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

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:

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

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.

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

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:

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

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

Filterets andet stadie er et FIR filter:

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

I en kaskade har filterets overføringsfunktion udseendet:

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

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

er derfor givet således:

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

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:

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

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.

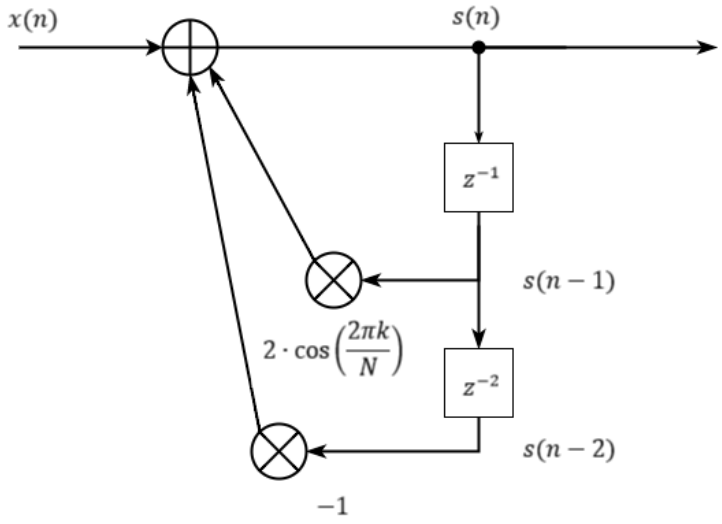
Frekvens binen for hver DTMF frekvens er derfor udregnet, se tabel 3, og kan bruges til implementeringen af `Goertzel.h` klassen.

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):

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

`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. 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.

5 Data Link Laget

5.1 Teori

CRC

CRC står for Cyklisk Redundant Check, og bruges til fejl-detektering. Formålet med fejl-detektering er at gøre det muligt for modtageren at afgøre om et datagram, sendt gennem en støjet kanal, er blevet beskadiget. For at gøre dette konstruer senderen en checksum og føjer denne til 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.[4]
CRC er baseret på polynomiel aritmetik, primært beregning af resten når et polynomium divideres med et andet. Addition og subtraktion udføres ved hjælp af modulo-2.

Modulo-2 Binary Division

Modulo er resten af en division mellem to tal, og er oftest udtrykt som "%".
I modulo defineres nye oprationer, som ofte ligner Boolske logiske operationer, heriblandt XOR, som bruges til addition, og AND, som bruges til mulitplikation. Bitvist er er modulo-2 det samme

som XOR, og her noteres modulo-2 additions operationen med en cirkel med et plus, f.eks.:

$$1 \oplus 0 = 1$$

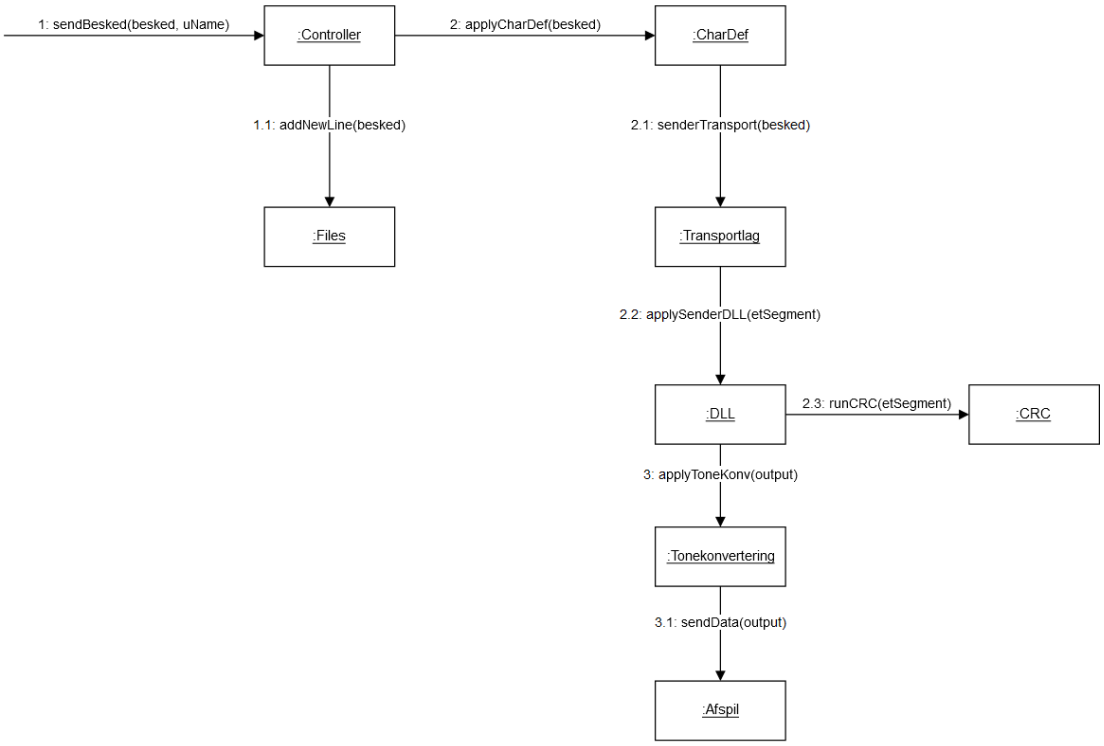
Stuffing

Bitstuffing er nødvendig for datagrammer der ikke overholder de størrelsesmæssige krav. I dette tilfælde tilføres ekstra bits, uden betydning, til datagrammet, indtil dette opfylder de nødvendige størrelsesmæssige krav.

6 Kontrolfunktioner

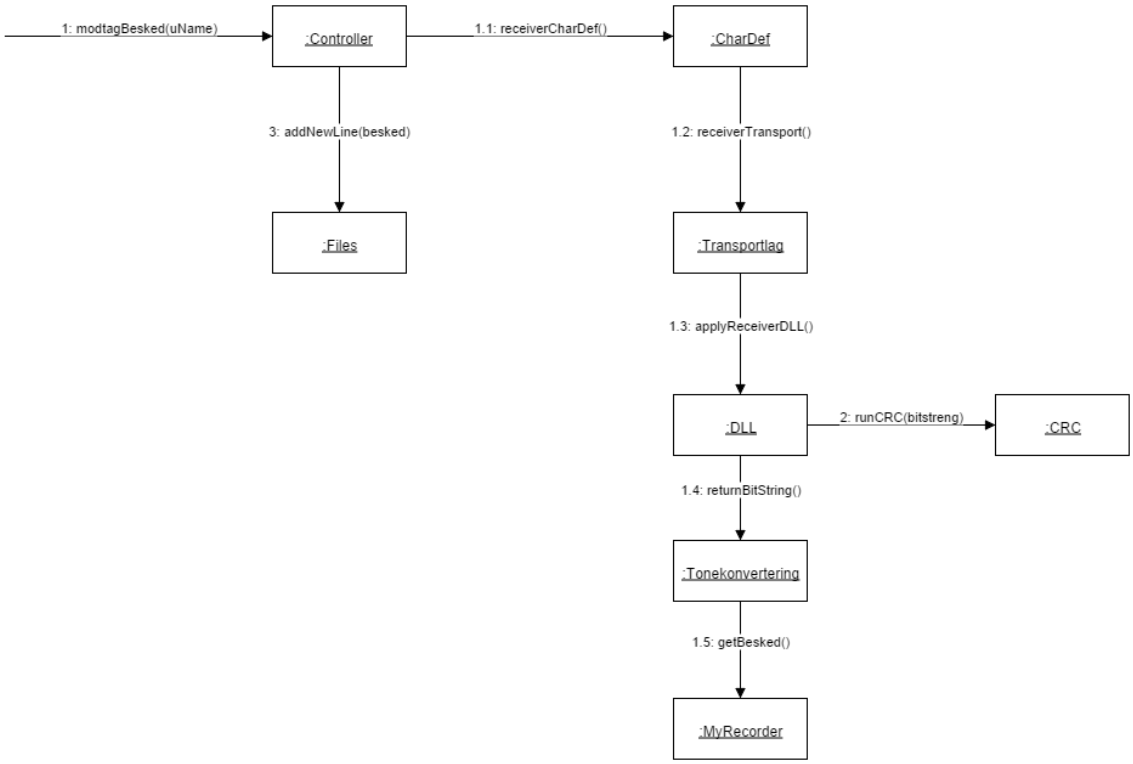
Controller klassen

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).
En af `Controller` klassens metoder er `sendBesked(besked, uName)`. Samarbejdsdiagrammet for `sendBesked()` er vist i figur 5.



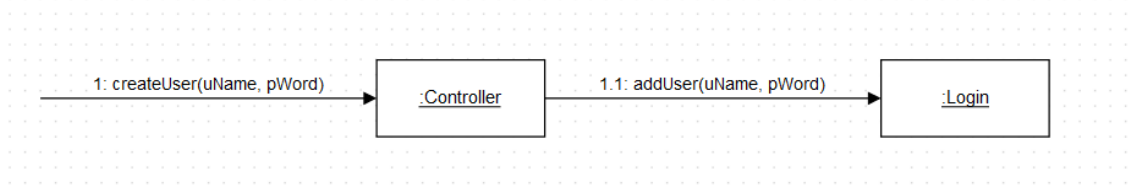
Figur 5: `sendBesked()`

Denne metode 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 og sørge for at sendingerne foregår som de skal. Pakkerne vil kommere 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 er det antal toner der er til rådighed ved DTMF. Til sidst bliver tonerne afspillet med klassen `Afspil`. Når en besked er sendt, bliver den gemt i en fil med klassen `Files`.
Klassen har desuden en metode, der hedder `modtagBesked(uName)`. Samarbejdsdiagrammet for `modtagBesked()` er vist i figur 6.
Denne går igennem de samme klasser som `sendBesked`, dog i stedet for at sende en besked afsted, sendes der en request om at modtage en besked. I klassen `MyRecorder` modtages beskeden, som



Figur 6: `modtagBesked()`

returneres som toner af tal mellem 0 og 15 til `Tonekonvertering`'en, der omdanner tonerne til en bitstreng, som returneres til Data link laget. Ved `DLL` tjekkes for CRC og stuffing fjernes inden det returneres til transportlaget, hvor beskeden sættes sammen igen, hvis beskeden var delt op i segmenter. Den kommer retur til `CharDef` og bliver lavet fra binær streng til karakterer, som derefter returneres til `Controller`, der viser beskeden på skærmen og samtidig bruger `Files` til at gemme beskeden i en historik. Til dette bruges `uName` i metoden for at gemme i den rigtige historik.
`testLogin(uName, pWord)` er metoden der kan teste om et brugernavn og password er korrekt. Dette gør den ved at bruge `Login` klassen.
`createUser(uName, pWord)` bruger igen `Login` klassen, denne gang til at oprette en bruger. Samarbejddiagrammet er vist i figur 7.



Figur 7: `createUser()`

Der er også metoder der kan vise historikken. De bruger alle `Files` klassen og kan enten vise

hele historikken, sidste besked eller en der kan definere hvor mange linjer der skal hentes og vises.

Login klassen

Klassen `Login` bruges til at oprette og teste brugernavn og password. Metoden `addUser(uName, Pword)` opretter nye brugere og gemmer brugernavn og password i et txt dokument.
Klassen har også metoden `testLogin(uName, pWord)`, som bruger metoden `validateLogin(uName, pWord)`. Den modtager et brugernavn og login, som bliver lavet om til en string. Denne string bliver efterfølgende sammenlignet med alle de brugernavne og passwords der er gemt i txt filen. Hvis der ikke findes et match bliver der returneret true og brugeren logges på.

Files klassen

`Files` klassen bruges til at gemme historik, 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 txt dokument med brugerens navn, som historikken kan gemmes i.
Funktionen `addNewLine(besked)` tilføjer en besked til tekstdokumentet. `updateVector()` bruges til at lave en vektor af linjerne fra historikken, som efterfølgende evt. kan printes ud på en skærm så brugeren kan se indholdet. Der er en metode `clearText()`, der kan slette alt fra historikken. Der er også metoden `printVector()`, der kan printe hver linje ud, som er i vektoren skabt med `updateVector()`. `printLatest()` printer den sidste besked ud og `printLines(startN, endM)` kan printe de valgte linjer ud fra linje n til m. Til sidst er der en metoden flipVector(), den bruges til vende vectoren, så fx den sidste modtagne besked kommer til at ligge øverst i vektoren, altså på plads 0.

CharDefinition klassen

`CharDefinition` klassens opgave er at omdanne forskellige karakterer om til binærer tal og tilbage igen. Hvis der oprettes et objekt af klassen, bliver der skabt en string af alle de tal, karakterer og andre tegn der tænkes at skulle kunne sendes. Den string kan bruges i metoderne. Der er en metode som hedder `applyCharDef(input)`, der bruger en anden metode `charToBinary`, som kan omdanne karakterer i en besked til binær, inden den sendes videre til Tranportlaget.
`charToBinary(input)` metoden tager inputtet som er en string af karakterer og laver det om til en binær string. Det gør den ved at kigge på de karakterer, der skal sendes og sammenligne dem med de definerede karakterer og tegn. Hvis der findes et match, bruges nummeret på placeringen af karakteren i den definerede string, som den binære værdi. Den binære værdi bliver 8 bits lang. Fx karakteren b, som har placeringen 12, får binær værdien 00001100.
`receiverCharDef()` er metoden der bruges når der returneres beskeder fra transportlaget og laver beskeden om fra binær til karakterer. Hvis beskeden der modtages er en fejlbesked sende den videre og vises på skærmen. Ellers bruges metoden `binaryToChar(messageReceived)` der omdanner de otte bits om til et decimaltal, som igen kan bruges til at finde placeringen af karakteren, der er modtaget og derefter sætte det på en string. Det gøres indtil hele beskeden er omdannet.

UI source

Til chat programmet er der valgt at lave et simpelt user interface. Det blev lavet ved at bruge konsolvinduet i Visual Studio. Et flowdiagram af systemet er vist på figur 8. Programmet 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 programmet 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. Her er mulighederne: Send besked, modtag besked, se historik og log ud. Ved send besked skærmen kan der skrives en besked, som efterfølgende sendes med metoden `sendBesked(besked, uName)`. Ved modtagBesked skærmen afventes der en besked. Den 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()`. Den sidste valgmulighed er retur til interface skærmen.

7 Konklusion

8 Perspektivering

9 Litteraturliste

9.1 Bøger

[1] John W. Dower *Readings compiled for History 21.479.* 1991.

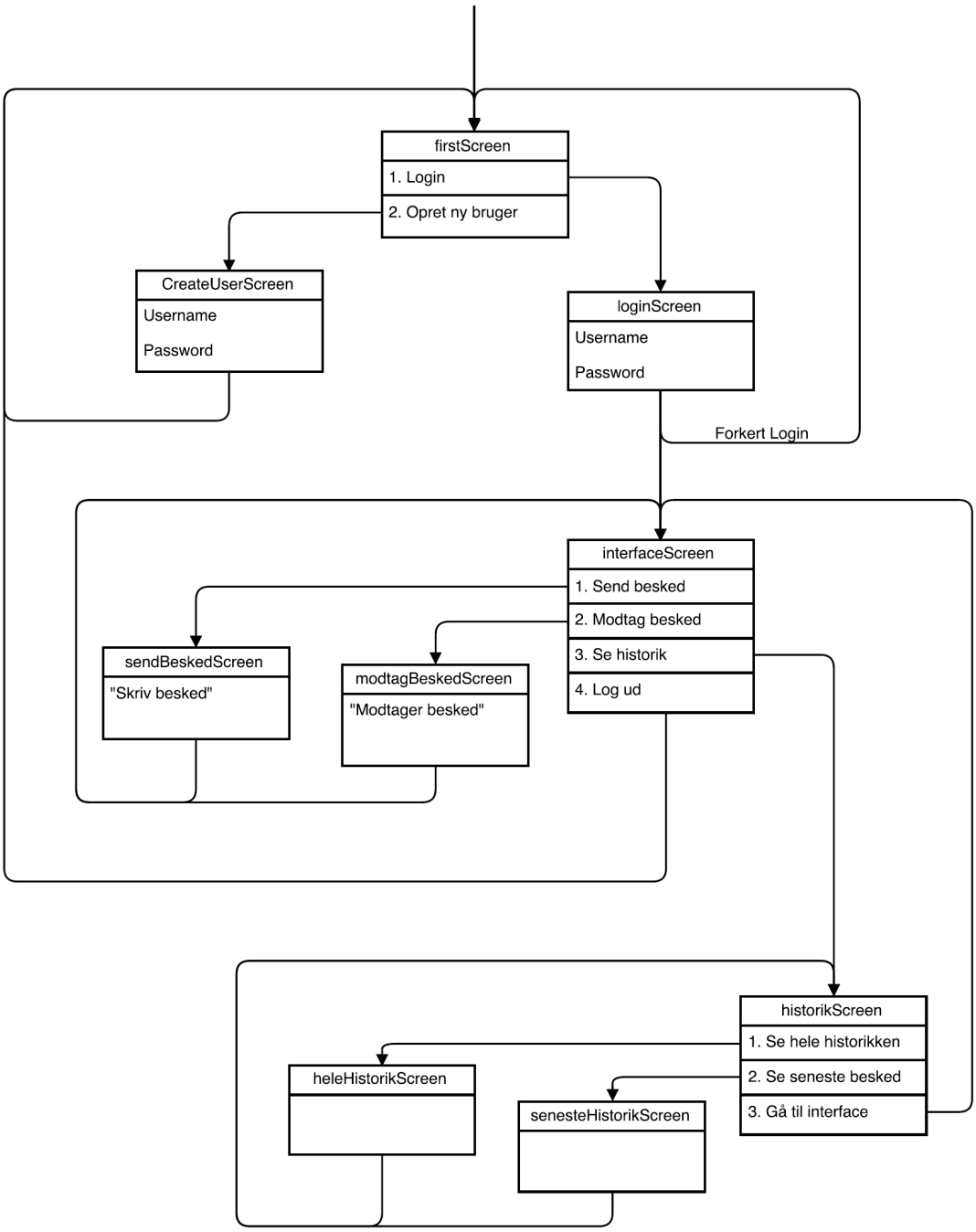
[2] The Japan Reader *Imperial Japan 1800-1945* 1973: Random House, N.Y.

9.2 Hjemmesider

[3] http://einstein.informatik.uni-oldenburg.de/papers/CRC-BitfilterEng.pdf

[4] http://www.ross.net/crc/crcpaper.html

[5] http://www.hackersdelight.org/crc.pdf



Figur 8: `UI flowdiagram`