Tin Can Skype

RB-MRO3 - Gruppe 3

Uddannelse og semester:

Robotteknologi - 3. semester

Afleveringsdato:

18. December 2015

Vejleder:

Ib Refer (refer@mmmi.sdu.dk)

Gruppemedlemmer:

Anders Ellinge (aelli14@student.sdu.dk)
Anders Fredensborg Rasmussen (andra14@student.sdu.dk)
Daniel Holst Hviid (dahvi14@student.sdu.dk)
Mathias Elbæk Gregersen (magre14@student.sdu.dk)
Rasmus Skjerning Nielsen (rasni14@student.sdu.dk)
René Tidemand Haagensen (rehaa14@student.sdu.dk)
Sarah Darmer Rasmussen (srasm14@student.sdu.dk)

Det Tekniske Fakultet
Syddansk Universitet

1 Abstract

This project demonstrates how to create at chat application using C++. The computers using the application must be able to communicate with DTMF tones and a communication protocol must be dessigned. The created application also includes a login and user history feature.
Agile software development, in partucular Scrum, has been used to simplify our project challenges by dividing them into smaller issues.
The created software is divided into layered softwarearchitecture by using datacommunication layers, such as the physical layer, the data link layer, the transportation layer and the application layer. Our main issues were solved by using: Goertzel for efficient tone recognition, CRC for error detection in data packages and Stop-and-Wait for establishing a connection between two processes.

2 Forord

Denne rapport er udarbejdet af gruppe 3, på andet semester på Civilingenør i Robotteknologi på Syddansk Universitet. Rapporten er blevet skrevet i forbindelse med dette semesters projekt og beskriver hvordan denne gruppe har valgt at løse opgaverne i det valgte projekt, "Tin Can Skype", som er et chatprogram, der bruger DTMF-toner og indeholder bla. et simpelt log-in og historik system.
Formålet med denne rapport er, at læseren skal være i stand til at læse og forstå projektet ved blot at have grundlæggende viden om C++ og datakommunikation, og ved at læse rapporten.

I forbindelse med dette projekt, blev følgende udstyr stillet til rådighed:

- To mikrofoner
- To højtalere

Indhold

3 Indledning

3.1 Projektbeskrivelse

I dette projekt er to højtalere og to mikrofoner blevet stillet til rådighed. Formålet med dette projekt er, at kunne sende data vha. DTMF-toner.
Det valgte projekt er et chatprogram, der udvikles i C++, og skal have de primære funktioner:

- Overførsel af tekst.
- Log-in funktioner.
- Historik af chat-samtale.

Desuden er disse sekundære funktioner blevet overvejet:

- Filoverførsel
- Gruppe chat
- Spil
- Redigering af tidligere beskeder
- Video streamings funktioner
- Humørikoner

Herudover er der desuden blevet overvejet at bruge en tredje computer, som kan bruges som en server. Her vil mindst to computere altså være i stand til at kommunikere med hinanden vha. DTMF-toner.

3.1.1 Krav til produktet

Følgende krav blev stillet til projektet:

- 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
- Der skal udvikles en distribueret applikation i C++
- Der skal anvendes en lagdelt softwarearkitektur
- Arkitekturen kunne være client/server med f.eks. tykke klienter

For at fuldføre dette projekt skal der anvendes to computere som skal være i stand til at kommunikere med hinanden ved hjælp af lyd i form af DTMF-toner. Derudover skal dette programmers i C++, her bruges klasser.

3.1.2 Metodebeskrivelse

Vi har i dette projekt valgt at benytte SCRUM, da alle gruppe medlemmer således er i stand til at arbejde med den metode der passer dem bedst. Vi har valgt at bruge brainstorm, som vores primære form for idégenererings-teknik. Desuden prøver gruppen så vidt muligt at beregne alle de ting, der kan beregnes på forhånd.

3.1.3 Afgrænsning

Her ses de emner, som gruppen har overvejet at arbejde med. De primære funktioner er de funktioner som skal løses først, mens de sekundære løses efter tidsbegrænsning.
Primære funktioner:

- Overførsel af tekst
  - Protokol
  - Karakter definition
  - Størrelse
- Historik
  - Tidspunkt
  - Størrelse
- Log-in

Sekundære funktioner:

- Fil-overførsel
  - Protokol
  - Queue
- Gruppe chat
  - Protokol
- Spil database
  - Protokol
  - Funktion
- Rediger tidligere beskeder
  - Protokol
  - Funktion
- Stream funktion
  - Protokol
  - Funktion

- Humørikoner

  Char def.

  Database

- GUI

  Agil

- Sky "server"

  Funktion

3.2 Workload (product backlog)

Der laves en product backlog i stedet for en tidplan (se figur 1).

Requirement	Status	Priority	Estimate (Hr)	
Overførsel af tekst	Not started	1	70	
Log-in	Not started	1	40	
Historik	Not started	1	40	
Fil-overførsel	Not started	2	120	
Gui	Not started	2	70	
Rediger tidligere besked	Not started	3	40	
Gruppe chat	Not started	3	40	
Streaming	Not started	4	120	
Cloud/server (tredje computer)	Not started	4	100	
Smileys	Not started	4	30	
Spil	Not started	4	100	
		I alt	770	

Figur 1: Product backlog

En product backlog er et værktøj inden for metoden SCRUM, som viser hvor langt tid en opgaver tager i mandetimer og hvilken status opgaven har (Not started, In process og Finished).

4 Det Fysiske Lag

4.1 Teori

I følgende teoriafsnit er der lagt vægt på Nyquist rasen, Goertzel, DTMF og C++ og SFML.

4.1.1 Nyquist raten

Hvis et signal skal analyseres, skal sampling frekvensen være større end to gange den frekvens signalet har, dvs:

$$f_s > 2 \times f_{max} \tag{1}$$

Hvis dette krav ikke opretholdes, vil signal samplingen være påvirket af aliasing.

4.1.2 Goertzel

Goertzel er en speciel algoritme brugt til at udregne DFT (Diskret Fourier Transformation) koefficienter og signal spektrum, uden at bruge kompleks algebra som DFT.
Goertzel algoritmen er en filtreringsmetode for udregningen af DFT koefficienterne X(k) ved en bestemt frekvens bin, k.

$$k = \frac{f}{f_s} \times N \tag{2}$$

hvor f er den bestemte frekvens der ledes efter, og N er det totale antal samples der samples over.

Goertzel filteret opererer med en indput sekvens x(n) i en kaskade af 2 stadier med en parameter f, som er den frekvens der skal analyseres.

Filterets første stadie er et andensordens IIR filter:

$$s(n) = x(n) + 2 \times cos(2\pi f) s(n-1) s(n-2) \tag{3}$$

hvor der ved samplen $x(0)$ gælder at $s(-2) = s(-1) = 0$

Filterets andet stadie er et FIR filter:

$$y(n) = s(n) - e^{2\pi i f} s(n-1) \tag{4}$$

I en kaskade har filterets overføringsfunktion udseendet:

$$G(Z) = \frac{Y(Z)}{X(Z)} = \frac{1}{1 - 2 \times cos(\frac{2\pi k}{N}) z^{-1} + z^{-2}} \tag{5}$$

Den kvadreret DFT koefficient X(k) ved en bestemt frekvens bin k, dvs. det enkeltsidet spektrum,

er derfor givet således:

$$|X(k)|^2 = s(N-1)^2 + s(N-2)^2 - 2 \times cos\left(\frac{2\pi k}{N}\right) s(N-1)s(N-2) \tag{6}$$

4.1.3 DTMF

DTMF står for: "Dual-tone multi-frequency signaling", og er de kendte dial toner man kender fra telefonen. Hver tone er en kombinering af to sinusoidale signaler med frekvenser valgt ud fra et sæt af otte standardiserede frekvenser. Se figur 2

Hz	1209	1336	1477	1633
697	1	2	3	a
770	4	5	6	b
852	7	8	9	c
941	*	0	#	d

Figur 2: DTMF toner

Hos DTMF er det vigtigt at hver tone afspilles i længere end 40 millisekunde,r og at mellem hver tone er det en pause på 50 millisekunder.

4.1.4 C++ og SFML

C++ har pr. standard ikke funktioner til at afspille og optage lyd. Derfor blev biblioteket SFML installeret til at håndtere disse funktioner. SFML er en simple interface til de forskellige komponenter på ens pc, for at lette udviklingen af f.eks. spil og multimedia applikationer.
De klasser der bliver brugt fra SFML er:

- `sf::Sound`
  Bruges til at afspille lyd.
- `sf::SoundBuffer`
  Lagring af audio samples der definere en lyd.
- `sf::SoundRecorder`
  En abstrakt grund klasse til at optage lyd.
- `sf::SoundBufferRecorder`
  En specialiseret `SoundRecorder`, der gemmer optaget lyd i en lyd buffer.

4.1.5 Optag

Den første problemstilling projektet stødte på i forhold til optag, var at være i stand til at optage en DTMF tone og analysere optagelsen, hvorefter at fortælle hvilken tone der blev optaget.

Før en optagelse kan finde sted, blev en samplingsfrekvens for optagelse nødvendigvis defineret og fastsat. Denne samplingfrekvens er yderst vigtig for projektet, eftersom denne samplingsfrekvens dikterer hvordan resten af det fysiske lag skal oprettes.
Som et krav skal samplingsfrekvensen overholde Nyquist raten:

$$f_s > 2 \times f_{max} \tag{7}$$

Den højeste frekvens hos en DTMF tone er 1633 Hz, derfor skal samplingsfrekvensen som minimum være 3267 Hz. Til projektet blev samplingsfrekvensen sat til 8000 Hz, eftersom det er en ofte brugt samplingsfrekvens ved implementering af DTMF genkendelse, og at den opretholder Nyquist raten.

Projektets optagefunktioner hviler tungt på SFML bibliotekets klasser, `sf::SoundRecorder` og `sf::SoundBufferRecorder`, eftersom C++ egets bibliotek ikke tilbyder optagefunktioner.

Projektets første løsning på at optage en DTMF tone og analysere optagelsen var en simpel løsning, der dog viste sig ikke at være tilstrækkelig for dette projekt.
En optagelse blev oprettet ved først at lave et objekt f.eks. `sf::SoundRecorder` optag.
Derefter startede SFML sin egen tråd og optog, når man brugte funktionen `optag.start(8000)` (8000 Hz for projektets samplingsfrekvens). Koden sættes til at sove den tid der skal optages, hvorefter at optagelsen stoppes med `optag.stop()`. Optagelsens information blev derefter hentet ved at lave en reference til optagelsens buffer `sf::SoundBuffer &enBuffer`, derefter blev der oprettet en vektor af pointerer til bufferen `sf::Int16* samples = enBuffer.getSamples();`. Vektoren består nu af diskrete værdier for en optagelse, hvorpå der blev udført en DFT. Denne DFT vil således være i stand til at vise om en DTMF tone er til stede.
Første løsning på at optage en DTMF tone og analysere optagelsen er ikke forkert, den skabte umiddelbart noget kompleksitet, som projektet blev nødt til at tage højde for:

- Løsningen kræver at optageren skal vide hvornår en afspiller af en sekvens af toner er færdig, eftersom analysen af optagelsen først kan finde sted efter en endt optagelse.
- DFT er en tung udregning. Den har et forhold der siger der er N^2 udregninger ved en DFT på N samples. Dvs. efter en optagelse på 10 sekunder sker der

  $$(10sek \times 8000Hz)^2 = 6.400.000.000$$

  udregninger, hvilket er en uheldig eksponentiel stigning for udregninger af længere optagelser.

Der blev derfor skrevet en klasse `MyRecorder.h` som arver funktionalitet fra `sf::SoundRecorder` klassen. Der arves 3 funktioner herfra:

- `virtual bool onStart()`
- `virtual bool onProcessSamples(const sf::Int16*samples, std::size_t sampleCount);`

- `virtual void onStop();`

`onStart()` og `onStop()` bliver kørt når et optagelses objekt af `MyRecorder.h` klassen f.eks. `MyRecorder optag`, bliver startet med `optag.start()` eller sluttet med `optag.slut()`. Der kan så defineres start eller slut betingelser for en optagelse, såsom variable eller buffere der cleares. `onProcessSamples()` er en funktion der bliver kørt automatisk og gentaget i samme tråd som optagelsen, afhængigt af hvad for et interval der sættes under `onStart();` (100 millisekunder pr. standard). Den bliver indlæst med parameteren `const sf::Int16* samples`, dvs. en vektor med diskrete værdier for en optagelse, og hver gang funktionen kaldes er det de seneste nye værdier der ligger i vektoren. `onProcessSamples()` fortsættes med at blive kaldt hvis den returnerer true, og den stoppes hvis der returneres false.
Dvs. `MyRecorder.h` klassen kan optage og analysere data på én gang, hvilket løser problematikken om hvorvidt optageren skal vide hvornår den skal optage og afslutte en optagelse. Nu kan der f.eks. laves en starttone og en sluttone, som optageren kan opfange og derfra igangsætte bestemte kode dele.

Derudover blev klassen `Goertzel.h` skrevet som alternativ til en DFT udregning.
`Goertzel.h` klassen, som navnet antyder, benytter en Goertzel algoritme til at analysere vektoren med de diskrete værdier fra optagelsen. Jf. teori afsnittet, er Goertzel en smart måde hvorpå der kun udføres udregninger på de frekvens bins der ledes efter.
Dvs. at denne algoritme udfører langt færre udregninger i forhold til DFT og FFT(Fast Fourier Transform), netop fordi der ikke analyseres over alle samtlige DFT/FFT koefficienter. Dette gør at det ikke er noget problem at optage i længere.

Før Goertzel algoritmen kan implementeres er der nogle få ting der skal på plads først. Frekvens binsene skal regnes ud, jf. teori om frekvens bins, skal en samplingfrekvens vides og hvor mange samples der skal samples over. Samplingsfrekvensen er fastsat til 8000 Hz, dog skal det antal samples N som Goertzel analyserer over fastsættes. Der er dog nogle pointer ved fastsættelsen af N, jo større N bliver, desto større antal samples skal der udregnes over, men samtidig jo størrer N bliver, jo smallere bliver bredden af frekvens binen, dvs. at den bliver mere præcis. Det handler om at vælge en N, der ikke er for høj, da dette resulterer i tab af beregningshastighed, eller for lav, hvilket resulterer i tab af præcision.
Derudover må N ikke blive større end 400 samples, jf. teori om DTMF, da der er et tidsrum af 50 millisekunder mellem hver tone, grundet samplingsfrekvensen på 8000 Hz. Dette tidsrum er der for at forhindre, at en analyse af N samples fra en optagelse ikke foregår over et skift fra én tone til en anden.
Dette taget i betragtning blev N valgt til at være lig 300, eftersom det er tilpas højt for en tilfredsstillende bredde af frekvens binen (8000 Hz / 300 = 26,7 Hz, den må være helt op til 70 Hz før det skaber problemer), samtidig med det tidsrum som projektet faktisk har tilføjet mellem hver tone, kun er på 400 samples, dvs. 50 millisekunder.
Frekvens binen for hver DTMF frekvens er derfor udregnet, se tabel 3, og kan bruges til implemen-

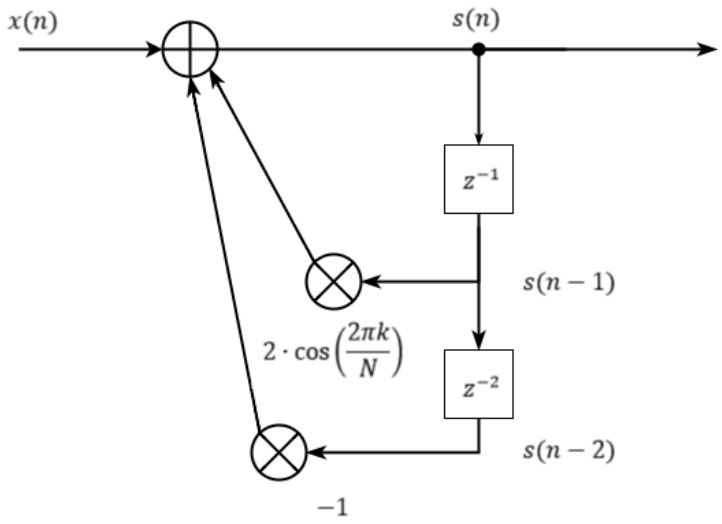
teringen af `Goertzel.h` klassen.

DTMF Frekvenser (Hz)	Frekvens bin
697	26
770	29
852	32
941	35
1209	45
1336	50
1477	55
1633	61

Figur 3: DTMF tabel

4.2 Implementering

Jf. teori om Goertzel vil filterets G(Z) realiserings struktur se således ud se figur 4. Denne realiserings struktur er blevet implementeret i `Goertzel` klassen ved hjælp af en for løkke, som kører en vektor af diskrete værdier, x(n), med N samples, igennem strukturens udregninger. Husk at ved $x(0)ers(-1) = s(-2) = 0$.



Figur 4: Goertzel realiserings struktur

Denne for løkke er blevet implementeret således:
Derefter kan DFT koefficienten bestemmes for en givet frekvens bin k, fordi nu kendes værdierne

```
for (int i = 0; i < N; i++)
{
    s = samples[i] + 2 * cos(omega) * prevS1 - prevS2;
    prevS2 = prevS1;
    prevS1 = s;
}
```

for sekvensen $s(n)$ ved $s(N-1)$ og $s(N-2)$ (prevS1 og prevS2):

$$|X(k)|^2 = s(N-1)^2 + s(N-2)^2 - 2 \times cos\left(\frac{2\pi k}{N}\right) s(N-1)s(N-2) \tag{8}$$

`Goertzel` klassens funktion er delt op i 2 metoder:

- `int detectFreqs(const sf::Int16* samples, int K);`
- `int findTone(const sf::Int16* samples);`

`detectFreqs()` bruger det ovenstående implementerede princip, hvorimod `findTone()` er en metode som bruger `detectFreqs()` til at gennemgå de tidligere nævnte frekvens bins og returnere en DTMF tone der er over en grænseværdi i forhold til DFT koefficienten. Tonerne er blevet defineret som integers fra 0 til 15, og en grænseværdi er nødvendig fordi en tilfældig støj der rammer en frekvens bin kan blive opfanget.
`MyRecorder` har seks metoder:

- `vector <int> getBesked();`
- `bool getNyBesked();`
- `bool getBeskedBegyndt();`

`getBesked()` er den metode der kaldes for at hente de DTMF toner, som blev optaget.
`getNyBesked()` og `getBeskedBegyndt()` er metoder til at kalde boolske udtryk, som bliver brugt til at manipulere og styre `MyRecorder`.

- `virtual bool onStart();`
- `virtual bool onProcessSamples(const sf::Int16* samples, std::size_t sampleCount);`
- `virtual bool onStop();`

`onStart()`, `onProcessSamples()` og `onStop()` fungere som tidligere nævnt, dog kaldes `findTones()` fra `Goertzel` klassen under hver `onProcessSamples()` kald, netop fordi der skal optages og analyseres samtidigt.

For at gøre det nemmere at skrive funktionaliteterne for klasserne `MyRecorder` og `Goertzel`, blev der taget udgangspunkt i sekvensdiagrammet SE BILLEDE BILAG WHATEVER.
`MyRecorder` bliver kaldt af `ToneKonvertering` i det, at det er den klasse som skal modtage

beskeden, som består af en vektor af toner, som derefter bliver pullet op igennem applikationens forskellige lag.
Når `ToneKonvertering` kalder `MyRecorder`, blev funktionaliteten for, hvorledes om en optagelse fanger en besked eller ej og om hvor lang en optagelse er, nødt til at blive implementeres under `ToneKonverterings` metoden `returnBitString()`.
Der er blevet implementeret en while løkke, som kører enten indtil der er gået 10 sekunder, eller at en sluttone på en besked blev opfanget vist i figur 5.

```
while (!optag.getNyBesked() && tid.asSeconds() < 10 )
{

    if (optag.getBeskedBegyndt() && !clockReset)
    {
        clock.restart();
        clockReset = true;
    }
    tid = clock.getElapsedTime();
}
```

Figur 5: While løkke

Der er en if sætning der tester for, om en starttone er blevet hørt, hvis den fanges, skal timeren tælle til 10 forfra. Herefter returneres en vektor med alle de optagede toner. Dvs. at optagelsen altid vil være maks. 10 sekunder lang, hvilket resten af applikationen tager højde for.
`MyRecorder` starter en optagelse med samplingsfrekvensen 8000 Hz i dens egen tråd, når metoden `start(8000)` bliver kaldt. `onProcessSampling()` kaldes hvert 10'ende millisekund i denne tråd. Under hvert kald bliver de optagede samples analyseret af Goertzel, som returnere tonen for de samples. Denne tone gemmes i `resusltatVektor`'en, som indeholder de toner, som hele optagelsen opfangede. Udover dette, sker der en mindre sortering af tonerne der optages under `onProcessSampling()` (dette fremgår ikke af sekvensdiagrammet). Tonerne skal ikke gemmes i `resusltatVektor`'en medmindre der først har været en starttone. Ved en sluttone stoppes `onProcessSampling()` for at blive kaldt igen, medmindre en helt ny optagelses session startes. Start og stop tonerne er blevet defineret som tone 15 (D) = start og tone 14 (#) = stop. Disse to toner bliver tilført som flag og taget højde for i de øvrige lag.

4.3 Diskussion

I projekts tidlige stadie var der en del usikkerhed omkring pålideligheden af `MyRecorder` og `Goertzel` klassens optagede besked, eftersom det virkede til at der var en tilfældig bouncing effekt mellem et skift af tone under optagelsen. Selv midt i en tone optagelse var der chance for et bounce. Ved en forholdsvis lav sendehastighed, 4 toner pr. sekund, var bouncing effekten tilstede, dog reduceret.
Dette var ikke et stort problem ved starten af projektet, eftersom der oprindeligt blev sendt med en hastig af 8 toner pr. sekund. De højttalere som blev udleveret med projektbeskrivelsen inducerer

mere af denne formodede bouncing, i forhold til nogle af projektets medlemmers private højtalere, som der blev testet på til at starte med.
Højttalernes kvalitet har derfor vist sig at have påvirkning på hvor et rent DTMF signal der laves.

Udviklingen af et filter til optagelse havde været en mulig løsning på højttalernes kvalitet, men i projektet blev der implementeret en debouncer i `MyRecorder` klassen, for at sikre der ikke sker fejl under behandlingen af optagelsen. Debounceren er en forholdsvis stor debouncer. Den tjekker for de to forrige værdier, og tjekker om signalet er færdig med at bounce. Dvs, at projektet kræver at en tone skal opfanges tre gange før den godtages. Denne debouncer gør, at kommunikation med DTMF i projektet bliver betydeligt langsommere, fordi en tone skal være til stede 3 gange så lang tid før tonen bliver accepteret. For projektet er sendehastigheden blevet langsommere, 4 toner pr. sekund.

Det har vist sig ved slutningen af projektet, at den bouncing effekt der opleves, højst sandsynligt skyldes en fejl i `Goertzel` klassens metode `findTone()`. En grænseværdi er ikke blevet initialiseret korrekt (der er ingen grænseværdi for en af sorterings løkkerne i metoden), men grundet tidspres er det en fejl der rettes i en senere udgivelse af applikationen, fordi en udvidet testfase skal udføres, og hele applikationen skal opdateres til at arbejde med de opnåede resultater fra testfasen.

4.4 Afspil

Projektet har den problemstilling, at der skal afspilles en forudbestemt sekvens af toner, i forhold til en besked.
Al funktionalitet som vedrører afspilning af DTMF toner er afhængig af SFML bibliotekets klasser `sf::Sound` og `sf::SoundBuffer`, eftersom C++ egets bibliotek ikke tilbyder afspilning af DTMF toner.

Før afspilning kan finde sted, skal der fastsættes en samplingsfrekvens, samt hvor mange samples pr. DTMF tone der er i en afspilning. Der kunne med fordel tages udgangspunkt i optage afsnittet for projektet, og benytte den samme samplingsfrekvens - 8000 Hz . Derudover skal afspil nødvendigvis afspille i den hastighed som applikationen kan optage med, dvs. 4 toner pr. sekund - så 2000 samples pr. tone.
De ønskede DTMF toner der skal afspilles (hvis det er rå data), skal indlæses i en vektor af diskrete værdier. Vektoren er defineret som :

```
vector<sf::Int16>raw1;
```

`raw1` er her vektoren med datatypen `signed Int16`. Dvs. at amplituden for en diskret værdi kan maks være af størrelsen $\frac{2^16}{2}$. Data'en for en DTMF tone kan med fordel laves ved hjælp af en for løkke, som lægger to sinusoidale funktioner sammen, bestående af de frekvenser som indgår i DTMF tonen. Se figur 6 for kode eksempel.

```
for (unsigned i = 0; i < SAMPLES; i++)
{
    raw1.push_back( AMPLITUDE*(sin(f1 * TWO_PI * i / SAMPLE_RATE)
        + sin(f2 * TWO_PI * i / SAMPLE_RATE)));
}
```

Figur 6: Kode eksempel

Denne vektor skal indlæses i et buffer objekt lavet af `sf::SoundBuffer` klassen. Bufferen skal også indlæses med en integer, for hvor mange samples der er i vektoren, og hvilken samplingsfrekvens afspilningen skal have. Derefter kan et `sf::Sound` objekt laves f.eks. `sf::Sound` lyd, dette objekt skal indlæses med bufferen, som indeholder den information som `sf::Sound` objektet skal afspille. Dette gøres med metoden `lyd.setBuffer(buffer)`. `sf::Sound` objektet kan nu afspilles med metoden `lyd.play()`, hvorefter applikationen sættes til at sove den tid afspilningen tager.

Projektets første løsning på at afspille en DTMF tone, var en simpel men utilstrækkelig løsning. Den brugte ovenstående metode til at afspille toner med, ved oprettelse af et objekt f.eks. tone(frekvens1, frekvens2), som blev kørt igennem ovenstående for løkke. `raw1` blev derefter afspillet. En sekvens af toner, var derfor en sekvens af tone objekter der først skulle oprettes, og derefter sendes til afspilning en af gangen. Hvis en tone indgik flere gange i en sekvens, så blev den oprettet igen hver gang. Første løsning skabte derfor følgende problemstillinger:

- Dette foregik i samme klasse, og skabte low cohesion eftersom det at afspille og generere toner er 2 forskellige ting.
- Det en langsom process, som var tydelig under projektets test stadie, og havde indflydelse på kvaliteten af en sekvens af afspilninger.
- Jf. teori om DTMF, mangler der en pause på 50 millisekunder mellem hver tone.
- Metoden skal have tone objekter som input, hvilket de øvre lag ikke kan tilbyde.

Der blev derfor skrevet to klasser til at løse problemstillingerne, `Afspil.h` og `Tone.h`. `Afspil` klassens ansvar er udelukkende at afspille data, og `Tone` klassen står for at oprette denne data.

4.4.1 Implementation

For at gøre det nemmere at implementere ovenstående funktionalitet blev der taget udgangspunkt i FIGUR SEKVENSDIAGRAM LORT.
Tone klassen opretter objekter som består af 2 frekvenser

- `Tone(int frekvens1, int frekvens2);`

Hvert objekt køres igennem den før nævnte sinusoidale for løkke, og opretter data for hvert tone objekt. Disse data kan så tilgås ved følgende metode som er blevet skrevet i `Tone` klassen:

- `vector<sf::Int16>getRaw();`

`Afspil` klassen har tre metoder:

- `sendData(vector<int>input)`
- `makeRaw0(vector<int>input)`
- `afspilToner(int længdeAfElementer)`

`sendData()` er en offentlig metode, som sørger for at sende en besked, ved at bruge `makeRaw0()` og `afspilToner()`. Inputtet fra de øvrige lag, er en sekvens af integers, som symboliserer toner.

`makeRaw0()` sørger for, at lave en ny vektor, `raw0`, med sekvenser af tone data. `makeRaw0()` slår op i vektoren `dtmfToner` i `Afspil` klassen når den får en besked den skal afspille, og kopier indholdet over i `raw0` det kan ses i 7. `dtmfToner` består af tone objekter og er sorteret fra 0 til 15,

```
raw0 = { 0 };

for (int k = 0; k < input.size(); k++)
{
    for (int i = 0; i < ((dtmfToner[input[k]]->getRaw()).size()); i++)
        raw0.push_back((dtmfToner[input[k]]->getRaw())[i]);

    for (size_t i = 0; i < 400; i++)
        raw0.push_back((dtmfToner[16]->getRaw())[i]);
```

Figur 7: Metode makeRaw0

som er de integers der symboliserer hver tone. Hver gang `makeRaw0` har kopieret en tones data til vektoren, så lægger den 400 samples af tone 16 oven i, (400 samples er 50 millisekunder), netop fordi der skal være en pause af 50 millisekunder mellem hver tone jf. teori om DTMF.
`afspilToner()` afspiller altid `raw0`, dog skal metoden vide hvor mange elementer der afspilles som input.

De tidligere nævnte problemstillinger er derfor løst. Der er to klasser med hver deres ansvar. Tone objekter bliver kun oprettet en gang, nemlig ved applikationens start, så det går betydeligt hurtigere at generere en afspilning. Der er blevet tilføjet de 50 millisekunders pause. De øvrige lag skal kun give en vektor af integers for at igangsætte en afspilning.

4.4.2 Disussion

Afspilning af DTMF toner var en af projektets første problemer der blev undersøgt. Der var ikke tænkt over hvordan applikationen optog endnu, og det var først efter at optage delen af applikationen kom på plads, at afspilnings delen blev revurderet. Derfor var en afspilning i starten af projektet indlæst med en samplingsfrekvens på 44100 Hz (cd kvalitet) uden videre tanke for, at det ville skabe lange udregningstider. Et alternativ, projektet kunne have brugt, var at have lydfiler for hver tone inkluderet i applikationen, disse lydfiler kunne afspilles med SFML i forhold til en besked.

5 Data Link Laget

Data Link Laget (DLL) blev indført for at opnå pålidelighed af modtagne beskeder. Derudover indeholder DLL en konverterings klasse, `ToneKonvertering`, hvis formål er at konvertere dataen til et format som det fysiske lag kan forstå. Dette kræver at dataen før `ToneKonvertering` går op i 3, da hver tone tilsvarer 3 bit.

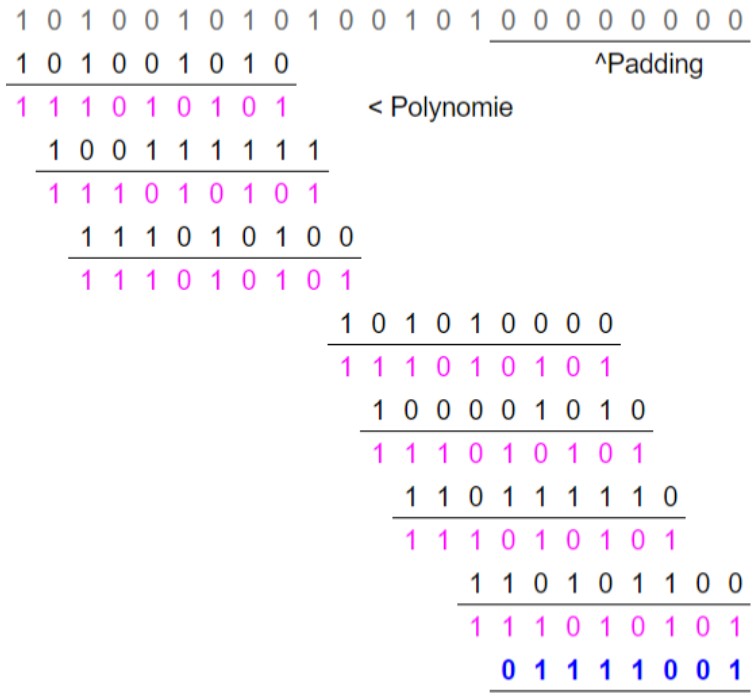
5.1 Teori

CRC står for Cyklisk Redundant Check, og bruges til fejldetektering. Formålet med fejldetektering er at gøre det muligt for modtageren at afgøre om et datagram, sendt gennem en støjet kanal, er blevet korrupt. For at gøre dette konstruerer senderen en checksum og tilføjer denne på datagrammet.
Modtageren er i stand til at bruge CRC til at beregne checksum af det modtagne datagram og sammenligne denne med den vedlagte checksum for at se, om datagrammet er blevet modtaget korrekt.
CRC er baseret på polynomiel aritmetik, som primært består af beregning af resten når et polynomium divideres med et andet polynomium. Et eksempel kan ses i figur 8.
Her er grå den originale besked, pink er polynomiet og blå er checksum.

5.2 Implementering

For at opnå pålidelig modtagelse af beskeder, blev et CRC indført. Dette gør at alle sendte beskeder skal indeholde et CRC, som bliver beregnet ud fra den originale besked.
Når denne besked modtages af en anden applikation, skal det samme CRC kunne udregnes og sammenlignes med det medfølgende CRC check. Hvis disse 2 checks er identiske, kan beskeden accepteres som korrekt.

For at oprette et CRC check, skal man først definere størrelsen af det CRC check man vil benytte. Originalt i applikationen var fejlsikring en stor prioritet, og beskeder blev sendt i en stor pakke, så et stort CRC-32 blev valgt, for at kunne finde flest mulige fejl. Dette kan udføres vha. polynomiel division.
For at benytte CRC-32, blev en tabel med 256 værdier oprettet på baggrund af en smart metode at implementere CRC checket. Dette tillader koden at lave den polynomielle aritmetik på forhånd,



Figur 8: Eksempel på polynomiel division

uden at vide hvilken bitstreng den modtager. Hver af disse værdier i tabellen tilsvarer CRC-32 checksummen for et 8 bits udtryk ($2^8 = 256$). Koden skal dermed hente dele af bitstrengen, og slå op i tabellen hvad hver eneste par af 8 bit tilsvarer. Dette udføres indtil der kun er 32 bits tilbage, og dette tilsvarer checksummen for beskeden. Denne løsning var præget af at skulle virke hurtigt, og var derfor limiteret af forskellige faktorer, såsom at CRC checken kun virker på udtryk der er længere end 32 bits. Derudover var den lavet overskuelig ved brug af hexadecimale tal, både i tabellen og i den resulterende checksum, som krævede ekstra konvertering tilbage til bitstreng.
Denne checksum bliver sendt sammen med beskeden, i en trailer, for at tillade modtageren at udføre samme CRC check, og verificere at beskeden er modtaget korrekt.
Denne checksum tilføjede 32 bits til datagrammet, og skabte en problemstilling i forhold til at sende dataen videre via `ToneKonvertering`, da det samlede datagram ikke gik op i 3 længere.
I `ToneKonvertering` skulle bitstrengen kunne opdeles i dele af 3 bits hver. Denne problemstilling bliver løst, ved at stuffe beskeden med ekstra bits. For at gøre det nemmere at fjerne stuffing, bliver der altid tilføjet stuffing, selv om beskeden originalt kunne opdeles. Ved at gøre dette slipper man uden om spørgsmålet, om der skal fjernes stuffing, da der altid skal fjernes stuffing. Stuffing bliver udført ved at tilføje 1 til 3 bits alt efter længden af beskedens bits. Mængden af stuffing, der skal pålægges beskeden, kan findes ved at tage modulus-3 af beskedens længde. Denne stuffing bestod af først et 1-tal, og dernæst 0'er indtil kravet om længden af stuffing er opfyldt. Dermed kan modtager siden identificere det 1-tal der forekommer først i traileren på datagrammet, og fjerne indtil denne.
Modtagersiden skulle dernæst kunne confirmere at beskedens checksum er korrekt, og derfor bliver datagrammer ved modtagelse splittet ud i besked data, og checksum. Den samme checksum skal

kunne udregnes fra den modtagne besked, ellers er der sket en fejl i transmission af beskeden.
Når transportlaget blev implementeret, blev meget mindre pakker sendt imellem applikationer, og CRC-32 checket viste sig at fylde mere end de sendte beskeder, og limitationer med længden af bitstrengen skabte ugyldige beskeder. Derfor blev CRC checket reduceret til et CRC-8. Dette skabte en kortere checksum, som dermed formindsker behovet for antallet af toner der skal sendes af applikationen.
Dette blev udført kodemæssigt med et 8 bits CRC check, som er en simpel XOR operation på beskeden. For at sikre håndtering af bits, snarere end integers, skal bitset dataholdere benyttes.
Det valgte polynomie er: $x8 + x7 + x6 + x4 + x2 + 1$, hvilket tilsvarer 111010101 i bits. Dette polynomie vil finde alle single-bit errors, og uneven-bit errors. Kodemæssigt en metode `runCRC()` implementeret, som ses på figur 9.

```
while (aug.size() > 8)
{
    std::bitset<9> remainder(std::string(aug, 0, 9));
    if (remainder[8] == 1)
    {
        tempRemainder = remainder ^ divisor;
        tempRemainderString = tempRemainder.to_string();
        tempAug = "";
        tempAug.append(aug, 9, aug.size());
        aug = tempRemainderString + tempAug;
    }
    else aug.erase(aug.begin());
}
```

Figur 9: `runCRC()`

`runCRC(string bitstreng)` er den offentlige metode, der udregner en checksum ud fra en bitstreng af vilkårlig længde, vha. polynomiel division. Dette er implementeret ved at hente 9 bits fra den paddede besked, her kendt som `aug` (augmented message). Der skal kun udføres polynomiel division hver gang at MSB er 1, ellers skal nullet slettes, med mindre at checksummen er opnået. Den polynomielle divison er implementeret som et XOR af polynomiet, her kendt som divisor. Dette resultat bliver lavet til en ny `aug` string, som består af resultatet fra den polynomielle divison, og resten af beskeden som endnu ikke er blevet behandlet. Dette gentages indtil checksummen er opnået, som i et CRC-8's tilfælde er 8 bits langt.

`addStuffing(string bitstring)` er den metode, der beregner længden af den besked med checksum, og dernæst tilføjer stuffing for at opnå kompatibilitet med `ToneKonvertering` klassens metoder. Tilsvarende er der en metode, `removeStuffing(string bitstring)`, som fjerner stuffing. Denne kigger på det bit længst til højre i beskeden, og sletter indtil den finder et 1-tal. Dette 1-tal signalerer begyndelsen af stuffingen, og når denne er fjernet består beskeden af data og en checksum. `addStuffing()` og `removeStuffing()` kan ses på figur 10.

removeStuffing()

```
int tempInt = bitstreng.size();
tempInt = tempInt % 3;

if (tempInt == 0)
    stuffing = "100";
if (tempInt == 1)
    stuffing = "10";
if (tempInt == 2)
    stuffing = "1";
```

addStuffing()

```
while (true)
{
    if (bitstreng.back() == '0')
        bitstreng.pop_back();

    if (bitstreng.back() == '1')
    {
        bitstreng.pop_back();
        break;
    }
```

Figur 10: `removeStuffing()` og `addStuffing()`

`ToneKonverterings` klassen fungerer som en basal oversætter, der transformerer datagrammet fra en bitstreng til en sekvens af lyde, som det fysiske lag kan behandle.
Der blev stødt ind i en problemstilling, der omhandlede hvor mange ens toner, der blev modtaget i træk. For at tage hensyn til dette, blev der defineret et flag. Dette flag bliver indsat i mellem to ens toner. Dette medførte at optager klassen, `MyRecorder`, ikke behøver synkronisering, men i stedet er i stand til at finde, når en ny tone modtages. Derudover kræver konverteringen i denne klasse, at antallet af bits kan gå op i 3. Dette tager stuffingen i DLL hensyn til.

Der ses på 3 bits, som hver oversættes i forhold til deres værdi i bits, til en bestemt tone.
Tone 0 defineres som bitværdi 000, mens tone 7 defineres som bitværdi 111. Dette bliver gemt i en vektor, som indeholder værdier der tilsvarer toner.
Da der der ikke kan vides hvor mange toner der kommer af gangen, defineres 8 (DTMF: 7) som en tone flag. Hvis der er behov for at afspille to ens toner afspilles efter hinanden, bliver dette tone flag benyttet.
Til det fysiske lag defineres et start flag og et slut flag, som henholdsvis tilsvarer tone 14(DTMF: #) og tone 15(DTMF: d), da dette gør det nemmere for `MyRecoder` klassen at optage.
Al denne information indsættes i en vektor, som videre benyttes af andre klasser.

6 Transportlaget

For at sikre at der pålideligt kan blive sendt beskeder mellem to applikationer er der brug for error control, hvilket normalt ligger hos transportlaget; Derfor oprettes et transportlag, selvom at det brugte system er et point-to-point netværk.

6.1 Teori

For at oprette et transportlag benyttes elementer fra stop-and-wait protokollen. Der benyttes både sender-og modtager buffere, samt inddeling af data i segmenter. Stop and wait er en connection-

orienteret service, så den kan inddeles i tre dele: Oprettelse af forbindelse, data transfer, og nedbrydelse af forbindelse. Fordelene ved at bruge denne protokol er åbenlyst, når man arbejder med et point-to-point netværk hvor der ikke kan sendes data begge veje samtidig.

6.1.1 Oprettelse af forbindelse

Når der skal oprettes en forbindelse med stop-and-wait, skal senderen være aktivt åben. Den sender så en probe besked der indeholder information om resten af pakken, uden at indeholde data. Hvis modtageren skal kunne modtage skal den være passivt åben. Når den modtager en besked bliver den så aktivt åben, da den nu ved at der er nogen der forsøger at sende en pakke til den. Ved at sende et ACK tilbage skabes der således en forbindelse, da de nu begge ved at den anden er klar.

6.2 Data transfer

Efter forbindelsen er blevet oprettet, er det således muligt at sende data. Dette gøres ved inddele dataen i segmenter, der bliver nummerert. Dette sekvensnummer er en del af headeren, så når der modtages et segment er der ikke tvivl om det er det rigtige der bliver modtaget. Selv om segmenterne kun bliver sendt i rækkefølge er det stadig smart at give dem sekvensnumre, så det er muligt at prædefinerer hvor mange segmenter der bliver sendt, og kontrollere at de alle bliver modtaget.

6.2.1 Nedbrydelse af forbindelse

Når alt dataen er blevet sendt skal forbindelsen nedbrydes. Dette gøres ved et three-way-handshake, hvor den der lukke forbindelsen sender en besked med FIN flag, og sætter sig selv til aktivt lukket. Den anden sender et ACK tilbage på det, samtidig med at den bliver passivt lukket. Til sidst sender den der lukkede forbindelsen et sidste ACK, der fortæller at den modtog det andet handshake, hvorefter den lukker forbindelsen. Når det sidste ACK ankommer ved den anden lukker den forbindelsen. Hvis det sidste ACK bliver tabt lukker den selv forbindelsen når der er gået nok tid.

6.3 Implementation

Stop-and-wait er blevet implementeret fordi den tilbyder services der er meget brugbare i et point-to-point system hvor der ikke kan sendes og modtages samtidig.

6.3.1 Header

Først defineres headeren. Da der bruges nummerede segmenter indeholder headeren et sekvens nummer, i 8 bit, som samtidig benyttes som ACK på vej tilbage. Da der ikke bruges piggy-backing er der ikke nogen grund til at sende både et sekvens nummer og et ACK på samme tid, hvilket sparer plads i headeren. Ved at benytte en byte til segment nummering er det muligt at sende 255 segmenter, foruden den første, hvilket på dette tidspunkt vurderes at være rigeligt; Af denne årsag

benyttes der ikke cirkulære buffere, men blot vektorer der kan indeholde samtlige beskeder, der bliver sendt og modtaget. Samtidig sendes der tre flag:

- FIN - et slut flag der sendes i det sidste segment.
- Accept - et accept flag der fortæller senderen at forbindelsen er godkendt
- Probe - et flag der fortæller modtageren at der bliver forsøgt at oprette en forbindelse

Strengt taget er disse flag ikke nødvendige i denne iteration, men de er inkluderet fordi de er nødvendige hvis applikationen skal udbygges til at kunde sende beskeder der har brug for mere end 255 segmenter. På nuværende tidspunkt er det klart at den første besked er proben, da denne altid har segment nummer 0, men hvis dette sekvents nummer skal kunne genbruges er der brug for et flag. FIN flaget er i samme situation, vi ved på nuværende tidspunkt at modtageren er i stand til at vide hvilket segment der er det sidste, men hvis der udviddes til en cirkulær buffer ved den ikke om det segment der er sat, som det sidste virkeligt er den sidste. Accept flaget er brugt for at reserverer muligheden til at afvise beskeder, hvis for eksempel der bliver bedt om at sende flere segmenter end modtageren kan klare.

6.3.2 Segment størrelse

Fordi der er en betydelig sandsynlighed for at pakker kan blive påvirket af støj er det meget brugbart at implementerer segmenter af en lille størrelse, så der ikke bliver spildt store pakker hvis et enkelt byte bliver påvirket. Dette påvirker sendehastigheden, så pakkerne skal heller ikke være for små, da der så skal sendes mange headere fra de forskellige lag, samtidig med at hver segment der bliver sendt afsted kræver et ACK tilbage. Derfor vælges en pakke længde på ti bytes, da headeren er på 11 bit, hvilket vil sige at der bliver sendt ca. 10 gange så meget data som headeren, hvis man ser bort fra headeren fra de andre lag.

6.3.3 Forbindelse

For at kunne oprette forbindelse mellem to applikationer skal der være en sender og en modtager. Senderen bliver, per teorien om oprettelse af forbindelse, sat til aktivt åben, mens der skal være en modtager der står som passivt åbent. Hvis dette er tifældet vil der blive sendt et ACK tilbage, og forbindelsen vil være åben. Da der ikke er brug for at blive sendt andet information end antallet af segmenter, samt at modtageren accepterer forbindelsen, er der ikke brug for et three-way-handshake, da piggy-backing ikke er blevet implementeret, fordi det blev vurderet at det ikke var den kompleksitet og tid værd det ville kræve at implementerer threads. Der bliver ikke sendt data med her, da det blev besluttet at det var vigtigere at senderen ikke stod og sendte unødvendig data hvis der ikke var en modtager.
Når forbindelsen er oprettet sendes segmenterne et ad gangen, mens ACK med samme nummering sendes tilbage. Normalt sættes ACK til at være det næste forventede byte, men da der benyttes faste

segment størrelser er det besluttet at det var mere praktisk at sætte dem til samme numre, da dette var en simplere løsning at implementerer i starten og slutningen af beskeden.
Når beskeden er sendt skal forbindelsen lukkes ned. For ikke at skulle ud i at sende tomme beskeder andre tidspunkter end i starten, bliver slut flaget sat i et segment der indeholder den sidste data, og senderen sættes til aktivt lukket. Dette segment er samtidig det eneste der kan have mindre end 10 bytes med ud over headers. Når modtageren modtager denne besked sættes den til passivt lukket, da den ved at der ikke kommer mere. Den sender dog et ACK tilbage, for at fortælle senderen at den har modtaget beskeden. Hvis senderen ikke modtager dette ACK, sender den segmentet igen, men hvis den modtager dette ACK lukker den forbindelsen helt. Modtageren vil dog forsøge at sende ACK flere gange, i forsøg på at sikre at senderen ved at beskeden er modtaget. Der blev valgt ikke at sende det sidste segment fra et three-way-handshake fordi det vigtigste er at modtageren får hele beskeden, og hvis dette handshake fejler vil et three-way-handshake også fejle. Havde der været tale om en duplex forbindelse ville situationen være anderledes, men med dette system er et two-way-handshake vurderet at være en god nok løsning. Når modtageren har sendt sit ACK nok gange vil den lukke sin forbindelse ned automatisk, og give beskeden videre til applikations laget.

6.4 Diskussion

Den måde transportlaget er implementeret giver error control, hvilket betyder at vi pålideligt kan sende date mellem to processer. Da der ikke arbejdes i et medium der er i stand til at sende fuld duplex, og hvor pakker ikke kan ankomme ude af rækkefølge, er der ikke brug for at implementerer en advanceret protokol som TCP, hvilket originalt var planen. Efter nogle overvejelser over hvad der nødvendigt at implementerer endte det med at den brugte protokol lignede stop-and-wait mere og mere. Til sidst blev TCP droppet fuldstændigt.
Segmenternes størrelse kunne være optimeret på, man da hastighed ikke var en prioritet faldt valget på små størrelser, da disse er mere sikre overfor støj, da en enkelt forkert tone kun betyder at nogle få toner skal gensendes. Samtidig betyder den brugte segment størrelse at den længste besked der kan sendes i en bid er på 2550 tegn, hvilket svarer til en hel A4 side. Det blev vurderet at dette var nok til de mål der var blevet sat til projectet.
Der kan argumenteres for at der skulle bruges fuldt three-way-handshake til at lukke forbindelsen, men fordi dette kun gav senderen mulighed for at vide om senderen vidste om beskeden var modtaget eller ej blev det valgt at dette ikke for vigtigt. Hvis det andet handshake fejler ved senderen jo alligevel ikke om beskeden er modtaget eller ej, og da der ikke bliver sendt data den anden vej er der ikke brug for et vindue hvor modtageren kan gøre sin besked færdig. I fremtidige itterationer kunne dette implementeres, men med denne var det ikke en prioritet at have kendskab til den andens status, men blot at sende beskeden.

Formålet med at lave et transportlag var at opdele beskeder i segmenter, således at hele beskeden ikke går tabt hvis der forekommer støj, samt at lave error control, så det er forsikret at applikationen pålideligt kan sende beskeder.
Dette er opnået ved at bruge de fordele der er ved stop-and-wait protokollen. Grundet den

connection-orienterede natur er det muligt at sende data i en strøm hvor modtageren kan sikre sig at den data der bliver sendt er komplet og i rækkefølge, og at den er resistent overfor ikke-permanent støj, hvilket ville gøre at det ikke var muligt at sende pålideligt.

7 Applikationslaget

Applikationslagets formål er at skabe en platform for en bruger, så der ikke bemærkes hvad der foregår i de underliggende lag, men skaber en illusion om en direkte forbindelse mellem to applikationslag.
I dette afsnit gives teorien bag persistens og beskrivelse af de klasser som bruges i applikationslaget. Desuden er der en gennemgang af appilkationens user interface.

7.1 Teori om persistens

Persistens er en problemstilling, der skal tages højde for, når man gerne vil bruge data mellem forskellige instanser af en applikation. For at kalde din data persistent skal man også kunne få adgang til dataen, selvom at applikationen har været lukket ned i mellemtiden. For at kunne lave dataen persistent skal man først bestemme sig for, hvordan man vil gemme sin data. Dette kan gøres på forskellige måder, så som at gemme det i filer eller gemme det på en database. For at kunne aflæse dataen nemt, skal der opstilles retningslinjer for din data. Dette kunne for eksempel være, at hver gang der gemmes noget nyt i en fil, skal der være på en ny linje[2].

7.2 Controller klassen

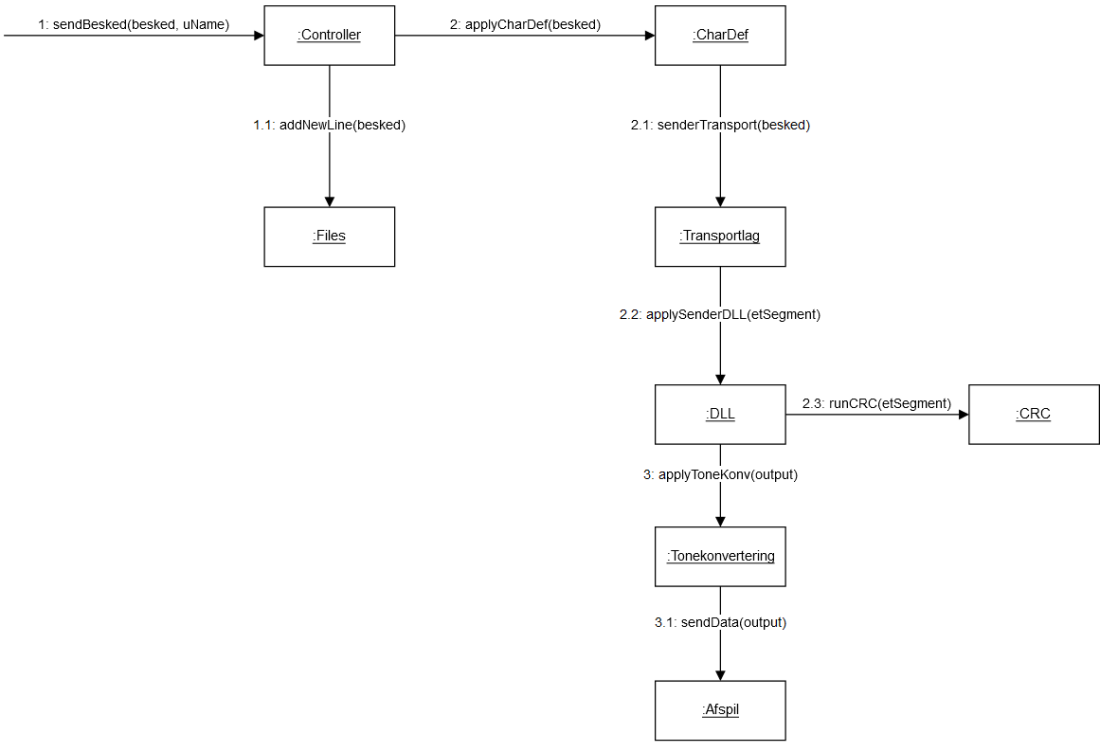
Der blev valgt at samle programmets vigtigste funktioner i klassen `Controller`. Dette blev gjort for at user interface kun skulle i kontakt med én klasse og fordi det ville blive nemmere i en senere iteration at implementere et Grafisk User Interface (GUI).

7.2.1 Afsending af besked

En af controller klassens funktioner er at sende en besked. Denne funktionalitet har fået navnet `sendBesked(besked, uName)`. Samarbejdsdiagrammet for `sendBesked()` er vist i figur 11
`sendBesked()` tager beskeden der skal sendes, samt navnet på den der har sendt den, og samler det i en besked. Denne besked bliver sendt videre til `Chardefinition` klassen, som laver teksten om til en binær streng. Den bliver efterfølgende sendt til transportlaget, der kan dele beskeden op i mindre segmenter, der sørger for at beskederne sendes pålideligt. Pakkerne kommer videre til Data Link Laget, hvor der vil blive tilføjet CRC og stuffing til bitstrengen. I `Tonekonvertering` bliver den binære streng omdannet til tal mellem 0 og 15, som svare til de toner vi har til rådighed, disse toner hedder DTMF. Til sidst bliver tonerne afspillet med klassen `Afspil`. Når en besked er sendt, bliver den gemt i en fil med klassen `Files`. Hvis der under dette forløb sker en fejl, bliver der sendt en fejlbesked retur til controlleren.

7.2.2 Modtagelse af besked

`Controller` klassen har desuen en metode til at modtage beskeder. Denne metode hedder `modtagBesked(uName)`. Samarbejdsdiagrammet for `modtagBesked()` er vist i figur 12
`modtagBesked()` går igennem de samme klasser som `sendBesked()`, dog i stedet for at sende

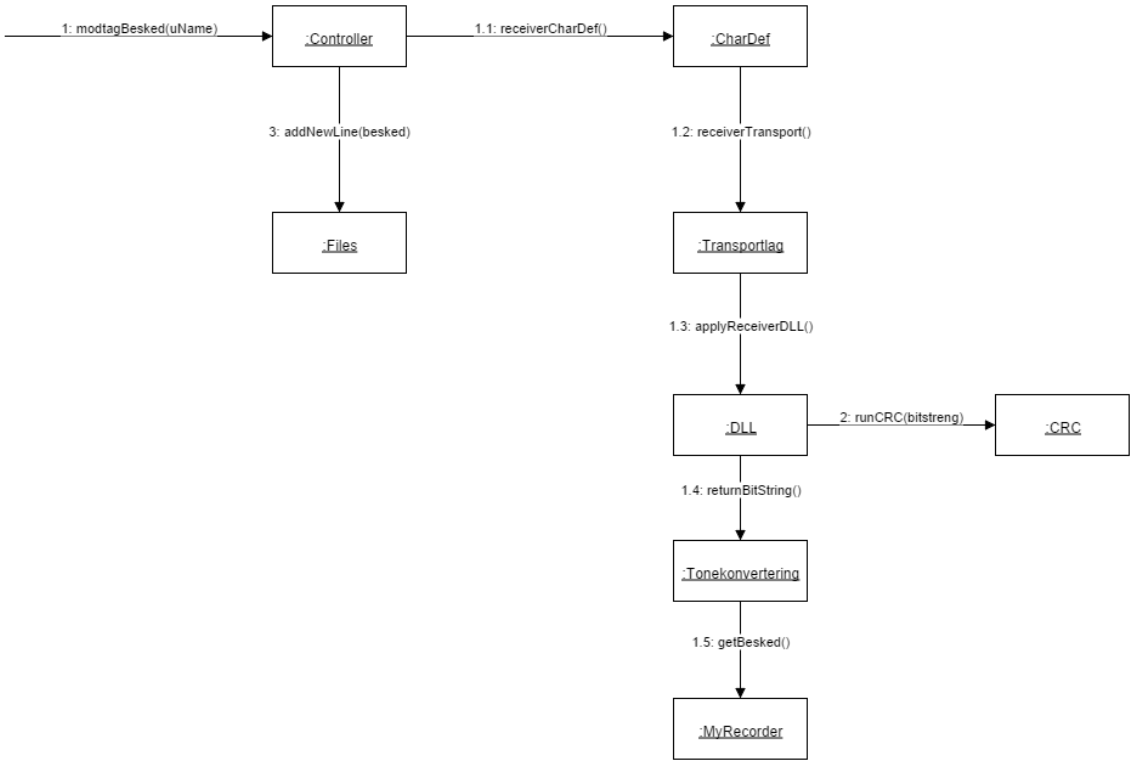


Figur 11: Samarbejdsdiagram for `sendBesked()`

en besked afsted, sendes der en request om at modtage en besked. I klassen `MyRecorder` modtages beskeden, som returnere en vektor af tal, der repræsenterer en tone mellem 0 og 15, til `Tonekonverteringen`. Her omdannes tonerne til en bitstreng, som returneres til `DLL` klassen. Ved DLL tjekkes for CRC og stuffing fjernes inden det returneres til Transportlaget, hvor beskeden sammensættes. Den kommer retur til `CharDef` og bliver lavet fra binær streng til karakterer som returneres til `Controller`, der ved hjælp af UI viser beskeden på skærmen. Samtidig bruges Files til at gemme beskeden i en historik. Til det bruges brugernavnet i `sendBesked()` for at gemme i den rigtige historik.

7.2.3 Øvrige metoder i controller

`testLogin(uName, pWord)` er metoden der kan teste om et brugernavn og password er korrekt. Det gør den ved at bruge `Login` klassen.
`createUser(uName, pWord)` bruger igen Login klassen, denne gang til at oprette en bruger. Der er også metoder der kan vise historikken. Disse metoder bruger alle funktioner fra klassen `Files` og kan enten vise hele historikken, sidste besked eller en der kan definere hvor mange linjer der skal hentes og vises.



Figur 12: Samarbejdsdiagram for `modtagBesked()`

7.3 Login klassen

Klassen `Login` bruges til at oprette og teste en bruger, som har et brugernavn og password. Metoden `addUser(uName, Pword)` opretter nye brugere og gemmer brugernavnet og passwordet i et txt dokument for at opnå persistens. Hvis der allerede er et brugernavn af denne type, fås en besked om at brugeren er taget.
Klassen har også metoden `testLogin(uName, pWord)`, som bruger metoden `validateLogin(uName, pWord)`. Den modtager et brugernavn og password, som bliver lavet om til et login token. Denne token bliver efterfølgende sammenlignet med alle de tokens der er gemt i txt filen. Hvis der findes et match bliver der returneret true og brugeren logges på.

7.4 Files klassen

`Files` klassen bruges til at gemme og tilgå historikken, når der bliver skrevet til hinanden med chat programmet. `Files` virker ved at når der oprettes et objekt at klassen, med en parameter, som er brugernavnet, bliver der oprettet et tekstdokument med brugerens navn, hvis dette ikke findes, som historikken kan gemmes i. Herefter bruges følgende metoder til at bearbejde historikken:

- `addNewLine(besked)` tilføjer en besked til tekstdokumentet, for at gøre det persistent.
- `updateVector()` bruges til at lave en vektor af linjerne fra historikken, som efterfølgende kan bearbejdes.

- `printVector()` kan printe hver linje ud, som er i vektoren skabt med `updateVector()`.
- `clearText()` kan slette alt fra historikken.
- `printLatest()` printes den sidste besked.
- `printLines(startN, endM)` kan printe de valgte linjer ud fra linje n til m.
- `flipVector()` bruge til vende vectoren, så f.eks. den sidste modtagne besked kommer til at ligge øverst i vektoren, altså på plads 0.

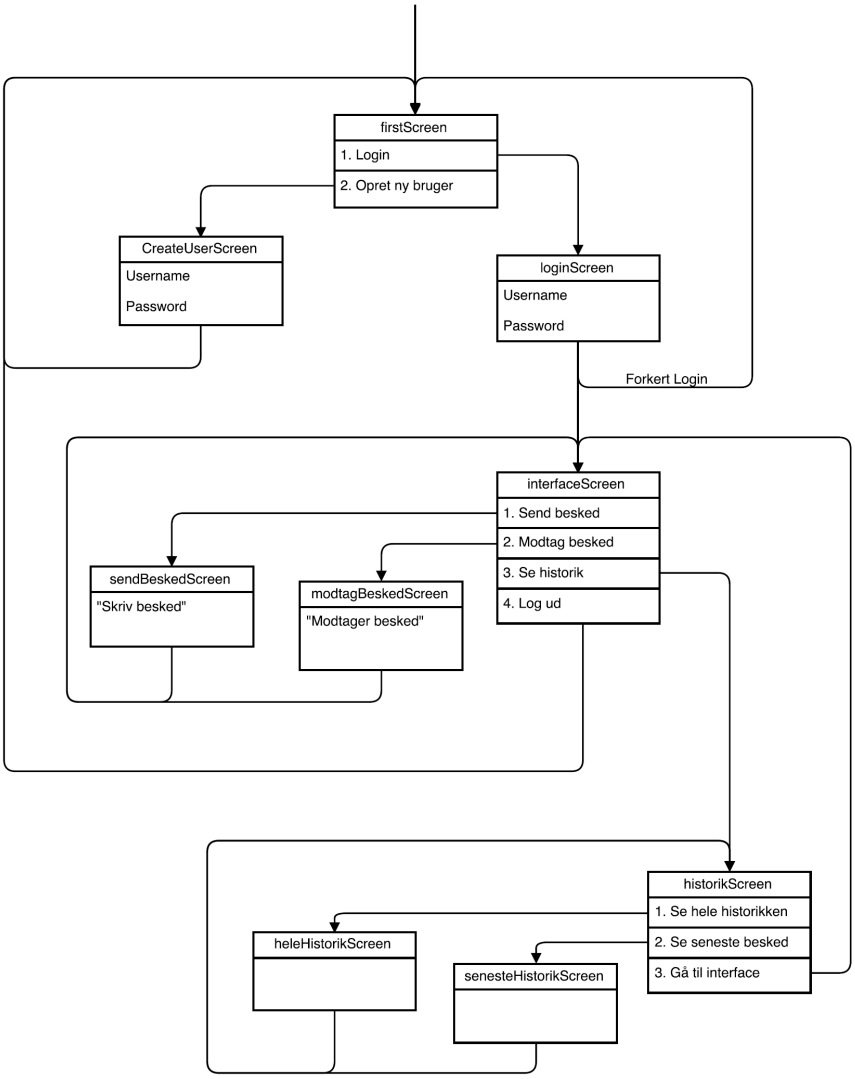
7.5 CharDefinition klassen

`CharDefinition` klassens opgave er at omdanne beskeder til en binær repræsentation som det underlæggende Transportlag kan forstå, og omvendt. Når der oprettes et objekt af klassen, bliver der lavet en liste over alle de tal, karakterer og andre tegn, der er mulighed for at sende.
Klassen indeholder metoden `charToBinary(input)` som omdanner indputtet, som er en string af karakterer, til en binær string. Det gør den ved at sammenligne karaktererne, der skal sendes, med dem i listen. Når der findes et match, bruges nummeret på placeringen af karakteren i listen, til at udregne en binær værdi, der bliver repræsenteret, ved hjælp af 8 bits. F.eks. vil karakteren b, som har placeringen 11, få den binære værdi 00001011.
`applyCharDef(input)` anvender denne metode og sørger for at den binære repæsentation sendes videre til `Tranportlaget`.
`binaryToChar(messageReceived)` omdanner hvert segment af otte bits om til et decimaltal, der bruges i listen til at finde de modtagne karaktere. Karaktererne sættes på en string indtil hele beskeden er omdannet.
`receiverCharDef()` bruger denne metode, når en besked returneres fra `Transportlaget`. Hvis beskeden der modtages er en fejlbesked sendes den videre og der bruges ikke `binaryToChar()`.

7.6 UI source

Til chat programmet er der valgt at lave et simpelt user interface. Dette blev lavet ved at bruge konsolvinduet i Visual Studio. Et flowdiagram af systemet er vist på figur 13. Applikationens UI er bygget op ved at der gives nogle valgmuligheder. På første skærm kan der vælges mellem login eller opret bruger. Der vælges ved at indtaste et tal, her 1 eller 2. Tallet sammenlignes med de muligheder der er, og applikationen går til næste skærm.
Når der oprettes ny bruger, indtastes brugernavn og password som efterfølgende bruges i `Controller` klassen i metoden `createUser(uName, pWord)`. Der vil efterfølgende gås videre til loginskærmen, hvor brugernavn og password indtastes igen og bruges i metoden `testLogin(uName, pWord)`. Hvis det indtastede er forkert kan der prøves igen ellers kommer interface skærmen.
På interface skærmen er mulighederne:

- Send besked
- Modtag besked



Figur 13: Flowdiagram af UI

- Se historik
- Log ud

Ved "Send besked"skærmen kan der skrives en besked, som efterfølgende sendes med metoden `sendBesked(besked, uName)` fra `Controller`. Ved "Modtag besked"skærmen afventes der en besked. Denne modtages med metoden `modtagBesked()` og bliver efterfølgende vist på skærmen. Hvis der vælges historik, kommer en skærm med 3 muligheder igen. Her kan der vælges at se hele historikken eller seneste besked som vises med metoderne `getHeleHistory()` og `getSenesteHistory()` fra `Controller`. Den ved den sidste valgmulighed sendes man retur til interface skærmen, dette kan ses i figur 14. For at kunne holde styre vores UI har vi valgt at bruge kommandoen GOTO som kan springe i mellem labels. For at vælge en af de givne valgmuligheder har vi brugt if sætninger til analysere inputtet fra kommandoen cin

```
historikScreen:

    std::cout << "V\x91lg en operation" << std::endl;
    std::cout << "1. Se hele historikken" << std::endl;
    std::cout << "2. Se seneste besked" << std::endl;
    std::cout << "3. G\x86 tilbage til interface" << std::endl;
    std::cin >> tempValg;

    if (tempValg == "1")
    {
        std::system("cls");
        goto heleHistorikScreen;
    }
```

Figur 14: Label historikScreen

8 Konklusion

Vi har benyttet Scrum, hvilket har "tvunget"gruppen til at kommunikere med hinanden, så projektet derfor er blevet mere gruppe-orienteret.

9 Perspektivering

I dette projekt har vi forsøgt at løse fysiske og kodemæssige problemstillinger, med en agile fremgangsmåde. Omend dette ikke altid er lykkedes, har vi lært hvilke faldgrupper og brugbare processer, der kan bruges som udgangspunkt i fremtidige software opgaver og projekter.
De problemer vi er stødt på, som kan optimeres, er blandt andet vores brug af Goertzel algoritmen, som ikke bruger thresholds korrekt. Dette har givet problemer med støj, som videre har givet problemer med hastighed af besked afsendelse/modtagelse. Yderligere ville den næste iteration involvere refakturering af de mere komplekse dele af koden, deriblandt transport laget's modtagerside.
Den næste iteration ville formodentlig indholde tråde, hvilke ville give nye problemstillinger og muligheder. Bl.a. ville vi være i stand til at lave dublex kommunikation, hvilket ville betyde at transportlaget skulle ændres til at include piggy-backing.
Brugen af Scrum har haft en væsentlig indflydelse på den måde gruppemedlemmerne har kommunikeret med hinanden på, men samtidig har vi indset at det ikke er nok til at arbejde agilt og at et undervisende projekt af folk, der ikke er vant til at arbejde sammen som en gruppe, da agilt arbejde fungerer bedst når der arbejdes med noget, mindst én i gruppen har kompetancer til. Desuden fungerer agilt arbejde bedst i grupper hvor alle kender deres rolle.

10 Litteraturliste

10.1 Artikler

[1] Bhavannam: Midasala, "DTMF Tone Generation and Detection Using Goertzel Algorithm with MATLAB", I: Proceedings of International Conference on Innovation in Electronics and Communications Engineering, 2013

10.2 Bøger

[2] Ashrafi: Noushin: Hessam. "*Object Orient System Analysis and Design."Prentice Hall, 2008, Chapter 13*

*[3] Tan, Li: Jiang, Jean, "*Digital Signal Processeing.", 2. edition, Elsvier, 2013

[4] Larman, Craig, "*Applying UML and Patterns.", Second edition, Prentice Hall, 2008*

*[5] Forouzan, Behrouz A., "*Data Communications and Networking.", Fifth edition, McGraw-Hill, 2013

10.3 Hjemmesider

[6] Kowalk, W.: "CRC Cyclic Redundancy Check Analysing and Correcting Errors". På http://www.uni-oldenburg.de, http://einstein.informatik.uni-oldenburg.de/papers/CRC-BitfilterEng.pdf Publiceret 08/2006. Besøgt 15/12/2015

[7] Ross, Williams N.: "A Painless Giode To CRC Error Detection Algorithms". På ross.net, http://www.ross.net/crc/download/crc_v3.txt, Publiceret 19/8/1993. Besøgt 9/11/2015.

[8] Koopman, Philip: Chakrabarty, Tridib,: "Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks". På http://repository.cmu.edu/, http://repository.cmu.edu/cgi/viewcontent.cgi?article=1672&context=isr, Publiceret 2004. Besøgt 11/12/2015.

[9] Barr, Michael,: "CRC Series, Part 3: CRC Implementation Code in C/C++"På barrgroup.com http://www.barrgroup.com/Embedded-Systems/How-To/CRC-Calculation-C-Code Publiceret 02/12/2007. Besøgt 11/12/2015.