Tijdsynchronisatie met Arduinos

Patrick van Looy & Bram Leenders

27 mei 2014

1 Inleiding

Om energiezuinige draadloze communicatie mogelijk te maken, is er tijdsynchronisatie nodig tussen de verschillende nodes. Wanneer nodes gesynchroniseerd zijn kunnen ze op vaste momenten naar elkaar zenden en luisteren, en kunnen ze de rest van de tijd in een slaapstand zijn. Omdat communicatie relatief veel energie vraagt en een slaapstand zeer weinig, biedt synchronisatie dus mogelijkheden voor energiebesparing.

Tijdsynchronisatie kan op verschillende manieren geimplementeerd worden. In dit onderzoek gaan we een verschijnsel nabootsen wat we in de natuur ook kunnen vinden. De voorbeelden die hierbij gehanteerd worden, zijn vuurvliegjes en krekels. Vuurvliegjes, bijvoorbeeld, willen van nature synchroon knipperen[1]. Ditzelfde gedrag kan door Arduino's nagebootst worden, met behulp van radio- of geluidssignalen.

In dit onderzoek kijken we naar een specifiek algoritme, het firefly algoritme[3, 6]. Dit algoritme implementeren we zowel voor radiocommunicatie als voor communicatie met behulp van geluid. In sectie 2 geven we een kort overzicht van eisen waaraan het protocol dient te voldoen. Sectie 3 geeft een overzicht van de context waarin dit werk gelezen dient te worden. In sectie 4.1 wordt het algoritme beschreven dat hiervoor gebruikt is. Sectie 4.2 beschrijft de implementatie voor radiocommunicatie, en sectie 4.3 beschrijft dit voor synchronisatie met behulp van geluid. In sectie 5 wordt besproken hoe de beide implemenaties functioneren, en geven we een korte discussie over de beide manieren van synchronisatie.

2 Probleemstelling

Voor het synchroniseren van Arduino's is een synchronisatiealgoritme nodig. Het algoritme en de implementatie dienen aan de volgende eisen te voldoen:

- Nodes kunnen uit een groep verdwijnen zonder rest te beinvloeden.
- Nodes kunnen toegevoegd worden aan een groep, en het geheel synchroniseert.
- Twee groepen kunnen samengevoegd worden, en het geheel synchroniseert.
- Wanneer nodes uit synchronisatie raken, herstellen ze hun gesynchroniseerde staat.

- Frequentie van communicatie ligt zo laag mogelijk.

Uit deze eisen volgt dat er geen master-slave configuratie mogelijk is, omdat de master uit de groep kan verdwijnen. Er is een peer-to-peer configuratie nodig, waarbij iedere node gelijk is.

In dit onderzoek zullen we de vraag behandelen of we (een variant op) het firefly algoritme kunnen implementeren op dusdanige manier dat de implementatie aan bovenstaande eisen voldoet.

3 Gerelateerd werk

De brede toepasbaarheid van draadloze (sensor)netwerken heeft er voor gezorgd dat er al veel onderzoek naar synchronisatie van netwerken met agents gedaan is. Ook in ons geval is er al veel met het Firefly-algoritme geëxperimenteerd. Gezien de toepasbaarheid in grotere draadloze netwerken (zie bijvoorbeeld [5]), focust onderzoek veelal op de invloed van transmissie en propagatie delay en de beperking dat een node niet tegelijkertijd kan zenden en ontvangen. Voor netwerken die een groot oppervlak overspannen kunnen deze factoren een beperking zijn.

Zoals gezegd in sectie 2 moet het netwerk zichzelf kunnen organiseren. Heel simpel gezegd bestaat het netwerk uit een heleboel kleine, op zichzelf staande entiteiten die simpele gedragsregels kennen. Ze hebben elk een beperkt lokaal zicht op het netwerk, en vormen samen het netwerk. Dit grote systeem is adaptief waardoor veranderingen in het netwerk geen problemen voor het netwerk in zijn geheel opleveren. Tevens is het netwerk volledig schaalbaar.

In feite zijn er een aantal mogelijkheden om een synchronisatiestrategie te bedenken voor zo'n netwerk. Er kan gekozen worden voor een master-slave implementatie, mutual synchronization of een combinatie van die twee. In de echte wereld kennen we dit als een monarchie, een democratie of een oligarchie. Een voorbeeld van een master-slave implementatie is bijvoorbeeld het Berkeley algoritme [2] dat wordt gebruikt in LAN-synchronisatie. Waar wij naartoe willen is een implementatie van het tweede principe; mutual synchronization.

Het firefly-algoritme [6] is op dit principe gebaseerd. Dit algoritme is echter niet practisch implementeerbaar, omdat het er van uit gaat dat synchronisatie pulsen oneindig kort zijn, er geen delays zijn, nodes tegelijkertijd kunnen luisteren en versturen en dat alle nodes samen een compleet netwerk vormen (iedereen binnen zendbereik van elkaar is). Waarneer er aan een of meerdere van deze aannames niet voldaan wordt, is synchronsatie onstabiel. Hierdoor is het onhaalbaar om dit een op een door te zetten naar draadloze systemen. Een voorbeeld waarom het niet zou kunnen werken is dat als er delays zijn, het mogelijk is dat nodes echos ontvangen van hun eigen puls.

Gelukkig is hiervoor een alternatief beschikbaar, namelijk het Meshed Emergent Firefly Synchronization (MEMFIS) algoritme [5]. Dit algoritme houdt rekening met de technologische beperkingen van draadloze netwerken met behoud van de eigenschappen van firefly synchronisatie. Een belangrijk kenmerk van het ontwerp is dat er een gemeenschappelijk synchronisatiewoord is ingebed in elk payload packet. Dit synschonisatiewoord wordt gedetecteerd bij de ontvanger door gebruik te maken van een cross-correlator. Delays worden afgehandeld door het synchronisatiealgoritme te verbeteren. Het resultaat is dat synchro-

nisatie geleidelijk ontstaat als nodes onderling willekeurig pakketen uitwisselen. Hierdoor is er ook geen speciale synchornisatiefase nodig.

4 Implementatie

4.1 Algoritme

Het gebruikte algoritme lijkt op het firefly algoritme: het is een homogeen peer-to-peer netwerk. Er zijn dus geen masters of slaves, maar iedere node reageert op de signalen van alle omliggende nodes.

Iedere ronde wacht een node eerst een bepaalde tijd, en stuurt daarna een signaal. Als de node een signaal ontvangt halveert hij de tijd tot het zenden. Om te voorkomen dat twee nodes elkaar triggeren, begint iedere ronde met een halve periode slaap. Deze halve periode, waarin de node niet reageert op signalen, wordt de refractory period genoemd.

Algorithm 1 Synchronisatiealgoritme

```
loop
    if t ≥ periode then
        Zendt signaal
        Sleep(½ periode)
        t ← ½ periode
    end if
    if Signaal ontvangen then
        t ← periode − ½(periode − t)
    else
        Sleep(stapgrootte)
        t ← (t + stapgrootte)
    end if
end loop
```

Algoritme 1 geeft een formele beschrijving van het algoritme. Hierin is t de huidige fase van een node. Merk op dat de stapgrootte significant kleiner moet zijn dan de periode. Dit algoritme voldoet aan de eisen die worden gesteld in de probleemstelling (sectie 2). Idealiter halveert het tijdsverschil tussen twee nodes iedere periode. In de praktijk zal dit iets lager zijn, in verband met de verwerkingstijd en de propagatietijd.

4.2 Radiosynchronisatie

Om tijdsynchronisatie op Arduinos uit te kunnen voeren, is een middel nodig waarmee dat gedaan wordt. In dit geval gebruiken we hier een radio voor. Hierdoor kunnen de Arduinos onderling met elkaar communiceren. Op deze manier kunnen ze synchroon gaan lopen door elkaar te informeren over hun status.

Onze Arduinos zijn allemaal met een ledje uitgerust om zo een firefly na te bootsen. De frequentie van het knipperen van een ledje is van tevoren vastgelegd. Telkens wanneer een node zijn ledje laat knipperen, zendt deze tevens een radiosignaal uit. Wanneer andere nodes dit signaal oppikken (dus in hun

luisterfase zitten), zullen zij zich de volgende ronde hierop aanpassen volgens het algoritme. Na enige tijd zullen de Arduinos nagenoeg synchroon lopen.

4.3 Geluidssynchronisatie

De implementatie beschreven in de vorige secties is niet afhankelijk van het precieze signaal dat de Arduino's geven. Het is alleen afhankelijk van het moment waarop het signaal uitgezonden en ontvangen wordt, en de tijd hiertussen mag niet exorbitant groot worden of wisselend lang en kort duren.

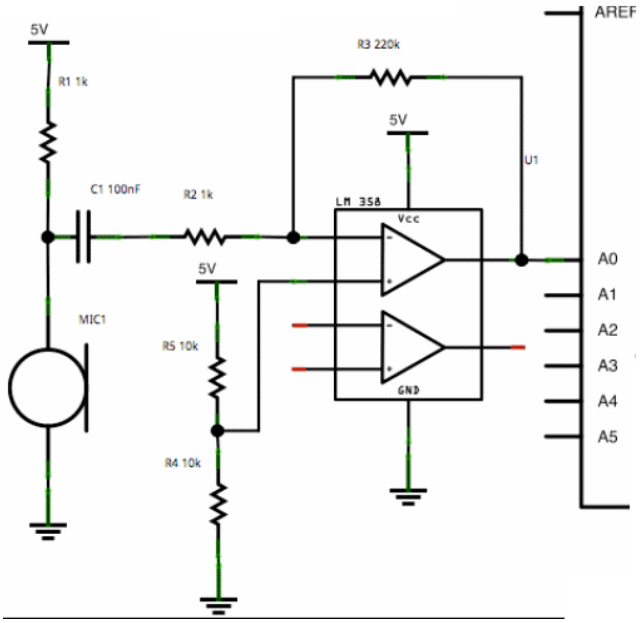
In plaats van een radiosignaal kunnen ook andere signalen uitgewisseld worden, bijvoorbeeld een geluidssignaal. De implementatie hiervan heeft wel wat meer voeten in de aarde, omdat er erg veel ruis is in de vorm van omgevingsgeluid. Tevens heeft de Arduino niet een standaardimplementatie die pieken kan detecteren; er is dus geen functie voor microfoons die vergelijkbaar is met `radio.available()`.

4.4 Analoge signaalverwerking

Een microfoon levert geen geschikt signaal op dat digitaal verwerkt kan worden. Het signaal is te zacht, bevat veel ruis en is analoog. Om het door de Arduino te laten verwerken moet het signaal versterkt worden en omgezet worden naar een digitaal signaal. Dit doen we in drie stappen:

- *High-pass filter:* dit filter laat alleen de tonen boven een bepaalde ondergrens door, waardoor lage omgevingsgeluiden gefilterd worden. Dit vermindert dus de hoeveelheid ruis in het signaal.
- *Versterker:* omdat de microfoon een zwak signaal levert, moet het versterkt worden.
- *Omzetten naar digitaal signaal:* met behulp van een ADC (analog to digital converter) kan het gefilterde, versterke signaal omgezet worden naar een digitale input.

Het circuit dat voor dit onderzoek gebruikt is, staat in figuur 1. Hierbij wordt gebruik gemaakt van de ADC die standaard beschikbaar is op de Arduino Uno, die een analoog input signaal tussen 0 en 5 volt heeft en als digitale output een getal tussen de 0 en 1024 geeft.



Figuur 1: High-pass filter met versterker en ADC. *Bron: CreaTe Protobox quick reference sheet.*

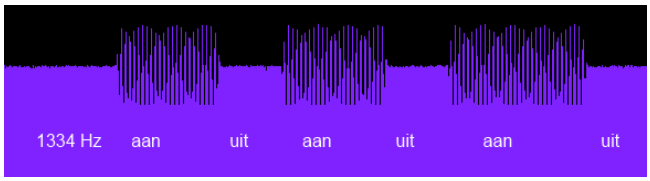
4.5 Signaalverwerking

In figuur 2 is zichtbaar dat het versturen van een geluidssignaal een sterk wisselend inputsignaal geeft. We kunnen dus stellen dat als het verschil tussen twee opeenvolgende metingen erg verschilt, dat er dan zeer waarschijnlijk een signaal ontvangen wordt. Neem $v(t)$ de waarde gemeten op tijdstip t zijn, dan $|v(t) - v(t+1)| > \text{threshold} \Rightarrow$ signaal ontvangen.

Hierbij is het van belang dat de threshold hoog genoeg gekozen wordt om ruis uit te sluiten, maar ook niet zo hoog dat signalen niet opgemerkt worden. Omdat dit moeilijk van tevoren vast te stellen is, hebben we gebruik gemaakt van een dynamische threshold gebaseerd op de gemiddelde afwijking. De gemiddelde afwijking (avgdiff) als functie van de tijd is

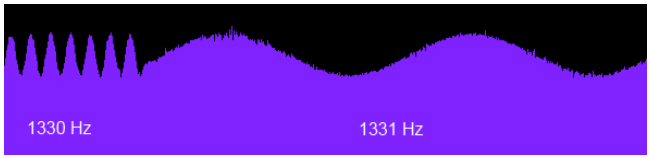
$$\text{avgdiff}(t+1) = 0.1 \times |v(t) - v(t+1)| + 0.9 \times \text{avgdiff}(t)$$

Dit is dus een gemiddelde van de afwijkingen tussen metingen, waarbij het "gewicht"van een meting exponentieel snel afneemt. De eerste meting telt dus vrijwel niet mee, en de laatste meting relatief zwaar (10%). De threshold is $n \cdot$ avgdiff: als het verschil tussen metingen meer dan een factor n verschillen ten opzichte van de gemiddelde afwijking, gaan we ervanuit dat er een signaal ontvangen wordt.



Figuur 2: Inkomend signaal bij pulserend signaal (van speakers; niet van Arduino).

In figuur 3 is het input signaal bij twee geluidsfrequenties geplot. Beide zijn met hetzelfde opstelling gemeten, deze opstelling staat hierboven beschreven. Merk op dat het verschil tussen twee opeenvolgende metingen bij 1331Hz veel lager is dan het verschil bij 1330Hz. Om de bovenstaande manier te gebruiken is het van belang dat opeenvolgende metingen erg verschillen, en is het dus beter om 1330Hz te gebruiken dan 1331Hz.



Figuur 3: Inkomend signaal bij verschillende geluidsfrequencies (van speakers; niet van Arduino).

5 Resultaten en discussie

5.1 Resultaten radiosynchronisatie

De radiosynchronisatie synchroniseert erg snel; bij een groep van twee nodes halveert het tijdsverschil per synchronisatieronde. Bij grotere groepen nodes is het moeilijker om een constante verbetering te voorspellen. Lucarelli en Wang[4] hebben bewezen dat een netwerk dat gebruik maakt van het firefly algoritme leidt tot een convergentie. Merk op dat hetzelfde geldt voor geluidssynchronisatie.

Tijdens de tests hebben we gemerkt dat synchronisatie minder snel verloopt als twee (groepen) nodes via één tussennode verbonden zijn. De twee groepen synchroniseerden wel, maar minder snel dan wanneer alle nodes binnen ontvangsbereik zijn. Wel is een voordeel van de radiosynchronisatie dat het bereik van een zender vrij groot is; twee nodes op 20 meter afstand kunnen nog synchroniseren.

Een nadeel aan het gebruik van de radiosynchronisatie, is dat de radio gedurende de helft van de tijd luistert naar signalen. Dit kost relatief veel energie, maar als de radio minder dan de helft van de tijd luistert kan niet gegarandeerd worden dat het netwerk in alle gevallen synchroniseert.

5.2 Resultaten geluidssynschronisatie

Zoals al gezegd in sectie 4.3 heeft de microfoon erg veel storing van omgevingsgeluid, zoals pratende mensen of rijdende auto's.

Het gebruikte algoritme wordt niet gestoord door false negatives (het niet ontvangen van een signaal). Bij een false negative duurt de synchronisatie iets langer, maar wordt de reeds behaalde synchronisatie niet gestoord.

Bij een false positive (ruis dat wordt herkend als signaal) is dat niet het geval. Als alle nodes perfect synchroon zouden lopen én allemaal de false positive ontvangen worden ze niet verstoord, omdat iedere node dan dezelfde tijd opschuift. Echter, in de praktijk ontvangt slechts een deel van de nodes het signaal en wordt de groep verstoord.

We kunnen hier dus uit afleiden dat de threshold vrij hoog moet staan: het is beter om een aantal signalen te missen dan om ruis aan te zien voor een signaal. Tijdens de tests bleek een factor $n = 3$ een optimaal resultaat te geven in onze ruimte. Een hogere factor filterde alle echte signalen, en een lagere factor liet de ruis door.

5.3 Vergelijking

Beide methodes hebben duidelijk voor- en nadelen. Radiosynchronisatie is erg betrouwbaar en convergeert snel maar kost relatief veel energie. Geluidssynchronisatie kost weinig energie maar convergeert langzamer door false negatives, en raakt vaak uit balans door false positives.

Helaas kon onze opstelling geen hoogfrequente signalen testen; alle pulsen lagen in het spectrum dat mensen kunnen horen. Voor veel toepassingen zal een hoogfrequent signaal praktischer zijn, omdat mensen dan geen last ondervinden van de signalen. Als het systeem bijvoorbeeld in een bibliotheek geïmplementeerd wordt, dan mag het geen hoorbare signalen geven omdat dit storend is voor de bezoekers.

6 Conclusie

In dit onderzoek zijn twee implementaties van het Meshed Emergent Firefly Synchronization algoritme getest. In de resultaten was duidelijk terug te zien dat de radiosynchronisatie minder last ondervindt van ruis. Op basis van onze resultaten raden we dus aan om een radio te gebruiken, mits hiervoor genoeg stroom is. In een stille omgeving, waarbij weinig ruis is, kan ook voor geluidssynchronisatie gekozen worden.

Dit onderzoek heeft slechts twee mogelijke implementaties onderzocht, een later onderzoek zou alternatieve signalen zoals licht kunnen onderzoeken. Tevens gebruiken onze implementaties een pulsfrequentie. Het zou interessant zijn om te onderzoeken of de pulsfrequentie verlaagd kan worden wanneer het netwerk gesynchroniseerd is; dit zou het energieverbruik misschien nog verder kunnen verlagen.

Referenties

[1] John Buck. Synchronous rhythmic flashing of fireflies. ii. *Quarterly Review of Biology*, pages 265–289, 1988.

[2] Riccardo Gusella and Stefano Zatti. The accuracy of the clock synchronization achieved by tempo in berkeley unix 4.3 bsd. *Software Engineering, IEEE Transactions on*, 15(7):847–853, 1989.

[3] Robert Leidenfrost and Wilfried Elmenreich. Firefly clock synchronization in an 802.15. 4 wireless network. *EURASIP Journal on Embedded Systems*, 2009:7, 2009.

[4] Dennis Lucarelli, I-Jeng Wang, et al. Decentralized synchronization protocols with nearest neighbor communication. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 62–68. ACM, 2004.

[5] Alexander Tyrrell, Gunther Auer, and Christian Bettstetter. Emergent slot synchronization in wireless networks. *Mobile Computing, IEEE Transactions on*, 9(5):719–732, 2010.

[6] Xin-She Yang and Xingshi He. Firefly algorithm: recent advances and applications. *International Journal of Swarm Intelligence*, 1(1):36–50, 2013.