(\tfrac{N}{2} - 1\right)$$

$$b(n) = x(n) - x(n + \tfrac{N}{2}) \quad , \quad 0 \leq n \leq \left(\tfrac{N}{2} - 1\right)$$

$$W_N = e^{-j2\pi/N} \quad , \quad j^2 = -1$$

Decimering i tid udregnes på følgende måde:

$$X(k) = G(k) + W_N^k H(k)$$

$$X\left(\tfrac{N}{2} + k\right) = G(k) - W_N^k H(k)$$

hvor

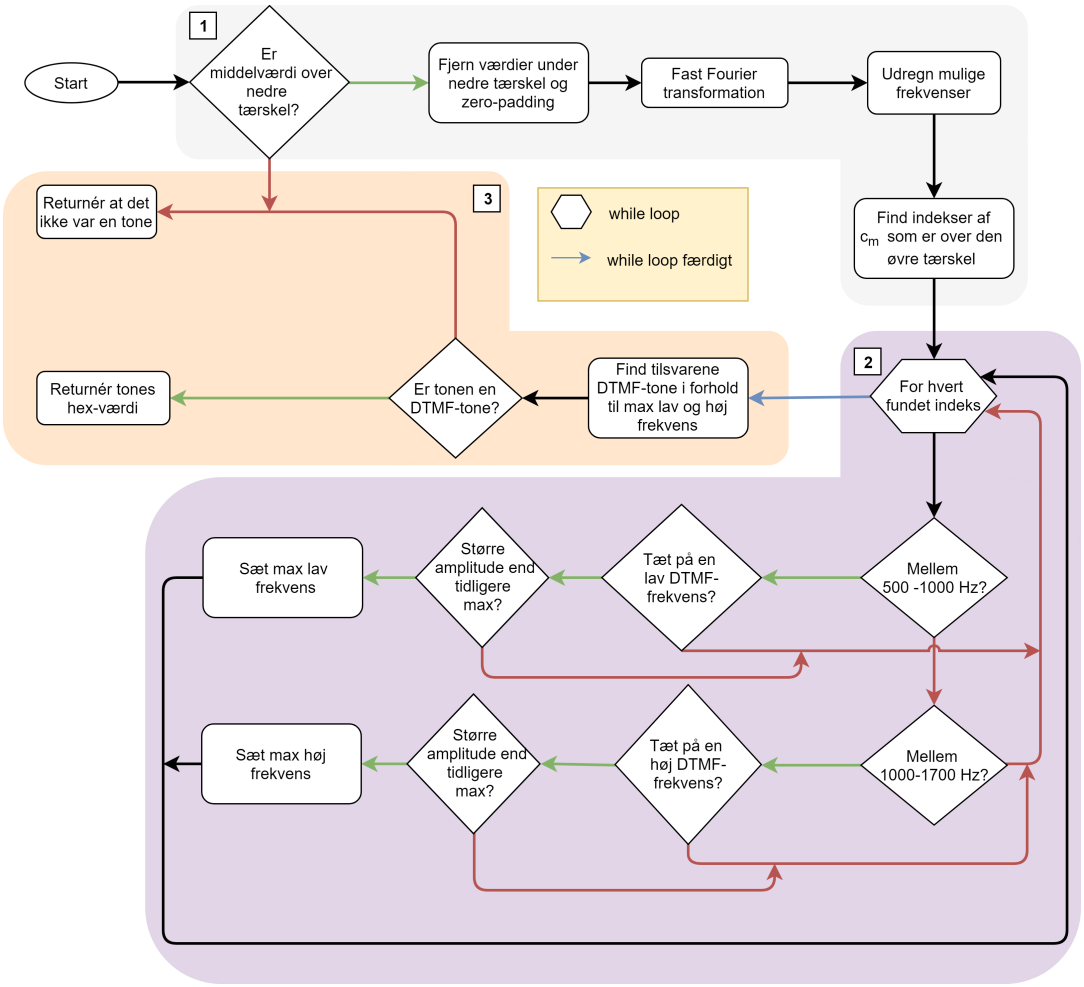
$$G(k) = \sum_{m=0}^{N/2-1} \left(x(2m) W_{N/2}^{mk}\right)$$

$$H(k) = \sum_{m=0}^{N/2-1} \left(x(2m+1) W_{N/2}^{mk}\right)$$

Til projektet blev der først implementeret metoden med decimering i frekvens. Denne metode blev i C++ testet på et signal med 40.000 samples, og efter 13 minutter var den kun halvvejs igennem udregningerne, hvorefter den blev stoppet. Eftersom MATLABs *fft*-funktion tog under 1 minut på samme signal, var det klart, at der var en fejl i implementeringen. Derfor blev der arbejdet videre på *fft*-metoden til projektet. Efter noget tid uden synlige fremskridt blev der fundet en *fft*-algoritme som pseudokode.[3] Denne algoritme brugte decimering i tid, og derfor blev den implementeret i stedet. Ved denne metode bliver Fast Fourier-transformation af 40.000 samples beregnet på 3,8 sekunder.

8.3 Frekvensanalyse-algoritme

Herunder ses det samlede flowchart for algoritmen til at detektere DTMF-toner i frekvensdomænet, hvor konverteringen sker ved FFT.



Figur 19: Flowchart over frekvensanalysen.

Boks 1: Frekvensanalyse

Når en analyse af signalet begynder, undersøges det først, om middelværdien af det absolutte signal er over en bestemt nedre amplitude-tærskel i tidsdomænet. Er dette ikke tilfældet, vil analysen stoppe og returnere 16, for at angive, at det ikke er en DTMF-tone. Dette sker for at mindske brugen af computerkraft og igen forhindre at bruge analysetid på et signal, som ikke indeholder nogen DTMF-tone. Er middelværdien større end den nedre tærskel, sættes værdierne herunder til nul.

Eftersom det skal bruges til FFT, er der prøvet med forskellige potenser af 2 for at finde den bedste afstand mellem hver Fourier konstant c_m. Det er valgt at lave "zero-padding" op til 1024, hvilket giver en brugbar opløsning med en trinstørrelse på 7,82 Hz.

Efter "zero-padding" bliver der foretaget FFT på signalet.
Efterfølgende bliver de forskellige frekvenser, som de forskellige c_m tilhører, udregnet. Derefter bliver amplituden på c_m tjekket igennem, og der findes indeksene af de c_m, som er over den øvre amplitudetærskel i frekvensdomænet. Grunden til at der anvendes en øvre amplitude-tærskel er, at der i en lydoptagelse kan findes næsten alle frekvenser, dog med meget lave amplitudeværdier. Når der kommer en DTMF-tone, vil de to frekvenser, som den består af, have større amplituder. Derfor kan der frasorteres frekvenser, som ikke er kraftige nok. Dette vil også reducere antallet af frekvenser, der skal analyseres på i den næste del herunder.

Boks 2: Udvælgelse af frekvenser

De fundne frekvenser vurderes i forhold til, at en DTMF-tone består af en lav og høj frekvens, henholdsvis under og over 1000 Hz. Der kan derfor deles op i lave og høje frekvenser, ved at udvælge den mest fremtrædende DTMF-frekvens i hvert område. For at tjekke de forskellige frekvens-indekser hurtigere igennem, bliver tjekket delt op i to dele.
I den ene del kigger man efter de mest fremtrædende lave frekvenser i området mellem 500 - 1000 Hz. I dette område undersøges kun frekvenserne inden for en rimelig afstand af en af de fire lave DTMF-frekvenser. Her kontrolleres det, om amplituden ved den nuværende frekvens er større end den tidligere godkendte. Hvis det er tilfældet, bliver denne amplitude gemt, og den mest dominerende frekvens bliver sat til den tilsvarende DTMF-frekvens. I den anden del gøres det samme, men nu for de høje frekvenser i området mellem 1000 - 1700 Hz. Derefter bliver den mest dominerende lave og høje frekvens sendt videre i systemet.

Boks 3: Parring af DTMF-frekvenser til DTMF-tone

Når den lave og høje frekvens er fundet i signalet og givet videre, bliver disse frekvenser sammenlignet med de forskellige DMTF-toner. For at angive, hvilken tone det er, får DTMF-tonerne en talværdi svarende til deres hex-værdi. Derudover bliver der tilføjet en ekstra værdi, som angiver, hvis der ikke er tale om en mulig tone. Denne værdi er sat til 16, grundet at DTMF-tonerne er tildelt værdierne fra 0 til 15.

8.3.1 Mulig optimering af analysen

Undervejs blev der opdaget flere områder, hvor man kunne optimere og spare tid, så analyseren går hurtigere. Disse områder bliver gemt til forbedringer i eventuelle fremtidige iterationer.
Det første område som kunne optimeres er, at anvende en funktion, som hedder *linspace*. Denne har samme funktionalitet som sin søsterfunktion i MATLAB; at generere en linje

med værdier mellem to endepunkter med en bestemt opløsning. Denne funktion tager omkring 4 ms, men cirka 90 % af disse værdier bliver ikke brugt på noget tidspunkt. Hvis man bestemmer opløsningen, kan man bruge de fundne indekser for godkendte signaler til at finde den tilsvarende frekvens da:

$$frekvens = opløsning \cdot indeks$$

Det andet område er ved udregning af Fast Fourier Transformation. Her bliver den delt op i lige og ulige indekser, hvorpå FFT igen anvendes, som også deles op i lige og ulige osv. Men de to dele afhænger ikke af hinanden, og derfor er det muligt at køre dem på hver sin tråd, så de kører parallelt. Hvis en FFT tager 100 ms vil den parallelle udførelse teoretisk kun tage 50 ms, da man antager, at det tager 50 ms for både de lige og de ulige dele.

8.4 Afkodning af signal

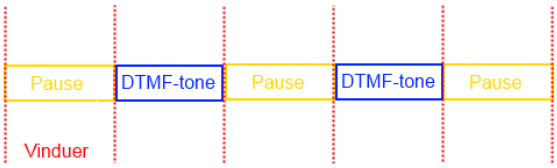
I det følgende afsnit antages det, at computeren kun kan udføre én ting ad gangen, altså at kun én tråd kører; det er lettere at forklare. Til sidst i afsnittet beskrives processen med flere tråde.
Når man laver frekvensanalyse på et signal, som indeholder flere DTMF-toner, vil man kunne opdage alle de rene toner ud fra signalet. Det er dog ikke til at sige, hvilken DTMF-tone, som kommer hvornår. Derfor kan hele signalet ikke afkodes på én gang.

Det er derfor nødvendigt at dele signalet op i mindre dele, kaldet vinduer. Jo flere dele signalet deles op i, desto længere tid og kraft skal der bruges for at afkode signalet. For at afkode og bestemme rækkefølgen af rene toner, bliver man dog nødt til, som minimum, at dele signalet op med vinduesstørrelser, som svarer til varigheden, hvormed en DTMF-tone bliver afspillet. Vindueslængden må altså ikke være større end længden af DTMF-tonerne. Eftersom DTMF-toners frekvensområde ligger inden for den menneskelige hørelse, betyder det at mange lyde, som vi kan høre, kan have indflydelse på signalet. Derfor kan tale, musik og lyde fra omverdenen blandt andet medføre, at bestemte DTMF-frekvenser forstærkes, og at der kommer andre frekvenser ind i signalet. Derfor er det nødvendigt at filtrere signalet ved f.eks. at designe filtre, som fjerner uønskede frekvenser, eller ved kun at kigge på DTMF-frekvenser over en bestemt øvre amplitudetærskel.

8.4.1 Vinduer

Opdeling af signalet i vinduer giver følgende muligheder:

- At bestemme rækkefølgen af toner.
- Synkronisering.
- Filtrere forkerte signaler fra.

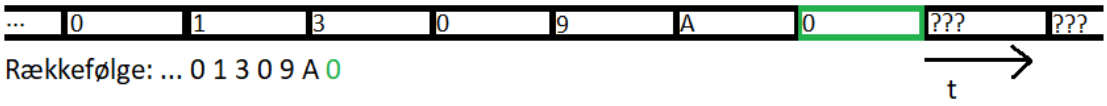


Figur 20: Eksempel på et signal delt op i vinduer.

Figur 20 viser, hvordan et signals DTMF-toner kan opdeles; her vist for to DTMF-toner, hvor de røde linjer er vinduerne, de blå bokse er DTMF-tonerne og de gule bokse har samme længde som DTMF-tonerne, men er pausen imellem dem.

8.4.2 Rækkefølgen af DTMF-toner

For at kunne skelne DTMF-tonerne fra hinanden i den rigtige rækkefølge, deles signalet som sagt op i stykker med samme længde som vinduerne. Vinduesstørrelsen må maksimalt være samme længde som DTMF-tonen, da det kan give noget uforudsigeligt, hvis der analyseres på pause-delen, hvor der udelukkende ligger tilfældig støj.
På den måde sikres det, at hvert vindue indeholder enten en DTMF-tone eller en pause. Hvis der laves kronologisk frekvensanalyse på hvert vindue, fås tonerne dermed også i den rigtige rækkefølge. Se figur 21.

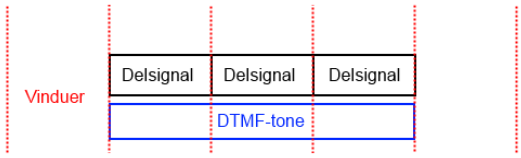


Figur 21: Vinduesrækkefølgen.

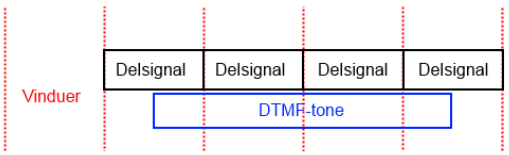
8.4.3 Synkronisering

Med en vinduesstørrelse svarende til en DTMF-tone, kræves der en form for synkronisering, for at undgå at vinduerne bliver skubbet.
Jo større synkroniseringsfejl, som er på signalet, desto større risiko er der for, at tonen ændrer sig. Enten kan man synkronisere modtager og afsender, eller man kan prøve at gøre vinduerne mindre.
Det blev besluttet at afprøve en metode, hvor vinduerne er mindre end DTMF-tonerne. For nærmere detaljer om disse forsøg, se *Journal 3* for Vindues metodik.
Dette vil samtidig gøre systemet mere modstandsdygtigt over for falske detekteringer, da der skal være flere godkendte delsignaler, før tonen genkendes.

Der bliver stadigvæk samplet i samplingsvinduer, altså dét, der kaldes vinduer (de røde linjer i figur 22). Hvis en DTMF-tone (den blå boks i figur 22) bliver opdelt i forhold til disse vinduer, opstår der delsignaler, som er bidder af n styk (de sorte bokse i figur 22). For at et signal bliver godkendt, kræver det at flere af disse n delsignaler bliver godkendt. En metode kunne være, at alle n delsignaler skulle godkendes, men dette ville stadigvæk kræve en forholdvis nøjagtig synkronisering mellem afspilning og sampling, og ville være ligeså følsom for støj som et enkelt vindue. Se nedenstående figur 22.



Figur 22: Komplet synkronisering.



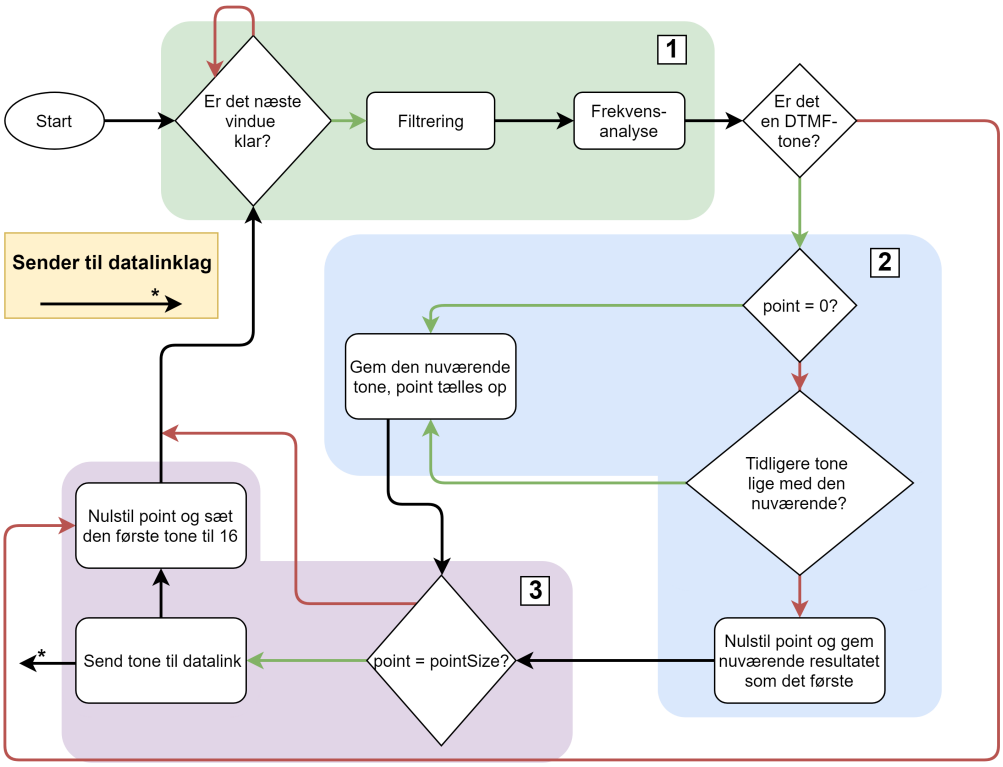
Figur 23: Ukomplet synkronisering.

Hvis der derimod kun kræves *n-1* godkendte delsignaler, burde synkronisering ikke være nødvendigt, da der i værste tilfælde altid vil ligge *n-1* komplette delsignaler. Dette tillader samtidigt en lille fejlmargin på signalet. Se figur 23.
Hvis hvert signal kun opdeles i to delsignaler, ville det kun kræve ét godkendt delsignal, før tonen var godkendt. Dette er ikke hensigtsmæssigt, da det netop ønskes at der skal flere detekteringer til, for at tonen skal godkendes. Derfor blev der først afprøvet med tre delsignaler, som på figur 22 og 23.

Dette gav et uventet resultat, for når situationen fra figur 23 opstod, kunne signalet være tilstrækkeligt tilstede til, at der blev godkendt fire delsignaler, hvilket resulterede i at algoritmen godkendte DTMF-tonen to gange.
For at undgå dette blev vinduesstørrelsen sat yderligere ned, så antallet af delsignaler blev hævet til fire, hvor der så kræves tre godkendelser, før tonen er godkendt. Hvis de to "halve" delsignaler bliver godkendt, og der derfor kommer fem godkendte delsignaler, så vil det stadigt kun give én godkendt DTMF-tone.

8.4.4 Algoritme

På figur 24 kan man se et flowchart over afkodningsalgoritmen.



Figur 24: Flowchart over afkodningsalgoritmen.

Ud fra overvejelserne fra de sidste par afsnit kan der fremstilles en algoritme til at afkode en besked bestående af DTMF-toner. Algoritmen kræver fire variable:

- *confirm*: Et array som indeholder tidligere resultater fra analysen.
- *point*: En heltalsværdi som peger på det næste element, der skal udskiftes i *confirm*.
- *pointSize*: En heltalsværdi som angiver den maksimale størrelse på *point* og *confirm*.
- r: En heltalsværdi, som indeholder det nuværende resultat af analysen.

1. Som det tidligere er nævnt, bliver signalet gemt i en buffer, som hedder *DTMF_Slices-BufferRecieve*. Først hentes et vindue ud af bufferen, som filtreres og analyseres. Når analysen er færdig gemmes resultatet i r. Dette er ikke direkte en del af algoritmen, men nødvendig for at kunne forsætte.

2. Der findes to mulige forløb. Det første forløb hænder, hvis r er under 16. Her bliver det tjekket om arrayet er tomt, altså om *point* er nul. Hvis dette er tilfældet, gemmes resultatet på den første plads i *confirm*, og *point* inkrementeres. Er arrayet ikke tomt bliver det undersøgt om r er lig det første element i arrayet, og er dette tilfældet, bliver resultatet tilføjet. Ellers nulstilles *point* og r bliver gemt.

3. Hvis *point* herefter er lig med *pointSize*, og arrayet derfor er fyldt, bliver det første element på *confirm* sendt til datalink-laget, hvorefter *point* sættes til nul, og det første element på *confirm* bliver sat til 16. Herefter starter algoritmen forfra.

Det andet forløb udføres, hvis r er større eller lig med 16, og er noget simplere. Her bliver *point* sat til nul, første element på *confirm* bliver sat til 16, og der startes forfra.

8.4.5 Afkodning med flere tråde

Det antages at man har en tonetid på 200 ms, en vinduesstørrelse på 50 ms, en analysetid på 80 ms per vindue uden filtre, samt at afkodning og samlingen af vinduer kun tager 1 ms. Dette medfører en forsinkelse på 50 ms på det første vindue og herefter 31 ms pr. vindue. Når hele signalet er analyseret og afkodet vil dette give en forsinkelse på 81 ms ved forsendelse af ét vindue. Det virker ikke umiddelbart som en særlig stor forsinkelse og har ikke specielt stor betydning, når man kun skal afkode én besked, men skal der modtages en fil med mange vinduer, kan 31 ms forsinkelse per vindue plus 50 ms fra første vindue, hurtigt løbe op.

Ud fra ovenstående eksempel kan man se to ting. For det første vil der altid være en forsinkelse på det første vindue. Derudover kan man se, at en stor forsinkelse kommer fra analysen. Ønskes en mindre forsinkelse er det kun disse ting, der kan ændres på. Sættes tonetiden ned, hvilket mindsker vinduesstørrelsen, vil forsinkelsen på det første vindue mindskes, men eftersom analysen forbliver uændret, har det ingen betydelig indflydelse.
Derfor må man forsøge at mindske forsinkelsen på analysen. En mulighed ville være at køre flere analyser samtidig, hvilket kan realiseres ved hjælp af tråde. Derfor blev algoritmen undersøgt for opgaver, som kan køre parallelt efter lidt modificering. Der blev fundet frem til tre forskellige steder: At samle vinduerne og filtrere dem, selve analysen, og afkodningen.

Man ville forvente, at filtre ville ligge under analysen, men da disse er afhængige af tidligere input og output, er de forhindret i at køre parallelt.
For at kunne få afkodning til at løbe parallelt bliver den opdelt i to metoder, og der tilføjes to ekstra. Den første metode håndterer samling af vinduer og filtrering. Analysen af signalet har sin egen klasse, og derfor skal den ikke modificeres. Den anden metode er selve afkodningen.

De to ekstra metoder håndterer en buffer, som holder styr på de forskellige analyser. Det blev valgt at lave bufferen som en ringbuffer af samme grund som beskrevet tidligere. Størrelsen på ringbufferen kommer til at afhænge af forsinkelsen på analysen. Den første metode står for oprettelse og udførelse af analysen. Den anden metode bliver brugt til at sikre, at afkodningen tager de analyserede resultater i den rigtige rækkefølge.

8.5 Kommunikationsparametre

For at lære mere om hardwarens virkemåde og undersøge om de udleverede mikrofoner og højtalere har dårlige frekvensområder, udføres et forsøg, beskrevet i *Journal 1.*
Da DTMF-tonerne bliver verificeret i forhold til, om deres amplitudeværdi er over en vis øvre amplitudetærskel, er det ret væsentligt at bestemme netop dennes værdi.
Denne værdi har en enhed, som er udtrykt ved volt. Grunden til at enheden er i volt er, at mikrofonen oversætter lydtrykket til en spænding, som igen oversættes til en bitværdi. Da amplitudeværdierne kun sammenlignes indbyrdes, ignoreres denne enhed fremover.
Det undersøges om der er forskellige amplitudeværdier til de forskellige rene toner, ift., hvilken PC der afspiller og optager, samt hvilket mikrofon- og højttalersæt (fremover kaldet hardwaresæt) der bruges.

De to udleverede mikrofoner og de to højttalersæt, er blevet parret sammen til to faste hardwaresæt. Derfor ønskes det i første omgang at bestemme hver rene tones amplitudeværdi, og derefter undersøge, om den er identisk med den samme tone afspillet og optaget på en anden PC med det andet hardwaresæt.
Det viste sig, at det ikke var muligt at gentage disse forsøg og få resultater der korrelerede; ikke engang for samme PC og hardwaresæt. Som eksempel kan det nævnes at DTMF-tonen F, bestående af de rene toner 941 Hz og 1633 Hz, blev målt med tre opstillinger, der hver havde 15 afspilninger og gav følgende resultat:

Frekvens	Amp forsøg 1	Amp forsøg 2	Amp forsøg 3
941 Hz	17000	26384	28716
1633 Hz	395590	669180	300690

Tabel 4: Amplitudeværdier til DTMF-tonen F.

Disse optagelser var lavet med to højttalere. En forskellig indbyrdes vinkling mellem disse, kunne give en tilfældig interferens eller fasedrejning, som gav udslag i ændring af amplitudeværdien mellem de forskellige opstillinger.

På trods af en omhyggelig opstilling og opmåling kan det ikke udelukkes, at de to højttalere måske kunne have været vinklet lidt forskelligt. Dette ledte til de næste forsøg, hvor der bliver afspillet igennem én højttaler .
Der blev lavet to små forsøg på samme PC og hardwaresæt med to separate opstillinger, hvor DTMF-tonerne 0, 5, A og F, i hvert forsøg blev afspillet 15 gange. Dette foregik i samme lokale som forsøgene i *Journal 1*, og denne gang lykkedes det at få sammenhæng mellem de to separate forsøg.
I et større lokale blev der herefter lavet to separate forsøg for alle DTMF-toner, hvor hver blev afspillet 15 gange. Dette forgik på to forskellige PC'er med hvert deres hardwaresæt.

For nærmere beskrivelse se *Journal 2*.
Det var igen muligt at se en sammenhæng mellem de to forsøg. Selvom der er lidt afvigelser, kan der ses en tendens. Her vises den gennemsnitlige amplitudeværdi for hver af de otte rene toners otte optrædener, samt deres gennemsnitlige afvigelse.

Frekvens	Gennemsnitlig amplitude	Gennemsnitlig afvigelse	i %
697 Hz	178599	27623	15,27
770 Hz	190933	23848	12,49
852 Hz	244181	5568	6,38
941 Hz	179779	17671	9,83
1209 Hz	149599	7681	5,13
1336 Hz	227106	14065	6,19
1477 Hz	335183	9020	2,69
1633 Hz	228161	5543	2,43

Tabel 5: Amplitudeværdier til DTMF-frekvenserne med gennemsnitlig afvigelse.

Over disse afspille- og optageforhold var det den rene tone ved frekvensen 1209 Hz, der gennemsnitligt gik svagest igennem.

8.5.1 Fastsættelse af den øvre amplitudetærskel

Ud fra observationerne gjort i journaler, samt utallige andre forsøg, er det erfaret, at det er særdeles vanskeligt at lave en 100% fejlfri kommunikation via DTMF-toner transmitteret gennem et luftmedie, da der ind imellem sker noget finurligt. Derfor er der valgt en kvantitativ metode, hvor der er foretaget en god mængde målinger, og ud fra disse vælges værdier, som vil virke for det meste, men ikke hver eneste gang. Her ses en tabel over de 8 DTMF-frekvensers gennemsnitlige amplitudeværdier og standardafvigelse:

Frekvens	Gennemsnitlig amplitude	Standardafvigelse
697 Hz	178599	34964
770 Hz	190933	28158
852 Hz	244181	20788
941 Hz	179779	20303
1209 Hz	149599	9611
1336 Hz	227106	20599
1477 Hz	335183	13080
1633 Hz	228161	7201

Tabel 6: Amplitudeværdier for DTMF-frekvenserne med standard afvigelse.

Det vælges at regne med to standardafvigelser, da detekteringerne teoretisk vil ramme inden for dette område i 95 % af tilfældene.
På trods af at 1209 Hz gik svagest igennem, kan 967 Hz, grundet to standardafvigelser, fremkomme svagere, og derfor bruges dennes amplitudeværdi.

Herved vil 967 Hz ift. to standardafvigelser, ligge i området 108000 til 248000 (se tabel 6). Dette svarer til en minimal amplitudeværdi på 108000. Da denne værdi er i maksimumspunktet, sættes den øvre godkendelsestærskel for amplituden til ca. det halve, altså 50000.

Dette medfører, at systemet bliver mere modtagelig for støj, men da algoritmen regner på de to kraftigste frekvenser, som er større end tærsklen, der oftest vil være DTMF-frekvenserne, når der analyseres på en DTMF-tone. Årsagen til at den øvre amplitudetærskel ikke sættes til 0 er, at hvis der analyseres på ren støj, vil algoritmen regne på de to stærkeste frekvenser, som kunne være identiske med to DTMF-frekvenser, og derfor fejlagtigt give en detektering. Halveringen gør også, at systemet vil virke på afstande større end 1 meter, som er valgt til den omtrentlige transmissionsafstand. Derfor behøver hardwaren ikke at blive opstillet med den helt store præcision, hverken ift. afstand eller retning.

Da der er valgt at bruge fire vinduer, skal tallet for den øvre amplitudetærskel, som repræsenterer værdien ved 100 ms, deles med fire, så det svarer til delsignalet på 25 ms. Dette resulterer i, at 12500 vælges som den øvre amplitudetærskel.

8.5.2 Delkonklusion

Der kommunikeres via én højttaler på en afstand op til en meter med PC-indstillinger som beskrevet i *Journal 1*. Hver DTMF-tone sendes i 100 ms og analyseres i fire delsignaler af 25 ms. Hver DTMF-tone adskilles af en pause, som har samme længde som et vindue. En DTMF-tone godkendes, hvis tre af disse delsignaler overskrider den øvre amplitudetærskel på 12500 og indeholder en DTMF-tone.

8.6 Filtrering af signalet

8.6.1 Filtertyper

Der er overordnet to forskellige filtertyper: IIR (Infinite Impulse Response) og FIR (Finite Impulse Response). Disse typer kan give samme resultat, men de har forskellige egenskaber, som foretrækkes i forskellige tilfælde. Det som angiver filtertypen, handler om, hvilke ting de afhænger af. FIR-filtre afhænger kun af tidligere inputs, hvorimod IIR også afhænger af tidligere outputs. Normalt bliver filtre beskrevet vha. en overførelsesfunktion i z-domænet:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + ... + b_m z^{-m}}{1 + a_1 z^{-1} + a_2 z^{-2} + ... + a_n z^{-n}}$$

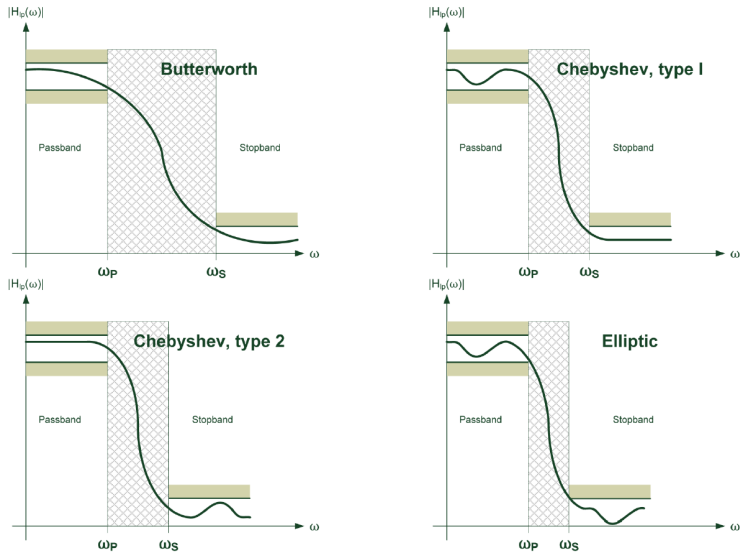
hvor z^{-i} angiver, at der er en forsinkelse på i samples. Denne overførelsesfunktion er et IIR-filter. For et FIR-filter gælder det, at $a_1, a_2, ...a_n = 0$ så der kommer til at stå:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + ... + b_m z^{-m}}{1} = b_0 + b_1 z^{-1} + b_2 z^{-2} + ... + b_m z^{-m}$$

Normalt er $m = n$, men det er ikke sikkert.
Et FIR-filter har en højere filterorden end IIR, hvilket betyder, at FIR-filteret normalt kræver mere computerkraft end et IIR-filter.
Derfor lægger det op til, at man altid vil bruge IIR. FIR-filtre har dog den fordel, at de har en lineær faseforskydning over frekvenserne, hvilket IIR ikke nødvendigvis har. Det betyder, at hvis man skal benytte fasen af signalet, vil man ofte anvende et FIR-filter. Derudover er et FIR-filter lettere at designe.
En af de store forskelle mellem FIR- og IIR-filtre er, at FIR-filtre er beskrevet ved hjælp af formler, som kan bruges direkte. IIR-filtre er derimod beskrevet analytisk, så de ofte designes ved hjælp af tabelopslag.
Eftersom fasen ikke har nogen betydning i dette projekt, er der valgt at lave et IIR-filter.

8.6.2 Oprettelse af IIR-filter

Der findes flere forskellige måder at lave et IIR-filter på. Den metode som bliver brugt her er Bilinear Transformation (BLT). Dog benyttes MATLABs Toolbox: Digital Signal Processing, som indeholder en app, der kan lave BLT. Man kan se en kort gennemgang af BLT i det elektroniske bilag: *Oprettelse af IIR*. Det er vigtigt at huske, at ved BLT laves et analogt prototype-lavpasfilter, som der findes forskellige af, og som har forskellige egenskaber. På figur 25 ses nogle af typernes egenskaber.



Figur 25: Forskellige typer af analoge prototyper.

Når man laver filtre skal man huske, at de skal være stabile (se bilag *Oprettelse af IIR*), hvilket MATLABs Toolbox også håndterer.

8.6.3 Anvendte filtre

Der benyttes flere filtre i projektet, og i tabel 7 ses de vigtigste oplysninger, som anvendes i "Filter Designer". Det første filter, der blev lavet, var et båndpasfilter til at fjerne støj lavere og højere end DTMF-frekvenserne, og derved er det med til at fjerne aliaseringen, hvilket er frekvenser, der er over den halve samplingsfrekvens. Hvis man kigger på figur 25 vil det optimale være Chebyshev type 2, da der ikke er ripples i pasbåndet. Den har dog et længere overgangsbånd end Elliptic.

Ved lidt arbejde og tilpasning af værdier kan næsten alle ripples i pasbåndet fjernes eller gøres så små, at de ikke har nogen signifikant betydning. Derudover fremstilles båndstopfiltre mellem hver DTMF-frekvens, der sammen med båndpasfilteret, udgør filterkonfigurationen kaldet "Party Mode". Når det kun er båndpasfilteret, bliver filterkonfigurationen kaldt for "Basis". Herunder kan man se de forskellige værdier, som blev brugt til at bestemme filtrene med.

	Filter	Type	Match exactly	[Hz]				[dB]	
				F_{stop1}	F_{pass1}	F_{pass2}	F_{stop2}	A_{pass}	A_{stop}
Party Mode	Basis	Båndpas	Stopbånd	500	600	1700	1800	0,1	60
	697 - 770 Hz	Båndstop	Stopbånd	700	720	750	760	0,1	60
	770 - 852 Hz	Båndstop	Stopbånd	780	790	840	850	0,1	60
	852 - 941 Hz	Båndstop	Stopbånd	855	865	928	938	0,1	60
	941 - 1209 Hz	Båndstop	Stopbånd	945	955	1195	1205	0,1	60
	1209 - 1336 Hz	Båndstop	Stopbånd	1210	1220	1325	1335	0,1	60
	1336 - 1477 Hz	Båndstop	Stopbånd	1340	1350	1460	1470	0,1	60
	1477 - 1633 Hz	Båndstop	Stopbånd	1480	1490	1620	1630	0,1	60

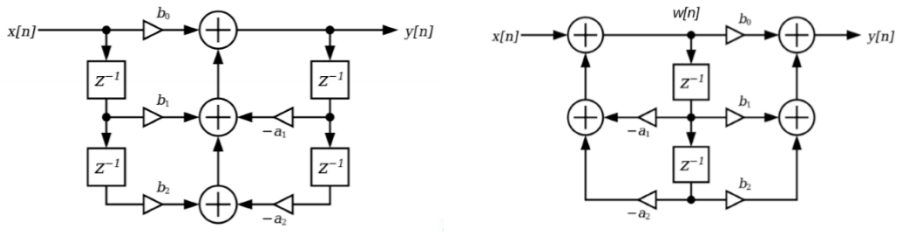
Tabel 7: Oplysninger til MATLAB-appen "Filter Designer" i Digital Signal Processing Toolbox.

Som det fremgår af tabel 7 er dæmpningerne i stopbåndet sat til 60 dB for at reducere påvirkningen af ripples mest muligt.

8.6.4 Implementering af filtre

Der er fire forskellige måder man kan implementere filtre på. Den første hedder *direkte form 1*, hvilket er den mest simple, da man her anvender overførelsesfunktionen direkte. Det kræver to buffere at implementere denne form, en til tidligere input og en til outputs. Dette betyder, at denne metode hurtigt kan tage meget plads i hukommelsen, og derfor er der lavet en *direkte form 2*. Den er mere kompleks, men skal derimod kun bruge én buffer, altså spares halvdelen af hukommelsesforbruget.
Figur 26 visualiserer realiseringsstrukturerne for *direkte form 1* og *direkte form 2*. Der er dog et problem ved disse to former. Når de skal implementeres i computersystemer, er

der et begrænset antal cifre til hver værdi, hvilket kan ændre, hvordan filteret fungerer, jo større orden det har. Derfor blev der lavet en anden måde at implementere dem på.



Figur 26: Viser strukturerne for henholdsvis direkte form 1 og 2.

Den hedder kaskadeform. Kaskadeformen handler om at man har delt overførelsesfunktionen op i flere mindre overførelsesfunktioner, som skal multipliceres sammen.

$$H(z) = H_1(z)H_2(z) \ldots H_i(z)$$

Hver H_i er af typen *direkte form 2* og er enten andenordens- eller førsteordens-overførselsfunktion. Da computere udviklede sig til at kunne have flere tråde, kom der en ny struktur, som var parallel:

$$H(z) = H_1(z) + H_2(z) + \ldots + H_n(z)$$

I projektet er der blevet gjort klar til at implementere filtre i form af deres parallelstruktur og som *direkte form 2*, da et parallelt vil tage kortere tid at udføre end samme filter på kaskadestruktur. MATLAB var kun i stand til at lave kaskadefiltre, og når der skulle laves partialbrøksopløsning på filtret, var det ikke muligt at holde konstanterne reelle, og derfor blev det nødvendigt at beholde filtrene på kaskadeform. Herefter skubbes det til næste iteration at implementere filtre med komplekse konstanter. For at forsimple udskiftning af filtre er filterkoefficienterne gemt i en .txt-fil, som bliver indlæst når programmet startes. På denne måde kan der fremstilles bedre filtre til forskellige computere og situationer.

8.6.5 Delkonklusion på filtre

Der anvendes tre forskellige filterniveauer. Ved første niveau anvendes der ikke noget filter. Det næste niveau er et enkelt IIR-båndpasfilter; i dette tilfælde kaldet "Basis", der frasorterer alle frekvenser, som ikke ligger mellem den laveste og højeste DTMF-frekvens. Det sidste niveau er "Party Mode", da det indeholder "Basis"-konfigurationen samt IIR-båndstopfiltre mellem hver DTMF-frekvens. Grunden til niveauopdelingen er for at reducere behandlingstiden af signalet og spare computerkraft, da de mere komplekse filterkonfigurationer kun er nødvendige i visse tilfælde.
Det er besluttet, at hvis en transmission fejler tre gange anvendes "Basis"-konfigurationen, og efter fem anvendes "Party Mode"-konfigurationen.

9 Konklusion

Softwaren er delt op i tre lag: Applikationslaget, der er opdelt i en serverside og en klientside, datalink-laget og det fysiske lag. Forespørgsler og svar sendes gennem datalink-laget og videre til det fysiske lag, hvor de afspilles og optages.
Herunder sammenlignes de fire overordnede problemstillinger fra problemformuleringen med den tilhørende konklusion.

- Til overførsel af data imellem de forskellige lag er der implementeret fire ringbuffere. Denne type er valgt for at genbruge hukommelsesadresser og spare på ressourcer, da elementerne skal flyttes efter First-In-First-Out-princippet. Eftersom hver buffer kan tilgås af to forskellige lag på samme tid, kan der opstå en såkaldt "race condition". For at undgå dette anvendes en mutex lock.
  Det er også muligt at håndtere processer i softwaren samtidigt uden fejl i dataet. Det realiseres vha. tråde. Trådene gør det muligt at køre processer samtidig, således, at de forskellige processer ikke skal vente på hinanden.

- Klienten er i stand til at oprette, hente og fjerne filer fra serveren; det er også muligt at få tilsendt en liste over alle filer på serveren. Klientens forespørgsler sendes igennem datalink-laget og videre til serveren. Serveren modtager de ønskede forespørgsler og udfører, hvad klienten ønsker, hvis det er acceptable instruktioner. Serveren udfører en besvarelse på instruktionerne, som sendes tilbage igennem datalink-laget og videre til klienten.

  Klienten udfører forespørgsler via en menu i Kommandoprompt, som er en simpel brugergrænseflade. Her er det muligt at skrive de fire forskellige kommandoer: opret, hent, fjern og katalogisering. Ydermere er det muligt at få yderligere hjælp fra en hjælp-menu. Brugeren kan maksimalt sende fem kommandoer på én gang; de sendes automatisk, hvis denne grænse opnås. Hvis ikke, kan man manuelt sende kommandoer, før der er indtastet fem. Brugergrænsefladen er visualiseret med et inputfelt og outputfelt; et felt til indtastede kommandoer samt et felt til visning af eksisterende filer på serveren. Det er ikke muligt at oprette andre filtyper end .txt.

- I datalink-laget pakkes dataet ind i frames, der indeholder flag til at angive begyndelse og ende, adressefelt, kontrolfelt til frametype- og nummer, samt et felt til fejlkontrol. Til sidstnævnte blev den cykliske kode CRC-8 anvendt, da dette er en meget udbredt og effektiv metode til fejldetektering.

  For at styre kommunikationen mellem noderne blev en Stop-and-Wait-protokol implementeret, hvor der ventes på en kvittering efter hver afsendt frame. Denne blev valgt for sin simplicitet, og da mediet er lyd undgås det at noderne snakker oven i hinanden.

Der skelnes ikke mellem forskellige enheder, der kan kun være én server og én klient. Serveradressen er altid "0000", og klientadressen er altid "1111".

- Lyde kan håndteres i C++ gennem lydbiblioteket SFML. Der kan genereres, afspilles og optages DTMF-toner.

  Højttalerne og mikrofonernes egenskaber er blevet undersøgt, og det viste sig, at det er væsentligt lettere at opnå reproducerbare målinger, når der kun blev afspillet fra én højttaler. Dette skyldes formentlig interferens, som kan opstå, når der afspilles fra to lydkilder.

  Lokalets størrelse viste sig at have stor indflydelse på, hvor tydeligt signalet bliver opfanget. Jo større et lokale, desto svagere blev DTMF-tonerne opfanget, hvilket formentlig skyldes forskellen på lokalernes rumklang. Der var på de to test PC'er fra journalerne ikke nævneværdig forskel på, hvilken PC målingerne blev foretaget på, ligeledes heller ikke mellem højtalere og mikrofoner. Signalets amplitudeværdi falder over afstand, og amplitudetærsklerne er fastsat for at sikre mulig transmission inden for en meters afstand. Afspilningsparametrene er sat med en sikkerhedsmargin, så systemet også virker på lidt længere afstand og helt udendørs.

  Alle de førnævnte tests blev udført på de samme to PC'er. Det viste sig, at afstanden gav problemer, når der blev anvendt andre PC'er end netop disse to, hvilket kan skyldes uforudsete forskelle i hardware. For at programmet kan virke på alle PC'er i projektgruppen, er den maksimale afstand reduceret til 60 cm.

  For at frasortere støj er der blevet sat to amplitudetærskler, som bruges til at lave en simpel filtrering. Først kigges der på amplitudeværdien af selve signalet i tidsdomænet kaldet nedre tærskel. Hvis gennemsnittet er under den nedre tærskel, bliver det set som et ikke-godkendt signal. Den øvre amplitudetærskel bliver brugt i frekvensdomænet, hvor DTMF-toner ofte vil have en betydeligt stærkere amplitude end almindelig støj. Hvis der er så meget støj, at kommunikationen mislykkedes, er der lavet en række filtre til at minimere støjens betydning. Filterne bliver delt op i to niveauer; "Basis"som kun er et båndpasfilter og "Party Mode", som består af et båndpasfilter og en række stopbåndfiltre mellem hver DTMF-frekvens.

  Det er lykkedes at oversætte et signal, som består af DTMF-toner i modtaget rækkefølge. Dette er gjort ved at dele signalet op i vinduer, hvor størrelsen er mindre end den afspillede tones tid. Ved hvert vindue bliver der analyseret for hvilken DTMF-tone, der optræder. Dette foretages ved frekvensanalyse ved hjælp af Fast Fourier Transformation. Kun den stærkeste af de lave og høje DTMF-frekvenser bliver brugt til at finde den rigtige DTMF-tone. Dog har det vist sig, at der opnås en mere robust kommunikation, ved at hæve antallet af vinduer fra 4 til 5, og godkende DTMF-tonen ved 4 i stedet for 3 godkendte delsignaler.

10 Perspektivering

1. Multi-bruger-funktionalitet

Programmet skelner ikke mellem forskellige personer, som anvender klienten. De ændringer, der foretages i serveren via klienten, realiseres for alle personer, som tilgår serveren. På denne måde er det ligegyldigt, hvem der opretter, henter eller fjerner en fil fra serveren. En forbedring ville være at tilføje en multi-bruger funktionalitet vha. DCHP-protokollen. Denne protokol sørger for korrekt tildeling og administration af adresser til hver bruger.[4] Med multi-bruger funktionaliteten implementeret, ville det være muligt at tildele personlige mapper til hver bruger. Herudover ville en delt mappe mellem brugerne også være mulig, hvor en funktion opdaterer en brugers adfærd på serveren for de andre brugere, når de logger ud og ind igen. Ydermere ville det være muligt at privatisere bestemte filer for bestemte brugere, således andre brugere ikke kan tilgå disse.
Multi-bruger funktionaliteten blev dog påbegyndt, men blev ikke fuldendt til anvendelse.

2. Flere filtyper

I problemformuleringen lød et krav på håndtering af forskellige filtyper i C++. Kravet blev hurtigt afslået som en reel implementering i programmet. Fokus lå primært på sikker kommunikation frem for applikationsmuligheder. Forbedringen blev udskudt til senere hen i projektforløbet, men der ikke tilstrækkelig tid. Tilføjelse af billeder i form af f.eks. .png og .jpeg, ville have været interessant, men besværligt.

3. Kommunikationsprotokol

Selve dialogen mellem klient og server blev implementeret med en Stop-and-Wait-protokol. Dette var tiltænkt som en pladsholder til testformål, for senere at implementere en mere avanceret protokol. Det ville være en åbenlys optimering at implementere en form af Go-Back-N, og endnu bedre en Selective Repeat-protokol, hvor modtageren kun skal sende ACKs efter f.eks. 10 frames, og enten sige "alt modtaget" eller at afsender skal gensende specifikke frames. Det samlede program kom desværre ikke hurtigt nok på benene til, at der kunne eksperimenteres med mere avancerede protokoller.

4. Redundans i frames

Der endte med at være en del redundans i hvert frame. Som nævnt blev adresserne aldrig brugt til noget, så de kunne helt udelades. I kontrolbittene er der desuden tre redundante bit, som kunne bruges i tilfælde af piggybacking, og da der aldrig blev implementeret andet end Stop-and-Wait, kunne man også håndtere frame-nummereringen med en enkelt bit i stedet for tre.

5. Framekonverteringer

Under Inframe bliver ASCII-strengen først lavet til en streng af binære værdier for at tilføje de nødvendige elementer, for til sidst at blive oversat tilbage til ASCII-værdier. Her opstår der yderligere redundans, da antallet af bit skal gå op i otte i stedet for fire. Når beskeden sendes videre til bufferen til det fysiske lag, konverteres den til typen bitset. Alle disse omskrivninger kunne formentlig være undgået ved at bruge bitset hele vejen igennem, men disse skal dog have en fastsat størrelse inden brug, hvor almindelige strenge er mere fleksible og nemmere at arbejde med.

6. Faseforskydning

Det ville være interessant at kigge på faseforskydning, som muligvis kan bruges til at lave kommunikation fra half-duplex til full-duplex. Idéen er, at når lyd bevæger sig, vil der ske en faseforskydning over afstanden. Derfor vil det være svært for to computere at have samme faseforskydning. Derudover vil der i teorien ikke være nogen faseforskydning, hvis en computer kigger på sit eget signal.

7. Goertzel-algoritmen

Undervejs i projekt blev det opdaget, at der findes en detekteringsmetode, der hedder Goertzel Algoritme, som kan bruges til at afkode DTMF-toner. Da denne opdagelse blev gjort, var algoritmen til detekteringen allerede skrevet, og derfor blev Goertzel ikke undersøgt nærmere.

8. Forskelle mellem PC'er

Der opstod problemer med kommunikationen, da programmet blev testet på forskellige PC'er over forskellige afstande. Det kunne muligvis skyldes forskelle i PC'ernes hardware og versioner af Windows. Det kunne være interessant at undersøge nærmere på denne hypotese.

11 Litteraturliste

I rapporten refereres til kilder fra anvendte bøger i lektionerne. Disse anvendes generelt til hele sektioner og opstilles her:

- Afsnit 5.2: Silberschatz: Structured Computer Organization. 9. udg. Wiley, 2013
- Afsnit 7: Forouzan, Behrouz A.: Data Communications and Networking. 5. udg. McGraw-Hill, 2013
- Afsnit 8: Li, Tan og Jiang Jean: Digital Signal Processing - Fundamentals and Applications. 2. udg. Elsevier Inc., 2013

Andre henvisninger:

[1] DTMF. Udgivet af Wikipedia. Internetadresse: https://da.wikipedia.org/wiki/DTMF - Besøgt d. 02.11.2018

[2] Halliday: Principles of Physics. 10. udg. Wiley, 2013

[3] Fast Fourier Transform. Udgivet af Wikipedia. Internetadresse: https://da.wikipedia.org/wiki/Fast_Fourier_Transform - Besøgt d. 14.11.2018

[4] DHCP. Udgivet af Wikipedia. Internetadresse: https://da.wikipedia.org/wiki/DHCP - Besøgt d. 06.12.2018

12 Oversigt over elektronisk bilag

Designklassediagrammer:

- SamletDKD.png

Flowcharts:

- Flowchart_Client.png
- Flowchart_Server.png

Journaler:

- Analyseværktøjer
- Journal 1: Hardware-egenskaber del 1
- Journal 2: Hardware-egenskaber del 2
- Journal 3: Vindues-metodik

Soundbox - Kode:

- SFML-2.5.1
- Soundbox - Header- og CPP-filer
- Soundbox - Programmet
- SFML guide.pdf

Øvrige bilag:

- Oprettelse af IIR.pdf
- Projektoplæg.pdf
- SoundBox - Guide til klientens brugergrænseflade.pdf
- Tidsplan.pdf