Realtime signaal synchronisatie met accoustic fingerprinting

Ward Van Assche

Promotoren: dr. Marleen Denert, Joren Six
Begeleider: prof. Helga Naessens

Masterproef ingediend tot het behalen van de academische graad van
Master of Science in de industriële wetenschappen: informatica

Vakgroep Informatietechnologie
Voorzitter: prof. dr. ir. Daniël De Zutter

Vakgroep Kunst-, Muziek- en Theaterwetenschappen
Voorzitter: prof. dr. Francis Maes

Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2015-2016

UNIVERSITEIT
GENT

Faculteit Ingenieurswetenschappen en Architectuur
Vakgroep Informatietechnologie
Voorzitter: Prof. Dr. Ir. Daniël De Zutter

Realtime signaal synchronisatie met accoustic fingerprinting

door

Ward Van Assche

Promotoren: Dr. Marleen Denert, Joren Six
Scriptiebegeleider: Prof. Helga Naessens

Masterproef ingediend tot het behalen van de academische graad van
Master of Science in de industriële wetenschappen: informatica

Academiejaar 2015–2016

Voorwoord

Zonder hulp van buitenaf zou ik er nooit in geslaagd zijn om mijn masterproef tot een goed einde te brengen. Daarom wil ik verschillende mensen bedanken die een grote rol gespeeld in één of meerdere fases van dit eindwerk.

Eerst en vooral wil ik mijn externe promotor, Joren Six, bedanken voor het vertrouwen en de ondersteuning die hij mij tijdens het uitwerken van deze masterproef heeft gegeven. De kennis en inzicht die ik van hem heb meegekregen op vlak van digitale audio zal mij zeker bijblijven. Ik vond het zeer leerrijk om mijn interesse in muziek en geluid te kunnen combineren met mijn opleiding informatica.

Verder wil ik ook mijn interne promotor, Marleen Denert, bedanken voor het opvolgen en nalezen van mijn thesis. Haar opbouwende kritiek was van onschatbare waarde.

Ten slotte wil ik ook mijn ouders, zus en vrienden bedanken die mij altijd gesteund hebben tijdens mijn opleiding tot industrieel ingenieur.

Verder wil ik ook u, de lezer, bedanken voor de interesse in mijn onderzoek. Ik wens u veel leesplezier.

Ward Van Assche, juni 2016

Toelating tot bruikleen

Realtime signaal synchronisatie met accoustic fingerprinting

door

Ward Van Assche

Masterproef ingediend tot het behalen van de academische graad van
Master of Science in de industriële wetenschappen: informatica

Academiejaar 2015–2016

Promotoren: Dr. Marleen Denert, Joren Six
Scriptiebegeleider: Prof. Helga Naessens

Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Vakgroep Informatietechnologie
Voorzitter: Prof. Dr. Ir. Daniël De Zutter

Samenvatting

Todo: samenvatting schrijven

Trefwoorden

synchronisatie, realtime, signalen, streams, musicologie, accoustic fingerprinting, kruiscovariantie, digitale signaalverwerking

Realtime signal synchronization with accoustic fingerprinting

Ward Van Assche

Supervisor(s): Joren Six, Marleen Denert

Abstract—**Sed nec tortor in libero rutrum pellentesque et gravida turpis. Phasellus gravida neque vitae elit fringilla, a efficitur purus sollicitudin. Proin lacus est, suscipit sed nibh ac, hendrerit eleifend leo. Suspendisse quis semper leo. Duis non elit commodo, sodales ex non, venenatis diam. Sed libero tortor, hendrerit et sollicitudin ut, facilisis vitae odio. Fusce vitae mi odio.**

Keywords—**kernwoord1, kernwoord2, kernwoord 3, kernwoord 4**

I. Introduction

SED Sed nec tortor in libero rutrum pellentesque et gravida turpis. Phasellus gravida neque vitae elit fringilla, a efficitur purus sollicitudin. Proin lacus est, suscipit sed nibh ac, hendrerit eleifend leo. Suspendisse quis semper leo. Duis non elit commodo, sodales ex non, venenatis diam. Sed libero tortor, hendrerit et sollicitudin ut, facilisis vitae odio. Fusce vitae mi odio. Cras vitae quam bibendum, elementum velit ut, varius enim. Donec sagittis elit ligula, laoreet viverra felis rhoncus nec. Donec mattis metus pretium, pulvinar enim a, luctus nunc. Pellentesque quis suscipit leo.

II. Sectie

A. Subsectie

Sed nec tortor in libero rutrum pellentesque[1] et gravida turpis. Phasellus gravida neque vitae elit fringilla, a efficitur purus sollicitudin. Proin lacus est, suscipit sed nibh ac, hendrerit eleifend leo. Suspendisse quis semper leo. Duis non elit commodo, sodales ex non, venenatis diam. Sed libero tortor, hendrerit et sollicitudin ut, facilisis vitae odio. Fusce vitae mi odio. Cras vitae quam bibendum, elementum velit ut, varius enim. Donec sagittis elit ligula, laoreet viverra felis rhoncus nec. Donec mattis metus pretium, pulvinar enim a, luctus nunc. Pellentesque quis suscipit leo.

B. Andere subsectie

Sed nec tortor in libero rutrum pellentesque et gravida turpis. Phasellus gravida neque vitae elit fringilla, a efficitur purus sollicitudin. Proin lacus est, suscipit sed nibh ac, hendrerit eleifend leo. Suspendisse quis semper leo. Duis non elit commodo, sodales ex non, venenatis diam. Sed libero tortor, hendrerit et sollicitudin ut, facilisis vitae odio. Fusce vitae mi odio. Cras vitae quam bibendum, elementum velit ut, varius enim. Donec sagittis elit ligula, laoreet viverra felis rhoncus nec. Donec mattis metus pretium, pulvinar enim a, luctus nunc. Pellentesque quis suscipit leo.

Sed nec tortor in libero rutrum pellentesque et gravida turpis. Phasellus gravida neque vitae elit fringilla, a efficitur purus sollicitudin. Proin lacus est, suscipit sed nibh ac, hendrerit eleifend leo. Suspendisse quis semper leo. Duis non elit commodo, sodales ex non, venenatis diam. Sed libero tortor, hendrerit et sollicitudin ut, facilisis vitae odio. Fusce vitae mi odio. Cras vitae quam bibendum, elementum velit ut, varius enim. Donec sagittis elit ligula, laoreet viverra felis rhoncus nec. Donec mattis metus pretium, *pulvinar* enim a, luctus nunc. Pellentesque quis suscipit leo.

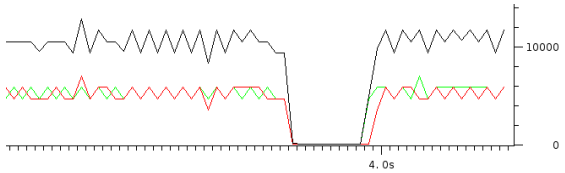


Fig. 1. Detailed capture of the stream at the moment of a handover between two simulated AP's with a strong signal. Notice the gap of 100 ms.

III. Sectie

Aenean auctor congue nisi, volutpat porta urna lobortis in. Donec accumsan fermentum lectus, sed aliquet lectus gravida eget. Ut turpis quam, fermentum eget sem sed, euismod facilisis sem. Etiam a sollicitudin purus. Vestibulum quis nisl et nibh condimentum suscipit tristique eget quam. Aenean eget varius lectus. Maecenas sit amet mi augue. Nullam semper ex et facilisis ullamcorper. Cras volutpat ornare arcu, ac suscipit nisi pellentesque iaculis. Vivamus sit amet ipsum consequat, egestas massa varius, aliquam lorem. Aenean ornare iaculis dolor, eget efficitur elit. Maecenas ut massa ac tortor hendrerit pulvinar a in ante.

IV. Conclusion

The simulation results show a nice advantage for the moving cell[3] concept. The traditional handover problem can be avoided so a workable model for broadband access on trains seems realistic. But there needs to be done a lot of research to make RAU's and RoF as reliable and cheap as possible.

References

[1] Joren Six, Olmo Cornelis, and Marc Leman, "TarsosDSP, a Real-Time Audio Processing Framework in Java," in *Proceedings of the 53rd AES Conference (AES 53rd)*. 2014, The Audio Engineering Society.

[2] Joren Six and Marc Leman, "Panako - A Scalable Acoustic Fingerprinting System Handling Time-Scale and Pitch Modification," in *Proceedings of the 15th ISMIR Conference (ISMIR 2014)*, 2014.

[3] Joren Six and Marc Leman, "Synchronizing Multimodal Recordings Using Audio-To-Audio Alignment," *Journal of Multimodal User Interfaces*, vol. 9, no. 3, pp. 223–229, 2015.

[4] A. L. Wang, "An industrial-strength audio search algorithm," in *ISMIR 2003, 4th Symposium Conference on Music Information Retrieval*, 2003, pp. 7–13.

Inhoudsopgave

Gebruikte afkortingen

IPEM	Instituut voor Psychoakoestiek en Elektronische Muziek
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
SFT	Short Time Fourier Transform
ECG	Elektrocardiogram
DTW	Dynamic timewarping
USB	Universal Serial Bus
ADC	Analog-to-digital converter
PCM	Pulse-code modulation
UML	Unified Modeling Language

Hoofdstuk 1

Introductie

1.1 Probleemschets

Het probleem dat in deze masterproef zal worden onderzocht doet zich heel specifiek voor bij verschillende experimenten die aan het IPEM worden uitgevoerd. Dit is de onderzoeksinstelling van het departement musicologie aan Universiteit Gent. De focus van het IPEM ligt vooral op onderzoek naar de interactie van muziek op fysieke aspecten van de mens zoals dansen, sporten en fysieke revalidatie. [3]

Om de relatie tussen muziek en beweging te onderzoeken worden er tal van experimenten uitgevoerd. Deze experimenten maken gebruik van allerhande sensoren om bepaalde gebeurtenissen om te zetten in analyseerbare data.

Bij een klassieke experiment wordt onderzocht wat de invloed is van muziek op de lichamelijke activiteit van een persoon. Alle bewegingen worden geregistreerd met een videocamera en verschillende sensoren.

Hierbij moeten minstens drie datastreams worden geanalyseerd: de videobeelden, de data van de accelerometer(s) en de afgespeelde audio. Een uitdaging hierbij is de synchronisatie van deze verschillende datastreams. Om een goede analyse mogelijk te maken is het zeer gewenst dat men exact weet (tot op de milliseconde nauwkeurig) wanneer een bepaalde gebeurtenis in een datastream zich heeft voorgedaan, zodat men deze gebeurtenis

kan vergelijken met de gebeurtenissen in de andere datastreams. Door de verschillen in samplefrequentie en door de latencies van elke opname is dit zeker geen sinecure. [21]

Bij het IPEM maakt men gebruik van een systeem waarbij audio opnames het synchronisatieproces vereenvoudigen. Het principe werkt als volgt: men zorgt ervoor dat elke datastream vergezeld van een perfect gesynchroniseerde audiostream, afkomstig van een opname van het omgevingsgeluid. In het voorgaande experiment is dit eenvoudig te verwezenlijken. Bij de videobeelden kan automatisch een audiospoor mee worden opgenomen. De accelerometer kan geplaatst worden op een microcontroller vergezeld van een kleine microfoon.. Aangezien beide componenten zo dicht op de hardware geplaatst zijn is de latency tussen beide datastromen te verwaarlozen.[1] De afgespeelde audio kan gebruikt worden als referentie, aangezien dit uiteraard al een perfecte weergave is van het omgevingsgeluid.

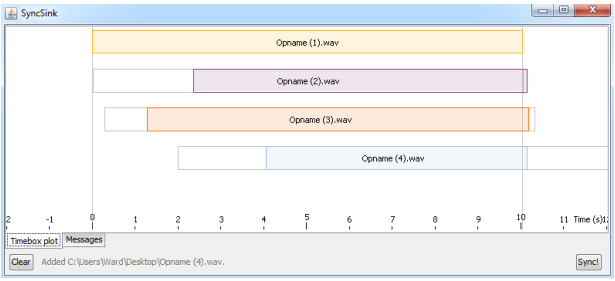
Na het uitvoeren van het experiment beschikt men dus over de gegevens van drie datastreams, waarbij er aan elke datastream een quasi perfect synchrone opname van het omgevingsgeluid is gekoppeld. Aangezien het experiment in één ruimte is uitgevoerd zijn de verschillende opnames van het omgevingsgeluid zeer gelijkend. Het probleem van de synchronisatie van de verschillende datastromen kan bijgevolg gereduceerd worden tot het synchroniseren van de verschillende audiostromen.

Door de typisch eigenschappen van geluid is het niet zo moeilijk om verschillende audiostromen te synchroniseren. Bij het IPEM heeft men een systeem ontwikkeld dat in staat is om verschillende audiostreams te synchroniseren.

Dit systeem heeft in de praktijk echter heel wat beperkingen. De grootste beperking is dat het synchronisatieproces pas kan worden uitgevoerd wanneer het experiment is afgelopen, en dit volledig handmatig. De opgenomen audiobestanden moet worden verzameld op een computer, vervolgens kan met behulp van de audiobestanden de latency van elke datastream worden berekend. Vervolgens kunnen de datastreams worden gesynchroniseerd. Voor de musicologen die deze experimenten uitvoeren is deze werkwijze veel te omslachtig. Daarom is een eenvoudiger realtime systeem om de synchronisatie uit te voeren zeer

[1] De latency van de audioverwerking op een *Axoloti* microcontroller is vastgesteld op 0.333 ms. Meer informatie: `http://www.axoloti.com/more-info/latency/`

gewenst.



Figuur 1.1: Huidige werkwijze om streams te synchroniseren: Een drag and drop interface waarin de opgenomen fragmenten gesleept kunnen worden na afloop van het experiment. Vervolgens wordt de latency berekend.

Een ander probleem is iets vager en minder duidelijk te omschrijven. De resultaten van het kruiscovariantie algoritme bevatten soms afwijkingen die moeilijk te verklaren zijn. De precieze oorzaak hiervan, en hoe dit kan worden opgelost zal ook worden onderzocht. Ook is het kruiscovariantie algoritme in vergelijking met het accoustic fingerprinting algoritme véél gevoeliger voor storingen en ruis, veroorzaakt door slechte opnames. Aangezien de opnameapparatuur (zeker op microcontrollers) bij de uit te voeren experimenten vaak van slechte kwaliteit is, is het belangrijk om de algoritmes voldoende robuust te maken zodat ze hier niet over struikelen.

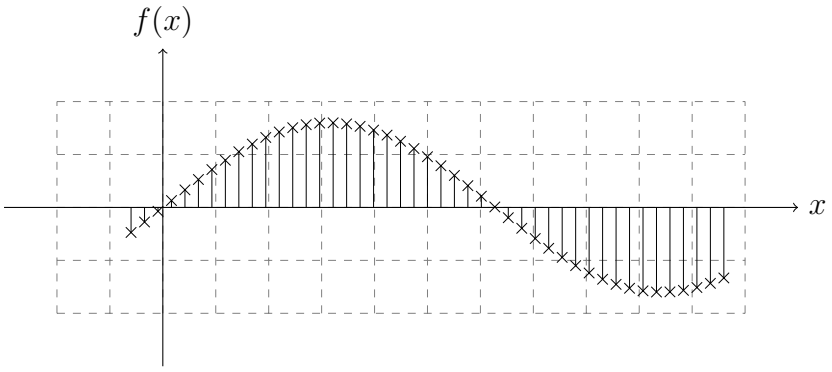
1.2 Digitale audio

Het vervolg van deze scriptie onderstelt dat de lezer een basiskennis heeft inzake digitale audio. In deze inleiding worden de belangrijkste zaken hieromtrent uitgelegd.

Om geluidsgolven digitaal te kunnen verwerken moeten ze worden geconverteerd naar reeksen van discrete waarden. Deze omzetting gebeurt met een ADC: een analog-to-digital converter. De meeste ADC's maken gebruik van de PCM (pulse-code modulation) voorstelling van audio. Bij PCM wordt het analoge signaal op regelmatige tijdstippen gesampled en omgezet in discrete waarden. PCM audio heeft verschillende parameters die een invloed

hebben op de uiteindelijke kwaliteit van de digitale audio. De belangrijkste parameters zijn de samplefrequentie (*sampling rate*) en bitdiepte (*bit depth*).

Figuur 1.2: Samplen van een analoog audiosignaal in de vorm van een sinusgolf. Met toestemming overgenomen van [19].



Samplefrequentie

De samplefrequentie bepaalt het aantal samples per seconde en wordt uitgedrukt in Hertz (Hz). Bij het bepalen van de samplefrequentie is het van belang om rekening te houden met het *bemonsteringstheorema van Nyquist-Shannon*. Deze stelling zegt dat de samplefrequentie minstens dubbel zo hoog moet zijn dan de maximumfrequentie van de te converteren informatie. Bij het gebruik van een lagere frequentie treedt er informatieverlies op. Deze stelling wordt in detail besproken in het originele artikel [15] van Nyquist.

Het menselijk oor is in staat om geluiden te detecteren tussen 20Hz en 20kHz. Om informatieverlies bij het samplen van geluiden binnen dit bereik te voorkomen is het dus vereist om een minimale samplefrequentie te hanteren van $2 \times 20kHz$. De standaard samplefrequentie voor muziek is net iets hoger: $44.1kHz$.

De frequentie van de menselijke stem varieert tussen $30Hz$ en $3000Hz$. De minimale samplefrequentie voor het digitaliseren van een stemopname is dus $6kHz$. In de praktijk wordt meestal een minimum gehanteerd van $8kHz$.

Bitdiepte

De bitdiepte is het aantal bits waarmee elke gesamplede waarde wordt voorgesteld. Meestal wordt er gebruik gemaakt van 16 bit *signed integers*.

De bitdiepte bepaalt het dynamische bereik van audio. Dit is de verhouding tussen het stilst en luidst mogelijk weer te geven volume. Deze verhouding, uitgedrukt in decibel, kan worden berekend met volgende formule:

$$DR = 20 \cdot \log_{10}\left(\frac{2^Q}{1}\right) = (6.02 \cdot Q)dB \tag{1.1}$$

In deze formule staat DR voor het dynamische bereik en Q voor de bitdiepte. Volgens deze formule heeft 16 bit audio een theoretisch dynamische bereik van ongeveer 96 dB. De werkelijke waarde kan hier echter van afwijken door filters die zijn ingebouwd in audiosystemen.

Bovenstaande informatie is gebaseerd op artikel [19], introductievideo [5] en boek [12].

Weergave in software

In computerprogramma's waarin digitale audio verwerkt wordt is het gebruikelijk om een sample voor te stellen als een getal. Afhankelijk van de implementatie kan een 16 bit sample op verschillende manieren worden verwerkt. Enkele mogelijke voorstellingen: signed integer (-32768 tot 32767), unsigned integer (0 tot 65536) of floating point (-1 tot 1). In deze thesis en de bijhorende software wordt altijd de floating point notatie gebruikt.

1.3 Evaluatiecriteria

Het te ontwikkelen systeem moet voldoen aan heel wat vereisten. In deze sectie zullen de vereisten eenduidig geformuleerd en besproken worden.

Realtime synchronisatie

Een cruciale vereiste is dat de toepassing in *realtime* moet kunnen werken. Concreet wil dit zeggen dat het tijdens het uitvoeren van het experiment mogelijk moet zijn om de huidige latencies van de streams op te vragen.

Het is moeilijk om het opvragen van deze gegevens echt in realtime mogelijk te maken. Veel algoritmes vereisen een bepaalde hoeveelheid aan data voordat de latency berekend kan worden. Daarom moeten de streams gebufferd worden, wat er toe leidt dat het systeem niet meer realtime is in de enge zin van het woord. Om een realtime systeem zo goed mogelijk te benaderen wordt een beperking opgelegd: een buffer met als maximumgrootte de hoeveelheid data verzameld in tien seconden. Deze tijd is de maximaal mogelijke achterstand ten opzichte van de realtime latency.

Detecteren van gedropte samples

De beperkte resources van een microcontroller kan voor problemen zorgen bij het verwerken van streams. Zo kan het gebeuren dat er gegevens van streams verloren gaan, in het vakjargon worden dit ook wel *gedropte samples* genoemd. Bij de synchronisatie leidt dit probleem tot een plotse verhoging van de latency. Hoewel het onmogelijk is om de gedropte samples te reconstrueren is het wel gewenst dat de wijziging in latency gedetecteerd wordt en dat hiermee wordt rekening gehouden bij de verdere verwerking. De snelheid waarmee dit probleem gedetecteerd kan worden hangt eveneens af van de manier waarop er gebufferd wordt. Een detectie is mogelijk vanaf het moment dat de buffer voor meer dan de helft gevuld is met data gegenereerd na het gegevensverlies. Rekening houdend met het eerste criterium zou een wijziging van de latency binnen vijf seconden gedetecteerd moeten kunnen worden.

Detecteren van drift

Elke stream heeft een bepaalde samplefrequentie. Het is belangrijk dat de samplefrequentie gekend is om de gegevens correct en precies te kunnen verwerken. Het kan echter voorvallen

dat de samplefrequentie bij de verwerking op microcontrollers minder nauwkeurig gekend is. Een stream waarbij de samplefrequentie $1Hz$ afwijkt van de theoretische waarde zal na 60 seconden een latency hebben opgebouwd van 60 samples. Bij een samplefrequentie van 8000 Hz komt dit overeen met 7,5 ms.[2] Dit probleem mag zeker niet worden verwaarloosd.

1.4 Bestaande methoden

Er bestaan verschillende methoden om datastreams te synchroniseren. Welke methode te verkiezen is hangt volledig af van de toepassing.

In deze sectie komen de belangrijkste methoden aan bod en zullen ze worden getoetst aan de eerder beschreven evaluatiecriteria.

1.4.1 Event-gebaseerde synchronisatie

Deze methode wordt beschreven in [6, 21] en is een eenvoudige, intuïtieve methode om synchronisatie van verschillende datastreams uit te voeren. De synchronisatie gebeurt aan de hand van markeringen die in de verschillende streams worden aangebracht. In audiostreams kan een kort en krachtig geluid een markering plaatsen. Een lichtflits kan dit realiseren in videostreams. De latency wordt bepaald door het verschil te berekenen tussen de tijdspositie van de markeringen in de streams. De synchronisatie kan vervolgens zowel manueel als softwarematig worden uitgevoerd.

Deze methode kent heel wat beperkingen. Zo vormt bij de synchronisatie van een groot aantal streams de schaalbaarheid een probleem. Ook wanneer er in een stream samples gedropt worden of er drift ontstaat, leidt dit tot foutieve synchronisatie. De methode kan deze twee problemen niet detecteren tot er opnieuw markeringen worden aangebracht en de streams gesynchroniseerd worden. Verder laten ook niet alle sensoren toe om markeringen aan te brengen: zo is de synchronisatie van een ECG onmogelijk met deze methode.

[2]Berekening: $60/8000Hz = 0.0075s = 7.5ms$

Het handmatig synchroniseren met behulp van deze methode blijkt derhalve in een realtime situatie niet mogelijk. Wanneer de synchronisatie echter door software wordt uitgevoerd is deze methode wel in realtime bruikbaar. In dat geval moet er per tijdsinterval een markering worden aangebracht om de problemen veroorzaakt door drift en gedropte samples te overbruggen.

1.4.2 Synchronisatie met een kloksignaal

Artikel [13] beschrijft een methode waarbij door een kloksignaal realtime streams van verschillende soorten toestellen worden gesynchroniseerd. Hiervoor gebruikt men standaard audio en video synchronisatieprotocollen. Elk toestel kan gebruik maken van verschillende samplefrequenties en communicatieprotocollen.

De methode gebruikt een *master time code* signaal dat verstuurd wordt naar elk toestel. Dit laat het realtime analyseren van elke stream toe. Bij deze analyse kan vervolgens meteen de samplefrequentie en latency bepaald worden.

Een groot nadeel van dit systeem is dat elk toestel een kloksignaal als input moet kunnen toelaten en verwerken. In het geval van de verwerking van videobeelden kan deze methode enkel gebruikt worden met zeer dure videocamera's waarbij de sluitertijd gecontroleerd kan worden. Bij goedkopere camera's (zoals's webcams) moet men op zoek gaan naar alternatieven. [21]

1.4.3 Dynamic timewarping

Dynamic timewarping (DTW) is een techniek die gebruikt wordt voor het detecteren van gelijkenissen tussen twee tijdreeksen[3]. Aangezien een gedigitaliseerde audiostream een tijdreeks is kan deze techniek worden aangewend om de latency te bepalen tussen gelijkaardige opnames van het omgevingsgeluid. In de probleemschets (1.1) is er uitgelegd hoe datastreams met behulp van het omgevingsgeluid gesynchroniseerd kunnen worden.

[3]Een tijdreeks is een sequentie van opeenvolgende datapunten over een continu tijdsinterval, waarbij de datapunten elk baar na telkens hetzelfde interval opvolgen.

DTW is een algoritme dat op zoek gaat naar de meest optimale *mapping* tussen twee tijdreeksen. Hierbij wordt gebruik gemaakt van een padkost. De padkost wordt bepaald door de manier waarop de tijdreeksen niet-lineair worden kromgetrokken ten opzichte van de tijdas[18]. De minimale kost kan in kwadratische tijd berekend worden door gebruik te maken van dynamisch programmeren [10]. DTW is een veelgebruikte techniek in domeinen zoals spraakherkenning, bio-informatica, data-mining, etc [17].

Aangezien DTW het toelaat om tijdreeksen krom te trekken is het gewenst dat zowel het verleden als de toekomst van de streams voor het algoritme toegankelijk is. Een uitbreiding op dit algoritme beschreven in [10] laat toe één tijdreeks in realtime te streamen mits de andere stream op voorhand is gekend. Toch houdt deze uitbreiding geen oplossing in voor het gestelde probleem. Alle streams komen immers in realtime toe en we willen zo snel mogelijk de latency tussen de streams achterhalen.

Het bufferen van de binnenkomende streams en vervolgens het DTW algoritme uit te voeren op de buffers leek een mogelijke manier om dit probleem te omzeilen.

Of het algoritme na deze aanpassing voldoet aan onze vereisten diende een klein experiment uit te wijzen. De resultaten hiervan zijn te vinden in appendix A: .

Het experiment toonde evenwel aan dat DTW niet bruikbaar is voor de realtime stream synchronisatie. De resultaten bleken niet nauwkeurig genoeg, zeker niet wanneer ook de performantie van het algoritme in beschouwing werd genomen.

1.4.4 Accoustic fingerprinting

Accoustic fingerprinting is een techniek die in staat is om gelijkenissen te vinden tussen verschillende audiofragmenten. Het is eveneens mogelijk om de latency tussen de audiofragmenten te bepalen. Net zoals bij DTW kan dit algoritme gebruikt worden om datastreams te synchroniseren met behulp van het omgevingsgeluid.

De techniek van accoustic fingerprinting extraheert en vergelijkt fingerprints van audiofragmenten. Een accoustic fingerprint bevat gecondenseerde informatie gebaseerd op typische eigenschappen van het audiofragment. De kracht van dit algoritme schuilt in haar snelheid

en robuustheid. Het is immers uitzonderlijk bestand tegen achtergrondgeluiden en ruis. Door deze eigenschappen is het algoritme in staat om in enkele seconden een database met miljoenen fingerprints van audiofragmenten te doorzoeken. De bekendste toepassing van accoustic fingerprinting is de identificatie van liedjes op basis van een korte opname[4].

Het is onder meer deze techniek die het IPEM gebruikt om de opgenomen audiostreams van experimenten te synchroniseren. In tegenstelling tot *Shazam* wordt er niet op zoek gegaan naar matches in een database maar worden ze gezocht tussen de opgenomen audiofragmenten. Het uitgangspunt is immers dat er tussen de opnames gelijkenissen moeten gevonden kunnen worden.

Door haar snelheid en robuustheid lijkt dit algoritme te voldoen aan de vereisten om datastreams realtime te kunnen synchroniseren. Het is wel noodzakelijk dat de streams gebufferd worden alvorens het algoritme kan starten. Drift en gedropte samples kunnen gedetecteerd worden door het algoritme iteratief op korte gebufferde fragmenten uit te voeren. Na elke iteratie kan een eventuele wijziging worden opgemerkt. Zie 2.1.1 voor een meer gedetailleerde bespreking van dit algoritme.

1.4.5 Kruiscovariantie

De laatste methode is net zoals de twee vorige methodes in staat om de latency tussen audiofragmenten te bepalen. Deze methode kan daarom ook aangewend worden om datastreams met behulp van opnames van het omgevingsgeluid te synchroniseren.

Kruiscovariantie (ook wel kruiscorrelatie genoemd) berekent de gelijkenis tussen twee audiofragmenten sample per sample en kent een getal toe aan de mate waarin de fragmenten overeenkomen. Door deze berekening voor elke verschuiving uit te voeren kan de latency tussen de fragmenten bepaald worden.

Deze methode is eveneens toepasbaar op realtime streams door gebruik te maken van buffering. Het iteratief uitvoeren van het algoritme op de opeenvolgende buffers zorgt

[4]Het grootste voorbeeld hiervan is de smartphone app Shazam. Deze app is de eerste toepassing dat gebruik maakte van dit algoritme.

ervoor dat gedropte samples en drift gedetecteerd kunnen worden.

In sectie 2.1.2 wordt deze materie verder in detail behandeld.

1.5 Doel van deze masterproef

Dit onderzoek wil drie zaken bereiken:

Selectie en optimalisatie van algoritmes

Er wordt op zoek gegaan naar de algoritmes waarmee het probleem kan worden opgelost. Verder dienen de algoritmes en bijhorende parameters te worden geoptimaliseerd om in deze toepassing zo efficiënt mogelijk te presteren. Indien mogelijk zal er worden geprobeerd om de algoritmes waarvan er bij het IPEM al een implementatie beschikbaar is te hergebruiken.

Het beoogde doel is dat de algoritmes in staat zijn om audio opgenomen met een basic microfoon op een microcontroller te synchroniseren met een nauwkeurigheid van minstens één milliseconde.

Ontwerp en implementatie van een softwarebibliotheek

Het tweede doel van het onderzoek betreft het schrijven van een softwarebibliotheek. Deze bibliotheek zal gebruik maken van de geoptimaliseerde algoritmes om de audiostromen te synchroniseren. Deze bibliotheek moet vanuit andere software kunnen worden opgeroepen en gedetailleerde informatie teruggeven over de synchronisatie van de verschillende audiostreams.

Ontwerp en implementatie van een gebruiksvriendelijke interface

Uiteindelijk is het de bedoeling dat dit onderzoek resulteert in een gebruiksvriendelijke applicatie die toegankelijk is voor onderzoekers/musicologen zonder uitgebreide informatica kennis. De software moet in staat zijn om van verschillende binnenkomende datastromen

(vergezeld met audiostream) te synchroniseren en op één of andere manier weg te schrijven naar een persistent medium.

Hoofdstuk 2

Methode

2.1 Algoritmen

In sectie 1.4 van deze scriptie zijn de voornaamste methoden waarmee datastreams gesynchroniseerd kunnen worden beknopt besproken. Hoewel de meeste algoritmen niet voldeden aan de vereisten bleken er twee toch zeer geschikt voor snelle en nauwkeurige synchronisatie van realtime streams. In dit gedeelte zullen deze methoden in detail worden behandeld. Ook wordt er onderzocht in welke mate het mogelijk is om deze algoritmes te combineren tot één systeem.

2.1.1 Accoustic fingerprinting

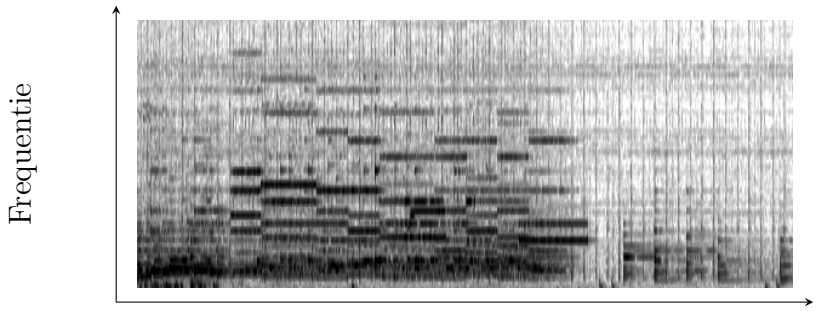
Bij het accoustic fingerprinting algoritme worden fingerprints geëxtraheerd uit audiofragmenten. Het zoek naar gelijkenissen gebeurt door de fingerprints met elkaar te vergelijken.

Features

Een cruciale stap bij de ontwikkeling van een accoustic fingerprinting systeem is het bepalen van een betrouwbare *feature* om de fingerprints op te baseren. Een feature is een kenmerk waarmee het mogelijk is om audiofragmenten van elkaar te onderscheiden. Mo-

gelijke features zijn bijvoorbeeld *onsets*[1] of frequentie. Een andere zeer goed bruikbare feature zijn de *spectrale pieken* in het tijd-frequentie spectrum van de geluidsfragmenten. Deze feature is compact op te slaan en bevat veel informatie over het opgenomen audiofragment. Hierdoor wordt de kans kleiner dat fingerprints gematcht kunnen worden zonder dat ze daadwerkelijk gebaseerd zijn op hetzelfde geluid.

Figuur 2.1: Spectrogram van *Talk Talk - New Grass*. De donkere vlekken zijn pieken zijn frequentie-intervallen die aan een relatief hoge energie voorkomen.



Werking

Een accoustic fingerprinting systeem gebaseerd op de extractie van spectrale pieken gaat in verschillende stappen te werk:

Eerst wordt het tijdsignaal (de typische golfvorm) van elk geluidsfragment omgezet tot een verzameling functies in het frequentiedomein. Deze omzetting gebeurt met het *Fast Fourier Transformation* algoritme (FFT). Het tijdsignaal wordt in kleine stukjes onderverdeeld (standaardgrootte: 512 samples, zie 3.2.2). Elk stukje audio wordt opgeslagen in een buffer waarop vervolgens het FFT algoritme op wordt uitgevoerd[2]. De opeenvolgende buffers worden genummerd met een *buffer index*. De inhoud van de buffer kan gezien

[1]Een onset is een markering in de tijd die het begin van een piek aanduidt. In artikel [8] wordt de betekenis en detectie van onsets uitgebreid beproken.

[2]Het uitvoeren van een FFT op zo'n klein stukje audio wordt ook wel de *Short Time Fourier Transformation* of SFT genoemd

worden als een signaal in het tijddomein. Het resultaat van het FFT algoritme is de fourier getransformeerde van dit signaal: een reeks van een eindig aantal frequentie-intervallen. Elke frequentie-interval is genummerd met een *bin index*. De verzameling van alle fourier getransformeerden stelt het audiofragment voor in het tijd-frequentie domein.

De grafische voorstelling van deze verzameling functies wordt het spectrogram genoemd. Een spectrogram is het duidelijkst wanneer op de x-as de tijd en op de y-as de frequentie wordt weergegeven. De intensiteit waarmee een bepaalde frequentie voorkomt kan worden aangeduid door gebruik te maken van verschillende kleuren of contrasten. Figuur 2.1 toont een spectrogram waarbij frequentie met een hoge intensiteit donkerder zijn weergegeven.

In artikel [16] wordt het FFT algoritme uitgebreid besproken.

Na het omzetten van de te vergelijken geluidsfragmenten naar hun tijd-frequentie representatie kan er naar kandidaat-pieken worden gezocht. Dit zijn lokale maxima waarbij de hoeveelheid energie waarmee de frequentie voorkomt hoger is dan bij zijn buren [20]. In het spectrogram kan elk donker vlekje gezien worden als een kandidaat-piek.

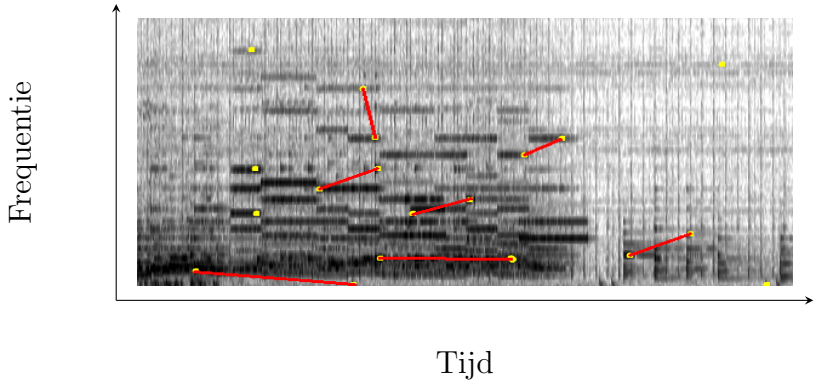
Wanneer deze stap is afgerond kunnen de fingerprints bepaald worden. Een fingerprint is een de verbinding tussen twee spectrale pieken. Welke kandidaat-pieken gebruikt zullen worden in fingerprints hangt af van de implementatie van het algoritme en de ingestelde parameters. Enkele parameters die hier invloed op hebben zullen in sectie 3.2.2 van deze scriptie besproken worden. Figuur 2.2 toont een spectrogram met waarop enkele kandidaat-pieken en fingerprints zijn aangeduid.

Na het bepalen van de fingerprints worden ze opgeslagen in een datastructuur waarin snel naar matches kan worden gezocht. Om dit mogelijk te maken moeten er van de fingerprints enkele typerende getallen bepaald worden.

- $f1$ en $f2$: de frequentie van de spectrale pieken van de fingerprint.
- $t1$ en $t2$: de tijd van de spectrale pieken van de fingerprint.
- Δf: het verschil van de frequenties van beide spectrale pieken van de fingerprint.
- Δt: het verschil van de tijd van beide spectrale pieken van de fingerprint.

Figuur 2.3 toont een schematische voorstelling van een fingerprint waarop deze getallen

Figuur 2.2: De kandidaat-pieken (gele stipjes) en fingerprints (rode lijnen) van *Talk Talk - New Grass*.



zijn aangeduid.

Bij het zoeken naar matches kan er gesteund worden op enkele typische eigenschappen van fingerprints:

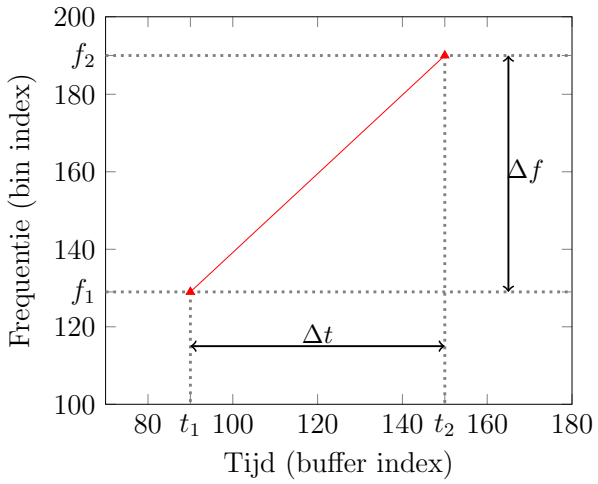
Twee overeenkomende fingerprints uit twee geluidsfragmenten zullen dezelfde frequenties ($f1$ en $f2$) hebben. Bijgevolg is ook het verschil in frequentie (Δf) gelijk.

De tijd van de spectrale pieken ($t1$ en $t2$) komen meestal niet overeen. Bij Shazam is het bijvoorbeeld geen vereiste om een opname te maken vanaf het begin van een liedje. Het moment van de opname mag volledig willekeurig worden gekozen. Bij het synchroniseren van streams wordt gezocht naar het verschil tussen de begintijden ($t1$ van elke fingerprint) van de overeenkomstige fingerprints van de audiofragmenten. Dit tijdverschil is wel gelijk bij elk paar overeenkomende fingerprints.

Hoewel de tijd ($t1$ en $t2$) van twee fingerprints meestal verschilt is dit niet het geval voor het verschil ervan (Δt). Bij twee overeenkomende fingerprints van twee audiofragmenten is het verschil in frequentie inherent gelijk.

Uit voorgaande eigenschappen kan geconcludeerd worden dat fingerprints uit twee audiofragmenten matchen wanneer $f1$, Δf en Δt gelijk zijn. Om deze parameters snel met elkaar kunnen te vergelijken wordt er een berekening uitgevoerd die deze parameters omzet in één enkel getal. Dit getal wordt de hash van de fingerprint genoemd. Samen met

Figuur 2.3: De anatomie van een fingerprint in het tijd-frequentie domein. De rode lijn stelt de fingerprint voor tussen twee (niet afgebeelde) spectrale pieken. De typische parameters van de fingerprint zijn aangeduid op de assen. Met toestemming overgenomen uit artikel [21].



deze hash wordt ook $t1$ en een identificatie van het geluidsfragment bijgehouden.

Artikel [20] geeft meer informatie over de omzetting van deze drie getallen tot een hash.

Een fingerprint kan bijgevolg gezien worden als verzameling gegevens met de volgende structuur: $(id; t1; hash(f1; \Delta f; \Delta t))$. Het zoeken naar fingerprints met overeenkomstige hashwaarden is mogelijk in $O(1)$ door gebruik te maken van een hashtabel. De precieze werking hiervan valt buiten de scope van deze scriptie.

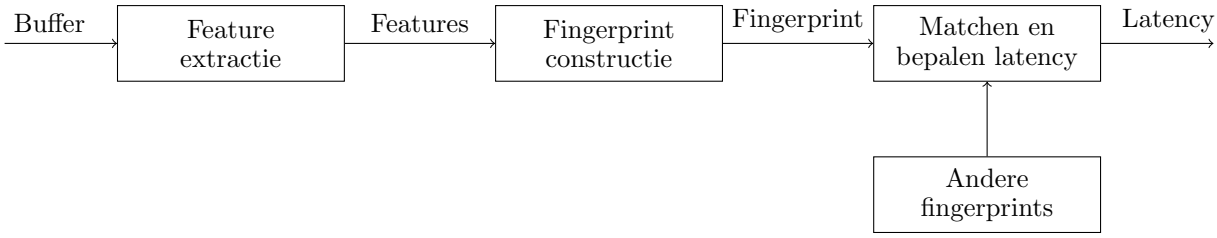
Om te bepalen of twee audiofragmenten wel degelijk overeenkomen wordt er gezocht naar alle fingerprints met een overeenkomende hashwaarde. Van elk paar overeenkomende fingerprints wordt het verschil tussen $t1$ berekend. Dit verschil wordt de offset genoemd. Het vinden van een groot aantal matches met dezelfde offset wijst op een sterke gelijkenis tussen de audiofragmenten. De precieze waarde van "een groot aantal" wordt bepaald door een parameter van het algoritme.

Bepalen van de latency

Accoustic fingerprinting kan gebruikt worden om streams te synchroniseren door de ze eerst te bufferen. Wanneer een buffer volledig is opgevuld kan deze net zoals een kort audiofragment worden verwerkt door het algoritme. De latency tussen streams wordt bepaald door de offset die in vorig paragraaf werd beschreven: het verschil tussen de $t1$ waarden stelt namelijk de verschuiving tussen de geluidsfragmenten voor.

Figuur 2.4 toont schematisch alle stappen die moeten worden doorlopen om met behulp van accoustic fingerprinting audiostreams te synchroniseren.

Figuur 2.4: Schematische voorstelling van synchronisatie met behulp van een accoustic fingerprinting systeem.



Een uitgebreidere beschrijving is te vinden in artikel [24]. De methode die in het artikel en deze scriptie besproken werd is beperkt tot het vergelijken van audiofragmenten die in tijd noch toonhoogte gewijzigd zijn. Aan het IPEM is een aangepaste methode ontwikkeld die dit wel toelaat [20].

Nauwkeurigheid

Zowel de snelheid waarmee wijzigingen van de latency bepaald kunnen worden als de nauwkeurigheid van de latency zelf hangt af van heel wat verschillende parameters van het algoritme.

De detectiesnelheid is vooral afhankelijk van de buffergrootte waarop het algoritme wordt uitgevoerd. Met deze instelling moet echter omzichtig worden omgegaan: een te kleine buffergrootte kan er toe leiden dat het algoritme niet meer in staat is om voldoende matches

te vinden. Het kan helpen om andere parameters te wijzigen waardoor het vinden van een groot aantal matches gegarandeerd blijft. Deze parameters worden in sectie 3.2.2 in detail besproken.

De nauwkeurigheid van de latency van het algoritme hangt af van de parameters van het FFT algoritme. Een nauwkeurigheid van 16 ms of 32 ms is standaard. De precieze werking van het FFT algoritme valt buiten de scope van deze scriptie.

2.1.2 Kruiscovariantie

Deze methode bepaalt de gelijkenis tussen twee audiofragmenten en resulteert in één getal. Dit getal is een soort van score die aangeeft in welke mate twee signalen overeenkomen. De latency tussen twee audiofragmenten kan bepaald worden door deze berekening uit te voeren voor **elke mogelijke verschuiving**. De verschuiving waarbij het resulterend getal het hoogst is bepaalt de latency.

Werking

Stel twee audioblokken a en b bestaande uit een gelijk aantal samples (n). Deze audioblokken worden telkens cyclisch één sample verschoven tot wanneer de kruiscovariantie waarde (k) voor elke mogelijke verschuiving berekend werd. De variabele i stelt de huidige verschuiving voor en gaat van 0 tot n. De kruiscovariantie wordt berekend met formule:

$$k = \sum_{j=0}^{n} a_j \cdot b_{(i+j) \ mod \ n} \tag{2.1}$$

De waarde van i waarbij de kruiscovariantie het hoogst is stelt de latency voor tussen beide audioblokken in aantal samples. De latency in seconden kan bepaald worden door dit resultaat te delen door de samplefrequentie.

De methode kan de latency **tot op één sample nauwkeurig** bepalen. De maximaal bereikbare nauwkeurigheid hangt dus af van de samplefrequentie van de audioblokken. Bij

een samplefrequentie van $8000Hz$ is dit $1/8000Hz = 0.125ms$. Dit is ruim voldoende voor het huidige probleem.

Een nadeel aan deze methode is de performantie. Het berekenen van de beste kruiscovariantie van twee audioblokken bestaande uit n samples kan gebeuren in $O(n^2)$. Het is dus belangrijk om bij deze berekening de grootte van de audioblokken te beperken.

In artikel [21] wordt deze techniek meer in detail besproken.

Toepassing in realtime

Het bufferen van de audiostreams maakt ook dit algoritme in realtime toepasbaar. In tegenstelling tot accoustic fingerprinting is het niet de bedoeling dat de berekeningen op de volledige buffer wordt uitgevoerd. Door de kwadratische tijdscomplexiteit zou het algoritme onnoemelijk veel rekenkracht vragen.[3] Er moet dus een manier gevonden worden waarmee het mogelijk is om het aantal samples waarop het algoritme wordt uitgevoerd beperkt wordt.

2.1.3 Toepasbaarheid

Het accoustic fingerprinting algoritme is zeer snel en robuust en kan gebruikt worden om gebufferde audiostreams te synchroniseren tot enkele tientallen milliseconden nauwkeurig (afhankelijk van de parameters van het FFT algoritme).

Het kruiscovariantie algoritme kan eveneens gebruikt worden om (gebufferde) audiostreams te synchroniseren. De grootste troef van dit algoritme is haar nauwkeurigheid: in de beste omstandigheden kan het algoritme resultaten bekomen tot op één sample nauwkeurig. Het bereiken van een dergelijke nauwkeurigheid is onmogelijk met eender welk ander besproken algoritme. De keerzijde is de performantie van het algoritme. Bij het synchroniseren van grote audioblokken kan dit problematisch zijn.

[3] Voor het berekenen van de kruiscovariantie tussen twee buffers met $10s$ audio en een samplefrequentie van $8000hz$ zijn er asymptotisch $6.4 \cdot 10^9$ berekeningen vereist.

De kenmerken van deze algoritmen zijn complementair. De gemakkelijkste manier om een robuust, snel én nauwkeurig systeem op te bouwen is door het beste van de twee werelden te combineren. Het accoustic fingerprinting algoritme kan zorgen voor de synchronisatie tot op enkele tientallen milliseconden nauwkeurig. Dit resultaat laat toe dat we het kruiscovariantie algoritme kunnen uitvoeren op zeer korte stukjes audio (een honderdtal milliseconden volstaat).

2.2 Bufferen van streams

Aangezien de algoritmes een bepaalde hoeveelheid audio nodig hebben vooraleer ze kunnen worden uitgevoerd is het noodzakelijk om de streams eerst te bufferen. Dit proces moet herhaald worden aangezien er mogelijk samples gedropt worden of drift kan ontstaan. In dit deel zal worden uitgelegd hoe het bufferen precies in zijn werk gaat. Om verwarring met andere soorten buffers te vermijden zal dit type buffer verder in deze scriptie een *streambuffer* genoemd worden.

Buffergrootte

De grootte van de buffer heeft invloed op de kwaliteit van de resultaten. Het spreekt voor zich dat het algoritme beter kan presteren wanneer er 10 seconden in plaats van 1 seconde audio geanalyseerd wordt. Een nadeel is echter dat het langer duurt vooraleer een wijzing van de latency gedetecteerd kan worden.

Naïeve implementatie

Indien er buffers gebruikt worden die t aantal seconden audio kunnen bevatten, dan zal het bij een naïeve implementatie in het slechtste geval pas mogelijk zijn om een wijziging van de latency na $\frac{3}{2}t$ seconden te detecteren. Dit is als volgt te verklaren: Een wijziging van de latency kan gedetecteerd worden wanneer meer dan de helft van de buffer gevuld is met audio met de nieuwe latency. Wanneer er samples gedropt worden net na het moment

dat de buffer voor de helft gevuld is ($\frac{1}{2}t$), dan zal het algoritme uitgevoerd op de huidige buffer de wijziging niet kunnen detecteren. De volgende buffer zal wel gevuld zijn audio met de nieuwe latency, het duurt echter nog een bijkomende t seconden vooraleer deze buffer gevuld is. De detectietijd bedraagt bijgevolg in het slechtste geval dus $\frac{3}{2}t$ seconden.

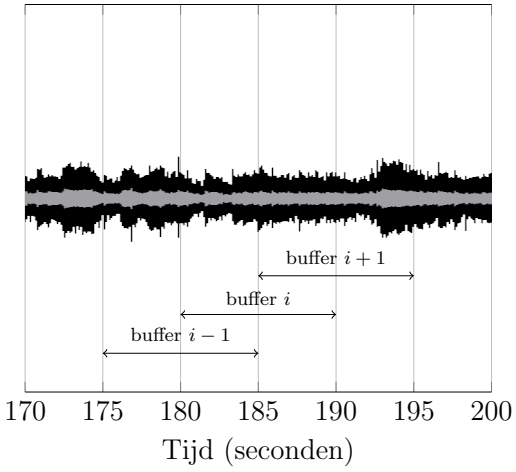
De naïeve implementatie kan een wijziging van de latency in het beste geval na $\frac{1}{2}t$ seconden detecteren. Wanneer er samples gedropt worden net voor het moment dat de buffer voor de helft gevuld is, dan kan het algoritme de nieuwe latency wel onmiddellijk detecteren.

Sliding window

Een meer doordachte manier van bufferen maakt gebruik van een *sliding window*. In onderstaande beschrijving wordt gebruik gemaakt van een buffer met t seconden capaciteit en een stapgrootte van s seconden, hierbij geldt dat $s \leq t$.

Het verschil met de naïeve methode is dat de buffer niet pas na t seconden wordt opgeschoven. Door de buffer al na s seconden op te schuiven zal een wijziging van de latency sneller gedetecteerd kunnen worden; dit terwijl het algoritme toch nog steeds t seconden audio kan analyseren. In figuur 2.5 wordt grafisch weergegeven hoe de buffer precies verschoven wordt met $t = 10$ en $s = 5$.

Figuur 2.5: Schematische weergave van een *sliding window* buffer over een audiostream.



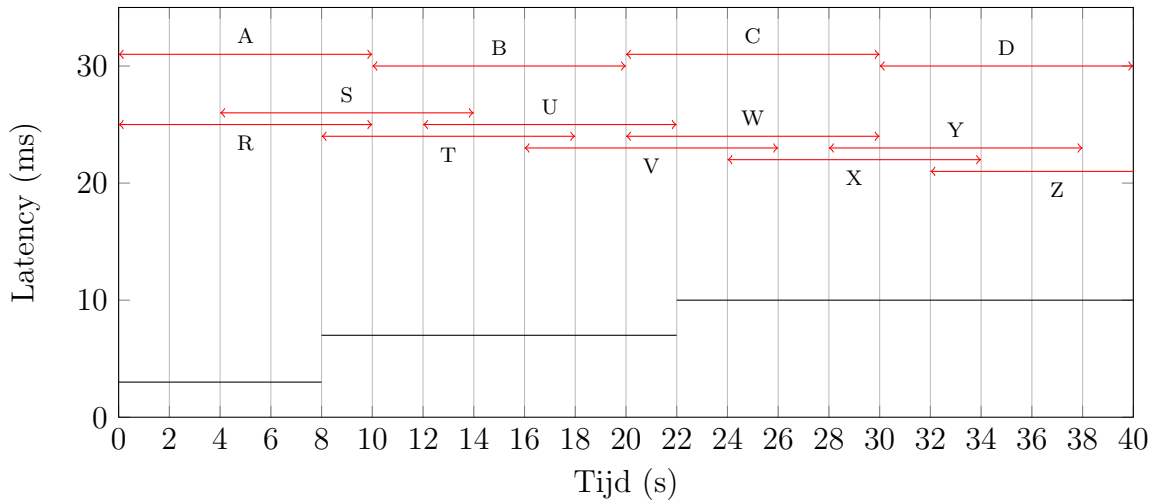
Door de buffer al na s seconden op te schuiven wordt het slechtste geval sterk verbeterd. In het slechtste geval wordt een wijziging van de latency gedetecteerd na $\frac{t}{2} + s$ seconden. Het beste geval blijft wel nog steeds $\frac{t}{2}$ seconden.

Het verkleinen van de stapgrootte zorgt ervoor dat het algoritme per hoeveelheid audio frequenter moet worden uitgevoerd. Een te kleine stapgrootte heeft bijgevolg een negatieve invloed op de performantie.

Voorbeeld

Een praktisch voorbeeld zal bovenstaande beschrijving wat verduidelijken. In het voorbeeld worden twee audiostreams van 40 seconden geanalyseerd. Door het droppen van samples neemt de latency tussen de streams stapsgewijs toe. Figuur 2.6 toont in het zwart hoe de latency gedurende de verwerking evolueert. De opeenvolgende buffers van de twee besproken methode's worden in het rood aangeduid.

Figuur 2.6: Grafisch weergave van de methode's waarop gebufferd kan worden. De zwarte lijn stelt de huidige latency voor. In het rood worden de opeenvolgende buffers weergegeven.



De initiële latency van 3 milliseconden wordt zowel met de naïeve methode als met het sliding window gedetecteerd na de analyse van de allereerste buffer (A of R) 10 seconden na aanvang van de analyse. De eerste verhoging tot 7 milliseconden vindt te laat plaats

om gedetecteerd te kunnen worden door de eerste buffer van beide methodes. Bij deze verhoging van de latency wordt het verschil tussen beide methodes zichtbaar: bij de sliding window methode vindt de detectie 6 seconden na de wijziging plaats. Bij de naïeve methode moet er echter gewacht worden tot wanneer buffer B is volgelopen 12 seconden na de wijziging. De tweede verhoging naar 10 milliseconden wordt zowel door de naïeve methode als door de sliding window methode gedetecteerd 8 seconden na de wijziging (buffer C of W).

Conclusie

De detectiesnelheid van een latencywijziging hangt af van twee parameters: de bufferlengte (t) en de staplengte (s). De snelheid waarmee een wijziging gedetecteerd kan worden (T) kan als volgt worden samengevat:

$$\frac{t}{2} < T < \frac{t}{2} + s \tag{2.2}$$

Het toepassen van deze formules op het vorige paragraaf levert volgende resultaten: Zowel bij de naïeve als sliding window methode is de ondergrens 5 seconden. De bovengrens bij de naïeve methode bedraagt 15 seconden. Bij de sliding window methode is dit begrenst tot 9 seconden.

Hoofdstuk 3

Implementatie

3.1 Technologieën en software

3.1.1 Java 7

Er zijn verschillende redenen waarom er gekozen is om de synchronisatie bibliotheek in Java te implementeren. De belangrijkste reden is dat de bestaande audio bibliotheken (Panako en TarsosDSP) van het IPEM ook ontwikkeld zijn in Java. Deze kunnen enkel aangeroepen worden via andere Java applicaties.

Het is ook de bedoeling is om de gebruikersinterface te ontwikkelen met behulp van Max/MSP modules. Het ontwikkelen van dergelijke modules is mogelijk in twee programmeertalen: C en Java. De eenvoud en hoge graad van abstractie gaf de doorslag om hierbij Java te gebruiken. De voordelen die C biedt op vlak van snelheid wegen (in deze toepassing) hier niet tegenop.

Hoewel de laatste versie van Max/MSP (7) het toelaat om Java 8 te gebruiken wordt deze versie in dit project om compatibiliteitsredenen vermeden.

3.1.2 JUnit

JUnit is een unit testing framework voor Java. JUnit heeft in dit onderzoek een belangrijke rol gespeeld bij het bepalen van de optimale parameters van de verschillende algoritmen. JUnit maakte het mogelijk om de algoritmes herhaald maar met verschillende parameters op een dataset los te laten. JUnit liet toe om in één oogopslag te zien hoe het algoritme heeft gepresteerd.

3.1.3 TarsosDSP

TarsosDSP is een Java bibliotheek voor realtime audio analyse en verwerking ontwikkeld aan het IPEM. De bibliotheek bevat een groot aantal algoritmes voor audioverwerking en kan nog verder worden uitgebreid. Deze bibliotheek wordt beschreven in artikel [22].

TarsosDSP is voornamelijk gebouwd rond het concept *processing pipeline*. Dit is een abstractie van een audiostream die via programmacode verwerkt kan worden. Een processing pipeline wordt voorgesteld als instantie van de klasse `AudioDispatcher`. Een `AudioDispatcher` kan aangemaakt worden van een audiobestand, een array van floating-point getallen of een microfoon en kan bewerkt of verwerkt worden met behulp van één of meerdere `AudioProcessors`. Objecten van klassen die de interface `AudioProcessor` implementeren kunnen aan de processing pipeline worden toegevoegd. In de implementatie van de `process` methode kan de audiostream op een bepaalde manier verwerkt, geanalyseerd of gewijzigd worden.

TarsosDSP bevat verder nog een groot aantal klassen met allerhande tools en audioverwerkings algoritmen. Een greep uit de features van TarsosDSP:

- Toevoegen van geluidseffecten (delay, flanger,...)
- Toevoegen van filters (low-pass, high-pass, band-pass,...)
- Conversie tussen verschillende formaten
- Toonhoogte detectie
- Wijzigen van de samplefrequentie

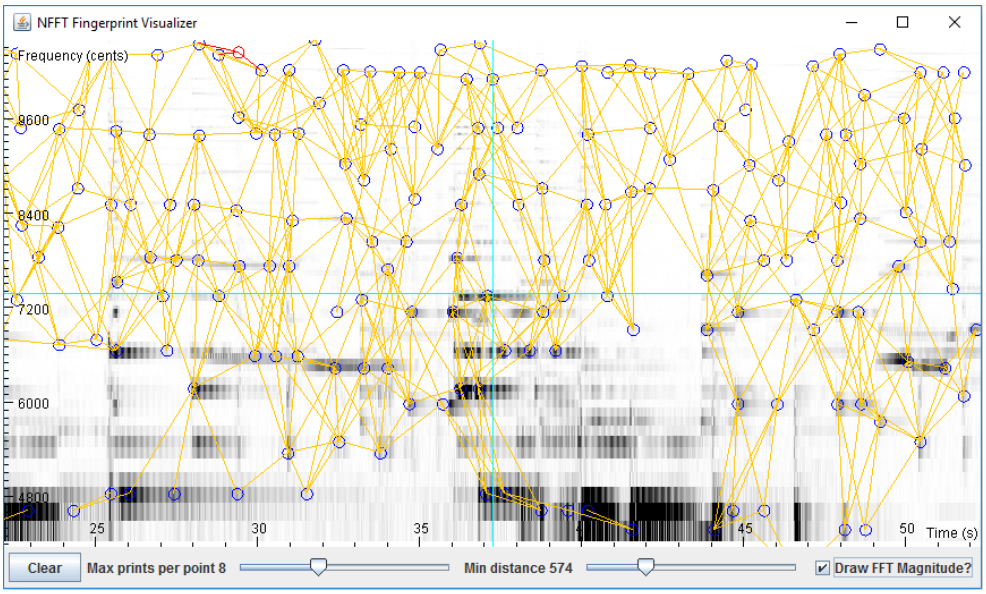
3.1.4 Panako

Panako is net zoals TarsosDSP een Java bibliotheek (door de zelfde auteur) ontwikkeld aan het IPEM. Panako bevat een arsenaal aan algoritmen voor het matchen of synchroniseren van audiofragmenten of audiostreams. Deze bibliotheek wordt uitgebreid beschreven in artikel [20].

Dit onderzoek gebruikt Panako's open-source implementatie van het accoustic fingerprinting algoritme dat beschreven is in artikel [24]. Panako bevat ook een uitbreiding van het algoritme dat overweg kan met audio waarbij de toonhoogte verhoogd of verlaagd is, of waarbij de audio sneller of trager is afgespeeld.

Buiten het algoritme bevat de bibliotheek ook verschillende toepassingen die hiervan gebruik maken. Zo is het mogelijk om de fingerprints van een geluidsfragment te bekijken, matches tussen verschillende geluidsfragmenten te visualiseren, en grafisch te experimenteren met de verschillende parameters. Figuur 3.1 toont een screenshot van deze toepassing.

Figuur 3.1: De gebruikersinterface van Panako's NFFT Fingerprint Visualiser. Onderaan kunnen de parameters van het algoritme met behulp van sliders gewijzigd worden.



Er is ook een applicatie beschikbaar om verschillende geluidsfragmenten te synchroniseren.

Deze applicatie gebruikt naast het accoustic fingerprinting algoritme ook het kruiscovariantie algoritme.

Na het bepalen van de latency tussen de verschillende audiofragmenten kan de applicatie een shell script genereren dat met behulp van *FFmpeg* stukjes van de geluidsbestanden wegknipt of er stilte aan toevoegt. Het resultaat is dat na het uitvoeren van het script de geluidsbestanden gesynchroniseerd zijn.

3.1.5 FFmpeg

FFmpeg is een command-line multimedia framework dat gebruikt wordt voor encoderen, decoderen, multiplexen, demultiplexen, streamen en afspelen van audio en video. [14]

In dit onderzoek wordt FFmpeg voornamelijk gebruikt in scripts bij het geautomatiseerd genereren van testdata.

3.1.6 SoX

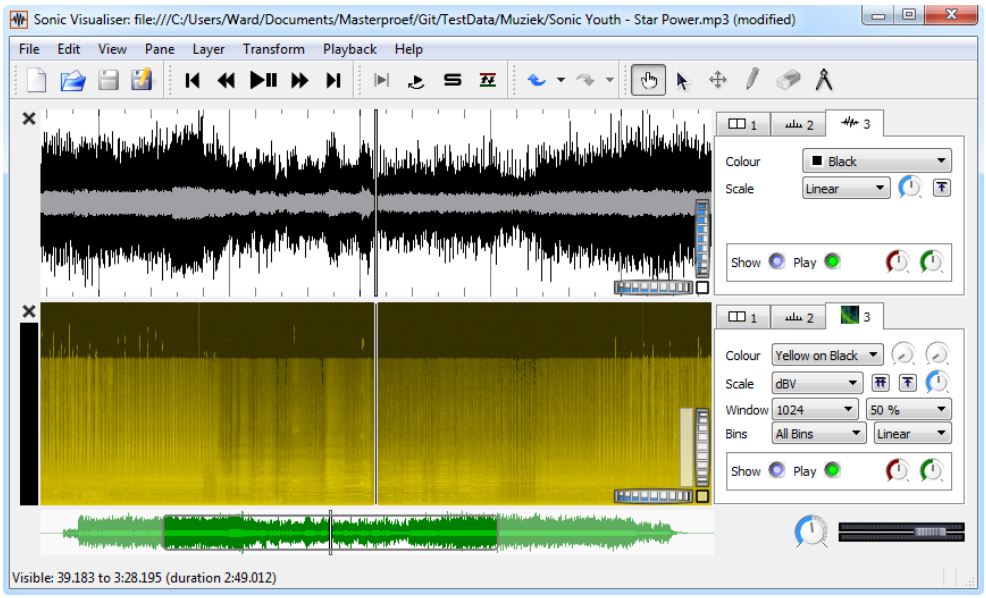
SoX is net zoals FFmeg een command-line tool voor audioverwerking. Buiten de mogelijkheid om audiobestanden te converteren laat SoX ook minder triviale operaties toe. Zo is het onder meer mogelijk om het volume aan te passen, effecten toe te voegen, de bestanden bij te knippen of gegenereerde geluiden in een audiobestand te mixen. [7]

In dit onderzoek wordt SoX gebruikt in scripts bij het manipuleren van de testdata.

3.1.7 Sonic Visualiser

Sonic Visualiser is een gebruiksvriendelijke desktopapplicatie voor de analyse, visualisatie van audiobestanden. Sonic Visualiser laat toe om audiobestanden vanuit verschillende perspectieven te analyseren, zo kan zowel de waveform als het spectrogram van een audiobestand gevisualiseerd worden. Sonic Visualiser is uitbreidbaar met plug-ins in het Vamp formaat. [9]

Figuur 3.2: De gebruikersinterface van Sonic Visualiser



Sonic Visualiser werd in dit onderzoek vooral gebruikt om handmatig de latency tussen verschillende audiofragmenten te bepalen. Ook heeft de applicatie dienst gedaan als educatieve tool om verschillende audioverwerkingsalgoritmen visueel voor te stellen.

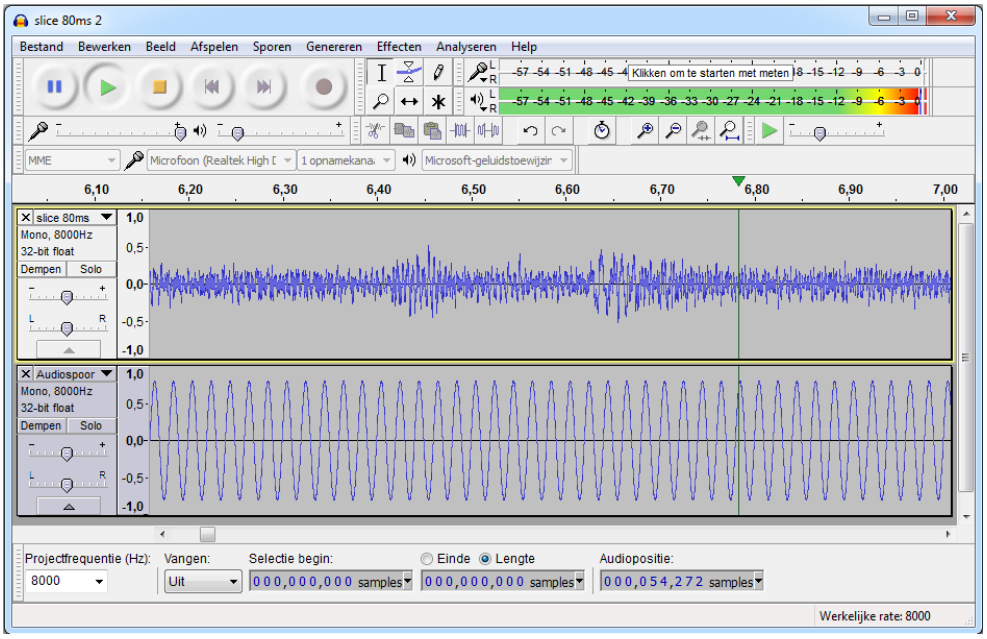
3.1.8 Audacity

Audacity is een open-source desktopapplicatie voor het bewerken, opnemen en converteren van audio. Met Audacity is het ook mogelijk om tal van effecten en filters aan audio toe te voegen.[1]

Figuur 3.3 toont de gebruikersinterface van dit programma.

Alle opnames en handmatige bewerkingen op audiobestanden in dit onderzoek zijn uitgevoerd met behulp van Audacity.

Figuur 3.3: De gebruikersinterface van Audacity



3.1.9 Max/MSP

Max/MSP is een visuele programmeertaal voor muziek en multimedia. Het is een systeem waarbij modules met elkaar verbonden kunnen worden om zo complexere systemen op te bouwen. Max/MSP beschikt ook over een API waarmee in Java of C nieuwe modules ontwikkeld kunnen worden. [2]

Met Max/MSP is het mogelijk om realtime audio te verwerken, daarom zal deze toepassing gebruikt worden voor het ontwikkelen van de gebruikersinterface.

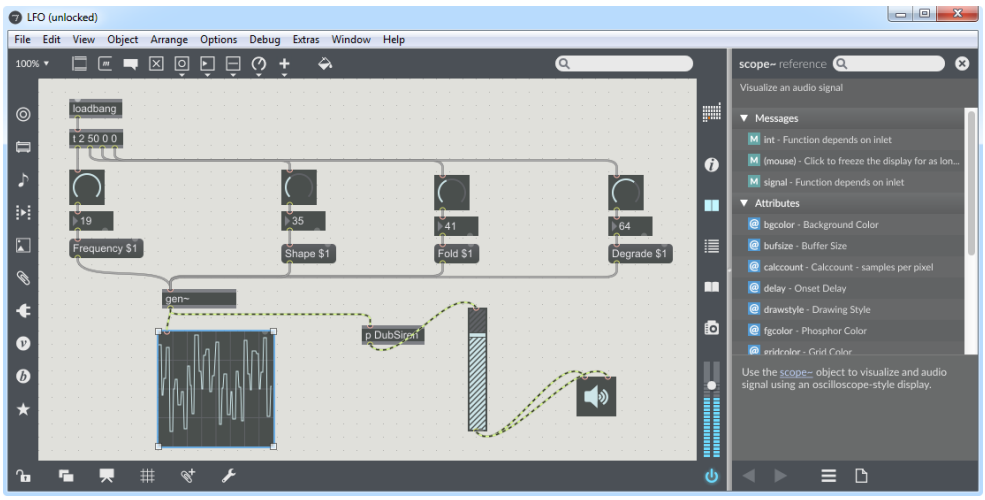
Figuur 3.4 toont hoe een eenvoudige Max/MSP patch er grafisch uitziet.

3.1.10 Teensy

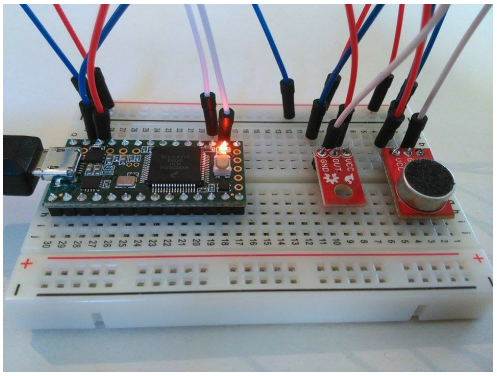
De Teensy is een kleine microcontroller die via USB geprogrammeerd kan worden. De Teensy is compatibel met de Arduino software en is hierdoor zeer gebruiksvriendelijk. [4]

De sensoren die gebruikt worden bij de experimenten van het IPEM zijn meestal aangesloten op Teensy microcontrollers.

Figuur 3.4: De gebruikersinterface van Max/MSP: een *patch panel* met daarop enkele modules die samen een complexere toepassing vormen.



Figuur 3.5: De Teensy microcontroller verbonden met een infraroodsensor en microfoon op een breadboard.



In hoofdstuk 4 worden de testen die met de Teensy microcontroller zijn uitgevoerd meer in detail besproken. Figuur 3.5 toont een typische testopstelling waarbij twee sensoren zijn aangesloten op de microcontroller.

3.2 Accoustic fingerprinting

De Panako softwarebibliotheek bevat een implementatie van het accoustic fingerprinting algoritme. Om wijzigingen mogelijk te maken hebben is de code van het algoritme overgeno-

men in het project van dit onderzoek. De aangepaste code blijft wel nog steeds afhankelijk van enkele klassen uit het Panako project.

3.2.1 Optimalisaties

Aan dit algoritme is één vereenvoudiging aangebracht. Het originele algoritme bevatte namelijk de mogelijkheid om alle offsets boven een bepaalde drempelwaarde te verwerken. Deze feature laat toe dat er meerdere matches kunnen gevonden worden binnen één uitvoering van het algoritme. Om dit te ondersteunen moeten alle matches echter wel één voor één worden vergeleken met de drempelwaarde. Omdat deze toepassing enkel de beste offsetwaarde nodig heeft is dit overbodig. De beste offset en bijhorende fingerprints wordt apart bijgehouden. De naverwerking wordt hierdoor vermeden.

3.2.2 Parameters en hun invloed op het algoritme

De werking van dit algoritme is afhankelijk van een aantal parameters die een grote invloed kunnen hebben op de performantie en de nauwkeurigheid van het uiteindelijke resultaat. Daarom is het van belang om voor het uitvoeren van het algoritme de waarde van deze parameters te controleren. De optimale waarde van elke parameter is afhankelijk van verschillende factoren die van situatie tot situatie kunnen verschillen:

- de vereiste nauwkeurigheid van het algoritme;
- de vereiste performantie van het algoritme;
- de mate waarin er omgevingsgeluid aanwezig is;
- de opnamekwaliteit van het omgevingsgeluid.

De meeste parameters worden bijgehouden in een configuratiebestand waardoor ze ook na compilatie wijzigbaar zijn. Dit zijn de belangrijkste parameters uit het configuratiebestand die invloed hebben op het algoritme:

`SAMPLE_RATE`

Deze parameter bepaalt de standaard samplefrequentie die gebruikt wordt tijdens

het synchronisatieproces. Het verhogen van deze parameter zorgt voor een tragere verwerking maar een betere nauwkeurigheid. Afhankelijk van op welke manier de synchronisatie wordt opgestart (via Max of met een `AudioDispatcher`) worden de binnenkomende streams geresamplet of wordt al een correcte samplefrequentie verondersteld.

`NFFT_BUFFER_SIZE`[1]

Dit is de grootte van de verschuivende buffer die gebruikt wordt in het FFT algoritme. Deze parameter is cruciaal aangezien de frequentiesterktes op een bepaalde plaats op de tijdas worden berekend per buffer. Deze parameter wordt uitgedrukt in aantal samples.

`NFFT_STEP_SIZE`

Dit is het aantal samples elke verschuiving in het FFT algoritme. De stepsize beïnvloedt rechtstreeks de nauwkeurigheid van het accoustic fingerprinting algoritme. Wanneer deze parameter is ingesteld op 128 samples en een samplefrequentie van 8000hz dan is de maximale nauwkeurigheid $128/8000Hz = 0.016s = 16ms$.

`MIN_ALIGNED_MATCHES`

Een match tussen twee audiofragmenten wordt pas als geldig beschouwd wanneer er een bepaald aantal fingerprint matches met dezelfde offset gevonden zijn. Dit aantal wordt ingesteld met deze parameter.

`NFFT_MAX_FINGERPRINTS_PER_EVENT_POINT`

Deze parameter bepaalt het maximum aantal fingerprints waaraan een event point (een punt op het spectrogram) kan deelnemen. Hoe hoger dit maximum hoe vlugger er matches kunnen gevonden worden. Bij een hoge waarde moeten meer berekeningen worden uitgevoerd, dit heeft invloed op de performantie.

`NFFT_EVENT_POINT_MIN_DISTANCE`

Dit is de minimale afstand tussen twee event points op het spectrogram die samen een fingerprint kunnen vormen.

[1]De letter N in NFFT heeft geen noemenswaardige betekenis. De naam is overgenomen van de gelijknamige parameter uit de Panako bibliotheek.

Verder maakt het algoritme nog gebruik van twee hardgecodeerde parameters die niet instelbaar zijn via het configuratiebestand: `MIN_FREQUENCY` en `MAX_FREQUENCY`. Deze parameters bepalen binnen welke frequentiebereik er naar fingerprints gezocht worden. De waarden waarop deze ingesteld staan bevinden zich op de rand van de frequenties die door muziek of stemgeluid geproduceerd worden.

3.2.3 Optimale instellingen

Het bepalen van de optimale waarden voor de parameters is geen exacte wetenschap maar eerder een probleem dat proefondervindelijk moet worden aangepakt.

Bij het accoustic fingerprinting algoritme is er een groot verschil tussen de meest "elegante" parameterwaarden en de in de praktisch presterende waarden. Dit verschil zal duidelijk worden in volgende opsomming waarin elke parameter zal worden besproken.

`SAMPLE_RATE`

Bij deze parameter is het van belang om een goede balans te vinden zodat de geluidskwaliteit aanvaardbaar blijft zonder een hypotheek te plaatsen op de performantie van het algoritme. De praktijk heeft uitgewezen dat bij een samplefrequentie van $8000Hz$ het algoritme goed presteert. Deze waarde wordt bevestigd in artikel [21].

`NFFT_BUFFER_SIZE` en `NFFT_STEP_SIZE`

De ingesteld waarden van de samplefrequentie, buffergrootte en stapgrootte zijn afhankelijk van elkaar. Om een goede werking van het FFT algoritme te garanderen bij een samplefrequentie van $8000Hz$ worden de buffergrootte en stapgrootte respectievelijk ingesteld op 512 en 128 (of 256) samples. Deze waarden worden eveneens vermeld in artikel [21].

`MIN_ALIGNED_MATCHES`

In een toepassing zoals het detecteren van liedjes ten opzichte van een database is het secuur instellen van deze parameter erg belangrijk. Deze parameter heeft namelijk een grote invloed op het voorkomen van *false positives* of *false negatives*.

Bij het synchroniseren van streams is de situatie echter helemaal anders. Het binnen-

krijgen van een false positive (=foute latency) is veel minder erg dan het helemaal niet binnenkrijgen van (mogelijk correcte) resultaten. Aangezien het algoritme per buffer enkel het beste resultaat teruggeeft is de kans dat bij geluidsfragmenten van behoorlijke kwaliteit eenzelfde foute latency meer voorkomt dan de correcte latency zéér klein.

Bij geluidsfragmenten van mindere kwaliteit kan het gebeuren dat er toch foute resultaten door de mazen van het net glippen. Om dit te vermijden is het mogelijk om nog een extra filtering toe te passen. In deze extra stap worden eventuele uitschieters geëlimineerd.

Bovenstaande argumenten stellen duidelijk dat het beter is om deze parameter een lage waarde te geven. In deze toepassing is gekozen voor de waarde 2 in plaats van het absolute minimum 1: hierdoor wordt pure willekeur bij het matchen van audiofragmenten van extreem slechte kwaliteit vermeden. Dit is laag in vergelijking met Panako waar 7 de standaard waarde is van deze parameter.

`NFFT_MAX_FINGERPRINTS_PER_EVENT_POINT`

In Panako is het standaard dat een event point uitmaakt van maximaal 2 fingerprints. Het hanteren van deze waarde leidt ertoe dat het matchen van audiofragmenten van behoorlijke kwaliteit zeer snel kan worden uitgevoerd.

In deze toepassing is het aantal te vergelijken audiofragmenten meestal erg beperkt. Ook is de kwaliteit van deze fragmenten vaak van ondermaats (bv. de opnames op microcontrollers). Daarom is het in dit geval een goed idee om de waarde van deze parameter zéér hoog in te stellen. Hierdoor verhoogt de kans sterk dat er bij zeer slechte audiofragmenten toch enkele overeenkomende fingerprints gevonden worden. Testen hebben uitgewezen dat de negatieve invloed op de performantie beperkt blijft en dat de resultaten sterk verbeteren.

Bij het zoeken naar matches tussen geluidsopnames opgenomen op microcontrollers heeft de praktijk uitgewezen dat het toelaten van maximaal 50 fingerprints per event point degelijke resultaten oplevert.

`NFFT_EVENT_POINT_MIN_DISTANCE`

In tegenstelling tot vorige parameter zorgt het verhogen van deze waarde ervoor dat er minder fingerprints worden gecreëerd. De argumenten die bij vorige parameter zijn aangehaald gelden bijgevolg in omgekeerde zin ook voor deze parameter. Hoewel de Panako standaard 600 is leveren waardes rond het getal 20 in deze toepassing de beste resultaten zonder de performantie sterk te beperken.

3.3 Kruiscovariantie

De Panako bibliotheek bevatte bij aanvang van dit onderzoek al een implementatie van het kruiscovariantie algoritme. In tegenstelling tot het accoustic fingerprinting algoritme was het echter minder grondig afgewerkt. Om degelijke resultaten te garanderen was het noodzakelijk om enkele anomalieën in de geleverde resultaten te analyseren en de oorzaak hiervan op te lossen.

Net zoals het accoustic fingerprinting algoritme is de code overgenomen in het project van dit onderzoek. De code is niet meer afhankelijk van de Panako bibliotheek.

3.3.1 Integratie met accoustic fingerprinting

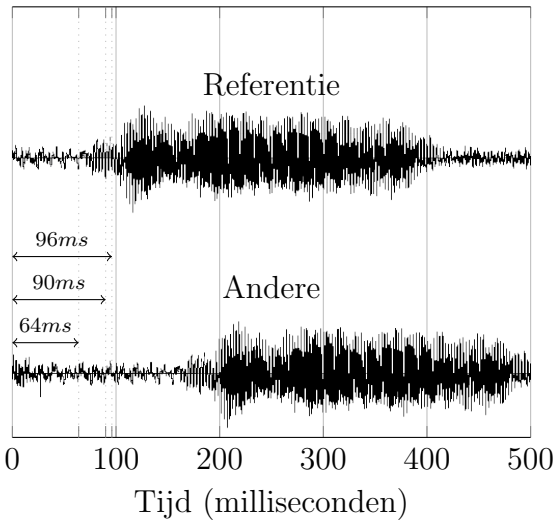
In sectie 2.1.3 is er geschreven dat de latency zeer nauwkeurig kan bepaald worden door het berekenen van de kruiscovariantie. Vanwege de performantie is het wel noodzakelijk om eerst de ruwe latency met accoustic fingerprinting te bepalen. Hoe deze algoritmes geïntegreerd worden zal hier worden besproken.

Onderstel twee geluidsopnames: de tweede opname heeft een vertraging van 90 ms ten opzichte van de eerste referentieopname. Het uitvoeren van het accoustic fingerprinting algoritme met standaard parameters (tot op $32ms$ nauwkeurig) kan twee resultaten opleveren: $64ms$ of $96ms$. Bij het eerste resultaat wordt de werkelijke latency onderschat. De tweede latency overschat deze waarde. Na het berekenen van de kruiscovariantie kan bepaald worden of er zich een onderschatting of overschatting heeft voorgedaan, dit is van

belang om het resultaat correct te verfijnen. Figuur 3.6 toont de mogelijke resultaten van het algoritme.

Als de ruw bepaalde latency positief is wordt deze waarde van het tweede fragment weggeknipt. Na deze stap is de latency van het resterende audiofragment zeker minder dan de minimale nauwkeurigheid van het accoustic fingerprinting algoritme.

Figuur 3.6: Twee audiofragmenten: het tweede audiofragment heeft een latency van 90 milliseconden ten opzichte van het eerste.



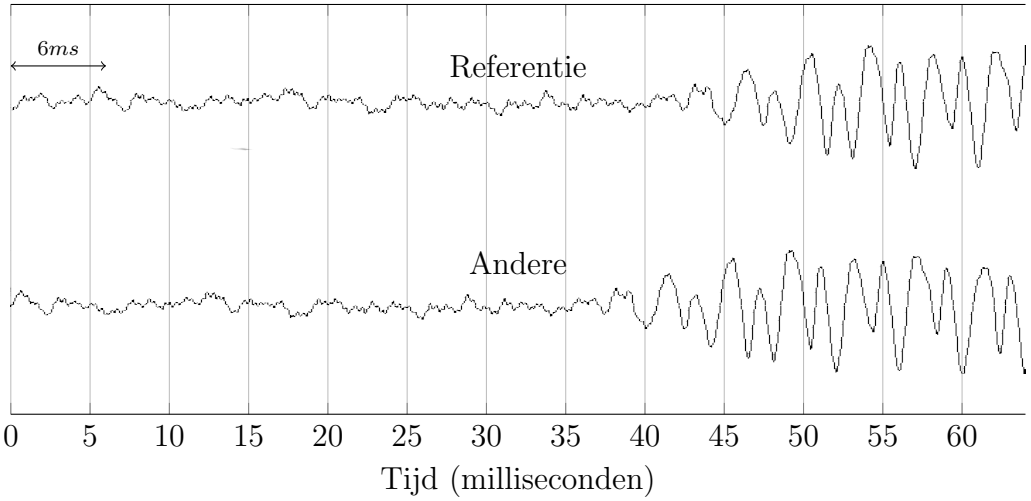
Vervolgens worden een aantal samples van elk audiofragment gekopieerd naar een buffer. De latency tussen deze twee buffers wordt berekend met het kruiscovariantie algoritme (beschreven in sectie 2.1.2). Figuur 3.7 toont de buffers waarop het algoritme kan worden uitgevoerd bij een buffergrootte van 512 samples en samplefrequentie van $8000Hz$.

Wanneer het accoustic fingerprinting algoritme de werkelijke latency heeft onderschat wordt de verfijnde latency bekomen door de ruwe latency en het resultaat van het kruiscovariantie algoritme op te tellen. Bij een overschatting is dit niet het geval.

Aangezien het kruiscovariantie algoritme de latency zoekt van de “andere” buffer ten opzichte van de referentiebuffer zal het resultaat niet $6ms$ maar $58ms$ zijn. Dit komt doordat het algoritme de “andere” buffer cyclisch (maar in de verkeerde zin) verschuift.

Om in een dergelijke situatie de verfijnde latency te berekenen moet men van de som van

Figuur 3.7: Kruiscovariantie buffers na het wegknippen van de $96ms$ latency bepaald door het accoustic fingerprinting algoritme. Hier heeft de referentie audio $6ms$ latency ten opzichte van het andere audiofragment.



de resultaten ($96ms + 58ms$) nog de lengte van de gebruikte buffer ($64ms$) aftrekken. De verfijnde latency is dus $154ms - 64ms = 90ms$.

3.3.2 Optimalisaties

Bugfixes

In de originele versie van het algoritme werd er geen rekening gehouden met het feit dat het accoustic fingerprinting algoritme het de werkelijke latency kan overschatten of onderschatten. De latency bepaalt door het kruiscovariantie algoritme werd telkens opgeteld bij de ruwe latency bepaald door het accoustic fingerprinting algoritme. In 50% van de gevallen leidde dit probleem tot foutieve resultaten.

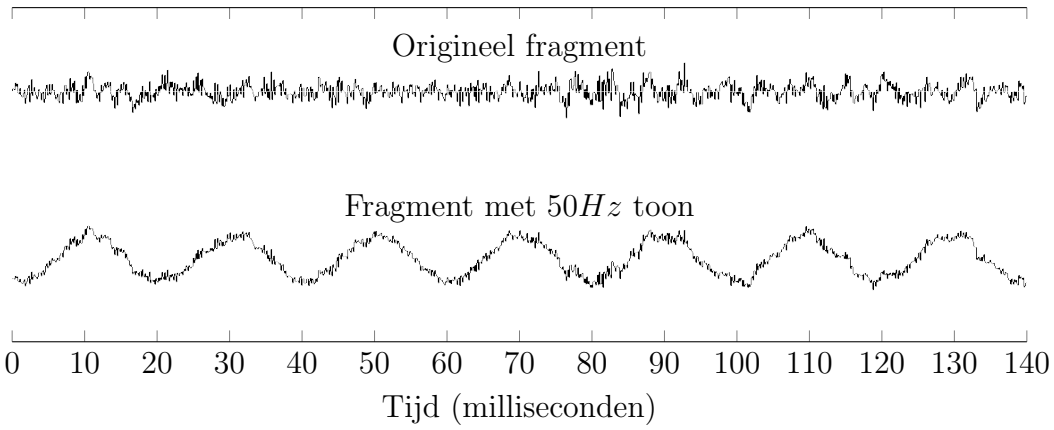
Meerdere malen uitvoeren van het algoritme

In tegenstelling tot accoustic fingerprinting is het bepalen van de latency met kruiscovariantie veel meer foutgevoelig. Bij opnames van slechte kwaliteit worden vaak foute resultaten

teruggegeven.

Hoewel bij accoustic fingerprinting mogelijk storende factoren (een zoemende toon, geruis,...) zichtbaar zijn in het spectrogram zullen er naar alle waarschijnlijkheid nog steeds voldoende fingerprints gegenereerd worden op basis van geluiden die wel in beide opnames voorkomen.

Figuur 3.8: Twee gelijke audiofragmenten. Aan het tweede audiofragment is een achtergrondtoon van $50Hz$ toegevoegd.



Dergelijke storende elementen hebben veel meer invloed op het kruiscovariantie algoritme. Een ongewenste zoemende bastoon zorgt voor een grote verandering van de geluidsgolf van het audiofragment. Aangezien de kruiscovariantie rechtstreeks tussen twee geluidsgolven berekent wordt kan dit serieuze gevolgen hebben. In figuur 3.8 wordt dit probleem visueel verduidelijkt.

De invloed van storende factoren zoals zoemende tonen en ruis is gemakkelijk te beperken. Aangezien het kruiscovariantie algoritme over een zéér klein deeltje van het audiofragment wordt uitgevoerd is het zeer eenvoudig om het algoritme tientallen keren, maar telkens op een andere plaats, uit te voeren. Na elke iteratie wordt de latency in het geheugen opgeslagen. Ten slotte wordt er gezocht naar de latency die het meeste voorkomt. Die latency wordt gebruikt om het resultaat van het accoustic fingerprinting algoritme te verfijnen.

3.3.3 Parameters en hun invloed op het algoritme

De parameters van het kruiscovariantie algoritme zijn net zoals die van het accoustic fingerprinting algoritme instelbaar via het configuratiebestand.

`NFFT_BUFFER_SIZE`

Hoewel het FFT algoritme eigenlijk niets te maken heeft met het berekenen van de kruiscovariantie speelt deze parameter bij dit algoritme toch een belangrijk rol. Deze parameter bepaalt namelijk de grootte van de buffers waartussen de kruiscovariantie berekent wordt. Om een goede werking van het kruiscovariantie algoritme te verzekeren is het een vereiste dat de gebruikte buffers groter zijn dan de minimale nauwkeurigheid van het accoustic fingerprinting algoritme (anders is het mogelijk dat na het knippen de audiofragmenten er geen gelijkenissen te vinden zijn tussen beide buffers). Aangezien deze nauwkeurigheid bepaald wordt de `NFFT_STEP_SIZE` parameter en de `NFFT_BUFFER_SIZE` hier altijd een veelvoud van is, is het gemakkelijk om deze parameter ook voor het kruiscovariantie algoritme te gebruiken.

`CROSS_COVARIANCE_NUMBER_OF_TESTS`

Deze parameter bepaalt het aantal keren dat het kruiscovariantie algoritme per streambuffer (zie 2.2) zal worden uitgevoerd. De meest voorkomende latency wordt als resultaat teruggegeven.

`CROSS_COVARIANCE_THRESHOLD`

Deze parameter bepaalt het minimum aantal keer dat een latency moet voorkomen om als geldig beschouwd te mogen worden. Stel dat het algoritme 50 keer wordt uitgevoerd maar elke keer een verschillend resultaat teruggeeft, dan zal er geen latency worden teruggegeven indien deze parameter staat ingesteld op een waarde groter dan 1.

3.3.4 Optimale instellingen

`NFFT_BUFFER_SIZE`

De optimale waarde van deze parameter werd in sectie 3.2.3 besproken.

`CROSS_COVARIANCE_NUMBER_OF_TESTS`

Bij audiofragmenten van goede kwaliteit is het niet noodzakelijk om deze parameter een hoge waarde te geven. Als het algoritme 5 à 10 keer wordt uitgevoerd zal de correcte latency hoogst waarschijnlijk al verschillende malen gevonden zijn.

Bij audiofragmenten van slechte kwaliteit is het van belang om deze parameter zo hoog mogelijk in te stellen. Aangezien het kruiscovariantie algoritme vrij intensief is hangt de waarde wat af van de beschikbare rekenkracht. De praktijk heeft uitgewezen dat de waarde 50 een goed evenwicht biedt tussen correctheid en performantie.

Dit aantal heeft als bovengrens (N_B) het aantal keren dat een kruiscovariantie buffer (bepaald door `NFFT_BUFFER_SIZE`) past in een streambuffer (zie 2.2) waarin de streams worden ingelezen (bepaald door `SLICE_SIZE_S`). Dit kan worden berekend met volgende formule:

$$N_B = \frac{\texttt{SLICE\_SIZE\_S}}{\texttt{NFFT\_BUFFER\_SIZE} \cdot \texttt{SAMPLE\_RATE}} \tag{3.1}$$

`CROSS_COVARIANCE_THRESHOLD`

Het hoog instellen van deze waarde heeft enkel nut als het van belang is dat het kruiscovariantie algoritme zeker een correct resultaat teruggeeft. In de huidige implementatie wordt het ruwe resultaat gebruikt als het aantal gelijke waarden niet boven deze drempel komt. Daarom wordt deze waarde meestal op 1 ingesteld. De best mogelijke verfijning van het resultaat wordt dan toegepast, ongeacht het aantal keer dat het resultaat voorkomt.

3.4 Filteren van de resultaten

Het is moeilijk om met zekerheid te bepalen of een bepaalde latency correct is. Toch kan er een soort van statistische optimalisatie worden uitgevoerd op de opeenvolgende latencies. De kans is namelijk klein dat de werkelijke latency verandert en na een korte tijd terugkeert

naar de originele waarde.[2] Ook is de kans klein dat een foute latency toevallig verschillende malen na elkaar gedetecteerd wordt.

Deze eigenschappen laten toe om toch een bepaalde vorm van "foutherstelling" te implementeren. Fouten kunnen namelijk weggefilterd worden door eventuele pieken in opeenvolgende latencies af te vlakken.

3.4.1 Werking

Het afvlakken van pieken kan in realtime geïmplementeerd worden door opnieuw gebruik te maken van verschillende *sliding windows*. Per audiostream wordt een buffer bijgehouden met daarin de opeenvolgende latencies ten opzichte van de referentie audiostream. Wanneer een buffer haar maximumcapaciteit heeft bereikt gaat het toevoegen van een nieuwe latency gepaard met het verwijderen van de oudste latency.

Het afvlakken van pieken wordt verwezenlijkt door in plaats van de meest recente latency het gemiddelde of de mediaan van de buffer te gebruiken. De mate waarin pieken worden afgevlakt hangt af van de grootte van de buffer en de gebruikte methode (gemiddelde of mediaan).

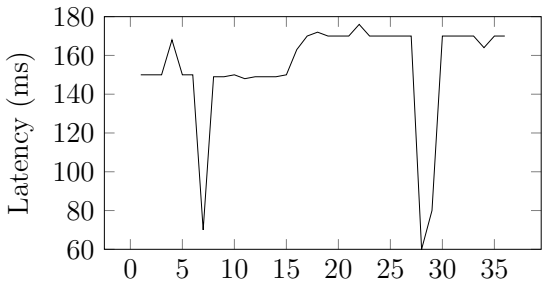
3.4.2 Voorbeelden

De effectiviteit van de verschillende parameters (buffergrootte en filtermethode) zal in dit gedeelte worden weergegeven aan de hand van enkele voorbeelden. Er zullen verschillende soorten filters worden toegepast op een verloop van 35 latencies. In het ongefilterde verloop (zie figuur 3.9) komen twee grote en drie kleine (waarschijnlijk foutieve) pieken voor. Na

[2]Theoretisch is een dergelijke situatie toch mogelijk: In dit voorbeeld wordt de latency bepaalt tussen de audiostreams ($8000Hz$ samplefrequentie) van twee Teensy microcontrollers. Wanneer er 50 samples gedropt worden van de referentie audiostream zorgt dit voor een latencyverhoging van de andere audiostream van $6ms$. Indien er enkele seconden later 50 samples gedropt worden van de andere audiostream dan leidt dit tot een vermindering van de latency van $6ms$. Visueel is dit een piek in de opeenvolgende latencies.

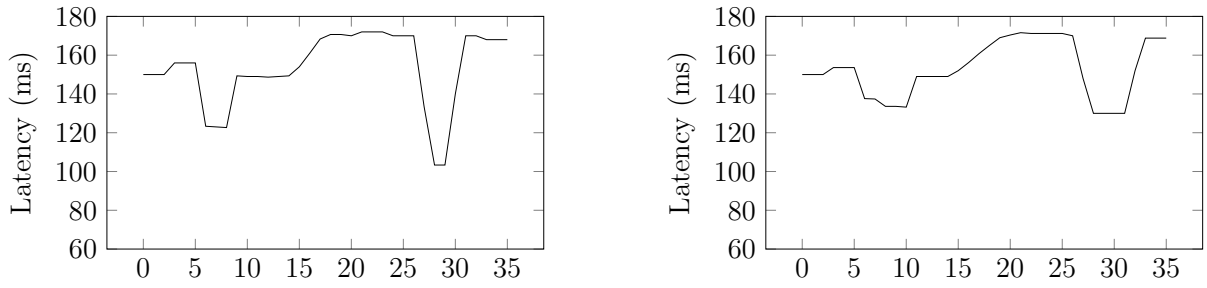
het vijftiende resultaat doet er zich een blijvende (correcte) verhoging van de latency voor.

Figuur 3.9: Grafische weergave van het ongefilterde verloop van de latency.



Moving average filter

De eerste twee voorbeelden tonen het verschil tussen de ongefilterde latencies en de latencies die gefilterd zijn door het gemiddelde van een buffer te nemen. Bij het eerste voorbeeld bevat de buffer maximum 3 latencies, bij het tweede voorbeeld kan de buffer maximum 5 latencies bevatten.



(a) Buffergrootte 3

(b) Buffergrootte 5

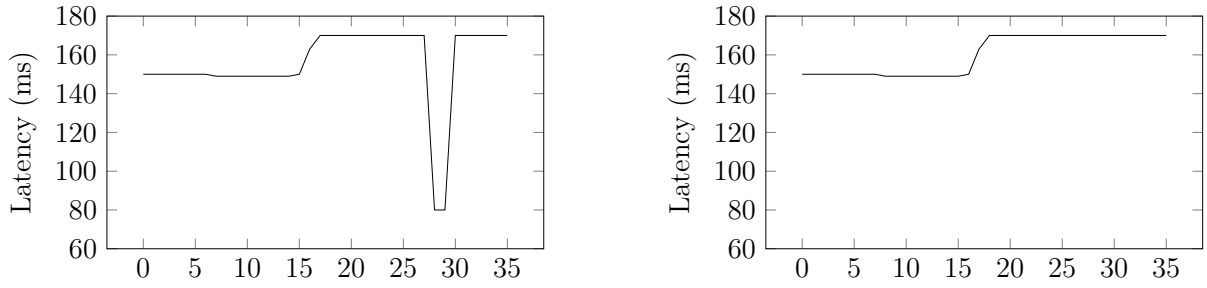
Figuur 3.10: Het verloop van de latencies na het toepassen van een moving average filter.

Ondanks het toepassen van de filter blijven de grootste pieken nog steeds zichtbaar. De oorzaak hiervan is dat deze pieken door hun grootte toch sterk doorwegen op het gemiddelde. Om er voor te zorgen dat grote pieken nog meer worden weggefilterd zou de maximumgrootte moeten worden verhoogd. In 3.4.4 zal duidelijk worden waarom dit toch geen goed idee is.

In de gefilterde resultaten valt ook op dat de hellingshoek van de latencyverhoging vermindert naarmate een grotere buffer gehanteerd wordt. Dit is geen goede eigenschap: het duurt namelijk veel langer tot er na een wijziging terug een stabiele latency bereikt wordt.

Moving median filter

In de volgende voorbeelden zal de mediaan van elke buffer berekend worden. Om de resultaten goed met elkaar te kunnen vergelijken worden dezelfde buffergroottes gehanteerd.



(a) Buffergrootte 3 **(b)** Buffergrootte 5

Figuur 3.11: Het verloop van de latencies na het toepassen van een moving median filter.

De grafieken tonen een duidelijk verschil tussen de moving median filter en de moving average filter. De pieken veroorzaakt door 1 (eventueel) foute latency worden bij de moving median filter onmiddellijk weggefilterd. In de grafiek is er geen enkel spoor meer van te vinden. De piek veroorzaakt door 2 (eventueel) foute latencies wordt enkel volledig weggefilterd wanneer buffergrootte 5 gehanteerd wordt.

De wijziging van de latency wordt niet afgevlakt zoals bij de moving average filter. De hellingsgraad blijft onaangetast.

3.4.3 Parameters

`LATENCY_FILTER_TYPE`

Deze parameter bepaalt welke soort filter zal worden toegepast op de binnenkomende

latencies. Mogelijke waarden zijn: `average`, `median` en `none`.

`LATENCY_FILTER_BUFFER_SIZE`

Dit is de grootte van de buffer waarin de meest recente latencies zullen worden opgeslagen. Het aangeraden om als waarde een oneven getal te kiezen.

3.4.4 Gevolgen

Het filteren van de latency heeft een negatief effect op de snelheid waarmee wijzigingen van de latency gedetecteerd kunnen worden. Zowel de grootte van de latency filter buffer als de grootte van de streambuffers (zie sectie 2.2) hebben invloed op de snelheid waarmee nieuwe latencies gedetecteerd zullen worden. De grootte van de streambuffers (parameter `SLICE_SIZE_S`) bepaalt namelijk met welk interval nieuwe latencies binnenkomen.

Een verhoging van de latency zal bij de moving average filter een onmiddellijke maar beperkte invloed hebben op het gefilterde resultaat. De tijd die nodig is om tot een stabiel resultaat te komen kan berekent worden met volgende formule:

$$(\texttt{SLICE\_SIZE\_S}) \cdot (\texttt{LATENCY\_FILTER\_BUFFER\_SIZE}) \tag{3.2}$$

Bij de moving median filter wordt een wijziging van de latency gedetecteerd wanneer meer dan de helft van de buffer de gewijzigde latency bevat. De tijd tot wanneer een detectie plaatsvindt kan met volgende formule berekent worden:

$$\frac{(\texttt{SLICE\_SIZE\_S}) \cdot (\texttt{LATENCY\_FILTER\_BUFFER\_SIZE})}{2} \tag{3.3}$$

3.5 Ontwerp van de softwarebibliotheek

De Java bibliotheek voor het synchroniseren van streams bestaat uit verschillende onderdelen onderverdeeld in `packages`. Na het bespreken van de packages zal het ontwerp van de applicatie gedetailleerd worden toegelicht.

`be.signalsync.stream`

Deze package bevat klassen die verantwoordelijk zijn voor het abstract voorstellen en verwerken van streams. Streams kunnen namelijk via verschillende kanalen worden ingelezen. Met behulp van deze klassen wordt een abstractere verwerking mogelijk.

`be.signalsync.slicer`

De klassen uit deze package zorgen voor het bufferen van de binnenkomende streams (zie sectie 2.2) zodat het bepalen van de latency in realtime mogelijk wordt.

`be.signalsync.syncstrategy`

Deze package bevat de implementatie van de synchronisatiealgoritmen. Deze algoritmen worden aangeroepen van uit het `be.signalsync.sync` package. Welk algoritme precies gebruikt wordt kan worden ingesteld via het configuratiebestand.

`be.signalsync.datafilters`

Deze package bevat de implementaties van de verschillende soorten filters besproken in sectie 3.4.

`be.signalsync.sync`

Deze package roept klassen uit het `be.signalsync.slicer` aan om binnenkomende streams op te splitsen in buffers (alias slices). Vervolgens worden de algoritmes uit `be.signalsync.syncstrategy` gebruikt om de latency tussen de opeenvolgende buffers te bepalen. Met behulp van het package `be.signalsync.datafilters` worden eventuele pieken uit de resulterende latencies weggefilterd.

`be.signalsync.msp`

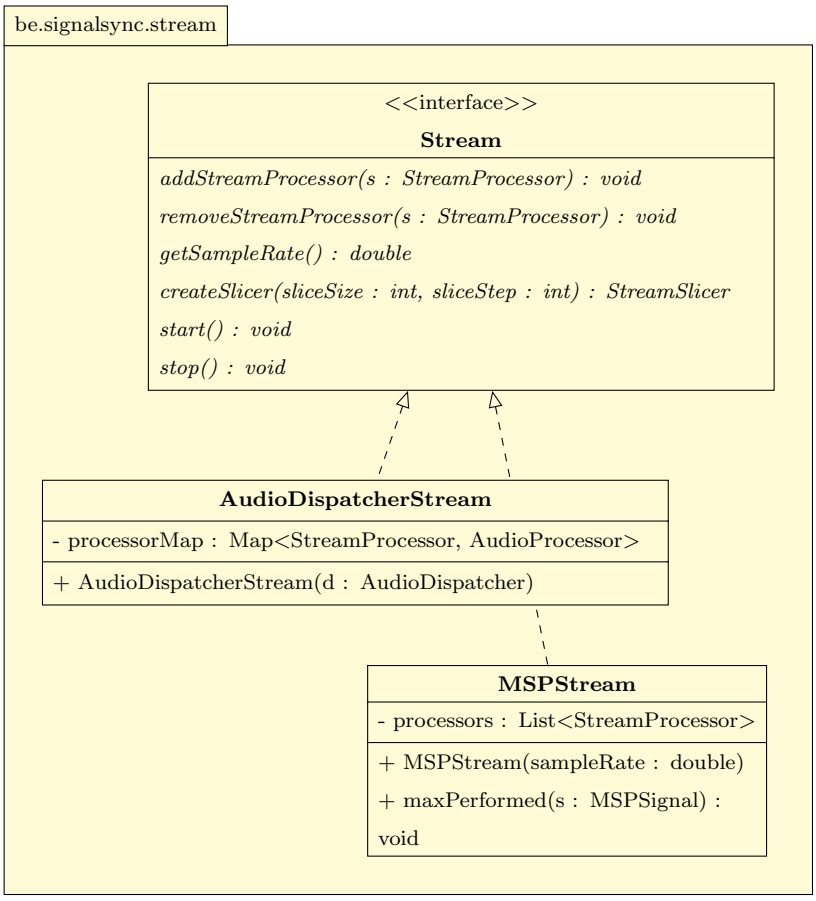
Deze package bevat de implementatie van enkele Max/MSP modules. Met behulp van deze modules is het mogelijk om de streams van microcontrollers in te lezen en de data ervan te synchroniseren.

3.5.1 Streams

Streams kunnen van verschillende bronnen afkomstig zijn: van een microfoon, van een bestand of van een virtuele patchkabel uit Max/MSP,... Streams afkomstig van een microfoon

of bestand kunnen met behulp van de klasse `AudioDispatcher` worden ingelezen (besproken in sectie 3.1.3). Deze klasse laat op eenvoudige wijze verdere verwerking toe. Het inlezen van data uit Max/MSP gebeurt op een fundamenteel andere manier die niet door de klasse `AudioDispatcher` ondersteund wordt. Daarom wordt er nog een extra abstractielaag voorzien waarmee het mogelijk wordt om zowel `AudioDispatchers` als Max/MSP streams op dezelfde manier aan te spreken en te verwerken. Figuur 3.12 toont de `Stream` interface en de klassen die deze interface implementeren. De methodes van de interface worden in de concrete klassen niet herhaald maar zijn uiteraard wel aanwezig.

Figuur 3.12: UML diagram: de `Stream` interface en haar implementaties.



AudioDispatcherStream

De klasse `AudioDispatcherStream` is een typisch voorbeeld van het *Adapter* ontwerppatroon (meer informatie hierover: [23]). Een bestaande klasse (`AudioDispatcher`) wordt namelijk verbonden met een nieuwe interface (`Stream`) met behulp van een adapter of *wrapper* (`AudioDispatcherStream`). In de adapterklasse wordt er geen echte logica toegevoegd. De enige logica die de klasse bevat heeft als doel het mappen van de methode's van de `Stream` interface naar de klasse `AudioDispatcher`. Het attribuut `processormap` is bijvoorbeeld een `HashMap` die gebruikt wordt voor het mappen van `StreamProcessors` (eigen implementatie) naar AudioProcessors (TarsosDSP implementatie). De werking van `StreamProcessors` wordt verderop in sectie 3.5.1 besproken.

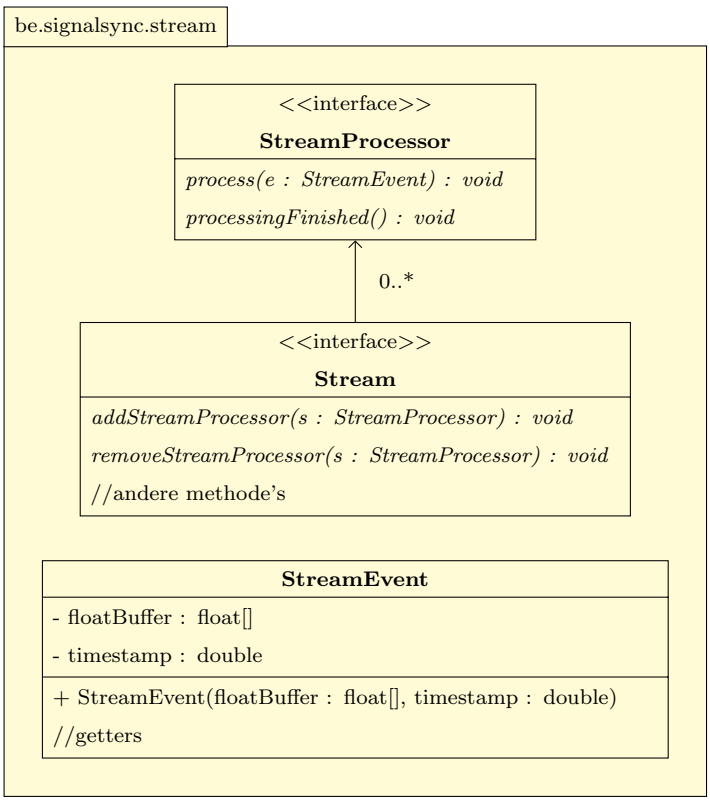
MSPStream

Een `MSPStream` is geen typische implementatie meer van het adapter pattern. Deze klasse voorziet namelijk een extra methode: `maxPerformed` met als parameter een `MSPSignal`. Dit is een belangrijk object in Max/MSP modules geschreven in Java. Een `MSPSignal` bevat een buffer van samples die via een virtuele Max/MSP patchkabel binnenkomen of buitengaan. Elke keer wanneer een Max/MSP module zo'n object binnenkrijgt moet het worden gedelegeerd aan de corresponderende `MSPStream`. Dit delegeren gebeurt met behulp van de `MaxPerformed` methode. Als alle `MSPSignal` objecten correct gedelegeerd worden dan kan het `Stream` object op exact dezelfde manier verwerkt worden als een stream afkomstig van een bestand of microfoon.

Het verwerken van streams

Streams kunnen door één of meerdere klassen worden verwerkt door de methode's van de interface `StreamProcessor` te implementeren. Na het inlezen van een bepaalde hoeveelheid data wordt `process` opgeroepen met als parameter een `StreamEvent` object. Dit object bevat metadata en een buffer van samplewaarden. Figuur 3.13 toont een UML diagram van de klassen die met dit proces te maken hebben.

Figuur 3.13: UML diagram van de klassen en interfaces die een rol spelen bij het verwerken van streams.

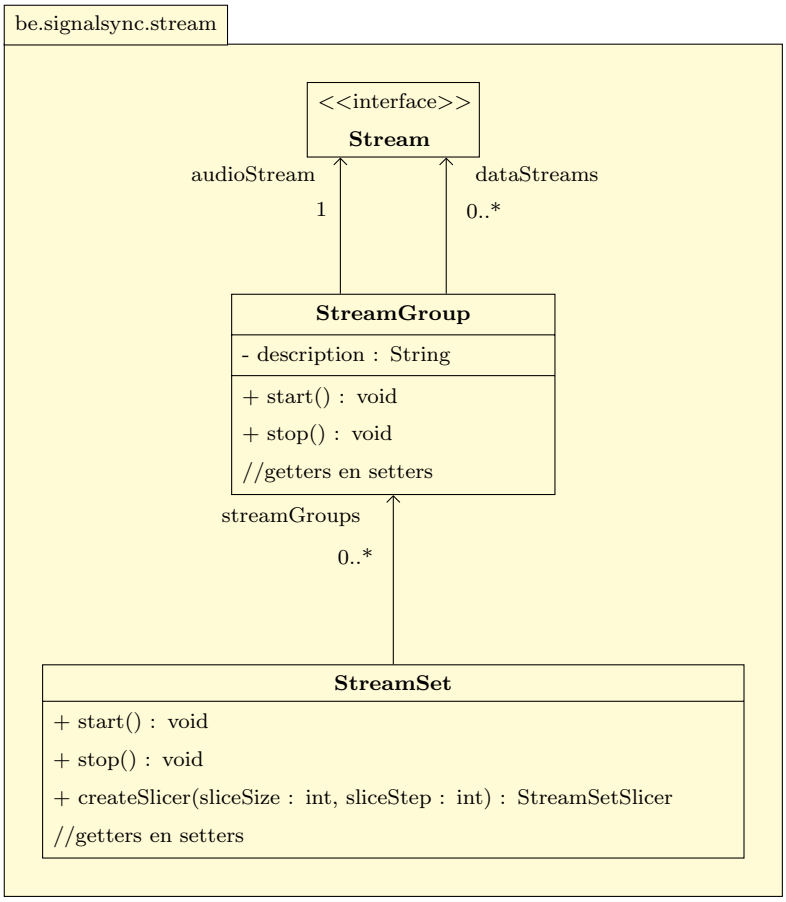


Het bijhouden van streams

In dit onderzoek is er altijd een onderscheid gemaakt tussen datastreams en audiostreams. Datastreams zijn streams afkomstig van sensoren of videocamera's, audiostreams zijn de opnames van het omgevingsgeluid waaraan de datastreams "gekoppeld" zijn. Puur softwarematig wordt er echter geen verschil gemaakt tussen deze verschillende soorten streams. Of een digitaal signaal nu afkomstig is van een microfoon of van een sensor, in beide gevallen is het niet meer dan een opeenvolging van samples.

Om de synchronisatiealgoritmes correct aan te roepen moet er toch een onderscheid gemaakt worden tussen de audiostreams en datastreams. Daarom is de klasse `StreamGroup` in het leven geroepen. Deze klasse bevat één audiostream en nul of meer gekoppelde datastreams. Meestal zijn dit streams die vanuit dezelfde microcontroller zijn ingelezen (zie sectie 1.1).

Figuur 3.14: UML klassendiagram waarop wordt verduidelijkt hoe de verschillende streams worden bijgehouden alvorens ze gesynchroniseerd kunnen worden.



Een verzameling van `StreamGroups` wordt een `StreamSet` genoemd. Een `StreamSet` is een soort van wrapper voor een lijst van `StreamGroups`. Om verschillende `StreamGroups` met elkaar te synchroniseren is het noodzakelijk dat er een `StreamSet` object van wordt aangemaakt. Vervolgens kan het worden doorgegeven aan de `RealtimeSignalSync` klasse, dit is de klasse die verantwoordelijk is voor het bepalen van de latency tussen de verschillende audiostreams.

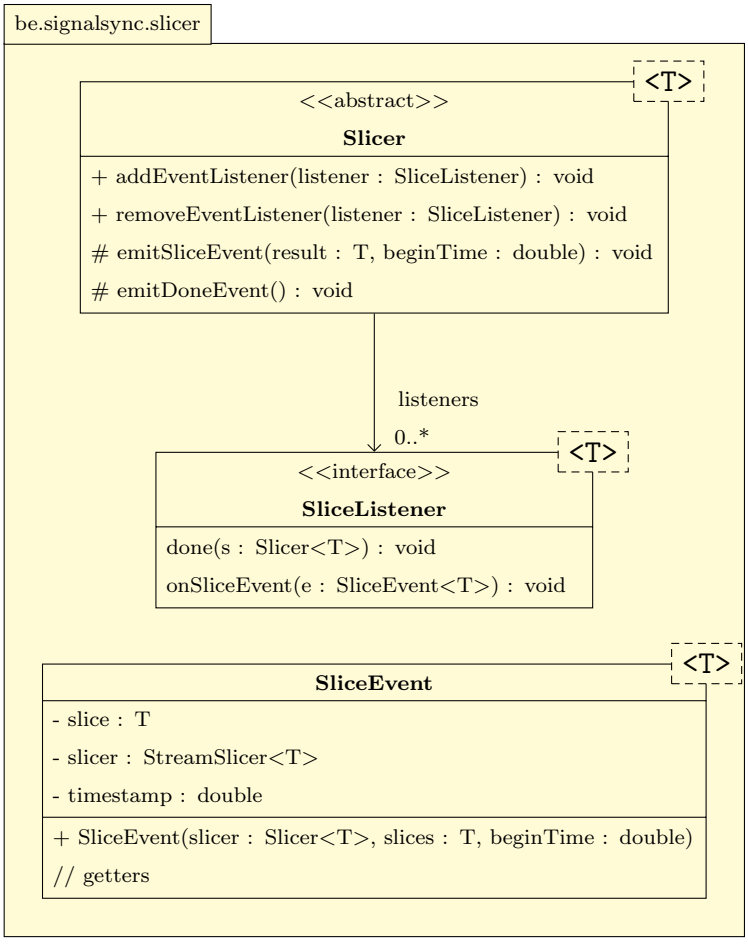
Figuur 3.14 toont een klassendiagram van hoe de streams intern worden bijgehouden.

3.5.2 Bufferen van streams

Alle klassen die te maken hebben met het bufferen van streams (het opsplitsen in zogenaamde *slices*) bevinden zich in het package `be.signalsync.slicer`. Het hoe en waarom van deze verwerking werd al uitgebreid behandeld in sectie 2.2.

De abstracte klasse Slicer

Figuur 3.15: UML diagram de observer logica van de klasse `Slicer`



Een `Slicer` is een abstracte klasse die een stream of verzameling van streams inleest, in stukjes knipt en vervolgens deze stukjes teruggeeft aan elke geïnteresseerd object. Hoe en wat er precies in stukjes geknipt wordt hangt af van het subtype van `Slicer`. De klasse `Slicer` is enkel verantwoordelijk voor het registreren, bijhouden en verwittigen van

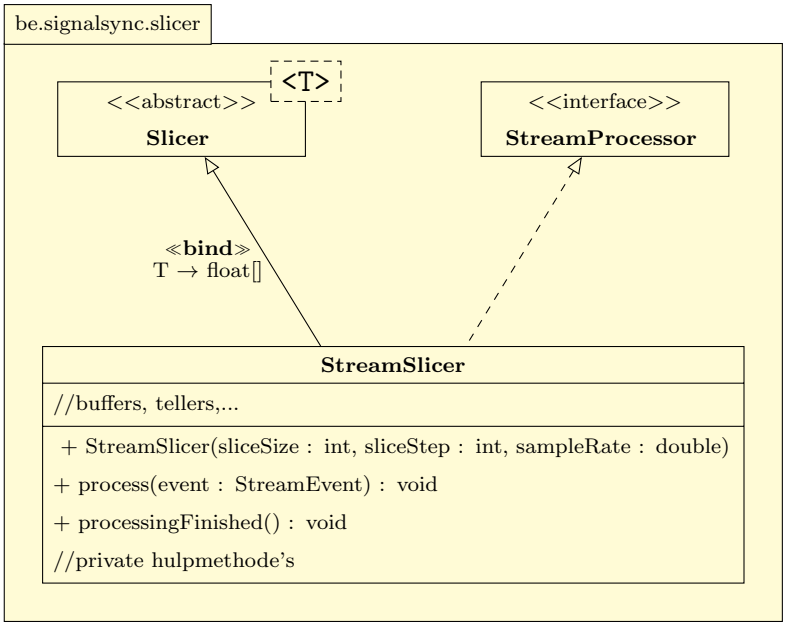
objecten die geïnteresseerd zijn in slices. Dit proces wordt verwezenlijk met behulp van het *observer* ontwerppatroon (uitgebreid besproken in boek [23]).

Figuur 3.15 toont de drie klassen die een rol spelen het observer mechanisme. Elke klasse is generiek en maakt gebruik van type parameter `T`. Dit is het type van hoe een slice wordt voorgesteld. Bij een slice van een stream is dit type bijvoorbeeld een array van `float` waarden.

Zoals in de figuur te zien is bevat `Slicer` een verzameling van geïnteresseerde objecten. De klasse van een geïnteresseerd object moet de interface `SliceListener` implementeren. Wanneer een object geregistreerd is kan het `SliceEvent` objecten ontvangen. Dit object bevat de laatste nieuwe slice, de `Slicer` vanwaar de slice afkomstig is en een timestamp. Wanneer er geen slices meer verzameld kunnen worden kan de methode `done` van elke geïnteresseerd object worden opgeroepen.

StreamSlicer

Figuur 3.16: UML diagram van de klasse `StreamSlicer` en haar supertypes.



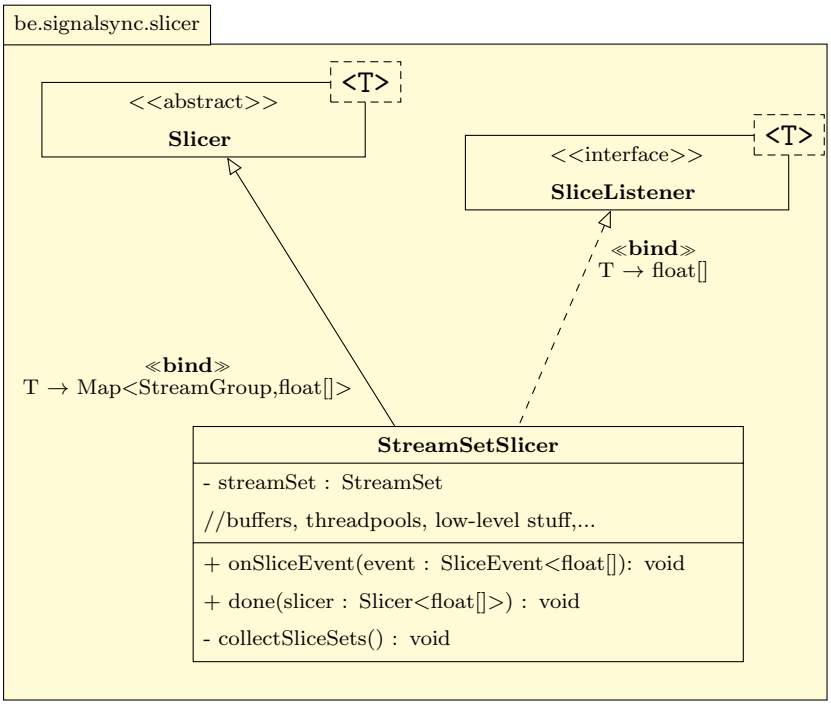
Een `StreamSlicer` is een subklasse van `Slicer` die de data afkomstig van een stream op-

deelt in `float` arrays van een welbepaalde lengte. Om toegang te krijgen tot de data van een `Stream` implementeert de `StreamSlicer` de interface `StreamProcessor`. Via `process` krijgt deze klasse opeenvolgende `float` arrays met samples binnen die worden opgeslagen in buffers. Wanneer er een slice gereed is worden de buffers samengevoegd tot één float array en wordt `emitSliceEvent` van de superklasse `Slicer` opgeroepen. Wanneer `processingFinished` wordt opgeroepen en de stream dus geen data meer beschikbaar heeft, dan wordt `emitDoneEvent` zodat alle `StreamListeners` hiervan op de hoogte zijn.

Figuur 3.16 toont een UML diagram met de besproken klassen en interfaces. Sommige low-level methoden en attributen zijn hierbij weggelaten.

StreamSetSlicer

Figuur 3.17: UML diagram van de klasse `StreamSetSlicer` en haar supertypes.



Een `StreamSetSlicer` is een subtype van de klasse `Slicer` die verantwoordelijk is voor het slicen van alle **audiostreams** van een `StreamSet`. Per `StreamGroup` wordt er dus maar

één stream in stukjes verdeeld. Dit is logisch aangezien de synchronisatiealgoritmen voor het bepalen van de latency enkel worden uitgevoerd op slices van de audiostream.

Bij de creatie van een `StreamSetSlicer` wordt van elke `StreamGroup` uit de `StreamSet` de audiostream opgehaald. Op dit `Stream` object wordt vervolgens de methode `createSlicer` opgeroepen waarop er een `StreamSlicer` wordt teruggegeven. De `StreamSetSlicer` voegt zichzelf als geïnteresseerd object aan de `StreamSlicer` toe. Ook wordt de `StreamSlicer` gekoppeld aan de `StreamGroup` door ze als sleutel en waarde toe te voegen aan een `Map`.

Na afloop van deze initialisatie wordt er gewacht tot wanneer alle streams waarop de `StreamSetSlicer` zich geregistreerd heeft een `SliceEvent` (met een `float` array) verstuurd hebben. Al deze slices worden vervolgens samen met hun corresponderende `StreamGroup` via de `emitSliceEvent` methode verstuurd naar de geïnteresseerde objecten. Dit proces wordt herhaald tot er geen enkele `Stream` nog data ter beschikking heeft.

Figuur 3.17 toont een vrij abstract klassendiagram van de `StreamSetSlicer` en haar supertypen. Low-level attributen zoals de buffers, threadpools en locks zijn hierop weggelaten.

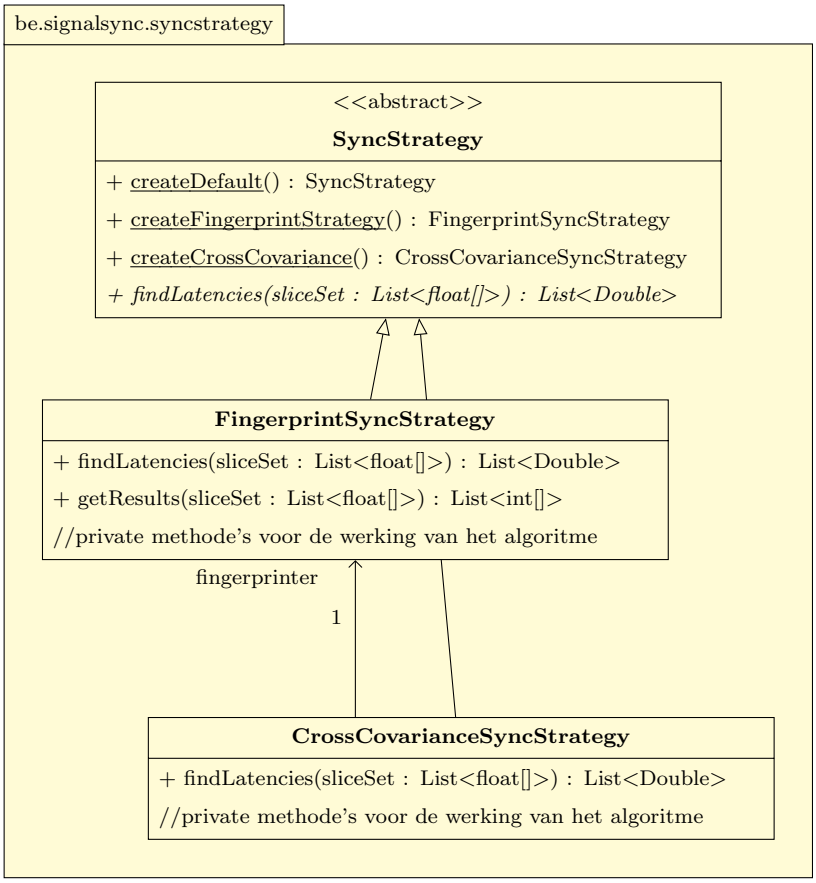
3.5.3 Oproepen van de algoritmen

De synchronisatiealgoritmen bevinden zich in subklassen van `SyncStrategy`. Deze klasse bevat een abstracte methode `findLatencies`. Deze methode ontvangt als parameter een lijst van `float` arrays waarbij elke array de samples bevat van één slice bevat. Na het uitvoeren geeft de methode een lijst terug met daarin de latencies in seconden ten opzichte van de eerste slice uit de lijst.

`SyncStrategy` bevat ook enkele statische methodes waarmee de verschillende mogelijke strategieën gemakkelijk kunnen worden aangemaakt. De parameters die aan de constructoren moeten worden meegegeven worden uit het configuratiebestand gehaald.

Figuur 3.18 toont het UML diagram van deze klasse en haar subtypen.

Figuur 3.18: UML diagram van de klassen met de synchronisatie-algoritmen.



Het fingerprinting algoritme

Het fingerprinting algoritme wordt geïmplementeerd in de klasse `FingerprintSyncStrategy`. Buiten de implementatie van `findLatencies` bevat deze klasse nog een andere publieke methode namelijk `getResults`. Deze methode is vergelijkbaar met `findLatencies` maar geeft de resultaten in een meer low-level formaat terug. In volgend paragraaf zal duidelijk worden waarvoor deze methode gebruikt wordt.

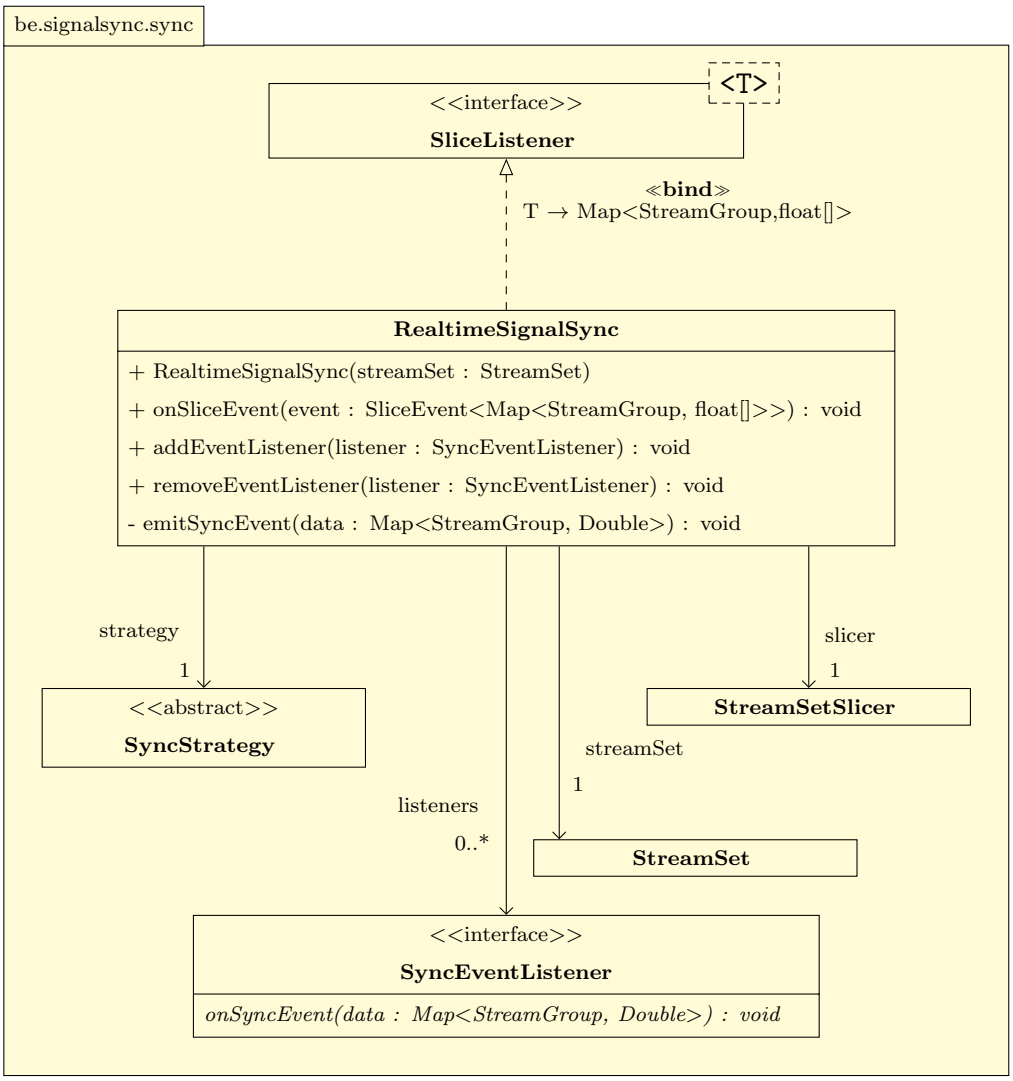
Het kruiscovariantie algoritme

De klasse `CrossCovarianceSyncStrategy` bevat een implementatie van het kruiscovariantie algoritme. De klasse bevat een instantie van `FingerprintSyncStrategy` waarmee

de ruwe latency wordt berekend. Eerst wordt de methode `getResults` opgeroepen, deze methode geeft een lijst terug met per stream de ruwe verschuiving in aantal samples. Het kruiscovariantie algoritme heeft deze informatie nodig om op efficiënte wijze het resultaat te kunnen verfijnen.

3.5.4 Bepalen van de latency

Figuur 3.19: UML diagram van `RealtimeSignalSync` en alle klassen waarvan ze afhankelijk is.



De klasse `RealtimeSignalSync` is verantwoordelijk voor het aanroepen van de klassen uit

verschillende packages waardoor de uiteindelijke synchronisatie kan plaatsvinden.

Een `RealtimeSignalSync` object wordt aangemaakt door aan de constructor een `StreamSet` object mee te geven dat de te synchroniseren streams bevat. Vervolgens wordt er met behulp van de methode `createSlicer` een `StreamSetSlicer` object aangemaakt. Aangezien `SliceListener` geïmplementeerd wordt kan `RealtimeSignalSync` zich registreren als geïnteresseerd object.

Wanneer er voor elke `StreamGroup` een slice beschikbaar is wordt de methode `onSliceEvent` opgeroepen. In deze methode wordt het `SyncStrategy` object gebruikt om de latencies te bepalen. Optioneel kunnen deze resultaten ook nog gefilterd worden (het ontwerp hiervan wordt in 3.5.5 besproken). Ten slotte worden de resultaten naar alle geregistreerde `SyncEventListeners` verstuurd via de private methode `emitSyncEvent`. Geïnteresseerden kunnen zich registreren via de `addEventListener` methode. Deze werkwijze leunt net zoals het *slice event* mechanisme aan bij het *observer* ontwerppatroon.

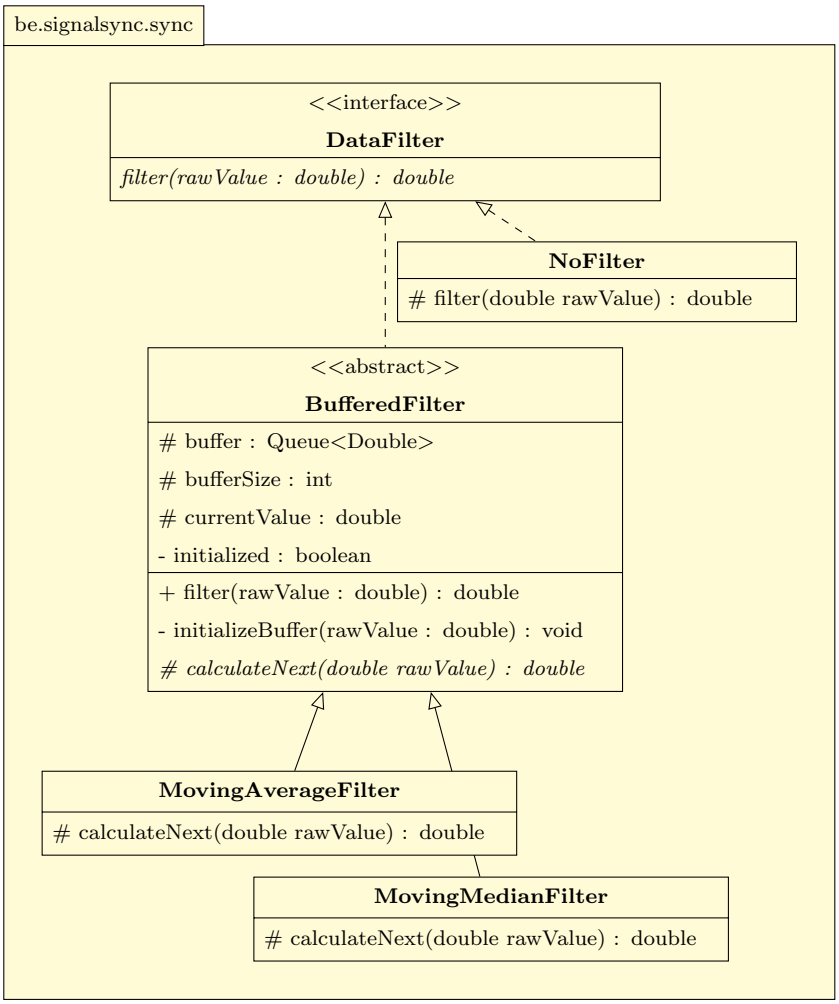
Het starten van de synchronisatie

Het starten van de synchronisatie gebeurt noch in deze klasse noch in een slicer- of strategyklasse. Zowel het slicen als het uitvoeren van de algoritmen is een deterministisch proces en afhankelijk van de samples die door een `Stream` object naar de `StreamProcessors` verstuurd worden. Bij een `AudioDispatcherStream` zal de synchronisatie starten wanneer de `start` methode opgeroepen wordt. Bij een `MSPStream` zal de synchronisatie pas starten wanneer de streams in Max/MSP zelf geactiveerd worden.

3.5.5 Filteren van de resultaten

Zoals in sectie 3.4 is besproken kunnen de latencies gefilterd worden. De meest elementaire filter wordt beschreven door de interface `DataFilter` waarin de methode `filter` wordt beschreven. Deze methode ontvangt een `double` als parameter en geeft een (gefilterde) `double` terug. De meest eenvoudige implementatie van deze klasse is de `NoFilter`. Deze filter geeft rechtstreeks de meegegeven waarde terug zonder iets te wijzigen.

Figuur 3.20: UML diagram van de verschillende datafilters.



Gebufferde filters

`DataFilter` wordt ook geïmplementeerd door de abstracte klasse `BufferedFilter`. Dit is een filter die de laatste n waarden in een buffer bijhoudt. De waarde van n wordt bepaalt in het configuratiebestand.

Deze klasse is abstract aangezien de manier waarop de uiteindelijk gefilterde waarde berekent wordt nog niet bepaald is. Hiervoor is de abstracte methode `calculateNext` voorzien. De klasse `MovingAverageFilter` doet dit door het gemiddelde van de waarden uit de buffer te nemen. Bij `MovingMedianFilter` gebeurt dit door de mediaan te berekenen.

Figuur 3.20 toont een UML diagram van deze verschillende klassen en interfaces.

3.6 Max/MSP modules

In de introductie van deze thesis (sectie 1.5) is er geschreven dat er voor het volledige synchronisatieproces een gebruiksvriendelijke interface ontwikkeld zal worden. Deze interface moet het voor musicologen/onderzoekers mogelijk maken om `StreamSets` aan te maken en te synchroniseren zonder één lijn code te hoeven schrijven.

Een heel flexibele manier om dit te doen is door het schrijven van één of meerdere Max/MSP modules. Hierbij wordt de gebruiker niet in een hokje geduwd van wat een stream precies moet zijn of hoe het moet worden ingelezen. Bij deze benadering is het een garantie dat elk Max/MSP signaal gesynchroniseerd kan worden. Max/MSP ondersteund standaard het inlezen van bestanden of microfoons.

Problematisch is het feit dat de streams afkomstig van Teensy microcontrollers standaard niet kunnen worden aangesproken vanuit een Max/MSP context. Aangezien dit voor dit onderzoek een vereiste is zal hier ook een module voor geschreven moeten worden.

3.6.1 Inlezen van de Teensy microcontroller

De klasse `TeensyReader` bevat de implementatie van een Max/MSP module waarmee signalen afkomstig van Teensy microcontrollers kunnen worden ingelezen. Het aanmaken van het object kan met volgende code:

```
mxj~ be.signalsync.msp.TeensyReader <<parameters>>.
```

Er zijn 5 parameters vereist:

Poort

De COM-poort waarmee de Teensy microcontroller communiceert met de computer. Wanneer de naam van de poort vaststaat kan deze hier letterlijk worden meegegeven. Het is ook mogelijk om een index (vanaf 0) op te geven die. Indien er 3 microcontrollers zijn aangesloten kan er 0, 1 of 2 worden opgegeven. Deze parameter kan ook worden weggelaten. In dat geval wordt automatisch de Teensy met index 0 gekozen.

Samplefrequentie

Dit is de samplefrequentie in Hertz waaraan de Teensy microcontroller met de computer communiceert. Alle binnenkomende signalen worden geresampled naar de samplefrequentie van Max/MSP. De meest courante waarden zijn 8000 of 11025.

Het is aangeraden (maar geen vereiste) om de Max/MSP samplefrequentie in te stellen op een veelvoud van de Teensy samplefrequentie. Dit vereenvoudigd het resamplingproces en heeft een positieve invloed op de geluidskwaliteit.

Startkanaal

De index van de eerste analoge pin waarvan het signaal moet worden ingelezen. Indien het eerste in te lezen signaal zich op pin A3 bevindt, dan moet deze parameter worden ingesteld met waarde 3.

Audiokanaal

De index van het audiokanaal dat gebruikt zal worden voor de synchronisatie. De index begint te tellen vanaf het startkanaal. Indien er gestart wordt op pin A3 en het audiosignaal zich op pin A5 bevindt, dan zal deze parameter waarde 2 moeten hebben. Het signaal van dit kanaal ondergaat na het inlezen een extra filtering waarbij het geluid geoptimaliseerd wordt. Bij het invullen van een ongeldig getal (te hoog of negatief) wordt de filtering op geen enkel kanaal toegepast.

Aantal kanalen

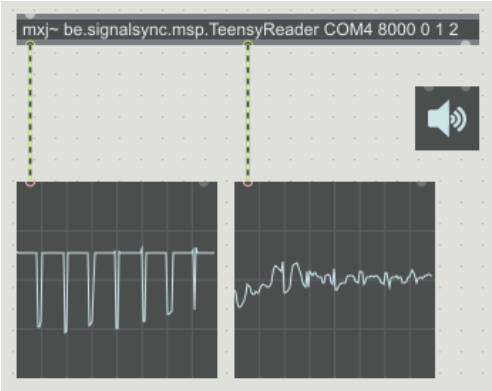
Met deze parameter wordt het aantal in te lezen kanalen ingesteld.

De parameters dienen als een door spaties gescheiden lijst te worden meegegeven aan de module. Na het valideren van de parameters wordt de module aangemaakt. Het aantal uitgaande verbindingen van deze module wordt bepaald door de parameter die het aantal kanalen instelt.

Figuur 3.21 toont deze module waarbij de uitgaande verbindingen zijn aangesloten op scope objecten. Hierdoor wordt de data gevisualiseerd.

Een uitgebreide handleiding voor het gebruik van deze module is te vinden in bijlage X (TODO).

Figuur 3.21: De TeensyReader module in actie. De Teensy samplet aan een frequentie van $8000Hz$, de microfoon is aangesloten op pin A1, de infraroodsensor is aangesloten op pin A0. De linkse scope toont de gegevens van de infraroodsensor, de rechtse sensor toont de geluidsgolf.



Implementatie

Een Max/MSP module kan in Java geschreven worden door een klasse te laten overerven van `MSPPerformer`. Bij de constructie worden de *outlets* (de uitgaande verbindingen) geïnitialiseerd en wordt er een `TeensyDAQ` object aangemaakt. Dit is een object uit afkomstig uit de `TeensyDAQ` bibliotheek ontwikkeld aan het IPEM. Deze bibliotheek biedt een low-level interface naar de Teensy microcontroller aan. Na het aanmaken van dit object registreert de module zich als *handler*. Om dit mogelijk te maken wordt de methode `handle` van de interface `DAQDataHandler` geïmplementeerd. Na het registreren wordt deze methode consequent opgeroepen met samples afkomstig van de Teensy microcontroller. Deze samples worden vervolgens gebufferd.

Een klasse die overerft van `MSPPerformer` moet de methode `perform` implementeren. In deze methode worden de samples uit de buffers gehaald, geresamplet naar de Max/MSP samplefrequentie en verstuurd door de outlets.

3.6.2 De synchronisatiemodule

De tweede module zal de synchronisatie van de streams verzorgen en is geïmplementeerd in de klasse `Sync`. De module maakt gebruik van `RealtimeSignalSync` om de latencies tussen de audiostreams te bepalen. De module kan in Max/MSP met volgende code worden aangemaakt: `mxj~ be.signalsync.msp.Sync <<parameter>>`

Instellen van de stream structuur

De module verwacht één parameter van het type `String` die beschrijft hoe de streams gestructureerd zijn. De structuur wordt voorgesteld als een door komma's gescheiden reeks van de letters 'a' en 'd'. Elk door deel bepaald de structuur van een `StreamGroup` waarin de letters 'a' en 'd' respectievelijk voor een audiostream en datastream staan. Per deel mag uiteraard maar eenmaal de letter 'a' voorkomen. De volledige tekenreeks (de verzameling van alle `StreamGroups`) stelt de structuur van een `StreamSet` voor. Op basis van deze string worden de `Stream` objecten aangemaakt en onderverdeeld in `StreamGroups` en `StreamSets`.

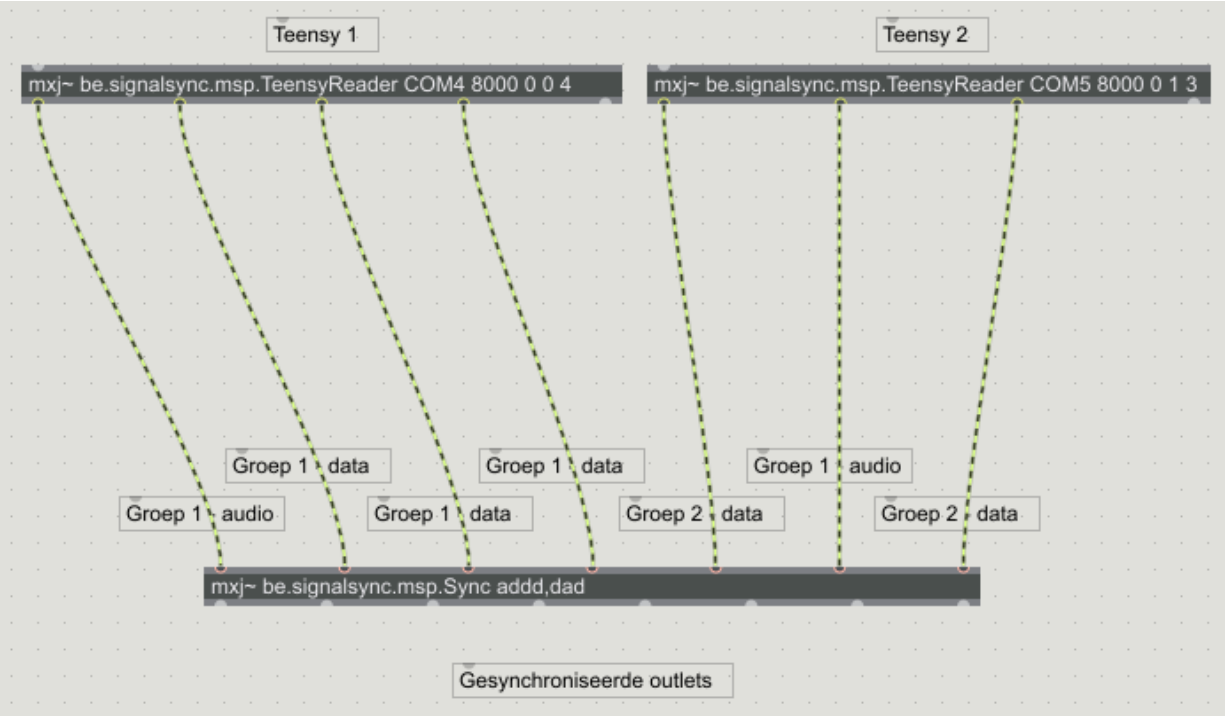
Dit zou een mogelijke parameter voor deze module kunnen zijn: `addd,dad`. De eerste `StreamGroup` bestaat uit 4 streams waarbij de synchronisatie met de eerste stream zal worden uitgevoerd. De tweede `StreamGroup` bestaat uit 3 streams. De tweede stream stelt hierbij de audiostream voor.

Figuur 3.22 toont hoe deze module gebruikt moet worden in combinatie met de `TeensyReader`. De audiostream *outlets* (bepaald door de vierde parameter van de twee `TeensyReaders`) worden verbonden met de audiostream *inlets* van de `Sync` module (bepaald door de letters 'a' van de parameter).

Implementatie

Net zoals `TeensyReader` erft deze module over van `MSPPerformer`. Ook wordt de interface `SyncEventListener` geïmplementeerd. In de constructor wordt op basis van de

Figuur 3.22: De synchronisatie van streams in Max/MSP afkomstig van twee Teensy microcontrollers.



parameter van de module een `StreamSet` bestaande uit `MSPStream` objecten aangemaakt. In de perform methode van de module worden de signaalvectoren doorgegeven aan de `MSPStream` objecten. De synchronisatie wordt verwezenlijkt door in de constructor een `RealtimeSignalSync` object aan te maken waarop de klasse zich registreert. Aan het `RealtimeSignalSync` object wordt de `StreamSet` meegegeven. Elke keer wanneer er informatie over de latency beschikbaar is wordt de methode `onSyncEvent` opgeroepen met alle latencies.

Met behulp van deze latencies moeten de binnenkomende Max/MSP datastreams gemanipuleerd worden zodat ze perfect synchroon lopen. Dit is in de laatste versie nog niet geïmplementeerd. In de huidige implementatie worden de latencies in de Max/MSP console geprint.

Hoofdstuk 4

Evaluatie

Om de kwaliteit van de softwarebibliotheek te kunnen garanderen zijn er verschillende soorten testen uitgevoerd.

De eerste soort testen zijn geschreven voor het bepalen en analyseren van de kwaliteit van de algoritmes. De algoritmes worden hierbij blootgesteld aan audiofragmenten waartussen de latency bepaalt moet worden. Met behulp van onder meer deze test zijn de optimale parameterwaarden bepaalt.

Buiten het testen van de algoritmes zijn er ook enkele unit testen geschreven voor het testen van enkele cruciale elementen van de softwarebibliotheek. Deze testen zijn cruciaal om het aantal bugs in de softwarelogica te beperken.

Het laatste deel van dit hoofdstuk zal dieper ingaan op de zaken die niet geïmplementeerd zijn maar die in de toekomst zeker een meerwaarde kunnen betekenen.

4.1 Testen van de algoritmes

Het testen van de algoritmes wordt uitgevoerd met behulp de JUnit testcase `SynchronizationTest` uit het package `be.signalsync.test`. Deze testcase laat toe om de slices van verschillende audiofragmenten met elkaar te matchen en te analyseren waar de algoritmes precies in de fout gaan.

4.1.1 Aanmaken de dataset

Bij het uitvoeren van deze test is het de bedoeling om enkel de algoritmes te testen. Om niet afhankelijk te zijn van andere softwareonderdelen wordt de dataset op voorhand aangemaakt. Het aanmaken van deze dataset gebeurt in twee stappen. Eerst wordt het originele audiofragment gewijzigd door er bijvoorbeeld latency aan toe te voegen. Dit gebeurt met behulp van een Perl script. Vervolgens worden de verschillende audiofragmenten in slices geknipt en opgeslagen. In de testcase worden de algoritmes rechtstreeks op deze slices uitgevoerd zonder andere softwareonderdelen aan te roepen.

4.2 Praktijktesten

4.3 Testen van de softwarecomponenten

4.4 Mogelijke verbeteringen

Hoofdstuk 5

Conclusie

Bijlagen

Bijlage A

Resultaten DTW experiment

In dit experiment proberen we de nauwkeurigheid van het DTW algoritme te bepalen wanneer streams gebufferd worden. Hiertoe bepaalden we eerst de latency tussen twee audiofragmenten. Vervolgens verkleinden we iteratief de duur van het fragment met 10 seconden waarop we het algoritme opnieuw uitvoerden. Tenslotte vergeleken we de buffergrootte en nauwkeurigheid van de resultaten.

We hebben gebruik gemaakt van twee audiofragmenten waarbij het ene fragment 2.390 seconden vertraging heeft ten opzichte van het andere fragment. Beide fragmenten hebben samplefrequentie van 8000 Hz. Eén van de twee fragmenten is een opname van het origineel en bijgevolg van matige kwaliteit.

Het experiment is uitgevoerd in *Sonic Visualiser* met behulp van de *Match Performance Aligner* plug-in. Deze plug-in laat synchronisatie toe met behulp van het DTW algoritme. De implementatie wordt uitgebreider besproken in artikel [11]. Voor dit experiment hebben we de default instellingen gebruikt. De plug-in bepaalt elke twintig milliseconden de latency tussen beide fragmenten.

De volgende tabel geeft de resultaten van het experiment weer. De eerste kolom bevat de lengte van de vergeleken fragmenten in seconden. Deze lengte stelt de buffergrootte voor van een audiostream. De tweede kolom geeft aan hoeveel seconden van de stream moet worden verwerkt tot er een stabiel resultaat wordt bekomen. De derde kolom geeft het

gemiddelde weer van de gevonden latencies. Deze waarde wordt berekend vanaf dat het algoritme een stabiel resultaat heeft gevonden. De vierde kolom bevat de standaardafwijking van dit resultaat.

Lengte	**Tijd tot stabiel**	**Gemiddelde latency**	**Standaardafwijking**
60s	2.540s	2,393s	0.048s
50s	2.540s	2,390s	0.095s
40s	2.540s	2,394s	0.020s
30s	2.540s	2,384s	0.145s
20s	2.540s	2,390s	0.108s
10s	2.540s	2,395s	0.025s

Uit bovenstaande resultaten kunnen we verschillende zaken concluderen. Ten eerste zien we aan de standaardafwijking dat de individuele resultaten (die iedere 20ms gegenereerd worden) niet nauwkeurig genoeg zijn om te gebruiken in onze toepassing. De gemiddelde waarde komt wel in de buurt van de werkelijke latency maar is nog steeds niet zo nauwkeurig. Ook moeten we bij de berekening van het gemiddelde rekening houden met het feit dat het algoritme pas na een bepaalde tijd een stabiel resultaat vindt, in dit geval 2.540s.

We hebben dit algoritme ook uitgetest op een fragment waaruit 500 ms hebben weggeknipt om het probleem met gedropte samples te simuleren. Het algoritme reageerde hier zeer snel op: de nieuwe latency werd na 240 ms gevonden. Het probleem is dat we zojuist hebben getracht de nauwkeurigheid te verbeteren door het gemiddelde te nemen van de resultaten. Dit heeft als gevolg dat wanneer er samples gedropt zijn het eindresultaat zich bevindt tussen de initiële en nieuwe latency.

Referentielijst

[1] Audacity. http://audacity.sourceforge.net/, 2015. [Online; geraadpleegd 12-maart-2016].

[2] Cycling '74 Max. https://cycling74.com/, 2016. [Online; geraadpleegd 12-maart-2016].

[3] Ipem - systematic musicology. https://www.ugent.be/lw/kunstwetenschappen/en/research-groups/musicology/ipem, 2016. [Online; geraadpleegd 05-maart-2016].

[4] Teensy USB Development Board. https://www.pjrc.com/teensy/, 2016. [Online; geraadpleegd 19-maart-2016].

[5] A Digital Media Primer for Geeks. https://xiph.org/video/vid1.shtml, 2016. [Online; geraadpleegd 21-maart-2016].

[6] David Bannach, Oliver Amft, and Paul Lukowicz. Automatic event-based synchronization of multimodal data streams from wearable and ambient sensors. In *Smart sensing and context*, pages 135–148. Springer, 2009.

[7] Benjamin Barras. Sox: Sound exchange. Technical report, 2012.

[8] Juan Pablo Bello, Laurent Daudet, Samer Abdallah, Chris Duxbury, Mike Davies, and Mark B Sandler. A tutorial on onset detection in music signals. *Speech and Audio Processing, IEEE Transactions on*, 13(5):1035–1047, 2005.

[9] Chris Cannam, Christian Landone, and Mark Sandler. Sonic visualiser: An open source application for viewing, analysing, and annotating music audio files. In *Proceedings of the 18th ACM international conference on Multimedia*, pages 1467–1468. ACM, 2010.

[10] Simon Dixon. Live tracking of musical performances using on-line time warping. In *Proceedings of the 8th International Conference on Digital Audio Effects*, pages 92–97. Citeseer, 2005.

[11] Simon Dixon and Gerhard Widmer. Match: A music alignment tool chest. In *ISMIR*, pages 492–497, 2005.

[12] B. Fries and M. Fries. *Digital Audio Essentials: A comprehensive guide to creating, recording, editing, and sharing music and other audio*. O'Reilly Digital Studio. O'Reilly Media, 2005. ISBN 9781491925638.

[13] Javier Jaimovich and Benjamin Knapp. Synchronization of multimodal recordings for musical performance research. In *NIME*, pages 372–374, 2010.

[14] Roman Kollár. Configuration of ffmpeg for high stability during encoding.

[15] Harry Nyquist. Certain topics in telegraph transmission theory. 1928.

[16] Alan V Oppenheim. Speech spectrograms using the fast fourier transform. *IEEE spectrum*, 8(7):57–62, 1970.

[17] Chotirat Ann Ratanamahatana and Eamonn Keogh. Everything you know about dynamic time warping is wrong. In *Third Workshop on Mining Temporal and Sequential Data*. Citeseer, 2004.

[18] Stan Salvador and Philip Chan. Toward accurate dynamic time warping in linear time and space. *Intelligent Data Analysis*, 11(5):561–580, 2007.

[19] Joren Six. *Digital Sound Processing and Java*. UGent, IPEM, Sint-Pietersnieuwstraat 41, 9000 Ghent - Belgium, 5 2015.

[20] Joren Six and Marc Leman. Panako - A Scalable Acoustic Fingerprinting System Handling Time-Scale and Pitch Modification. In *Proceedings of the 15th ISMIR Conference (ISMIR 2014)*, 2014.

[21] Joren Six and Marc Leman. Synchronizing Multimodal Recordings Using Audio-To-Audio Alignment. *Journal of Multimodal User Interfaces*, 9(3):223–229, 2015. ISSN 1783-7677. doi: 10.1007/s12193-015-0196-1.

[22] Joren Six, Olmo Cornelis, and Marc Leman. TarsosDSP, a Real-Time Audio Processing Framework in Java. In *Proceedings of the 53rd AES Conference (AES 53rd)*. The Audio Engineering Society, 2014.

[23] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.

[24] Avery Li-Chun Wang. An industrial-strength audio search algorithm. In *ISMIR 2003, 4th Symposium Conference on Music Information Retrieval*, pages 7–13, 2003.

Lijst van figuren

Lijst van tabellen

Lijst van codefragmenten