Tampereen yliopisto

Ekku Laukkarinen

DATAN SYNKRONOINTI REAALIAIKAISEN VIDEON SUORATOISTOON

Suoratoistoprotokollasta riippumaton datan synkronointi

Informaatioteknologian ja viestinnän tiedekunta
Diplomityö
Toukokuu 2019

TIIVISTELMÄ

Ekku Laukkarinen: Datan synkronointi reaaliaikaisen videon suoratoistoon
Diplomityö
Tampereen yliopisto
Tietotekniikka
Toukokuu 2019

Suoratoistopalveluiden räjähdysmäisestä kasvusta on seurannut suoratoisto- ja koodekkiteknologioiden nopea kehitys. Selainympäristö ei ole pysynyt tämän nopean kehityksen mukana. Tarjotakseen suoratoistoa kaikilla yleisesti käytetyillä selaimilla, sovelluskehittäjän on tuettava useampaa teknologiaa. Kasvava suoratoistopalveluiden kysyntä on aiheuttanut useita tarpeita multimediaratkaisuihin. Multimediassa on erityisen tärkeää, että kaikki datalähteet on synkronoitu. Tällaisen synkronoinnin toteuttaminen kaikille suoratoistoteknologioille on kallista ja teknologioihin perustuvien ratkaisujen käyttäminen estää uusien suoratoisto- ja koodausteknologioiden käyttöönoton.

Tässä työssä esitetään median synkronointiin toteutus, joka ei ole riippuvainen suoratoisto- tai koodausteknologiasta. Toteutus pohjautuu äänisignaalin avulla dataa siirtäviin järjestelmiin. Suoratoistettavaan videon äänisignaaliin lisätään reaaliaikaisia aikaleimoja sisältävä äänisignaali taajuusavainnuksen avulla. Taajuusavainnuksessa käytetään heksadesimaalimerkkejä ja 16 niitä vastaavia eri taajuutta. Aikaleimoja tunnistetaan HTML5-standardin määrittämän rajapinnan avulla, joka on toteutettu kaikissa yleisimmissä selaimissa. Aikaleimojen tunnistuksessa käytetään nopeaa Fourierin muunnos -algoritmia, jolla signaali saadaan muutettua taajuustasoon. Taajuuksien tunnistamisessa käytetään hyvin häiriötä sietävää tapaa. Tämä takaa, ettei äänen koodausprosessilla ole merkitystä järjestelmän toimiseen, kunhan äänenlaatu säilytetään riittävänä.

Tunnistettujen aikaleimojen avulla JavaScript-komponentti voi luoda aikajanan, joka mahdollistaa useamman datalähteen synkronoinnin tarkasti ja tehokkaasti. Toteutus evaluoitiin toimivaksi ja sen käyttämät parametrit optimoitiin käyttäen tarjolla olevia avoimia työkaluja.

Avainsanat: suoratoisto, multimedia, synkronointi, reaaliaikainen, protokolla-riippumattomuus

ABSTRACT

Ekku Laukkarinen: Real-time data synchronization in streaming
Master of Science Thesis
Tampere University
The Computing Sciences
May 2019

The amount of video streaming services has drastically increased during the past few years. New streaming and encoding technologies have been implemented to improve the quality of the streaming services. Even though new HTML-standards are introduced rapidly, the standards have failed to support all of the new technologies. This results in a varying support of protocols on browsers of different vendors. Supporting all major browsers require the application developer to support multiple technologies. Supporting multiple technologies is costly.

The increase in streaming services has also resulted in a need to synchronize secondary data streams to the video. This is called multimedia. The existing methods to the synchronization of multiple data streams are bound to the selected streaming technology. Existing implementations of the synchronization make it costly to change a streaming or encoding technology.

This thesis presents a synchronization method using a common factor for all streaming and encoding protocols. The metadata required for the synchronization is embedded to the audio signal. The presented method uses a frequency modulation to include a timestamp in the audio signal. These modulated timestamps can survive an encoding process. The HTML5-standard defines an API that offers a possibility to modify and analyze audio after the decoding process. Using this API, the embedded frequencies are converted to frequency domain using Fast Fourier Transform. The frequency domain is analyzed for peak frequencies. Each frequency corresponds to a hexadecimal character. The detected hexadecimal characters form a timestamp, that can be used to synchronize a timeline. This timeline can then be used to provide synchronization for all data streams. The implementation is evaluated to be accurate using an open-source tools.

Keywords: streaming, multimedia, synchronization, real-time, protocol independent

ALKUSANAT

Ensimmäisenä haluan kiittää diplomityöni tarkastajia Hannu-Matti Järvistä ja Jarno Vannetta joustavasta aikataulusta ja rakentavista kommenteista. Kiitokset myös Adalia Oy:lle diplomityön aiheesta ja mahdollisuudesta työskennellä mielenkiintoisen projektin parissa. Lopuksi haluan kiittää avopuolisoani, perhettäni ja ystäviäni tukemisesta tämän diplomityön ja muiden opintojen aikana.

Tampereella, 22. toukokuuta 2019

Ekku Laukkarinen

SISÄLLYSLUETTELO

KUVALUETTELO

TAULUKKOLUETTELO

LYHENTEET JA MERKINNÄT

AAC	Suoratoistossa yleisesti käytetty äänikoodekki (engl. Advanced Audio Coding)
AVC	Suoratoistossa yleisesti käytetty videokoodekki (engl. Advanced Video Coding)
DASH	HTTP-protokollaan pohjautuva suoratoistoprotokolla (engl. Dynamic Adaptive Streaming over HTTP)
FM	Taajuusmodulaatio (engl. Frequency Modulation)
FSK	Taajuusavainnus (engl. Frequency Shift Keying)
HEVC	Suoratoistossa käytetty moderni videokoodekki (engl. High Efficiency Video Coding)
HLS	HTTP-protokollaan pohjautuva suoratoistoprotokolla (engl. HTTP Live Streaming)
HTML	Verkkoselaimien käyttämä kuvauskieli (engl. HyperText Markup Language)
HTTP	Hypertekstin siirtoprotokolla (engl. Hypertext Transfer Protocol)
ISO	Kansainvälinen standardointiorganisaatio
W3C	HTML-standardeja hallinnoiva järjestö (engl. World Wide Web Consortium)

1 JOHDANTO

Kasvaneet tiedonsiirtonopeudet ovat mahdollistaneet suoratoistopalveluiden räjähdysmäisen kasvun. Kuluttajat katsovat videoita suoratoistopalveluista jatkuvasti kasvavissa määrin. Videon lisäksi on syntynyt erilaisia tarpeita synkronoida suoratoistolähetyksiin erinäistä dataa. Synkronoinnilla voidaan esimerkiksi mahdollistaa käyttäjän interaktiivinen kokemus. Tätä useamman medialähteen kokonaisuutta kutsutaan multimediaksi.

Multimedian toistaminen selainympäristössä on pitkään vaatinut kolmannen osapuolen liitännäisiä. Viime vuosina ongelmaa on pyritty korjaaman määrittelemällä multimedialle standardisoituja rajapintoja. Nämä standardit eivät ole kyenneet vastaamaan suoratoiston nopeasti muuttuviin vaatimuksiin ja eri yritykset ovat päätyneet tarjoamaan omia teknologioitaan vaihtelevalla menestyksellä. Selainympäristöjen suoratoisto-menetelmien hajanainen tuki pakottaa suoratoistopalveluiden kehittäjät ylläpitämään tukea useammalle protokollalle. Eri lähteiden synkronointi katselijaa varten on useasti toteutettu, joko suoratoisto- tai koodaus-menetelästä riippuvaisesti.

Tämä diplomityö on konstruktiivinen ja diplomityön tavoitteena on selvittää, onko mahdollista kehittää multimedian synkronointiratkaisu, joka ei ole riippuvainen käytettävistä suoratoistoprotokollista ja sietää erilaisia multimediaan tehtyjä prosesseja, kuten koodausta ja transkoodausta. Ratkaisun on myöskin tuettava yleisimpiä selaimia ilman erillisiä liitännäisiä. Työn teknisenä toteutuksena syntyi JavaScript-kirjasto, jonka avulla usea datalähde voidaan synkronoida videodataan.

Luvussa 2 käydään läpi olemassa olevia toteutuksia, jotka ohjasivat suunnitteluprosessia. Toteutuksiin käytettyjä menetelmiä tarkastellaan teoriapohjaisesti. Luvussa myöskin käydään läpi minkälaisia toteutukseen liittyviä teknologioita selainympäristö mahdollistaa ja toisaalta mitä rajoituksia se asettaa. Lopuksi käydään läpi olemassa olevat toteutukset suoratoiston synkronointiin ja tarkastellaan niiden ongelmia.

Luvussa 3 käydään läpi järjestelmän suunnittelun lähtökohdat eli minkälaisiin vaatimuksiin järjestelmän on kyettävä vastaamaan. Luvussa käydään läpi, miten aiemmin esiteltyjä menetelmiä voidaan hyödyntää. Luvussa kuvataan myös toteutuksen arkkitehtuuri komponenttitasolla ja esitellään työn toteutettu JavaScript-kirjasto tarkemmin. Toteutus käydään yksityiskohtaisesti läpi ennen seuraavassa luvussa esitettyä prototyypin evaluointia, jotta lukijalla on selkeä kuva järjestelmän toiminnasta.

Luvussa 4 käydään läpi, kuinka toteutusta on evaluoitu suunnitteluvaiheessa. Luvussa myös kuvataan minkälaisilla, prosesseilla on kyetty todentamaan järjestelmän toimivuus.

Järjestelmää myös evaluoidaan yleisesti käytettyjä koodausmenetelmiä vastaan.

Luvussa 5 käydään läpi evaluaation tulokset, arvioidaan toteutuksen toimivuutta ja käydään läpi järjestelmän parannusmahdollisuuksia. Luvussa käydään myös läpi, kuinka onnistuneesti työn toteutus täyttää vaatimukset. Luvussa 6 kootaan yhteen työn tavoitteet ja esitetään työn toteutus tiivistettynä yleisellä tasolla.

2 TAPAHTUMAPOHJAISEN DATAN SYNKRONOINTI

Datan synkronoinnissa on pääjärjestelmä, johon liitetyt sekundäärijärjestelmät synkronoidaan. Tapahtuma-pohjaisella synkronoinnilla tarkoitetaan sekundäärijärjestelmien synkronointia vain tietyn tapahtuman hetkellä. Tapahtuma voi olla esimerkiksi laitteelta tuleva signaali, jonka sekundäärijärjestelmä huomaa. Aliluvussa 2.1 käydään läpi datan synkronoinnin toteutuksia perinteisessä mediassa. Aliluvussa 2.2 käydään läpi mitä haasteita ja mahdollisuuksia selainympäristö tarjoaa synkronoinnin toteutukseen.

2.1 Käyttö perinteisessä mediassa

Vuosien saatossa interaktiivisuus televisiolähetyksissä on kasvanut. Interaktiivisuuden saavuttamiseksi käyttäjä on tarvittu katsomaan tai kuuntelemaan lähetystä, ja välittämään tieto laitteelle, jolla interaktio tapahtuu. Käyttäjää tarvittiin syöttämään tarvittava metatieto, esimerkiksi puhelinnumero, jotta tieto saatiin lähetyksen tuottajalle.

Mobiililaitteiden yleistyessä erilaiset internettiä hyödyntävät mobiilisovellukset ovat mahdollistaneet suoran interaktiivisuuden lähetyksen kanssa. Näissä ratkaisuissa televisiolähetyksen viive on tiedossa ja lähes tulkoon vakio. Tämä mahdollistaa mobiililaitteella ennalta määritetyn viiveen tuottamisen, jotta käyttäjästä tuntuisi kuin sovelluksessa tapahtuvat muutokset on synkronoitu televisiolähetyksen kanssa.

Kyseisellä ratkaisulla ei voida taata tapahtumien synkronisuutta. Tähän ongelmaan on esitetty erilaisia ratkaisuja [4]. Useasti ehdotettu menetelmä on piilottaa äänisignaaliin tarvittava metatieto, jotta lähetyksen viive saadaan tietoon ja tapahtumat synkronoitua. Mobiililaite kuuntelee äänisignaalia ja tunnistaa siihen sisällytetyn signaalin. Tunnistettua signaalia vastaava tieto noudetaan palvelimelta ja näytetään käyttäjän mobiililaitteella.

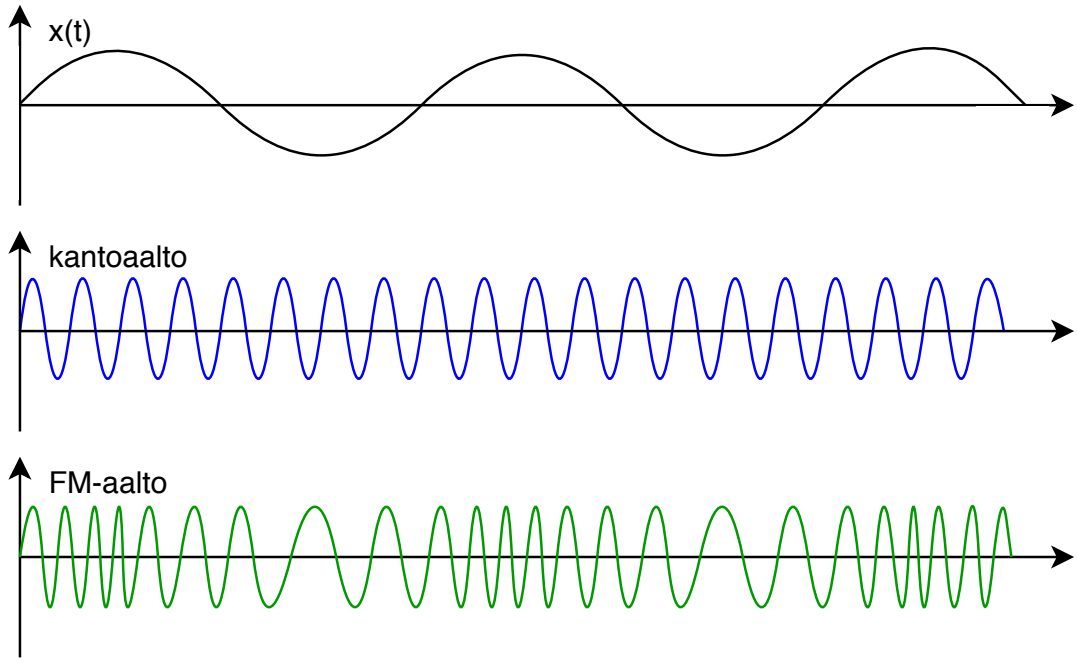
2.1.1 Metadatan sisällytys äänisignaaliin

Äänisignaalia datan siirrossa on käytetty useissa kaupallisissa sovelluksissa [5]. Äänisignaalin etuna muihin signaaleihin verrattuna on komponenttien halpa hinta ja yhteensopivuus olemassa oleville laitteille. Laite ei tarvitse erillistä signaalipiiriä, vaan mikrofoni ja kaiutin riittävät.

Äänisignaaliin voidaan sisällyttää dataa samaan tapaan kuin mihin tahansa signaaliin.

Yleisesti signaalinkäsittelyssä käytetty tapa on modulaatio. Modulaatiossa kantosignaaliin liitetään toinen dataa sisältävä signaali. Vastaanottajalla on tiedossa kantosignaalin parametrit, jolloin yhdistetystä signaalista voidaan erottaa kantosignaali ja alkuperäinen datasignaali. Tätä prosessia kutsutaan demodulaatioksi.

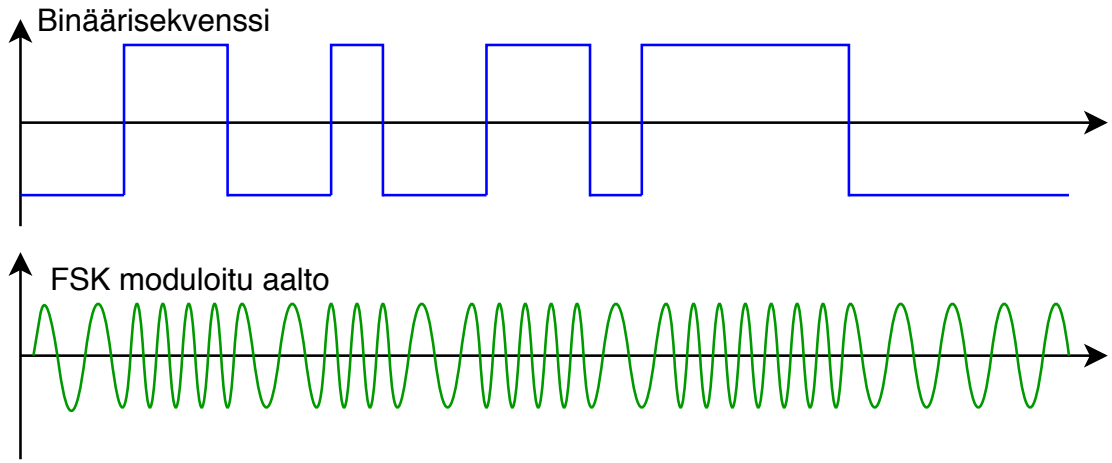
Modulaation tekemiseen on useita menetelmiä. Yksi menetelmä on taajuusmodulaatio (engl. frequency modulation). Taajuusmodulaatio on häiriötä hyvin kestävä menetelmä [24]. Taajuusmodulaatiossa kantoaallon taajuus vaihtelee kapealla taajuusalueella. Taajuusmodulaatio on käytössä FM-radioissa. Kuva 2.1 havainnollistaa taajuusmodulaatiota. Kantosignaaliin, jonka taajuus FM-lähetyksissä on tyypillisesti 87.5-108.0 MHz, liitetään ajasta riippuva äänisignaali x(t). Radiovastaanotin demoduloi signaalista kantosignaalin ja toistaa jäljelle jääneen äänisignaalin.



Kuva 2.1. *Taajuusmodulaation hyödyntäminen FM-radioissa.*

Taajuusmodulaatiota voidaan käyttää myös digitaalisen signaalin kuljettamiseen. Menetelmää kutsutaan taajuusavainnukseksi ja siitä käytetään yleisesti lyhennettä FSK (engl. Frequency Shift Keying). FSK-menetelmässä binäärisignaali moduloidaan kantosignaaliin. Kuva 2.2. havainnollistaa binäärisignaalin modulointia. Signaalin taajuus muuttuu, riippuen onko binäärisekvenssissä yksi vai nolla.

FSK-modulaatio on yleisesti käytetty menetelmä datan siirrossa äänisignaalien avulla [5]. Äänen käyttö datansiirrossa on havaittavissa ihmiskorvalla. Ongelmaa on pyritty ratkaisemaan erilaisilla menetelmillä, jotka käyttävät hyödyksi ihmisen kuuloaistin ominaisuuksia. Erilaisten tutkimusten perusteella ihmisen on mahdollista kuulla äänet 20-20 000 Hz taajuusalueella. Ihmisen vanhetessa taajuusalueen maksimi on keskimäärin 12-14 kHz. Taajuusaluetta ihmisen kuuloaistin yläpuolella kutsutaan ultraääneksi. Datasignaali voidaan piilottaa sijoittamalla se ultraäänitaajuuksille. [6]

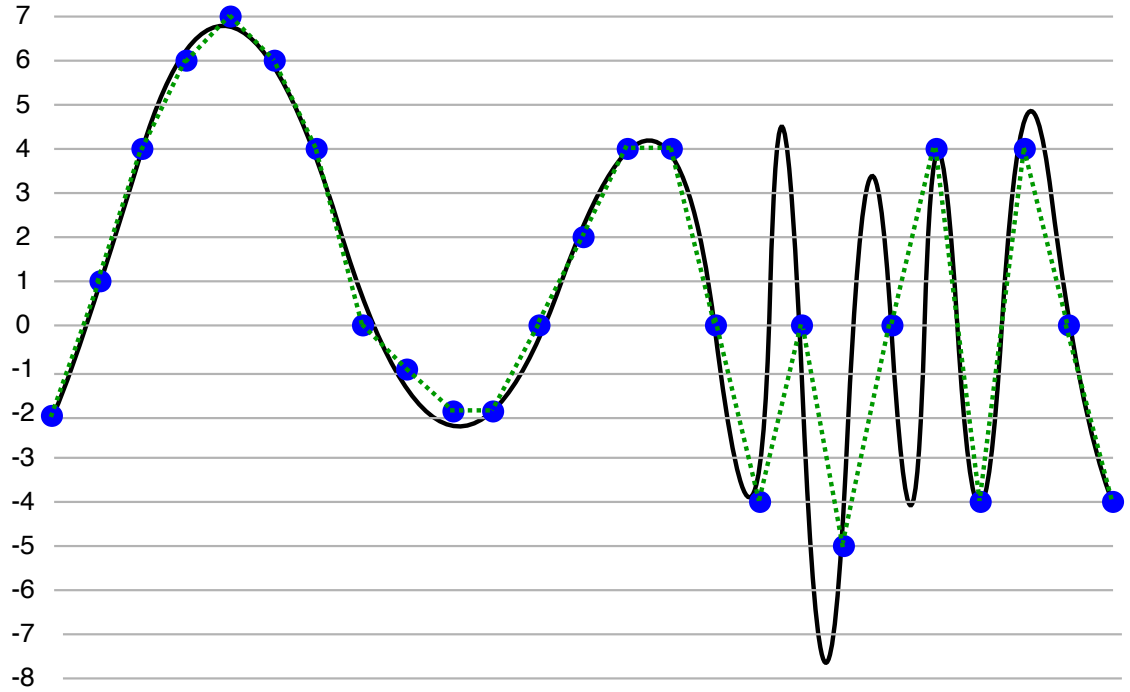


Kuva 2.2. *FSK-modulaation käyttäminen binäärisekvenssin kuljettamiseen.*

2.1.2 Digitaalinen ääni

Digitaalisessa äänessä äänisignaali on näytteistetty jatkuvaksi sarjaksi. Näytteistäminen tapahtuu ottamalla äänisignaalista näytteitä tasaisin aikavälein. Näytteenottotaajuus (engl. sample rate) vaikuttaa äänenlaatuun ja kuinka korkeaa taajuutta sillä voidaan ilmaista. Kuvassa 2.3 havainnollistetaan äänisignaalin näytteistystä. Näytteenottoteoreeman (engl. Nyquist–Shannon sampling theorem) mukaisesti näytteenottotaajuuden täytyy olla vähintään kaksikertainen alkuperäisessä signaalissa esiintyvään maksimitaajuuteen nähden, jotta signaalin taajuus voidaan ilmaista [14]. Mikäli näytteenottotaajuus ei ole riittävä signaalin ilmaisemiseksi, tapahtuu laskostumista. Kuvassa 2.3 tämä voidaan huomata signaalin loppuosassa, jossa alkuperäisen signaalin taajuus kasvaa niin isoksi, ettei näytteenottotaajuus riitä sitä ilmaisemaan. Alkuperäisen äänisignaalin vaihtelu jää virheellisesti näytteistetystä signaalista kokonaan pois.

Yleisesti käytettyjä näytteenottotaajuuksia (engl. sample rate) ovat 44.1kHz, 48kHz ja 96kHz [24]. Näistä matalin näytteenottotaajuus mahdollistaa 22,5kHz taajuuden toistettavassa äänisignaalissa. Äänen näytteenottotaajuus vaikuttaa digitaalisen äänen tiedonsiirtonopeuteen. Tiedonsiirtonopeus on tietyssä ajassa siirretyn tiedon määrä. Näytteenottotaajuuden lisäksi äänen tiedonsiirtonopeuteen vaikuttavat bittisyvyys (engl. bit depth) ja äänikanavien määrä (engl. audio channels). Bittisyvyys ilmaisee, kuinka monta bittiä yhden näytteen tallentamiseen käytetään. Kuvan 2.3 tapauksessa tämä on 4-bittiä, jolla voidaan ilmaista 16 arvoa. Yleisesti käytetty bittisyvyys on 16-bittiä, jolla voidaan ilmaista 65536 eri arvoa. Äänikanavat digitaalisessa äänessä erotetaan omina näytteistettyinä signaaleissaan. Yleisesti suoratoistossa käytetään kahta äänikanavaa eli stereoääntä. Äänikanavien määrällä on iso vaikutus tarvittavaan tiedonsiirtonopeuteen. Tämän takia useampaa äänikanavaa käytetään vain tapauksissa, joissa äänenlaatu on kriittinen. Syn-



Kuva 2.3. *Analogisen äänisignaalin näytteistäminen. Alkuperäinen äänisignaali mustalla, siniselle signaalista otetut näytteet.*

tyvä tiedonsiirtonopeus (engl. bitrate) saadaan laskettua kaavalla 2.1.

$$bitrate = sample\_rate * bit\_depth * audio\_channels \tag{2.1}$$

Tiedonsiirtonopeus on tärkeä parametri videon suoratoiston optimoinnissa. Tiedonsiirtonopeus halutaan pitää mahdollisimman pienenä, kuitenkin siten, ettei äänenlaatu huomattavasti heikkene. Perinteisillä tallennusmedioilla, kuten CD ja DVD, on huomattavasti enemmän tallennustilaa käytettävissä. Isommat näytteenotto taajuudet ovatkin käytössä DVD ja BluRay-medioilla [24].

2.1.3 Äänen taajuusanalysointi

Signaali saadaan muutettua taajuustasoon tekemällä sille Fourier'n muunnos (FT). Digitaalinen äänisignaali saadaan muutettua taajuustasoon tekemällä diskreetti Fourier'n muunnos (DFT). Nopea Fourier'n muunnos (FFT) on tehokas algoritmi diskreetin Fourier'n muunnoksen laskemiseksi. DFT:n laskeminen on suoritusteholtaan raskas operaatio. DFT:n määritelmän mukainen laskennallinen kompleksisuus on $O(n^2)$, kun taas FFT:n laskennallinen kompleksisuus on $O(n)$. FFT-algoritmit tekevät oletuksia ja pyöristyksiä muunnoksen laskemisessa. Tästä seuraa, että niillä saatu muunnos ei ole yhtä tarkka, kuin määritelmän mukaan laskettu. [26]

FFT on yleisesti käytetty työkalu taajuusanalyysissä [26]. Yleinen FFT:n käyttökohde on spektogrammin piirtäminen. Spektogrammi on visuaalinen esitys signaalin taajuudesta ajan suhteen. Signaalin taajuustasosta voidaan myös tarkastella, mitä taajuuksia signaali sisältää. FFT:tä voidaankin käyttää etsimään äänisignaaliin FSK-menetelmällä piilotettu datasignaali.

Toinen tapa tunnistaa taajuuksia on Gerald Goertzelin kehittämä ja hänen nimeään kantava Goerzelin algoritmi. Algoritmilla yksittäisen taajuuden laskeminen on lähes kaksi kertaa tehokkaampaa kuin DFT:n määritelmän mukaisella menetelmällä. Goertzel-algoritmi suoriutuu myös FFT-menetelmää paremmin yksittäisen tai muutaman taajuuden laskemisessa. Mikäli tunnistettavia taajuuksia on kuitenkin enemmän kuin $log(n)$, jossa n on näytteiden määrä, FFT-menetelmästä tulee tehokkaampi. 1024 näytteellä tämä raja on 3 taajuutta. Goertzel-algoritmin yleinen käyttökohde onkin tunnistaa binäärisekvenssejä FSK-menetelmällä moduloidusta signaalista, koska tunnistettavia taajuuksia on kaksi. [26]

2.2 Selainympäristössä

Multimedian toistaminen selainympäristössä on pitkään ollut kolmannen osapuolen ohjelmia vaativa asia. Viime vuosina on kuitenkin tapahtunut merkittäviä muutoksia ja esitelty erilaisia standardeja. Spesifikaatioita multimedian esittämiseen on määritelty, joita suurimmat selainkehittäjät ovat lähteneet toteuttamaan. Tässä aliluvussa käydään läpi työn kannalta merkittäviä standardeja ja spesifikaatioita.

2.2.1 HTML5

HTML (engl. HyperText Markup Language) on avoimesti standardoitu kuvauskieli, jonka pohjalta verkkoselaimet kuvantavat webbisivuja. HTML-standardin ylläpitämisestä ja kehittämisestä vastaa 448 jäsenestä koostuva World Wide Web Consortium (W3C) [2].

HTML5 on HTML-standardin viides versio. Mediatoiston mahdollistamiseksi ilman liitännäisiä HTML5-standardin osana määritettiin mediaelementit. Mediaelementtejä käytetään esittämään dataa, videota ja ääntä. [9]

Spesifikaation mukaisesti mediaelementit lataavat toistettavan median kokonaan, ennen kuin se voidaan toistaa. Tämä spesifikaatio ei ole yhteensopiva suoratoistoperiaatteen kanssa, jossa videota toistetaan sitä mukaa, kun sitä on ladattu. Suoratoiston mahdollistamiseksi määritettiin toinen spesifikaatio nimeltään MSE (engl. Media Source Extensions).

MSE:n tarkoituksena on mahdollistaa videoelementille työnnettävät bittivirrat (engl. bytestream) JavaScript-koodista. Tämä mahdollistaa suoratoisto-kirjastojen toteutuksen. Kirjaston tehtävänä on noutaa ja synkronoida videodata videoelementille. MSE:n avulla

on ollut mahdollista tehdä JavaScript-pohjainen toteutus erilaisille suoratoistoprotokollille. JavaScript-pohjaiset toteutukset onkin hallitsevassa asemassa suoratoistopalveluissa [29].

2.2.2 JavaScript selainympäristössä

JavaScript on korkean tason ohjelmointikieli, jota voidaan suorittaa selainympäristössä. Se on oliopohjainen ja tulkattava kieli. Tulkattavaa ohjelmakoodia ei käännetä konekieliseksi, vaan se suoritetaan lause kerrallaan komentotulkin päällä. Tästä seuraa yleisesti se, että tulkattava ohjelmakoodi on hitaampaa suorittaa kuin valmiiksi konekielelle käännetty. Tulkin on analysoitava ohjelmakoodi ja tehtävä tarvittavat optimoinnit ennen ohjelmakoodin suorittamista. Tulkattavuudella saavutetaan muun muassa laiteriippumattomuus ja helpompi virheiden jäljittäminen (engl. debugging).

Ennen HTML5-standardin toteuttamista JavaScriptiä suoritetaan selainympäristössä yhden säikeen avulla. Tämä säie suorittaa tapahtumasilmukkaa (engl. event loop). Tapahtumasilmukka purkaa funktiokutsupinoa. Funktiokutsupino toimii LIFO-periaattella (engl. Last in, First out). Tämä tarkoittaa, että viimeiseksi tullut funktiokutsu otetaan ensimmäisenä suoritukseen. Yksittäinen funktiokutsu on lyhyt osa JavaScript ohjelmakoodia. Tapahtumasilmukka varmistaa, että jokainen funktiokutsu suoritetaan aina alusta loppuun, eikä tätä prosessia voida keskeyttää. Tämä takaa, ettei JavaScript-ohjelma aiheuta ikinä resurssien käytön estymistä. Mikäli yksittäisen funktion suorituksessa kestää liian kauan, voi käyttöliittymä jähmettyä, koska näytönpäivitykset suoritetaan samassa tapahtumasilmukassa. Loppukäyttäjälle tämä ilmenee verkkoselaimen reagoimattomuutena esimerkiksi painalluksiin. Tämä toiminnallisuus teki haastavaksi suorittaa suoritustehoa vaativia algoritmeja selainympäristössä. Tämän ratkaisemiseksi HTML5-standardissa esiteltiin worker-rajapinta. Worker-rajapinta mahdollistaa JavaScript-ohjelmakoodin suorittamisen usealla säikeellä yhtäaikaisesti selainympäristössä. Säikeiden välinen kommunikaatio hoituu tapahtumien avulla. Tapahtuma lisää funktiokutsun funktiokutsupinoon. Funktiokutsulle voidaan antaa parametreja. Näillä parametreilla voidaan välittää tietoa säikeiden välillä. Toinen merkittävä selainympäristön suorituskykyyn vaikuttava uudistus on WebAssembly.

2.2.3 WebAssembly

WebAssembly (Wasm) on matalan tason ohjelmaformaatti, jota suoritetaan virtuaalikoneen avulla. Sen päätavoitteena on mahdollistaa korkea suorituskyky selainympäristössä. Wasm on suunniteltu toimimaan siten, että korkeamman tason ohjelmointikielillä toteutettuja ohjelmia voidaan kääntää Wasm-formaattiin. Tällaisiä korkean tason ohjelmointikieliä ovat muun muassa C, C++ ja Rust, jotka ovat tunnettuja tehokkuudestaan. Näillä kielillä toteutettuja ohjelmointikirjastoja voidaan hyödyntää selainympäristössä hyvin pit-

kälti sellaisenaan. Wasmia ei ole tarkoitettu JavaScriptin korvaajaksi, vaan toimimaan sen rinnalla. Ideaaleja käyttötapauksia Wasmille on suoritustehoa vaativien algoritmien suorittaminen. [32]

Wasmia kuvaillaan nopeaksi, turvalliseksi, laitteisto-, ohjelmointikieli- ja alustariippumattomaksi. Wasmin nopeus on seurausta sen kyvystä hyödyntää laitteistoläheisiä ominaisuuksia. Turvallisuus taataan koodivalidoinneilla ja rajoittamalla muistialue (engl. sandboxing). Riippumattomuus taataan virtualisoinnilla ja kääntäjätuella. Wasmia voidaan integroida selainympäristön lisäksi myös muihin ympäristöihin toteuttamalla tarvittava virtualisointi.

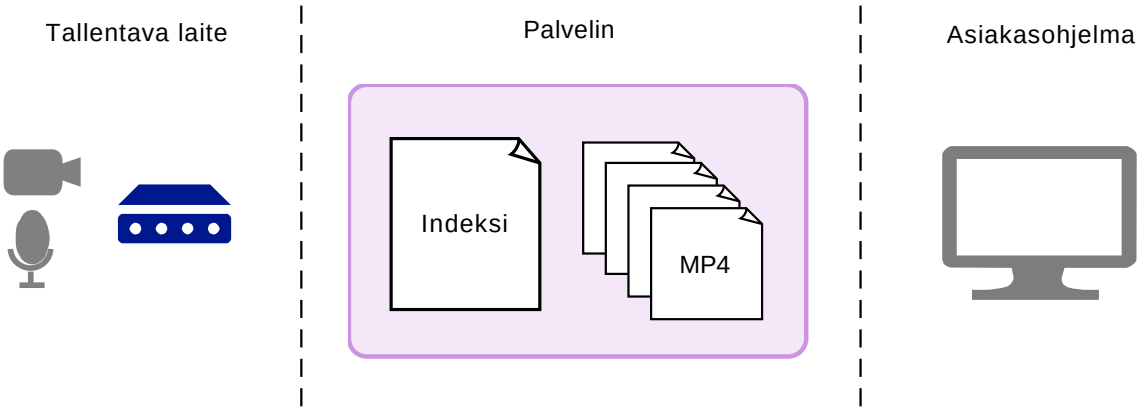
Wasmin kehittämistä tukevat kaikki neljän suurimman selaimen Firefoxin, Googlen, Edgen ja Safarin takana olevaa yritystä. Wasm-tuki on ollut saatavilla näillä neljällä selaimella jo vuodesta 2017. Wasmin ensimmäinen standardi julkaistiin kuitenkin vasta vuoden 2018 syyskuussa. Standardin julkaisu tarkoittaa, että kaikki uudet muutokset Wasmiin ovat takaperin yhteensopivia. Ensimmäisen standardin pohjalta perustettu sovellus on siis tuettu taaksepäin. [31]

2.2.4 Suoratoistoteknologiat

BitMovin suorittaman kyselyn perusteella sovelluskehittäjien kaksi selvästi eniten käyttämää suoratoistoteknologiaa ovat HLS ja MPEG-DASH. Nämä ovat kasvattaneet suosiotansa tasaisesti syrjäyttäen vanhentuneita tekniikoita kuten Flash, HDS ja Smooth Streaming. [28][29]

HLS (engl. HTTP Live Streaming) on Applen kehittämä suoratoistoteknologia. Apple kehitti HLS-teknologian vuonna 2010 korvatakseen Adobe Flash Playerin, jota se moitti heikoksi suorituskyvyltään ja tietoturvaltaan [18]. Kuvassa 2.4 kuvaillaan tyypillinen HLS-konfiguraatio. Konfiguraatio koostuu kolmesta osasta. Ensimmäinen osa on kooderi, jonka tehtävänä on muuntaa sisään tulevat ääni ja video tarvittavaan muotoon. Toinen osa on palvelin, jonka tehtävä on segmentoida saamansa datavirta (engl. stream) paloiksi ja ylläpitää listaa eri palasista. Viimeinen osa on asiakasohjelma, jonka tehtävänä on pyytää palvelimelta lista ladattavista segmenteistä. Listan perusteella asiakasohjelma pyytää tarvittavat mediasegmentit, järjestää ja toistaa ne käyttäjälle.

Nimensä mukaisesti HLS on rakennettu HTTP-protokollan päälle. HTTP-protokolla toimii pyyntö-vastaus periaatteella. Asiakasohjelma tekee pyynnön palvelimelle ja palvelin vastaa. Pyynnön suoritukseen kuluva aika ei ole tiedossa ja voi vaihdella. Tästä syystä HLS-asiakasohjelma pyrkii ylläpitämään useampaa segmenttiä valmiina toistoa varten. Tätä kutsutaan puskuriksi (engl. buffer). Puskurin tarkoituksena on varata aikaa vastaanottaa vastaus HTTP-pyyntöihin. Mikäli asiakasohjelma tunnistaa, ettei käyttäjä ehdi vastaanottaa segmenttejä riittävän nopeasti, se voi pyytää palvelimelta huonompilaatuisia segmenttejä. Tätä kutsutaan adaptiiviseksi striimaamiseksi (engl. adaptive streaming). Adaptiivinen striimaus edellyttää, että palvelimella media on koodattu eri tiedonsiirtono-



Kuva 2.4. *Tyypillinen HLS-konfiguraatiomalli.*

peuksille. Tyypillinen ja Applen suosittelema segmentin kesto on 6 sekuntia [8]. HLS-konfiguraatiosta riippuen vähintään kolme segmenttiä täytyy olla ladattuna, jotta media voidaan toistaa. Applen suosittelema segmentti määrä on kuusi [8]. Live-striimiä katseltaessa puskuri aiheuttaa viivettä. Edellä mainitulla tyypillisellä konfiguraatiolla HLS-striimin viive on 36 sekuntia. Viivettä voidaan pienentää konfiguroimalla palvelin lähettämään pienempiä segmenttejä. Pienemmistä segmenteistä seuraa moninkertainen määrä HTTP-pyyntöjä. HTTP-pyynnön vastaus sisältää aina dataa liittyen protokollaan. Pienentämällä segmenttien kokoa ja sitä kautta HTTP-pyyntöjen määrää, videodatan määrä pienenee suhteessa koko tiedonsiirron määrään. HTTP-pyyntöjen käsittelyssä on myös tehtävä protokollan määrittämät tarkastelut, joihin kuluu aikaa. Tästä seuraa, ettei HLS-teknologialla striimin viivettä voida pienentää lähes reaaliaikaiseksi. Parhaimmillaan HLS-teknologialla saadaan luotettavasti noin kuuden sekunnin viive [19].

DASH (Dynamic Adaptive Streaming over HTTP) -standardi julkaistiin vuonna 2012 ja on eniten käytetyistä suoratoistoteknologioista tuorein. DASH-standardin kehittämisen takana on suuret teknologia yritykset kuten Microsoft, Netflix, Qualcomm, Ericsson ja Samsung. Kuten HLS, myös MPEG-DASH on HTTP-pohjainen suoratoistoprotokolla, joka tukee adaptiivista striimausta. Toimintaperiaatteiltaan DASH on hyvin samankaltainen kuin HLS [15]. Koodatut videot segmentoidaan ja lähetetään asiakasohjelmalle pyydettäessä. Suurimpana erona HLS-teknologiaan on DASH-teknologian koodekkiriippumattomuus. HLS-teknologia määrittää tarkasti, mitä koodekkia käyttäen video ja ääni pitää olla koodattu.

2.2.5 Videokoodekit

HTML5 spesifikaatiossa määritetään, että selaimet voivat tukea mitä tahansa video- tai äänikoodekkia [10]. Tästä on seurannut hajanainen tuki eri koodekeille eri valmistajien selamilla.

BitMovin teettämän kyselyn perusteella selvästi eniten käytetyt videokoodekit ovat H.264/MPEG-4 AVC (Advanced Video Codec) ja H.265/MPEG-H HEVC (High Efficiency Video

Coding) [29]. Tämä selittyy osittain sillä, että ne ovat ainoat HLS-teknologian tukemat koodekit [8].

Työssä käytetään tästä eteenpäin lyhenteitä AVC ja HEVC selkeyden vuoksi. AVC ja HEVC ovat standardoituja videonpakkausmenetelmiä. AVC-standardin ensimmäinen versio julkaistiin vuonna 2003. AVC-standardista on julkaistu 17 versiota, joista viimeisin vuoden 2017 huhtikuussa [16]. HEVC kehitettiin AVC:n seuraajaksi ja sen standardin ensimmäinen versio julkaistiin vuonna 2013. Siitä on julkaistu viisi versiota, joista viimeisin helmikuussa 2018 [17]. Vaatimukset videonpakkaukselle ovat nopean muutoksen kohteena, mikä selittää standardien nopeaa kehitystä. Toinen nopeaan muutokseen ajava asia on AVC:n ja HEVC:n lisensointi. Teknologia on suojattu useilla patenteilla ja teknologian kaupallinen käyttö vaatii rojaltien maksamista [25].

HEVC- ja AVC-teknologioiden käytön maksullisuus on ollut osa syynä muiden koodekkien kehittämiseen ja kasvuun. Kyselyn perusteella kolmanneksi eniten käytetty koodekki on vuonna 2013 julkaistu VP9. VP9 on Googlen kehittämä avoin ja rojaltiton koodekki. Suurimpana esteenä VP9:n laajemmalle kasvulle oli selaintuki. Tällä hetkellä eniten käytetyistä selaimista tukea ei löydy Microsoftin Internet Explorerista ja Applen Safarista [7]. Microsoft ilmoitti vuonna 2016 lopettavansa Internet Explorer kehittämisen ja julkaisevan vain tietoturvapäivityksiä [13]. VP9:n seuraajaksi kehitettiin AV1. AV1:n kehittämisestä vastaa siihen varten perustettu Alliance for Open Media (AOMedia). Vuonna 2018 Apple liittyi AOMedian jäseneksi, mikä voi tarkoittaa tulevaisuudessa AV1-tukea myös Applen selaimille.

2.2.6 Äänikoodekit

Erilaisia äänikoodekkeja on kehitetty paljon eri tarkoituksiin. Eri tarkoituksilla tarkoitetaan eri taajuusalueelle keskittyviä koodekkeja, joilla voidaan käyttää erityyppisien äänien siirtämisessä. Esimerkiksi puheviestinnässä käytetään yleisesti kapeaa taajuusaluetta, koska ihmisen puhe sisältää ääniä kapealta taajuusalueelta. Eri taajuusalueilla on myös erilaisia ominaisuuksia, joita voidaan hyödyntää koodauksessa. Tästä syystä osa koodekeista keskittyy vain kapealle taajuusalueelle. Esimerkiksi vuonna 1999 kehitetty AMR-NB (engl. Adaptive Multi-Rate narrowband) mahdollistaa äänen koodauksen 200–3400 hertsin alueella. Sitä käytettiinkin puhelimen puheviestinässä.

Tallenteiden ja musiikin osalta MP3 on pitkään ollut hallitsevassa asemassa äänen koodausteknologioissa. Se on laajalti käytetty erilaisissa tallennetuissa medioissa. Sillä voidaan tallentaa korkealaatuista ääntä. MP3-formaatin vaatima tiedonsiirtonopeus on kuitenkin suhteellisen korkea, ja tämän takia sitä ei suosita suoratoistossa. MP3 on avoin ja lisenssitön koodekki.

Suoratoistossa käytettävien äänikoodekkien osalta muutos on ollut huomattavasti videokoodekkeja maltillisempaa. AAC kehitettiin tarjoamaan korkealaatuista ääntä pienillä tiedonsiirtonopeuksilla. Siitä kasvoi nopeasti laajalti käytetty ääniformaatti niin suoratoistos-

sa kuin tallennetuissa medioissa.

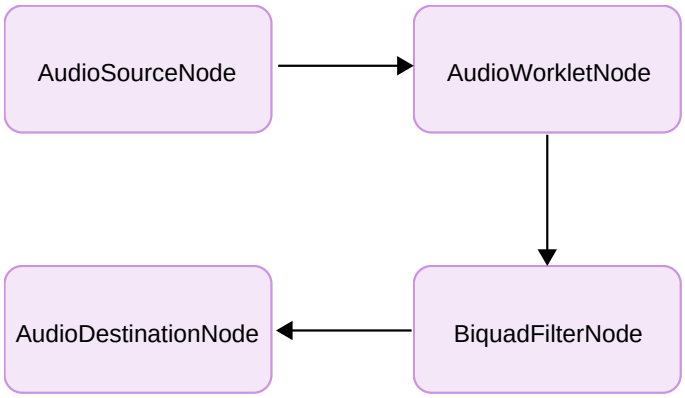
HLS-teknologia vaatii äänen koodaukseen käytettävän AAC (Advanced Audio Coding) tai uudempaa HE-AAC -teknologiaa [8]. AAC on myöskin suojattu patenteilla, mutta sen toteuttavan kooderin käytöstä ei tarvitse maksaa lisenssimaksuja. Teknologiaa käyttävien koodereiden ja dekoodereiden kehittäjän on maksettava lisenssimaksuja teknologian omistavalle VIA-CORP:ille [1].

Lisenssitön ja selvästi suosiotaan kasvattanut äänikoodekki on Opus [23][29]. Opus mahdollistaa äänen pakkaamisen tehokkaasti millä tahansa taajuusalueella ja tarjoaa markkinoiden parasta äänenlaatua [21]. Opuksen etuina ovat myös lisenssittömyys ja avoin lähdekoodi. Erona muihin yleisesti käytettyihin koodekkeihin on, että Opus rajoittaa äänisignaalin maksimitaajuudeksi 20 kilohertsiä. Kaikki sen ylittävät taajuudet suodattuvat pois. Opus on kehitetty ihmisten kuunneltavaa ääntä varten ja tällä halutaan mahdollistaa parempi äänen pakkaussuhde.

2.2.7 Äänen prosessointi

Selainympäristössä äänen prosessointi ja analysointi on mahdollista Web Audio APIn avulla. W3C:n määrittelemä spesifikaatio APIlle on suhteellisen nuori ja selaimet toteuttavat APIn rajapintoja vaihtelevasti. Spesifikaatiossa mukaan rajapinnan tarkoitus on mahdollistaa minkä tahansa äänenprosessointityökalun toteutus, joka on järkevästi toteutettavissa skriptikielen kontrolloiman optimoidun C++-ajoympäristön avulla.

API määrittelee rajapinnat, joista voi koota äänen käsittelyyn tai analysointiin mahdollistavan liukuhihnan (engl. pipeline). Liukuhihna muodostuu solmuista (engl. node). Jokaisella solmulla voi olla sisääntuloja ja ulostuloja. Sisääntulosolmulla ei ole sisääntuloa (engl. SourceNode) ja yksi ulostulo. Ulostulosolmulla on yksi sisääntulo, eikä yhtään ulostuloa. Kuvassa 2.5 on havainnollistettu yksinkertainen liukuhihna.



Kuva 2.5. *Yksinkertainen Web Audio APIa hyödyntävä kaavio.*

Sisääntulosolmu voidaan määrittää kaappaamaan ääni mistä tahansa HTML5-spesifikaation määrittelemästä mediaelementistä. Kaapattu ääni syötetään määritettyyn ulostuloon. Web Audio API sisältää paljon valmiita solmuja, joita yhdistelemällä ja konfiguroimalla

saadaan toteutettua monimutkaisia äänenprosessointiliukuhihnoja. Kuvassa esiintyvällä 2.5 BiquadFilter-solmulla voidaan esimerkiksi määrittää alipäästösuodin (engl. low pass filter). Alipäästösuodin päästää lävitseen määritetyn rajan alittavat taajuudet. Ulostulosolmu lähettää saamansa sisääntulon selaimelle, joka puolestaan toistaa äänen käyttäjän kaiuttimista. [30]

Esimerkin toinen solmu, AudioWorklet mahdollistaa äänen prosessoinnin toteuttamisen algoritmitasolla hyödyntäen JavaScript-ohjelmointikieltä. AudioWorklet-rajapinta on aiemmin esitellyn Worklet-rajapinnan erikoistapaus. AudioWorkletin spesifikaatio on nuori ja sitä vastaavaa toiminnallisuutta eivät vielä toteuta muut selaimet kuin Google Chrome. Muille selaimille AudioWorklet-tuki saadaan polyfill-kirjastojen avulla. Nämä kirjastot toteuttavat AudioWorklet rajapinnan hyödyntäen Worklet-rajapintaa, joka löytyy useilta moderneilta selaimilta.

AudioWorkletNode määritettiin korvaamaan jo spesifikaatiosta löytyvää ScriptProcessorNode-rajapintaa. ScriptProcessorNoden ongelmana oli sen toimiminen selaimen pääsäikeessä. Raskas audioprosessointi ScriptProcessorNodea hyödyntäen saattaa estää käyttöliittymän päivittämisen. Raskas käyttöliittymän päivitys taas voi aiheuttaa äänen prosessointiin viivettä. Tämä ongelma ratkaistaan AudioWorkletNodessa suorittamalla JavaScript-koodi omassa säikeessään. Pääsäikeellä ja äänen prosessointisäikeellä on kaksi jaettua muistiosiota (engl. shared memory).

AudioWorkletNodesta periytetyn luokan täytyy toteuttaa process-metodi, joka saa parametrina sisääntulo- ja ulostulopuskurin toisen yhteisen muistiosion avulla. Process-metodille ohjataan spesifikaation mukaisesti 128 kehystä (engl. frame) kerrallaan. Yksi kehys koostuu yhdestä ääninäytteestä jokaista äänikanavaa kohden, eli stereoäänellä 256 näytteestä. 44100 hertsin näytteenottotaajuudella tämä tarkoittaa, että process-metodia kutsutaan 344 kertaa sekunnissa.

Toista yhteistä muistiosiota käytetään välittämään viestejä pääsäikeen ja audiosäikeen ohjelmakoodien välillä. Viestien välittäminen tapahtuu tapahtumien avulla ja on kaksisuuntaista. Säikeissä ajettavat ohjelmakoodit voivat rekisteröidä kuuntelijat kuuntelemaan näitä tapahtumia. Tapahtumiin voidaan liittää dataa minkä tahansa JavaScriptin tukeman muuttujan avulla. Koska viestit käyttävät JavaScriptin tapahtumarajapintaa, niiden vastaanottamisessa voi olla viivettä. Selainympäristön pääsäie ei keskeytä keskeneräistä fuktiokutsua tapahtuman käsittelyn ajaksi, vaan tapahtuma käsitellään tapahtuma-silmukan periaatteiden mukaisesti.

2.2.8 Ajoitus JavaScript-ympäristössä

Selainympäristössä ajettavassa JavaScriptissä ei ole tarkkaa ajoitustoiminnallisuutta. Yleisesti käytettäville setTimeout- ja setInterval-metodeille annetaan parametrina aika millisekunteina, jolloin toisena parametrina annettu funktiokutsu haluttaisiin suoritettavan. Spesifikaation mukaan nämä kutsut eivät ole tarkkoja [11]. Koska myös nämä kutsut suo-

ritetaan pääsäikeessä, ajastimen laukeamiseen yhdistetyn ohjelmakoodin on odotettava nykyisen ohjelmakutsun suoritus loppuun. Tästä voi seurata muutaman millisekunnin viivästys.

Ajastintoteutukset tehdäänkin yleensä hyödyntäen JavaScriptin Date-rajapintaa. Date-rajapinnassa on metodi, joka palauttaa, kuinka monta millisekuntia on kulunut tammikuun ensimmäisestä päivästä vuonna 1970 kello 00:00:00.000 GMT. Yleinen tapa toteuttaa suhteellisen tarkka ajastin on ketjuttaa setTimeout-metodeita lisäämällä aina uusi ajastin setTimeout-metodille parametrina annetussa funktiossa. Funktiossa noudetaan Date-rajapinnasta kuluneet millisekunnit ja vähentämällä ne edellisen kutsun aikaleimasta saadaan kuluneet millisekunnit. Mikäli funktiokoodi halutaan ajettavan tasaisesti, esimerkiksi 1000 millisekunnin välein, voidaan tarkkuutta parantaa arvioimalla kutsun mahdollista viivästymistä laskemalla setTimout-parametrille määritetyn arvon ja Date-rajapinnasta saatujen arvojen eroa. Uuteen setTimeout-funktiokutsuun voidaan asettaa arviota pienempi parametri.

Toinen mahdollisuus tarkemman ajastimen toteutukseen on hyödyntää AudioWorkletNode-rajapintaa. Koska rajapinnan toteuttava koodi ajetaan omassa säikeessään, siellä ajettaviin ajastimiin eivät vaikuta pääsäikeessä tapahtuvat raskaat operaatiot. Tässä ympäristössä aiemmin mainitut setInterval ja setTimeout-funktiot ovat itsessään hyvin tarkkoja.

2.3 Olemassa olevat toteutukset

Ajoitusdatan ja ajoitetun metadatan (engl. timed metadata) lisäämiseen suoratoistoon on olemassa monia eri ratkaisuja. Näitä ratkaisuja voitaisiin käyttää ongelman ratkaisemisessa. Ajoitusdatan ja ajoitetun metadatan avulla usea datalähde voitaisiin synkronoida toistettavaan videoon. Tässä luvussa käydään olemassa olevia toteutuksia läpi ja todetaan, minkä takia niitä ei voida hyödyntää määritellyssä käyttötapauksessa.

2.3.1 Ajoitusdatan sisältävä siirtoprotokolla

RTP (engl. Real-time Transport Protocol) on internet-protokolla, joka mahdollistaa äänen ja videon siirtämisen reaaliaikaisesti. Protokolla aikaleimaa lähettämänsä paketit, mitä käytetään videon ja äänen synkronointiin. Tätä aikaleimaa voitaisiin myös käyttää metadatan synkronoinnin toteutukseen. RTP ei ole kuitenkaan tuettu sellaisenaan selainympäristöissä.

WebRTC (engl. Web Real-Time Communications) on RTP:tä hyödyntävä reaaliaikainen kommunikointiprotokolla. WebRTC-protokolla tukee media- ja datakanavia. Datakanava ei ole synkroninen mediakanavan kanssa ja synkronointi täytyy toteuttaa itse. Toteutuksen tekee haastavaksi WebRTC:n rajapinta selaimilla, josta ei löydy RTP-aikaleimatietoa [3]. WebRTC on jatkuvan kehityksen alainen ja mahdollisuus aikaleimoihin voi syntyä tu-

levaisuudessa.

2.3.2 Ajoitusdatan sisältävä säiliömuoto

Säiliömuoto (engl. container) on tiedostoformaatti, joka määrittää kuinka yksittäiseen tiedostoon tallennetaan useampia dataelementtejä. Nämä dataelementit voivat olla koodattua ääntä tai videota, tekstityksiä tai muuta metadataa. Yleisesti käytettyjä säiliömuotoja ovat muun muassa MP4, FLV, MPEG ja Matroska. Useat säiliömuodot mahdollistavat ajoitusdatan. Yksi tällainen säiliömuoto on Matroska.

Matroskan spesifikaation mukaan ajoitusdataa voidaan käyttää synkronoimaan videota, ääntä, tekstityksiä tai mitä tahansa Matroska-tiedostoon liitettyä raitaa. Spesifikaatiossa myös määritetään aikaleimojen tarkkuuden olevan 1 ms, joka on enemmän kuin tarpeeksi videon synkronoinnin tapauksessa. Yleisesti suoratoistossa näytetään maksimissaan 60 kuvaa sekunnissa. Tämä tarkoittaa ruudun päivitystä keskimäärin 17 millisekunnin välein. [20]

Yleisesti käytetyt suoratoistoprotokollat eivät tue Matroska-säiliömuotoa. Googlen sponsoroima WebM Project kehitti Matroskan pohjalta oman WebM-säiliömuodon. WebM on internet-käyttöön optimoitu säiliömuoto, joka tukee AV1-, VP9- ja VP8-videokoodekkeja. Ongelmaksi säiliömuodon aikaleimojen käytössä muodostuu selaintuki. WebM ei ole tuettu yleisesti käytössä olevalla Applen Safari-selaimella, joka suosii Applen itsensä kehittämää HLS-teknologiaa. Apple on kuitenkin liittynyt AV1-koodekkia hallinnoivaan AOMediaan, joten on mahdollista, että tulevaisuudessa jokin ajoitusdatan mahdollistava säiliömuoto on tuettu kaikilla yleisesti käytetyillä selaimilla. [33]

2.3.3 Ajoitetun metadatan mahdollistava suoratoistoprotokolla

Suoratoistoteknologioihin on lisätty erilaisia tekniikoita tukea ajoitettua metadataa. Yleisesti nämä toimivat videon suoratoistoa tarjoavalla palvelimella. Palvelin lisää suoratoistoprotokollasta riippuen metadatan joko säiliömuotoon (engl. in-band) tai kuljettaa tiedon HTTP-protokollan avulla säiliömuodon ulkopuolella (engl. out-of-band). HLS-protokolla määrittää ajoitetun metadatan ID3-leimojen (engl. tags) avulla. Standardoitua ID3-metadataa käytettiin alun perin metadatan lisäämisessä MP3-tiedostoihin ja sen nimi onkin lyhenne englannin kielisistä sanoista *IDentify an MP3* [12]. Yksi ID3-leima voi sisältää yhden tai useamman kehyksen. Yksi kehys sisältää yhden tai useamman tavun dataa. Leimoihin lisätään aikatieto ja ne voidaan tunnistaa videotoistimessa. Vastaava metadatan säiliötoteutus on AMF (engl. Action Message Format). Se on Adoben kehittämä binääriformaatti, jolla voidaan lähettää metadataa RTMP-suoratoistoprotokollan kanssa, jota voidaan käyttää esimerkiksi Adobe Flash Playerin kanssa.

Yhdistelemällä ID3- ja AMF-tekniikoita saadaan ajoitetun metadatan selaintuki yleisimmille selaimille. Tällä tavalla toimii muun muassa Wowza Streaming Engine [34]. Yhteistä näille tekniikoille on, ettei ajoitettu metadata ole osa videon säiliömuotoa. Tämä tarkoittaa, että ajoitettu metadata tulee lisätä joka kerta tallenteen suoratoistossa. MPEG-DASH puolestaan tarjoaa oman toteutuksensa ajoitetulle metadatalle: EMSG:n ja EventStreamit [22]. Näistä jälkimmäinen mahdollistaa myös säiliömuotoon lisätyt ajoitetut metadatat.

Ongelma palvelimella lisättävässä ajoitetussa metadatassa on synkronisuus. Videodatan saapumisessa sitä eteenpäin jakavalle palvelimelle on viivettä. Jotta metadata saadaan tarkasti synkronoitua, pitää tämä viive kyetä arvioimaan palvelimella ja asettaa ajoitus tämän mukaisesti. Mitä lähempänä median lähtöpistettä eli tallenninta metadata lisätään, sitä synkronisempi on lopputulos.

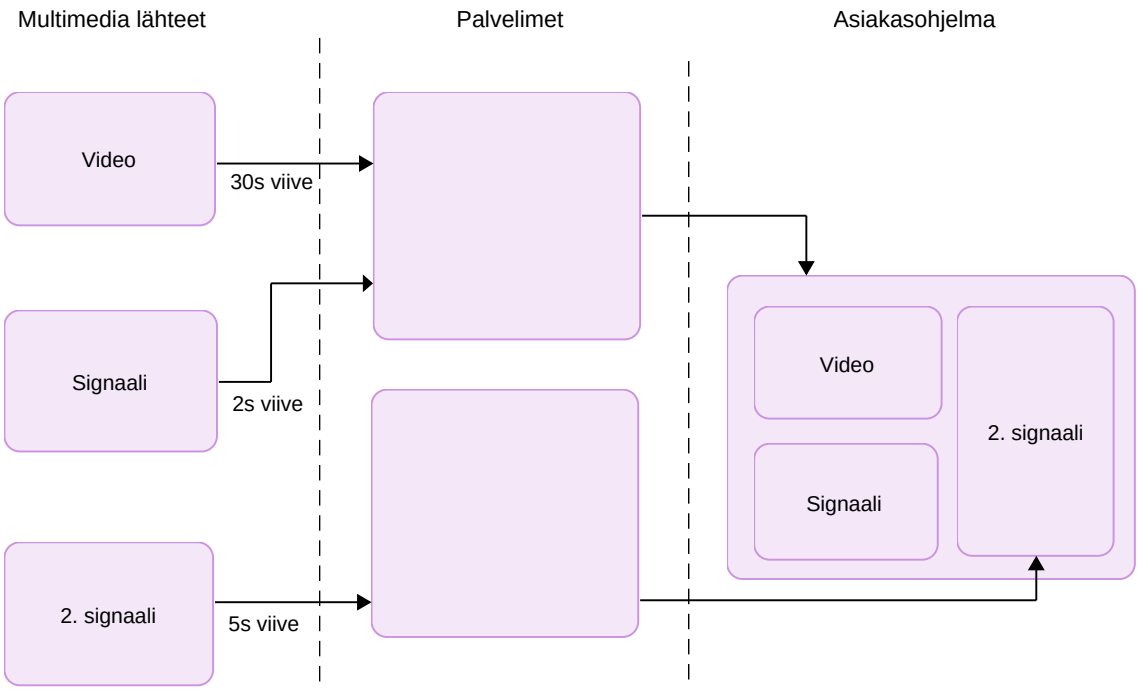
Edellä mainittuja tekniikoita hyödyntämällä suoratoistoon voitaisiin lisätä aikaleimoja sisältävää metadataa, jonka avulla useampi datalähde saataisiin synkronoitua. Näiden tekniikoiden lisääminen alustoille on kuitenkin työlästä, sillä jokaisen tekniikan toimintaperiaatteet ovat hyvin erilaiset. Esimerkiksi maksullinen Wowza Streaming Engine tukee ajoitettua metadataa HLS:n ja RTMP:n päällä toimiville Adobe HDS- sekä Flash-toistimille, mutta ei kuitenkaan MPEG-DASHin käyttämiä tekniikoita. Suuren työmäärän välttämiseksi työssä pyritään toteuttamaan järjestelmä, joka toimii kaikilla suoratoisto- ja koodausmenetelmillä.

3 JÄRJESTELMÄN SUUNNITTELU JA TOTEUTUS

Järjestelmän suunnittelua ohjasivat vanhat tekniikat ja kuinka niitä voidaan hyödyntää uusissa ympäristöissä. Tässä luvussa kuvataan toiminallisuus, joka toteutettavan järjestelmän tulee sisältää. Seuraavaksi käydään läpi suunnitteluun vaikuttaneet järjestelmävaatimukset ja miten nämä on huomioitu, sekä käydään läpi toteutukseen käytetyt menetelmät yleisellä tasolla. Lopuksi käydään yksityiskohtaisesti läpi järjestelmän toteutus komponenttitasolla.

3.1 Toteutettava toiminnallisuus

Toteutettavan järjestelmän on mahdollistettava useamman datalähteen synkronointi videolähteeseen. Datalähteellä tarkoitetaan tässä yhteydessä joko aikapohjaista jatkuvaa dataa, kuten signaalia, tai tapahtumapohjaista aikaleimattua dataa. Tapahtumapohjainen aikaleimattu data voi olla esimerkiksi chat-viesti. Kuvassa 3.1 on hahmoteltu järjestelmän toiminta. Asiakasohjelma on HTML5:n Web Audio APIa tukeva selainympäristö.



Kuva 3.1. *Toteutettavan synkronointijärjestelmän havainnekuva.*

Kuten aliluvussa 2.2.4 todettiin, suoratoistoprotokollasta riippuen videon viive voi olla huo-

mattava. Tyypillinen viive HLS-striimille on 36 sekuntia johtuen tarvittavan puskurin rakentamisesta. Datalähteen viive on vaihteleva lähteestä riippuen. Viiveellä tarkoitetaan aikaa, kuinka kauan kestää saada tieto tapahtumasta asiakasohjelmalle. Kuvan esimerkin tapauksessa asiakasohjelman on rakennettava signaaleista videon viiveen mittainen puskuri, jotta ne voidaan toistaa videon kanssa synkronisesti.

Palvelimien tehtävänä on jakaa eri lähteistä saatava data eteenpäin asiakasohjelmille. Eri datalähteet voivat kulkea eri palvelimien kautta asiakasohjelmalle. Palvelimet eivät saa rajoittaa yhdistettyjen asiakasohjelmien määrää. Työssä ei käsitellä palvelinarkkitehtuuria tarkemmin, koska se on riippuvainen käytettävästä suoratoistoteknologiasta.

Datalähdettä ei yhdistetä videodataan, vaan se lähetetään erillisenä datana asiakasohjelmalle. Asiakasohjelma mallintaa sen pohjalta näkymän käyttäjälle. Tällä saavutetaan huomattavasti pienempi tarvittava tiedonsiirtonopeus. Järjestelmän on toimittava sekä reaaliaikaiselle suoratoistolle, kuin tallenteen suoratoistolle (engl. video on demand). Tallenteen suoratoistolla tarkoitetaan tallennetun videon lataamista suorotoistoteknologioilla. Videota ei siis ladata kokonaisuudessaan, vaan pienissä osissa sitä mukaa, kun käyttäjä sitä toistaa.

3.2 Järjestelmän vaatimukset

Työhön liittyvään tekniseen toteutukseen oli ennalta tiedossa joitakin vaatimuksia. Nämä vaatimukset asettivat toteutukselle rajoitteita käytettävissä tekniikoissa ja ohjasivat suunnittelua.

3.2.1 Suoratoistomenetelmä- ja koodekkiriippumattomuus

Järjestelmän on kyettävä toimimaan yleisesti käytössä olevilla suoratoistomenetelmillä. Järjestelmän käyttöönotto eri suoratoistomenetelmillä ja videotoistimilla pitää olla suoraviivaista, eikä suoratoistomenetelmäkohtaista konfigurointia tarvita. Toteutetun järjestelmän tulee toimia kaikkien yleisesti suoratoistossa käytettyjen koodekkien kanssa. Yleisesti käytetyillä suoratoistomenetelmillä tarkoitetaan: HLS-, MPEG-DASH- ja WebRTC-teknologioita. Nämä on esitelty tarkemmin luvussa 2. Yleisesti käytetyillä koodekeilla tarkoitetaan videon osalta AVC-, HEVC-, VP9- ja AV1-, sekä äänen osalta AAC-, Opus- ja Vorbis-koodekkeja.

3.2.2 Selainriippumattomuus

Järjestelmän on kyettävä toimimaan yleisimmin käytetyillä moderneilla työpöytä käyttöön tarkoitetuilla selaimilla. Näillä tarkoitetaan Firefox-, Edge-, Chrome- ja Safari-selaimia.

Vielä huomattavan markkinaosuuden kattava Internet Explorer päätettiin jättää pois, koska sen tuki HTML5-standardille ja videoteknologialle on vajavainen.

Toteutuksen on tuettava jokaisen selaimien viimeisimpiä versioita diplomityön kirjoitushetkellä. HTML5-standardin eri osa-alueiden tuki on tullut eri selaimille eri ajankohtina ja toteutuksen tukeman tarkan version määrittäminen olisi työlästä. Modernit selaimet pitävät huolen, että käyttäjä lataa uusimmat päivitykset. On oletettavaa, että suurimmalla osalla käyttäjistä on käytössään selaimen viimeisin versio.

3.2.3 Tarkkuus ja suorituskyky

Järjestelmän on kyettävä saavuttamaan synkronoinnissa tarvittava tarkkuus. Selainympäristön asettamat rajoitukset ajastimien tarkkuudessa rajoittavat realistisen tavoitteen 5 millisekunnin tarkkuuteen. Käyttötapauksesta riippuen suoratoistossa näytetään videoissa 20-60 kuvaa sekunnissa (engl. frames per second). 60 kuvaa sekunnissa tarkoittaa kuvan vaihtumista 16,67 millisekunnin välein. Tätä pienempi tarkkuus ei paranna järjestelmän synkronisuutta, joten 16,67 millisekuntia on tarkkuus, johon järjestelmän on pyrittävä.

Tarkkuuteen vaikuttaa erityisesti järjestelmän suorituskyky. Toteutuksen resurssivaatimuksien on oltava niin vähäiset, että siitä ei aiheudu ylimääräistä viivettä aikaleimojen synkronointiin. Järjestelmän on kyettävä synkronoimaan liitetyt datalähteet 2 sekunnin kuluessa videon toiston alkamisesta. Tällä tarkoitetaan aikaa, joka saa kulua ajastimen saamisessa synkronoituun tilaan.

3.3 Toteutuksen yleiskuvaus

Vaatimus toteutuksen riippumattomuudesta suoratoistomenetelmästä ja käytetyistä koodekeista mahdollistaa toteutuksen, jossa synkronointiin tarvittava tieto on lisätty joko ääneen tai videoon. Nämä ovat ainoat elementit, jotka pysyvät lähes muuttumattomina tekniikoista riippuen. Molempiin syntyy muutosta koodaus-prosessissa, joten tarvittava tieto on lisättävä siten, että se säilyy prosessissa riittävän muuttumattomana, jotta se voidaan lukea asiakasohjelmassa.

Videoon tarvittavan tiedon piilottaminen siten, ettei katselija sitä havaitse on haastavaa. Lisäksi video pakkautuu tehokkaammin koodaus-prosessissa, joten siihen lisätty tieto on alttiimpi muutoksille. Tästä syystä työssä päädyttiin käyttämään ääntä ajoitukseen tarvittavan datan siirtämiseen.

3.3.1 Käytettävät menetelmät

Tarvittava ajoitustieto siirretään käyttäen FSK-menetelmää. FSK-menetelmällä luodaan äänisignaali, joka miksataan alkuperäiseen äänisignaaliin. Binäärisekvenssin sijaan käytetään heksadesimaalisekvenssiä, jossa jokaista 16 merkkiä vastaa ennalta määritetty taajuus. Heksadesimaalisekvenssiin päädyttiin, koska aikaleiman ilmaisemiseen binäärisekvenssinä tarvittaisiin 31 merkkiä. Heksadesimaaleja käyttämällä vastaava määrä on 8. Molempien sekvenssien tapauksessa kaikki perättäiset merkit on tunnistettava oikein, jotta dataa voidaan käyttää. Tästä seuraa, että heksadesimaalisekvenssi ei ole yhtä altis häiriöille.

Ääni prosessoidaan asiakasohjelmassa hyödyntämällä HTML5-standardin määrittämää Web Audio APIa. APIn avulla saatu videon äänisignaali muutetaan taajuustasoon käyttäen FFT-algoritmia. Taajuustasosta tunnistetaan heksadesimaalisekvenssi. Sekvenssi on aikaleima, jota käytetään ajoituskomponentin ajastimen päivittämiseen. Useammat lähteet synkronoidaan näin saatavan ajastimen avulla. Ajastimen päivittäminen pelkkien leimojen avulla mahdollistaisi maksimissaan yhden sekunnin tarkkuuden synkronointiin. Ajastimen tarkkuutta parannetaan JavaScriptin ajastin-toiminnallisuuksien avulla.

3.3.2 Vaatimusten täyttäminen

Kuten aliluvussa 2.2.7 todetaan, työn toteutuksessa käytettävä Web Audio API on spesifikaationa nuori ja vaihtelevasti toteutettu. Selainvaatimuksen takia työssä päädyttiinkin käyttämään polyfill-kirjastoa. Polyfill-kirjastolla tarkoitetaan JavaScript-kirjastoa, joka toteuttaa web-selaimesta puuttuvan spesifikaation mukaisen toiminnallisuuden. Toteutukseen tarvittava AudioWorklet-rajapinta on toteutettu vasta Google Chrome selaimessa. Käytetty polyfill-kirjasto toteuttaa AudioWorklet-rajapinnan hyödyntäen vanhentunutta (engl. deprecated) ProcessingNode-rajapintaa ja HTML5:stä löytyvää Worklet-rajapintaa.

Suorituskyvyn ja tarkkuuden saavuttamiseksi toteutuksessa käytetään HTML-standardin uusia ominaisuuksia, jotka mahdollistavat tehokkaan laskennan. Nämä ominaisuudet ovat Worklet-rajapinta, joka mahdollistaa JavaScript-koodin suorittamisen useassa säikeessä, sekä WebAssembly-rajapinta, joka mahdollistaa lähes natiivin suorituskyvyn selainympäristössä.

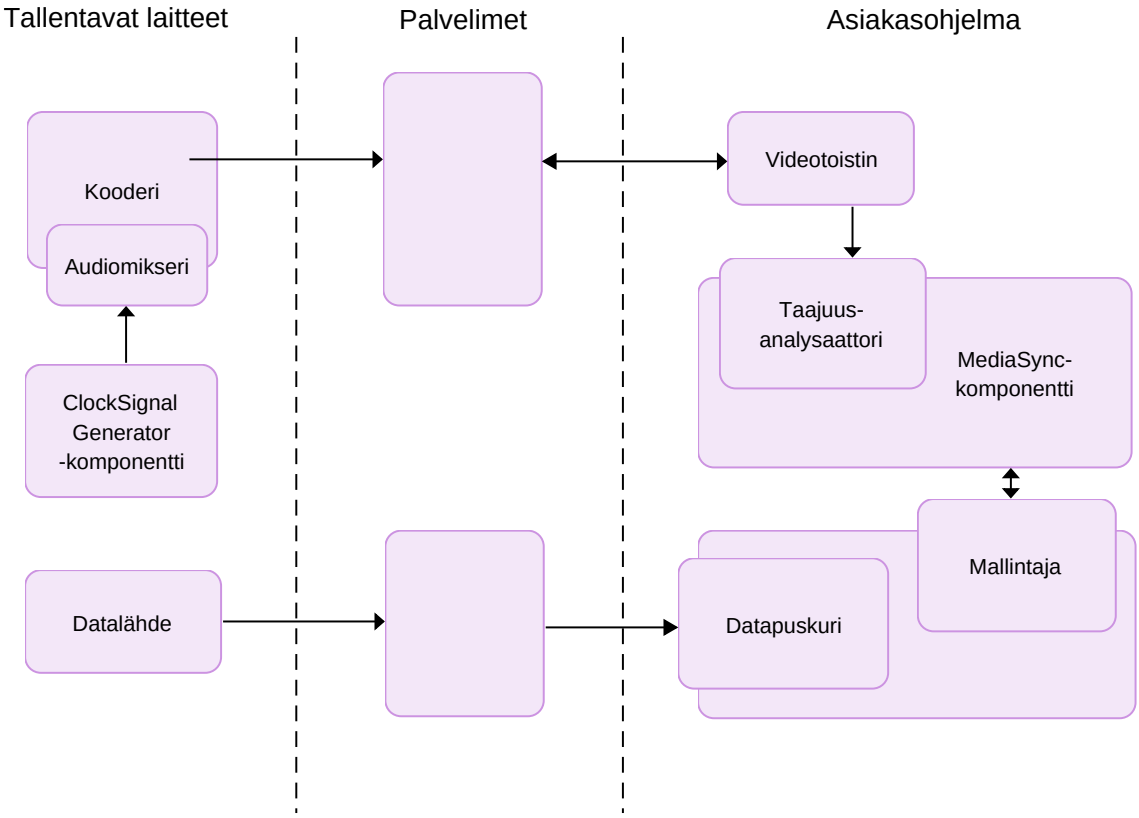
Suoratoistoteknologia- ja koodekkiriippumattomuus pyritään saavuttamaan käyttämällä taajuuksien analysointiin häiriötä hyvin sietäviä menetelmiä. Häiriötä tai taajuuksien muuttumista voi syntyä koodausprosessissa. Analysointiin käytetään FFT-menetelmää, jolloin voidaan analysoida koko taajuusaluetta, eikä vain yksittäistä signaalia.

3.4 Tekninen kuvaus

Järjestelmän arkkitehtuuri voidaan jakaa kolmeen osaan seuraavasti:

- datalähteet
- palvelimet
- asiakasohjelma.

Nämä kolme osaa on esitelty kuvassa 3.2 yleisellä tasolla. Datalähteet koostuvat videon ja äänen kaappaukseen, koodaukseen ja synkronointiin tarvittavista komponenteista sekä mahdollisista muista datalähteistä, jotka on tarkoitus synkronoida videolähteeseen. Toinen osa on palvelimet, joiden tehtävänä on jakaa datalähteet mahdollisille katsojille. Kuten aiemmin mainittiin, palvelimien rakenne ei vaikuta työssä toteutettuihin komponentteihin, joten sitä ei tarkastella tarkemmin. Kolmannen osan eli asiakasohjelman vastuulla on vastaanottaa, synkronoida ja esittää katselijalle medialähteet.



Kuva 3.2. *Järjestelmän arkkitehtuuri komponenttitasolla.*

Kaaviossa näkyvistä komponenteista toteutettiin työhön liittyen MediaSync-komponentti. MediaSync-komponentti on riippuvainen ClockSignalGenerator-komponentista. Näiden kahden komponentin toteutusta tarkastellaan tarkemmin seuraavissa luvuissa.

Taulukko 3.1. *Heksadesimaalimerkkiä vastaava taajuus*

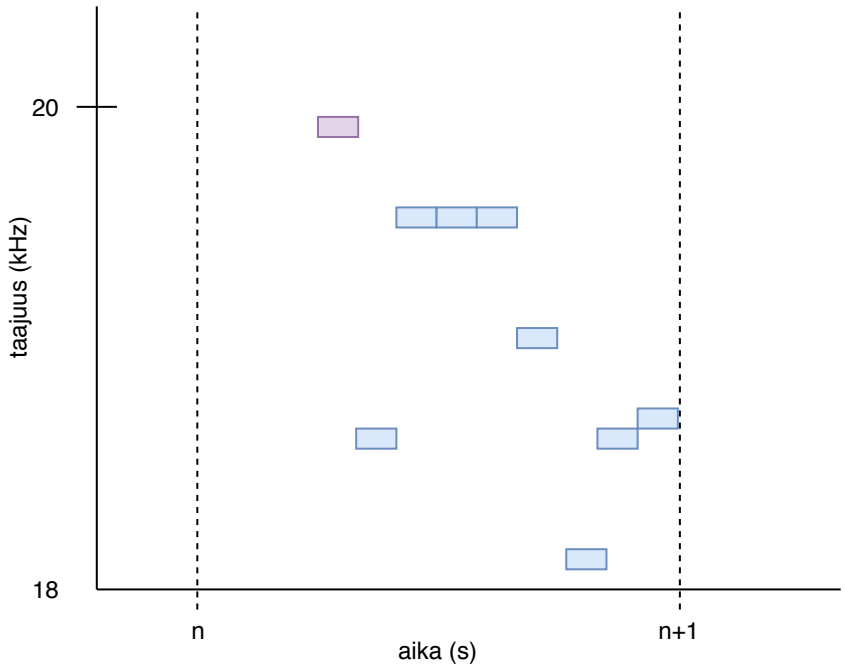
heksadesimaalimerkki	vastaava taajuus (Hz)
0	18000
1	18128
2	18256
3	18384
4	18512
5	18640
6	18768
7	18896
8	19024
9	19152
A	19280
B	19408
C	19536
D	19664
E	19792
F	19920

3.4.1 ClockSignalGenerator-komponentti

ClockSignalGenerator (CSG) -komponentin tehtävänä on luoda synkronointiin käytettävä kellosignaali. Kellosignaalin sisällytetään aikaleimoja UNIX-aika formaatissa. UNIX-aika ilmaisee kuluneiden sekuntien määrän ajankohdasta 1. tammikuuta 1970 kello 00:00:00 UTC. Aikaleiman sisällytykseen käytetään aliluvussa 2.1.1 esiteltyä FSK-menetelmää. Binäärisekvessin sijaan käytetään ennalta määrättyjä kuuttatoista taajuutta. Aikaleimat sijoitetaan signaaliin hexadesimaaliformaatissa. Jokaista heksadesimaalimerkkiä vastaava taajuus on määritetty taulukon 3.1 mukaisesti.

Taajuudet on valittu taajuusalueen yläpäästä siten, että ne on mahdollisimman huonosti ihmiskorvalla havaittavissa. Tutkimusten perusteella ihmiskorvalla on mahdollista kuulla maksimissaan 20 kHz taajuuksia. Jotta Opus-koodekkia voidaan tukea, täytyy taajuuksien kuitenkin olla alle 20 kilohertsiä. 128 hertsin erotus on saatu FFT-algoritmin mahdollistamasta tarkkuudesta erotella taajuuksia.

UNIX-aikaleimaan tarvitaan heksadesimaalimuodossa kahdeksan merkkiä. Kahdeksalla merkillä voidaan maksimissaan ilmaista aikaleima 07.02.2106, joten tarvetta yhdeksälle merkille ei ole. Kahdeksan merkin lisäksi aikaleiman alkuun lisätään aina yksi merkki varmistusmerkkinä, jonka tunnistamalla voidaan aikaleiman laskeminen aloittaa. CSG-komponentille voidaan konfiguroida, kuinka kauan yhden merkin toisto kestää eli digitaalisen äänen tapauksessa käytetään termiä näytettä merkkiä kohden (engl. samples per



Kuva 3.3. *Spektogrammi aikaleiman ajoituksesta*

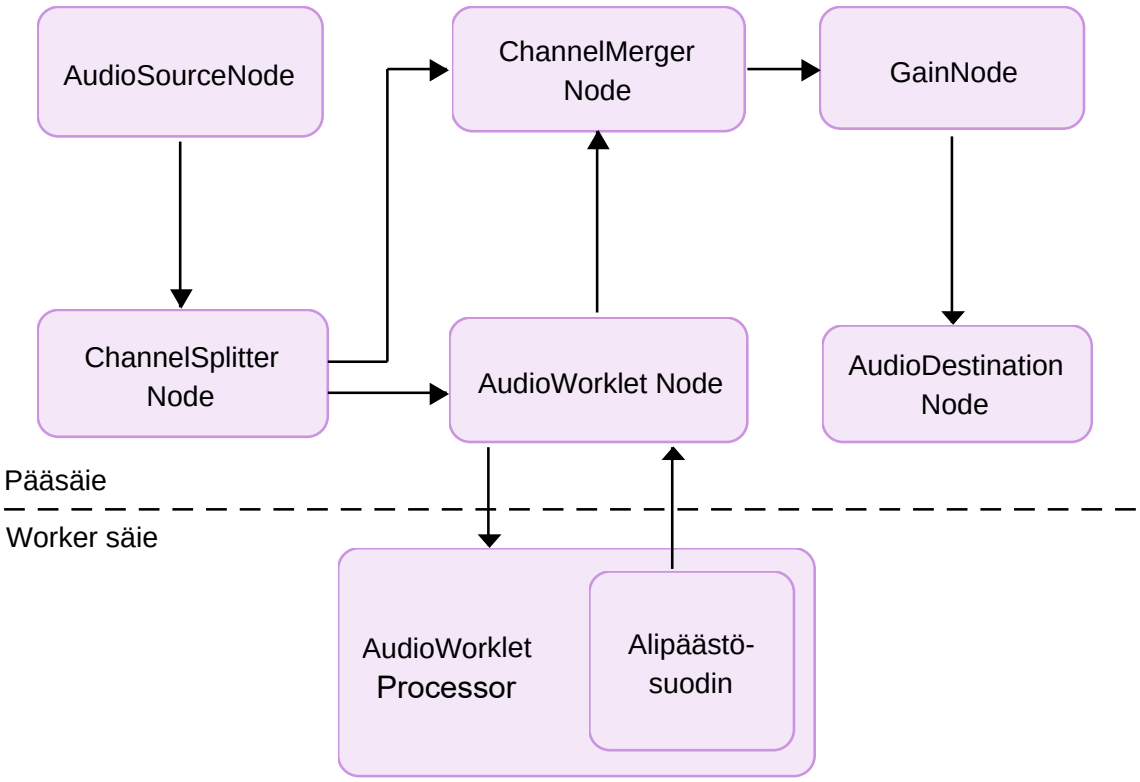
character).

Aikaleimaan käytettävät merkit asetetaan signaaliin siten, että viimeistä merkkiä vastaava taajuus on synkronoitu merkkisarjan sisältämään aikaleimaan. Esimerkiksi jos lisättävä aikaleima on 5C2A9160, mikä vastaa ajankohtaa 01.01.2019 00:00, viimeistä merkkiä vastaava taajuus lisätään äänisignaaliin vastaavana ajanhetkenä. Kuvassa 3.3 havainnollistetaan aikaleiman ajoitus spektogrammin avulla. Aikaleiman yhdeksän merkin vaatimuksesta seuraa, että jokaista taajuutta voidaan maksimissaan toistaa 111 millisekuntia. Koska FFT-algoritmi vaatii näytteitä kahden potensseissa, tämä tarkoittaa maksimissaan 4096 näytettä merkkiä kohden 44100 hertsin näytteenottotaajuudella.

3.4.2 MediaSync-komponentti

MediaSync-komponentti on JavaScript-pohjainen komponentti, jonka tehtävä on lukea aikaleimat äänisignaalista ja tuottaa tarvittava ajastin, jonka avulla muut datalähteet synkronoidaan. Komponentti käyttää hyödykseen JavaScriptin Web Audio -rajapintaa. Rajapinnan avulla luodaan kuvan 3.4 mukainen äänenprosessointiliukuhihna (engl. pipeline). MediaSync-luokan rakentajalle annetaan toistettavan äänen näytteenottotaajuus, äänikanava, kuinka monta näytettä on käytetty merkkiä kohden ja HTML5-elementti, jonka ääntä analysoidaan.

Liukuhihna hyödyntää ChannelSplitter-rajapintaa. Rajapinnan avulla erotetaan toistettavasta äänestä yksittäinen äänikanava, johon aikaleimat on lisätty. Erotetut äänikehykset syötetään AudioWorklet-rajapintaa käyttävälle solmulle. Solmu syöttää saamansa kehykset toisessa säikeessä ajettavalle AudioWorkletProcessor-luokan toteutukselle. Au-



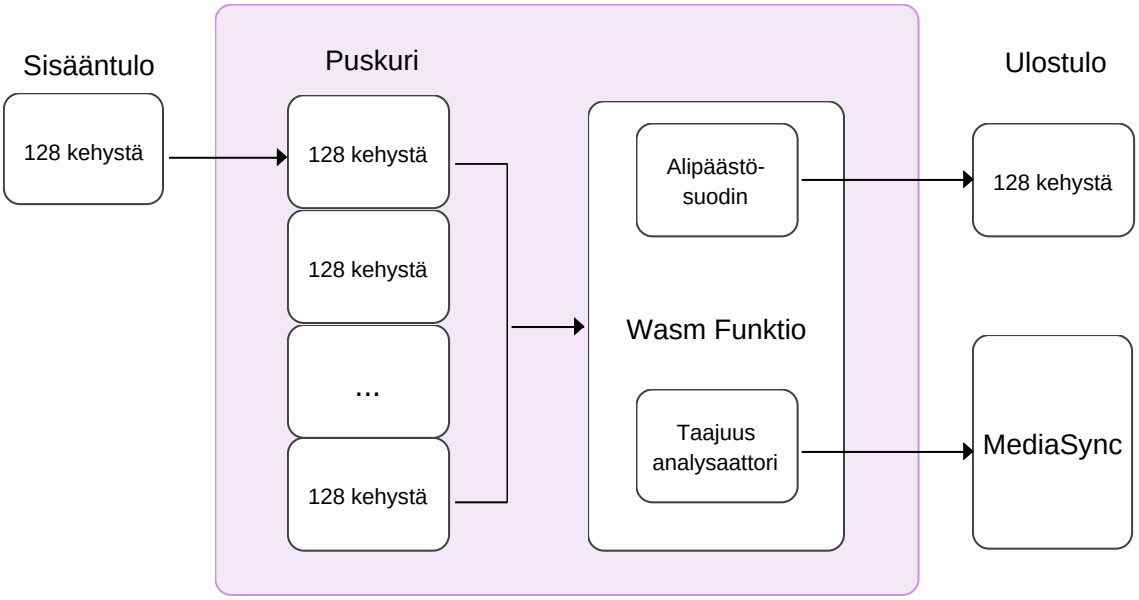
Kuva 3.4. *MediaSync-komponentin äänen prosessointikaavio ja rajapinnan jakautuminen kahteen säikeeseen*

dioWorkletProcessor suorittaa saamillensa näytteille alipäästösuodatuksen ja kopioi näytteet puskuriin. Suodatetut näytteet palautetaan AudioWorklet-solmulle, joka syöttää kehykset Channel-Merger-solmulle. ChannelMerger-solmu yhdistää suodatetun ja muut alkuperäiset äänikanavat samoihin kehyksiin. Nämä yhdistetyt kehykset työnnetään Gain-solmulle.

Gain-solmun tehtävänä on toteuttaa mykistystoiminnallisuus. Mikäli video-elementin ääni mykistetään käyttämällä HTML-videoelementin rajapintaa, äänikehyksiä ei työnnetä äänenprosessointitulosolmulle. Tästä seuraa analysointikaavion toimimattomuus, eikä synkronointi onnistu. Tämä asettaakin esiehdon videotoistimelle, jonka on estettävä äänen mykistys. Äänen vaimennuksella ei ole merkitystä, koska analysointivaiheessa tarkastellaan signaalien huippuarvoja. Vaimennuksessa huiput putoavat tasaisesti, joten vaimennuksella ei ole vaikutusta. MediaSync-komponentin rajapinnasta löytyvät mute- ja unmute-funktiot, joita videotoistin voi kutsua. Tällöin Gain-solmu vaimentaa kaikki äänet, eikä ulostulosolmu saa toistettavaa ääntä. Ääni on kuitenkin kiertänyt koko analysointirakenteen läpi, joten datalähteet saadaan synkronoitua.

AudioWorkletProcessor rakentaa saamistaan kehyksistä puskurin kuvan 3.5 mukaisesti. Puskurin pituuden määrittää, kuinka monta näytettä on käytetty yhtä merkkiä kohden. Puskuriin tallennetaan analysointia varten vastaava määrä. Puskuri annetaan parametrina WebAssemblyä hyödyntävälle funktiolle, joka ensiksi suorittaa alipäästösuodatuksen puskuriin viimeisimmäksi lisätylle 128 kehykselle. Alipäästösuodatus tehdään, jotta aika-

leimaan käytetyt taajuudet saadaan poistettua käyttäjälle toistetusta äänestä. Suodatetut 128 kehystä syötetään ulostuloon. Suorittamalla alipäästösuodatus heti puskuriin viimeiseksi liitetyille kehykselle, saadaan pienennettyä viivettä. Mikäli puskuri toimisi rengaspuskurimenetelmällä, aiheutuisi ääneen puskurin mittainen viive. 2048 näytteellä viive olisi vain 23 millisekuntia, joka olisi vielä vaikea huomata. Isommilla näytemäärillä viive kuitenkin kasvaisi, mikä ei ole haluttua.



Kuva 3.5. *Kehysten puskurointi analysointia varten ja analysointiin käytettävän Wasm-funktion toiminta.*

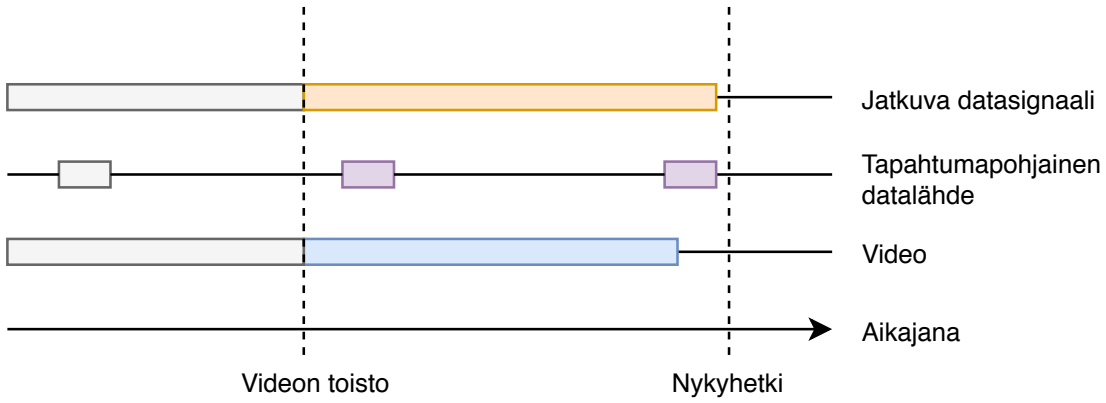
Wasm-funktio suorittaa myös aliluvussa 2.1.3 esitetyn FFT-muunnoksen. FFT-muunnoksen tulos on taulukko, jonka koko riippuu sille syötetystä näytemäärästä. FFT-muunnoksen koko on puolet syötteenä käytettyjen näytteiden määrästä. Taulukon arvot ilmaisevat signaalin voimakkuutta indeksiä vastaavalla taajuusalueella. Indeksiä vastaava taajuusalue määräytyy taulukon koosta eli syötetystä näytemäärästä. 2048 näytteellä FFT-muunnoksen koko on 1024 ja yksi taulukon arvo ilmaisee 21,5 hertsin alueella olevien taajuuksien voimakkuutta. Muunnoksesta keskitytään analysoimaan yli 18 kHz taajuuksia, johon aikaleimat on moduloitu.

FFT-muunnoksesta etsitään huippuarvo (engl. peak detection). Huippuarvo etsitään yhdistämällä FFT-muunnoksen alkioita toisiinsa kolme kerrallaan. Tällä tavoin yksi alkio ilmaisee 64,5 hertsin aluetta. Yhdistelemisellä saadaan poistettua signaalissa mahdollisesti olevaa häiriötä tai vääristymää. Määritetyt merkkejä vastaavat taajuudet ovat 128 hertsin välein. Löydettyä huippuarvoa eli taajuusväliä verrataan taulukossa 3.1 esitettyihin arvoihin. Mikäli tunnistettu taajuus on varmistusmerkkiä vastaava taajuus, tyhjennetään taulukko, johon tunnistetut merkit tallennetaan. Taajuusanalyysi voitaisiin suorittaa 128 näytteen välein. Tämä ei kuitenkaan paranna tunnistustulosta, sillä yksittäinen merkki toistetaan huomattavasti pidempään. Testaamisen perusteella päädyttiin tekemään taajuusanalyysi aina kun neljännes näytteistä, joita merkkiä kohden on käytetty, on päivittynyt. 2048 näytteellä tämä tarkoittaa 512 näytteen välein.

Suorittamalla taajuusanalyysi 512 näytteen välein yksittäistä merkkiä vastaava taajuus tunnistetaan useasti. Mikäli sama merkki esiintyy merkkijonossa useaan kertaan peräkkäin, määrä kasvaa. Toteutus pitää kirjaa, kuinka monen näytteen osalta taajuus on tunnistettu ja tämän perusteella tunnistetaan oikea määrä merkkejä. Ennalta on myös tiedossa, kuinka monta merkkiä pitää tunnistaa. Kun kahdeksan merkkiä on tunnistettu, analysaattori työntää tunnistetun aikaleiman AudioWorklet-solmulle, joka puolestaan MediaSync-komponentille.

MediaSync-komponentti päivittää saadun aikaleiman sisäisen ajastimensa alkuarvoksi. Alkuarvoa hyödyntämällä MediaSync-komponentti ylläpitää ajastinta, johon liitettävät datalähteet synkronoidaan. Ajastin hyödyntää setTimeout-funktiota aliluvussa 2.2.8 esitellyn logiikan mukaisesti. SetTimeout-silmukkaa suoritetaan 16,67 millisekunnin välein.

Komponentin ajastimen toiminta ja datalähteiden synkronointi on kuvattu kuvassa 3.6. Kuvan esimerkissä videossa on HTTP-suoratoistolle tyypillinen puskuri, jolla mahdollistetaan katkeamaton suoratoisto. Puskurin pituus voi vaihdella johtuen muutoksissa internetyhteyden laadussa ja sitä kautta tiedonsiirtonopeudessa. Ideaalitilanteessa videon toistaminen on katkeamatonta. Mikäli videon toisto kuitenkin katkeaa, aikaväli videon toiston ja nykyhetken välillä kasvaa. Puskurin lopun ja nykyhetken välillä on viive, joka riippuu käytetyistä koodaus- ja suoratoistomenetelmistä.



Kuva 3.6. *Ajastimen toiminta videon HTTP-teknologialla toimivassa suoratoistossa.*

Etenkin HTTP-teknologioita hyödyntävässä suoratoistossa datakomponentit vastaanottavat mallintamiseen tarvittavan datan huomattavasti ennen videon toistoa. Tapahtumapohjaisella datalla tarkoitetaan dataa, joka laukaisee asiakasohjelmalla jonkin ennaltamääritetyn tapahtuman. Asiakasohjelma voi esimerkiksi ladata ennakkoon diaesityksen, jonka diojen vaihtumista ohjaa videodatan lähettäjä. Diojen vaihtumiset synkronoidaan tapahtumapohjaisilla aikaleimoilla, joita asiakasohjelma vastaanottaa ennen videon toistoa. Asiakasohjelma suorittaa dian vaihtumisen, kun aikajanalta saadaan pakettia vastaava aikatieto. Jatkuvalla datasignaalilla tarkoitetaan jotain sellaista dataa, jonka vastaanottaminen on katkeamatonta. Tällainen käyttötarkoitus voisi olla esimerkiksi oskilloskoopin tai muun vastaavan signaalin tuottajan näyttäminen synkronoituna videoon.

MediaSync-komponentti tarjoaa rajapinnan, johon ulkoista dataa mallintava komponent-

ti voi kiinnittyä. Komponentti tuottaa synkronointiin tarvittavat tapahtuman timeupdate, johon dataa mallintava komponentti voi rekisteröidä kuuntelijan. Timeupdate-tapahtuma laukaistaan (engl. trigger) aina, kun MediaSync-komponentin sisäinen ajastin on kasvanut 16,67 millisekunnin verran, eli setTimeout-silmukka on pyörähtänyt. Tapahtuma sisältää aika-muuttujan, joka kertoo mallintavalle komponentille, mistä kohtaa ladattua puskuria data pitää mallintaa. 16,67 millisekunnin välein päivittämällä saavutetaan 60 kuvaa sekunnissa. Tämä mahdollistaa sulavan näköiset animaatiot.

Mikäli videon toistossa tulee katkos, ajastimen aikamuuttuja pysyy vakiona. Tällöin timeupdate-tapahtuma laukaistaan kaksi kertaa samalla pysähtyneellä aikaleimalla, jotta mallintava komponentti tietää toiston pysähtyneen. Tallenteen suoratoistossa käyttäjä voi hakea (engl. seek) jotain ajankohtaa videon aikajanalta. Tallenteissa videon kokonaispituus on tiedossa. Kun aikajana on saatu kerran synkronoitua, voidaan siitä hakea vastaavaan tapaan.

4 TOTEUTUKSEN TOIMINNALLISUUDEN EVALUOINTI

Toteutuksen soveltuvuus ongelman ratkaisuun haluttiin varmistaa jo prototyypin kehittelyvaiheessa. Ennen järjestelmän lopullisen toteutuksen tekemistä erinäisistä epävarmuustekijöistä haluttiin saada parempi käsitys. Tälläisia epävarmuustekijöitä ovat esimerkiksi kooderit ja alkuperäisen äänen vaikutus analysointiin, joiden vaikutuksesta ei voida olla varmoja.

Yleisen sovelutuvuuden lisäksi testaamalla haluttiin määrittää optimaaliset parametrit, joilla järjestelmää voidaan lähteä kehittämään jatkossa. Tässä luvussa käydään läpi edellä mainittuun prosessiin käytetyt työkalut ja kuvaillaan testien suoritustapa.

4.1 Testausympäristö ja työkalut

Epävarmuustekijöiden vaikutukset pystytään testaamaan käyttämällä äänen ja videon prosessointiin tarkoitettuja avoimia työkaluja. Nämä työkalut sisältävät tarvittavat menetelmät toteutuksen toiminnallisuuden evaluointiin.

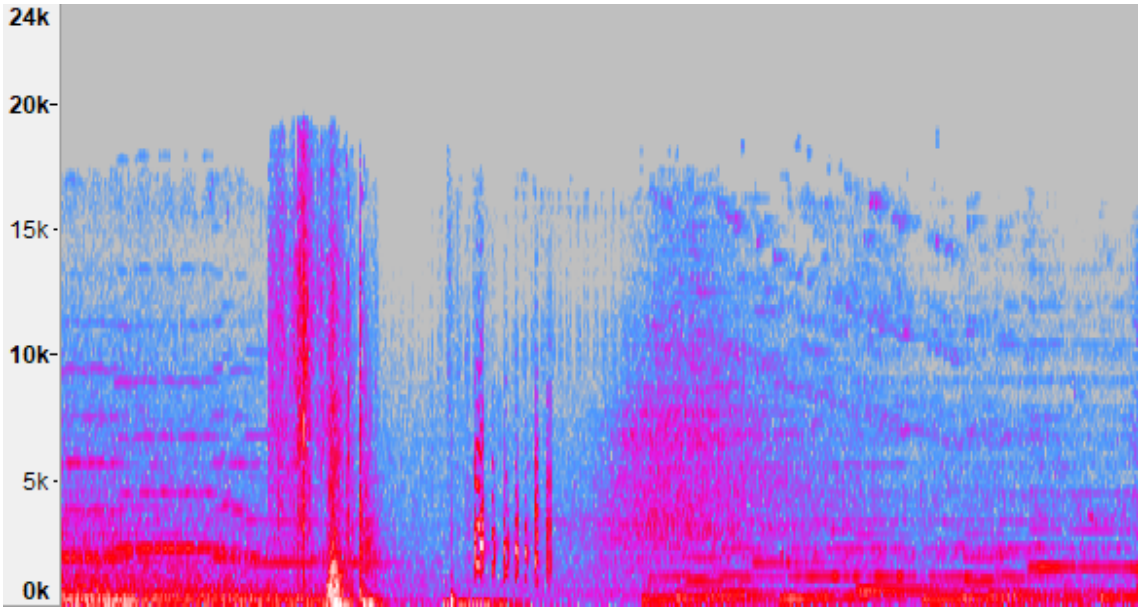
FFmpeg on multimedian hallintaan käytettävä ohjelmistokehys, jolla voi muun muassa koodata, dekoodata ja muokata ääni- ja videotiedostoja erilaisten kirjastojen avulla. FFmpeg sisältää tuen kaikille työssä mainituille koodausteknologioille. FFmpegiä käytetään testauksessa videon ja äänen koodaamiseen sekä aikaleimatun datasignaalin miksaamiseen. FFmpegiä voidaan käyttää komentoriviltä. Se tukee valtavan määrän erilaisia komentoriviparametreja. Testauksen kannalta oleellisia näistä olivat seuraavat parametrit:

- cutoff, joka määrittää koodattavalle äänelle alipäästösuodattimen
- filter_complex, jolla voidaan määrittää kompleksisia suodattimia, joiden avulla voidaan esimerkiksi miksata useampi äänisignaali yhdeksi
- ac, joka määrittää kuinka monta äänikanavaa lopullisessa äänessä on
- b:a, jolla voidaan määrittää äänelle haluttu tiedonsiirtonopeus
- ar, jolla voidaan määrittää äänisignaalissa käytetty näytteenottotaajuus.

Äänisignaalien ja videon ääniraitojen tulkitsemiseen käytettiin avoimen lähdekoodin ohjelmaa nimeltään Audacity. Audacityllä on monipuolisesti erilaisia työkaluja, kuten suodat-

timia ja erilaisia visualisointeja. Visualisoinnit auttavat luodun äänidatan olemassaolon ja laadun varmistamisessa.

Reaaliaikaisen videon sijaan testeissä käytetään vapaasti käytettävissä olevaa lyhytelokuvaa Big Buck Bunny. Elokuva valikoitui testattavaksi vapaan käytön lisäksi sen ääniraidan monipuolisuuden perusteella. Kuvassa 4.1 näkyy alkuperäisen elokuvan ääniraidasta otettu spektogrammi hyödyntäen aiemmin mainittua Audacity-ohjelmaa. Elokuvassa esiintyy taajuuksia lähes 20 kHz asti. Alkuperäinen ääniraita on viisikanavainen. Se muutettiin testejä varten kaksikanavaiseksi FFmpeg-työkalua hyödyntämällä.



Kuva 4.1. *Audacity-ohjelman mallintama Big Buck Bunny -elokuvan vasemman äänikanavan spektogrammi.*

FFmpegillä suoritettava koodaus on laskennallisesti raskasta. Elokuvan videon tarkkuudella ei ole merkitystä komponentin toimintaan. Yleisesti muutokset pelkkään ääniraitaan saadaan suoritettua nopeasti. Opus-koodekkia käyttäessä ilmeisesti myös videon koodaus suoritettiin uudestaan. Tämä voi johtua Opus-koodekkikirjaston toteutuksen tuoreudesta ja MP4-säiliömuodosta, jonka tuki on kokeellinen. Lyhempien koodaussuoritusaikojen takia, elokuvasta käytettiin tarjolla olevista laaduista heikointa eli 854x480-resoluutioista versiota.

Kellosignaalin tuottavan komponentin toteuttamisen sijaan aikaleimattu datasignaali generoidaan hyödyntäen NumPy-nimistä ohjelmointikirjastoa. NumPy on Python-kielinen ohjelmointikirjasto, joka on tarkoitettu tieteelliseen laskemiseen. Kirjasto sisältää signaalien generoimiseen tarvittavia funktioita ja rakenteita, joista voidaan tehdä pakkaamatonta äänidataa. Ääni generoidaan siniaaltona, jonka taajuus vaihtelee koodattavan merkin mukaisesti.

Reaaliaikaisten aikaleimojen sijaan testauksessa käytettiin ennaltamääritettyjä satunnaisesti generoituja 11-merkkisiä heksadesimaalimerkkijonoja. Näistä saatava äänisignaali miksattiin videon alkuperäiseen ääniraitaan FFmpeg-työkalun avulla. Merkkijonojen jär-

```
1  # Dict to describe the character frequency relation
2  FREQ_DICT = {'0': 20000,  '1': 20128,  ...,  'f': 21920}
3
4  hexval = ['171bbbea2fd'] # To be added to the signal
5  spc = 2048 # Samples per char
6  sr = 48000 # Sample rate
7
8  ## Fill with silence for the beginning of the second
9  # Use the first value of the generated sin-signal
10 freq = FREQ_DICT[hexval[0]]
11 s = 100 * np.sin(2 *np.pi * freq * np.arange(1) / sr )
12 sl = sr - y.size%sr - spc*len(hexval) # Silence length
13 y = np.append(y,np.repeat(s[0], sl ))
14 x = np.arange(spc)
15
16 for h in hexval:
17   freq = FREQ_DICT[h]
18   y = np.append(y, 100 * np.sin(2 * np.pi * freq * x / sr ))
```

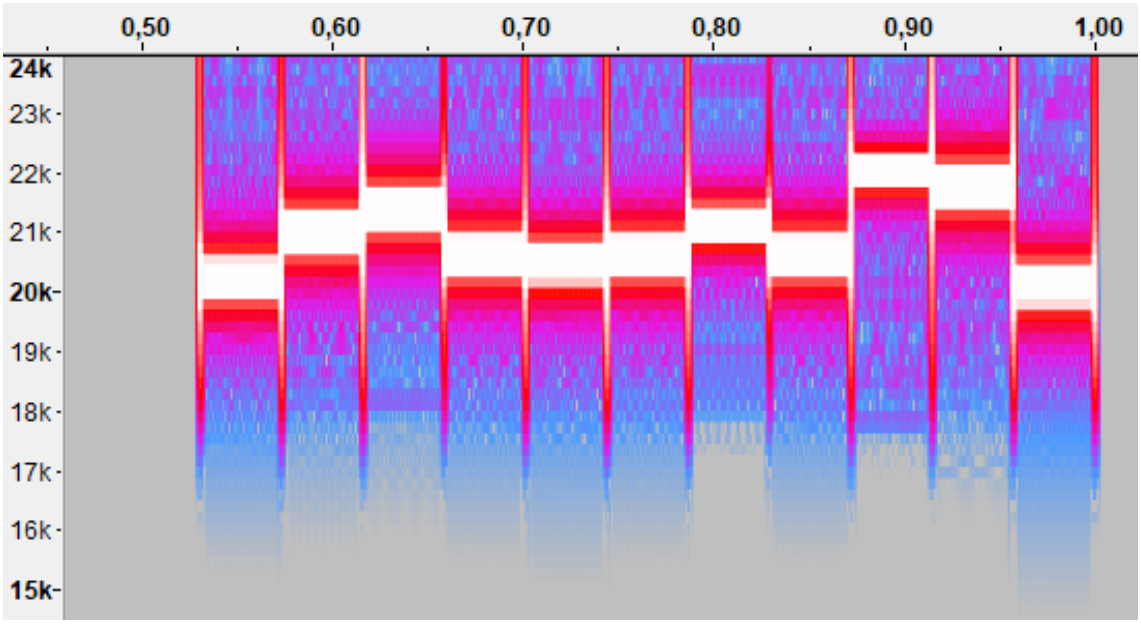
Ohjelma 4.1. *Raa'an äänisignaalin generoiminen käyttäen NumPy-kirjastoa.*

jestys ja aikaleimat ovat tiedossa, joten niiden tunnistustarkkuutta voidaan verrata MediaSync-komponentissa. Ohjelma 4.1 esittää äänisignaalin generoimiseen käytetyn ohjelmakoodin osan, jolla raaka äänisignaali generoidaan.

Signaali aloitetaan luomalla hiljaisuutta, eli tasaista signaalia. Hiljaisuuden pituus riippuu, kuinka monta näytettä käytetään yhtä heksadesimaalimerkkiä kohden. Hiljaisuuden pituuden laskenta on havainnollistettu ohjelman 4.1 rivillä 12. Tämän jälkeen käydään heksadesimaalimerkkijono merkki kerrallaan läpi for-silmukassa. Jokaista merkkiä vastaavaa taajuutta lisätään äänisignaaliin muuttujan spf-verran. Tämä ohjelmakoodi suoritetaan jokaiselle määritellylle merkkijonolle. Tällöin saadaan yhtämittainen äänisignaali, johon on sijoitettu merkkijonoja analysoitavaksi.

Merkkijonon taajuudet haluttiin ajoittaa loppumaan sekunnin tarkkuudella. Numpy-kirjasto hoitaa signaalin generoimisen ja näytteistämisen. Saadut arvot ovat etumerkittömiä 8-bittisiä lukuja, jotka kuvaavat signaalin amplitudia. Käyttäen 2048 näytettä merkkiä kohden ohjelmalla 4.1 saadaan generoitua ääni, jonka spektogrammi on ylipäästösuodatuksen jälkeen kuvan 4.2 mukainen. Ylipäästösuodatus tehtiin, jotta signaalin generoinnin yhteydessä syntyneet kuultavissa olevat äänet saatiin pois. Kuultavissa olevien äänien syntyminen on seurausta signaalin näytteistämisestä. Ylimääräisen äänidatan poistamisella saatiin myös miksattava äänidata pienemmäksi, jolloin kooderin on helpompi pakata ääntä. Ylipäästösuodatukseen käytettiin Audacity-ohjelmaa.

Spektogrammista huomataan, että yksittäinen merkki näkyy noin 500 Hz taajuusvälilllä. Tämä on seurausta laskostumisesta. Näytteenottotaajuus ei riitä ilmaisemaan signaalia riittävän tarkasti. Tämä ei kuitenkaan haittaa taajuuden tunnistamista, koska haluttu taajuus on amplitudiltaan suurin ja spektogorammissa näkyvän kirkkaan alueen keskellä.



Kuva 4.2. *Testeissä käytetyn FSK-koodatun äänisignaalin spektogrammi käyttäen 2048 näytettä merkkiä kohden.*

Merkin vaihtuessa spektogrammissa näkyy kirkas pystyviiva. Tämä on seurausta tavasta, jolla siniaallot yhdistetään. Taajuuden vaihtuessa signaalia ei muuteta hallitusti, vaan muutos on äkillinen, josta seuraa voimakas amplitudin muuttuminen. Testeissä tätä ei kuitenkaan kuule, koska äänet on korkeilla taajuuksilla. Muutos on myös niin lyhyt, ettei se haittaa äänen analysointia MediaSync-komponentilla. Tämä on kuitenkin hyvä tiedostaa ClockSignalGenerator-komponenttia toteuttaessa, jossa taajuuden vaihtuminen kannattaa tasoittaa käyttämällä merkin vaihtumiseen muutama näyte.

Analyysin suorittamiseen käytettiin Google Chrome selainta. Google Chrome valittiin, koska se on selaimista ainoa, joka toteuttaa AudioWorklet-rajapinnan täysin ilman tarvetta erilliselle polyfill-kirjastolle, ja täten mahdollistaa yksinkertaisemman toteutuksen.

Eri suoratoistoprotokollien testaamiseen käytettiin Wowza Streaming Engine -ohjelmistoa. Ohjelmistolle voidaan määrittää tiedosto, jota se toistaa. Suoratoisto voidaan määrittää joko live- tai tallennetilaan. Live-tilassa kaikille yhdistetyille katselijoille toistetaan videota samasta kohtaa. Tallenteen suoratoistossa videon toistaminen alkaa aina videon alusta ja käyttäjä voi hakea videosta eri kohtia. Koska selaimet eivät tue suoratoistoprotokollia sellaisenaan, päädyttiin käyttämään videojs-toistinkirjastoa. Kirjasto on avoimesti lisensoitu ja siinä on tuki HLS- sekä MPEG-DASH-suoratoistoprotokollille.

4.2 Testauksen pohjalta määritettävät parametrit

Toteutuksen toimivuuteen vaikuttavat videon prosessoinnissa käytetyt parameterit. Parametreilla tarkoitetaan sellaisia muuttujia, joilla voidaan vaikuttaa toteutuksen toimintaan. Parametreja voidaan muuttaa aliluvussa 4.1 määritetyillä FFmpeg komentoriviparamet-

reillä tai muuttamalla ohjelmassa 4.1 esitettyjä muuttujia.

4.2.1 Tiedonsiirtonopeus

Tiedonsiirtonopeus ilmaisee, kuinka paljon tietoa siirtyy ajan kuluessa. Mitä pienempi tiedonsiirtonopeus, sitä vähemmän verkkoyhteydeltä vaaditaan. Suoratoistossa tiedonsiirtonopeus pyritään minimoimaan, kuitenkin siten, ettei tallenteen laatu kärsi merkittävästi. Taajuusanalyysin näkökulmasta pyritään löytämään sellainen tiedonsiirtonopeus, jolla tunnistettavat taajuuden löydetään.

Tiedonsiirtonopeuteen voidaan vaikuttaa koodausprosessissa. Koodausprosessille voidaan määrittää haluttu tiedonsiirtonopeus. Kooderi ei takaa tiedonsiirtonopeuden pysymistä vakiona, vaan kooderi pyrkii pakkaamaan äänen siten, että sen keskiarvollinen tiedonsiirtonopeus on lähellä annettua. Hetkellinen tiedonsiirtonopeus riippuu koodattavasta äänisignaalista.

Koodaus on yleisesti käytetyillä koodereilla häviöllinen tapahtuma. Tämä tarkoittaa, ettei dekoodausprosessissa saada tuotettua alkuperäistä vastaavaa äänisignaalia. Toteutuksen kannalta tämä on kriittistä, koska taajuusalueen, jolle merkit on lisätty, äänet halutaan säilyttää mahdollisimman alkuperäisinä, jotta niistä voidaan tunnistaa aikaleimat. Applen suositus on AAC-koodatulle äänelle 32-130 kb/s [8]. Ihmisen puheessa ei normaalisti esiinny yli 500Hz taajuuksia, ja 32 kb/s tiedonsiirtonopeudella koodattua ääntä käytetäänkin yleisesti ääni- ja videokeskustelujen siirrossa [27]. Tiedonsiirtonopeudet testataan aloittamalla suositellun asteikon yläpäästä asteittain pudottamalla 64 kb/s asti. Tallenteessa on kaksi äänikanavaa, joiden sisältö on lähes identtinen.

4.2.2 Taajuusalue

Aikaleiman piilotukseen pyritään löytämään mahdollisimman optimaalinen taajuusalue. Optimaalisella taajuusalueella tarkoitetaan sellaista taajuusaluetta, joka selviää koodausprosessista mahdollisimman hyvin, taajuusalueen äänet eivät ole ihmiskorvalla kuultavissa ja alue on riittävän iso eri taajuuksien toisista erottamiseen.

Kuten aliluvussa 2.1.1 todettiin, ihmisen maksimi kuuloalue on 20-20000 Hz. Yleisesti käytetyllä 44100 Hz näytteenottotaajuudella signaalin maksimitaajuus on 22050 Hz. Taajuusalue pyritään löytämään väliltä 20000-22050 Hz. Taajuusalueen aivan yläpäätä ei käytetä mahdollisen laskostumisen takia.

Tarvittavat 16 taajuutta pyritään levittämään mahdollisimman laajalle jäljelle jäävälle taajuusalueelle, jotta ne voidaan erottaa helpommin. Mikäli testauksessa huomataan, että jotkin taajuudet sekoittuvat toistuvasti toisiinsa tai niitä ei havaita ollenkaan, voidaan eroja kasvattaa.

Eri tutkimusten perusteella keskiverto ihmiskorva ei kuule yli 16 kHz taajuuksia. Taajuusalue voitaisiin valita väliltä 16000-22050 Hz. Tällöin olisi kuitenkin tarpeen toteuttaa alipäästösuodin, etteivät korkeat äänet häiritse pientä marginaaliosaa kuuntelijoista. Alipäästösuotimen toteuttaminen päätettiin jättää prototyypin toteutusvaiheessa pois.

4.2.3 Näytteenottotaajuus

Näytteenottotaajuuksista testataan pienin yleisesti käytetty 44100 Hz ja hieman isompi 48000 Hz. Kolmas yleisesti käytetty 96000 Hz jätetään testaamatta, koska se on tarpeettoman suuri suoratoistamiseen.

Näytteenottotaajuudella oletetaan olevan pienin vaikutus lopputulokseen, sillä pieninkin testattu näytteenottotaajuus estää käytettävien taajuuksien laskostumisen. Tämä kuitenkin varmistetaan optimoimalla muut parametrit ensiksi 48000 hertsin näytteenottotaajuudella. Tämän jälkeen muita parametreja muuttamatta testataan 44100 hertsin näytteenottotaajuus, jotta oletus saadaan testattua todeksi.

4.2.4 Näytettä merkkiä kohden

Näytettä merkkiä kohden ilmaisee, kuinka monta näytettä käytetään kuvaamaan yhtä merkkiä eli kuinka kauan yksi taajuus toistetaan. Aliluvussa 2.1.3 mainitaan yhden kehyksen sisältävän 128 näytettä kanavaa kohden. 128 näytettä 44100 Hz näytteenottotaajuudella on 2 millisekunnin mittainen ääni. Tästä seuraa, että koko aikaleiman toistamiseen kuluisi 24 millisekuntia. Tämä asettaa rajoituksen, ettei taajuuksia voida tarkastella pienemmistä näytemääristä.

Toisen rajoituksen parametrille asettaa käytettävä taajuuden tunnistustekniikka. Koska tunnistukseen käytetään FFT-algoritmia, näytteistä on tunnistettava taajuuksia yhtä useasti kuin näytteitä on käytetty merkkiä kohden. Suorittamalla FFT-algoritmi 128 näytteen erälle, FFT:n taajuuskoreja syntyisi 64 kappaletta. Taajuuskori on koko taajuusalueen osa, johon kerääntyy alueen kaikkien taajuuksien amplitudien summa. Se määrittää kuinka tarkasti FFT-menetelmällä voidaan erotella taajuuksia. 64 koria 22050 hertsin alueella tarkoittaa yhden korin edustavan 344 hertsin aluetta. Aliluvussa 4.2.2 todettiin merkkien olevan koodattu 128 hertsin välein, joten 128 näytteen erästä taajuuksia ei voida erottaa toisistaan. 512 näytteellä saadaan 78 hertsin frekvenssikorit. Tämä riittää 128 taajuuksien erottamiseen toisistaan. Arvoa kasvatetaan kahden potensseina alkaen arvosta 512 aina arvoon 4096.

Käyttämällä 4096 merkkiä näytettä kohden yhdeksän merkin tuottamiseen tarvitaan 36 864 näytettä. Näytettä merkkiä kohden parametri pyritään minimoidaan testaamalla eri konfiguraatiota yleisesti käytettyjä koodekkeja vasten. Minimoimalla aikaleiman toistamiseen käytetty aika saadaan kellosignaaliin tarvittava tiedonsiirtonopeus mahdollisimman

Taulukko 4.1. *Testausmenetelmä*

tiedonsiirtonopeus (kb/s)	näytettä merkkiä kohden	näytteenottotaajuus
128	4096	48000
115	2048	44100
102	1024	
89	512	
76	256	
64		

pieneksi. Tällöin alkuperäiselle äänisignaalille jää isompi osa käytettävissä olevasta tiedonsiirtonopeudesta.

4.3 Testitapahtuman kuvaus

Prototyyppiä testattiin kahdesta eri näkökulmasta. Ensimmäinen näkökulma oli, kuinka suurella todennäköisyydellä piilotettu aikaleima saadaan luettua videon äänisignaalista. Toinen tarkastelukohta oli kuinka hyvin toteutus sietää eri koodereita.

Testattavat parametrit on koottu taulukkoon 4.1. Testattavista parametreista ensiksi tarkasteluun otettiin näytettä merkkiä kohden. Python ohjelmalla 4.1 tuotettiin 600 sekuntia pitkä äänisignaali, joka sisältää 11 merkin mittaisia ennalta määrättyjä heksadesimaalimerkkijonoja. Merkkijono lisättiin signaaliin siten, että viimeisen merkin toistaminen loppuu tasasekunnilla. Tästä seuraa 600 piilotettua heksadesimaalimerkkijonoa. Määrä oli seurausta videon pituudesta, joka oli 10 minuuttia ja 14 sekuntia. Määrän koettiin olevan riittävän suuri, jotta taajuusanalyysin toimivuus voidaan todeta ja tarkkuudesta saadaan tilastollisesti järkevä. Edellä mainittu äänisignaali luotiin jokaista taulukossa 4.1 esitettyä näytettä merkkiä kohden parametriä kohden. Tällä tavoin saadut viisi äänisignaali miksattiin alkuperäiseen AVC/AAC-koodatun videon ääniraitaan käyttäen FFmpeg-työkalua. Lopputulos oli viisi videota, joiden ääniraidat sisälsivät eri parametrin arvoilla tuotettua äänisignaalia.

Näytettä merkkiä kohden parametrin määritykseen käytettiin tiedonsiirtonopeutena testattavista suurinta 128 kb/s. Tämän tarkoituksena oli eliminoida tiedonsiirtonopeuden vaikutus analyysiin. Kun riittävällä tarkkuudella toimiva näytettä merkkiä kohden parametri on tunnistettu, testattiin tiedonsiirtonopeuden vaikutus. Videon ääni koodattin eri tiedonsiirtonopeuksilla FFmpeg-työkalua hyödyntäen ja yllä mainitut testit suoritettiin uudestaan kyseisille videoille. Lopuksi videon alkuperäinen ääni alinäytteistettiin käyttämään pienempää 44100 hertsin näytteenottotaajuutta. Python ohjelmalla luotiin 44100 hertsin näytteenottotaajuudella vastaava aikaleimat sisältävä äänisignaali, jolle suoritettiin edellä mainitut testit.

JavaScriptillä toteutettu taajuusanalyysi-komponentti rekisteröitiin tunnistamaan merkkijonoja. Komponentille konfiguroidaan kyseiselle videolle asetettu näytettä merkkiä kohden -parametri ja toistettavan videon näytteenottotaajuus. Komponentti aloittaa tunnistamalla hiljaisuuden taajuusalueen yläpään taajuuksista. Kun yläpään taajuusalue aktivoidaan, komponentti pyrkii tunnistamaan heksadesimaalimerkkejä vastaavat taajuudet. Jokaista taajuutta vastaava heksadesimaalimerkki kirjataan ylös. Kun hiljaisuus taas tunnistetaan, kirjoitetaan saatu heksadesimaalimerkkijono muistiin. Videon loputtua komponentti tulostaa tunnistamansa heksadesimaalimerkkijonot.

Kaikki edellä mainitut testit ajetaan AAC-koodekilla koodattuun ääneen. Edellä mainitun testauksen jälkeen toteutus testataan vielä eri äänikoodekeilla. Kaikkia testitapauksia ei ole tarkoitus toistaa jokaisella kooderilla, vaan ensimmäisten testien optimaaliset parametrit testataan muita koodereita vastaan. Mikäli toteutuksen tarkkuus eli aikaleimojen tunnistusprosentit putoavat merkittävästi, testataan parametrien vaikutusta muillakin koodereilla. Koodekeista testataan aiemmin mainitut yleisesti käytetyt Opus ja Vorbis.

Kun kaikki aiemmin mainitut testit on suoritettu, testataan toteutuksen toiminnallisuus MPEG-DASH- ja HLS-tekniikoita käyttävässä suoratoistossa. Suoratoistoon käytetään aiemmin mainittua Wowza Streaming Engineä.

5 TULOKSET JA ARVIOINTI

Tässä luvussa käydään läpi luvussa 4 esiteltyjen testitapausten tulokset ja analysoidaan, miten niiden perusteella lopullista toteutusta kannattaa lähteä toteuttamaan. Luvun toisessa aliluvussa arvioidaan lopullisen toteutuksen onnistumista ja pohditaan mahdollisia kehitysvaihtoehtoja.

5.1 Prototyyppitestauksen tulokset

Jokaisen parametrin vaikutus käydään läpi tarkastelemalla aikaleimojen tunnistamisen onnistumista. Aikaleimojen tunnistamisen lisäksi tarkastellaan yksittäisen taajuuden oikeaa tunnistamista. Aikaleiman oikea tunnistaminen vaatii 11 peräkkäistä onnistunutta taajuuden tunnistamista.

5.1.1 Näytettä merkkiä kohden

Parametrin vaikutus taajuuksien ja aikaleiman tunnistukseen on koottu taulukkoon 5.1. Paras tunnistusprosentti saatiin käyttämällä 2048 näytettä merkkiä kohden. Testituloksista huomattiin, että 2048 näytteellä virheitä tapahtui lähinnä alimpien taajuuksien tunnistamisesta. Alkuperäisessä äänisignaalissa oli hetkittäin niin korkeita ääniä, että taajuusanalyysi tulkitsi taajuuden astetta alemmaksi. 4096 merkillä saatiin lähes vastaava tulos, mikä oli myös oletettavaa. 2048 näytteellä aika, jolloin signaalia ei tulkita, on suurempi kuin 4096 näytteellä. Tällöin alkuperäisen äänisignaalin vaikutus voi olla pienempi, ja se voi selittää pienen eron tunnistusprosenteissa. Tämän varmistamiseksi testejä olisi hyvä suorittaa erilaisilla alkuperäisillä ääniraidoilla.

1024 näytteellä taajuuksia tunnistettiin edelleen hyvällä prosentilla. Merkit, joita ei tunnistettu jakautuivat hyvin tasaisesti eri aikaleimoihin, jolloin aikaleiman tunnistusprosentti on merkittävästi huonompi. Pienemillä 512 ja 256 näytemäärillä myös taajuuden tunnistusprosentti oli merkittävästi heikompi. Niillä ei saatu tunnistettua yhtään kokonaista aikaleimaa. Tämä on todennäköisesti seurausta signaalin taajuuden suuresta varianssista, joka on seurausta signaalin laskostumisesta. Pienellä näytemäärällä taajuuksia on hankalampi tunnistaa käyttäen FFT-menetelmää.

Tunnistusprosenttia voitaisiin parantaa erilaisilla menetelmillä. Yksi helposti hyödynnettä-

Taulukko 5.1. *Näytettä merkkiä kohden parametrin vaikutus aikaleimojen ja taajuuksien tunnistamiseen*

näytettä merkkiä kohden	4096	2048	1024	512	256
taajuuden tunnistus-%	98,3	98,4	93.8	65.5	4,8
aikaleiman tunnistus-%	95	96,67	61.7	0	0

Taulukko 5.2. *Koodauksessa käytettävän tiedonsiirtonopeuden vaikutus aikaleimojen tunnistustarkkuuteen*

tiedonsiirtonopeus (kb/s)	taajuuden tunnistus-%	aikaleiman tunnistus-%
128	98,4	96,7
115	96,9	95,0
102	96,8	95,0
89	96,8	93,3
76	96,8	93,3
64	98,6	95,0

vissä oleva menetelmä olisi heikentää alkuperäisen äänisignaalin korkeita taajuuksia. Tämä voitaisiin tehdä kuvausvaiheessa alipäästösuotimella. Alipäästosuotimen rajana voitaisiin käyttää esimerkiksi 16 kilohertsiä, jonka ylittäviä ääniä keskiverto ihmiskorva ei kuule.

5.1.2 Tiedonsiirtonopeus

Tiedonsiirtonopeuden vaikutus aikaleimojen tunnistamiseen tutkittiin testaamalla taulukossa 4.1 esitettyjä arvoja vastaan. Näissä testeissä päädyttiin käyttämään edellisessä aliluvussa 5.1.1 parhaimmaksi todettua 2048 näytteen määrää merkin koodaamiseen. Ääni koodattiin käyttäen AAC-koodekkia. Testien tulokset on esitetty taulukossa 5.2.

Tiedonsiirtonopeuden vaikutus tunnistuksen onnistumiseen oli vain muutaman prosentin luokkaa. Tunnistusprosentti putosi tasaisesti pienemmillä tiedonsiirtonopeuksilla, pois lukien 64 kb/s tiedonsiirtonopeudella. 64 kb/s tiedonsiirtonopeudella koodatessa äänisignaalista hävisi selkeästi enemmän matalataajuuksien ääniä eli alkuperäistä äänisignaalia. Tämä oli selkeästi havaittavissa kuuntelemalla. Kooderi selkeästi keskittyi pakkaamaan piilotetun aikaleiman paremmin kuin alkuperäisen äänisignaalin. Tämä voi johtua aikaleima signaalin isommasta amplitudista suhteessa videon alkuperäiseen äänisignaaliin.

Kuuntelemalla 128 kb/s ja 64 kb/s tiedonsiirtonopeuksilla koodattuja äänisignaaleja, ero oli selkeä. Optimaalisen tiedonsiirtonopeuden löytämiseen pitäisi kuitenkin tehdä useamman kuuntelijan testaus. Tiedonsiirtonopeuden vaikutus äänenlaatuun on moninainen ongelma ja vahvasti sidoksissa käytettyyn koodekkiin.

Taulukko 5.3. *Tunnistuksen testaus pienemmällä 44,1 kHz:n näytteenottotaajuudella.*

näytettä merkkiä kohden	2048	1024
taajuuden tunnistus-%	98,3	93.4
aikaleiman tunnistus-%	96,8	63.4

Taulukko 5.4. *Tunnistuksen testaus yleisesti käytetyillä ääni-koodekeilla.*

koodekki	Vorbis	Opus
taajuuden tunnistus-%	98,3	98.6
aikaleiman tunnistus-%	95.0	98.3

5.1.3 Näytteenottotaajuus ja taajuusalue

Kaikki aiemmat testit suoritettiin 48 000 hertsin näytteenottotaajuudella. Toteutuksen toimiminen toisella yleisesti käytetyllä näytteenottotaajuudella 44 100 hertsillä varmistettiin testaamalla. Edellä mainituista testeistä valittiin toimivat parametrit ja testattiin näytteenottotaajuus niillä. Nämä parametrit ja tulokset on esitetty taulukossa 5.3.

Tuloksista nähdään, ettei merkittävää vaikutusta ole. Tämä on seurausta näytteenottoteoreeman seuraamisesta valittaessa tunnistettavia taajuuksia. Järjestelmä saatiin toimimaan alun perin teoriapohjalta valittujen taajuusalueiden kanssa. Testauksen perusteella taajuusaluetta ei ole tarvetta laajentaa tai supistaa.

5.1.4 Koodauksen sieto

Koodauksen sieto testattiin käyttämällä äänen koodaamiseen yleisempiä suoratoistossa käytettyjä koodekkeja. Kaikki aiemmat testit suoritettiin AAC-koodekilla. Taulukossa 5.4 on esitelty tunnistustarkkuudet käyttäen Vorbus- ja Opus-koodekkia. Koodauksessa käytettiin 128 kb/s tiedonsiirtonopeutta ja äänisignaalissa 2048 näytettä ilmaisemaan yhtä merkkiä.

Testien suorittamisen aikana kävi ilmi, ettei Opus-koodekki mahdollista yli 20 kHz taajuuksia. Opus-testiin koodatun äänisignaalin taajuusalue pudotettiin 18-20 kHz välillle. Molemmissa tapauksissa merkit tunnistettiin riittävällä tarkkuudella ja ero AAC-koodattuihin on marginaalinen.

5.1.5 Suoratoistoprotokollien sieto

Prototyyppi saatiin onnistuneesti kiinnitettyä suoratoistettaviin videoihin. Aikaleimoja saatiin tunnistettua vastaavasti kuin tiedostoa toistamalla. Prototyyppitoteutus ei osannut ottaa huomioon toiston katkeamista. Mikäli toisto katkesi kesken aikaleiman lukemisen,

aikaleima luettiin väärin.

Tämä oli seurausta prototyyppitoteutuksen yksinkertaisesta toteutuksesta, jossa ei huomioitu kaikkia mahdollisia suoratoiston tiloja. Aikaleimat saatiin luettua katkeamattomasta suoratoistosta, joten testit koettiin onnistuneeksi.

5.1.6 Testien vaikutukset toteutukseen

Testauksessa havaittiin erilaisia ongelmia toteutuksessa, jonka pohjalta lopullista toteutusta lähettiin parantamaan. Yksi testausvaiheessa selvinnyt ongelma oli Opus-koodekin asettama rajoite äänen 20 kHz:n maksimitaajuuteen. Tämän seurauksena lopullisessa toteutuksessa taajuusalue pudotettiin 18-20 kilohertsiin. Toteutuksessa päädyttiin käyttämään alipäästösuodinta asiakasohjelmassa, etteivät aikasignaalin taajuudet varmasti ole kuultavissa.

Useissa eri testitapauksissa havaittiin, että etenkin ensimmäisen merkin taajuus helposti havaitaan väärin. Tämän voi selittää toteutuksen ominaisuus odottaa korkeiden taajuuksien aktivoitumista, jolloin osa signaalista jää analysoimatta. Tämän seurauksena toteutuksessa päädyttiin käyttämään ennalta määrättyä taajuutta merkkinä aikaleiman alkamisesta. Tällöin hiljaisuuden tunnistamisen sijaan tunnistetaan taajuus ja alkuperäisen äänen häiriö vaikuttaa vähemmän. Häiriön vähentämiseksi lopullisessa toteutuksessa päätettiin käyttää taajuuksien huippuarvojen löytämisessä paremmin virheitä sietävää laskentatapaa.

Prototyyppitoteutus aiheutti satunnaisesti äänen puuroutumista. Toteutuksen analysoinnin suorituskyky vaihteli, jolloin äänipuskuriin ei saatu tarpeeksi nopeasti toistettavaa ääntä. Koska toteutuksen kohdeselaimet tukevat myös WebAssembly-rajapintaa, päädyttiin analysointi toteuttamaan sitä hyödyntäen. Tällä saavutetaan huomattavasti parempi suorituskyky. WebAssemblyn tukemille ohjelmointikielille löytyy myös huomattavasti enemmän valmiita avoimen lähdekoodin analysointitoteutuksia, joita toteutuksessa voidaan hyödyntää.

Koska testit suoritettiin vain Google Chrome -selaimella, lopullisen toteutuksen ensimmäinen tavoite oli testata polyfill-kirjaston toimivuus. Polyfill-kirjasto ei toteuta Web Audio APIa täydellisesti ja esimerkiksi kehyksiä kopioidaan kerrallaan 256, eikä 128.

5.2 Toteutuksen arviointi ja kehitysideat

Toteutuksen tavoitteena oli mahdollistaa protokollariippumaton usean lähteen median synkronointijärjestelmä, joka on helppo ottaa käyttöön olemassa oleviin videon suoratoistoratkaisuihin. Protokollariippumattomuudella tarkoitetaan riippumattomuutta käytettävistä koodekeista tai suoratoistoprotokollista. Koska ainoa yhdistävä tekijä suoratoistoprotokollilla on video ja ääni, täydellinen protokollariippumattomuus on mahdotonta to-

teuttaa. Ääni ja video prosessoidaan koodauksessa. Toteutuksen toimivuus riippuu siitä, ettei tässä prosessissa poisteta ääneen lisättyä ajoitusdataa eli koodausprosessi suoritetaan sellaisilla parametreilla, jotta ajoitusdata saadaan luettua. Toteutus kuitenkin evaluoitiin toimivaksi yleisillä käytössä olevilla koodekeilla.

Toteutuksen vaatimuksena oli toimia yleisimmillä moderneilla verkkoselaimilla. Tämä vaatimus saatiin täytettyä polyfill-kirjaston avulla. Selaimet toteuttavat jatkuvasti lisää HTML5-standardin määrittämiä ominaisuuksia ja tämän myötä komponentin toiminta tehostuu tulevaisuudessa.

Toteutuksen suorituskykyvaatimukseen vastattiin toteuttamalla komponentti hyödyntäen HTML5-standardin tarjoamaa WebAssembly-rajapintaa. WebAssemblyä hyödyntämällä toteutus toimii lähes vastaavasti kuin natiivi äänenprosessointitoteutus. Aikaleimoja sijoitettiin sekunnin välein toistettavaan videoon. Aikaleiman prosessoimiseen meni muutamia millisekunteja, joten kahden sekunnin tavoitteeseen päästiin selvästi.

Tavoitteet saatiin toteutettua ja toteutettu järjestelmä vastaa vaatimuksia. Toteutus kuitenkin asettaa joitakin esiehtoja. Alkuperäisessä äänisignaalissa ei saa olla liikaa häiriötä käytetyillä taajuuksilla. Alkuperäinen äänisignaali on suodatettava alipäästösuotimella, jotta signaalista saadaan 100 prosentin varmuudella taajuus tunnistettua. Videotoistimen on estettävä videon täydellinen mykistäminen video-rajapinnan kautta. Sen sijaan mykistämiseen on käytettävä MediaSync-rajapinnan metodeja. On myös huomioitava, että kaikkien datalähteiden kellon tulee olla synkronoituja, jotta datan synkronointi suoritetaan oikein.

Toteutus heikentää alkuperäisen äänisignaalin laatua ja vaatii suurempaa tiedonsiirtonopeutta. Laadun heikkeneminen ja alipäästösuodatuksen vaatimus voitaisiin poistaa lisäämällä haluttu äänisignaali omana kanavanaan. Uusi kanava kasvattaisi tarvittavaa tiedonsiirtonopeutta, mutta alkuperäinen äänisignaali pysyisi käsittelemättömänä. JavaScript Web Audio API mahdollistaisi kuusi eri kanavaa. Usean kanavan ääni ei kuitenkaan ole kaikilla koodekeilla mahdollista ja on täten ristiriidassa alkuperäisen vaatimuksen kanssa.

Toteutuksen suorituskykyä ei myöskään ole luotettavasti testattu. Järjestelmälle pitäisi tehdä suorituskykytestaus eri laitteilla ja ympäristöillä, jotta voidaan varmistua järjestelmän toimivuudesta kaikissa tilanteissa. Mikäli suorituskyky ei ole riittävä, voitaisiin heksadesimaalisekvenssin sijaan käyttää binäärisekvenssiä ja FFT:tä tehokkaampaa Goertzelin algortmia. Tämä toteutusvaihtoehto olisi herkempi häiriöille, mutta voisi toimia nykyistä toteutusta tehokkaammin aiemmin mainitun monikanavaratkaisun kanssa.

6 YHTEENVETO

Suoratoistopalvelut ovat yleistyneet huimaa tahtia. Tämän seurauksena alalle on syntynyt useita kilpailevia teknologioita, joista osa on menestynyt toisia huomattavasti paremmin. Suoratoiston kasvaminen on aiheuttanut tarpeita erilaisille videoon synkronoiduille järjestelmille. Synkronoidut järjestelmät mahdollistavat katselijalle laaja-alaisemman katselukokemuksen ja interaktion suoratoistolähetysten kanssa.

Verkkoselainkehitys ei ole pysynyt nopeasti muuttuvien suoratoistoteknologioiden perässä. Tästä syystä viime vuosina on määritetty tiuhaan tahtiin erilaisia standardeja teknologiaeron tasoittamiseksi. Standardit eivät kuitenkaan vieläkään ota kantaa suoratoistoprotokollien toteuttamiseen. Tästä syystä eri selainvalmistajien verkkoselaimet tukevat vaihtelevasti eri suoratoistoteknologioita. Suoratoistoteknologiat myös useasti määrittävät, mitä koodekkeja videon suoratoistoon voidaan käyttää. Koodekeissa on huomattavia eroja niin suorituskyvyssä, maksullisuudessa kuin avoimuudessa.

Kaikkia selainympäristöjä tukevan synkronisten järjestelmien toteuttaminen on työlästä ja sitä kautta kallista. Useat kaupalliset ratkaisut tarjoavat tukensa vain osalle suoratoistoteknologioista. Tämä vaikeuttaa uusien ja mahdollisesti parempien suoratoistoprotokollien ja koodekkien käyttöönottoa.

Tämän työn tarkoituksena oli suunnitella järjestelmä, joka käyttää kaikille suoratoisto- ja koodausteknologioille yhteistä elementtiä eli ääntä tai videota synkronoinnin toteuttamiseksi. Toteuksen vaatimuksina oli selaintuki yleisimmille selaimille ja helppo liitettävyys olemassa oleville suoratoistoratkaisuille. Työssä päädyttiin käyttämään ääntä videon sijasta. Ääneen voidaan käyttää signaalinkäsittelyn menetelmiä tarvittavan ajoitusdatan lisäämiseksi. Ääni voidaan myös piilottaa käyttäen ihmiskorvan ja digitaalisen äänen ominaisuuksia. Videodata on herkempi muutoksille koodausprosessissa ja datan piilottaminen täysin näkymättömiin on haastavaa.

Järjestelmän toimivuus evaluoitiin testaamalla toteutuksen periaatteet ennen työn lopullista toteuttamista. Prototyypin evaluoinnilla varmistettiin onnistuneesti järjestelmän toimivuus ja optimaaliset parametrit, jolla lopullinen toteutus voidaan toteuttaa. Evaluoinnissa ilmeni asioita, joita ei oltu osattu ottaa suunnitteluvaiheessa huomioon. Nämä kuitenkin onnistuttiin korjaamaan lopulliseen toteutukseen.

Lopullinen toteutus hyödyntää HTML5-standardin määrittelemiä rajapintoja, jotka on toteutettu yleisimmillä selaimilla. Nämä rajapinnat mahdollistavat äänen analysoinnin dekoodausprosessin jälkeen. Videon ääneen on piilotettu aikaleimoja taajuusmodulaation

avulla. Rajapintojen avulla äänisignaali muutetaan tehokkaasti taajuustasoon ja siitä saadaan analysoitua piilotetut aikaleimat. Aikaleimojen avulla päivitetään JavaScript-komponentin ajastin. Ajastimen avulla usea datalähde saadaan synkronoitua sulavasti ja tarkasti videon toistoon. Toteutus asettaa tiettyjä esiehtoja toiminnallisuuden takaamiseksi. Alkuperäinen äänisignaali on suodatettava käyttäen alipäästösuodinta, jotta aikaleimat sisältävä data saadaan luettua. Myös koodausprosessissa on huomioitava tarvittava suurempi tiedonsiirtonopeus, jotta aikaleimat sisältävä äänisignaali ei häviä häviöllisessä prosessissa.

LÄHDELUETTELO

[1] *AAC Frequently Asked Questions*. URL: http://www.via-corp.com/licensing/aac/faq.html (viitattu 25. 04. 2019).

[2] *About W3C*. URL: https://www.w3.org/Consortium/ (viitattu 25. 04. 2019).

[3] I. M. Arntzen, N. T. Borch ja F. Daoust. Media Synchronization on the Web (2018). URL: https://www.w3.org/community/webtiming/files/2018/05/arntzen_mediasync_web_author_edition.pdf.

[4] D. Chang, X. Zhang, Q. Liu, G. Gao ja Y. Wu. Location Based Robust Audio Watermarking Algorithm for Social TV System (2012). ID: - 10.1007/978-3-642-34778-8_68, 726–728.

[5] *Chirp, Solutions*. URL: https://chirp.io/solutions/ (viitattu 25. 04. 2019).

[6] J. Cutnell ja K. Johnson. Physics (4th Edition, Volume 1). John Wiley & Sons Inc, 1997, 466.

[7] A. Deveria. *Can I use WebM?* URL: https://caniuse.com/#search=WebM (viitattu 25. 04. 2019).

[8] *HLS Authoring Specification for Apple Devices*. URL: https://developer.apple.com/documentation/http_live_streaming/hls_authoring_specification_for_apple_devices (viitattu 25. 04. 2019).

[9] *HTML - Living Standard - Media elements*. URL: https://html.spec.whatwg.org/multipage/media.html#media-elements (viitattu 25. 04. 2019).

[10] *HTML5 - Specification*. URL: https://www.w3.org/TR/html52/semantics-embedded-content.html#the-video-element (viitattu 27. 04. 2019).

[11] *HTML5 Standard - Timers*. URL: https://www.w3.org/TR/2011/WD-html5-20110525/timers.html#windowtimers (viitattu 08. 05. 2019).

[12] *ID3 - Introduction*. URL: http://id3.org/Introduction (viitattu 03. 05. 2019).

[13] *IE11 - Product Lifecycle information*. URL: https://support.microsoft.com/en-us/lifecycle/search?alpha=Internet%5C%20Explorer%5C%2011 (viitattu 25. 04. 2019).

[14] R. J. M. II. *Introduction to Shannon Sampling and Interpolation Theory*. New York: Springer-Verlag, 1991, 324.

[15] *ISO/IEC 23009-1:2014*. URL: https://www.iso.org/standard/65274.html (viitattu 25. 04. 2019).

[16] *ITU-T H.264 (V12) (04/2017), Advanced video coding for generic audiovisual services*. URL: http://handle.itu.int/11.1002/1000/13189 (viitattu 25. 04. 2019).

[17] *ITU-T H.265 (V5) (02/2018) - High efficiency video coding*. URL: http://handle.itu.int/11.1002/1000/13433 (viitattu 25. 04. 2019).

[18] S. Jobs. *Thoughts on Flash*. URL: https://www.apple.com/hotnews/thoughts-on-flash/ (viitattu 04. 04. 2019).

[19] *Low Latency Streaming*. URL: https://www.wowza.com/low-latency (viitattu 25. 04. 2019).

[20] *Matroska - Specification Notes*. URL: https://matroska.org/technical/specs/notes.html (viitattu 27. 04. 2019).

[21] G. Maxwell. *64kbit/sec stereo multiformat listening test*. URL: https://people.xiph.org/~greg/opus/ha2011/ (viitattu 25. 04. 2019).

[22] *MPEG-DASH - Event Schemes*. URL: https://dashif.org/identifiers/event_schemes/ (viitattu 03. 05. 2019).

[23] *Opus - Licensing details*. URL: https://www.opus-codec.org/license/ (viitattu 25. 04. 2019).

[24] K. C. Pohlmann. *Principles of digital audio*. eng. 4th ed. New York (NY): MacGraw-Hill, 2000, 736 sivua. ISBN: 0-07-134819-0 nidottu.

[25] *Revolutionizing Intellectual Property Rights Management*. URL: https://www.mpegla.com/about/ (viitattu 25. 04. 2019).

[26] D. Sundararajan. *The discrete fourier transform: theory, algorithms and applications*. Singapore; River Edge, NJ: World Scientific, 2001. ISBN: 9812810293.

[27] I. R. Titze. Principles of Voice Production. Prentice Hall, 1994, 354.

[28] *Video Developer Report 2017*. Tekninen raportti. Bitmovin, 2017. URL: https://bitmovin.com/whitepapers/Bitmovin-Developer-Survey.pdf (viitattu 20. 04. 2019).

[29] *Video Developer Report 2018*. Tekninen raportti. Bitmovin, 2018. URL: https://go.bitmovin.com/hubfs/Bitmovin-Video-Developer-Report-2018.pdf (viitattu 20. 04. 2019).

[30] *Web Audio API - W3C Candidate Recommendation, 18 September 2018*. URL: https://www.w3.org/TR/webaudio (viitattu 25. 04. 2019).

[31] *WebAssembly - Roadmap*. URL: https://webassembly.org/roadmap/ (viitattu 03. 05. 2019).

[32] *WebAssembly - Specification*. URL: https://www.w3.org/TR/wasm-core-1/ (viitattu 01. 05. 2019).

[33] *WebM - About*. URL: https://www.webmproject.org/about/faq/ (viitattu 27. 04. 2019).

[34] *Wowza - Timed metadata*. URL: https://www.wowza.com/docs/developers-guide-to-using-timed-metadata-in-wowza-workflows (viitattu 03. 05. 2019).