Optimalisatie van de gesprekskwaliteit bij VoIP

Auteur:
J.A.H.L. Kaijen

Begeleiders:
Dr. ir. A. Pras
Dhr. F.P.R. Overkamp
Ir. R. van de Meent

Enschede, oktober 2005

Voorwoord

Voor de afsluiting van de bachelor Telematica aan de Universiteit Twente heb ik bij SpeakUp een externe opdracht gedaan. Het afgelopen half jaar heb ik met veel plezier gewerkt aan het onderzoek naar kwaliteitsverbetering van VoIP. De goede sfeer die bij SpeakUp heerst heeft zeker positief bijgedragen aan het resultaat van dit onderzoek. Het doet mij goed om te merken dat mijn oplossingen door SpeakUp in de praktijk worden toegepast en niet in een archieflade terecht zal komen.

Graag gebruik ik deze mogelijkheid om mijn dank uit te spreken aan de mensen die mij hebben gesteund tijdens de bacheloropdracht. Graag wil ik de volgende personen in het bijzonder bedanken:

- Dr. ir. A. Pras, begeleider vanuit Universiteit Twente
- Dhr. F.P.R. Overkamp, begeleider vanuit SpeakUp
- Ir. R. van de Meent, begeleider vanuit Universiteit Twente en SpeakUp
- Dhr. J.M. van der Neut-Stulen, voor het verbeteren van de implementaties
- Dr. A.H.M. Kayen, drs. L.W.E. le Blanc - Kaijen en mevr. O.M. Schrevel voor hun beoordeling van stijl en grammatica.

Jesse Kaijen
5 oktober 2005

Inhoudsopgave

ABSTRACT: QUALITY IMPROVEMENT IN VOIP COMMUNICATIONS ... 3

INLEIDING ... 6

1. PUBLIC SWITCHED TELEPHONE NETWORK ... 8

1.1. GESCHIEDENIS ... 8
1.2. WERKING ... 8
1.3. TECHNISCHE VOOR- & NADELEN ... 9

2. VOICE OVER IP ... 10

2.1. GESCHIEDENIS ... 10
2.2. WERKING ... 10
2.3. TECHNISCHE VOOR- & NADELEN ... 11

3. KWALITEITSMETING (VOIP)-GESPREK ... 12

3.1. MEAN OPINION SCORE ... 12
3.2. PERCEPTUAL EVALUATION OF SPEECH QUALITY (PESQ) ... 13
3.3. E-MODEL ... 13
3.4. VERSCHILLEN TUSSEN PESQ EN E-MODEL ... 14

4. INVLOEDEN OP KWALITEIT VAN VOIP–GESPREK ... 15

4.1. DELAY ... 15
4.2. ECHO ... 15
4.3. JITTER ... 16
4.4. PACKETLOSS ... 17
4.5. CODEC ... 17

5. BEPERKING VAN NEGATIEVE KWALITEITSINVLOEDEN ... 19

5.1. JITTERBUFFER ... 19
5.2. KEUZE VAN CODEC ... 20
5.3. ECHO DEMPING ... 20
5.4. CODECSWITCH ... 20
5.5. PACKET LOSS CONCEALMENT ... 21
5.6. PACKETSIZE ... 22

6. VERBETERINGEN IN HET E-MOS ALGORITME ... 23

6.1. E-MOS ALGORITME ... 23
6.2. ECHO EN DELAY ... 24
6.3. PACKETLOSS ... 25
6.4. NIEUWE OPTIMALISATIE FUNCTIES ... 26
6.5. VERBETERD ALGORITME ... 27
6.6. EXTRA DELAY ... 29

7. VERBETERING VAN GESPREKSKWALITEIT DOOR CODECSWITCH ... 30

8. CONCLUSIES EN AANBEVELINGEN ... 32

BRONVERMELDING / LITERATUURLIJST ... 33

BIJLAGE A: OPDRACHTOMSCHRIJVING ... 35

BIJLAGE B: BANDBREEDTE TEST ... 36

BIJLAGE C: JITTERBUFFER HEADERFILE ... 37

BIJLAGE D: JITTERBUFFER ... 44

Abstract: Quality improvement in VoIP communications

Conversational quality in VoIP has been optimized, with focus on jitter buffer (jb) aspects.

Fujimoto, Ata and Murata[1] proposed a new type of jitter buffer, weighing delay (d) against packet loss (p). Based on their assumptions, we conducted further optimization, considering five important parameters on conversational quality i.e. echo, delay, jitter, type of codec and packet loss.
In our judgment, Fujimoto, Ata and Murata implicitly assumed the absence of any echo in the communications. We however, included the reality of echo by assuming a non-perfect echo cancellation between 35 and 40dB.
Figure 1 provides the relation between the perceived quality parameter MOS (Mean Opinion Score) and delay at various echo cancellation levels.

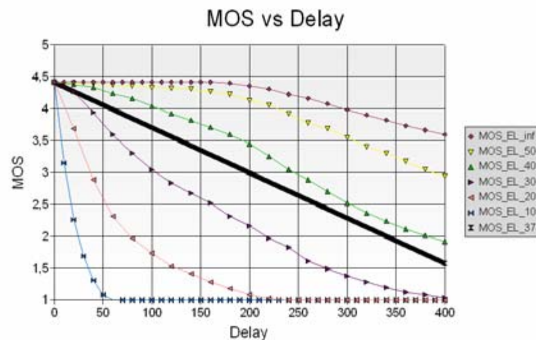


Figure 1. Call-quality rating as a function of mouth-to-ear delay for echo loss values between 10 dB and infinity.[2]

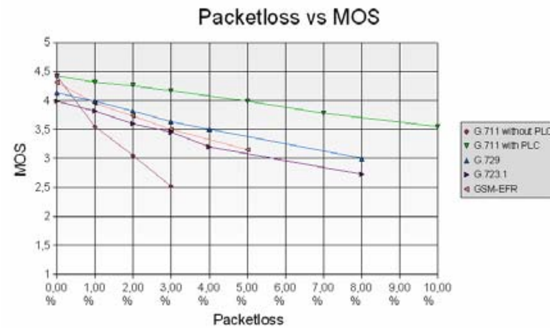


Figure 2. Call-quality rating as a function of packet loss for several codecs. [3]

This justified our interpolative assumption of a linear relation between M and d at 37dB echo cancellation:

$$[1]\ M(d) \approx m - 0{,}0071 \cdot d$$

with m the MOS intercept at $d=0$ and $p=0$, varying by the type of codec. However, quality loss by delay is independent of codec.

The experimentally determined impact of packet loss p on MOS deterioration for 5 types of codec (as reflected in figure 2) seems to be reasonably linear. We therefore assume in general:

$$[2]\ M(p) = m - s \cdot p$$

[1] Bron: [3] K. Fujimoto, S. Ata en M. Murata, "Adaptive Playout Buffer Algorithm for Enhancing Perceived Quality of Streaming Applications," Globecom '02. IEEE, p.2451- p.2457 vol.3, November 2002.
[2] Bron: [7] J. Jansen, D de Vleeschauwer, M. Büchli en G.H. Petit, "Assessing Voice Quality in Packet-Based Telephony," IEEE Internet Computing vol. 6, no 3. p.48-p.57, Mei 2002.
[3] Bron: [7] J. Jansen, D de Vleeschauwer, M. Büchli en G.H. Petit, "Assessing Voice Quality in Packet-Based Telephony," IEEE Internet Computing vol. 6, no 3. p.48-p.57, Mei 2002.

with m the MOS intercept at $d=0$ and $p=0$ and s the 'slope of deterioration'. Both m and s are dependent of the codec:

[2a] $M_{G.711}(p) \approx 4,42 - 0,63 \cdot p$
[2b] $M_{G.711_PLC}(p) \approx 4,42 - 0,087 \cdot p$
[2c] $M_{G.723.1}(p) \approx 3,99 - 0,16 \cdot p$
[2d] $M_{G.729(A)}(p) \approx 4,13 - 0,14 \cdot p$
[2e] $M_{GSM\text{-}EFR}(p) \approx 4,31 - 0,23 \cdot p$

Delay (d) reflects transmission delay (d_n) from the network (which cannot be influenced) as well as delay incurred in the jitter buffer (d_{jb}).
Therefore:

[3] $d = d_n + d_{jb}$

Packet loss p includes p_n (the packet loss occurred in the network) and p_{jb}. The latter is determined by the length of the jitter buffer and can consequently be described by $f(d_{jb})$.

[4] $p = p_n + p_{jb} = p_n + f(d_{jb})$

Fujimoto, Ata and Murata stated that the packet loss ratio and the playout delay affect the MOS value independently, hence can functions [1 and 2] be combined and rewritten as:

[5] $M(d,p) \approx m - 0,0071 \cdot d - s \cdot p$

Considering the effects described in [3 and 4]:

[5a] $M_{G.711}(d_n, d_{jb}, p_n) \approx 4,42 - 0,63 \cdot p_n - 0,63 \cdot f(d_{jb}) - 0,0071 \cdot d_n - 0,0071 \cdot d_{jb}$

[5b] $M_{G.711_PLC}(d_n, d_{jb}, p_n) \approx 4,42 - 0,087 \cdot p_n - 0,087 \cdot f(d_{jb}) - 0,0071 \cdot d_n - 0,0071 \cdot d_{jb}$

[5c] $M_{G.723.1}(d_n, d_{jb}, p_n) \approx 3,99 - 0,16 \cdot p_n - 0,16 \cdot f(d_{jb}) - 0,0071 \cdot d_n - 0,0071 \cdot d_{jb}$

[5d] $M_{G.729(A)}(d_n, d_{jb}, p_n) \approx 4,13 - 0,14 \cdot p_n - 0,14 \cdot f(d_{jb}) - 0,0071 \cdot d_n - 0,0071 \cdot d_{jb}$

[5e] $M_{GSM\text{-}EFR}(d_n, d_{jb}, p_n) \approx 4,31 - 0,23 \cdot p_n - 0,23 \cdot f(d_{jb}) - 0,0071 \cdot d_n - 0,0071 \cdot d_{jb}$

In optimal delay calculations, transmission delay d_n does not need to be included as it is an unaffectedly constant in a linear relationship. The same holds for p_n. As a

consequence, functions [5] (with d_n, d_{jb} and p_n as variable) can be rephrased as [6] with d_{jb} as only floating parameter and d_n /p_n as 'fixed' values, to define a practical quality parameter Q.

[6a] $Q_{G.711}(d_{jb}) \approx 4,42 - 0,63 \cdot p_n - 0,63 \cdot f(d_{jb}) - 0,0071 \cdot d_n - 0,0071 \cdot d_{jb}$

[6b] $Q_{G.711_PLC}(d_{jb}) \approx 4,42 - 0,087 \cdot p_n - 0,087 \cdot f(d_{jb}) - 0,0071 \cdot d_n - 0,0071 \cdot d_{jb}$

[6c] $Q_{G.723.1}(d_{jb}) \approx 3,99 - 0,16 \cdot p_n - 0,16 \cdot f(d_{jb}) - 0,0071 \cdot d_n - 0,0071 \cdot d_{jb}$

[6d] $Q_{G.729(A)}(d_{jb}) \approx 4,13 - 0,14 \cdot p_n - 0,14 \cdot f(d_{jb}) - 0,0071 \cdot d_n - 0,0071 \cdot d_{jb}$

[6e] $Q_{GSM-EFR}(d_{jb}) \approx 4,31 - 0,23 \cdot p_n - 0,23 \cdot f(d_{jb}) - 0,0071 \cdot d_n - 0,0071 \cdot d_{jb}$

The ideal value of d_{jb} can be determined at any time continuously by maximizing $Q(d_{jb})$ through an appropriate algorithm.

Fujimoto, Ata and Murata proposed the Pareto Distribution to model delay. Rather than using this processor intensive approach, we propose to use the discrete actual data of the delay history. By using this approach any tedious discussion which model fits best is avoided. Delay and packet loss are determined as followed:

`history[]` is array of relative delays, sorted at time of reception. The relative delay is the time difference between reception and transmission time.
`history_sorted[]` reflects `history[]`, incrementally sorted at relative delay.
`history_length` is the size of `history[]` and `history_sorted[]`.

While calculating the 'ideal' d_{jb}, `history_sorted[history_length - i]` is decreased by `history_sorted[0]` and therefore eliminating possible clock discrepancies between sender and receiver. By this approach, $f(d_{jb})$ reflects the percentage of the history of the packets received that would be lost at this d_{jb}.

In short:
The jitter buffer performance differs by codec, in order to optimize conversational quality. As a consequence, codec specific functions have been defined.
By using the discrete actual data of the delay history we reduce the needed processor power and avoid the discussion how to model the delay distribution.
We trust to have identified some first steps in indicating a novel direction for further VoIP conversational quality improvement.

Inleiding

Bellen via het Internet is 'hot'.
Internetproviders adverteren er mee en steeds meer mensen zijn te bereiken via VoIP. Iedereen kan elkaar bellen voor een fractie van de kosten die de KPN berekent.
Voice over IP (VoIP) is de techniek die gebruikt wordt voor spraakcommunicatie via het Internet. Dit is een nieuwe technologie waarin er vele nieuwe ontwikkelingen plaatsvinden. Voice over IP wordt in de toekomst wellicht de vervanger van het 'vaste net' (PSTN).

Eén van de bedrijven die zich bezig houdt met het ontwikkelen van VoIP toepassingen is SpeakUp. Dit jonge en dynamische bedrijf levert sinds 2003 (IP-)telefonie oplossingen voor het bedrijfsleven. Enkele voorbeelden van diensten die SpeakUp levert zijn: Voice Response Systemen, koppelingen tussen VoIP en analoge telefonie, telefonie infrastructuur en VoIP toestellen.

Dit onderzoek naar kwaliteitsverbetering van VoIP is door SpeakUp ondersteund. De volledige opdrachtomschrijving is te vinden in bijlage A. Bij kwaliteit is uitgegaan van de gesprekskwaliteit die de gebruiker ervaart. Er is vooral gekeken naar oplossingen waarbij er geen interactie van de gebruiker wordt vereist. In dit onderzoek zijn invloeden van services die werken op netwerkniveau[4] buiten beschouwing gelaten.

De probleemstelling is als volgt:
Tot dusver ontbreekt een applicatie waarbij de kwaliteit van een VoIP-gesprek, zonder interactie van de eindgebruiker, wordt gemaximaliseerd door verschillende technieken. Een dergelijke applicatie biedt mogelijkheden tot verdere verbetering van gesprekskwaliteit.

De bijbehorende onderzoeksvraag luidt:
Hoe kan de kwaliteit van een VoIP-gesprek worden verbeterd zonder interactie van de eindgebruiker? Waardoor ook de gebruikers zonder VoIP-kennis de optimale gesprekskwaliteit tot hun beschikking hebben.

Om deze onderzoeksvraag te beantwoorden zijn de volgende deelvragen opgesteld:

1. Hoe werkt het PSTN?
2. Hoe werkt VoIP?
3. Hoe wordt de kwaliteit van een (VoIP)-gesprek beoordeeld?
4. Wat beïnvloedt de kwaliteit van een VoIP-gesprek?
5. Welke oplossingen om de kwaliteit te beheersen zonder interactie van de gebruiker zijn er op dit moment?
6. Welke verbeteringen zijn er aan de huidige oplossingen mogelijk?
7. Welke verbeteringen zijn het meest van belang voor kwaliteitsverbetering zonder interactie van de eindgebruiker?

[4] Enkele voorbeelden zijn: DiffServ, IntServ en 802.1p/q VLAN based QoS.

De deelvragen 1 tot en met 5 zijn aan de hand van een literatuuronderzoek beantwoord in hoofdstuk 1 tot en met paragraaf 6.1. In hoofdstuk 1 is de werking van het PSTN in een vereenvoudigde weergave beschreven. Door dit hoofdstuk is de werking van VoIP en de problemen die daarbij ontstaan beter te begrijpen. De werking van VoIP staat in hoofdstuk 2 beschreven. Hoofdstukken 1 en 2 zijn als inleiding op VoIP bedoeld.

In hoofdstuk 3 worden aan de hand van literatuur verschillende schalen en scores beschreven die worden gebruikt om de gesprekskwaliteit weer te geven.
Hoofdstuk 4 beschrijft de mate waarin verschillende factoren van invloed zijn op de gesprekskwaliteit. In hoofdstuk 5 worden de huidige oplossingen ter verbetering van de kwaliteit beschreven.

In hoofdstukken 6 en 7 worden twee oplossingen behandeld die een relatief grote invloed hebben op de gesprekskwaliteit. In hoofdstuk 6 wordt een verbetering op een huidige oplossing gegeven. De implementatie hiervan is te vinden in bijlagen D en E. Hoofdstuk 7 beschrijft een nieuwe methode om gesprekskwaliteit te verbeteren. Door middel van deze twee hoofdstukken wordt er antwoord gegeven op deelvragen 6 en 7.

De conclusies en aanbevelingen worden in hoofdstuk 8 gegeven.

1. Public Switched Telephone Network

Om werking van VoIP goed te begrijpen is de werking van het 'vaste net' (PSTN) een handige leidraad. PSTN staat voor Public Switched Telephone Network. De uitleg in dit hoofdstuk geeft de werking in grote lijnen weer.

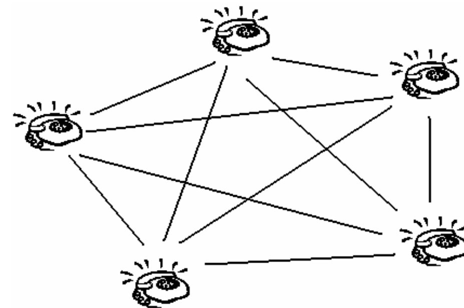
1.1. Geschiedenis

Op 14 februari 1876 patenteerde Alexander Bell één van zijn succesvolste en tevens ook zijn meest omstreden uitvinding: de telefoon. Omstreden vanwege het feit dat slechts twee uur na Bell, Elisha Gray ook een patent op de telefoon indiende. Wat minder mensen weten is dat in 1871 Antonio Meucci een patent klaar had voor de 'teletrofono'. Wegens geldgebrek heeft Meucci nooit het patent kunnen indienen. In 2002 heeft het Amerikaanse Congres Meucci alsnog aangewezen als de werkelijke uitvinder van de telefoon [C,K].

In de begindagen van de telefonie werd de telefoon vooral gebruikt om telegrammen over te brengen. Al snel werd de telefoon gebruikt om gesprekken te voeren. Binnen tien jaar na de introductie hadden alleen al in de Verenigde Staten 150.000 [C] mensen een telefoon. De telefoniemarkt bleef sterk groeien en in 1927 was het eerste transatlantische gesprek mogelijk. In 1935 vond het eerste gesprek rondom de wereld plaats [K]. Hierna is de telefoniemarkt blijven groeien. In de jaren '90 is de mobiele telefoon aan zijn opmars begonnen en ook bellen via Internet heeft zijn intrede gemaakt.

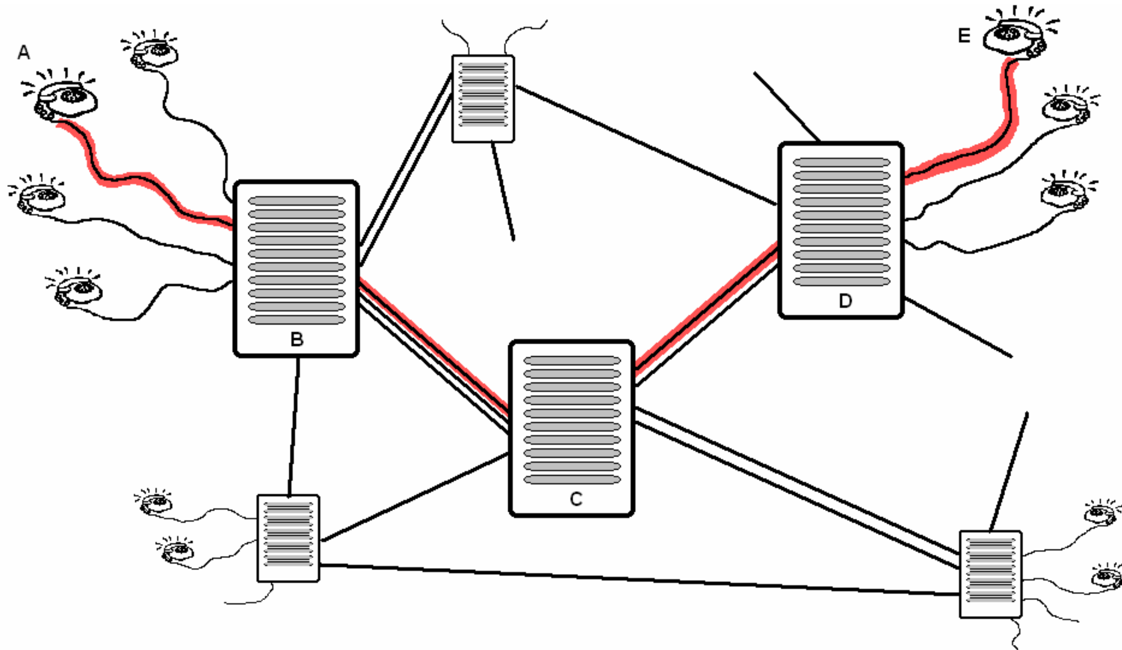
1.2. Werking

In het begin van de telefonie waren alle toestellen direct met elkaar verbonden. Dit is weer gegeven in figuur 1.



Figuur 1. Telefoonsysteem zonder centrale

Wanneer het systeem, weergegeven in figuur 1, met één telefoon wordt uitgebreid, betekent dit vijf extra kabels. Meer toestellen aansluiten resulteert in een explosie van kabels. Daarom is dit systeem niet op grote schaal toepasbaar.
In 1878 werd het eerste schakelbord in gebruik genomen. De telefooncentrale was een feit.[L] Telefooncentrales verbinden twee telefoons met elkaar. Een voorbeeld is weergegeven in figuur 2. Door de komst van de telefooncentrale spreken we van het Public Switched Telephone Network (PSTN).



Figuur 2. Werking van het PSTN: Toestellen A en E bellen met elkaar via centrales B,C,D.

Een telefoongesprek over het PSTN is technisch uit drie delen opgebouwd: verbinding maken, gesprek voeren en de verbinding verbreken. We spreken hier van een circuit-switched connection [M].
Bij het opzetten van de verbinding wordt door het kiezen van het telefoonnummer geprobeerd een circuit te sluiten met de persoon die gebeld wordt. Als de telefoon opgenomen wordt, ontstaat een gesloten circuit. De QoS (Quality of Service) van het gesprek is constant doordat de connectie exclusief voor de gebruikers is.[I] Aan het eind van het gesprek wordt het circuit onderbroken.

1.3. Technische voor- & nadelen

Het PSTN heeft, omdat het circuit-switched is, verschillende voordelen. De vertraging (delay) van het audiosignaal is constant. Tevens is er garantie dat het signaal aankomt zolang er verbinding is. De nadelen die de gebruiker kan ervaren zijn autovervorming, ruis en echo.[12]

In een telefoongesprek bestaan twee audio paden. Voor elke richting één audio pad. Tijdens een telefoongesprek spreekt een gebruiker ongeveer 1/3 van de tijd. De overige tijd luistert hij naar de andere spreker of wordt er niet gesproken. De periode waarin de gebruiker niet spreekt wordt er stilte verstuurd. Dit niet optimale gebruik van het audio pad is een belangrijk technisch nadeel.

2. Voice over IP

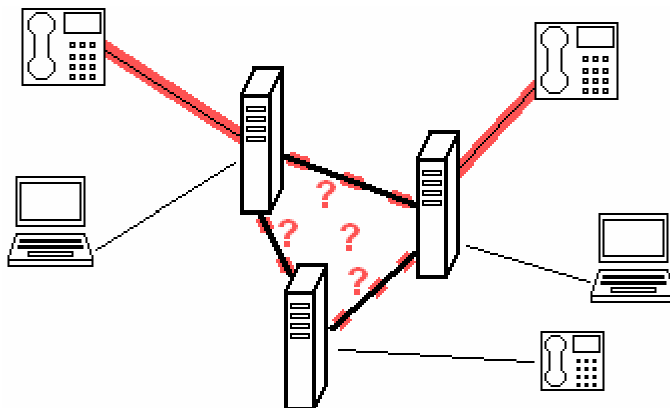
Voice over IP en VoIP zijn beide afkortingen voor Voice over Internet Protocol. In dit hoofdstuk wordt de geschiedenis en werking van VoIP uitgelegd. Simpel gezegd is VoIP telefoneren over computernetwerken, zoals het Internet of bedrijfsnetwerken.

2.1. Geschiedenis

Tijdens de opkomst van het Internet zijn hobbyisten begonnen met spraak over internet. Hierbij was het noodzakelijk dat beide gebruikers dezelfde apparatuur gebruikten. In 1994 was het technisch gezien mogelijk om van PC naar PC te 'bellen'. In het opvolgende jaar kwam Vocaltec Inc. met een commercieel product op de markt waarmee bellen tussen twee computers mogelijk werd. In de opvolgende jaren werd het mogelijk om van PC naar telefoon en ook van telefoon via Pc's naar telefoon te bellen. Vanaf 1998 zijn er steeds meer commerciële toepassingen ontstaan, zoals de koppeling tussen computernetwerken (VoIP) en het PSTN, dit noemt men gateways [K]. Tijdens de internethype in het begin van dit millennium zijn deze technieken verder ontwikkeld [2,E].

2.2. Werking

Bij Voice over IP wordt, zoals de naam al verraadt, de gespreksinformatie over een IP-netwerk verstuurd. Deze netwerken zijn packetswitched. Dit betekent dat alle informatie middels pakketten verzonden moet worden. Elk pakket kan een andere route door het IP-netwerk volgen via verschillende node's. Dit is weergegeven in figuur 3. Node's in een IP-netwerk zijn bijvoorbeeld computers en switches.



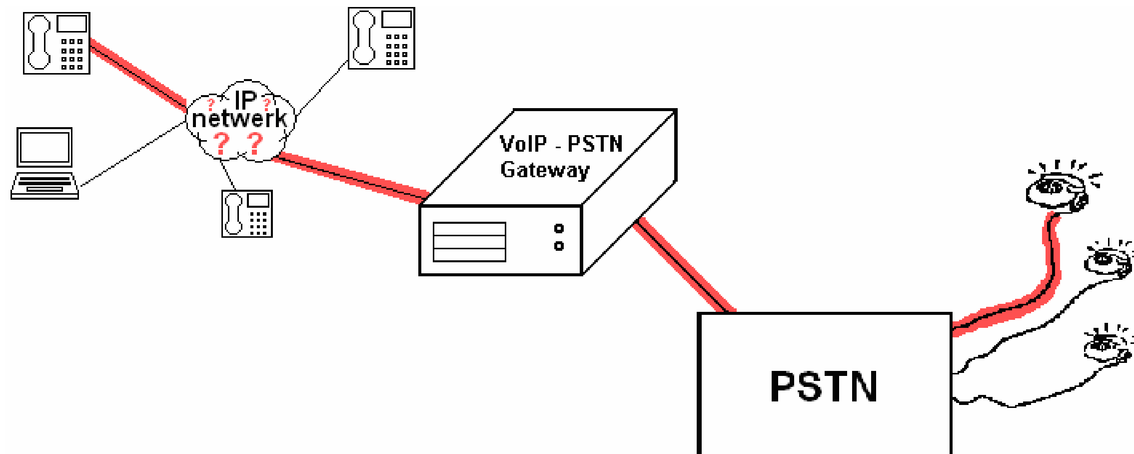
Figuur 3. VoIP gesprek over IP-netwerk. De route die de pakketten afleggen is onbekend.

Omdat een VoIP-gesprek over een IP-netwerk gaat en er pakketten verzonden moeten worden. Wordt het audiosignaal in blokken, van bijvoorbeeld 20 ms, gedeeld en in een pakket verzonden. Het digitaliseren en eventueel comprimeren van het blok audio gebeurt door een codec. Paragraaf 4.5 gaat dieper in op codec's.

Het signaleren van VoIP wordt afgehandeld door middel van een protocol, zoals SIP, H.323 en IAX2 [J]. Enkele signaleringsvoorbeelden zijn: het opzetten van de verbinding, afspreken welke codec gebruikt gaat worden en het verbreken van de verbinding [6,11].

Om een verbinding op te kunnen zetten moet bekend zijn waar een telefoon zich bevindt. Een router houdt bij waar telefoons zich bevinden en welk telefoonnummer(s) bij welke telefoon horen.[11]
Een router weet waar een telefoon zich bevindt doordat de telefoon zich regelmatig registreert bij de router. Deze registratie wordt ook afgehandeld door het protocol.

De koppeling tussen VoIP en PSTN is mogelijk via een VoIP-PSTN gateway. Deze gateway maakt signaaluitwisseling tussen deze twee netwerken mogelijk.[2,R] Een gesprek tussen een VoIP telefoon en een analoge telefoon is weergegeven in figuur 4.



Figuur 4. VoIP-PSTN gesprek d.m.v. een VoIP-PSTN Gateway.

2.3. Technische voor- & nadelen

Het is niet gegarandeerd dat de pakketten hetzelfde pad volgen. Pakketten kunnen verschillende vertragingen oplopen, zelfs als ze dezelfde route over het IP-netwerk zouden volgen.
Bij het PSTN worden stiltes verzonden, met VoIP is dit niet nodig. Het protocol kan signaleren dat er geen audio verzonden wordt, maar dat het gesprek nog wel actief is. Hierdoor hoeft het overbodige audiosignaal, stilte, niet verzonden worden.
Doordat het signaal digitaal verstuurd wordt, kan er tijdens de transmissie geen signaalvervorming of ruis optreden.
In het IP-netwerk kunnen wel hele pakketten verloren gaan, stukken audio raken dus kwijt.

3. Kwaliteitsmeting (VoIP)-gesprek

Om over kwaliteit te kunnen spreken en vergelijkingen maken tussen verschillende gesprekken is een score of schaal nodig. Hieronder staat een bondige uitleg van verschillende manieren van kwaliteitsmeting en -schaling. Verdere diepgang biedt het artikel van Telchemy.[9]

Bij het meten van gesprekskwaliteit zijn er twee belangrijke kwaliteitscategorieën te onderscheiden.

- *Listening Quality* (LQ) geeft de kwaliteit weer van wat de gebruiker 'hoort'.
- *Conversational Quality* (CQ) geeft de kwaliteit van het volledige gesprek weer. Dit houdt in dat naast de geluidskwaliteit ook rekening wordt gehouden met hoe goed gebruikers kunnen converseren.

3.1. Mean Opinion Score

De *Mean Opinion Score (MOS)* wordt gebruikt voor het weergeven van de kwaliteit van een telefoongesprek. Deze score is ontwikkeld door de International Telecommunication Union (ITU) en is samengesteld uit het gemiddelde van subjectieve luistertests die gedaan zijn door een panel. De score is ontwikkeld om telefoonlijnen te evalueren.[19] Een bekende methode is de *Absolute Category Rating* (ACR) test. Tijdens zo'n test krijgt het panel zinnen voorgelezen door zowel mannen als vrouwen, waarna ze een waardering geven op de schaal 1 tot 5.[9] In tabel 1 staat de MOS-schaal beschreven.

Rating	Definition	Description
5	Excellent	As good as perfect AM radio reception
4	Good	Some minor noise or distortion but completely understandable speech
3	Fair	Requires some hearing effort
2	Poor	Speech is difficult to understand
1	Bad	Speech is recognized but unintelligible

Tabel 1. MOS-schaal [F]

Om elke keer de kwaliteit te laten testen door een panel is duur en tijdrovend. Daarom zijn er verschillende geautomatiseerde methodes ontwikkeld door telecombedrijven. In dit document worden PESQ[5] en het E-model besproken.

[5] PESQ is de vervanger voor de methode's PAMS, PEAQ, PSQM en PSQM+.

3.2. Perceptual Evaluation of Speech Quality (PESQ)

Perceptual Evaluation of Speech Quality (PESQ) is een methode om gesprekskwaliteitsmeting te doen. Deze methode is in 2001 ontwikkeld door Opticom, Psytechnics Limited, TNO Telecom (voormalige KPN Research) en British Telecommunications.[16,G]

PESQ is een actieve methode. Dit houdt in dat de opname van het origineel van de zender vergeleken wordt met de opname van het gedegradeerde signaal van de ontvanger. Deze twee worden met elkaar vergeleken waar een waarde uit berekend wordt. Deze waarde geeft de audio kwaliteit weer in MOS-LQ. De correlatie tussen PESQ en ACR-luistertests is 93,5%. Dit is een zeer acceptabele overeenkomst en PESQ kan goed gebruikt worden als indicatie voor de MOS.[16]

3.3. E-model

Het E-model is in eerste instantie ontwikkeld als een planningtool voor telecommunicatie netwerken door het ETSI (European Telecommunications Standardization Institute).[18] Tegenwoordig wordt het ook gebruikt voor kwaliteitsmeting van VoIP. Het E-model is een passieve methode. Dit heeft onder andere het voordeel dat het E-model geen origineel als referentie nodig heeft.
Het E-model maakt gebruik van de zogenaamde “R” factor. Het bereik van deze factor loopt van 0-120. De R-factor kan berekend worden met de volgende formule:

```
R = R0 - Is - Id - Ie-eff + A
```

“*Ro* represents in principle the basic signal-to-noise ratio, including noise sources such as circuit noise and room noise. The factor *Is* is a combination of all impairments which occur more or less simultaneously with the voice signal. Factor *Id* represents the impairments caused by delay and the effective equipment impairment factor *Ie-eff* represents impairments caused by low bit-rate codecs. It also includes impairment due to packet-losses of random distribution. The advantage factor *A* allows for compensation of impairment factors when there are other advantages of access to the user.”[18]

In de literatuur zijn verschillende relaties tussen de R-factor en MOS gedefinieerd. In dit onderzoek is uitgegaan van de relatie zoals deze gedefinieerd is door de International Telecommunications Union (ITU). Dit wordt weergegeven met MOS (ITU-schaal).
Het is mogelijk om de R-factor om te rekenen naar MOS (ITU-schaal) door middel van de formule gegeven in figuur 5.

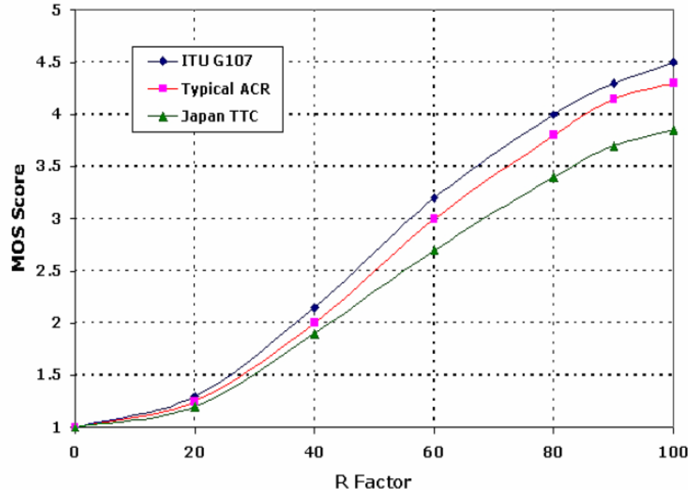
For $R < 0$: $MOS_{CQE} = 1$

For $0 < R < 100$: $MOS_{CQE} = 1 + 0.035R + R(R-60)(100-R)7 \cdot 10^{-6}$

For $R > 100$: $MOS_{CQE} = 4.5$

Figuur 5. Omrekenformule tussen R-factor en MOS (ITU-schaal). [18]

Naast de relatie tussen de R-factor en MOS zoals gedefinieerd door de ITU, zijn er ook andere interpretaties van deze relatie. Er zijn meerdere interpretaties van de relatie tussen R-factor en MOS door verschillende culturele percepties van gesprekskwaliteit.[9] In figuur 6 zijn enkele relaties weergegeven. Verschil in interpretatie is mogelijk doordat MOS een subjectieve score is.



Figuur 6. Verschillende relaties tussen R-factor en MOS. [9]

Bij het vergelijken van verschillende MOS-scores is het belangrijk om te weten welke interpretatie is gebruikt.

3.4. Verschillen tussen PESQ en E-Model

PESQ en het E-model zijn beide methodes om gesprekskwaliteit te berekenen. PESQ heeft, vergeleken met het E-model, veel meer rekenkracht nodig om een MOS-waarde te berekenen. PESQ is een actieve methode en heeft daarom twee opnames nodig om tot een vergelijking te komen.
Het E-model kan tijdens een gesprek gebruikt worden en is licht qua rekenkracht. Dit komt omdat er geen vergelijking tussen twee audiosignalen optreedt.

PESQ kan alleen een MOS-LQ waarde berekenen. Deze MOS-waarde is volgens de ACR-schaal. Het E-model kan zowel R-LQ als R-CQ berekenen. Door de formule gegeven in figuur 5 kunnen deze waardes omgerekend worden naar respectievelijk MOS-LQ en MOS-CQ.

Beide methodes worden gebruikt om gesprekskwaliteit weer te geven. Bij vergelijking van gegevens uit verschillende onderzoeken moet goed opgelet worden welke interpretatie is gebruikt.

4. Invloeden op kwaliteit van VoIP–gesprek

Verschillende factoren hebben invloed op de kwaliteit van een VoIP-gesprek. De vijf belangrijkste factoren die invloed hebben op de gesprekskwaliteit van VoIP zijn delay, echo, jitter, packetloss en codec. In dit hoofdstuk worden deze factoren uitgelegd waarbij de engelse benaming wordt gehanteerd.

4.1. Delay

Bij een telefoongesprek is de volledige “mond-tot-oor” delay (vertraging) van invloed op de gesprekskwaliteit.[7] Bij het PSTN is dit de transmissietijd over de kabels. Deze delay is bij het PSTN nagenoeg constant[6].

Bij VoIP wordt delay opgelopen door verschillende oorzaken. Er treedt delay op tijdens verzenden, transmissie en ontvangst.

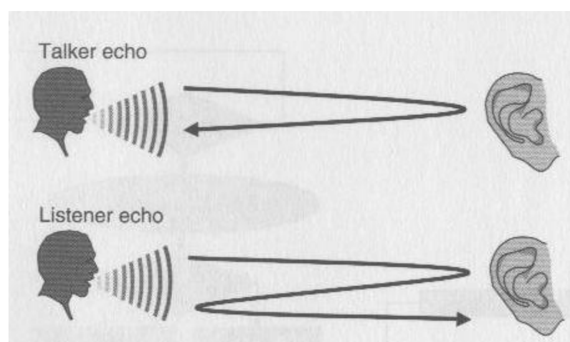
De delay die optreedt bij het verzenden ontstaat doordat er een aantal milliseconden moet worden gewacht totdat er een blok audio gevuld is om te coderen en als pakket te versturen.[11]

Tijdens transmissie treedt er, naast de fysieke transmissietijd, extra delay op in het IP-netwerk. Dit door de buffers van de node’s in het netwerk. Node’s in een IP-netwerk zijn computers, switches, etcetera.

Er treedt tevens variatie op in de transmissie delay. Dit wordt jitter genoemd. Om deze jitter te ondervangen wordt gebruik gemaakt van een jitterbuffer. Deze jitterbuffer zorgt voor een continue audiostroom aan de gebruiker. Om deze continue stroom te behouden zorgt de jitterbuffer voor extra delay bij ontvangst. Op de werking van de jitterbuffer wordt dieper ingegaan in paragraaf 5.1.
De invloed van delay op de gesprekskwaliteit is mede afhankelijk van de sterkte van de echo. Beide staan weergegeven in figuur 8.

4.2. Echo

De belangrijkste echo is de 'talker' echo. Hier hoort de spreker zijn eigen stem terug. Bij een 'listener' echo hoort de luisteraar de spreker twee keer. Eén keer direct en nog een keer sterk vertraagd doordat het nogmaals via de spreker terug komt.[1]

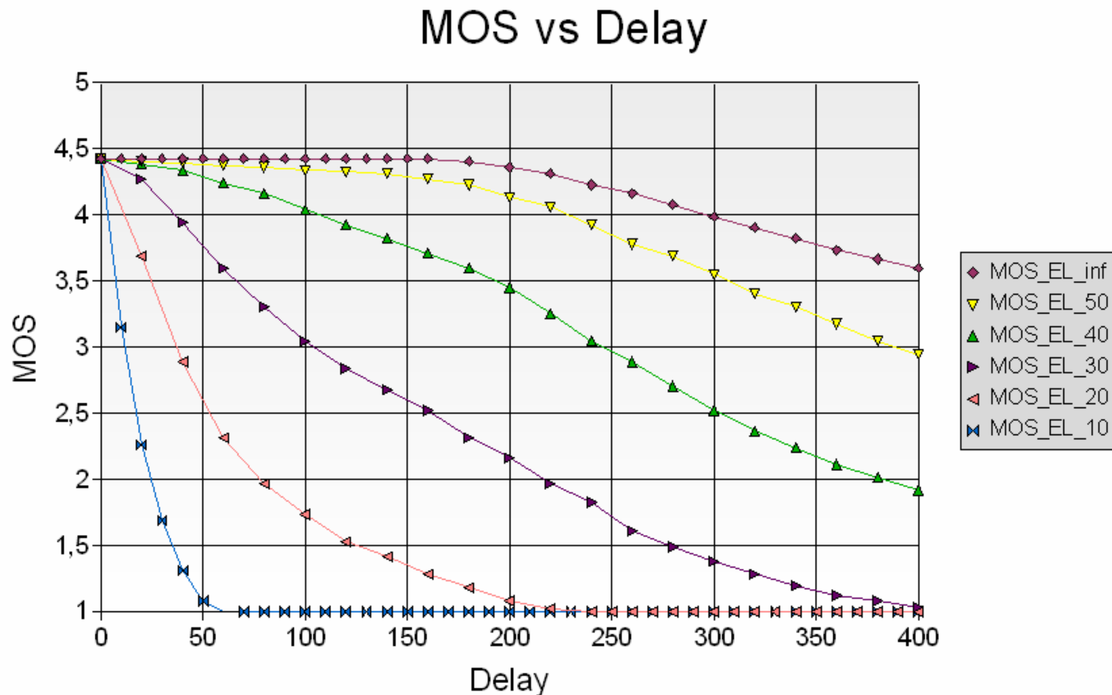


Figuur 7. Talker en listener echo. [1]

[6] Het effect op de gesprekskwaliteit van variatie in de delay bij het PSTN is te verwaarlozen.

De oorzaak van een echo kan akoestisch of elektronisch zijn. Bij een akoestische echo gaat het geluid via de luidspreker van de ontvanger weer de microfoon van de ontvanger in. In het PSTN kan tevens een elektronische echo optreden. Dit houdt in dat het circuit niet met de juiste impedantie is afgesloten.[7] [1]

De mate van onderdrukking van een echo wordt uitgedrukt in decibellen (dB). Als er geen echo is, gaat de onderdrukking richting oneindig. De invloed van echo en delay is weergegeven in figuur 8. De kwaliteit is met een delay van 0 ms bij alle soorten echo 4,4. Bij een delay van 200 ms is de kwaliteit met een demping van 40 dB ongeveer 3,5.



Figuur 8. Invloed van echo en delay op de MOS-CQ (ITU-schaal). [7]

4.3. Jitter

Jitter is bij IP-netwerken de variatie in de delay van pakketten. Dit is van invloed bij VoIP. Deze jitter wordt uitgedrukt in milliseconden.Er zijn twee soorten jitter die in een IP-netwerk kunnen optreden:

- Normale jitter die optreedt door fluctuaties van belasting van tussenliggende node's.
- Een spike treedt op als er een piek belasting is of als er een node uitvalt en/of een andere route gevolgd wordt.

[7] Het signaal wordt niet volledig omgezet naar een akoestisch signaal, een deel wordt gereflecteerd.

4.4. Packetloss

Bij VoIP is er sprake van twee soorten packetloss.[3]
Ten eerste kunnen pakketten verloren zijn gegaan in het IP-netwerk. Enkele voorbeelden zijn bufferoverflows bij node's of het uitvallen van een node.
Ten tweede spreekt men van packetloss als pakketten te laat aankomen. Een pakket wordt verloren beschouwd als het op het moment dat het afgespeeld moet worden nog niet ontvangen is.
De invloed van packetloss op de gesprekskwaliteit is afhankelijk van de gebruikte codec. Dit is weergegeven in figuren 9 en 10.

4.5. Codec

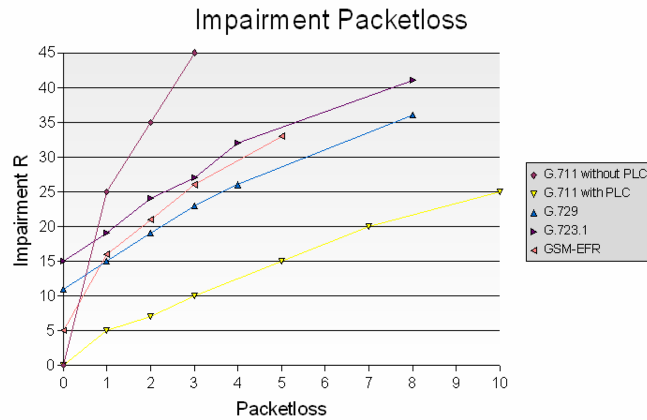
Een codec is een stuk soft- of hardware dat toelaat data te coderen en/of te decoderen. [O]
Bij het coderen van de data, bij VoIP het audiosignaal, kan compressie[8] toegepast worden. De manier waarop compressie wordt toegepast is per codec verschillend. Het audiosignaal wordt in blokken verwerkt door de codec. De meest voorkomende bloklengtes zijn 20 of 30ms.
Er zijn verschillende codec's met allemaal hun eigen eigenschappen, zoals bitrate[9], benodigde rekenkracht en maximaal haalbare MOS-waarde. Tabel 2 geeft een aantal codec's weer met hun maximaal haalbare MOS waarde en de benodigde bitrate.[1]
Bij codec G.711 wordt het audiosignaal niet gecomprimeerd. Hierdoor is de benodigde bandbreedte en (theoretische) kwaliteit overeenkomstig met de bandbreedte en kwaliteit van het PSTN. De overige codec's comprimeren het audiosignaal wel. De benodigde bandbreedte is lager dan bij het PSTN. Dit gaat ten koste van de maximaal haalbare kwaliteit.

Codec	**G.711**	**G.726**	**G.729**	**G.723.1**	**GSM-FR**	**GSM-HR**	**GSM-EFR**
MOS (ACR-schaal)	4,2	?/?/4/4,2	4	3,9/3,7	3,6-3,8	3,5-3,7	4,1
kbit/s	64	16/24/32/40	8/6,4/11,8	6,3/5,3	13	5,6	12,2

Tabel 2. Codec's met hun MOS en benodigde bitrate. [1]

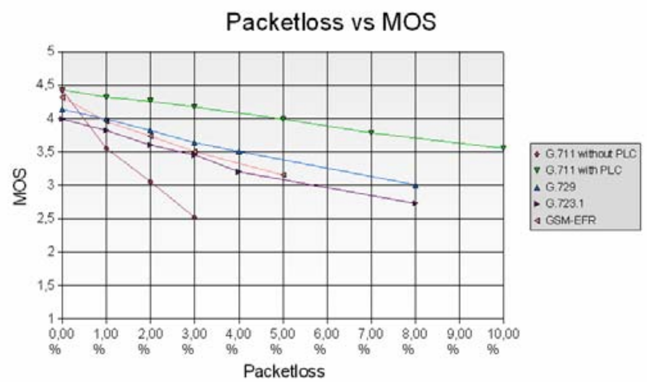
Ook de mate waarin de gesprekskwaliteit achteruit gaat als een pakket verloren gaat, is per codec verschillend. Sommige codec's hebben in hun algoritmes al technieken ingebouwd om packetloss op te vangen. De invloed van packetloss op de R-factor is per codec weergegeven in figuur 9.

[8] Comprimeren is het verkleinen van de data waardoor het minder ruimte inneemt. Dit kan ten koste van informatie gaan.
[9] Benodigde bits/seconde



Figuur 9. Invloed van packetloss op de R-factor. [7]

De invloed van packetloss op de MOS-CQ (ITU-schaal) is weergegeven in figuur 10. Deze berekening is tot stand gekomen met behulp van de formule in figuur 5.



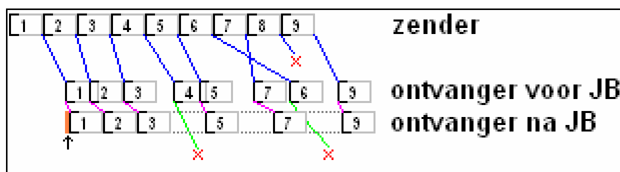
Figuur 10. Per codec de invloed van packetloss op de MOS-waarde (ITU-schaal).

5. Beperking van negatieve kwaliteitsinvloeden

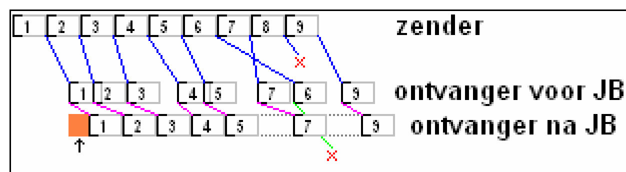
Er zijn verschillende oplossingen voor de problemen die kunnen ontstaan door de invloeden die gegeven zijn in het vorige hoofdstuk. Dit zijn de huidige oplossingen.

5.1. Jitterbuffer

Een jitterbuffer (JB) zorgt ervoor dat de variatie in delay (jitter) ondervangen wordt. Dit gebeurt door de pakketten op te vangen en te vertragen. Zo ontstaat een constante stroom van pakketten om af te leveren aan de decoder. De grootte van een jitterbuffer moet zorgvuldig gekozen worden. Er moet een afweging gemaakt worden tussen delay en packetloss. In Figuur 111 en Figuur 122 is dit principe uitgewerkt. In Figuur 111 is er een kleine jitterbuffer, daardoor worden pakketten 4, 6 en 8 als verloren beschouwd. Ondanks dat pakketten 4 en 6 toch aankomen. Doordat er in Figuur 122 een grotere jitterbuffer is, kan pakket 4 wel afgespeeld worden. De lengte van de jitterbuffer is met een pijl weergeven in beide figuren.[1]



Figuur 11. Werking van de jitterbuffer. Kleine jitterbuffer.



Figuur 12. Werking van de jitterbuffer. Grote jitterbuffer.

De grootte van de jitterbuffer bepaalt welke delay er optreedt aan de ontvangende kant. Hoe groter de delay hoe lager de gesprekskwaliteit is, zoals te zien is in Figuur 8. Als de jitterbuffer kleiner is, kunnen pakketten onterecht als verloren beschouwd worden. In Figuur 10 is te zien welk effect packetloss heeft op de gesprekskwaliteit.
De keuze van de JB omvang is een compromis tussen kwaliteitsparameters: een te grote JB leidt tot kwaliteitsverlies door delay, terwijl bij een kleine JB door verlies van pakketten informatie verloren gaat.

Voor het bepalen van de optimale grootte van de jitterbuffer zijn verschillende algoritmes ontworpen.[3] Hieronder staan er enkele beschreven.

De statische jitterbuffer heeft een vaste lengte. Het nadeel hiervan is onnodige vertraging bij weinig jitter. Bij veel jitter zullen er echter veel pakketten als verloren worden beschouwd, doordat ze te laat zijn. Dit is geen optimale situatie.

Een dynamische jitterbuffer past zich aan de hoeveelheid jitter die optreedt. Bij weinig jitter is hij klein en bij veel jitter groter. Om de grootte van de jitterbuffer te bepalen

wordt (een deel van) de geschiedenis van ontvangen pakketten bewaard. Een algoritme berekent aan de hand van deze gegevens de optimale grootte.

Eerder ontwikkelde dynamische algoritmes richten zich vooral op het behoud van zo veel mogelijk pakketten of het behoud van een bepaald percentage van de pakketten. In deze algoritmes wordt geen rekening gehouden met kwaliteitsfactoren, zoals delay en de gebruikte codec.

Fujimoto, Ata en Murata [3] hebben een nieuw soort algoritme ontworpen wat de invloeden op de gesprekskwaliteit als uitgangspunt heeft. Dit is het Enhanced-MOS (E-MOS) algoritme. Uit een vergelijkende test bleek dat het E-MOS algoritme onder verschillende omstandigheden steeds het beste resultaat leverde. Voor mijn onderzoek is daarom gekozen voor het E-MOS algoritme om als basis te gebruiken. De werking wordt uitgelegd in hoofdstuk 6.

5.2. Keuze van codec

De keuze van de codec is afhankelijk van een aantal factoren. Zo moet de benodigde bandbreedte beschikbaar zijn. Er ontstaat ongewenst gedrag[10] als er meer bandbreedte gevraagd wordt dan dat er beschikbaar is.
Tevens zijn sommige verbindingen zijn aan packetloss onderhevig, bijvoorbeeld draadloos internet. Als dit van tevoren bekend is, is het verstandig een codec te gebruiken die hier goed tegen kan.[13]
Bij de keuze van een codec moet rekening gehouden worden met de beschikbare rekenkracht en ondersteuning van de codec aan de ontvangende kant.

5.3. Echo demping

In figuur 8, paragraaf 4.2, staat de invloed van echo op de gesprekskwaliteit weergegeven. Zoals uit het figuur blijkt, is de invloed van delay op de MOS-waarde veel sterker als er weinig echo demping optreedt.
Wanneer er een betere echo-onderdrukking is, is de invloed van delay op de MOS-waarde minder sterk.[7] Er zijn verschillende oplossingen om echo te verminderen. Enkele voorbeelden zijn: volume instellingen van microfoon en luidspreker, en softwarematige detectie en onderdrukking.

5.4. Codecswitch

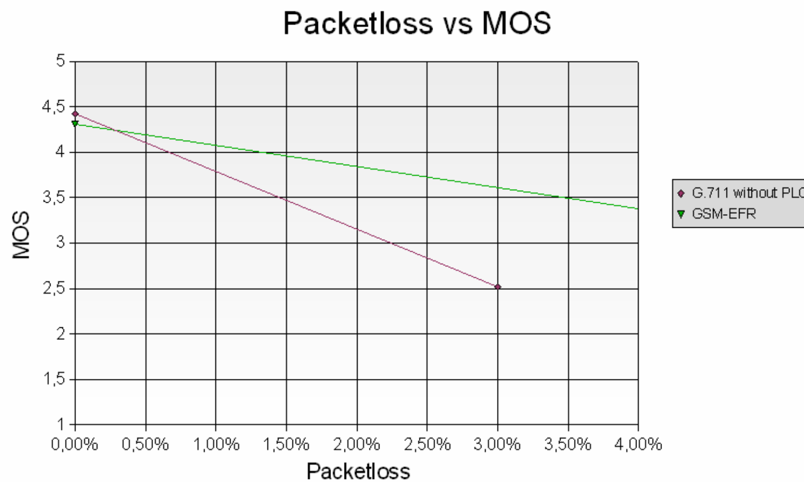
Het veranderen van codec tijdens een gesprek wordt codecswitch genoemd. Uit onderzoek van Robustelli, Loreto, Fresa, Longo en Spinelli [8] is gebleken dat wanneer er onder bepaalde voorwaarden een codecswitch optreedt, een hogere gesprekskwaliteit kan worden bereikt. Deze oplossing wordt op moment van schrijven nog niet breed toegepast, maar biedt wel goede mogelijkheden voor de toekomst.

De voorwaarden voor een codecswitch worden onder andere bepaald door de netwerkcondities. Wanneer er packetloss optreedt, kan het zijn dat met dezelfde packetloss een andere codec beter presteert. Grafisch is dit weergegeven in Figuur 133. Hier is te zien dat bij een packetloss van minder dan 0,4% de codec G.711 beter presteert

---

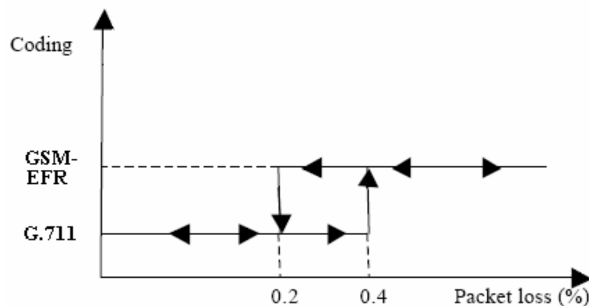
[10] Dit ongewenste gedrag is aangetoond en beschreven in bijlage B.

dan codec GSM-EFR. Bij een packetloss hoger dan 0,4% levert GSM-EFR een hogere gesprekskwaliteit.



Figuur 13. Invloed van packetloss op de MOS-waarde voor codec's G.711 en GSM-EFR.

In dit onderzoek wordt voorgesteld om bij een packetloss van < 0,2% de codec G.711 te gebruiken. Wanneer de packetloss > 0,4% is wordt GSM-EFR gebruikt. Bij packetloss tussen 0,2% en 0,4% blijft met de reeds gebruikte codec gebruiken. Dit is om te voorkomen dat te vaak van codec geswitched wordt.



Figuur 14. Switchen tussen codec's [8]

5.5. Packet Loss Concealment

Packet Loss Concealment (PLC) is een algoritme dat in werking treedt wanneer een pakket als verloren beschouwd wordt. Verschillende codec's bevatten reeds een PLC algoritme.[13] De werking van zo'n algoritme is als volgt:
Indien een pakket als verloren wordt beschouwd, wordt een nieuw pakket aangemaakt. Zodanig dat er toch audio afgespeeld kan worden. De meest eenvoudige algoritmes herhalen het vorige pakket. Door deze herhaling kan het gesprek wel 'robotachtig' klinken. Geavanceerdere algoritmes kunnen dit voorkomen, maar hebben wel meer rekenkracht nodig. Op de site van VoIPtroubleshooter[H] zijn audio files te vinden die het effect goed weergeven.

5.6. Packetsize

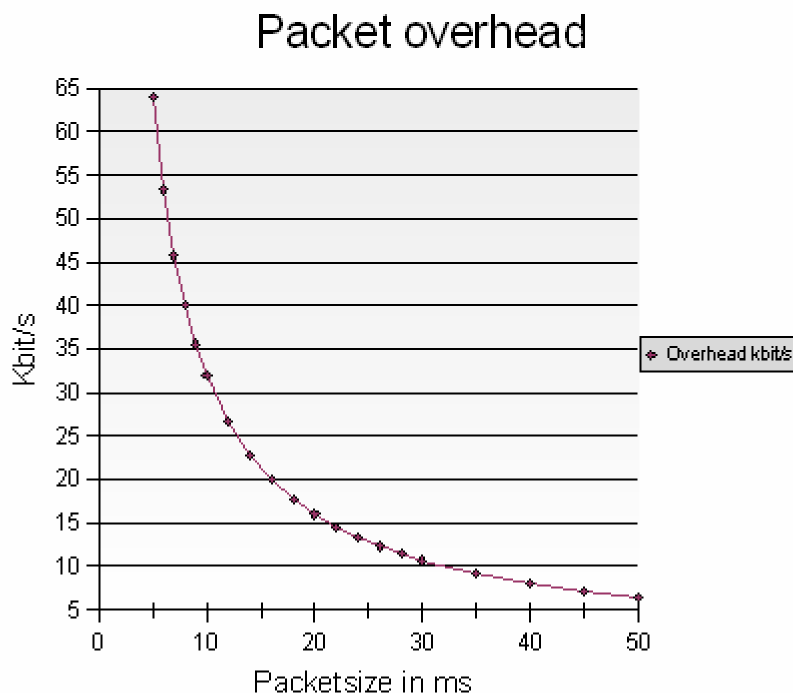
De delay die ontstaat bij het verzenden is beïnvloedbaar. Deze delay is de hoeveelheid milliseconden audio die per pakket wordt verzonden.
Elk pakket bestaat uit twee delen, een header en data.[P] De header zorgt ervoor dat het pakket bij de ontvanger aankomt. De data is bij VoIP de gecodeerde audio.

Figuur 15. Pakket bestaat uit een header en data.

Wanneer er minder milliseconden aan audio per pakket verzonden worden, moeten er meer pakketten per seconde verzonden worden. Doordat er meer pakketten nodig zijn, worden er meer headers mee verzonden. Hierdoor stijgt de benodigde bandbreedte.[6] De bandbreedte die de headers in beslag nemen wordt overhead genoemd.
De overhead (kbit/s) is te berekenen met onderstaande functie [6] en is weergegeven in figuur 16. Een pakketlengte van 20 of 30 milliseconden is gebruikelijk.

```
Overhead = H/S = 320/S
```

H=headersize in bits, deze is protocol afhankelijk. Bij IAX2 is de headersize 320 bits.
S=packetsize in seconde.



Figuur 16. Overhead in kbit/s voor gebruik van IAX2 als protocol.

6. Verbeteringen in het E-MOS algoritme

In paragraaf 6.1 wordt de werking van het E-MOS algoritme beschreven, zoals deze is voorgesteld door Fujimoto, Ata en Murata.[3] In de opvolgende paragrafen worden mijn verbeteringen op het E-MOS algoritme voorgesteld, die in paragraaf 6.5 leiden tot een verbeterd algoritme.

6.1. E-MOS algoritme

De belangrijkste aanname in het E-MOS algoritme is dat delay en packetloss onafhankelijk van elkaar de gesprekskwaliteit beïnvloeden. Daarnaast is men alleen uitgegaan van codec G.711.[3]
Voor de afzonderlijke invloed van delay (d) en packetloss (p) op de gesprekskwaliteit geven de auteurs twee functies voor de MOS-waarde[11] (M).

$$M(d) \approx 4{,}10 + 2{,}64\cdot10^{-3}\cdot d - 1{,}86\cdot10^{-5}\cdot d^2 + 1{,}22\cdot10^{-8}\cdot d^3$$
$$M(p) \approx 4{,}10 - 0{,}195\cdot p$$

Omdat beiden onafhankelijk de gesprekskwaliteit beïnvloeden kunnen de functies samengevoegd worden.

$$M(d,p) \approx 4{,}10 - 0{,}195\cdot p + 2{,}64\cdot10^{-3}\cdot d - 1{,}86\cdot10^{-5}\cdot d^2 + 1{,}22\cdot10^{-8}\cdot d^3$$

De delay is de <u>volledige</u> delay, dus de tijd tussen spreken van de zender en het afspelen bij de ontvanger. De packetloss is onder te verdelen in packetloss in het netwerk (p_n) en packetloss door de grootte van de jitterbuffer (p_{jb}).[12]

$$p = p_n + p_{jb}$$

p_{jb} is afhankelijk van de lengte van de jitterbuffer, de delay. Hierdoor kan p_{jb} in d uitgedrukt worden.

$$p_{jb} = 100\left({}^{k}/_{d}\right)^{\alpha}$$

k en α zijn statistische parameters van de cumulatieve Pareto distributie[S] die wordt verkregen uit de relatieve delay van de ontvangen pakketten.

$$M(d,p_n) \approx 4{,}10 - 0{,}195\cdot\left(p_n + 100\left({}^{k}/_{d}\right)^{\alpha}\right) + 2{,}64\cdot10^{-3}\cdot d - 1{,}86\cdot10^{-5}\cdot d^2 + 1{,}22\cdot10^{-8}\cdot d^3$$

Vanwege het feit dat er geen invloed is op p_n, kan het E-MOS algoritme gebruik maken van de onderstaande functie.

$$Q(d) \approx 4{,}10 - 0{,}195\cdot p_n + 19{,}5\left({}^{k}/_{d}\right)^{\alpha} + 2{,}64\cdot10^{-3}\cdot d - 1{,}86\cdot10^{-5}\cdot d^2 + 1{,}22\cdot10^{-8}\cdot d^3$$

---

[11] Beide functies leveren een MOS-waarde op volgens de ACR-schaal.
[12] Zie paragraaf 4.4

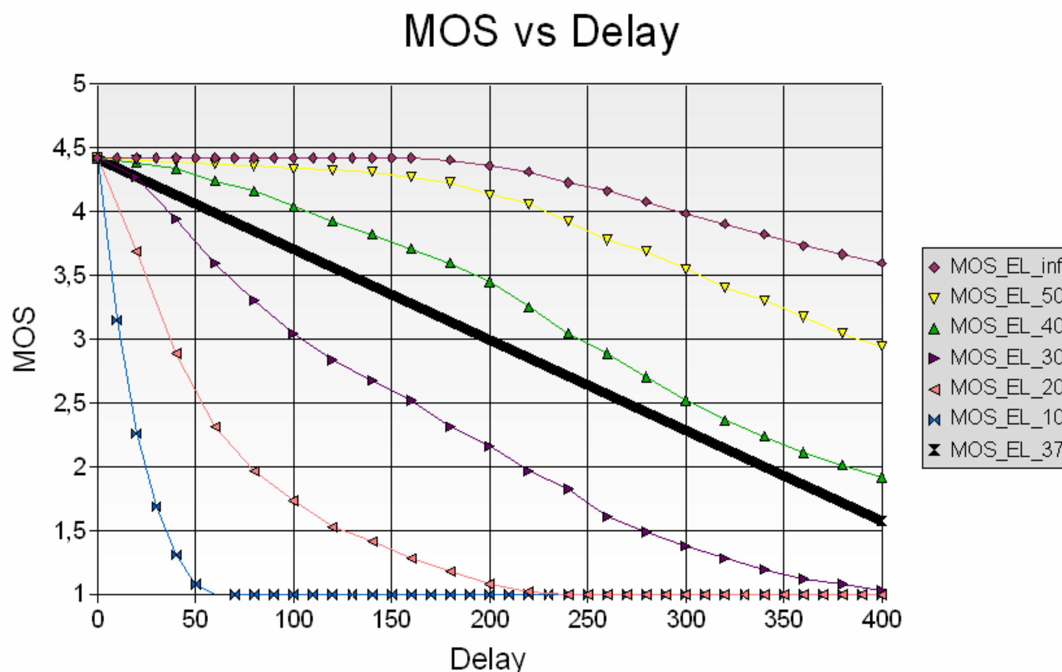
Door maximalisatie van Q(d) kan de optimale delay berekend worden. Dit gebeurt door het nulpunt te vinden van Q′(d). En kan de lengte van de jitterbuffer aangepast worden aan deze delay. Om dit te bereiken moeten de volgende stappen ondernomen worden:

1. Meet de relatieve delays van de ontvangen pakketten. Dit kan alleen als de twee computerklokken exact gelijk lopen.
2. Bereken de parameters van de Pareto distributie (k, α).
3. Voeg de waardes (k, α) in de functie Q(d) toe.
4. Bereken voor welke waarde van d geldt: Q′(d)==0, door gebruik te maken van de false position method.[13]
5. Bereken de lengte van de jitterbuffer voor de gevonden waarde van d.
6. Ga terug naar stap 1.

Fujimoto, Ata en Murata hebben in hun artikel al enkele punten aangegeven die aanleiding geven voor verder onderzoek. Een voorbeeld hiervan is een nauwkeuriger model voor het weergeven van de delay distributies en een effectievere berekening van het algoritme.

6.2. Echo en delay

Ik ben tot de conclusie gekomen dat Fujimoto, Ata en Murata de aanname hebben gemaakt dat er geen echo optreedt in een gesprek.[3,7] Ik heb de aanname gemaakt dat er wel echo optreedt in de realiteit. Hierbij ga ik uit van een demping tussen de 35dB en 40dB.[1] Dit houdt in dat er echo demping optreedt maar dat deze niet perfect is. In figuur 17 is een inschatting gemaakt welke invloed de delay heeft op de gesprekskwaliteit bij een demping van ongeveer 37dB.



Figuur 17. De invloed van echo en delay op de gesprekskwaliteit. [7]

[13] Voor meer informatie over de false position method zie [20].

Er zijn twee redenen om te kiezen voor een demping van ongeveer 37dB. Ten eerste wordt er rekening gehouden met niet perfecte echo demping. Ten tweede kan, door interpolatie tussen de gegevens van 30 en 40dB demping, een lineair verband worden verondersteld.
Later zal blijken dat dit het algoritme aanzienlijk vereenvoudigt. De afname van de MOS-waarde, voor codec G.711, bij een demping van ongeveer 37dB is weergegeven in de functie M(d). Deze functie levert de MOS-waarde volgens de ITU-schaal.

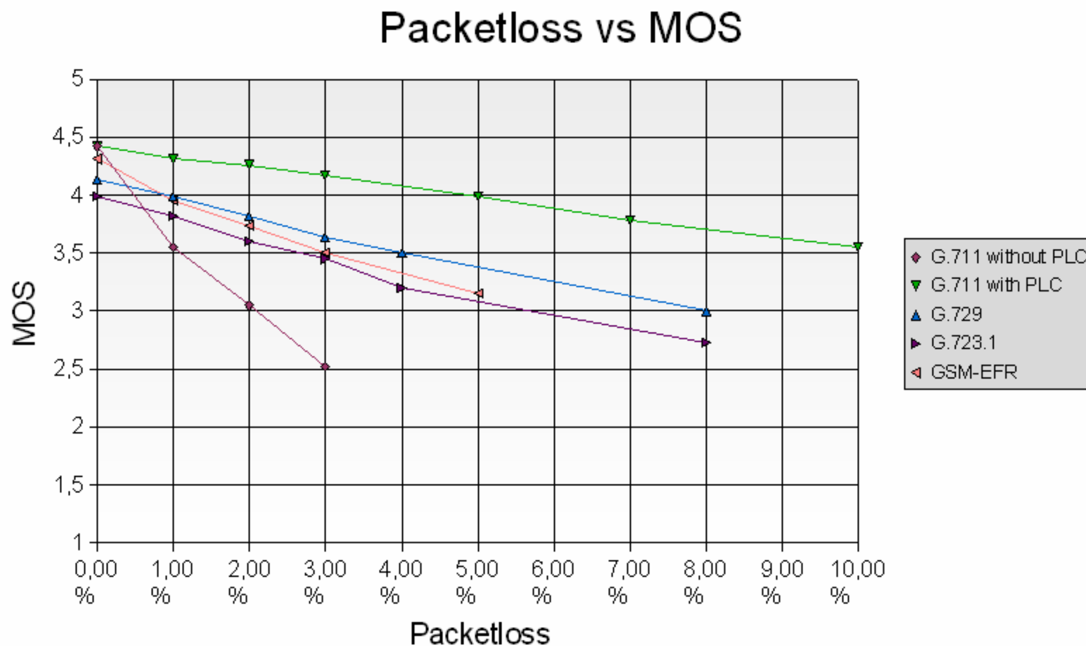
$$M(d) \approx 4{,}42 - 0{,}0071 \cdot d$$

Deze functie is alleen voor codec G.711 afgeleid. Ik heb de aanname gemaakt dat de voor alle codec's de relatie M(d) ↔ d gelijk is aan die bij codec G.711. De functie M(d) kan nu voor alle codec's als volgt geschreven worden:

$$M(d) \approx m - 0{,}0071 \cdot d \qquad ; \; m \text{ is afhankelijk van gebruikte codec}$$

6.3. Packetloss

Zoals in paragrafen 4.4 en 4.5 reeds beschreven staat, is de invloed van packetloss afhankelijk van de gebruikte codec, zoals weergegeven in figuur 9. In figuur 18 is deze invloed op de MOS-waarde per codec nogmaals gegeven.



Figuur 18. Per codec de invloed van packetloss op de MOS-waarde (ITU-schaal).

Wat opvalt in figuur 18 is dat voor alle codec's de gesprekskwaliteit, MOS, redelijk lineair afneemt bij toename van de packetloss. Ik heb de volgende functies voor de MOS-waarde (ITU-schaal) opgesteld:

$M_{G.711}(p) \approx 4{,}42 - 0{,}63 \cdot p$
$M_{G.711_PLC}(p) \approx 4{,}42 - 0{,}087 \cdot p$
$M_{G.723.1}(p) \approx 3{,}99 - 0{,}16 \cdot p$
$M_{G.729(A)}(p) \approx 4{,}13 - 0{,}14 \cdot p$
$M_{GSM\text{-}EFR}(p) \approx 4{,}31 - 0{,}23 \cdot p$

6.4. Nieuwe optimalisatie functies

Als de functies uit paragraaf 6.2 en 6.3 worden gecombineerd, geeft dit het volgende resultaat:

$M_{G.711}(d,p) \approx 4{,}42 - 0{,}63 \cdot p - 0{,}0071 \cdot d$
$M_{G.711_PLC}(d,p) \approx 4{,}42 - 0{,}087 \cdot p - 0{,}0071 \cdot d$
$M_{G.723.1}(d,p) \approx 3{,}99 - 0{,}16 \cdot p - 0{,}0071 \cdot d$
$M_{G.729(A)}(d,p) \approx 4{,}13 - 0{,}14 \cdot p - 0{,}0071 \cdot d$
$M_{GSM\text{-}EFR}(d,p) \approx 4{,}31 - 0{,}23 \cdot p - 0{,}0071 \cdot d$

Er kan alleen invloed worden uitgeoefend op de delay veroorzaakt door de jitterbuffer. Op de transmissie delay hebben we geen invloed. Daarom kan de delay geschreven worden als.

$$d = d_n + d_{jb}$$

Packetloss p is onder te verdelen in p_n en p_{jb}. p_{jb} is echter afhankelijk van de delay veroorzaakt door de jitterbuffer, d_{jb}. Deze afhankelijkheid wordt weergegeven met $f(d_{jb})$.

$$M_{G.711}(d_n,d_{jb},p_n) \approx 4{,}42 - 0{,}63 \cdot p_n - 0{,}63 \cdot f(d_{jb}) - 0{,}0071 \cdot d_n - 0{,}0071 \cdot d_{jb}$$

$$M_{G.711_PLC}(d_n,d_{jb},p_n) \approx 4{,}42 - 0{,}087 \cdot p_n - 0{,}087 \cdot f(d_{jb}) - 0{,}0071 \cdot d_n - 0{,}0071 \cdot d_{jb}$$

$$M_{G.723.1}(d_n,d_{jb},p_n) \approx 3{,}99 - 0{,}16 \cdot p_n - 0{,}16 \cdot f(d_{jb}) - 0{,}0071 \cdot d_n - 0{,}0071 \cdot d_{jb}$$

$$M_{G.729(A)}(d_n,d_{jb},p_n) \approx 4{,}13 - 0{,}14 \cdot p_n - 0{,}14 \cdot f(d_{jb}) - 0{,}0071 \cdot d_n - 0{,}0071 \cdot d_{jb}$$

$$M_{GSM\text{-}EFR}(d_n,d_{jb},p_n) \approx 4{,}31 - 0{,}23 \cdot p_n - 0{,}23 \cdot f(d_{jb}) - 0{,}0071 \cdot d_n - 0{,}0071 \cdot d_{jb}$$

Doordat eerder (paragraaf 6.2) de invloed van de delay lineair verondersteld is, hoeven we geen rekening te houden met de transmissie delay (d_n) bij het berekenen van de optimale delay. Tevens is er geen invloed uit te oefenen op p_n en hoeft deze ook niet

gebruikt te worden bij optimalisatie. Hierdoor kunnen de volgende functies gebruikt worden voor het berekenen van de optimale delay.

$$Q_{G.711}(d_{jb}) \approx 4,42 - 0,63 \cdot p_n - 0,63 \cdot f(d_{jb}) - 0,0071 \cdot d_n - 0,0071 \cdot d_{jb}$$

$$Q_{G.711_PLC}(d_{jb}) \approx 4,42 - 0,087 \cdot p_n - 0,087 \cdot f(d_{jb}) - 0,0071 \cdot d_n - 0,0071 \cdot d_{jb}$$

$$Q_{G.723.1}(d_{jb}) \approx 3,99 - 0,16 \cdot p_n - 0,16 \cdot f(d_{jb}) - 0,0071 \cdot d_n - 0,0071 \cdot d_{jb}$$

$$Q_{G.729(A)}(d_{jb}) \approx 4,13 - 0,14 \cdot p_n - 0,14 \cdot f(d_{jb}) - 0,0071 \cdot d_n - 0,0071 \cdot d_{jb}$$

$$Q_{GSM-EFR}(d_{jb}) \approx 4,31 - 0,23 \cdot p_n - 0,23 \cdot f(d_{jb}) - 0,0071 \cdot d_n - 0,0071 \cdot d_{jb}$$

6.5. *Verbeterd algoritme*

Bij het berekenen van de optimale delay kan er nu gebruik gemaakt worden van de in paragraaf 6.4 voorgestelde functies. Deze zijn nauwkeuriger dan de functie die Fujimoto, Ata en Murata hadden voorgesteld in het E-MOS algoritme.

In het E-MOS algoritme worden aan de hand van de geschiedenis van de delay van de ontvangen pakketten, de waarden van de Pareto-distributie berekend. Deze waarden worden daarna gebruikt om de delay distributie te modelleren. Vervolgens wordt met dit model de ‘ideale’ delay berekend.
Het voordeel hiervan is dat er een continue functie $Q(d_{jb})$ ontstaat, die gemaximaliseerd kan worden en dat de ingestelde jitterbuffer lengte ook de optimale lengte is.
Deze aanpak heeft verschillende nadelen. Zo is er rekenkracht nodig voor het herberekenen van de waardes van de Pareto-distributie bij elk nieuw ontvangen pakket. Tevens gaven Fujimoto, Ata en Murata aan dat er verder onderzoek nodig is naar een beter model om de delay distributie weer te geven.

Volgens Fujimoto, Ata en Murata modelleert de Pareto-distributie het beste de delay. De waardes van de Pareto-distributie zijn gebaseerd op de discrete gegevens in de geschiedenis van de delay.[5] Zo kan er een afweging tussen packetloss en delay gemaakt worden.

In Bijlagen C en D is een implementatie van een jitterbuffer die gebruik maakt van mijn algoritme. Hierbij is in plaats van een continue functie, een discrete functie gebruikt om de delay distributie weer te geven. Delay en packetloss worden bepaald door onderstaande methode.

`history_length` is grootte van `history[]` en `history_sorted[]`

`history[]` is de geschiedenis van de relatieve delays.[14] Gesorteerd op moment van ontvangst.
`history_sorted[]` is de geschiedenis van de relatieve delays. Gesorteerd van klein naar groot.

d_{jb} = history_sorted[history_length - i] - history_sorted[0]
p_{jb} = f(d_{jb}) = 100·i/history_length

Bij de berekening van d_{jb} wordt `history_sorted[history_length - i]` verminderd met `history_sorted[0]` zodat eventuele verschillen in computerklokken tussen verzender en ontvanger geen invloed hebben op berekening 'ideale' delay.
p_{jb} is nu het percentage van de geschiedenis van de ontvangen pakketten wat verloren zou gaan bij deze d_{jb}.

Omdat er nu gebruik wordt gemaakt van een discrete functie kan false position method niet worden toegepast. Daardoor wordt de onderstaande methode gebruikt om de 'ideale' delay te vinden.

```
1. WHILE (i<25%·history_length) {
2.   djb = history_sorted[history_length - i] -
       history_sorted[0];
3.   pjb = 100·i/history_length
4.   MOS = CALCULATE_MOS(pjb, djb, current_codec);
5.   IF (MOS > HIGHEST_MOS) {
6.     HIGHEST_MOS = MOS;
7.     IDEAL_DELAY = djb;
8.   }
9. }
```

`CALCULATE_MOS` maakt gebruik van de functies gedefinieerd in paragraaf 6.4.
`MOS`, `HIGHEST_MOS` en `IDEAL_DELAY` zijn tijdelijke variabele. Aan het einde van de methode levert `IDEAL_DELAY` de 'ideale' delay.

De stappen om de 'ideale' delay te vinden worden nu:

1. Bereken relatieve delay van ontvangen pakket.
2. *Mochten de verzamelingen `history[]` en `history_sorted[]` vol zitten, verwijder dan `history[0]` uit verzamelingen `history[]` en `history_sorted[]`.*
3. Voeg relatieve delay van ontvangen pakket toe aan `history[]` en `history_sorted[]`
4. Vind de 'ideale' delay door bovenstaande methode toe te passen.
5. Verander de lengte van de jitterbuffer.
6. Ga terug naar stap 1.

[14] Met relatieve delay wordt bedoeld het verschil tussen ontvangsttijd en zendtijd

6.6. Extra delay

De optimale delay zoals berekend in paragraaf 6.5 kan helaas niet altijd gebruikt worden als lengte voor de jitterbuffer. Groeien van de lengte is geen probleem. Bij groeien kan er voor een paar milliseconden geen audio worden afgespeeld: stilte. Ook is het mogelijk om een PLC algoritme in werking te laten treden, mits dit beschikbaar is. Hiermee is er een groei mogelijk met de lengte van een pakket.
Het probleem zit in het krimpen. Krimpen met enkele milliseconden is niet mogelijk omdat de jitterbuffer niet het aantal milliseconden die het pakket duurt, kan veranderen. Krimpen kan alleen door het verwijderen van een volledig pakket.

Daardoor moet er rekening gehouden worden met een extra delay. Deze extra delay is minimaal de lengte van een pakket.

7. Verbetering van gesprekskwaliteit door codecswitch

Uit het onderzoek van Robustelli, Loreto, Fresa, Longo en Spinelli [8], wat beschreven staat in paragraaf 5.4, is gebleken dat het veranderen van de gebruikte codec tot een hogere gesprekskwaliteit kan leiden. In mijn onderzoek heb ik naar codecswitchen gekeken en is er een prototype ontworpen waarmee handmatig codecswitchen mogelijk was.
Dit biedt mogelijkheden voor enkele aannames, aanbevelingen en aandachtspunten voor toekomstig onderzoek, die hieronder staan beschreven.

Algoritme
Robustelli, Loreto, Fresa, Longo en Spinelli [8] hebben aangetoond dat een codecswitch tot een hogere gesprekskwaliteit kan leiden. Bij het ontwerp van een algoritme voor het switchen van codec's kunnen de formules uit paragraaf 6.4 goed van pas komen.

Extra delay
Sommige codec's hebben een extra delay aan de zendende kant, omdat de technieken die deze codec's wapenen tegen packetloss extra informatie nodig hebben.[11]

Codecswitch voorwaarden
Additioneel onderzoek naar de grenzen waarop een codecswitch plaatsvindt is aan te bevelen. Dit om te voorkomen dat er onnodig tussen codec's geswitched wordt.[8]

Audiokwaliteit
Tijdens het testen van mijn prototype trad er incidenteel gekraak op wanneer een codecswitch plaatsvond. Het niet goed aansluiten van de audiosignalen die de codec's aanleverden kan de oorzaak zijn. Verder onderzoek naar de oorzaak en eventuele impact op de gesprekskwaliteit is aan te bevelen

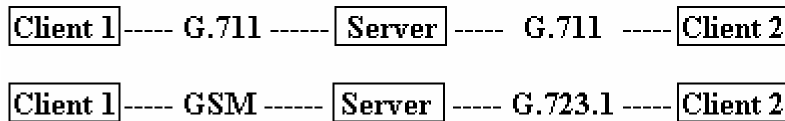
Packetloss
Bij het vergelijken van verschillende codec's is het verstandig om rekening te houden met de volledige packetloss, omdat deze packetloss de gesprekkwaliteit beïnvloedt.

Netwerkomstandigheden
Eén van de redenen voor een codecswitch is, dat onder dezelfde netwerkomstandigheden (delay en packetloss) een andere codec beter presteert. Er zou goed naar de mogelijkheid gekeken kunnen worden dat, door het veranderen van de codec, de netwerkomstandigheden beïnvloed worden. Er treedt bijvoorbeeld minder packetloss op doordat een lagere bandbreedte gebruikt wordt. Waardoor de vorige codec betere kwaliteit lijkt te leveren. Het systeem gaat 'dansen'.

Type audiosignaal
In sommige gevallen, zoals bij het verzenden van een fax kunnen bepaalde codec's niet gebruikt worden. Wanneer er toch gebruik wordt gemaakt van deze codec's komt de fax niet aan. Het is aan te bevelen om te controleren wat voor soort audiosignaal er verzonden wordt.

Transcoding[15]

Door transcoding kan er extra verlies van de gesprekskwaliteit optreden. Dit is weergegeven in figuur 19. In het bovenste deel van het figuur wordt de audio van client 1 doorgestuurd naar client 2 zonder interferentie van de server. Dit komt doordat beide clients codec G.711 begrijpen. In het onderste deel wordt de audio gecodeerd met GSM tussen client 1 en de server en tussen client 2 en de server met G.723.1. Er vindt dan transcoding tussen GSM en G.723.1 plaats door de server. Hiermee treedt er extra verlies van kwaliteit op, waar rekening mee moet worden gehouden.



Figuur 19. Transcoding tussen GSM en G.723.1.

Comptabiliteit

Niet alle protocollen ondersteunen een codecswitch. Als er ondersteuning is, is de vraag of deze in alle implementaties te vinden is. Eventuele detectie van ondersteuning is wellicht nodig.

[15] Transcoding: digitale conversie tussen twee codecs zonder het omrekenen naar analoge vorm.[Q]

8. Conclusies en Aanbevelingen

De kwaliteit van een VoIP-gesprek is op verschillende manieren beïnvloedbaar.
In dit onderzoek zijn verschillende oplossingen beschreven voor kwaliteitsverbetering.
De beste kwaliteit is te bereiken wanneer een combinatie van alle oplossingen in onderlinge afstemming worden geoptimaliseerd.

De oplossingen zoals beschreven in hoofdstuk 5 zijn in twee groepen te verdelen. De eerste groep bestaat uit: keuze van codec, echo demping, PLC en packetsize. De mogelijkheden tot verbeteringen in deze groep lijken voorlopig gering.
Verbeteringen zijn vooral in de tweede groep oplossingen: codecswitchen en met name de jitterbuffer.

In paragraaf 6.5 heb ik in een nieuw algoritme verbeteringen aangebracht ten opzichte van zijn voorganger, het E-MOS algoritme. Een implementatie van een jitterbuffer met dit algoritme is te vinden in bijlagen C en D.

Mijn algoritme verbetert het E-MOS algoritme op verschillende punten. Ten eerste is per codec het gedrag van de jitterbuffer verschillend, om zo de gesprekskwaliteit te optimaliseren voor die codec. Om dit te bereiken heb ik voor verschillende codec's functies gedefinieerd.
Ten tweede wordt er geen model gebruikt om de delay distributie als continue functie weer te geven maar worden de (discrete) gegevens van de ontvangen pakketten gebruikt.
Hiermee wordt de lastige discussie welk model de gegevens van de ontvangen pakketten het best weergeeft vermeden.
Tevens is de benodigde rekenkracht laag, hierdoor is mijn jitterbuffer toepasbaar in systemen waar rekenkracht schaars is.

Codecswitchen biedt eveneens veel potentie voor kwaliteitsverbetering. In hoofdstuk 7 staan verschillende aanbevelingen en aandachtspunten voor verder onderzoek naar codecswitchen.

De tijd en middelen waren niet aanwezig om het voorgestelde algoritme voor de jitterbuffer te vergelijken met andere algoritmes zoals het E-MOS algoritme. Dit is wel aan te bevelen.
De functies zoals beschreven in paragraaf 6.4 bieden ruimte voor verdere optimalisatie.
De voorlopige aanname dat er in een gesprek 37dB echo demping optreedt dient wellicht nader te worden beschouwd.
Tevens kan verder onderzoek naar de manier waarop de jitterbuffer zijn optimale lengte bereikt nog extra kwaliteitswinst opleveren.

Bronvermelding / literatuurlijst

[1] O. Hersent, J.P. Petit en D. Gurle, “Beyond VoIP Protocols,” Wiley, 2005. ISBN 0-470-02362-7.

[2] D. Minoli en E. Minoli, “Delivering Voice over IP Networks,” Wiley, 2002. ISBN: 0-471-38606-5.

[3] K. Fujimoto, S. Ata en M. Murata, “Adaptive Playout Buffer Algorithm for Enhancing Perceived Quality of Streaming Applications,” Globecom '02. IEEE, p.2451-p.2457 vol.3, November 2002.

[4] K. Fujimoto, S. Ata en M. Murata, “Statistical Ananlysis of Packet delays in the Internet and Its Application to Playout Cotrol for Streaming Applications,” IEICE Transactions on Communications, Vol. E00-B, no. 6, juni 2001.

[5] V. Brazauskas en R. Serfling, “Robust and Efficient Estimation of the Tail Index of a Single-Parameter Pareto Distribution,” April 2000. http://www.utdallas.edu/~serfling/

[6] B. Goode, “Voice Over Internet Protocol (VoIP),” Proceedings of the IEEE, vol. 90, no. 9, September 2002.

[7] J. Jansen, D de Vleeschauwer, M. Büchli en G.H. Petit, “Assessing Voice Quality in Packet-Based Telephony,” IEEE Internet Computing vol. 6, no 3. p.48-p.57, Mei 2002.

[8] A.L. Robustelli, S.Loreto, A. Fresa, M. Longo en D. Spinelli, “Prototype of an Adaptive Voice Coder for IP Telephony,” datum onbekend. http://www.coritel.it/publications/IP_download/protoype%20of%20an%20adaptive%20voice%20coder%20for%20ip%20telephony.pdf

[9] Telchemy, “Voice Quality Measurement,” Januari 2005. http://www.telchemy.com/appnotes/TelchemyVoiceQualityMeasurement.pdf

[10] M.J. Karam en F.A. Tobagi, “Analysis of the Delay and jitter of Vocie Traffic Over the Internet,” Infocom 2001, vol2. p824-p.833, 2001.

[11] S. Brunner en A.A. Ali, “Voice Over IP 101, Understanding VoIP Networks,” Juniper Networks White Paper, Augustus 2004. http://www.juniper.net/solutions/literature/white_papers/200087.pdf

[12] ETSI, “Telecommunications and Internet Protocol Harmonization Over Networks (TIPHON) Release 2,” ETSI TR 101 329-6 v2.1.1, Februari 2002.

[13] W. Jiang en H. Schulzrinne, “Comparisons of FEC and codec robustness on VoIP quality and bandwidth efficiency,” ICN, Augustus 2002.

[14] A. Barberis, C. Casetti, J.C. De Martin en M. Meo, "A simulation study of adaptive voice communications on IP networks," Computer Communications 24, p.757-p.767, November 2000.

[15] R.J.B. Reynolds en A.W. Rix, "Quality VoIP – an engineering challenge," BT Technol J Vol. 19 No 2, April 2001.

[16] ITU-T Recommendation P.862, "Perceptual evaluation of speech quality (PESQ): An objective method for end-to-end speech quality assessment of narrow-band telephone networks and speech codecs ," Februari 2001.

[17] ITU-T Recommendation G.711, "Pulse code modulation (PCM) of voice frequencies," November 1988.

[18] ITU-T Recommendation G.107, "The E-model, a computational model for use in transmission planning," Maart 2005.

[19] ITU-T Recommendation P.800.1, "Mean Opinion Score (MOS) terminology," maart 2003.

Internet bronnen:
[A] http://www.commweb.com/howto/showArticle.jhtml?articleId=159400968&pgno=3 (MOS)
[B] http://portal.etsi.org/stq/presentations/emodel.asp
[C] http://www.bbc.co.uk/history/historic_figures/bell_alexander.shtml
[D] http://www.kpn.com
[E] http://www.intertangent.com/023346/Articles_and_News/1413.html (VoIP hist)
[F] http://www.connect802.com/encyclopedia/enc-m.htm
[G] http://www.pesq.org/
[H] http://www.voiptroubleshooter.com/problems/plc.html
[I] http://www.iarchive.com/_library/terminology/c.htm
[J] http://www.tomsnetworking.com/Sections-article94-page3.php
[K] http://en.wikipedia.org/wiki/Timeline_of_the_telephone
[L] http://en.wikipedia.org/wiki/Automatic_telephone_exchange
[M] http://en.wikipedia.org/wiki/Circuit_switching
[N] http://en.wikipedia.org/wiki/Jitter
[O] http://nl.wikipedia.org/wiki/Codec
[P] http://en.wikipedia.org/wiki/Packet
[Q] http://en.wikipedia.org/wiki/Transcoding
[R] http://nl.wikipedia.org/wiki/Gateway
[S] http://en.wikipedia.org/wiki/Pareto_distribution

Bijlage A: Opdrachtomschrijving

Name: Jesse Kaijen
Title: Quality control of Voice-over-IP
Cluster: DACS/ASNA
Supervisors: Florian Overkamp (SpeakUp) , Remco van de Meent (SpeakUp/UT), Aiko Pras (UT)

Period: February-June 2005

External assignment at SpeakUp, Enschede

Description:

Voice-over-IP ("VoIP") technology is increasingly getting attention as an alternative to the ordinary telephone service that is currently dominating the telephony market for both business as well as individuals. In VoIP, telephony data is sent through the Internet between VoIP applications (e.g., a PC running a piece of software, or a VoIP phone).

It is of crucial importance to broad adaptation of VoIP technology that the quality of VoIP connections -as perceived by the user- is sufficient. Quality in this respect refers to availability and quality of speech. The quality of VoIP is influenced by:

- the equipment.
- the VoIP protocol, for instance choice of codecs.
- the underlying technology, such as: the IP packet, available bandwidth, latency and delay variation.

In this assignment, the student will assess the quality of VoIP (primarily as it is perceived by users) under different (realistic) circumstances, including (but not limited to): various codecs, different networking environment (high/low latency, high/low available bandwidth, etc.) and various types of telephony equipment. The focus is on:

- Realistic home and small to middle-sized business networking environments (say DSL up to 100 Mbps Ethernet connections).
- Inter-carrier VoIP traffic (e.g. ENUM and peering traffic).

The student will also investigate whether the use of Quality of Service technology will further stretch the `usability' of VoIP, especially during times when resources are scarce (e.g. DiffServ, IntServ or 802.1p/q VLAN based QoS).

The following approach is proposed for this assignment:
Literature study, focusing on VoIP technology, Quality of Service and perceived quality for speech.
Various experiments carried out (following the outline set above) to get quantitative figures on the `usability' of VoIP.
Recommendations given on (especially) the minimal requirements for networks/links to support VoIP.

SpeakUp will provide the student with the equipment necessary to perform the above tasks.

Bijlage B: Bandbreedte test

In de beginfase van het onderzoek is gekeken of het switchen naar een lagere of hogere bitrate invloed heeft op de ontvangst van pakketten. In deze verkennende test werd een VoIP gesprek geïmiteerd, door om de 20ms een pakket te versturen. Het pakket bevatte zijn volgnummer, de verzendtijd en werd voor de rest gevuld met willekeurige data. Deze opvulling is om te voorkomen dat compressietechnieken toegepast kunnen worden. Tevens werd aan elk pakket 32 bits toegevoegd, die de header informatie van het protocol voorstelde. Bij ontvangst van het pakket werd de ontvangsttijd genoteerd.

De pakketten hadden de volgende groottes, naar boven afgerond op hele bytes:
((8 kbit/s)/20ms + 32)/8 = 25 bytes
((16 kbit/s)/20ms + 32)/8 = 45 bytes
((32 kbit/s)/20ms + 32)/8 = 86 bytes
((64 kbit/s)/20ms + 32)/8 = 168 bytes
((128 kbit/s)/20ms + 32)/8 = 332 bytes
((256 kbit/s)/20ms + 32)/8 = 660 bytes

De test bestond uit het om de 20 ms voor de periode van vijf seconden werd een pakket verstuurd met lengte Y. Na deze periode van 5 seconden werd een andere lengte Y verstuurd. Deze werd ook voor een periode van vijf seconden om de 20 ms verstuurd. Voor lengte Y deze volgorde gebruikt: 25, 45, 86, 168, 332, 660, 660, 332, 168, 86, 45 en 25 bytes.

De test is gedaan over een 14k4 modem verbinding. Hierdoor was het zeker, dat op sommige momenten de benodigde bandbreedte hoger was dan de beschikbare bandbreedte.

Uit de resultaten konden enkele conclusies getrokken worden. Als de verbinding (zwaar) overbelast raakte werd het verschil tussen ontvangst- en verzendtijd steeds groter en trad er meer jitter op. Tevens was bij overbelasting meer packetloss waar te nemen. Dit is ongewenst gedrag bij een VoIP-gesprek.
Er kwam niet naar voren dat het verzenden van een lagere bitrate tot minder jitter leidde wanneer de verbinding niet overbelast was.

.

Bijlage C: Jitterbuffer headerfile

```
/*******************************************************
 * jitterbuffer:
 * an application-independent jitterbuffer, which tries
 * to achieve the maximum user perception during a call.
 * For more information look at:
 * http://www.speakup.nl/opensource/jitterbuffer/
 *
 * Copyright on this file is held by:
 * - Jesse Kaijen <jesse@speakup.nl>
 * - SpeakUp <info@speakup.nl>
 *
 * Contributors:
 * Jesse Kaijen <jesse@speakup.nl>
 *
 * This program is free software, distributed under the terms of:
 * - the GNU Lesser (Library) General Public License
 * - the Mozilla Public License
 *
 * if you are interested in an different licence type, please contact us.
 *
 * How to use the jitterbuffer, please look at the comments
 * in the headerfile.
 *
 * Further details on specific implementations,
 * please look at the comments in the code file.
 */

#ifndef _JITTERBUFFER_H_
#define _JITTERBUFFER_H_

#ifdef __cplusplus
extern "C" {
#endif

/***********
 * The header file consists of four parts.
 * - configuration constants, structs and parameter definitions
 * - functions
 * - How to use the jitterbuffer and
 *   which responsibilities do YOU have
 * - debug messages explained
 */


// configuration constants
/* Number of historical timestamps to use in calculating jitter and
jitterbuffer size */ #define JB_HISTORY_SIZE 500
/* minimum jitterbuffer size, disabled if 0 */ #define JB_MIN_SIZE 0
/* maximum jitterbuffer size, disabled if 0 */ #define JB_MAX_SIZE 0
 /* maximum successive interpolating frames, disabled if 0 */ #define
JB_MAX_SUCCESSIVE_INTERP 0
/* amount of extra delay allowed before shrinking */
```

```
#define JB_ALLOW_EXTRA_DELAY 30


/* ms between growing */
#define JB_WAIT_GROW 60
/* ms between shrinking */
#define JB_WAIT_SHRINK 250
/* ms that the JB max may be off */
#define JB_MAX_DIFF 6000 //in a RTP stream the max_diff may be 3000
packets (most packets are 20ms)

//structs
typedef struct jb_info {
      long frames_received;       /* Number of frames received by the
jitterbuffer */
  long frames_late;           /* Number of frames that were late */
  long frames_lost;           /* Number of frames that were lost */
  long frames_ooo;            /* Number of frames that were Out Of
Order */
  long frames_dropped;        /* Number of frames that were dropped due
shrinkage of the jitterbuffer */
  long frames_dropped_twice;  /* Number of frames that were dropped
because this timestamp was already in the jitterbuffer */

  long delay;     /* Current delay due the jitterbuffer */
      long jitter;            /* jitter measured within current history
interval*/
      long losspct;     /* recent lost frame percentage (network and
jitterbuffer loss) */

      long delay_target;   /* The delay where we want to grow to */
      long losspct_jb;     /* recent lost percentage due the
jitterbuffer */
      long last_voice_ms;      /* the duration of the last voice frame
*/
  short silence;       /* If we are in silence 1-yes 0-no */
  long iqr;            /* Inter Quartile Range of current history, if
the squareroot is taken it is a good estimate of jitter */
} jb_info;

typedef struct jb_frame {
      void *data;                       /* the frame data */
      long ts;                        /* the senders timestamp */
      long ms;                        /* length of this frame in ms */
      int  type;                    /* the type of frame */
      int codec;                    /* codec of this frame, undefined
if nonvoice */
      struct jb_frame *next, *prev; /* pointers to the next and
previous frames in the queue */ } jb_frame;

typedef struct jb_hist_element {
      long delay; /* difference between time of arrival and senders
timestamp */
      long ts;    /* senders timestamp */
      long ms;    /* length of this frame in ms */
      int codec;  /* wich codec this frame has */
} jb_hist_element;
```

```
typedef struct jb_settings {
  /* settings */
      long min_jb;                    /* defines a hard clamp to use in
setting the jitterbuffer delay */
  long max_jb;                  /* defines a hard clamp to use in
setting the jitterbuffer delay */
  long max_successive_interp; /* the maximum count of successive
interpolations before assuming silence */
  long extra_delay;           /* amount of extra delay allowed before
shrinking */
  long wait_grow;             /* ms between growing */
  long wait_shrink;           /* ms between shrinking */
  long max_diff;              /* maximum number of milliseconds the
jitterbuffer may be off */
} jb_settings;

typedef struct jitterbuffer {
      struct jb_hist_element hist[JB_HISTORY_SIZE]; /* the history of
the last received frames */
      long hist_sorted_delay[JB_HISTORY_SIZE];      /* a sorted buffer
of the delays (lowest first) */
      long hist_sorted_timestamp[JB_HISTORY_SIZE];  /* a sorted buffer
of the timestamps (lowest first) */

      int  hist_pointer;          /* points to index in history for
next entry */
      long last_adjustment;       /* the time of the last adjustment
(growing or shrinking) */
  long next_voice_time;       /* the next ts is to be read from the jb
(senders timestamp) */
      long cnt_successive_interp; /* the count of consecutive
interpolation frames */
      long silence_begin_ts;      /* the time of the last CNG frame,
when in silence */
      long min;                         /* the clock difference within
current history interval */
      long current;                     /* the present jitterbuffer
adjustment */
      long target;                        /* the target jitterbuffer
adjustment */
      long last_delay;            /* the delay of the last packet, used
for calc. jitter */

      jb_frame *voiceframes;   /* queued voiceframes */
      jb_frame *controlframes; /* queued controlframes */
      jb_settings settings;    /* the settings of the jitterbuffer */
      jb_info info;            /* the statistics of the jitterbuffer */
} jitterbuffer;

//parameter definitions
/* return codes */
#define JB_OK           0
#define JB_EMPTY  1
#define JB_NOFRAME      2
#define JB_INTERP 3
```

```
/* frame types */
#define JB_TYPE_CONTROL 1
#define JB_TYPE_VOICE   2
#define JB_TYPE_SILENCE 3

/* the jitterbuffer behaives different for each codec. */
/* Look in the code if a codec has his function defined */
/* default is g711x behaiviour */
#define JB_CODEC_SPEEX 10        //NOT defined
#define JB_CODEC_ILBC 9          //NOT defined
#define JB_CODEC_GSM_EFR 8
#define JB_CODEC_GSM_FR 7        //NOT defined
#define JB_CODEC_G723_1 6
#define JB_CODEC_G729A 5
#define JB_CODEC_G729 4
#define JB_CODEC_G711x_PLC 3
#define JB_CODEC_G711x 2
#define JB_CODEC_OTHER 1         //NOT defined


/*
 * Creates a new jitterbuffer and sets the default settings.
 * Always use this function for creating a new jitterbuffer.
 */
jitterbuffer *jb_new();

/*
 * The control frames and possible personal settings are kept.
 * History and voice/silence frames are destroyed.
 */
void jb_reset(jitterbuffer *jb);

/*
 * Resets the jitterbuffer totally, all the control/voice/silence
frames are destroyed
 * default settings are put as well.
 */
void jb_reset_all(jitterbuffer *jb);

/*
 * Destroy the jitterbuffer and any frame within.
 * Always use this function for destroying a jitterbuffer,
 * otherwise there is a chance of memory leaking.
 */
void jb_destroy(jitterbuffer *jb);

/*
 * Define your own settings for the jitterbuffer. Only settings !=0
 * are put in the jitterbuffer.
 */
void jb_set_settings(jitterbuffer *jb, jb_settings *settings);

/*
 * Get the statistics for the jitterbuffer.
 * Copying the statistics directly for the jitterbuffer won't work
because
 * The statistics are only calculated when calling this function.
```

```
 */
void jb_get_info(jitterbuffer *jb, jb_info *stats);

/*
 * Get the current settings of the jitterbuffer.
 */
void jb_get_settings(jitterbuffer *jb, jb_settings *settings);

/*
 * Gives an estimation of the MOS of a call given the
 * packetloss p, delay d, and wich codec is used.
 * The assumption is made that the echo cancelation is around 37dB.
 */
float jb_guess_mos(float p, long d, int codec);

/*
 * returns JB_OK if there are still frames left in the jitterbuffer
 * otherwise JB_EMPTY is returned.
 */
int jb_has_frames(jitterbuffer *jb);

/*
 * put a packet(frame) into the jitterbuffer.
 * *data - points to the packet
 * type - type of packet, JB_CONTROL|JB_VOICE|JB_SILENCE
 * ms - duration of frame (only voice)
 * ts - timestamp sender
 * now - current timestamp (timestamp of arrival)
 * codec - which codec the frame holds (only voice), if not defined,
g711x will be used
 *
 * if type==control @REQUIRE: *data, type, ts, now
 * if type==voice   @REQUIRE: *data, type, ms, ts, now @OPTIONAL: codec
 * if type==silence @REQUIRE: *data, type, ts, now
 * on return *data is undefined
 */
void jb_put(jitterbuffer *jb, void *data, int type, long ms, long ts,
long now, int codec);

/*
 * Get a packet from the jitterbuffer if it's available.
 * control packets have a higher priority above voice and silence
packets
 * they are always delivered as fast as possible. The delay of the
jitterbuffer
 * doesn't work for these packets.
 * @REQUIRE 1<interpl <= jb->settings->extra_delay (=default
JB_ALLOW_EXTRA_DELAY)
 *
 * return will be:
 * JB_OK, *data points to the packet
 * JB_INTERP, please interpolate for interpl milliseconds
 * JB_NOFRAME, no frame scheduled
 * JB_EMPTY, the jitterbuffer is empty
 */
int jb_get(jitterbuffer *jb, void **data, long now, long interpl);
```

```
/* debug functions */
typedef          void (*jb_output_function_t)(const char *fmt, ...);
void             jb_setoutput(jb_output_function_t warn,
jb_output_function_t err, jb_output_function_t dbg);


/*******************************
 * The use of the jitterbuffer *
 *******************************
 * Always create a new jitterbuffer with jb_new().
 * Always destroy a jitterbuffer with jb_destroy().
 *
 * There is no lock(mutex) mechanism, that your responsibility.
 * The reason for this is that different environments require
 * different ways of implementing a lock.
 *
 * The following functions require a lock on the jitterbuffer:
 * jb_reset(), jb_reset_all(), jb_destroy(), jb_set_settings(),
 * jb_get_info(), jb_get_settings(), jb_has_frames(), jb_put(),
 * jb_get()
 *
 * The following functions do NOT require a lock on the jitterbuffer:
 * jb_new(), jb_guess_mos()
 *
 * Since control packets have a higher priority above any other packet
 * a call may already be ended while there is audio left to play. We
 * advice that you poll the jitterbuffer if there are frames left.
 *
 * If the audiopath is oneway (eg. voicemailbox) and the latency
doesn't
 * matter, we advice to set a minimum jitterbuffer size. Then there is
 * less loss and the quality is better.
 */


/****************************
 * debug messages explained *
 ****************************
 * N  - jb_new()
 * R  - jb_reset()
 * r  - jb_reset_all()
 * D  - jb_destroy()
 * S  - jb_set_settings()
 * H  - jb_has_frames()
 * I  - jb_get_info()
 * S  - jb_get_settings()
 * pC - jb_put() put Control packet
 * pT - jb_put() Timestamp was already in the queue
 * pV - jb_put() put Voice packet
 * pS - jb_put() put Silence packet
 *
 * A  - jb_get()
 * // below are all the possible debug info when trying to get a packet
 * gC - get_control() - there is a control message
 * gs - get_voice() - there is a silence frame
 * gS - get_voice() - we are in silence
 * gL - get_voice() - are in silence, frame is late
```

```
 * gP - get_voice() - are in silence, play frame (end of silence)
 * ag - get_voicecase() - grow little bit (diff < interpl/2)
 * aG - get_voicecase() - grow interpl
 * as - get_voicecase() - shrink by voiceframe we throw out
 * aS - get_voicecase() - shrink by interpl
 * aN - get_voicecase() - no time yet
 * aL - get_voicecase() - frame is late
 * aP - get_voicecase() - play frame
 * aI - get_voicecase() - interpolate
 */

#ifdef __cplusplus
}
#endif


#endif
```

Bijlage D: Jitterbuffer

```
/*******************************************************
 * jitterbuffer:
 * an application-independent jitterbuffer, which tries
 * to achieve the maximum user perception during a call.
 * For more information look at:
 * http://www.speakup.nl/opensource/jitterbuffer/
 *
 * Copyright on this file is held by:
 * - Jesse Kaijen <jesse@speakup.nl>
 * - SpeakUp <info@speakup.nl>
 *
 * Contributors:
 * Jesse Kaijen <jesse@speakup.nl>
 *
 * This program is free software, distributed under the terms of:
 * - the GNU Lesser (Library) General Public License
 * - the Mozilla Public License
 *
 * if you are interested in an different licence type, please contact us.
 *
 * How to use the jitterbuffer, please look at the comments
 * in the headerfile.
 *
 * Further details on specific implementations,
 * please look at the comments in the code file.
 */

#include "jitterbuffer.h"
#include <stdlib.h>
#include <string.h>
#include <limits.h>

#define jb_warn(...) (warnf ? warnf(__VA_ARGS__) : (void)0)
#define jb_err(...) (errf ? errf(__VA_ARGS__) : (void)0)
#define jb_dbg(...) (dbgf ? dbgf(__VA_ARGS__) : (void)0)


//public functions
jitterbuffer *jb_new();
void jb_reset(jitterbuffer *jb);
void jb_reset_all(jitterbuffer *jb);
void jb_destroy(jitterbuffer *jb);
void jb_set_settings(jitterbuffer *jb, jb_settings *settings);

void jb_get_info(jitterbuffer *jb, jb_info *stats);
void jb_get_settings(jitterbuffer *jb, jb_settings *settings);
float jb_guess_mos(float p, long d, int codec);
int jb_has_frames(jitterbuffer *jb);

void jb_put(jitterbuffer *jb, void *data, int type, long ms, long ts,
long now, int codec);
int jb_get(jitterbuffer *jb, void **data, long now, long interpl);
```

```
//private functions
static void set_default_settings(jitterbuffer *jb);
static void reset(jitterbuffer *jb);
static long find_pointer(long *array, long max_index, long value);
static void frame_free(jb_frame *frame);

static void put_control(jitterbuffer *jb, void *data, int type, long
ts);
static void put_voice(jitterbuffer *jb, void *data, int type, long ms,
long ts, int codec);
static void put_history(jitterbuffer *jb, long ts, long now, long ms,
int codec);
static void calculate_info(jitterbuffer *jb, long ts, long now, int
codec);

static int get_control(jitterbuffer *jb, void **data);
static int get_voice(jitterbuffer *jb, void **data, long now, long
interpl);
static int get_voicecase(jitterbuffer *jb, void **data, long now, long
interpl, long diff);

static int get_next_frametype(jitterbuffer *jb, long ts);
static long get_next_framets(jitterbuffer *jb);
static jb_frame *get_frame(jitterbuffer *jb, long ts);
static jb_frame *get_all_frames(jitterbuffer *jb);

//debug...
static jb_output_function_t warnf, errf, dbgf;
void jb_setoutput(jb_output_function_t warn, jb_output_function_t err,
jb_output_function_t dbg) {
    warnf = warn;
    errf = err;
    dbgf = dbg;
}


/***********
 * create a new jitterbuffer
 * return NULL if malloc doesn't work
 * else return jb with default_settings.
 */
jitterbuffer *jb_new()
{
  jitterbuffer *jb;

  jb_dbg("N");
  jb = malloc(sizeof(jitterbuffer));
  if (!jb) {
    jb_err("cannot allocate jitterbuffer\n");
    return NULL;
  }
  set_default_settings(jb);
  reset(jb);
  return jb;
}
```

```
/***********
 * empty voice messages
 * reset statistics
 * keep the settings
 */
void jb_reset(jitterbuffer *jb)
{
  jb_frame *frame;

  jb_dbg("R");
  if (jb == NULL) {
    jb_err("no jitterbuffer in jb_reset()\n");
    return;
  }

  //free voice
  while(jb->voiceframes) {
    frame = get_all_frames(jb);
    frame_free(frame);
  }
  //reset stats
  memset(&(jb->info),0,sizeof(jb_info) );
  // set default settings
  reset(jb);
}


/***********
 * empty nonvoice messages
 * empty voice messages
 * reset statistics
 * reset settings to default
 */
void jb_reset_all(jitterbuffer *jb)
{
  jb_frame *frame;

  jb_dbg("r");
  if (jb == NULL) {
    jb_err("no jitterbuffer in jb_reset_all()\n");
    return;
  }

  // free nonvoice
  while(jb->controlframes) {
    frame = jb->controlframes;
    jb->controlframes = frame->next;
    frame_free(frame);
  }
  // free voice and reset statistics is done by jb_reset
  jb_reset(jb);
  set_default_settings(jb);
}


/***********
 * destroy the jitterbuffer
```

```
 * free all the [non]voice frames with reset_all
 * free the jitterbuffer
 */
void jb_destroy(jitterbuffer *jb)
{
  jb_dbg("D");
  if (jb == NULL) {
    jb_err("no jitterbuffer in jb_destroy()\n");
    return;
  }

  jb_reset_all(jb);
  free(jb);
}


/***********
 * Set settings for the jitterbuffer.
 * Only if a setting is defined it will be written
 * in the jb->settings.
 * This means that no setting can be set to zero
 */
void jb_set_settings(jitterbuffer *jb, jb_settings *settings)
{
  jb_dbg("S");
  if (jb == NULL) {
    jb_err("no jitterbuffer in jb_set_settings()\n");
    return;
  }

  if (settings->min_jb) {
    jb->settings.min_jb = settings->min_jb;
  }
  if (settings->max_jb) {
    jb->settings.max_jb = settings->max_jb;
  }
  if (settings->max_successive_interp) {
    jb->settings.max_successive_interp = settings-
>max_successive_interp;
  }
  if (settings->extra_delay) {
    jb->settings.extra_delay = settings->extra_delay;
  }
  if (settings->wait_grow) {
    jb->settings.wait_grow = settings->wait_grow;
  }
  if (settings->wait_shrink) {
    jb->settings.wait_shrink = settings->wait_shrink;
  }
  if (settings->max_diff) {
    jb->settings.max_diff = settings->max_diff;
  }
}


/***********
 * validates the statistics
```

```
 * the losspct due the jitterbuffer will be calculated.
 * delay and delay_target will be calculated
 * *stats = info
 */
void jb_get_info(jitterbuffer *jb, jb_info *stats)
{
  long max_index, pointer;

  jb_dbg("I");
  if (jb == NULL) {
    jb_err("no jitterbuffer in jb_get_info()\n");
    return;
  }

  jb->info.delay = jb->current - jb->min;
  jb->info.delay_target = jb->target - jb->min;

  //calculate the losspct...
  max_index = (jb->hist_pointer < JB_HISTORY_SIZE) ?
                      jb->hist_pointer : JB_HISTORY_SIZE-1;
  if (max_index>1) {
    pointer = find_pointer(&jb->hist_sorted_delay[0], max_index,
                      jb->current);
    jb->info.losspct = ((max_index - pointer)*100/max_index);
    if (jb->info.losspct < 0) {
      jb->info.losspct = 0;
    }
  } else {
    jb->info.losspct = 0;
  }

  *stats = jb->info;
}


/***********
 * gives the settings for this jitterbuffer
 * *settings = settings
 */
void jb_get_settings(jitterbuffer *jb, jb_settings *settings)
{
  jb_dbg("S");
  if (jb == NULL) {
    jb_err("no jitterbuffer in jb_get_settings()\n");
    return;
  }

  *settings = jb->settings;
}


/***********
 * returns an estimate on the MOS with given loss, delay and codec
 * if the formula is not present the default will be used
 * please use the JB_CODEC_OTHER if you want to define your own formula
 *
 */
```

```
float jb_guess_mos(float p, long d, int codec)
{
  float result;

  switch (codec) {
    case JB_CODEC_GSM_EFR:
      result = (4.31 - 0.23*p - 0.0071*d);
    break;

    case JB_CODEC_G723_1:
      result = (3.99 - 0.16*p - 0.0071*d);
    break;

    case JB_CODEC_G729:
    case JB_CODEC_G729A:
      result = (4.13 - 0.14*p - 0.0071*d);
    break;

    case JB_CODEC_G711x_PLC:
      result = (4.42 - 0.087*p - 0.0071*d);
    break;

    case JB_CODEC_G711x:
      result = (4.42 - 0.63*p - 0.0071*d);
    break;

    case JB_CODEC_OTHER:
    default:
      result = (4.42 - 0.63*p - 0.0071*d);

  }
  return result;
}


/***********
 * if there are any frames left in JB returns JB_OK, otherwise returns
JB_EMPTY
 */
int jb_has_frames(jitterbuffer *jb)
{
  jb_dbg("H");
  if (jb == NULL) {
    jb_err("no jitterbuffer in jb_has_frames()\n");
    return;
  }

  if(jb->controlframes || jb->voiceframes) {
    return JB_OK;
  } else {
    return JB_EMPTY;
  }
}


/***********
 * Put a packet into the jitterbuffers
```

```
 * Only the timestamps of voicepackets are put in the history
 * this because the jitterbuffer only works for voicepackets
 * don't put packets twice in history and queue (e.g. transmitting
every frame twice)
 * keep track of statistics
 */
void jb_put(jitterbuffer *jb, void *data, int type, long ms, long ts,
long now, int codec)
{
  long pointer, max_index;

  if (jb == NULL) {
    jb_err("no jitterbuffer in jb_put()\n");
    return;
  }

  jb->info.frames_received++;

  if (type == JB_TYPE_CONTROL) {
    //put the packet into the contol-queue of the jitterbuffer
    jb_dbg("pC");
    put_control(jb,data,type,ts);

  } else if (type == JB_TYPE_VOICE) {
    // only add voice that aren't already in the buffer
    max_index = (jb->hist_pointer < JB_HISTORY_SIZE) ? jb->hist_pointer
: JB_HISTORY_SIZE-1;
    pointer = find_pointer(&jb->hist_sorted_timestamp[0], max_index,
ts);
    if (jb->hist_sorted_timestamp[pointer]==ts) { //timestamp already
in queue
      jb_dbg("pT");
      free(data);
      jb->info.frames_dropped_twice++;
    } else { //add
      jb_dbg("pV");
      /* add voicepacket to history */
      put_history(jb,ts,now,ms,codec);
      /*calculate jitterbuffer size*/
      calculate_info(jb, ts, now, codec);
      /*put the packet into the queue of the jitterbuffer*/
      put_voice(jb,data,type,ms,ts,codec);
    }

  } else if (type == JB_TYPE_SILENCE){ //silence
    jb_dbg("pS");
    put_voice(jb,data,type,ms,ts,codec);

  } else {//should NEVER happen
    jb_err("jb_put(): type not known\n");
    free(data);
  }
}


/***********
 * control frames have a higher priority then voice frames
```

```
 * returns JB_OK if a frame is available and *data points to the packet
 * returns JB_NOFRAME if it's no time to play voice and no control
available
 * returns JB_INTERP if interpolating is required
 * returns JB_EMPTY if no voice frame is in the jitterbuffer (only
during silence)
 */
int jb_get(jitterbuffer *jb, void **data, long now, long interpl)
{
  int result;

  jb_dbg("A");
  if (jb == NULL) {
    jb_err("no jitterbuffer in jb_get()\n");
    return;
  }

  result = get_control(jb, data);
  if (result != JB_OK ) { //no control message available maybe there is
voice...
    result = get_voice(jb, data, now, interpl);
  }
  return result;
}


/***********
 * set all the settings to default
 */
static void set_default_settings(jitterbuffer *jb)
{
  jb->settings.min_jb = JB_MIN_SIZE;
  jb->settings.max_jb = JB_MAX_SIZE;
  jb->settings.max_successive_interp = JB_MAX_SUCCESSIVE_INTERP;
  jb->settings.extra_delay = JB_ALLOW_EXTRA_DELAY;
  jb->settings.wait_grow = JB_WAIT_GROW;
  jb->settings.wait_shrink = JB_WAIT_SHRINK;
  jb->settings.max_diff = JB_MAX_DIFF;
}


/***********
 * reset the jitterbuffer so we can start in silence and
 * we start with a new history
 */
static void reset(jitterbuffer *jb)
{
  jb->hist_pointer = 0; //start over
  jb->silence_begin_ts = 0; //no begin_ts defined
  jb->info.silence =1; //we always start in silence
}


/***********
 * Search algorithm
 * @REQUIRE max_index is within array
 *
```

```
 * Find the position of value in hist_sorted_delay
 * if value doesn't exist return first pointer where array[low]>value
 * int low;   //the lowest index being examined
 * int max_index; //the highest index being examined
 * int mid;  //the middle index between low and max_index.
 * mid ==(low+max_index)/2
 * at the end low is the position of value or where array[low]>value
 */
static long find_pointer(long *array, long max_index, long value)
{
  long low, mid, high;
  low = 0;
  high = max_index;
  while (low<=high) {
    mid= (low+high)/2;
    if (array[mid] < value) {
      low = mid+1;
    } else {
      high = mid-1;
    }
  }
  while(low < max_index && (array[low]==array[(low+1)]) ) {
    low++;
  }
  return low;
}


/***********
 * free the given frame, afterwards the framepointer is undefined
 */
static void frame_free(jb_frame *frame)
{
  if (frame->data) {
    free(frame->data);
  }
  free(frame);
}


/***********
 * put a nonvoice frame into the nonvoice queue
 */
static void put_control(jitterbuffer *jb, void *data, int type, long
ts)
{
  jb_frame *frame, *p;

  frame = malloc(sizeof(jb_frame));
  if(!frame) {
    jb_err("cannot allocate frame\n");
    return;
  }
  frame->data = data;
  frame->ts = ts;
  frame->type = type;
  frame->next = NULL;
```

```
  data = NULL;//to avoid stealing memory

  p = jb->controlframes;
  if (p) { //there are already control messages
    if (ts < p->ts) {
      jb->controlframes = frame;
      frame->next = p;
    } else {
      while (p->next && (ts >=p->next->ts)) {//sort on timestamps! so
find place to put...
        p = p->next;
      }
      if (p->next) {
        frame->next = p->next;
      }
      p->next = frame;
    }
  } else {
    jb->controlframes = frame;
  }
}


/***********
 * put a voice or silence frame into the jitterbuffer
 */
static void put_voice(jitterbuffer *jb, void *data, int type, long ms,
long ts, int codec)
{
  jb_frame *frame, *p;
  frame = malloc(sizeof(jb_frame));
  if(!frame) {
    jb_err("cannot allocate frame\n");
    return;
  }

  frame->data = data;
  frame->ts = ts;
  frame->ms = ms;
  frame->type = type;
  frame->codec = codec;

  data = NULL; //to avoid stealing the memory location
  /*
   * frames are a circular list, jb->voiceframes points to to the
lowest ts,
   * jb->voiceframes->prev points to the highest ts
   */
  if(!jb->voiceframes) {  /* queue is empty */
    jb->voiceframes = frame;
    frame->next = frame;
    frame->prev = frame;
  } else {
    p = jb->voiceframes;
    if(ts < p->prev->ts) { //frame is out of order
      jb->info.frames_ooo++;
    }
```

```
    if (ts < p->ts) { //frame is lowest, let voiceframes point to it!
      jb->voiceframes = frame;
    } else {
      while(ts < p->prev->ts ) {
        p = p->prev;
      }
    }
    frame->next = p;
    frame->prev = p->prev;
    frame->next->prev = frame;
    frame->prev->next = frame;
  }
}


/***********
 * puts the timestamps of a received packet in the history of *jb
 * for later calculations of the size of jitterbuffer *jb.
 *
 * summary of function:
 * - calculate delay difference
 * - delete old value from hist & sorted_history_delay &
sorted_history_timestamp if needed
 * - add new value to history & sorted_history_delay &
sorted_history_timestamp
 * - we keep sorted_history_delay for calculations
 * - we keep sorted_history_timestamp for ensuring each timestamp isn't
put twice in the buffer.
 */
static void put_history(jitterbuffer *jb, long ts, long now, long ms,
int codec)
{
  jb_hist_element out, in;
  long max_index, pointer, location;

  // max_index is the highest possible index
  max_index = (jb->hist_pointer < JB_HISTORY_SIZE) ? jb->hist_pointer :
JB_HISTORY_SIZE-1;
  location = (jb->hist_pointer % JB_HISTORY_SIZE);

  // we want to delete a value from the jitterbuffer
  // only when we are through the history.
  if (jb->hist_pointer > JB_HISTORY_SIZE-1) {
    /* the value we need to delete from sorted histories */
    out = jb->hist[location];
    //delete delay from hist_sorted_delay
    pointer = find_pointer(&jb->hist_sorted_delay[0], max_index,
out.delay);
    /* move over pointer is the position of kicked*/
    if (pointer<max_index) { //only move if we have something to move
      memmove(  &(jb->hist_sorted_delay[pointer]),
                &(jb->hist_sorted_delay[pointer+1]),
                ((JB_HISTORY_SIZE-(pointer+1)) * sizeof(long)) );
    }

    //delete timestamp from hist_sorted_timestamp
```

```
    pointer = find_pointer(&jb->hist_sorted_timestamp[0], max_index,
out.ts);
    /* move over pointer is the position of kicked*/
    if (pointer<max_index) { //only move if we have something to move
      memmove(  &(jb->hist_sorted_timestamp[pointer]),
                &(jb->hist_sorted_timestamp[pointer+1]),
                ((JB_HISTORY_SIZE-(pointer+1)) * sizeof(long)) );
    }
  }

  in.delay = now - ts;    //delay of current packet
  in.ts = ts;      //timestamp of current packet
  in.ms = ms;      //length of current packet
  in.codec = codec;      //codec of current packet

  /* adding the new delay to the sorted history
   * first special cases:
   * - delay is the first history stamp
   * - delay > highest history stamp
   */
  if (max_index==0 || in.delay >= jb->hist_sorted_delay[max_index-1]) {
    jb->hist_sorted_delay[max_index] = in.delay;
  } else {
    pointer = find_pointer(&jb->hist_sorted_delay[0], (max_index-1),
in.delay);
    /* move over and add delay */
    memmove(  &(jb->hist_sorted_delay[pointer+1]),
              &(jb->hist_sorted_delay[pointer]),
              ((JB_HISTORY_SIZE-(pointer+1)) * sizeof(long)) );
    jb->hist_sorted_delay[pointer] = in.delay;
  }

  /* adding the new timestamp to the sorted history
   * first special cases:
   * - timestamp is the first history stamp
   * - timestamp > highest history stamp
   */
  if (max_index==0 || in.ts >= jb->hist_sorted_timestamp[max_index-1])
{
    jb->hist_sorted_timestamp[max_index] = in.ts;
  } else {

    pointer = find_pointer(&jb->hist_sorted_timestamp[0], (max_index-
1), in.ts);
    /* move over and add timestamp */
    memmove(  &(jb->hist_sorted_timestamp[pointer+1]),
              &(jb->hist_sorted_timestamp[pointer]),
              ((JB_HISTORY_SIZE-(pointer+1)) * sizeof(long)) );
    jb->hist_sorted_timestamp[pointer] = in.ts;
  }

  /* put the jb_hist_element in the history
  * then increase hist_pointer for next time
  */
  jb->hist[location] = in;
  jb->hist_pointer++;
}
```

```
/***********
 * this tries to make a jitterbuffer that behaves like
 * the jitterbuffer proposed in this article:
 * Adaptive Playout Buffer Algorithm for Enhancing Perceived Quality of
Streaming Applications
 * by: Kouhei Fujimoto & Shingo Ata & Masayuki Murata
 * http://www.nal.ics.es.osaka-
u.ac.jp/achievements/web2002/pdf/journal/k-fujimo02TSJ-
AdaptivePlayoutBuffer.pdf
 *
 * it calculates jitter and minimum delay
 * get the best delay for the specified codec

 */
static void calculate_info(jitterbuffer *jb, long ts, long now, int
codec)
{
  long diff, size, max_index, d, d1, d2, n;
  float p, p1, p2, A, B;
  //size = how many items there in the history
  size = (jb->hist_pointer < JB_HISTORY_SIZE) ? jb->hist_pointer :
JB_HISTORY_SIZE;
  max_index = size-1;

  /*
   * the Inter-Quartile Range can be used for estimating jitter
   * http://www.slac.stanford.edu/comp/net/wan-
mon/tutorial.html#variable
   * just take the square root of the iqr for jitter
   */
  jb->info.iqr = jb->hist_sorted_delay[max_index*3/4] - jb-
>hist_sorted_delay[max_index/4];


  /*
   * The RTP way of calculating jitter.
   * This one is used at the moment, although it is not correct.
   * But in this way the other side understands us.
   */
  diff = now - ts - jb->last_delay;
  if (!jb->last_delay) {
    diff = 0; //this to make sure we won't get odd jitter due first ts.
  }
  jb->last_delay = now - ts;
  if (diff <0){
    diff = -diff;
  }
  jb->info.jitter = jb->info.jitter + (diff - jb->info.jitter)/16;

  /* jb->min is minimum delay in hist_sorted_delay, we don't look at
the lowest 2% */
  /* because sometimes there are odd delays in there */
  jb->min = jb->hist_sorted_delay[(max_index*2/100)];

  /*
```

```
   * calculating the preferred size of the jitterbuffer:
   * instead of calculating the optimum delay using the Pareto equation
   * I use look at the array of sorted delays and choose my optimum
from there
   * always walk trough a percentage of the history this because
imagine following tail:
   * [...., 12, 300, 301 ,302]
   * her we want to discard last three but that won't happen if we
won't walk the array
   * the number of frames we walk depends on how scattered the sorted
delays are.
   * For that we look at the iqr. The dependencies of the iqr are based
on
   * tests we've done here in the lab. But are not optimized.
   */
  //init:
  //the higest delay..
  d = d1= d2 = jb->hist_sorted_delay[max_index]- jb->min;
  A=B=LONG_MIN;
  p = p2 =0;
  n=0;
  p1 = 5; //always look at the top 5%
  if (jb->info.iqr >200) { //with more jitter look at more delays
    p1=25;
  } else if (jb->info.iqr >100) {
    p1=20;
  } else if (jb->info.iqr >50){
    p1=11;
  }

  //find the optimum delay..
  while(max_index>10 && (B >= A ||p2<p1)) {
    //the packetloss with this delay
    p2 =(n*100/size);
    // estimate MOS-value
    B = jb_guess_mos(p2,d2,codec);
    if (B > A) {
      p = p2;
      d = d2;
      A = B;
    }
    d1 = d2;
    //find next delay != delay so the same delay isn't calculated twice
    //don't look further if we have seen half of the history
    while((d2>=d1) && ((n*2)<max_index) ) {
      n++;
      d2 = jb->hist_sorted_delay[(max_index-n)] - jb->min;
    }
  }
  //the targeted size of the jitterbuffer
  if (jb->settings.min_jb && (jb->settings.min_jb > d) ) {
    jb->target = jb->min + jb->settings.min_jb;
  } else if (jb->settings.max_jb && (jb->settings.max_jb > d) ){
    jb->min + jb->settings.max_jb;
  } else {
    jb->target = jb->min + d;
  }
```

```
}


/***********
 * if there is a nonvoice frame it will be returned [*data] and the
frame
 * will be made free
 */
static int get_control(jitterbuffer *jb, void **data)
{
  jb_frame *frame;
  int result;

  frame = jb->controlframes;
  if (frame) {
    jb_dbg("gC");
    *data = frame->data;
    frame->data = NULL;
    jb->controlframes = frame->next;
    frame_free(frame);
    result = JB_OK;
  } else {
    result = JB_NOFRAME;
  }
  return result;
}


/***********
 * returns JB_OK if a frame is available and *data points to the packet
 * returns JB_NOFRAME if it's no time to play voice and or no frame
available
 * returns JB_INTERP if interpolating is required
 * returns JB_EMPTY if no voice frame is in the jitterbuffer (only
during silence)
 *
 * if the next frame is a silence frame we will go in silence-mode
 * each new instance of the jitterbuffer will start in silence mode
 * in silence mode we will set the jitterbuffer to the size we want
 * when we are not in silence mode get_voicecase will handle the rest.
 */
static int get_voice(jitterbuffer *jb, void **data, long now, long
interpl)
{
  jb_frame *frame;
  long diff;
  int result;

  diff = jb->target - jb->current;

  //if the next frame is a silence frame, go in silence mode...
  if((get_next_frametype(jb, now - jb->current) == JB_TYPE_SILENCE) ) {
    jb_dbg("gs");
    frame = get_frame(jb, now - jb->current);
    *data = frame->data;
    frame->data = NULL;
    jb->info.silence =1;
```

```
    jb->silence_begin_ts = frame->ts;
    frame_free(frame);
    result = JB_OK;
  } else {
    if(jb->info.silence) { // we are in silence
      /*
       * During silence we can set the jitterbuffer size to the size
       * we want...
       */
      if (diff) {
        jb->current = jb->target;
      }
      frame = get_frame(jb, now - jb->current);
      if (frame) {
        if (jb->silence_begin_ts && frame->ts < jb->silence_begin_ts) {
          jb_dbg("gL");
          /* voice frame is late, next!*/
          jb->info.frames_late++;
          frame_free(frame);
          result = get_voice(jb, data, now, interpl);
        } else {
          jb_dbg("gP");
          /* voice frame */
          jb->info.silence = 0;
          jb->silence_begin_ts = 0;
          jb->next_voice_time = frame->ts + frame->ms;
          jb->info.last_voice_ms = frame->ms;
          *data = frame->data;
          frame->data = NULL;
          frame_free(frame);
          result = JB_OK;
        }
      } else {    //no frame
        jb_dbg("gS");
        result = JB_EMPTY;
      }
    } else { //voice case
      result = get_voicecase(jb,data,now,interpl,diff);
    }
  }
  return result;
}


/***********
 * The voicecase has four 'options'
 * - difference is way off, reset
 * - diff > 0, we may need to grow
 * - diff < 0, we may need to shrink
 * - everything else
 */
static int get_voicecase(jitterbuffer *jb, void **data, long now, long
interpl, long diff)
{
  jb_frame *frame;
  int result;
```

```
   // * - difference is way off, reset
  if (diff > jb->settings.max_diff || -diff > jb->settings.max_diff) {
    jb_err("wakko diff in get_voicecase\n");
    reset(jb); //reset hist because the timestamps are wakko.
    result = JB_NOFRAME;
  //- diff > 0, we may need to grow
  } else if ((diff > 0) &&
                   (now > (jb->last_adjustment + jb-
>settings.wait_grow)
                    || (now + jb->current + interpl) <
get_next_framets(jb) ) ) { //grow
    /* first try to grow */
    if (diff<interpl/2) {
      jb_dbg("ag");
      jb->current +=diff;
    } else {
      jb_dbg("aG");
      /* grow by interp frame len */
      jb->current += interpl;
    }
    jb->last_adjustment = now;
    result = get_voice(jb, data, now, interpl);
  //- diff < 0, we may need to shrink
  } else if ( (diff < 0)
                && (now > (jb->last_adjustment + jb-
>settings.wait_shrink))
                && ((-diff) > jb->settings.extra_delay) ) {
    /* now try to shrink
     * if there is a frame shrink by frame length
     * otherwise shrink by interpl
     */
    jb->last_adjustment = now;

    frame = get_frame(jb, now - jb->current);
    if(frame) {
      jb_dbg("as");
      /* shrink by frame size we're throwing out */
      jb->info.frames_dropped++;
      jb->current -= frame->ms;
      frame_free(frame);
    } else {
      jb_dbg("aS");
      /* shrink by interpl */
      jb->current -= interpl;
    }
    result = get_voice(jb, data, now, interpl);
  } else  {
    /* if it is not the time to play a result = JB_NOFRAME
     * else We try to play a frame if a frame is available
     * and not late it is played otherwise
     * if available it is dropped and the next is tried
     * last option is interpolating
     */
    if (now - jb->current < jb->next_voice_time) {
      jb_dbg("aN");
      result = JB_NOFRAME;
    } else {
```

```
      frame = get_frame(jb, now - jb->current);
      if (frame) { //there is a frame
        /* voice frame is late */
        if(frame->ts < jb->next_voice_time) {   //late
          jb_dbg("aL");
          jb->info.frames_late++;
          frame_free(frame);
          result = get_voice(jb, data, now, interpl);
        } else {
          jb_dbg("aP");
          /* normal case; return the frame, increment stuff */
          *data = frame->data;
          frame->data = NULL;
          jb->next_voice_time = frame->ts + frame->ms;
          jb->cnt_successive_interp = 0;
          frame_free(frame);
          result = JB_OK;
        }
      } else { // no frame, thus interpolate
        jb->cnt_successive_interp++;
        /* assume silence instead of continuing to interpolate */
        if (jb->settings.max_successive_interp && jb-
>cnt_successive_interp >= jb->settings.max_successive_interp) {
          jb->info.silence = 1;
          jb->silence_begin_ts = jb->next_voice_time;
        }
        jb_dbg("aI");
        jb->next_voice_time += interpl;
        result = JB_INTERP;
      }
    }
  }
  return result;

}


/***********
 * if there are frames and next frame->ts is smaller or equal ts
 *   return type of next frame.
 * else return 0
 */
static int get_next_frametype(jitterbuffer *jb, long ts)
{
  jb_frame *frame;
  int result;

  result = 0;
  frame = jb->voiceframes;
  if (frame && frame->ts <= ts) {
    result = frame->type;
  }
  return result;
}


/***********
```

```
 * returns ts from next frame in jb->voiceframes
 * or returns LONG_MAX if there is no frame
 */
static long get_next_framets(jitterbuffer *jb)
{
  if (jb->voiceframes) {
    return jb->voiceframes->ts;
  }
  return LONG_MAX;
}


/***********
 * if there is a frame in jb->voiceframes and
 * has a timestamp smaller/equal to ts
 * this frame will be returned and
 * removed from the queue
 */
static jb_frame *get_frame(jitterbuffer *jb, long ts)
{
  jb_frame *frame;

  frame = jb->voiceframes;
  if (frame && frame->ts <= ts) {
    if(frame->next == frame) {
      jb->voiceframes = NULL;
    } else {
      /* remove this frame */
      frame->prev->next = frame->next;
      frame->next->prev = frame->prev;
      jb->voiceframes = frame->next;
    }
    return frame;
  }
  return NULL;
}

/***********
 * if there is a frame in jb->voiceframes
 * this frame will be unconditionally returned and
 * removed from the queue
 */
static jb_frame *get_all_frames(jitterbuffer *jb)
{
  jb_frame *frame;

  frame = jb->voiceframes;
  if (frame) {
    if(frame->next == frame) {
      jb->voiceframes = NULL;
    } else {
      /* remove this frame */
      frame->prev->next = frame->next;
      frame->next->prev = frame->prev;
      jb->voiceframes = frame->next;
    }
    return frame;
```

```
  }
  return NULL;
}


//EOF
```

Afkortingen en verklarende woordenlijst

ACR	Absolute Category Rating
Codec	coder/decoder
CQ	Conversational Quality
dB	decibel
E-MOS	Enhanced-MOS
ETSI	European Telecommunications Standardization Institute
G.711	codec ontwikkeld door ITU
G.723.1	codec ontwikkeld door ITU
G.726	codec ontwikkeld door ITU
G.729	codec ontwikkeld door ITU
GSM	codec ontwikkeld door ESTI
GSM-FR	codec ontwikkeld door ESTI
GSM-HR	codec ontwikkeld door ESTI
GSM-EFR	codec ontwikkeld door ESTI
H.323	ITU protocol voor real-time communicatie over packetswitched network
IAX2	Inter-Asterisk eXchange Protocol versie 2.
IP	Internet Protocol
ITU	International Telecommunication Union
JB	jitterbuffer
LQ	Listening Quality
MOS	Mean Opinion Score
PESQ	Perceptual Evaluation of Speech Quality
PLC	Packet Loss Concealment
PSTN	Public Switched Telephone Network
QoS	Quality of Service
SIP	Session Initiation Protocol
VoIP	Voice over IP, Voice over Internet Protocol