Juuso Tuononen

Kaiutinsovelluksen äänen viiveen pienentäminen

Tietotekniikan pro gradu -tutkielma

4. maaliskuuta 2023

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Juuso Tuononen

Yhteystiedot: juuso.j.tuononen@student.jyu.fi

Ohjaajat: Timo Hämäläinen ja Ari Viinikainen

Työn nimi: Kaiutinsovelluksen äänen viiveen pienentäminen

Title in English: Reducing speaker app's sound latency

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmisto- ja tietoliikennetekniikka

Sivumäärä: 62+0

Tiivistelmä: Tässä pro gradu -tutkielmassa jatkokehitetään kaiutinsovelluksen prototyyppiä. Kyseisellä sovelluksella Android-puhelinta voi käyttää tietokoneen kaiuttimena. Jatkokehittäminen toteutetaan tutkimuksena, jossa käytetään suunnittelutieteelle tyypillistä iteratiivista prosessia. Tutkimuksen tavoitteena on pienentää sovelluksen äänen viivettä, joka on tutkimuksen lähtötilanteessa noin 135 millisekuntia. Tutkimuksen myötä kaiutinsovelluksen äänen viive pienentyi noin 30 millisekuntiin.

Avainsanat: Android, Kotlin, Rust, TCP, USB, äänen viive

Abstract: In this master's thesis a speaker app prototype is developed further. With the speaker app it is possible to use an Android phone as a computer speaker. Developing is done as a research with an iterative process typical to design science. Research objective is to reduce the speaker app's sound latency, which is about 135 milliseconds at the beginning of the research. The research reduced the sound latency to about 30 milliseconds.

Keywords: Android, Kotlin, Rust, TCP, USB, sound latency

Kuviot

Sisältö

1 Johdanto

Tässä tutkielmassa jatkokehitetään TIES504 Tietotekniikan erikoistyö -kurssilla toteutettua ja avointa lähdekoodia olevaa Jonect-kaiutinsovellusta. Kyseisellä sovelluksella voi tehdä Android-puhelimesta tietokoneen kaiuttimen. Jatkokehityksen päämääränä tässä tutkielmassa on saada sovelluksen äänen viive lähemmäs normaalia tietokoneen kaiutinta. Jatkokehitys tehdään suunnittelutieteellisenä tutkimuksena, jonka tutkimuskysymyksenä voi pitää sitä, että kuinka lähelle kaiutinsovelluksella päästään normaalin kaiuttimen äänen viivettä.

Tutkimus on hyödyllinen käytännössä, mikäli sovellus julkaistaan jossain vaiheessa. Tällöin käyttäjät pääsevät hyödyntämään tutkimuksen tuloksia. Sovelluksesta on hyötyä erityisesti niille käyttäjille, jotka omistavat vain puhelimen sekä tietokoneen, eikä heidän tietokoneessa ole sisäänrakennettua kaiutinta. Tällöin heidän ei välttämättä tarvitse hankkia erillistä kaiutinta tietokoneeseen. Sovellus myös mahdollisesti säästää luonnonvaroja kaiuttimien valmistuksen osalta, koska sovellus mahdollisesti vähentää erillisten kaiuttimien kysyntää.

Äänen viive vaikuttaa sovelluksen käyttömukavuuteen. Esimerkiksi videota katsottaessa sen ääni ja kuva toistuu käyttäjälle epäsynkronisesti. Epäsynkronisuuden havaitsemisesta on tehty erilaisia tutkimuksia. Kansainvälisen televiestintäliiton (ITU) radioviestintäsektorin (ITU-R) tekemän tutkimuksen (1998) mukaan kynnysarvo ihmisen havaittavissa olevalle äänen viiveelle on keskimäärin noin 125 millisekuntia. Steinmetzin (1996) tutkimuksessa 80 millisekuntia on hyväksyttävä määrä viivettä useimmille katselijoille. Euroopan yleisradiounionin (2007) suositus äänen viiveelle on 60 millisekuntia tai vähemmän.

Tutkimuksessa yritetään tavoitella mahdollisimman pientä viivettä, mutta vähimmäisvaatimus tutkimuksen onnistumiselle on käytännössä huomaamaton äänen viive. Edellä esitettyihin viiveisiin luultavasti on mahdollista päästä Android-laitteilla, jotka tukevat `android.hardware.audio.low_latency`-ominaisuutta. Googlen (2022) dokumentaation perusteella äänen jatkuvan ulostulon viive on enintään 45 millisekuntia kyseisen ominaisuuden omaavilla laitteilla. Kaiutinsovelluksen tapauksessa pitää huomioida myös äänidatan siirron viive tietokoneelta puhelimelle. Luultavasti USB-yhteydellä tiedonsiirrosta aiheutuva äänen viive on riittävän pieni, joten sitä käytetään tutkimuksessa tiedonsiirtoon.

Luvussa 2 käsitellään tutkimuksessa käytettävää tutkimusmenetelmää, eli suunnittelutiedettä, ja miten sitä sovelletaan tähän tutkimukseen. Luvussa 3 käsitellään kaiutinsovelluksen rakennetta tutkimuksen lähtötilanteessa. Luvussa 4 käsitellään kaiutinsovelluksen äänen viiveeseen mahdollisesti vaikuttavia asioita. Luvussa 5 käsitellään erilaisia äänen viiveen mittausmenetelmiä ja tässä tutkimuksessa käytettävää äänen viiveen mittausmenetelmää. Luvussa 6 yritetään vähentää kaiutinsovelluksen äänen viivettä. Luvussa 7 mietitään kuinka luotettavia tutkimuksen tulokset ovat ja onnistuiko tutkimus. Luvussa 8 tehdään yhteenveto tutkimuksesta.

2 Suunnittelutiede

Suunnittelutieteessä luodaan uusia artefakteja iteratiivisella prosessilla, jossa luotua artefaktia arvioidaan arviointimenetelmillä ja arvioinnin perusteella kehitetään parempi versio artefaktista. Tietojärjestelmien tutkimisen ja suunnittelutieteen kontekstissa artefaktilla voidaan tarkoittaa esimerkiksi tietokoneohjelmia. (Hevner ym. 2004)

Suunnittelutiede tutkimusmenetelmänä lienee käytännössä synonyymi konstruktiiviselle tutkimukselle. Ainakin Rochan ym. (2012) artikkelissa näitä termejä pidetään ilmeisesti samaa tarkoittavina asioina.

2.1 Tyypillisiä asioita suunnittelutieteellisille tutkimuksille

Hevner ym. (2004) ovat listanneet seitsemän suuntaa antavaa ohjetta tietojärjestelmiin liittyville ja suunnittelutieteellisille tutkimuksille.

Ensimmäisessä ohjeessa neuvotaan tekemään suunnittelua siten, että lopputuloksena on tietotekniikkaan liittyvä artefakti. Artefaktin ei välttämättä tarvitse olla kokonainen käytännössä toimiva ohjelma, vaan esimerkiksi ohjelmasta löytyvää metodia voidaan pitää artefaktina. (Hevner ym. 2004)

Toisessa ohjeessa neuvotaan tekemään tutkimusta kohdistuen oikeisiin liiketoimintaan liittyviin ongelmiin, joilla on merkitystä. Suunnittelutieteen avulla näihin ongelmiin voidaan löytää käytännön ratkaisu artefaktin muodossa. Tähän liittyen huomioidaan, että yritykset ovat liikevoittoa tuottavia, joten yritysten tutkimat ongelmat liittyvät yleensä liiketoiminnallisten prosessien parantamiseen, jonka seurauksena kustannukset pienenevät tai voitot nousevat. (Hevner ym. 2004)

Kolmantena ohjeistetaan tutkimuksen aikana syntyneen artefaktin arvioinnista. Tieteellinen tarkkuus on tärkeää arvioinnissa, jonka pitäisi käsitellä artefaktin hyödyllisyyttä, laatua ja tehokkuutta. Arviointi on tärkeää erityisesti sen takia, koska suunnittelutieteellisessä tutkimuksessa tehdään tutkimusta iteratiiviseen ja inkrementaaliseen tyyliin, joten arviointi antaa tärkeää palautetta suunnittelun toimivuudesta. Artefaktin arviointiin käytetään tapauskohtai-

sesti erilaisia arviointimenetelmiä, jotka tyypillisesti ovat jo tunnettuja. (Hevner ym. 2004)

Suunnittelun arviointimenetelmät voidaan jakaa viiteen eri luokkaan, jotka ovat tarkkailevat, analyyttiset, kokeelliset, testaavat ja kuvaavat. Tarkkaileviin arviointimenetelmiin kuuluu tapaustutkimus ja kenttätutkimus. Tapaustutkimuksessa yhtä artefaktia tarkkaillaan tarkasti liiketoimintaympäristössä. Kenttätutkimuksessa artefakti on käytössä useissa projekteissa ja artefaktin käyttöä tarkkaillaan. (Hevner ym. 2004)

Analyyttisiin arviointimenetelmiin kuuluu arkkitehtuurianalyysi, dynaaminen analyysi, staattinen analyysi ja optimointi. Arkkitehtuurianalyysissä tutkitaan kuinka hyvin luotu artefakti ja käytössä oleva tekninen tietojärjestelmäarkkitehtuuri sopivat yhteen. Dynaamisella analyysillä tarkoitetaan muuttuvien määreiden mittaamista, kun artefakti on käytössä. Tällainen määre voi olla esimerkiksi suorituskyky. Staattisessa analyysissä tarkastellaan artefaktin rakennetta tarkastellen staattisia määreitä. Tällainen määre voi olla esimerkiksi kompleksisuus. Optimoinnilla tarkoitetaan tässä yhteydessä artefaktin käyttäytymisen optimaalisten rajojen selvittämistä tai artefaktissa olevien ominaisuuksien optimaalisuuden demonstroimista. (Hevner ym. 2004)

Kokeellisiin arviointimenetelmiin kuuluu kontrolloitu koe ja simulaatio. Kontrolloidussa kokeessa tutkitaan erilaisia määreitä, kun artefakti on jossain kontrolloidussa ympäristössä. Määreenä voi olla esimerkiksi käytettävyys. Simulaatiossa hankitaan tai luodaan keinotekoista dataa, jota käytetään artefaktin suorittamiseen. (Hevner ym. 2004)

Testaaviin arviointimenetelmiin kuuluu musta- ja valkolaatikkotestaus. Mustalaatikkotestaus on funktionaalista testaamista, jossa artefaktin rajapintoja suoritetaan. Tämän avulla artefaktista selviää vikoja. Valkolaatikkotestaus on rakenteellista testaamista, jossa tarkastellaan artefaktin toteutusta jonkin kattavuusmääreen kannalta. Tällainen määre voi olla esimerkiksi ohjelman suorituspolkujen määrä. (Hevner ym. 2004)

Kuvaaviin arviointimenetelmiin kuuluu skenaarioiden tarkastelu ja valistunut argumentti. Skenaarioiden tarkastelussa rakennetaan yksi tai useampi yksityiskohtainen skenaario, jossa artefaktin hyödyllisyyttä tuodaan käytännössä esille. Valistuneen argumentin tapauksessa käytetään jo olemassa olevaa tietoa muodostamaan argumentteja artefaktin hyödyllisyyden puolesta. Argumenttia rakentaessa voi käyttää hyödyksi esimerkiksi aiemmin tehtyjä tutki-

muksia. (Hevner ym. 2004)

Neljäs ohje suunnittelutieteellisen tutkimuksen tekemiseen on, että suunnittelutieteellisessä tutkimuksessa tutkimustulosten pitää tarjota uutta tietoa. Suunnittelutieteellisen tutkimuksen on mahdollista tehdä uusia löydöksiä kolmelta eri alueelta: artefaktista, perustuksista ja metodiikasta. Artefaktiin liittyvillä tutkimustuloksilla tarkoitetaan tilannetta, jossa artefakti ratkaisee jonkun aiemmin ratkaisemattoman ongelman. Perustuksilla tarkoitetaan tilannetta, jossa artefaktin kehityksen seurauksena syntyy uutta tietoa yleisesti käytettäväksi tutkimuksen seuraaviin iteraatioihin tai uusiin tutkimuksiin. Metodiikalla tarkoitetaan iteraatiossa syntyneen artefaktin arvioinnissa käytettävien uusien arviointimittareiden keksimistä tai arviointimenetelmien kehittymistä. (Hevner ym. 2004)

Viides ohje on, että tutkimuksen pitäisi olla suoritettu tieteellisellä tarkkuudella. Tämä pätee sekä artefaktin rakentamiseen että sen arviointiin. Suunnittelutieteessä tieteellinen tarkkuus perustuu aiemman tutkimustiedon vaikuttavaan hyödyntämiseen. Artefaktien arvioinnissa käytettyjen mittareiden järkevyyttä pitäisi seurata ja artefakteja pitäisi testata artefaktien kannalta järkevässä ympäristössä. (Hevner ym. 2004)

Kuudes ohje on, että tutkimuksessa olevan suunnittelun pitäisi olla iteratiivinen prosessi, jossa etsitään paras ratkaisu tutkittavaan ongelmaan. Suunnittelutieteessä on tyypillistä, että yksi ongelma jaetaan pienempiin ongelmiin ja keskitytään aluksi jonkun pienen ongelman ratkaisemiseen. Täten seuraavissa iteraatioissa voidaan ratkaista vielä ratkaisemattomia ongelmia. (Hevner ym. 2004)

Seitsemäs ohje on, että suunnittelutieteellisen tutkimuksen tulee olla esitettävissä teknisistä asioista ymmärtävien lisäksi myös henkilöille, joiden osaamisen painopiste on hallinnossa. Teknisistä asioista ymmärtävillä pitää olla riittävästi tietoa, että artefakti voidaan ottaa käyttöön jossain toisessa ympäristössä kuin mihin artefakti oli alun perin tehty. Lisäksi riittävän yksityiskohtainen kuvaus tutkimuksesta mahdollistaa ulkopuolelta tulevan arvioinnin ja tutkimuksen jatkamisen. Hallinto tarvitsee tutkimuksesta tietoa päätöksenteon tueksi kysymykseen, onko järkevää resurssoida artefaktin implementointi oman organisaation ympäristöön. (Hevner ym. 2004)

2.2 Suunnittelutiede tässä tutkimuksessa

Tässä tutkimuksessa sovelletaan suunnittelutieteen ideaa kaiutinsovelluksen kehittämiseen. Jokaisessa iteraatiossa kaiutinsovelluksen äänen toistamiseen käytettävää äänipuskuria pienennetään mikäli mahdollista ja tämän jälkeen suoritetaan äänen viiveen mittaus. Mittauksen avulla sovelluksen kehittäjä pysyy tietoisena sovelluksen äänen viiveestä jokaisessa iteraatiossa ja pystyy arvioimaan iteraatiossa tehtyjen muutosten vaikutusta äänen viiveeseen.

Äänen viiveen mittaus voi vaikuttaa seuraavassa iteraatiossa tehtäviin muutoksiin. Kehittäjän on käytettävä omaa harkintaa ja katsottava sovelluksen kehitystä kokonaisuutena. Esimerkiksi mikäli kehittäjä näkee järkeväksi kirjoittaa jonkin osuuden ohjelmasta kokonaan uusiksi, niin vanhaan koodiin ei kannata tehdä esimerkiksi äänen viiveeseen liittyviä optimointeja, erityisesti mikäli ne joutuisi tekemään uudelleen uuteen koodiin. Kehittäjällä on täten valta viivästyttää äänen viiveen mittauksen perusteella syntyviä ja järkeviä parannusehdotuksia ohjelmaan.

Seuraavaksi tarkastellaan kuinka hyvin tämä tutkimus täyttää Hevnerin ym. (2004) ohjeet suunnittelutieteelliselle tutkimukselle. Ensinnäkin heidän artikkeli käsittelee suunnittelutiedettä tietojärjestelmien kontekstissa, niin jo tältä osin vertailu voidaan suorittaa, koska tässä tutkimuksessa tehdään tietotekniikkaan liittyviä artefakteja. Heidän ensimmäisessä ohjeessa kerrottiin, että tutkimuksen lopputuloksena pitäisi olla tietotekniikkaan liittyvä artefakti. Tämä tutkimus on tämän ohjeen mukainen, koska tässä tutkimuksessa tehdään jokaiseen iteraatioon uusi versio kaiutinsovelluksesta.

Hevnerin ym. (2004) toisessa ohjeessa kerrottiin, että tutkimuksen pitäisi ratkaista oikeita liiketoimintaan liittyviä ja merkittäviä ongelmia. Kaiutinsovelluksella voinee olla kysyntää markkinoilla. Tutkimuksen merkittävyyttä ei tässä tutkimuksessa käsitellä, eli vertailua muihin käytännössä saman asian tekeviin tuotteisiin ei tehdä. Tämä tutkimus ei täten ole täysin tämän ohjeen mukainen.

Hevnerin ym. (2004) kolmannessa ohjeessa käsiteltiin artefaktin arviointia. Heidän listaamista arviointimenetelmistä käytetään tässä tutkimuksessa ainakin dynaamista analyysiä ja valistunutta argumenttia. Äänen viiveen mittaukset tehdään ohjelman ollessa toiminnassa ja äänen viive on sovelluksen ajonaikainen suure, johon vaikuttaa sovelluksen toimintaympä-

ristö. Valistuneita argumentteja käytetään järkevien kehitysehdotusten määrittämisessä artefaktille. Argumenttien perusteluissa käytetään joissain määrin viittauksia aiempaan tutkimukseen tai teknisiin dokumentteihin.

Kaiutinsovelluksen iteraatioiden arvioinnissa olisi myös mahdollista käyttää arviointimenetelmänä Hevnerin ym. (2004) listaamaa kontrolloitua koetta. Tällöin sovellusta käytettäisiin oikeassa ympäristössä, eli käyttäjät arvioisivat sovelluksen äänen viivettä. Käyttäjien arviot voivat olla subjektiivisia ja tarkempia arvioita äänen viiveestä on mahdollista tehdä, niin tässä tutkimuksessa käytetään mittauslaitetta apuna mittauksissa.

Artefakteja olisi periaatteessa mahdollista arvioida äänen viiveen teoreettisella minimillä. Se olisi luultavasti mahdollista laskea mikäli sovelluksen ja sen ympäristöön liittyvät lähdekoodit (muun muassa laiteajurien) sekä laitteiden tarkat tekniset tiedot olisivat käytettävissä. Prosessi olisi luultavasti työläs ja sillä ei välttämättä saisi tietoa kuinka suuri äänen viive on käytännössä. Ohjelman käyttäjillä on todennäköisesti muitakin sovelluksia tietokoneella tai puhelimella käynnissä ja ne voivat vaikuttaa kaiutinsovelluksen äänen viiveeseen. Tällainen tarkastelu on luultavasti staattista analyysiä, jonka Hevner ym. (2004) ovat listanneet yhdeksi arviointimenetelmäksi.

Hevnerin ym. (2004) listaamista arviointimenetelmistä dynaamista analyysiä olisi luultavasti myös mahdollista käyttää sovelluksen komponenttien testaamiseen suorituskyvyn osalta. Esimerkiksi voitaisiin tarkastella kuinka nopeasti jokin sovelluksen komponentti suorittaa jonkun tehtävän. Mikäli yksi tehtävä ei ole riittävän suuri tai järkevä kohde testattavaksi, niin tällöin voitaisiin tarkastella useamman eri tehtävän ketjun suorittamisen nopeutta.

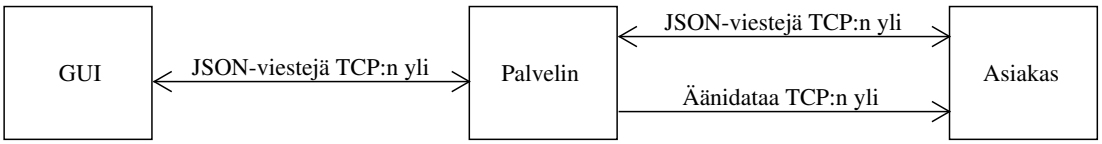
Hevnerin ym. (2004) neljännessä ohjeessa käsiteltiin uuden tiedon muodostamista kolmelta eri alueelta: artefaktista, perustuksista ja metodiikasta. Artefaktin osalta muihin suunnilleen vastaaviin sovelluksiin ei tehdä vertailua, niin artefaktin osalta uuden tiedon syntymistä ei luultavasti tapahdu. Perustuksiin ja metodiikkaan liittyvistä asioista voi mahdollisesti löytyä jotain uutta esimerkiksi sen muodossa, että miten asioita ei kannattaisi tehdä.

Hevnerin ym. (2004) viidennessä ohjeessa sanotaan, että suunnittelutieteessä kuuluisi käyttää tieteellistä tarkkuutta. Tässä tutkimuksessa tämä voi olla joissain valistuneissa argumenteissa niin tai näin, mikäli niissä ei ole viittausta mihinkään lähteeseen.

Hevnerin ym. (2004) kuudennessa ohjeessa käsitellään tutkimuksessa olevan suunnittelun iteratiivisuutta. Tässä tutkimuksessa tehdään useampi versio kaiutinsovelluksesta arvioiden jokaista versiota, niin tässä tutkimuksessa käytetään suunnitteluun iteratiivista prosessia.

3 Kaiutinsovelluksen rakenne

Tässä luvussa käsitellään kaiutinsovelluksen rakennetta tutkimuksen lähtötilanteessa.



Kuvio 1. Kaiutinsovelluksen rakenne korkealla tasolla

Kaiutinsovellus koostuu kolmesta eri osuudesta: palvelimenhallintaohjelmasta, palvelinohjelmasta ja asiakasohjelmasta. Ensimmäiset kaksi ovat kirjoitettu Rust-ohjelmointikielellä ja toimivat Ubuntulla. Molemmat ohjelmat ovat samassa binäärissä, eli binääriä käynnistettäessä pitää valita kumman ohjelman käynnistää. Palvelimenhallintaohjelmaa käytetään palvelinohjelman hallitsemiseen TCP:n yli lähetettävillä JSON-viesteillä. Tämä ominaisuus on vielä suunnitteluasteella, kun järkeviä JSON-viestejä ei ole toteutettu.

Palvelinohjelma muun muassa nauhoittaa ääntä ja lähettää sitä Androidilla toimivalle asiakasohjelmalle, joka on kirjoitettu Kotlin- ja Rust-ohjelmointikielillä. Asiakasohjelma voi luoda yhteyden palvelimeen, ja ne keskustelevat keskenään TCP:n yli lähetettävillä JSON-viesteillä. Äänen siirto asiakkaalle käynnistyy, mikäli asiakas yhdistää äänidatalle tarkoitettuun TCP-porttiin palvelimen lähettämän JSON-viestin jälkeen.

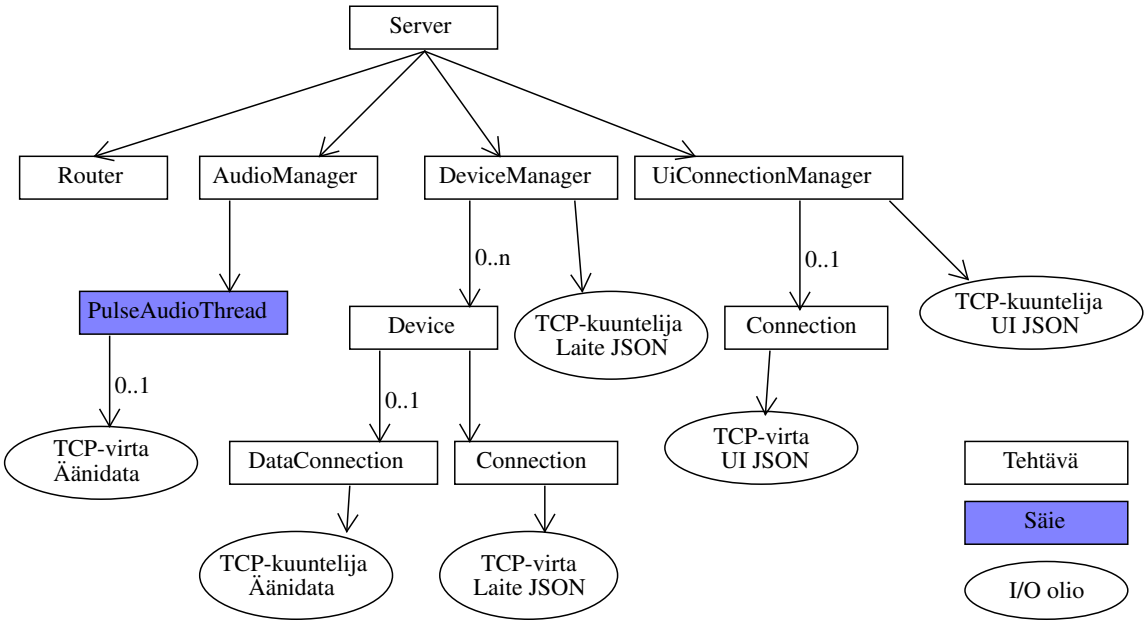
Palvelimen ja asiakkaan välinen TCP-viestintä on tarkoitettu kulkemaan lähiverkossa tai USB-yhteydellä. Lähiverkon tapauksessa tiedonsiirto tapahtuu luultavasti yleensä ainakin osan matkasta WLAN-yhteyden kautta, koska puhelimet yhdistetään lähiverkkoon yleensä WLAN-yhteydellä. USB-yhteyttä on mahdollista käyttää Android Debug Bridge (adb) -työkalun avulla.

Palvelimen ja asiakkaan lähdekoodit ovat saatavilla GitHub-palvelussa palvelinohjelman osalta osoitteessa `https://github.com/jutuon/jonect` ja asiakasohjelman osalta osoitteessa `https://github.com/jutuon/jonect-android`.

3.1 Palvelinohjelman rakenne

Rust-ohjelmointikieli on muistiturvallinen ja kääntyy natiiviksi koodiksi (Yegulalp 2021). Näiden lisäksi kaiutinsovelluksen palvelinohjelma päätettiin kirjoittaa Rustilla, koska siinä ei ole roskienkeruulla toteutettua muistinhallintaa ja kaiutinsovelluksen ohjelmoijalla oli Rust-ohjelmoinnista jo aiempaa kokemusta. Aiempi Rust-ohjelmointikokemus autttoi kaiutinsovelluksen palvelinohjelman kehittämistä.

Palvelinohjelmaa päätettiin alkaa kehittää käyttäen Rustin asynkronisen ohjelmoinnin ominaisuuksia, eli lähdekoodissa käytettäisiin async- ja await-syntaksia jonkin verran. Asynkronisen koodin suorittaminen tehtiin Tokio-kirjastolla. Ohjelman rakenne muotoutui kehityksen aikana siten, että ohjelman logiikassa selkeästi eroteltavista osuuksista tehtiin erillisiä komponentteja, joita suoritetaan Tokio-kirjaston hallinnoimina tehtävinä tai tarvittaessa normaaleina säikeinä.



Kuvio 2. Palvelinohjelman komponentit lähtötilanteessa. Kaavio on tehty suurin piirtein, joten esimerkiksi komponenttien nimet eivät välttämättä vastaa ohjelmakoodissa olevia.

Kuviossa 2 nähdään palvelinohjelman erilaisten komponenttien looginen sijainti ohjelmassa. Palvelinohjelman perusrakenteeseen kuuluvat Server- ja Router-komponentit. Server-komponentti käynnistää muut komponentit ja pitää huolen, että ohjelma sammuu hallitusti,

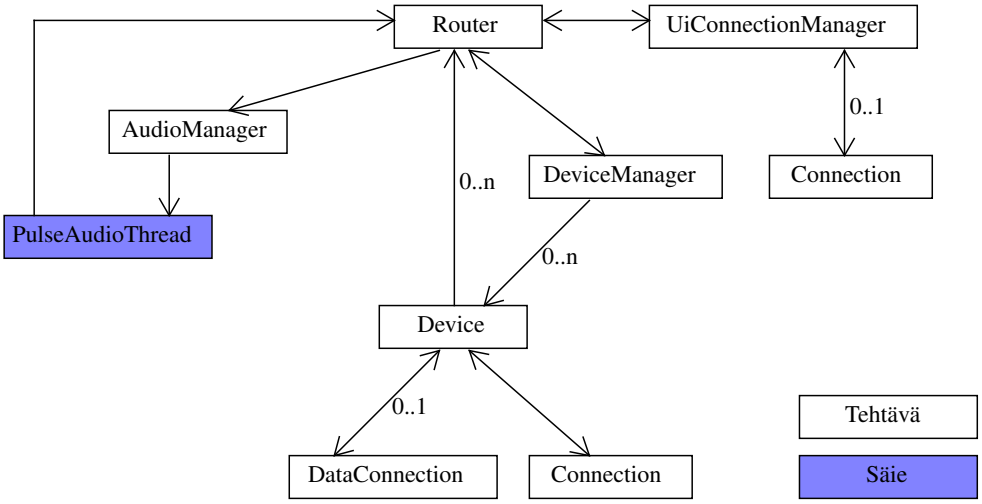
eli ohjelma sammutetaan vasta sitten, kun kaikki sen komponentit ovat sammutettu. Router-komponenttia käytetään helpottamaan ohjelman eri komponenttien välistä viestintää. Eri komponenteilla on käytössä omat viestityyppinsä, joille tarvitaan erilliset viestikanavat. Yhden komponentin lähettäessä useammalle kuin yhdelle komponentille viestejä tarvittaisiin viestejä lähettävään komponenttiin useita kahvoja eri viestikanaviin. Router-komponentilla on kaikkien aina olemassa olevien komponenttien viestikahvat, joten tällöin yksittäiset komponentit tarvitsevat ainoastaan kahvan Router-komponentin viestikanavaan.

AudioManager-komponentti tekee rajapinnan äänen nauhoittamista varten. Tällöin ohjelmaan on mahdollista tehdä erilaisia rajapintatoteutuksia, mikäli palvelinohjelma halutaan toimivan esimerkiksi Windows-käyttöjärjestelmillä. Palvelinohjelma tehtiin käytettäväksi Ubuntulla, niin AudioManagerin rajapinta on toteutettu PulseAudion avulla. Rustille tehty PulseAudio-kirjasto ei tue Tokio-kirjastoa, niin ääntä nauhoittava koodi ajetaan omassa säikeessänsä.

DeviceManager-komponentti kuuntelee tiettyä TCP-porttia ja luo uuden Device-komponentin asiakasohjelman yhdistäessä palvelimeen. Yhteen Device-komponenttiin liittyvä Connection-komponentti hallinnoi asiakasohjelmaan liittyvien viestien lähetystä ja vastaanottamista. Yksi äänivirran vastaanottamiseen käytettävä DataConnection-komponentti käynnistetään mikäli asiakasohjelma lähettää tietoja itsestään.

Palvelinohjelmaa on tulevaisuudessa tarkoitus hallita käyttöliittymän kautta, josta pystyy esimerkiksi sulkemaan yhteydet asiakasohjelmiin. UiConnectionManager-komponentti on tehty käyttöliittymätoiminnallisuutta varten. Käyttöliittymäohjelman on tarkoitus yhdistää palvelinohjelmaan TCP-yhteydellä. Tämän jälkeen käyttöliittymä ja palvelinohjelma vaihtelevat JSON-viestejä kyseisen TCP-yhteyden yli. Myös UiConnectionManager-komponentilla on Connection-komponentti vastaavasti kuin Device-komponentilla.

Kuviossa 3 esitetään kuinka eri komponentit lähettävät toisilleen viestejä. Kuten kuviosta nähdään, niin Router-komponentti on keskeisessä osassa viestien välittäjänä. Viestien vaihtaminen eri komponenttien välillä voi olla joko yksi- tai kaksisuuntaista.



Kuvio 3. Palvelinohjelman komponenttien kommunikointi lähtötilanteessa. Kaavio on tehty suurin piirtein, joten esimerkiksi komponenttien nimet eivät välttämättä vastaa ohjelmakoodissa olevia.

3.2 Äänen nauhoittaminen

Palvelinohjelma käyttää PulseAudiota äänen nauhoittamiseen. Äänen nauhoittaminen tapahtuu omassa säikeessänsä. Kaiutinsovelluksessa voi käyttää pakkaamatonta PCM-formaattia tai pakattua Opus-formaattia. Opus-formaatti aiheuttaa minimissään viiden millisekunnin viiveen (Valin ym. 2016). Kyseisen viiveen takia tässä tutkimuksessa käytetään vain PCM-formaattia.

Palvelin nauhoittaa PulseAudiolla PCM-formaatissa olevia 16-bittisiä ja little-endian -tavujärjestyksellä olevia näytteitä. Nauhoituksen näytteistystaajuutena käytetään Android-laitteen näytteistystaajuutta. Nauhoituksessa on kaksi äänikanava, eli nauhoitus on stereoääntä. PulseAudion nauhoitusasetuksista on asetettu uuden äänidatan lähettäminen palvelimelle tapahtumaan 64:n näytteen välein. Tämä aiheuttaa esimerkiksi 48 000 Hz näytteistystaajuudella ja kahdella äänikanavalla noin 0,67 millisekunnin viiveen.

Asiakas on kertonut palvelimelle Android-laitteen näytteistystaajuuden. Edellinen tapahtuu siksi, että tällöin äänidataa ei tarvitse mahdollisesti muokata vastaamaan Android-laitteen näytteistystaajuutta itse Android-laitteessa. Tietokoneessa on todennäköisesti enemmän laskentakapasiteettia, niin mahdollisesta muunnoksesta johtuva viive on pienempi tietokoneella

suoritettuna.

Seuraavassa kuvataan suurin piirtein ja vaiheittain, miten ääni lähetetään palvelimelta asiakkaalle:

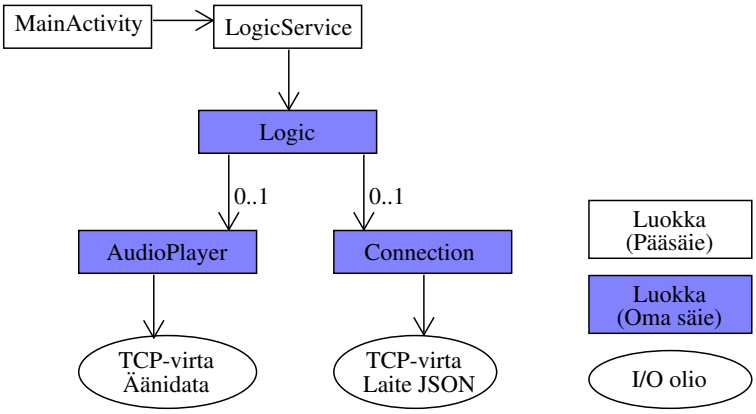
1. Uuden äänidatan ollessa valmiina palvelinohjelman käsiteltäväksi kutsutaan takaisinkutsufunktiota, joka lisää tapahtumankäsittelyjonoon tiedon hakea äänidataa.
2. Säikeen tapahtumankäsittely etenee edellisessä kohdassa lisättyyn tapahtumaan.
3. Yritetään hakea viite äänidataa sisältävään taulukkoon. Mikäli äänidataa ei ole niin siirrytään kohtaan 1.
4. Mikäli on olemassa asiakkaalle aiemmin lähettämätöntä äänidataa, niin yritetään kirjoittaa se TCP-sokettiin.
5. Yritetään lähettää kohdassa 3 hankittu äänidata asiakkaalle kirjoittamalla se TCP-sokettiin. Kirjoittamaton äänidata tallennetaan odottamaan seuraavaa iteraatiota.
6. Palataan kohtaan 1.

Edellisen algoritmin yksityiskohdat löytyvät lukemalla palvelinohjelman Git-varastosta lähdekkoodia commitista `c4724ed` ja tiedostosta `src/server/audio/pulseaudio/stream.rs`.

3.3 Asiakasohjelman rakenne

Asiakasohjelma on kirjoitettu Kotlin- ja Rust-ohjelmointikielelillä. Suurin osa ohjelmasta on kirjoitettu Kotlinilla. Rustia on käytetty Opus-äänivirtaa dekoodaavassa komponentissa. Korkealla tasolla asiakasohjelman rakenne muistuttaa palvelinohjelmaa, koska ohjelmalogiikka on jaettu erillisiin komponentteihin, jotka keskustelevat toistensa kanssa erilaisilla viesteillä ja sijaitsevat omissa säikeissään. Asiakasohjelman Opus-äänivirtaa dekoodaavaa komponenttia ei käsitellä tässä osiossa, koska tutkimuksessa äänidata on PCM-muodossa.

Kuviossa 4 näkyy kaiutinsovelluksen asiakasohjelman oleellisimmat komponentit. MainActivity-komponentti on aktiviteetti (engl. *Activity*), joka sisältää käyttöliittymän palvelimeen yhdistämistä varten. Aktiviteetti on Android-ohjelmoinnissa sovelluksen yksi näkymä, jossa käyttäjä voi käyttää sovelluksen kyseiseen näkymään liittyviä toimintoja (Google 2023).



Kuvio 4. Asiakasohjelman komponentit lähtötilanteessa. Kaavio on tehty suurin piirtein, joten esimerkiksi komponenttien nimet eivät välttämättä vastaa ohjelmakoodissa olevia. Ohjelmassa on muitakin luokkia kuin kuvassa esiintyvät luokat.

Toiminnallisuudet käyttöliittymässä ovat seuraavat: tekstikenttä IP-osoitetta varten, painike palvelimeen yhdistämistä varten, tekstikomponentti tilatietojen näyttämistä varten ja asetus automaattista palvelimeen yhdistämistä varten asiakasohjelman käynnistyessä. Viimeisin IP-osoite ja edellä mainitun asetuksen totuusarvo tallennetaan Androidin SharedPreferences-rajapinnalla. Käyttöliittymässä ei ole kovin paljon toiminnallisuutta, joten koko käyttöliittymä mahtuu kohtuullisen hyvin yhteen komponenttiin. MainActivity on sovelluksen ainoa aktiviteetti.

Käyttöliittymän rinnalla, ja myös ohjelman pääsäikeessä, toimii LogicService-komponentti, joka on palvelu (Service). Android-ohjelmoinnissa palvelu on komponentti, jolla voi esimerkiksi soittaa musiikkia vaikka sovelluksen käyttöliittymä ei olisi näkyvissä (Google 2023). LogicService-komponentti käynnistää uuden säikeen Logic-komponentille, joka hallinnoi ohjelmalogiikan AudioPlayer- ja Connection-komponenttien käynnistymistä. Logic-komponentti käynnistää edellä mainitut komponentit tarvittaessa. Connection-komponentti käynnistetään, mikäli käyttäjä antaa käyttöliittymästä komennon yhdistää palvelimelle tai vaihtoehtoisesti heti asiakasohjelman käynnistyessä, mikäli tähän liittyvä asetus on päällä. AudioPlayer-komponentti käynnistetään mikäli palvelinohjelma antaa pyynnön alkaa soittamaan palvelimen tarjoamaa äänivirtaa.

Connection-komponentti toimii omassa säikeessänsä ja lähettää sekä vastaanottaa palveli-

nohjelman tukemia JSON-viestejä. Myös AudioPlayer-komponentti toimii omassa säikeessänsä, mutta sen tehtävä on eri kuin Connection-komponentilla. AudioPlayer-komponentti yhdistää palvelinohjelman TCP-porttiin, josta äänivirtaa voidaan vastaanottaa.

Eri käyttötarkoituksesta huolimatta AudioPlayer- ja Connection-komponentit ovat toteutettu samalla tyylillä viestinvälityksen ja TCP-soketin käyttämisen osalta. Kyseisten komponenttien viestinvälityksessä on kaksi erillistä osaa: komponentin sammutuskäsky ja muiden viestien lähettäminen. AudioPlayer-komponentti käyttää käytännössä vain ensimmäistä, koska asiakasohjelman äänentoiston toteuttaminen ei vaatinut muiden viestien toteuttamista. Sammutuskäskyn lähettäminen on toteutettu Pipe-luokalla, joka löytyy Javan ei-sulkevan (engl. *non-bloking*) I/O:n tarjoavasta nio-paketista. Pipe-luokka on rajapinta käyttöjärjestelmän putki-objektille, jonka kautta voidaan lähettää dataa. Logic-komponentti lähettää sulkemispyynnön komponentille kirjoittamalla yhden tavun putki-objektiin.

Putki-objekti on käytössä myös muiden viestien lähettämisessä, mutta vain merkkinä tarkistaa uusi viesti erillisestä puskurista, jota on turvallista käyttää eri säikeiden välillä. Erillistä puskuria käytetään sen takia, että olioina olevia viestejä ei tarvitse muuttaa tavuiksi tai toisin päin.

AudioPlayer- ja Connection-komponentit käyttävät TCP-sokettia Javan nio-paketista löytyvällä SocketChannel-luokalla, joka tarjoaa rajapinnan käyttöjärjestelmän soketti-objektille. Javan nio-paketista löytyvien muiden luokkien avulla on mahdollista kuunnella molempien objektien tapahtumia yhtä aikaa, eli säikeen suoritus voidaan pysäyttää siksi aikaa, kunnes joko sokettiin voi kirjoittaa tai sitä voi lukea tai vaihtoehtoisesti putki-objektia voi lukea.

3.4 Äänen toistaminen

Asiakasohjelman AudioPlayer-komponentti käyttää Androidin AudioTrack-luokkaa äänen toistamiseen. Äänen toistaminen tapahtuu suurin piirtein seuraavasti:

1. Vaihdetaan äänen toistamiseen käytettävän säikeen prioriteetiksi vakio `Process.THREAD_PRIORITY_AUDIO`.
2. Luodaan AudioTrack-luokan olio asetuksilla, jotka vastaavat palvelimen lähettämän

äänidatan ominaisuuksia. Mikäli äänidatan näytteistystaajuus vastaa Android-laitteen natiivia näytteistystaajuutta, niin asetetaan `AudioTrack.PERFORMANCE_MODE_LOW_LATENCY`-asetus käyttöön.

3. Yhdistetään palvelimen TCP-porttiin, joka on tarkoitettu äänidatalle.
4. Luetaan TCP-soketista 32 tavun verran äänidataa.
5. Kirjoitetaan äänidata AudioTrack-olion äänipuskuriin.
6. Mikäli edellinen kohta on suoritettu vähintään neljä kertaa ja AudioTrack-olio ei vielä toista äänidataa, niin kutsutaan AudioTrack-olion play-metodia.
7. Palataan kohtaan 4.

Edellisen algoritmin yksityiskohdat löytyvät lukemalla asiakasohjelman Git-varastosta lähdekoodia commitista `4a2441c` ja tiedostosta `app/src/main/java/com/example/jonect/Audio.kt`.

4 Kaiutinsovellus ja äänen viive

Tässä luvussa käsitellään, mitkä asiat mahdollisesti aiheuttavat äänen viivettä kaiutinsovelluksessa. Sovelluksessa oleva äänen viive koostuu kolmesta komponentista: äänen nauhoituksesta, siirtämisestä ja toistamisesta. Nauhoittaminen tapahtuu palvelinohjelmassa ja siirtäminen TCP-soketilla palvelinohjelmassa sekä asiakasohjelmassa. Toistaminen tapahtuu asiakasohjelmassa.

4.1 Äänipuskurien koko

Balsini ym. (2019) paransivat reaaliaikaisen äänenprosessoinnin suorituskykyä Androidilla. He kirjoittavat, että isot äänipuskurit ja useat abstraktiot ohjelmassa lisäävät äänen viivettä. Luvussa 3 olleista algoritmeista voidaan havaita, että äänidataa luetaan ja kirjoitetaan muistialueesta toiseen sekä palvelinohjelmassa että asiakasohjelmassa.

Nauhoittamiseen käytetyn äänipuskurin kokona voi pitää aliluvussa 3.2 kerrottua PulseAudion äänidatan lähetystiheyttä. Palvelinohjelman toisena äänipuskurina voi pitää TCP-sokettia ja se kuuluu siirtämisen komponenttiin. Palvelinohjelma voi myös mahdollisesti varastoida äänidataa, mikäli TCP-sokettiin ei mahdu kirjoittamaan mitään. Ohjelman kehityksessä syntyneiden kokemusten perusteella äänidatan varastointia ei tapahdu, mikäli asiakasohjelma vastaanottaa ja toistaa ääntä normaalisti.

Androidilla toimiva asiakasohjelma käyttää Androidin Java-rajapintaan kuuluvaa AudioTrack-luokkaa äänen toistamiseen. Balsini ym. (2019) sanovat, että interaktiivisia ja pienen viiveen tilanteita varten sovelluksissa pitäisi käyttää pienen viiveen C/C++-äänirajapintoja C/C++-komponentista. He perustelevat tätä siten, että Androidin Java-rajapintaan kuuluvat ja ääntä toistavat luokat pakottavat käyttämään isoja äänipuskureita. Tämä asia liittyy toistamisen komponenttiin.

4.2 Tiedonsiirto ja TCP-protokolla

Goelin, Krasicin ja Walpolen (2008) mukaan TCP-protokolla on suosittu median suoratoistoon, koska yleisesti TCP:n ruuhkanhallinta aiheuttaa internetiin vakautta sen toimivuuden suhteen. Lisäksi he sanovat, että median suoratoiston kannalta on hyvä, että TCP tekee tarvittavat toimenpiteet pakettien häviämisen tapahtuessa. Tämän takia heidän mukaan sovelluksen ei tarvitse itse kysyä palvelimelta median kadonneen osan uudelleenlähetystä.

TCP:n ongelma median suoratoistamisessa on Goelin, Krasicin ja Walpolen (2008) mukaan siitä mahdollisesti aiheutuvat viiveet sovellustasolla. Heidän tekemässä tutkimuksessa he tutkivat mistä viive johtuu ja miten sitä voi pienentää.

Goel, Krasic ja Walpole (2008) jaottelevat suoratoistoon liittyvässä sovelluksessa esiintyvän suoratoistoviiveen kahteen luokkaan: sovelluksesta aiheutuvaan viiveeseen ja protokollasta aiheutuvaan viiveeseen. Käytännössä he määrittelevät protokollasta aiheutuvan viiveen siten, että kuinka kauan datan siirrossa kestää lähettävästä soketista vastaanottavaan sokettiin. Heidän mukaan TCP aiheuttaa viivettä lähettäjän puolella tapahtuvan puskuroinnin, ruuhkanhallinnan ja pakettien uudelleenlähetyksen takia.

TCP kuuluu ikkunapohjaisten protokollien joukkoon. TCP:n tekemä pakettien lähettäminen toimii pitämällä kirjaa lähetetyistä paketeista, joille ei ole vielä saapunut vahvistusviestiä (ACK). Tällaisten pakettien lukumäärää nimitetään ikkunakooksi ja siitä pidetään kirjaa CWND-nimisessä muuttujassa. Kun nykyisen ikkunan ensimmäiselle paketille saapuu vahvistusviesti, niin lähetetään uusi paketti, ja uuden paketin data pidetään lähetyspuskurissa tallessa mahdollisen uudelleenlähetyksen varalta. CWND-muuttujan arvo pidetään TCP-yhteyden alussa pienenä ja sitä kasvatetaan kaistanleveyden selvittämiseksi. Mikäli matkalla ollut paketti häviää, niin tämä on ilmoitus kaistanleveyden rajoista TCP-yhteydelle. TCP-yhteyden tiedonsiirtonopeutena voi suurin piirtein pitää CWND-lukua jaettuna paketin edestakaisen kulkumatkan nopeudella (engl. *round-trip time, RTT*). (Goel, Krasic ja Walpole 2008)

Goel, Krasic ja Walpole (2008) tarkoittavat ilmeisesti ikkunakoolla TCP:n ruuhkanhallintaikkunan (engl. *congestion window, CWND*) kokoa. Blanton, Paxson ja Allman (2009) määrittelevät ruuhkanhallintaikkunan TCP:n tilamuuttujaksi, joka määrittää datan lähettä-

misen rajat TCP:lle. He huomioivat, että CWND-muuttujassa käytettävä yksikkö on TCP-implementaatiokohtainen. Heidän mukaan joissain implementaatioissa yksikkönä on tavujen lukumäärä ja toisissa täysikokoisten segmenttien lukumäärä. He määrittelevät segmenteiksi minkä tahansa TCP/IP kuittaus- tai datapaketin.

Pakettien uudelleenlähetyksestä aiheutuva viive voi Goelin, Krasicin ja Walpolen (2008) mukaan olla vähintään yhden RTT:n verran tai mikäli paketti häviää useasti, niin tällöin kyseessä on monikerta RTT-ajasta. Lisäksi he huomioivat, että TCP:ssä data kulkee järjestyksessä, niin kadonnut paketti hidastaa myös muiden pakettien vastaanottamista sovellusohjelman puolella.

Ruuhkanhallinnasta aiheutuva viive liittyy siihen, että hävinnyt paketti aiheuttaa RTT-ajan mittaisen viiveen matkalla oleviin paketteihin. Paketin häviäminen antaa TCP-yhteyden ruuhkanhallinnalle merkin pienentää CWND-muuttujan arvoa, josta seuraa se, että TCP:n lähetyspuskurissa ja CWND-pakettimäärästä ylimenevät paketit lähetetään myöhemmin kuin ennen ruuhkanhallinnan merkkiä. (Goel, Krasic ja Walpole 2008)

Pakettien häviäminen ei ole ainut mahdollisuus antaa TCP-yhteydelle ruuhkanhallintamerkkiä. Eksplisiittisen ruuhkamerkin (engl. *explicit congestion notification, ECN*) avulla TCP:n ruuhkanhallinta saa viestin ruuhkasta ennen kuin paketti häviää. Tällä tyylillä reitittimet viestivät ruuhkasta asettamalla ECN-bitin päälle niihin TCP-paketteihin, jotka olisivat muuten pudonneet. Pakettien vastaanottaja tarkkailee kyseistä bittiä ja tieto bitin päällä olemisesta lähtee pakettien lähettäjälle ACK-paketilla. (Goel, Krasic ja Walpole 2008)

Laitteiden ja reitittimien pitää tukea ECN:ää, että se toimii. TCP-yhteyden muodostusvaiheessa laitteet neuvottelevat käytetäänkö ECN:ää. Reitittimet voivat ruuhkatilanteessa merkata ECN-tuetun liikenteen paketteja. (Pentikousis, Badr ja Kharmah 2002)

Myös lähetyspuskurit aihetuttavat TCP:ssä viivettä. Ne ovat tietyn kokoisia ja CWND-arvo voi olla enintään lähetyspuskurin kokoinen. Lisäksi lähetyspuskurin pitää olla riittävän iso, että se ei rajoita TCP:n suorituskykyä, eli CWND-arvon pitää pystyä kasvamaan hyödyntämään koko kaistanleveys. (Goel, Krasic ja Walpole 2008)

Goel, Krasic ja Walpole (2008) kertovat, että silloisissa Unix-käyttöjärjestelmäytimissä TCP:n

lähetyspuskuri on ainakin 64 kilotavun kokoinen. Lähetyspuskurin viiveisiin liittyen he antavat esimerkin, jossa käsitellään korkealaatuista 300 kilobittiä sekunnissa kulkevaa videostriimiä, jota mahtuu heidän mukaansa täyteen lähetyspuskuriin 1700 millisekuntia. Heidän mukaan viive kasvaa mikäli striimin käyttämä kaistaleveys pienenee kilpailun seurauksena tai huonompilaatuisella puskurilla. He vertaavat tätä viivettä RTT-aikaan Yhdysvaltojen länsi- ja itärannikon välillä, jonka he kertovat yleensä olevan 50-100 millisekuntia.

Goel, Krasic ja Walpole (2008) ehdottavat artikkelissaan tekniikkaa, jolla lähetyspuskurista aiheutuva viive saadaan minimoitua. Tätä he kutsuvat nimellä adaptiivinen lähetyspuskurin koon säätäminen. Lisäksi he nimittävät tätä tekniikkaa käyttäviä TCP-yhteyksiä MIN_BUF TCP -virroiksi. Heidän mukaan kyseisen tekniikan avulla lähetyspuskuri pidetään aina CWND-pakettimäärän kokoisena ja lähetyspuskurissa ei ole paketteja, jotka ovat odottamassa lähetystä. He kertovat, että tällöin lähetystä odottavat paketit säilyvät sovellusohjelman puolella, joten sovellusohjelmalla on mahdollisuus tehdä lähetystilanteeseen sopivia toimenpiteitä. Esimerkkinä kyseisistä toimenpiteistä he antavat, että sovellusohjelman on mahdollista lähettää korkeamman prioriteetin paketteja ensin.

Goel, Krasic ja Walpole (2008) kokeilivat MIN_BUF TCP -virtoja muun verkkoliikenteen kanssa käyttäen Linux 2.4 -testiympäristöä. Kyseisessä testiympäristössä he simuloivat WAN-yhteyksiä lisäämällä liikennettä kuljettavalle Linux-reitittimelle viivettä pakettien kuljetukseen. Muuna verkkoliikenteenä he käyttivät eri määriä lyhyt- ja pitkäaikaisia TCP-virtoja. Lisäksi he rasittivat testeissä käytettyä verkkoa yhdellä bittivirralla, jonka kaistanleveyttä pidettiin koko ajan samana.

Goel, Krasic ja Walpole (2008) tekivät kaksi eri verkkoa MIN_BUF TCP -testeille. Ensimmäisessä verkossa lähettävä tietokone teki kahdelle vastaanottavalle tietokoneelle liikennettä ja liikenne kulki reitittimen kautta. Toisessa verkossa oli kaksi lähettäjä-vastaanottaja-paria, jotka olivat kytketty samaan reitittimeen.

Mittausdataa Goel, Krasic ja Walpole (2008) keräsivät tarkkailemalla jokaisen lähetettävän paketin kirjoitusaikaa lähettävillä tietokoneilla. Vastaanottavien tietokoneiden päässä he tekivät vastaanotetun paketin lukuajan ylöskirjaamisen. Ensimmäisen testiverkon kanssa he tekivät kaksi testiajoa, joista toisessa testattiin normaalia TCP-virtaa ja toisessa MIN_BUF

TCP -virtaa. Yksi testi kesti noin 80 sekuntia, ja sen aikana he generoivat muuta liikennettä tietyissä ajankohdissa. Heidän artikkelissa olevat kuvaajat näyttävät suuria viivepiikkejä normaalilla TCP-yhteydellä ja MIN_BUF TCP -virran viivepiikit näyttävät erittäin pieniltä.

Goel, Krasic ja Walpole (2008) tekivät toisen testiverkon kanssa samat testiajot kuin ensimmäisellä. MIN_BUF TCP -virtaa kuvaava kaavio näyttää myös toisella testiverkolla merkittävästi pienempiä viivepiikkejä verrattuna normaaliin TCP-yhteyteen. Toisen testiverkon kanssa MIN_BUF TCP -virran viiveet näyttävät olevan kaavion perusteella suuremmat verrattuna ensimmäiseen testiverkkoon.

MIN_BUF TCP -virran käyttäminen kaiutinsovelluksen äänen suoratoistoon palvelimelta asiakkaalle olisi teoriassa mahdollista, ja tällöin WLAN-yhteyden kanssa kaiutinsovelluksen palvelin voisi esimerkiksi heikentää äänen laatua mikäli äänidatan kirjoittamisen odottaminen kestää liian kauan. Lisäksi olisi mahdollista pudottaa vanhaa lähettämätöntä äänidataa ja lähettää vain uusin data, mikäli sokettiin kirjoittamisen odottaminen on kestänyt äänivirran katkeamattoman toiston kannalta liian kauan. Tällöin äänentoisto on jo katkennut, joten lienee parempi aloittaa uudestaan mahdollisimman uudella äänidatalla. Tässä tutkimuksessa käsitellään äänen suoratoistoa langallisesti USB:n kautta, joten MIN_BUF TCP -virrasta ei liene hyötyä sillä oletuksella, että paketteja ei katoa USB:n kanssa.

Krasic, Li ja Walpole (2001) kertovat, että yleisesti ottaen TCP-protokollaa pidetään huonona vaihtoehtona multimedian suoratoistamiseen erityisesti pakettien uudelleenlähetyksen ja ruuhkanhallinnan takia. Kaiutinsovelluksen ja tämän tutkimuksen tapauksessa pakettien uudelleenlähetystä ei luultavasti tapahdu, koska äänidataa siirretään USB-yhteydellä, joka on langallinen yhteys. Ruuhkanhallinta voi olla ongelma kaiutinsovelluksen kannalta. Mikäli äänidatalla ei ole riittävän suurta kaistanleveyttä käytössä, niin tämä aiheuttanee äänipuskurin alivuotoja asiakasohjelman puolella. TCP-protokollan ja yleisesti äänidatan siirrosta puhelimelle aiheutuva viive kuuluu siirtämisen komponenttiin.

TCP-protokollassa on vuonvalvonta, joka tarkoittaa sitä, että vastaanottaja voi säädellä lähettäjän lähetysnopeutta (Fairhurst, Trammell ja Kühlewind 2017). Kaiutinsovelluksen kanssa vuonvalvonta ei liene ongelmana ainakaan kun äänen toistaminen on meneillään, koska vastaanotettu äänidataa luetaan soketista sitä mukaa, kun uutta äänidataa tarvitaan.

4.3 Muita äänen viiveeseen vaikuttavia asioita

Perneel, Fayyad-Kazan ja Timmerman (2012) käsittelevät artikkelissaan voiko Androidia käyttää reaaliaikaista suoritusta vaativiin tarkoituksiin. Kaiutinsovelluksen kannalta reaaliaikaisen suorituksen vaatimuksia ei ole, mutta tähän aihepiiriin liittyvien ongelmien havaitseminen auttanee tiedostamaan kaiutinsovellukseen ja sen alustaan liittyvät rajoitteet ja ongelmat.

Androidin arkkitehtuuri koostuu viidestä eri komponenttiluokasta, jotka voidaan sijoitella kerrosarkkitehtuurin tyylisesti eri tasoille. Alimmaisena on Linux-ydin, joka sisältää esimerkiksi laiteajureita. Linux-ytimen päällä on erilaisia kirjastoja, kuten Googlen C-standardikirjasto Bionic. Androidissa käyttäjän käyttämät ohjelmat suoritetaan Dalvik virtuaalikoneen päällä. Kirjastojen ja virtuaalikoneen päälle on rakennettu sovellusten käyttämät palvelut, kuten ilmoitusten hallinta. Sovellukset sijaitsevat ylimmällä kerroksella. (Perneel, Fayyad-Kazan ja Timmerman 2012)

Androidin C-standardikirjasto sisältää Androidille tiettyjä ominaisuuksia ja optimointeja. Esimerkiksi siihen on sisäänrakennettu mahdollisuus käyttää Androidin lokituspalvelua. Lisäksi kyseinen C-standardikirjasto ei tue C++ poikkeuksia, josta saadaan suorituskykyhyötyjä. Niiden tukeminen ei ole välttämätöntä, koska Androidin ohjelmat kirjoitetaan yleensä Javalla. Reaaliaikaisen suorituksen kannalta kyseinen C-standardikirjasto ei sisällä prioriteetin periytymistä tukevia mutekseja. (Perneel, Fayyad-Kazan ja Timmerman 2012)

Perneel, Fayyad-Kazan ja Timmerman (2012) suorittivat testejä Beagle-XM Board Rev C -alustalla, jossa on ARM-prosessori. Testeissä käytössä olleessa käyttöjärjestelmälevykuvassa oli Android 2.3.5. Testit suoritettiin osana arviointia, sopiiko Android reaaliaikaisen suorituksen alustaksi. Kellokeskeytyksen käsittelyajan keston testeissä he havaitsivat, että 350 mikrosekuntia on huonoin mahdollinen käsittelyaika. Tähän he pitivät syynä sitä, että käyttöjärjestelmäytimessä käytetään 32 kilohertsin ajastinta käyttöjärjestelmän kellolähteenä, koska korkeamman kellotaajuuden ajastimella tätä ei ilmennyt.

Perneelin, Fayyad-Kazanin ja Timmermanin (2012) testattaessa kontekstinvaihtoa saman prioriteetin säikeiden välillä, he huomasivat että kellokeskeytykset aiheuttivat yli 300 mikrosekunnin yksittäisen viiveen. Lisäksi säikeiden määrän kasvaessa viiveen keskiarvo ja pienin

viive kasvoivat. Tämän syyksi he totesivat sen, että säikeiden konteksti ei suuremmalla säie määrällä pysty olemaan välimuistissa.

Perneel, Fayyad-Kazan ja Timmerman (2012) testasivat keskeytysten viivettä ohjelmistopuolen osalta mittaamalla aikaa, joka kuluu suorituksen vaihtamiseen säikeestä keskeytyksen käsittelijään. He saivat keskiarvoksi tässä testissä 1,9 mikrosekuntia. Heidän testattaessa kuinka paljon poikkeuksia käsitellään, kun niitä tulee monta tietyn ajan välein, he huomasivat muun muassa, että kymmenen miljoonaa keskeytystä tarvitsi 410 mikrosekunnin ajanvälin, että kaikki keskeytykset käsiteltiin. Kun ajanväli oli 390 mikrosekuntia, niin kolme keskeytystä jäi käsittelemättä.

Perneelin, Fayyad-Kazanin ja Timmermanin (2012) tekemässä semafori-synkronointiprimitiivitestissä tarkasteltiin semaforin haltuunoton ja vapautuksen kestoa kiistelytilanteessa. Kyseisessä testissä 90 eri prioriteeteilla olevaa säiettä odotti semaforin lukitsemista. He testasivat semaforin haltuunoton kestoa mittaamalla aika kuinka kauan semaforin haltuunottaminen kestää. He tarkentavat, että kyseiseen aikaan kuuluu aika, joka kuluu säikeen vaihtamiseen, koska testi on tehty siten, että semaforia ei pysty haltuunottamaan heti ensimmäisellä yrityksellä. Heidän semaforin lukon vapautukseen liittyvässä mittauksessa mitattiin aika kuinka kauan semaforin lukon siirtyminen seuraavalle odottavalle säikeelle kestää.

Perneel, Fayyad-Kazan ja Timmerman (2012) käsittelivät semaforin haltuunoton testiä vain kaaviolla, jossa oli lukitsemisen kesto millisekunteina ja absoluuttinen aika millisekunteina. Absoluuttisella ajalla tarkoitetaan ilmeisesti testin alkamisesta laskettua aikaa. Semaforin lukon vapautuksen testituloksia käsiteltiin artikkelissa enemmän. Yhdellä odottavalla säikeellä vapautusaika oli 15 mikrosekuntia ja 90 odottavalla säikeellä tämä aika oli noin 450 mikrosekuntia. Syyksi testituloksille he esittävät artikkelissaan Androidin Bionic C-standardikirjaston semaforin toteutusta. Kyseinen semaforitoteutus ei heidän mukaan huomioi säikeiden prioriteetteja ja sen vapautusajat riippuvat odottavien säikeiden määrästä.

Perneel, Fayyad-Kazan ja Timmerman (2012) mukaan Androidin Bionic C-standardikirjastossa ei ole sellaista muteksia, joka tukisi säikeen prioriteetin periytymistä. Heidän mukaansa tämän ominaisuuden puuttuminen on riittävä asia olla käyttämättä Androidia reaaliaikajärjestelmissä.

Android 4.4 -versiossa esiteltiin uusi ajoaikaohjelmisto nimeltään Android Runtime (ART). Aiemmin Androidissa on käytetty vain Dalvik-ajoaikaohjelmistoa. Kehitettävän ohjelman lähdekoodi käännetään ohjelmapaketiksi, joka sisältää tavukoodia. Ohjelmapaketin suoritus menee Androidilla jonkun ajoaikaohjelmiston kautta. (Yadav ja Bhadoria 2015)

Kuten edellä mainittiin, niin ajoaikaohjelmistolle kaksi vaihtoehtoa. Voisiko kaiutinsovelluksen äänen viiveeseen vaikuttaa käytössä oleva ajoaikaohjelmisto? Yadav ja Bhadoria (2015) vertailivat ART- ja Dalvik ajoaikaohjelmistoja.

Dalvik käyttää Just-in-time (JIT) -käännöstekniikkaa tavukoodin kääntämiseen laitteessa olevalle prosessorille (Yadav ja Bhadoria 2015). Kääntäminen tapahtuu tällöin ohjelman suorituksen aikana. ART käyttää kääntämiseen Ahead-of-time (AOT) -käännöstekniikkaa, jossa tavukoodin kääntäminen puhelimen prosessorille tapahtuu ennen ohjelman suorittamista (Yadav ja Bhadoria 2015).

Yadav ja Bhadoria (2015) kertovat, että ART-ajoaikaohjelmistoon tehtiin parannuksia muun muassa roskankeruun (engl. *garbage collection, GC*) osalta. Heidän mukaansa roskankeruu voi vaikuttaa suoritettavan sovelluksen suorituskykyyn jossain määrin. He kertovat, että parannuksia roskankeruuseen on tehty muun muassa pienentämällä roskankeruutaukojen määrät kahdesta yhteen, kerääjä käyttää vähemmän aikaa hiljattain allokoiduille ja lyhytaikaisille objekteille sekä roskankeruupysähdyksessä käytetään rinnakkaista suoritusta pysähdyksen loppuosasta.

Yadav ja Bhadoria (2015) listaavat kolme asiaa, jolla ajoaikajärjestelmän tehokkuutta voidaan verrata toiseen ajoaikajärjestelmään. Ensimmäisenä he listaavat lyhytkestoisemmat hereilläololukot, jotka johtuvat sovellusten taustatoiminnasta. Toisena asiana he listaavat tehokkaan ajoaikaohjelmiston käyttävän vähemmän prosessorin kellojaksoja tietyn asian tekemiseen. Kolmantena he listaavat järkevämmän laitteiston käyttämisen prosessoriytimien osalta ja nostavat esimerkiksi ARM:in big.LITTLE-arkkitehtuurin. Heidän mukaan kyseisessä arkkitetuurissa vähävirtaisempia prosessoriytimiä käytetään yksinkertaisimpaan laskentaan ja suuritehoisempia vaativaan laskentaan.

Yadav ja Bhadoria (2015) suorittivat testejä ensimmäisen sukupolven Moto G -puhelimella, jonka käyttöjärjestelmänä oli Android 4.4.2. Testiohjelmana he käyttivät AnTuTu-suoritus-

kykytestausohjelmaa. He kertovat, että puhelimeen oli asennettuna kolmannen osapuolen sovelluksia, ja Wi-Fi sekä matkapuhelinverkko olivat käytössä. Testauksella he pyrkivät selvittämään onko Dalvik- vai ART-ajoaikaohjelmisto tehokkaampi. He kertovat, että testitulosten perusteella ART-ajoaikaohjelmisto tarjoaa hieman parempaa akunkestoa ja suorituskykyä.

Googlen (2022) dokumentaatio neuvoo toistamaan hiljaisuutta ennen oikeaa äänidataa. Dokumentaation mukaan edellisen tekemällä vältetään Android-laitteen äänipiirin lämpenemisestä aiheutuva viive. Googlen (2022) Android 11 yhteensopivuusmäärittelyssä, joka on ilmeisesti tarkoitettu laitevalmistajien käyttöön, suositellaan vahvasti pitämään kylmän ulostulon viive enintään 100 millisekunnissa. Lisäksi jatkuvan ulostulon viiveeksi määritellään vahvalla suosituksella enintään 45 millisekuntia. Edelliset luvut eivät ole aivan vertailukelpoiset äänipiirin lämpenemisen viiveen vertailua ajatellen, koska yhteensopivuusmäärittelyssä kerrotaan jatkuvan ulostulon tarkoittavan peräkkäisten äänikehysten toistamisen viivettä, kun äänen toistaminen on jo käynnistynyt. Yhteensopivuusmäärittelyssä kylmällä ulostulolla tarkoitetaan ensimmäisen äänikehyksen toistamisen viivettä suljetulla äänijärjestelmällä. Luvuissa on kuitenkin sen verran eroa, että äänipiirin lämmittämisen viive kannattaa huomioida erityisesti kaiutinsovellusta toteuttaessa.

5 Äänen viiveen mittaaminen

Tässä luvussa käsitellään erilaisia äänen viiveen mittausmenetelmiä ja tässä tutkimuksessa käytettävää äänen viiveen mittausmenetelmää.

5.1 Erilaisia mittausmenetelmiä

Googlen (2023) Android-dokumentaatio kertoo erilaisista äänen viiveen mittausmenetelmistä. Dokumentaatiossa äänen viiveen mittaaminen on jaettu äänen ulostulon, äänen edestakaisen kulkemisen ja äänen sisääntulon viiveen mittauksiin. Dokumentaatiossa kerrotaan äänen edestakaisen kulkeminen tarkoittavan sitä, että kuinka paljon viivettä äänen ulostulo ja sisääntulo aiheuttaa yhdessä. Seuraavaksi kerrotaan dokumentaatiossa olevista mittausmenetelmistä.

Äänen ulostulon viiveen mittaamiseen Googlen (2023) dokumentaatio kertoo LED-valolla ja oskilloskoopilla tehtävästä testistä. Kyseisessä testissä mittauksen kohteena olevassa puhelimessa on kiinni LED-valo, jota välkytetään samaan aikaan sen toistaessa ääntä. LED-valon vieressä on valosensori, joka taas on kiinni oskilloskoopissa. Testin toinen oskilloskoopissa oleva signaalilähde on puhelimen kuulokeliitännästä tuleva äänisignaali. Dokumentaation mukaan oskilloskoopilla pystyy mittaamaan kyseisten signaalien välisen viiveen. Dokumentaatio kertoo, että tyypillisesti LED-valon ja valosensorin viive on luokkaa yksi millisekunti tai vähemmän. Dokumentaation mukaan kyseisestä viiveestä ei tarvitse välittää, koska se on riittävän pieni.

Äänen edestakaisen kulkemisen viiveeseen liittyen Googlen (2023) dokumentaatio kertoo akustiseen kiertoon liittyvästä testistä. Testissä käytettävä puhelinsovellus nauhoittaa ääntä puhelimen mikrofonista ja toistaa kaapatun äänen ulos puhelimen kaiuttimesta. Dokumentaatio kertoo, että nauhoituksen ja toiston ollessa päällä pitää tehdä aluksi jokin äänimerkki, joka aloittaa äänen kiertämisen. Lisäksi testissä käytössä oleva puhelinsovellus tallentaa nauhoitteen, josta kierto on nähtävissä äänisignaalissa. Dokumentaation mukaan äänen viiveen voi laskea nauhoitteesta tarkastelemalla äänimerkkien välistä aikaa.

Äänen sisääntulon viivelle Googlen (2023) dokumentaatio esittää kaksi testiä. Ensimmäisessä hyödynnetään kahta edellä esitettyä testiä laskemalla äänen sisääntulon viive vähentämällä äänen ulostulon viive äänen edestakaisen kulkemisen viiveestä. Dokumentaation mukaan tällöin lopputuloksena on arvio äänen sisääntulon viiveestä. Toinen testi käyttää puhelimessa mahdollisesti olevaa GPIO-pinniä ja mikrofonia. Dokumentaation mukaan jonkin ulkoisen testilaitteen pitää lähettää samaan aikaan GPIO-pinniin signaali ja mikrofonille äänisignaali. Dokumentaatiossa neuvotaan mittaamaan kyseisen kahden signaalin vastaanottamisen aikaero puhelinsovelluksella.

Brandt ja Roger (1998) selvittivät, miten reaaliaikaisten musiikkiohjelmien vaatimukset täyttyisivät sen aikaisilla käyttöjärjestelmillä. Vertailussa oli mukana Windows NT 4, Windows 95 ja Irix 6.4. Ilmeisesti he käyttivät heidän tekemäänsä reaaliaika objektijärjestelmään pohjautuvaa Aura-arkkitehtuuria mittausten tekemisessä. Windows NT:llä äänen ulostulon viive oli 186 millisekuntia. Alustavien testien mukaan Irix 6.4:llä kyseinen luku oli seitsemän millisekuntia. Windows 95 -käyttöjärjestelmälle he eivät anna mitään tiettyä lukua. Artikkelista jää epäselväksi miten luvut ovat saatu. Tutkimuksen yhteydessä olisi pitänyt raportoida, mitä he tarkoittavat äänen ulostulon viiveellä. Artikkelissa on myös testattu viiden millisekunnin takaisinkutsufunktion viiveitä erilaisilla prosessorikuormilla.

Bouillot ja Cooperstock (2009) käsittelevät artikkelissaan muun muassa äänen suoratoistoa ja he esittävät menetelmän, joka mittaa tarkasti äänen suoratoiston viivettä. Menetelmässä äänen lähde on kiinni suoratoistoa lähettävässä tietokoneessa ja monikanavanauhurissa kiinni. Suoratoistoa vastaanottavassa laitteessa äänen ulostulo ohjataan monikanavanauhuriin. Monikanavanauhurilla nauhoitetaan molemmat äänilähteet ja nauhoite tutkitaan monikanavaäänieditorilla. Nauhoitteesta etsitään manuaalisesti kummaltakin kanavalta selkeästi erottuva kohta signaalista ja näiden välinen aika on suoratoiston viive.

Zigunovs ym. (2017) artikkelissa käsitellään syitä ja ratkaisu Android-laitteiden äänen toistamisen viiveeseen. Heidän käyttämässä mittausmenetelmässä puhelimen LED-valo on valosensorin kautta yhdistettynä oskilloskooppiin. Puhelimen kuulokeliitäntä on myös oskilloskoopissa kiinni. Heidän käyttämä sovellus tekee komennot toistaa ääntä ja kytkeä LED-valo päälle eri säikeissänsä. Kyseisellä sovelluksella on mahdollista toistaa ääntä MediaPlayer, OpenSL ES, Superpowered Audio API -rajapinnoilla. Lisäksi he testasivat AAC-, MP3- ja

PCM-ääniformaatteja, sekä 44,1 kHz ja 48 kHz -näytteistystaajuuksia äänitiedostoissa. He kertovat, että PCM-formaatti toistetaan nopeiten. Lisäksi he kertovat, että äänidata prosessoidaan nopeammin, kun ääni on digitoitu suoraan laitteen tukemaksi. Tällä he ilmeisesti tarkoittavat näytteistystaajuutta. Tutkimuksen äänirajapinnoista viiveen kannalta ilmeisesti paras oli OpenSL ES.

Wright, Cassidy ja Zbyszynski (2004) mittasivat äänen sisääntulon ja äänen toistamisen välistä viivettä MacOS, Red Hat Linux ja Windows XP -järjestelmillä. Mittaukset he tekivät nauhoittamalla äänen sisääntuloa ja kyseistä äänen sisääntuloa toistavan tietokoneen ääniulostuloa. Nauhoituksessa syntyvät äänitiedostot he analysoivat Matlabilla käyttäen heidän tekemiänsä ohjelmia.

Wang, Stables ja Reiss (2010) mittasivat äänen viivettä tietokoneisiin integroiduilla äänikorteilla ja eri käyttöjärjestelmillä. He perustelevat tutkimuksensa motivaatiota siten, että yleisesti kaupasta saatavia tietokoneita käytetään digitaalisen äänen työskentelyasemina ja jotkin toiminnot vaativat pientä äänen viivettä. Heidän mukaan tällainen toiminto on live-äänen monitorointi, kun käytössä on digitaalisella signaalinprosessoinnilla tuotettuja ääniefektejä. Heidän käyttämänsä mittausmenetelmä vaikuttaa olevan sama kuin aiemmin mainitulla monikanavanauhurilla. Äänilähteen äänisignaalina he käyttävät matemaattisesti laskettua äänipulssia latenssin mittaamiseen. Lisäksi useampi äänipulssi on käytössä, kun he testaavat informaation häviämistä, häiriöitä ja äänen viiveen vaihteluita.

Laitteistona Wang, Stables ja Reiss (2010) käyttivät Intel-prosessorin sisältäviä Applen tietokoneita, koska näiden avulla oli mahdollista käyttää kolmesta suositusta käyttöjärestelmästä jokaista. Kyseisillä käyttöjärjestelmillä he ilmeisesti tarkoittavat Windowsia, Linuxia ja Mac OS X:ää. Heidän tekemissä äänen viiveen testeissä pienin äänen viive 1,68 millisekuntia saavutettiin käyttäen Linuxia, ALSA-äänirajapintaa, $64 * 2$ bufferikokoa, Ardour-ohjelmaa ja 96 000 Hz -näytteistystaajuutta.

Myös MacMillan, Droettboom ja Fujinaga (2001) käyttivät omissa äänen viiveen mittauksissaan Windowsia, Linuxia ja MacOS:ää. Myös heidän mittausmenetelmä oli monikanavanauhurityylinen. Mittaustuloksia he vertasivat mikserin äänen viiveeseen. Ensimmäiset kaksi tulosta menivät Linux-ytimelle, joka sisälsi viiveitä pienentäviä muutoksia, ja normaa-

lille Linux-ytimelle. ALSA-äänirajapinta oli kyseisissä mittauksissa käytössä. Molemmissa mittauksissa äänen viive oli 2,72 millisekuntia. Kolmantena oli Windows 2000 ASIO-äänirajapinnalla viiveellä 3,11 millisekuntia. Neljäntenä oli MacOS X CoreAudio-äänirajapinnalla viiveellä 3,97 millisekuntia. Kaikki edellä olevat mittaukset tehtiin, kun järjestelmää ei kuormitettu muuten. Mikserillä tehty mittaus oli 1,81 millisekuntia.

Lisäksi MacMillan, Droettboom ja Fujinaga (2001) tekivät mittauksia, kun järjestelmää kuormitettiin kahdella eri ohjelmalla, joista toinen teki laskentaa ja toinen kuormitti järjestelmän tallennuslevyn käyttöä jotenkin. Näistä mittauksista MacOS X CoreAudio-rajapinnalla antoi pienimmän viiveen 3,97 millisekuntia. Toisena, kolmantena ja neljäntenä olivat erilaiset Linux-ytimet LAAGA-äänirajapinnalla (Linux Audio Application Glue API) viiveellä 4,3 millisekuntia. Viidentenä oli Windows 2000 ASIO-äänirajapinnalla viiveellä 6,03 millisekuntia.

Ye ym. (2018) tekivät äänen viiveen mittaukset käyttäen äänen viiveen mittausmenetelmänä monikanavanauhurimenetelmältä vaikuttavaa menetelmää. He käyttävät yhtä äänipulssia viiveen mittauksiin ja useamman pulssin ketjuja häiriöiden, äänidatan häviämisen ja vaihtelevien viiveiden seurantaan. Käyttöjärjestelminä testeissä oli mukana Linux, Mac OS ja Windows 10. Pienimmän latenssin testeissä pärjäsi parhaiten MacOS CoreAudio-äänirajapinnalla yhdeksän millisekunnin äänen viiveellä. Ääniohjelmana käytössä oli Reaper v5.23. Toisena on Linux ALSA-äänirajapinnalla 10 millisekunnin viiveellä. Ääniohjelmana käytössä oli Pure Data 0.46.7.

Lisäksi Ye ym. (2018) mittasivat myös äänen viivettä prosessorin ylimääräisen rasituksen ollessa käynnissä. Heidän havainto oli, että tällä ei ollut vaikutusta äänen viiveeseen. Näissä testeissä Windows 10 WaveOut-äänirajapinnalla ja Reaper v5.24pre 12 -ääniohjelma sai 55 millisekuntia viivettä puskurikoon ollessa 56. MacOS CoreAudio-äänirajapinnalla sai seitsemän millisekuntia lohkokoon ollessa 24 ja ääniohjelman ollessa Reaper v5.23. Linux ALSA-äänirajapinnalla sai 11 millisekuntia äänen viivettä lohkokoon ollessa 64, viiveen kahdeksan ja ääniohjelman Pure Data 0.46.7.

Edellisten tutkimusten perusteella voinee sanoa, että monikanavanauhoitus äänen viiveen mittauksessa on kohtuullisen yleinen ratkaisu. Normaaleilla tietokoneilla se on kohtuullisen

helppo toteuttaa, koska ainakin useimmissa pöytätietokoneissa on liitännät äänen toistamiseen ja nauhoittamiseen. Mikäli puhelimessa on kuulokemikrofoniliitäntä, niin samaa menetelmää voisi käyttää myös puhelimilla tehtäviin äänen viiveen mittauksiin. Tässä tapauksessa puhelimen kuulokemikrofoniliitäntään pitäisi varmaankin hankkia adapteri, jolla siihen voi kytkeä kaksi eri 3,5 mm äänijohtoa äänen sisääntulolle ja ulostulolle.

Äänen viiveen mittausmenetelmän valinnassa tulee kiinnittää huomiota, että mitä viivettä mikäkin mittausmenetelmä oikeasti mittaa. Esimerkiksi edellä mainittu monikanavanauhurimenetelmä mittaa äänen sisääntulon, ulostulon ja niiden välisestä prosessoinnista aiheutuvan viiveen.

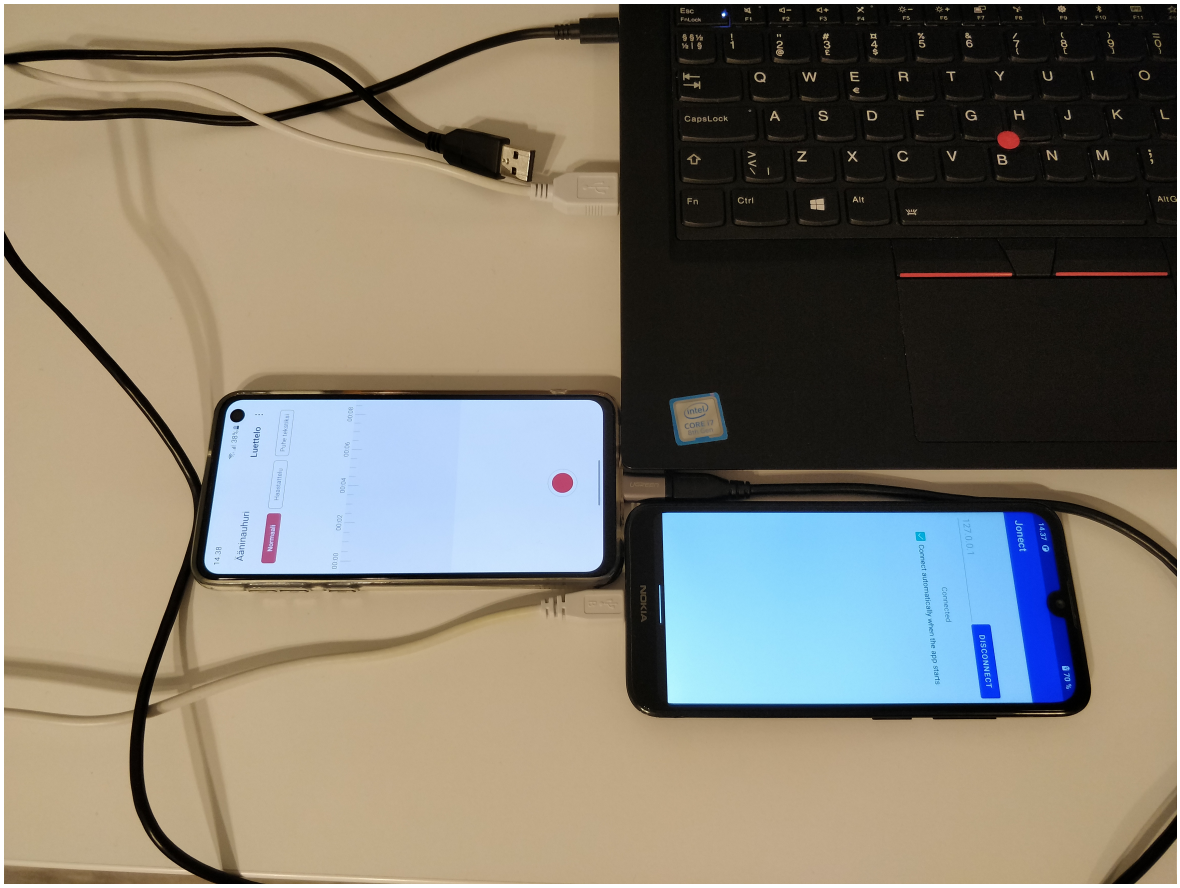
5.2 Tutkimuksen mittausmenetelmä

Kaiutinsovelluksen prototyypin kehityksen aikana huomattiin, että oli mahdollista soittaa kaiutinsovelluksesta ja tietokoneen kaiuttimista ääntä. Tämä oli mahdollista toteuttaa PulseAudion tarkkailuäänilähteiden (engl. *monitor sources*) avulla. Ääniulostuloilla on ainakin yleensä yksi tarkkailuäänilähde, josta pystyy nauhoittamaan saman äänen mikä kaiuttimista soitettaisiin.

Prototyypin äänen viiveen huomasi, kun tietokoneesta toisti jotain ääntä samaan aikaan kaiuttimista ja kaiutinsovelluksesta. Edellistä tapahtumaa keksittiin soveltaa äänen viiveen mittauksiin lisäämällä mukaan tapahtuman ääntä nauhoittava puhelin, tekemällä sopiva äänimerkki Audacity-ohjelmalla ja asettamalla eri äänenvoimakkuudet tietokoneen ja puhelimen kaiuttimille.

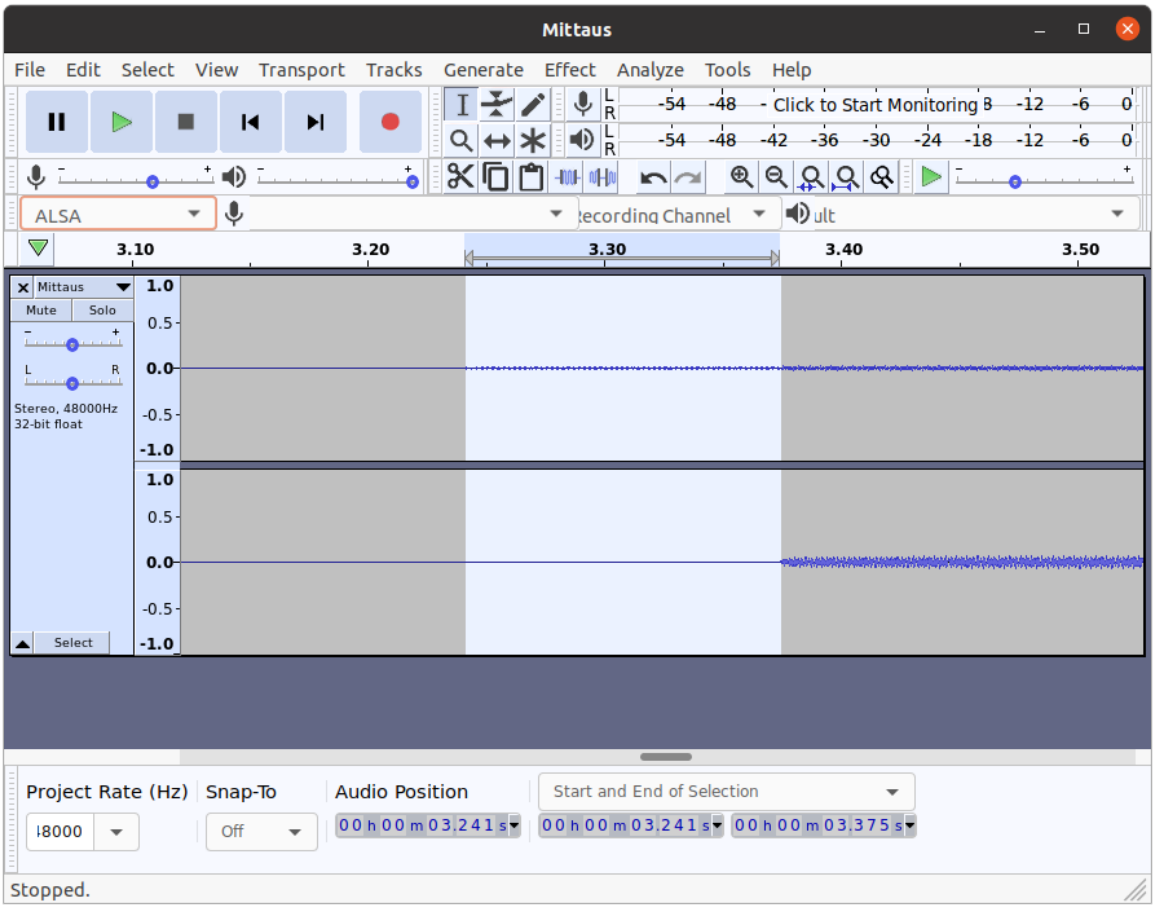
Mittausmenetelmän mittauslaitteisto on nähtävissä kuviossa 5. Mittauksessa kuuluva ääni nauhoitettiin Samsung Galaxy S10e -puhelimen ääninauhurisovelluksella. Nauhoitetuista äänitiedostoista katsottiin silmämääräisesti alkukohdat tietokoneen ja puhelimen äänimerkeille Audacity-ohjelmalla ja tämän jälkeen hyödynnettiin Audacityn ominaisuutta, joka näyttää äänen pituuden ajallisesti sille osalle äänidataa, joka on ohjelmassa valittuna. Kyseinen vaihe mittausmenetelmästä on nähtävillä kuviossa 6.

Tarkemmin sanottuna äänen viivettä mitattiin suurin piirtein seuraavalla tavalla:



Kuvio 5. Äänen viiveen mittauslaitteisto. Voi olla, että luvussa 6 tehdyissä mittauksissa puhelimet olivat toisin päin kuin kuvassa, että kaiuttimet ja mikrofonit ovat lähempänä toisiansa.

1. Asennetaan tietokoneelle kaiutinsovelluksen palvelinohjelmasta mittauksessa käytettävä versio. Lisäksi asennetaan puhelimeen asiakasohjelman mittauksessa käytettävä versio.
2. Asetetaan laitteet siten, että mikrofonit ja kaiuttimet ovat sopivan etäisyyden päässä toisistansa.
3. Yhdistetään palvelinohjelma ja asiakasohjelma toisiinsa.
4. Säädetään äänenvoimakkuutta kummastakin laitteesta siten, että korvakuulon perusteella tietokoneen äänenvoimakkuus on pienemmällä ja puhelimen isommalla.
5. Toistetaan mittauksissa käytettävä äänimerkki tietokoneessa käynnissä olevasta Audacity-ohjelmasta.
6. Tehdään yksi testimittaus sopivien äänenvoimakkuuksien tarkistamiseksi tai suoraan



Kuvio 6. Äänen viiveen mittaustiedoston tutkiminen Audacity-ohjelmalla.

kolmen mittauksen sarja. Tarvittaessa jälkimmäisessä tapauksessa tehdään uusia mittauksia epänormaalin pituisten viiveiden takia.

7. Siirretään mittaustiedostot tietokoneelle ja katsotaan äänen viive tiedostoista avaamalla ne Audacity-ohjelmalla, etsimällä silmämääräisesti tietokoneen kaiuttimen äänen alku ja puhelimen äänen alku sekä lopuksi laskemalla viive hyödyntäen Audacityn valintatyökalua, joka näyttää aloitusajan ja lopetusajan valitulle ääniraidan osalle.
8. Mikäli mittauksista ei erota tietokoneen ja puhelimen äänen alkamista, niin säädetään äänenvoimakkuutta ja tehdään mittaukset uusiksi.

Edellisellä mittausmenetelmällä saa vain suuntaa antavia tuloksia, koska ainakin silmämääräisessä ääniraidan tulkitsemisessa on virhemahdollisuuksia. Myöskään mittausasetelman äänenvoimakkuuksia ja tarkkoja sijainteja mikrofoneille ja kaiuttimille ei ole määritelty mit-

tausmenetelmän kuvauksessa, mikä vaikeuttaa yhden mittauksen toistamista myöhemmin uudestaan.

Tutkimuksen painopisteen ollessa ohjelmistokehityksessä mittausmenetelmä on sopiva tutkimuksessa käytettäväksi. Toki olisi ollut mahdollista käyttää tarkempaa mittausmenetelmää tähän tutkimukseen ja edellä kuvattua mittausmenetelmää olisi voinut parantaa esimerkiksi määrittämällä tarkka sijainti ääntä nauhoittavalle puhelimelle. Tämä ei kuitenkaan aika- ja motivaatioresursseista johtuen ollut mahdollista.

6 Tutkimus

Tässä luvussa yritetään pienentää kaiutinsovelluksen äänen viivettä tekemällä erilaisia parannuksia sovellukseen. Ensin mitataan tutkimuksen lähtötilanteen sovelluksen äänen viive, jonka jälkeen tehdään kolmen iteraation verran sovelluksen jatkokehittämistä ja muun muassa äänen viiveen mittauksia.

Äänen viiveen mittaukset suoritettiin kaiutinsovelluksen ajamisen osalta Lenovo Thinkpad T480s kannettavalla tietokoneella, Nokia 2.2 -puhelimella ja viimeisen iteraation tapauksessa myös OnePlus 5 -puhelimella. Käyttöjärjestelmänä tietokoneessa oli Ubuntu 20.04 ja ensimmäisessä puhelimessa Android 11 ja jälkimmäisessä Android 10. Molempien puhelinten äänijärjestelmä käyttää 48 000 Hz -näytteistystaajuutta ainakin äänen toistamiseen, joten palvelinohjelma käytti nauhoituksessa kyseistä näytteistystaajuutta.

Asiakasohjelman äänipuskurin koon asettamisessa käytettiin apuna `AudioTrack.getMinBufferSize`-aliohjelmaa tutkimuksen lähtötilanteessa ja myöhemmissä vaiheissa `OUTPUT_FRAMES_PER_BUFFER`-vakiota. Kyseinen vakio on Googlen (2022) dokumentaation mukaan Android-laitteen pienen latenssin äänivirran natiivi tai optimaalinen äänipuskurikoko. Nokia 2.2 -puhelimella kyseinen vakio on 256, joka tarkoittaa 48 000 Hz -näytteistystaajuudella noin 5,3 millisekuntia äänidataa. OnePlus 5 -puhelimella kyseinen vakio on 192, joka on samalla näytteistystaajuudella 4 millisekuntia.

Mittauksen nauhoittamiseen käytettiin Samsung Galaxy S10e -puhelimen ääninauhurisovellusta. Mittaukset nauhoitettiin 48 000 Hz -näytteistystaajuudella ja stereo-asetuksella.

Testien palvelinohjelman versiot käännettiin release-asetuksella. Asiakasohjelman osalta käytettiin debug-asetusta, pois lukien Rust- ja C++-lähdekoodit, jotka käännettiin release-asetuksella. Syy debug-asetuksen käyttöön oli se, että asiakasohjelman asentaminen puhelimeen oli helpompaa debug-asetuksella.

6.1 Lähtötilanne

Tutkimuksen lähtötilanteena käytetään luvussa 3 esiteltyä sovellusta. Mittauksessa käytettiin palvelinohjelman committia `c4724ed` ja asiakasohjelman committia `64f4637`.

Asiakasohjelman äänipuskurin kooksi asetettiin `AudioTrack.getMinBufferSize`-aliohjelman palauttama arvo kerrottuna kahdella. Puskurin koko oli tällöin äänikehyksissä mitattuna 2052, jolloin äänipuskuriin mahtuu 42,75 millisekuntia ääntä. Kyseinen äänipuskurikoko oli riittävän suuri, eikä puskurin alivuotoja esiintynyt mittaustilanteen aikana. Mahdollisesti pienempi puskurikoko, esimerkiksi 1,5-kertoimella kerrottu koko, olisi myös toiminut, mutta tätä ei kokeiltu. Ääntä puskuroitiin äänipuskuriin 2040 äänikehyksen (8160 tavun) verran ennen kuin äänen toistaminen aloitettiin.

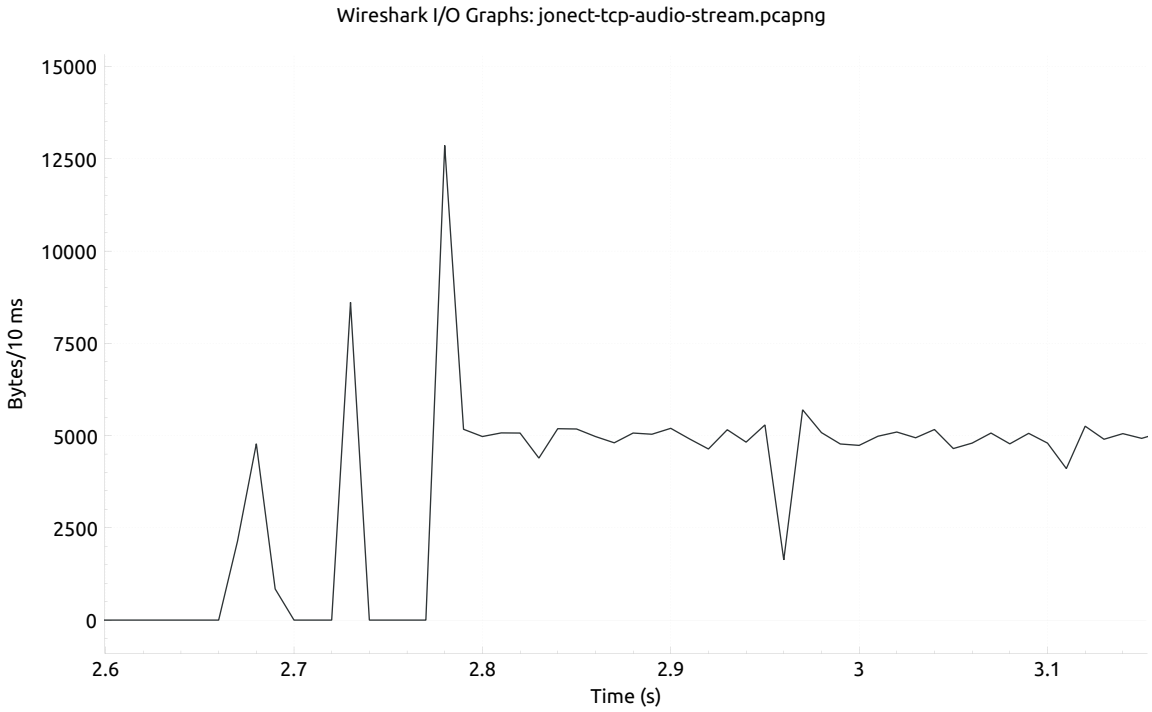
Lähtötilanteen mittaustulokset olivat noin 134, 138 ja 132 millisekuntia. Mittaustulosten keskiarvo oli noin 135 millisekuntia.

Viivettä aiheuttanee erityisesti iso äänipuskuri, mutta se on puskurin alivuotojen estämisen kannalta oleellinen. Äänen siirtäminen ei ole tarpeeksi tasaista ADB:n ja USB:n kautta kulkevalla TCP-yhteydellä, että `AudioTrack.getMinBufferSize`-aliohjelman palauttamaa arvoa voisi suoraan käyttää äänipuskurin kokona. Tähän liittyvää lisätietoa varten tehtiin palvelinohjelmasta lähtevälle TCP-yhteydelle pakettikaappaus Wireshark-ohjelmalla. Pakettikaappaus tehtiin samalla puhelimella, mutta eri tietokoneella kuin äänen viiveen mittaus. Tietokoneen käyttöjärjestelmä oli Ubuntu 20.04, joka on sama kuin äänen viiveen mittauksessa käytetyssä kannettavassa tietokoneessa.

Kuviossa 7 nähdään TCP-yhteyden tiedonsiirrossa epävakautta erityisesti, kun äänen siirtäminen alkaa. Yhteydessä on jonkin verran epävakautta myös alussa esiintyvien suurten vaihteluiden jälkeen.

Kuviossa 8 nähdään TCP-yhteyden yksittäisten pakettien kokojen vaihtelu. Kuviosta erottuu erityisesti kaksi muita paketteja paljon suurempaa pakettia. Nämä paketit näkyvät kuvion 7 käyrän alussa olevina hyppyinä.

Kuviossa 9 edellä mainitut kaksi pakettia on rajattu pois kaaviosta, että normaali pakettien kokovaihtelu on paremmin näkyvissä. Pakettien koossa on vaihtelua, joka selittää kuviossa



Kuvio 7. TCP-yhteyden käyttäytyminen äänensiirrossa: tavuja/10 ms

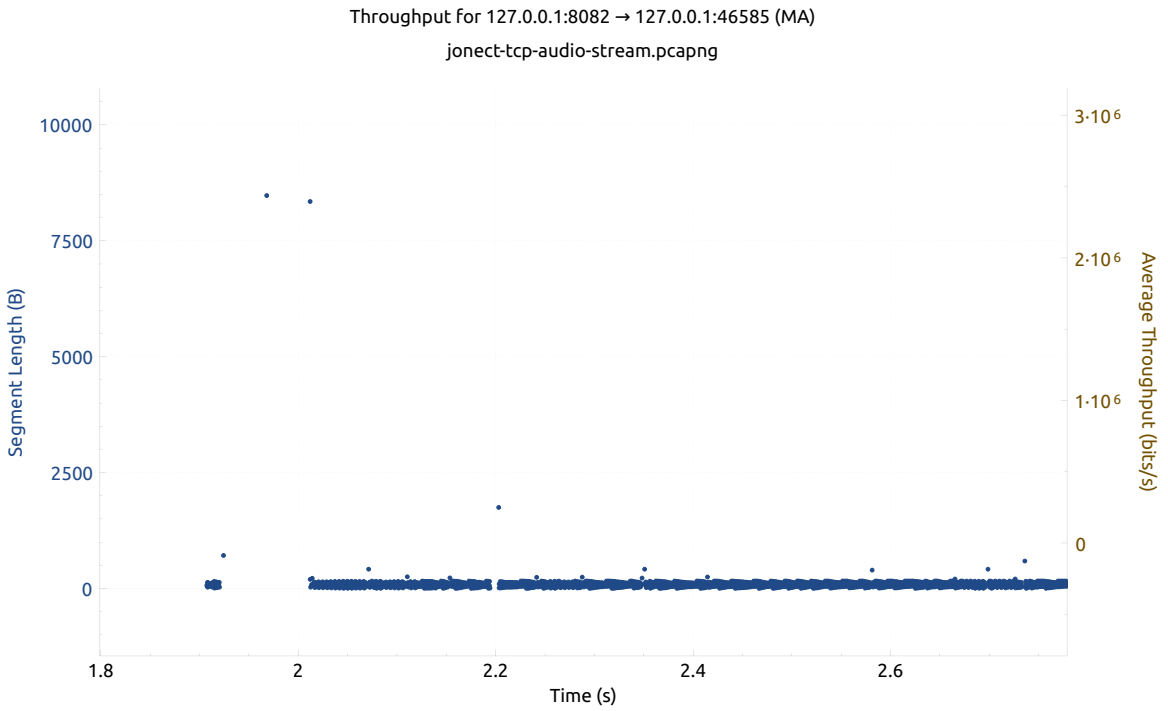
7 näkyvää normaalia tiedonsiirron nopeuden vaihtelua.

Edellisen perusteella voidaan tehdä se johtopäätös, että sovelluksen tiedonsiirtomenetelmä äänidatan siirtämiseen ei ole sopiva sovelluksen käyttötarkoitusta ajatellen. Pakettien kokovaihtelu aiheuttanee puskurin alivuotoja.

Pakettikaappauksessa ei näkynyt TCP-yhteydelle ominaisia pakettien uudelleenlähetyksiä, joten UDP-protokollan käyttäminen tiedonsiirtoon luultavasti antaisi tasaisemman tiedonsiirron ainakin USB:n yli siirrettynä. UDP-protokollaa käytettäessä paketteja lähettävä ohjelma voi itse määrittää pakettien koon, joten pakettikaappauksessa havaittua äänidataa sisältävien pakettien kokojen vaihtelua ei enää tapahtuisi.

6.2 Ensimmäinen iteraatio: Oboe ja Rust

Ensimmäinen iteraatio päätettiin käyttää asiakasohjelman arkkitehtuurin muokkaamiseen sovelluksen kehityksen kannalta järkevämpään suuntaan jo ennen lähtötilanteen äänen viiveen mittausta. Iteraatiossa palvelinohjelman toimintalogiikkaan liittyvä koodi muokattiin kirjas-

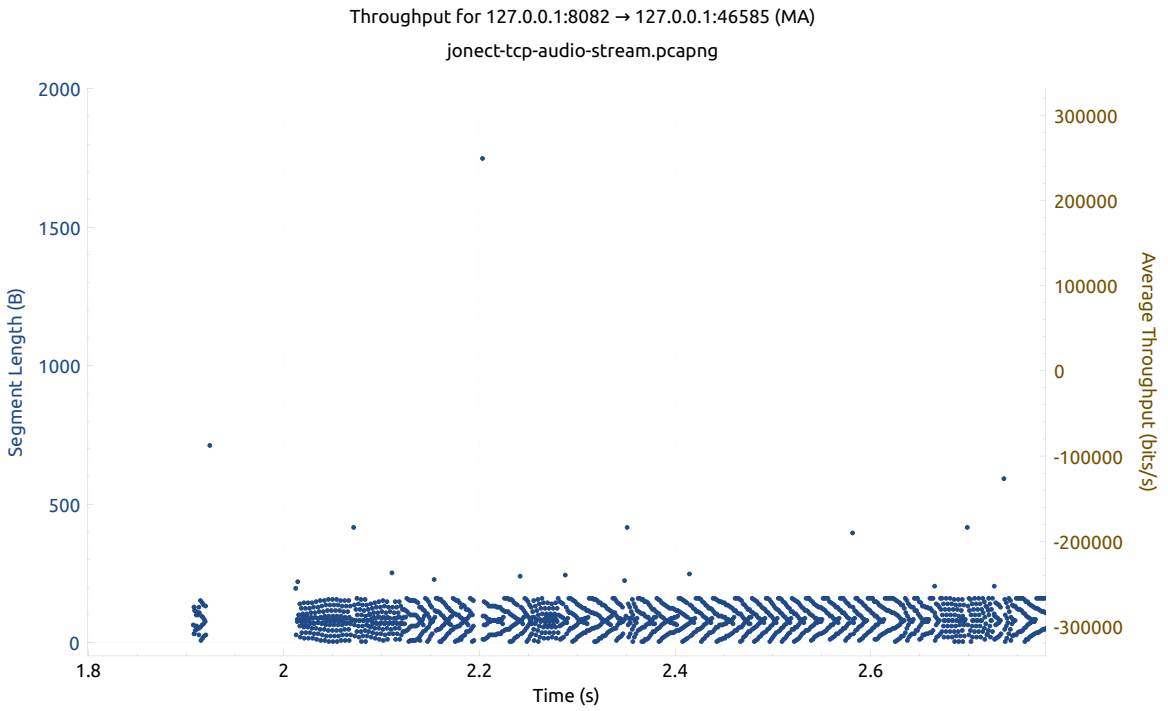


Kuvio 8. TCP-yhteyden käyttäytyminen äänensiirrossa: pakettien koko

toksi, jota on mahdollista käyttää myös asiakasohjelmasta. Kyseinen muutos vaikutti helpottavan sovelluksen kehittämistä, koska jaetun koodin ansiosta ohjelmia tarvitsee muokata vähemmän esimerkiksi asiakas- ja palvelinohjelman välisen JSON-protokollan tapauksessa.

Edellisen muokkauksen yhteydessä tietokoneen ja puhelimen välillä olevasta JSON-protokollasta poistettiin erilliset asiakas- ja palvelinroolit, mikä tarkoittaa käytännössä, että eri alustoilla toimivat sovellukset ovat vertaisia JSON-protokollan tasolla. Käytännössä asiakas- ja palvelinroolit ovat edelleen olemassa, koska äänen nauhoittamista puhelimella ja äänen toistamista tietokoneella ei ole toteutettu.

Asiakasohjelmaan lisätyn logiikkakirjaston myötä palvelinohjelmaan yhteyden tekevä koodi on kirjoitettu Rust-ohjelmointikielellä, joten sen ja luultavasti myös äänen viiveen pienentämisen kannalta oli järkevää toteuttaa myös äänen toistaminen tapahtumaan Rust-koodista. Tämä tehtiin lisäämällä logiikkakirjastoon toteutus äänen toistamiseen Oboe-äänikirjaston kautta. Balsinin ym. (2019) mukaan Oboe-kirjasto kuuluu pienen viiveen C- tai C++-äänirajapintojen joukkoon.



Kuvio 9. TCP-yhteyden käyttäytyminen äänensiirrossa: pakettien koko (rajattu)

Androidin AudioTrack-luokalla ja Nokia 2.2 -puhelimella pienin mahdollinen äänipuskurin koko on 1026 äänikehystä (noin 21,4 millisekuntia 48 000 Hz -näytteistystaajuudella). Oboe-kirjaston ja Nokia 2.2 -puhelimella pienin mahdollinen äänipuskurin koko on ilmeisesti 768 äänikehystä (16 millisekuntia 48 000 Hz -näytteistystaajuudella). Oboe palauttaa 768 äänikehyksen kokoisen äänipuskurin vaikka siltä pyytää 512 äänikehyksen kokoisen äänipuskurin, joten Nokia 2.2 -puhelimella kyseinen äänipuskurikoko lienee pienin mahdollinen.

Ilmeisesti puskurin alivuotojen ja/tai sovelluksen äänen viiveen vertailukelpoisuuden lisäämistä lähtötilannetta ajatellen Oboen äänipuskuri asetettiin 2048 äänikehykseen (noin 42,7 millisekuntia 48 000 Hz -näytteistystaajuudella), joka on `OUTPUT_FRAMES_PER_BUFFER`-vakion monikerta ja lähellä lähtötilanteen äänipuskurin kokoa.

Äänen toistamiseen käytettävä algoritmi muuttui lähtötilanteesta, joten se on hyvä huomioida äänen viivettä arvioidessa. Algoritmi on suurin piirtein seuraava:

1. Yhdistetään palvelimen TCP-porttiin, joka on tarkoitettu äänidatalle. Siirretään TCP-sokettiin liittyvä olio viestikanavia pitkin äänen toistoa hallinnoivalle säikeelle.

2. Luodaan äänidatan lukemiseen liittyvä säie. TCP-soketista luetaan dataa tässä säikeessä.
3. Vaihdetaan äänidatan lukemiseen käytettävän säikeen prioriteetiksi vakio `Process.THREAD_PRIORITY_AUDIO`.
4. Luetaan TCP-soketista äänidataa 2048 äänikehyksen verran viestikanavaan, jossa äänidata on jaettu kahdeksan äänisivun paloihin.
5. Lähetetään äänen toistoa hallinnoivalle säikeelle tieto käynnistää äänen toistaminen. Äänidatan lukeminen TCP-soketista viestikanavaan jatkuu.
6. Oboe käynnistetään äänen toistoa hallinnoivassa säikeessä ja se luo korkean prioriteetin säikeen äänen toistamista varten. Kyseisessä säikeessä suoritetaan takaisinkutsufunktiota, jossa kirjoitetaan uutta äänidataa äänipuskuriin.
7. Luetaan takaisinkutsufunktiossa äänidataa viestikanavasta tarvittava määrä. Mikäli uutta äänidataa ei ole saatavilla, niin jäädään odottamaan sitä.

Edellisen algoritmin yksityiskohdat voi lukea mittauksessa käytettyjen ohjelmien lähdekoodeista, jotka löytyvät palvelinohjelman osalta palvelinohjelman git-varaston commitista `396-4c6f` ja asiakasohjelman osalta asiakasohjelman git-varaston commitista `9cfd397`.

Äänen viive mitattiin neljällä mittauksella, jotka olivat noin 96, 98, 2018 ja 96 millisekuntia. Mittauksia tehtiin yksi enemmän kuin edellisellä kerralla, koska kolmannessa mittauksessa äänen viive oli epänormaalin suuri jostain syystä. Äänen viiveen keskiarvo epänormaali mittaus pois lukien on noin 97 millisekuntia.

Hieman yllättävästi äänen viive oli pienempi vaikka tämän iteraation algoritmissa on yksi säie äänidatan välissä ja äänentoistoon liittyvä äänipuskuri luodaan juuri ennen kuin ääntä voi alkaa toistamaan. Yksi syy tälle voi olla se, että lähtötilanteessa äänentoisto tapahtui Kotlin-koodista, joka käännettiin debug-tilassa, joten kääntäjän tekemiä optimointeja ei mahdollisesti ole päällä.

6.3 Toinen iteraatio: Android USB accessory -tila

Toisessa iteraatiossa oli alun perin tarkoitus vaihtaa äänidatan siirto tapahtumaan TCP-protokollan sijaan UDP-protokollalla. UDP-tuki toteuttamalla palvelinohjelma pystyisi hallitse-

maan pakettien kokoa, niin aliluvussa 6.1 huomattu äänipakettien koon vaihtelu loppuisi. UDP-tuen toteutuksen aikana huomattiin, että ADB:n kautta ei pysty käyttämään UDP-soketteja USB:n yli kuten TCP-soketteja. Täten päätettiin, että tässä iteraatiossa käsitellään-kin Android USB accessory -tilaa.

Androidin USB accessory -tilan avulla Android-sovellus voi kommunikoida USB-isäntänä toimivan laitteen kanssa, jos se noudattaa tiettyä protokollaa (Google 2022). Tietokone toimii USB-isäntänä ja mikäli käyttöjärjestelmästä löytyy USB-rajapinta, niin kyseistä tilaa on mahdollista käyttää tiedonsiirtoon. Tutkimuksen tilannetta ajatellen tämä on UDP-protokollaa parempi ratkaisu, koska USB accessory -tilaa käyttämällä palvelinohjelma pystyy lähettämään dataa suoraan USB:n yli, joten tiedonsiirto toimii palvelinohjelman näkökulmasta matalimmalla mahdollisella tasolla, mistä on luultavasti hyötyä äänen viiveen kannalta.

Palvelinohjelmaan lisättiin USB-tuki rusb-kirjastolla, joka on Rust-kääre C-ohjelmointikielellä kirjoitetulle libusb-kirjastolle. Huomionarvoista on, että palvelinohjelma käyttää libusb-kirjaston synkronista tiedonsiirtorajapintaa, koska rusb-kirjastoon ei oltu vielä toteutettu libusb:n asynkronista tiedonsiirtorajapintaa. Synkronisen tiedonsiirtorajapinnan käyttö luultavasti ei ole hyvä asia ainakaan tiedonsiirron viiveen kannalta. Libusb-kirjaston synkronisia tiedonsiirtokomentoja käytettäessä ohjelmakoodi pysähtyy odottamaan, että USB-laite on vastaanottanut tai lähettänyt tietoa (Libusb 2022).

Äänipuskurin alivuotojen ja viiveen mittaamisen kannalta äänipuskurin koko oli mahdollista asettaa minimiin, eli viivemittauksiin käytetyn puhelimen tapauksessa äänipuskurin koko oli 768 äänikehystä.

Äänen toistamiseen käytettävä algoritmi muuttui verrattuna edelliseen iteraatioon, joten se on hyvä huomioida äänen viivettä arvioidessa. Algoritmi on suurin piirtein seuraava:

1. Vaihdetaan USB accessory -tilan deskriptoria lukevan säikeen prioriteetiksi vakio `Process.THREAD_PRIORITY_AUDIO`.
2. Asiakasohjelma vastaanottaa palvelinohjelmalta äänivirran toistamiskäskyn. Yhteyksiä hallinnoiva moduuli luo viestikanavan äänivirtaa varten. Äänivirran toistavalle komponentille lähetetään käsky alkaa toistamaan viestikanavan äänivirtaa. USB accessory -tilan deskriptoria lukeva säie alkaa lähettämään USB:n yli vastaanotettua äänivirtaa

viestikanavaan.

3. Äänen toistoa hallinnoiva säie luo äänidatan lukemiseen liittyvä säikeen. Kohdan 2 viestikanavasta luetaan dataa jälkimmäisessä säikeessä.
4. Vaihdetaan äänidatan lukemiseen käytettävän säikeen prioriteetiksi vakio `Process.THREAD_PRIORITY_AUDIO`.
5. Luetaan kohdan 2 viestikanavasta äänidataa 512 äänikehyksen verran Oboen kanssa käytettävään viestikanavaan, jossa äänidata on jaettu kahdeksan äänisivun paloihin.
6. Lähetetään äänen toistoa hallinnoivalle säikeelle tieto käynnistää äänen toistaminen. Äänidatan lukeminen viestikanavasta toiseen jatkuu.
7. Oboe käynnistetään äänen toistoa hallinnoivassa säikeessä ja se luo korkean prioriteetin säikeen äänen toistamista varten. Kyseisessä säikeessä suoritetaan takaisinkutsufunktiota, jossa kirjoitetaan uutta äänidataa äänipuskuriin.
8. Luetaan takaisinkutsufunktiossa äänidataa Oboen kanssa käytettävästä viestikanavasta tarvittava määrä. Mikäli uutta äänidataa ei ole saatavilla, niin jäädään odottamaan sitä.

Edellisen algoritmin yksityiskohdat voi lukea mittauksessa käytettyjen ohjelmien lähdekoodeista, jotka löytyvät palvelinohjelman osalta palvelinohjelman git-varaston commitista `ace-329b` ja asiakasohjelman osalta asiakasohjelman git-varaston commitista `2210074`.

Äänen viiveen mittauksessa tehtiin seitsemän mittausta, jotka olivat noin 66, 1315, 66, 2480, 859, 6575 ja 62 millisekuntia. Edellisen vaiheen epänormaalin pitkiä viiveitä tapahtui tällä kertaa useampi kuin yksi. Äänen viiveen keskiarvo kolme pienintä mittausta tarkastellen on noin 65 millisekuntia.

6.4 Kolmas iteraatio: yksinkertaistamista ja hiljaisuuden toistaminen

Tässä iteraatiossa päätettiin tehdä vielä kaksi kohtuullisen helposti toteutettavaa muutosta asiakas- ja palvelinohjelmaan. Asiakasohjelman äänen toistamisen algoritmista otettiin tässä tapauksessa yksi ylimääräinen säie pois, eli äänidata menee suoraan USB accessory -tilan deskriptoria lukevasta säikeestä Oboen-takaisinkutsufunktion käytettäväksi. Lisäksi asiakas- ja palvelinohjelman väliseen protokollaan lisättiin uusi viesti, jolla asiakasohjelma voi käynnistää äänivirran palvelimelta asiakkaalle.

Edellisten muutosten seurauksena äänen toistamiseen käytettävä algoritmi muuttui verrattuna edelliseen iteraatioon, joten se on hyvä huomioida äänen viivettä arvioidessa. Algoritmi on suurin piirtein seuraava:

1. Vaihdetaan USB accessory -tilan deskriptoria lukevan säikeen prioriteetiksi vakio `Process.THREAD_PRIORITY_AUDIO`.
2. Asiakasohjelma vastaanottaa palvelinohjelmalta äänivirran toistamiskäskyn. Yhteyksiä hallinnoiva moduuli luo viestikanavan äänivirtaa varten. Äänivirran toistavalle komponentille lähetetään käsky alkaa toistamaan viestikanavan äänivirtaa. USB accessory -tilan deskriptoria lukeva säie alkaa lähettämään USB:n yli vastaanotettua äänivirtaa viestikanavaan. Palvelinohjelma ei vielä lähetä äänivirtaa.
3. Äänen toistoa hallinnoiva säie käynnistää Oboen, jolla toistetaan hiljaisuutta kunnes äänivirran ensimmäinen äänipaketti on vastaanotettu. Kohdan 2 viestikanavasta luetaan dataa Oboen korkean prioriteetin säikeessä.
4. Lähetetään yhden sekunnin kuluttua palvelinohjelmalle käsky aloittaa äänivirran lähettäminen.
5. Oboen korkean prioriteetin säikeessä käytettävä takaisinkutsufunktio havaitsee ensimmäisen äänipaketin ja vaihtaa hiljaisuuden toistamisen äänivirran toistamiseen. Yhdessä äänipaketissa on äänidataa 120 äänikehyksen verran.

Edellisen algoritmin yksityiskohdat voi lukea mittauksessa käytettyjen ohjelmien lähdekoodeista, jotka löytyvät palvelinohjelman osalta palvelinohjelman git-varaston commitista `e4a-7031` ja asiakasohjelman osalta asiakasohjelman git-varaston commitista `16910fd`.

Äänen viiveen mittausten osalta suoritettiin kaksi mittauskertaa, koska ainakaan ensimmäisellä kerralla puhelimesta kuuluvaa ääntä ei pystynyt kunnolla havaitsemaan ilman Audacityssä näkyvän äänikäyrän y-akselin suuntaista lähentämistä. Edellinen huomattiin jälkikäteen, niin tehtiin toinen mittauskerta, jolloin tietokoneesta toistettava testiääni asetettiin toistumaan stereona.

Ensimmäisellä mittauskerralla tehtiin kolme mittausta, jotka olivat noin 31, 28 ja 33 millisekuntia. Toisella mittauskerralla tehtiin kolme mittausta, jotka olivat noin 26, 31 ja 30 millisekuntia. Kaikkien mittausten keskiarvo on noin 30 millisekuntia.

Äänipuskurin alivuotoja ei tapahtunut mittausten aikana, mutta sovelluksen tämän version kehityksen aikana havaittiin, että ainakaan sovelluskehityksessä käytetyn tietokoneen kanssa äänivirta ei kulje sopivassa tahdissa puhelimelle. Äänivirran ollessa päällä pidemmän aikaa puhelimessa tapahtuu äänipuskurin alivuotoja. Alivuotoja ei saisi tapahtua, koska ne lisäävät äänen viivettä ainakin nykyisen äänentoistoalgoritmin kanssa.

Tätä versiota kaiutinsovelluksesta päätettiin kokeilla myös OnePlus 5 -puhelimella, että näkisi kuinka paljon eri puhelin voi vaikuttaa äänen viiveeseen. Pienin äänipuskurikoko kyseisellä puhelimella on 384 äänikehystä, joka on pienempi verrattuna aiemmissa mittauksissa käytettyyn Nokia 2.2 -puhelimen 768 äänikehykseen. Täten voisi kuvitella, että äänen viive olisi OnePlus 5 -puhelimella pienempi.

Sovelluksen kokeileminen kyseisen puhelimen kanssa osoitti, että äänipuskurin alivuotoja tapahtuu yleensä äänen toiston alettua. Tämän takia palvelinohjelmaan tehtiin kaksi muutosta, jotka vähensivät USB-liikennettä: palvelinohjelmasta otettiin äänivirran siirrolta aikaa vievä ping-toiminnallisuus pois käytössä ja äänivirran ollessa päällä asiakasohjelman lähettämiä JSON-viestejä ei enää lueta. Asiakasohjelman puolelle äänen toistoon käytettyyn takaisinkutsufunktioon asetettiin äänipakettien puskurointi päälle. Äänivirran toistaminen alkaa, kun kaksi äänipakettia on vastaanotettu. Edelliset muutokset aiheuttivat toivotun lopputuloksen, eli äänipuskurin alivuotoja ei yleensä enää näkynyt ainakaan heti äänentoiston alettua.

OnePlus 5 -puhelimella tehdyissä mittauksissa käytettiin palvelinohjelman osalta palvelinohjelman git-varaston commitista `2746179` löytyvää ohjelmaversiota ja asiakasohjelman osalta asiakasohjelman git-varaston commitista `d6db985` löytyvää ohjelmaversiota.

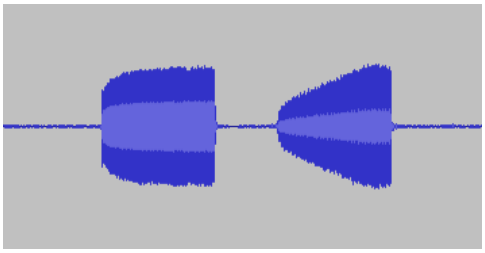
Tällä mittauskerralla äänen viiveen mittauksia tehtiin neljä, koska tapahtui yksi epänormaalin suuri mittaus. Äänen viiveet olivat noin 47, 2288, 47 ja 43 millisekuntia. Kolmen pienimmän mittauksen keskiarvo on noin 46 millisekuntia.

Edellisen mittauksen keskiarvo on suurempi kuin Nokia 2.2 -puhelimella mitattu keskiarvo. Ottaen huomioon, että OnePlus 5 -puhelimella käytetty asiakasohjelma odottaa kahta äänipakettia ennen äänivirran äänidatan kirjoitusta äänipuskuriin, niin äänen viive pitäisi äänipuskurin kokoa ajatellen olla yhteensä 576 [1] äänikehystä sillä oletuksella, että takaisin-

1. Luku 576 on `OUTPUT_FRAMES_PER_BUFFER`-vakion arvo 192 kerrottuna kolmella.

kutsufunktiota kutsutaan aina 192 äänikehyksen välein ja toinen äänipaketti saapuu toisella takaisinkutsufunktion kutsulla. Tällöin hiljaisuutta kirjoitetaan äänipuskuriin 192 äänikehyksen verran, joka lisää äänen viivettä. Takaisinkutusufunktion tarkkaa toimintaa ei kuitenkaan ole selvitetty, joten tämä on arvaus.

Edellinen arvaus ei päde ainakaan äänen viiveen mittauksia ja äänipuskurikokoja vertailtaessa. Äänen viiveen pitäisi silti olla pienempi OnePlus 5 -puhelimella, koska äänipuskuri on vielä isompi Nokia 2.2 -puhelimella. Yksi selittävä tekijä OnePlus 5 -puhelimen äänen viiveeseen voi olla, että ääntä prosessoidaan ennen sen toistamista. Mahdollinen prosessointi havaittiin, koska äänen viiveen mittausten äänimerkki on OnePlus 5 -puhelimesta kuultuna äänenvoimakkuudeltaan nouseva. Nokia 2.2 -puhelimesta äänenvoimakkuus on tasainen. Tämä voidaan havaita kuviosta 10.

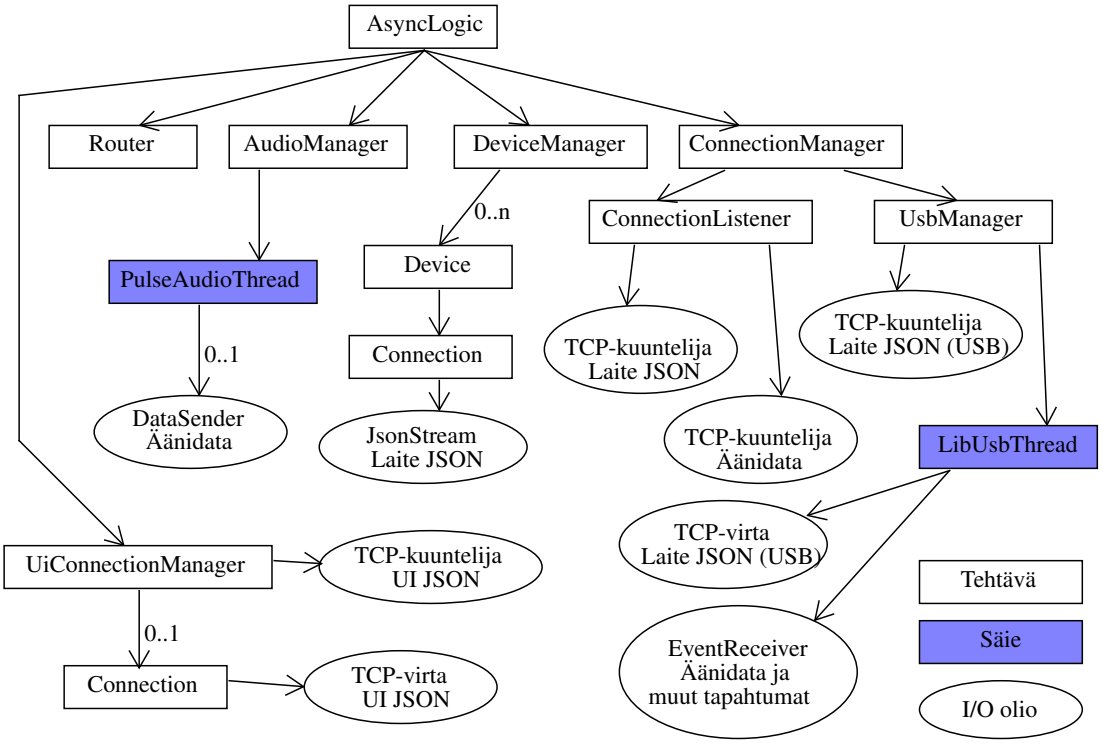


Kuvio 10. Äänimerkin äänenvoimakkuden muutos Nokia 2.2 ja OnePlus 5 -puhelimilla. Nauhoitteet molempien puhelinten kaiuttimien toistamasta äänestä on kopioitu samalle Audacityn ääniraidalle. Vasemmalla näkyy Nokia 2.2 -puhelimen tasainen äänenvoimakkuus ja oikealla OnePlus 5 -puhelimen nouseva äänenvoimakkuus. Äänimerkkien ympärillä oleva kohina voi olla jommankumman puhelimen.

6.5 Palvelinohjelman rakenne kolmannessa iteraatiossa

Kuviossa 11 nähdään palvelinohjelman komponentit tutkimuksen lopputilanteessa, eli kolmannessa iteraatiossa käytetyn ohjelman komponentit. Kyseisessä iteraatiossa käsiteltiin kahta eri palvelinohjelman versiota. Kaavio on tehty ensimmäisenä käsitellystä palvelinohjelmasta, joka löytyy palvelinohjelman git-varaston commitista `e4a7031`.

Lähtötilanteeseen verrattuna palvelinohjelmaan rakenteeseeseen on tullut uusia komponentteja, mutta ohjelman perusrakenne on ohjelman komponenttien osalta on säilynyt ennallaan.



Kuvio 11. Palvelinohjelman komponentit tutkimuksen lopputilanteessa. Kaavio on tehty suurin piirtein, joten esimerkiksi komponenttien nimet eivät välttämättä vastaa ohjelmakoodissa olevia.

Ensimmäisen iteraation (aliluku 6.2) yhteydessä tapahtuneen JSON-protokollan asiakas- ja palvelinroolien poistamisen myötä lähtötilanteen rakenteessa (aliluku 3.1) ollut Server-komponentti on uudelleen nimetty AsyncLogic-komponentiksi.

PulseAudioThread-komponentissa äänen lähettämiseen käytettävä olio on vaihtunut TCP-virrasta DataSender-rajapinnan toteuttavaksi olioksi. Kyseisen rajapinnan avulla voidaan lähettää äänidataa eikä äänen lähettävän koodin tarvitse huomioida lähetetäänkö dataa esimerkiksi soketilla.

DeviceManager-komponentin hallinnoimat laitteet käyttävät käytännössä edelleen TCP-virtaa JSON-protokollan vastaanottamiseen ja lähettämiseen. Lähdekoodin tasolla Device-komponentin alla oleva Connection-komponentti käyttää JsonStream-tyyppiä, joka kapseloi TCP-soketin. Kapseloinnin avulla muita yhteysmenetelmiä on tulevaisuudessa helpompi ottaa käyttöön Connection-komponentille, mikäli niille on tarvetta.

ConnectionManager-komponentti on uusi verrattuna lähtötilanteeseen. Se hallinnoi käytössä olevia yhteyksiä eri laitteisiin ja eri tiedonsiirtoväylillä. Kyseisen komponentin ansiosta DeviceManager-komponentissa ei tarvitse olla toiminnallisuutta kaikkien ohjelman tukemien tiedonsiirtoväylien käyttämiseen. DeviceManagerin tehtäväksi jää JSON-protokollan logiikan toteuttaminen.

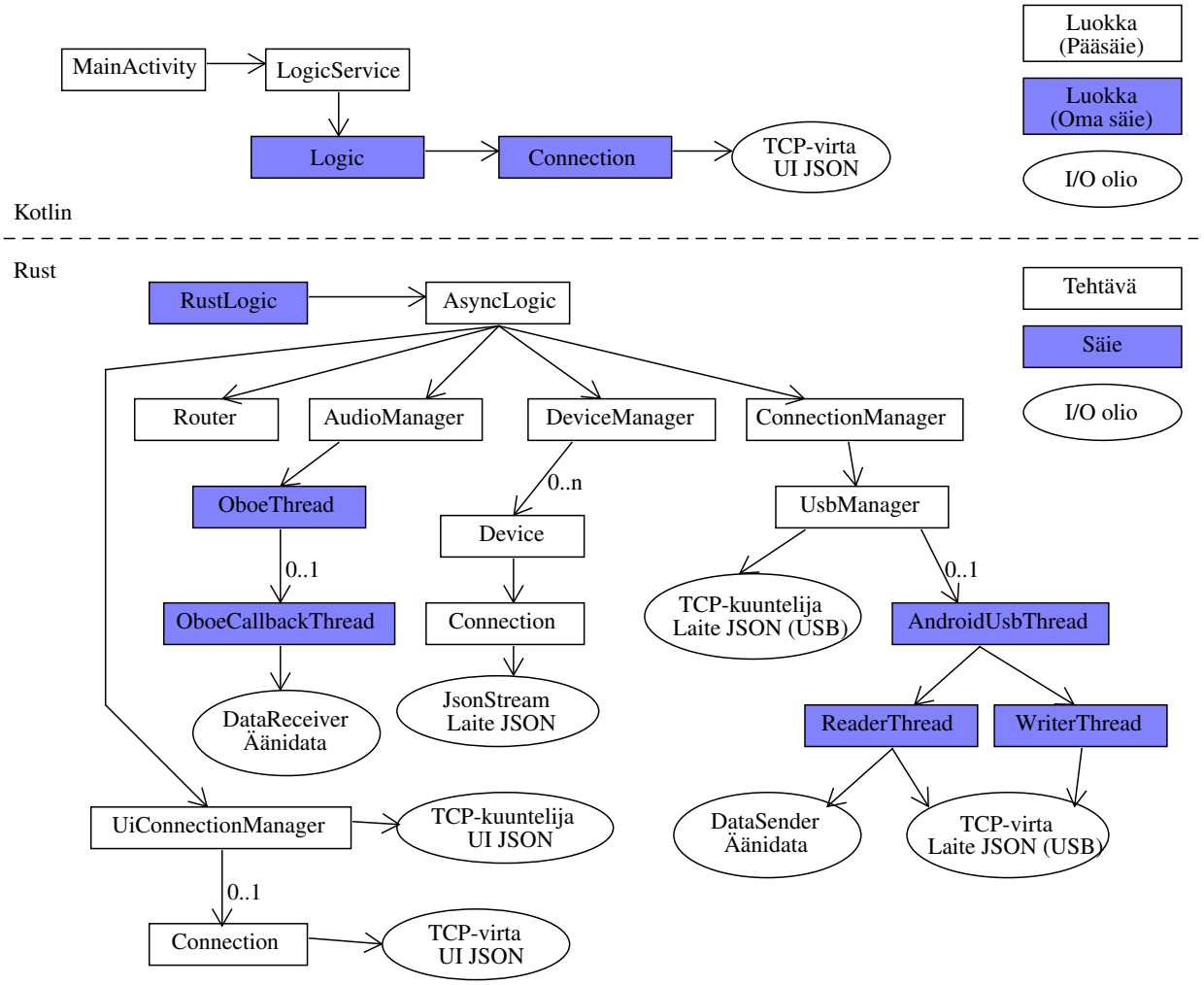
ConnectionListener-komponentti sijaitsee ConnectionManager-komponentin alla ja sisältää TCP-kuuntelijat JSON-protokollaa ja äänidataa varten, joita käytetään mikäli asiakasohjelma yhdistää palvelinohjelmaan käyttäen TCP-yhteyttä. Toinen komponentti mikä sijaitsee ConnectionManager-komponentin alla on UsbManager. Tämä komponentti toteuttaa palvelinohjelman USB-yhteensopivuuden. USB:n ollessa käytössä JSON-protokollaa siirretään palvelinohjelman sisällä TCP-yhteyden avulla, koska tällöin Connection-komponenttiin ei tarvinnut tehdä muutoksia.

UsbManager-komponentti luo säikeen, jossa ajetaan LibUsbThread-komponentin koodia. Kyseinen komponentti käyttää käyttöjärjestelmän USB-rajapintaa rusb-kirjaston avulla. Kyseisellä kirjastolla käytetään C-kielellä kirjoitettua libusb-kirjastoa Rust-ohjelmointikielestä käsin. LibUsbThread-komponentti yhdistää puhelimeen USB:n kautta. Kun yhteys laitteeseen on muodostunut, niin komponentti luo uuden TCP-virran UsbManager-komponentin TCP-kuuntelijan porttiin. USB:n yli lähetettävää äänidataa vastaanotetaan PulseAudioThread-komponentilta erilaisten tapahtumien vastaanottamiseen käytetyn viestikanavan kautta. Kyseiseen viestikanavaan lähetetään äänidataa DataSender-olion kautta.

6.6 Asiakasohjelman rakenne kolmannessa iteraatiossa

Kuviossa 12 nähdään asiakasohjelman rakenne kolmannen iteraation commitissa `16910fd`. Asiakasohjelma on muuttunut merkittävästi verrattuna lähtötilanteeseen. Aiemmin Kotlinilla kirjoitetut äänen toistaminen ja palvelinohjelman kanssa kommunikointi on kirjoitettu nyt Rustilla ja äänen toistamisessa on käytössä myös C++-ohjelmakoodia Oboe-kirjaston kanssa kommunikointiin.

Kotlin-puolella olevaa Connection-komponentia käytetään nyt Rust-ohjelmakoodin kanssa kommunikointiin ja kyseinen komponentti käynnistetään Logic-komponentin käynnistymi-



Kuvio 12. Asiakasohjelman komponentit tutkimuksen lopputilanteessa. Kaavio on tehty suurin piirtein, joten esimerkiksi komponenttien nimet eivät välttämättä vastaa ohjelmakoodissa olevia.

sen yhteydessä. Connection-komponentti käynnistää Rustilla kirjoitetun osuuden käynnistämällä RustLogic-komponentin kutsumalla Rustilla kirjoitettua funktiota. Kyseisen komponentin käynnistystä ja sammuttamista lukuun ottamatta Kotlin- ja Rust-ohjelmakoodien välinen kommunikaatio tapahtuu JSON-viestien avulla.

Asiakasohjelman Rust-koodi on enimmäkseen samaa kuin palvelinohjelmassa, mutta ääneen ja USB-tukeen liittyvät osat ovat uusia. Tällä kertaa AudioManager-komponentti toistaa ääntä tallentamisen sijasta ja toteuttaa tämän OboeThread-komponentin avulla. Kyseinen komponentti kutsuu Oboe C++-kirjastoa käyttäen asiakasohjelman koodin C++-osuutta, jossa on

äänen toistamiseen tarvittavaa koodia C-funktiokutsurajapinnalla. Rust-koodi ei suoraan kutsu Oboe-kirjaston koodia, että C++-ohjelmointikielen asiat, kuten destruktorit, toimivat oikein. Kaavion OboeCallbackThread-komponentti esittää Oboe-kirjaston käynnistämää korkean prioriteetin säiettä.

UsbManager-komponentti käyttää nyt eri koodia kuin palvelinohjelma USB-tuen toteuttamiseksi. Rust-koodi saa Kotlin-koodilta tiedon, että Android-laite on yhdistetty Accessory-tilaa tukevaan laitteeseen ja tietystä tiedostodeskriptorista voidaan lukea ja kirjoittaa laitteeseen. Tällöin AndroidUsbThread-komponentti käynnistetään. Kyseinen komponentti käynnistää vielä tiedostodeskriptoria lukevan ReaderThread-komponentin ja kirjoittavan WriterThread-komponentin. Molemmat kaksi komponenttia käyttävät samaa TCP-virtaa JSON-protokollan siirtämiseksi asiakasohjelman sisällä. ReaderThread-komponentti lähettää äänidataa Oboelle UsbDataConnectionSender-tyypin avulla. Rakennekuvassa kyseinen tyyppi on DataSender-nimellä.

7 Pohdinta

Tutkimuksessa onnistuttiin pienentämään kaiutinsovelluksen äänen viivettä. Luvussa 1 olleista äänen viiveen huomaamisen rajoista tai siihen liittyvistä suosituksista pienin esillä ollut viive oli 60 millisekuntia. Tutkimuksen kolmannessa iteraatiossa Nokia 2.2 -puhelimella päästiin noin 30 millisekuntiin, joten kyseinen tavoitearvo saavutettiin ja äänen viive on tavoitearvoa puolet pienempi.

7.1 Tutkimuksen luotettavuus

Tutkimuksen painopiste oli kaiutinsovelluksen ohjelmoinnissa, mikä näkyy tutkimuksen raportoinnissa ja sen tarkkuudessa. Tarkkoja versiotietoja mittausten aikana käytössä olleista käyttöjärjestelmistä ei otettu ylös mittausten yhteydessä. Osa mittauksista suoritettiin siten, että käytössä olevassa laitteessa on internet-yhteys. Tämä voi vaikuttaa muiden sovellusten tietojen synkronointiin ja käyttöjärjestelmäpäivityksien asentamiseen taustalla, jotka kuluttavat laitteen laskentaresursseja. Taustalla tapahtuvat asiat voivat vaikuttaa kaiutinsovelluksen äänen viiveeseen. Molemmat edellä olleet asiat vaikuttavat mittausten toistettavuuteen.

Äänen viiveen mittausmenetelmästä johtuen tutkimuksessa saadut lukuarvot äänen viiveestä ovat suuntaa antavia. Tämän voi nähdä merkittävänä haittana tutkimuksen toistettavuudelle ja luotettavuudelle. Mittausmenetelmä oli tutkimuksen tavoitteisiin nähden hyvä valinta, koska suuntaa antavat mittaustulokset riittivät ylläpitämään sovelluksen jatkokehittämistä oikeaan suuntaan.

Tutkimuksessa käytettyyn tutkimusmenetelmään ja siihen liittyvien Hevnerin ym. (2004) kirjoittamien suositusten kannalta ainakin tutkimuksen tieteellisen tarkkuuden voi edellä esitettyjen seikkojen johdosta kyseenalaistaa. Hevner ym. kuitenkin sanovat, että heidän suunnittelutieteeseen liittyviä ohjeita ei tarvitse orjallisesti noudattaa, vaan niitä voi soveltaa parhaaksi katsomallaan tavalla pitäen kuitenkin huolen, että ne ovat jotenkin huomioitu tutkimuksessa. Ottaen huomioon, että tutkimuksen prioriteetti oli sovelluksen jatkokehityksessä käyttäen apuna suuntaa antavia äänen viiveen mittauksia, niin luvussa 6 raportoitu tutkimus lienee onnistunut suunnittelutieteen näkökulmasta.

Teknisiin asioihin liittyen tutkimuksen ehkä ainoa selkeä ongelma on, että mittauksia ei tehty kokonaan release-tilassa käännetyllä asiakasohjelmassa. Tämä ongelma lienee vaikuttanut vain lähtötilanteen asiakasohjelman äänen viiveeseen, koska Kotlin-koodi käännetään debug-tilassa ja äänen toistoon liittyvä koodi on lähtötilanteessa kirjoitettu sillä.

Mikäli tutkimuksen mittausmenetelmässä olisi käytetty monikanavanauhuria, kuten joissain luvussa 5.1 kerrotuissa tutkimuksissa, niin luultavasti äänen viive olisi ollut helpompi selvittää nauhoituksista.

7.2 Jatkokehitys- ja jatkotutkimusideat

Tutkimuksessa ollut OnePlus 5 -puhelimella suoritettujen testien perusteella selvisi, että sovellus ei käyttäydy kaikilla puhelimilla samaan tyyliin kuin Nokia 2.2 -puhelimella. Puhelimet voivat erota esimerkiksi Android-version ja natiivien äänipuskurikokojen suhteen. Tällöin sovelluksen olisi hyvä olla dynaamisesti konfiguroitavissa. Tutkimuksessa olleessa sovelluksessa on olemassa laitteen natiivin näytteistystaajuuden kysely palvelimelta asiakkaalle, mutta tätä pitäisi laajentaa sisältämään laitteen natiivin puskurikokoon. Lisäksi palvelimen pitäisi pystyä määrittelemään asiakkaan käyttämät äänipuskurikoot.

Palvelinohjelmaan ehdittiin tutkimuksen aikana toteuttaa tiedonsiirto USB:n kautta käyttäen libusb-kirjaston synkronista rajapintaa. Jatkokehityksessä palvelinohjelmaan kannattaa vaihtaa käyttöön kyseisen kirjaston asynkroninen rajapinta, niin äänidatan lähetyksessä ei tällöin jäädä odottamaan lähetyksen valmistumista. Tämä voi vaikuttaa äänen viiveeseen jonkin verran, koska odottaminen voi vaikuttaa asiakkaan äänipuskurin täyttymisen nopeuteen.

Tutkimuksen mittauksissa havaittiin erikoisen pitkiä viiveitä. Jatkokehityksessä olisi hyvä selvittää, että mistä viiveet johtuvat. Onko ongelma esimerkiksi pelkästään palvelinohjelmassa? Käynnistyykö palvelinohjelman äänidatan tiedonsiirto USB:n kautta välillä liian myöhään? Ongelman pitäisi selvittää ja korjata viimeistään ennen sovelluksen julkaisemista.

Jatkokehityksen aikana äänen viiveen mittaaminen on edelleen oleellista, että tiedetään aiheuttaako ohjelman muutokset viiveen kasvamista tai pienenemistä vai pysyykö viive samana. Tutkimuksessa käytössä olleeseen mittausmenetelmään kannattaa vaihtaa ääntä nauhoit-

tavan puhelimen tilalle luvussa 5.1 esillä ollut monikanavanauhuri, johon syötetään ääni johtoja pitkin. Tällöin mittaustulosten tulkinta silmämääräisesti on tarkempaa kuin nykyisellä menetelmällä.

Äänen viiveen mittausten automatisointia olisi hyvä harkita. Mikäli automatisointi tehtäisiin, niin tällöin ohjelmointiin voisi käyttää mahdollisesti enemmän työaikaa pitkässä juoksussa. Tämä riippuu siitä kuinka paljon käytännössä viiveen mittauksia on tarpeellista tehdä. Suurin haaste automatisoinnissa lienee viiveen laskeminen mittauksesta. Mikäli mittausmenetelmää päivitetään edellisen kappaleen mukaisesti monikanavanauhurilla, niin viiveen automaattinen laskeminen lienee helppoa. Tietokone ja puhelin toistavat molemmat aluksi hiljaisuutta, minkä avulla on helppo tunnistaa se hetki, jolloin ääninäyte alkaa. Molemmilta nauhoitteen kanavilta tunnistetaan kyseinen hetki ja lasketaan viive.

Tutkimuksen aikana tehtiin TCP-pakettikaappaus. USB:n yli tietoa siirtäessä se kulkee paketeissa. Tutkimuksessa olisi voinut tehdä USB-pakettikaappauksen, jolloin ehkä oltaisiin saatu lisätietoa esimerkiksi miksi epänormaalin pitkiä äänen viiveitä tapahtuu. Lisäksi TCP-pakettikaappausta oltaisiin voitu verrata USB-pakettikaappaukseen. Mikäli jatkokehityksessä libusb:n asynkroninen rajapinta otetaan käyttöön palvelinohjelmassa, niin USB-pakettikaappauksen avulla voitaisiin nähdä pieneneekö pakettien välinen viive verrattuna tutkimuksen lopputilanteeseen.

Tässä tutkielmassa tehdyissä mittauksissa mitattiin tietokoneen kaiuttimista ja puhelimen kaiuttimista ulostulevan äänen havainnoitava aikaero. Asiakas- ja palvelinohjelmaan olisi mahdollista lisätä myös joidenkin komponenttien välille ajan mittausta. Esimerkiksi asiakasohjelmassa voitaisiin mitata kuinka tasaisin väliajoin ääntä toistavaa takaisinkutsufunktiota kutsutaan. Palvelinohjelmassa taas voitaisiin esimerkiksi mitata kuinka tasaisin väliajoin äänidata liikkuu ääntä nauhoittavasta komponentista USB-yhteyttä hallinnoivalle komponentille.

Tutkimuksessa keskityttiin vain langalliseen tiedonsiirtoon, koska oletettavasti viive on helpoin saada mahdollisimman pieneksi langallisella tiedonsiirrolla. Ohjelmaa olisi kuitenkin hyvä pystyä käyttämään myös ilman johtoja, joten jatkotutkimuksessa voisi kokeilla, miten tiedon siirtäminen WLAN-yhteydellä käyttäen TCP:tä ja UDP:tä toimii luotettavuuden

ja viiveen kannalta. Lisäksi äänen pakkaamisen vaikutusta kahteen edelliseen kannattanee myös tutkia.

Mikäli TCP vaikuttaa hyvältä ratkaisulta esimerkiksi langatonta tiedonsiirtoa ajatellen, niin tällöin jatkotutkimuksessa kannattanee tarkastella myös kaiutinsovelluksen toimintaa erilaisten TCP-ruuhkanhallinta-algoritmien kanssa. Näistä algoritmeista on kerrottu esimerkiksi Afanasyevin ym. (2010) artikkelissa. Lisäksi voisi tutkia kaiutinsovelluksen käyttäytymistä silloin kun TCP:n kanssa käytetään ECN-ominaisuutta.

8 Yhteenveto

Tässä tutkimuksessa jatkokehitettiin kaiutinsovelluksen prototyyppiä käyttäen suunnittelutieteelle tyypillistä iteratiivista prosessia. Tämän tutkimuksen tapauksessa jokaisen iteraation lopussa arvioitiin kaiutinsovelluksen äänen viivettä. Tutkimusta tehtiin kolmen iteraation verran.

Kaiutinsovelluksen prototyypissä tietokoneessa toimiva palvelinohjelma tarjosi puhelimessa toimivalle asiakasohjelmalle äänivirtaa TCP-paketteina, jotka kulkivat langallisesti USB:n yli. Kaiutinsovellusta olisi ollut mahdollista käyttää myös WLAN-yhteyden yli, mutta USB:n käyttäminen tiedonsiirtoon nähtiin tutkimuksen tavoitteihin nähden parempana vaihtoehtona USB:n langallisuuden takia.

Kaiutinsovelluksen prototyypissä olevia mahdollisia ongelmia liittyen äänen viiveeseen käsiteltiin luvussa 4. Mahdollisiksi ongelmiksi havaittiin muun muassa äänipuskurien koko, käytössä oleva äänirajapinta ja TCP-protokolla.

Äänen viiveen mittausmenetelmistä monikanavanauhurimenetelmä vaikuttaa olevan yleisesti käytössä oleva menetelmä, kun se tai siltä muistuttavia mittausmenetelmiä esiintyi useammassa kuin yhdessä tutkimuksessa. Tässä tutkimuksessa käytettiin mittausmenetelmää, jossa kahden laitteen kaiuttimista nauhoitettiin ääntä puhelimen mikrofonin avulla.

Tutkimuksen aikana asiakas- ja palvelinohjelmaan toteutettiin tuki siirtää tietoa USB:llä ilman asiakas- ja palvelinohjelman välillä olevaa TCP-yhteyttä. Tämän ja muiden parannusten avulla kaiutinsovelluksen prototyypin noin 135 millisekunnin mittainen äänen viive saatiin pienennettyä noin 30 millisekuntiin.

Tutkimuksen onnistumiselle asetettiin luvussa 1 käytännössä huomaamaton äänen viive. Tähän tavoitteeseen päästiin, mikäli käytännössä huomaamattomana äänen viiveenä pidetään esimerkiksi Euroopan yleisradiounionin (2007) suositusta äänen viiveelle. Kyseisessä suosituksessa äänen viiveen pitäisi olla 60 millisekuntia tai vähemmän. Tutkimuksen tutkimuskysymyksenä oli, että kuinka lähelle kaiutinsovelluksella päästään normaalin kaiuttimen äänen viivettä. Tämän tutkimuksen perusteella voi sanoa, että ainakin noin 30 millisekunnin päähän

normaalista kaiuttimesta.

Jatkotutkimuksessa voisi muun muassa uusia tämän tutkimuksen äänen viiveen mittauksia käyttäen mittausmenetelmänä monikanavanauhurimenetelmää ja tehdä tutkimuksen, jossa tutkittaisiin huomaavatko koehenkilöt äänen viivettä kaiutinsovellusta käytettäessä. Kaiutinsovelluksen jatkokehityksessä olisi hyvä muun muassa korjata kaksi bugia, jotka havaittiin tämän tutkimuksen aikana. Nämä olivat epätavallisen pituiset äänen viiveet ja äänen viiveen kasvaminen hiljalleen ajan myötä.

Lähteet

Afanasyev, Alexander, Neil Tilley, Peter Reiher ja Leonard Kleinrock. 2010. “Host-to-Host Congestion Control for TCP”. *IEEE Communications Surveys & Tutorials* 12 (3): 304–342. doi:10.1109/SURV.2010.042710.00114.

Balsini, Alessio, Tommaso Cucinotta, Luca Abeni, Joel Fernandes, Phil Burk, Patrick Bellasi ja Morten Rasmussen. 2019. “Energy-efficient low-latency audio on android”. *Journal of Systems and Software* 152:182–195. ISSN: 0164-1212. doi:https://doi.org/10.1016/j.jss.2019.03.013. https://www.sciencedirect.com/science/article/pii/S0164121219300585.

Blanton, Ethan, Dr. Vern Paxson ja Mark Allman. 2009. *TCP Congestion Control.* RFC 5681, syyskuu. doi:10.17487/RFC5681. https://www.rfc-editor.org/info/rfc5681.

Bouillot, Nicolas, ja Jeremy R. Cooperstock. 2009. “Challenges and Performance of High-Fidelity Audio Streaming for Interactive Performances”. Teoksessa *New Interfaces for Musical Expression (NIME).* Pittsburgh, PA, USA, kesäkuu. http://srl.mcgill.ca/publications/2009-NIME.pdf.

Brandt, Eli, ja Roger B Dannenberg. 1998. “Low-Latency Music Software Using Off-The-Shelf Operating Systems”. San Francisco: International Computer Music Association, *Proceedings of the International Computer Music Conference.* doi:10.1184/R1/6607067.v1. https://kilthub.cmu.edu/articles/journal_contribution/Low-Latency_Music_Software_Using_Off-The-Shelf_Operating_Systems/6607067.

EBU. 2007. *The relative timing of the sound and vision components of a television signal.* Recommendation R37-2007. European Broadcasting Union. Viitattu 11. helmikuuta 2022. https://tech.ebu.ch/docs/r/r037.pdf.

Fairhurst, Gorry, Brian Trammell ja Mirja Kühlewind. 2017. *Services Provided by IETF Transport Protocols and Congestion Control Mechanisms.* RFC 8095, maaliskuu. doi:10.17487/RFC8095. https://www.rfc-editor.org/info/rfc8095.

Goel, Ashvin, Charles Krasic ja Jonathan Walpole. 2008. "Low-Latency Adaptive Streaming over Tcp". *ACM Trans. Multimedia Comput. Commun. Appl.* (New York, NY, USA) 4, numero 3 (syyskuu). ISSN: 1551-6857. doi:10.1145/1386109.1386113.

Google. 2022. *Android 11 Compatibility Definition.* Google. Viitattu 23. helmikuuta. https://source.android.com/compatibility/11/android-11-cdd.

———. 2022. *Android API Reference: AudioManager.* Google. Viitattu 7. huhtikuuta. https://developer.android.com/reference/android/media/AudioManager.

———. 2022. *Audio latency.* Google. Viitattu 23. helmikuuta. https://developer.android.com/ndk/guides/audio/audio-latency.

———. 2022. *USB accessory overview.* Google. Viitattu 15. huhtikuuta. https://developer.android.com/guide/topics/connectivity/usb/accessory.

———. 2023. *Introduction to activities.* Google. Viitattu 8. tammikuuta. https://developer.android.com/guide/components/activities/intro-activities.

———. 2023. *Measuring Audio Latency.* Google. Viitattu 10. helmikuuta. https://source.android.com/devices/audio/latency/measure.

———. 2023. *Services overview.* Google. Viitattu 13. tammikuuta. https://developer.android.com/guide/components/services.

Hevner, Alan R., Salvatore T. March, Jinsoo Park ja Sudha Ram. 2004. "Design Science in Information Systems Research". *MIS Quarterly* 28 (1): 75–105. ISSN: 02767783. http://www.jstor.org/stable/25148625.

ITU-R. 1998. *Relative timing of sound and vision for broadcasting.* Recommendation ITU-R BT.1359-1. ITU Radiocommunication Sector. Viitattu 11. helmikuuta 2022. https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.1359-1-199811-I!!PDF-E.pdf.

Krasic, Charles, Kang Li ja Jonathan Walpole. 2001. "The Case for Streaming Multimedia with TCP". Teoksessa *Interactive Distributed Multimedia Systems,* toimittanut Doug Shepherd, Joe Finney, Laurent Mathy ja Nicholas Race, 213–218. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-540-44763-4. doi:10.1007/3-540-44763-6_22.

Libusb. 2022. *Synchronous and asynchronous device I/O.* Libusb. Viitattu 15. huhtikuuta. https://libusb.sourceforge.io/api-1.0/libusb_io.html.

MacMillan, Karl, Michael Droettboom ja Ichiro Fujinaga. 2001. "Audio Latency Measurements of Desktop Operating Systems". Teoksessa *ICMC.* http://hdl.handle.net/2027/spo.bbp2372.2001.055.

Pentikousis, K., H. Badr ja B. Kharmah. 2002. "On the performance gains of TCP with ECN". Teoksessa *2nd European Conference on Universal Multiservice Networks. ECUMN'2001 (Cat. No.02EX563),* 82–91. doi:10.1109/ECUMN.2002.1002092.

Perneel, Luc, Hasan Fayyad-Kazan ja Martin Timmerman. 2012. "Can Android be used for real-time purposes?" Teoksessa *2012 International Conference on Computer Systems and Industrial Informatics,* 1–6. doi:10.1109/ICCSII.2012.6454350.

Rocha, Cecilia G. da, Carlos T. Formoso, Patricia Tzortzopoulos-Fazenda, Lauri Koskela ja Algan Tezel. 2012. "Design Science Research in Lean Construction: Process and Outcomes" [kielellä English]. Teoksessa *20th Annual Conference of the International Group for Lean Construction,* toimittanut Iris D. Tommelein ja Chrisitne L. Pasquire. San Diego, California, USA. http://www.iglc.net/papers/details/770.

Steinmetz, R. 1996. "Human perception of jitter and media synchronization". *IEEE Journal on Selected Areas in Communications* 14 (1): 61–72. doi:10.1109/49.481694.

Valin, Jean-Marc, Gregory Maxwell, Timothy B. Terriberry ja Koen Vos. 2016. *High-Quality, Low-Delay Music Coding in the Opus Codec.* arXiv: 1602.04845 [cs.MM].

Wang, Y., R. Stables ja J. Reiss. 2010. “Audio latency measurement for desktop operating systems with onboard soundcards” [kielellä English]. Teoksessa *128th Audio Engineering Society Convention 2010,* 2:707–716. http://www.open-access.bcu.ac.uk/2783/.

Wright, Matthew, Ryan J. Cassidy ja Michael Zbyszynski. 2004. “Audio and Gesture Latency Measurements on Linux and OSX”. Teoksessa *Proceedings of the 2004 International Computer Music Conference, ICMC 2004, Miami, Florida, USA, November 1-6, 2004*. Michigan Publishing. https://hdl.handle.net/2027/spo.bbp2372.2004.159.

Yadav, Radhakishan, ja Robin Singh Bhadoria. 2015. “Performance Analysis for Android Runtime Environment”. Teoksessa *2015 Fifth International Conference on Communication Systems and Network Technologies,* 1076–1079. doi:10.1109/CSNT.2015.52.

Ye, Dongsheng, Juanjuan He, Wei Hu ja Jing Liu. 2018. “Measurement and analysis on audio latency for multiple operating systems”. Teoksessa *2018 13th IEEE Conference on Industrial Electronics and Applications (ICIEA),* 2496–2500. doi:10.1109/ICIEA.2018.8398130.

Yegulalp, Serdar. 2021. “What is Rust? Safe, fast, and easy software development”. Viitattu 31. joulukuuta 2022. https://www.infoworld.com/article/3218074/what-is-rust-safe-fast-and-easy-software-development.html.

Zigunovs, Dmitrijs, Jekaterina Smirnova, Gatis Vitols ja Gintautas Stonys. 2017. “Solution for Sound Playback Delay on Android Devices”. ICTE 2016, Riga Technical University, Latvia, *Procedia Computer Science* 104:413–420. ISSN: 1877-0509. doi:10.1016/j.procs.2017.01.154. https://www.sciencedirect.com/science/article/pii/S1877050917301552.