Low Power IoT encoding -klusteri

Ammattikorkeakoulututkinnon opinnäytetyö

Riihimäki, Tieto- ja viestintätekniikka

Syksy 2019

Petri Pakkanen

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tieto- ja viestintätekniikka
Riihimäki

Tekijä Petri Pakkanen **Vuosi** 2019

Työn nimi Low Power IoT encoding -klusteri

Työn ohjaaja Teemu Järvenpää

TIIVISTELMÄ

Tämän opinnäytetyön tarkoituksena oli luoda raskasta laskentaa suorittava hajautettu järjestelmä, joka kykenee suorittamaan paljon laskentatehoa vaativaa videoiden transkoodausta vähävirtaisilla ja kustannustehokkailla komponenteilla. Työssä toteutetun järjestelmän pääpirteinä ovat sen dynaamisuus ja skaalattavuus.

Teoriaosuudessa käsitellään, minkä takia videoiden muunnostyö tekee siitä laskennallisesti raskasta ja kuinka videoita pakataan tehokkaasti H.264-enkooderilla. Lisäksi käydään läpi virtualisoinnin eri tyyppejä sekä yksi moderni tapa toteuttaa hajautettu järjestelmä konttiteknologian avulla.

Opinnäytetyön tuloksena syntyi moderneilla työkaluilla toteutettu hajautettu järjestelmä, joka kykenee muuntamaan suositulla H.264-koodekilla pakattuja videotiedostoja nopeammin kuin yksittäinen tyypillinen videotyöasema.

Avainsanat Hajautetut järjestelmät, Videonpakkaus, Virtualisointi

Sivut 33 sivua, joista liitteitä 7 sivua

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Information and Communication Technology
Riihimäki

Author Petri Pakkanen **Year** 2019

Subject Low Power IoT encoding cluster

Supervisors Teemu Järvenpää

ABSTRACT

The aim of this thesis project was to create a system which would be capable of transcoding a video with low power in a cost-effective manner. Transcoding is a computationally heavy task for workstations and the aim here was be able accomplish the task faster than with a moderate video-editing computer.

The theory part in the thesis is an introduction to video coding and compression and discusses to see why transcoding is computationally heavy. There is also a brief introduction to virtualization and containerization technologies.

The outcome of this thesis project was a distributed system created with modern software tools and methods, which can transcode popular H.264 encoded videos faster and with less energy than a typical workstation computer.

Keywords Distributed computing, Video Compression, Virtualization

Pages 33 pages including appendices 7 pages

SISÄLLYS

Liitteet

1 JOHDANTO

Tämä opinnäytetyön tavoitteena on luoda järjestelmä, joka kykenee mahdollisimman kustannustehokkaasti transkoodaamaan eli muuttamaan videotiedostojen pakkaus- ja tallennusmuotoa. Tehtävä on laskennallisesti raskas ja usein tarvitaan luoda yhdestä videolähteestä useita eri versioita, koska kaikkien videontoistolaitteiden kirjo on hyvin laaja ja ne eroavat suorituskyvyltään hyvin paljon toisistaan. Nykyteknologialla on kustannustehokkaampaa lähettää toistolaitteelle videokuvaa, jonka näyttäminen vie mahdollisimman vähän energiaa eli video on valmiiksi sovitettu oikeaan resoluutioon ja formaattiin. Työssä käsitellään ainoastaan H.264-videonpakkausstandardin mukaisia videoita, koska se on tällä hetkellä maailman käytetyin standardi (Encoding.com, 2019).

Henkilökohtaisena tavoitteena oli muodostaa tarkempi käsitys, miten nykymaailmassa kyetään luomaan paljon eri laitteilla tallennettua videomateriaalia monelle erilaiselle toistolaitteelle. Toisena motivaattorina oli tehdä sama prosessi mahdollisimman kustannustehokkailla komponenteilla ja vielä niin, että järjestelmään pystyy tarvittaessa lisäämään laskentatehoa mahdollisimman kivuttomasti ilman uudelleenohjelmointia tai järjestelmän uudelleenkonfigurointia. Oma kiinnostus hajautettuun laskentaan toi ajatuksen luoda klusteri Raspberry Pi-minitietokoneista, koska olen käyttänyt niitä useassa aiemmassa projektissa.

Videonpakkauksella tarkoitetaan lyhyesti videokuvan tiivistämistä. Tiivistäminen tapahtuu etsimällä tiedoston bittivirrasta redundanssia eli toistoa. Kun tiedossa eli tässä tapauksessa peräkkäisissä kuvissa esiintyy redundanssia, pyritään löytämään se ja poistamaan se mahdollisimman tarkkaan niin, että tietosisältö säilyy ennallaan. Tämä on laskennallisesti raskas tehtävä koska kuvia käsitellään jopa 1/8 pikselien tarkkuudella. (Chen, 2011, s. 742)

Pakkausmenetelmät jaetaan kahteen eri perustyyppiin, häviölliseen ja häviöttömään pakkaukseen. Pakkaus perustuu videokuvassa esiintyvän samankaltaisuuden eli redundanssin poistamiseen. Videokuvassa esiintyy kolmea erityyppistä redundanssia: paikallista-, ajallista ja väriredundanssia.

Häviöttömässä pakkauksessa ei kadoteta tietosisältöä vaan pyritään identifioimaan ja poistamaan sisältöä, joka toistuu videossa useasti. Häviöttömään pakkaukseen pyrkivät algoritmit etsivät tiedosta mahdollisimman tehokkaasti redundanssia ja pyrkivät esittämään saman tiedon vähemmällä

bittimäärällä. Häviötön pakkaus on aina palautettavissa alkuperäiseen muotoonsa.

Häviöllisessä pakkauksessa tietosisältö muuttuu pakkauksen aikana. Jotta digitaalisen kuvan sisältö säilyisi katsomiskelpoisena on pakkauksessa hävitettävä tieto oltava sellaista, jota ei pystytä ihmissilmällä havaitsemaan. Häviöllisellä pakkausmenetelmällä pakattua materiaalia ei voida palauttaa takaisin alkuperäiseen muotoonsa.

Paikallisella eli spatiaalisella redundanssilla tarkoitetaan sitä, että yhdessä kuvakehyksessä pikseleissä esiintyy paljon toistoa. Eli samanlaisia pikseleitä on paljon ja usein vierekkäiset pikselit ovat riippuvaisia toisistaan. Kuvassa 1 on esimerkki kuvakehyksestä, jossa esiintyy runsaasti paikallista toistoa kehyksen sisällä.

Kuva 1. Runsaasti spatiaalista redundanssia. (Carranza, 2010)

Ajallisella eli temporaalisella redundanssilla tarkoitetaan sitä, että videokuvan peräkkäiset kuvat sisältävät paljon samankaltaista informaatioita. Kun kuva vaihtuu sekvenssissä toiseen, niin kuvassa tapahtuu usein vain vähän liikettä verrattuna koko kuvan kokoon. Tätä ylimääräistä toistoa pyritään minimoimaan ja välittää informaatioita ainoastaan pikseleistä, jotka muuttuvat sekvenssin edetessä. Sivulla 3 kuvassa 2 on kaksi peräkkäistä kuvakehystä, jossa esiintyy runsaasti niiden välistä toistoa.

Kuva 2. Runsaasti temporaalista redundanssia kuvakehyksissä. (O'Reilly Media, n.d.)

Digitaalisissa videoissa esiintyvien kuvien pienin yksikkö on pikseli, joka koostuu kolmesta värikomponentista R (punainen), G (vihreä) ja B (sininen). Kuvien pikseleistä otetaan riippuen kuvasta joko 8-, 10- tai 12 -bittisiä näytteitä jokaisesta erillisestä värikanavasta.

Videon pakkaamisen kannalta on kuitenkin tärkeämpää saada tietoa tiivistettyä tehokkaasti niin, että kuvasta saadaan poistettua informaatiota, jota ihminen ei pysty herkästi havaitsemaan. Ihmissilmä havaitsee herkemmin luminanssimuutokset kuin krominanssimuutokset kuvassa. Tämän tiedon häviäminen pyritään minimoimaan digitaalisessa videon käsittelyssä usein siten, että tehdään muunnos RGB-väriavaruudesta YCbCr-väriavaruuteen, joka koostuu myös kolmesta eri komponentista. Y eli luma tarkoittaa kirkkautta, Cb eli sinikroma ja Cr eli punakroma. (Hanhijärvi, 2009, s. 7)

2 **H.264**

H.264 eli AVC (Advanced Video Coding) on 2003 nimensä saanut standardi. Tämä nimi on ITU:n (International Telecommunication Union) tuntema, mutta ISO (International Organization for Standardization) ja IEC (International Electrotechnical Commission) tuntevat sen nimellä MPEG4-AVC. Standardin käyttö edellyttää lisenssimaksua, mutta esimerkiksi Cisco Systems on julkaissut OpenH264-kirjaston avoimena lähdekoodina.

AVC-standardia on paranneltu usealta osin sen määrittelyn jälkeen, mutta se on edelleen erittäin isossa roolissa nykyajan videonpakkauksessa. Laaja käytettävyys johtuu siitä, että se on alun perin suunniteltu toimimaan mahdollisimman monessa eri prosessoriarkkitehtuurissa ja ympäristössä.

H.264 onkin suunniteltu perusrakenteeltaan kaksikerroksiseksi, johon kuuluvat videokoodauskerroin (VLC, Video Coding Layer), johon kuuluu varsinainen videon pakkaaminen sekä verkkokerrokseen (NAL, Network Abstraction Layer) johon sisältyy mekanismit, joita tarvitaan datan lähetys- ja vastaanottoliikennöinnissä. H.264 seuraaja on H.265 eli HEVC (High Efficiency Video Coding), jonka avulla pystytään pakkaamaan dataa tehokkaammin ja näin ollen sisältämään enemmän videodataa samaan bittinopeuteen kuin H.264:lla. (Adda & Benyamina, 2017)

2.1.1 Videokoodauskerroin

Videon pakkaaminen ei tapahdu millään yksittäisellä algoritmilla, vaan se on sarja jo aiemmin kehitettyjä tekniikoita, joita on paranneltu ja ne ovat yhdessä luoneet uuden standardin. H.264:ta voidaankin kutsua hybridikooderiksi, jossa käytetään hyväksi aikaisemmista standardeista tuttua lohkopohjaista lähestymistapaa. Hybridin siitä tekee se, että se on lohkopohjaisesti muunnosalueittain liikekompensoitu, joka tarkoittaa käytännössä sitä, että spatiaalinen eli paikallinen redundanssi pyritään hävittämään muunnosalueen koodauksella ja temporaalinen eli ajallinen redundanssi kehittyneillä tavoilla valita liikevektoreita lohkojen välillä. (Hanhijärvi, 2009, s. 12)

2.1.2 Makrolohkot

Makrolohkot luodaan ottamalla videossa sisältämien kehyskuvien pikseleistä 8-, 10- tai 12- bittisiä näytteitä kustakin värikanavasta. Värikanavana on usein RGB (Red, Green, Blue), joka muunnetaan toiseen värikanavaan, jossa on yksi luma- ja kaksi kroma-elementtiä. Esimerkiksi YC_oC_g sisältää värikanavat Y, joka tarkoittaa kirkkautta (luma), sekä oranssi (C_o) ja viher (C_g) kromat eli värisävyt. Muunnos RGB ja YC_oC_g –väriavaruuksien välillä voidaan suorittaa algoritmilla:

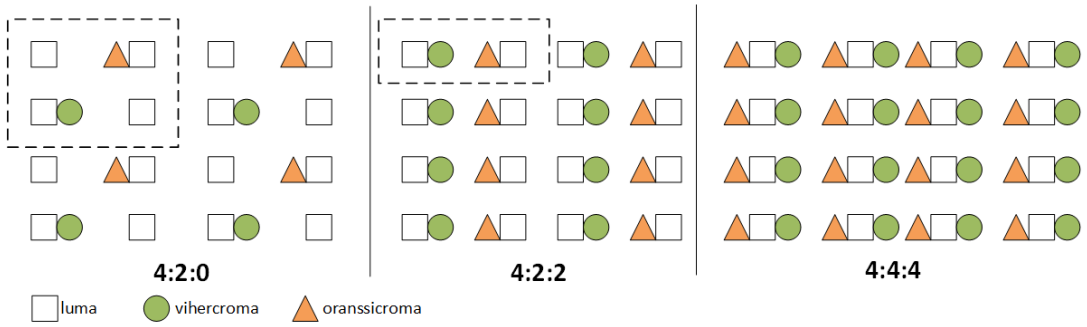
$$
\begin{aligned}
C_o &= R - B \\
t &= B + (C_o >> 1) \\
C_g &= G - t \\
Y &= t + (C_g >> 1) \qquad (1)
\end{aligned}
$$

YC_oC_g väriavaruuden etu verrattuna YC_bC_r –avaruuteen on se, että muunnos on kahden bitin lisäyksellä RGB:n arvoon häviötön. Eli jos on 24-bittinen RGB kuva, jossa jokaisessa kanavassa on 8-bittiä tietoa väristä, voidaan muuntaa 26-bittiseen YC_oC_g 4:4:4 kuvaksi ja takaisin häviöttömänä. Toisena etuna on, että muunnos on laskennallisesti kevyempi oranssi-viher -kromaan kuin sini-puna –kromaan. (Malvar, Sullivan & Srinivasan, 2008)

Näistä värikomponenteista otetaan näytteitä erilaisilla tavoilla. Tyypillisimmät tavat ovat:

- 4:2:0, jokaista neljää lumanäytettä (2x2 matriisissa) kohden otetaan yksi oranssi- ja viherkromanäyte.
- 4:2:2, jokaista kahta lumanäytettä (2x1 matriisissa) kohden otetaan yksi oranssi- ja viherkromanäyte.
- 4:4:4, kroma- ja lumanäytteitä otetaan yhtä monta.

Kuvassa 3 on hahmoteltu yllä mainitut näytteenottotavat.



Kuva 3. Yleisimpiä näytteenottotapoja.

Jokainen näytteenottopiste edustaa kuvan pikseliä, joka on sama piste kuin lumanäyte, koska jokaisessa mainituissa tavoista kaikista pikseleistä otetaan lumanäyte. Eli kun otetaan esimerkiksi 4:2:0 näytteenotolla 16x16 kokoiseen makrolohkoon lumanäytteet, niin otetaan 8x8 kokoiset kromanäytteet molemmista värikomponenteista. Syy siihen miksi luma on oleellisempi tieto ottaa talteen kuin värit, on että ihmissilmä havaitsee tarkemmin kirkkauden (musta-valko) muutokset kuin värimuutokset kuvassa (Winkler, Lambrecht & Kunt 2001, s. 209). Otetut näytteet käsitellään siis makrolohkoina ja joka sisältää yhden 16x16 ja kaksi 8x8 kokoista taulukkoa. (Richardson, 2010, s. 119)

2.1.3 Viipaleet

Makrolohkot voidaan ryhmitellä hyvin joustavasti viipaleisiin. Viipaleet ovat annetusta kuvasta alueita, joita voidaan käsitellä toisistaan riippumattomasti. Tästä on suuri hyöty siinä mielessä, että enemmän huomiota vaativat kohdat, jotka sisältävät enemmän yksityiskohtia tai liikettä voidaan jakaa pienempiin viipaleisiin ja nämä viipaleet voitaisiin jakaa toisen prosessorin käsiteltäväksi. Viipaleet on jaoteltu viiteen eri tyyppiin viipaletyyppiin: I-, P-, B-, SP- ja SI- viipaleet.

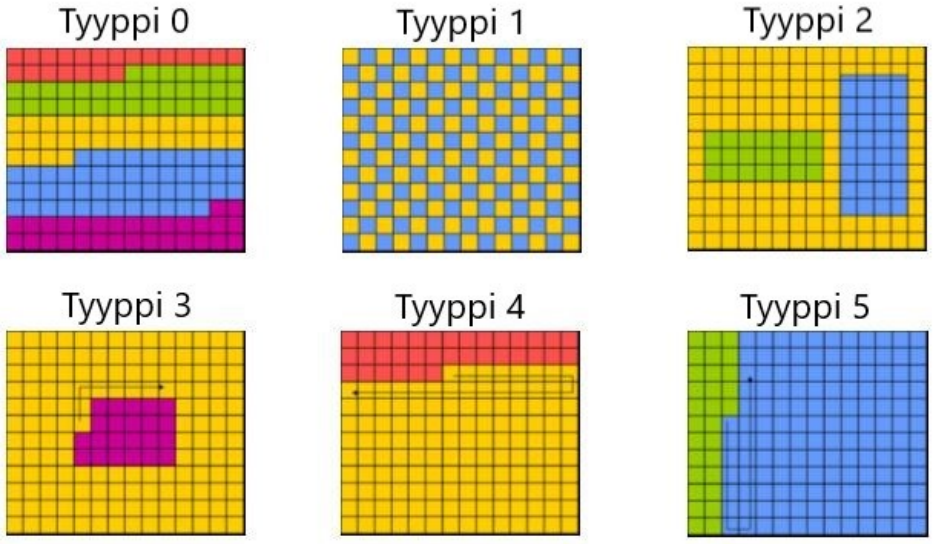
I-viipaleiden sisältämät kaikki makrolohkot ovat intrakoodattuja eli koodauksessa etsitään ainoastaan spatiaalista redundanssia. Viipale on täysin itsenäinen ja sitä voidaan käyttää referenssinä muille viipaleille.
P-viipaleet ovat taas joukko makrolohkoja, jotka ovat interkoodattuja eli tämän viipaleen lohkoille etsitään liikevektoreita referenssiviipaleesta. Etsintä suoritetaan kuitenkin niin, että jokaiselle lohkolle muodostuu korkeintaan yksi liikevektori. P-tyyppinen viipale voi toimia myös referenssinä muille viipaleille.

B-viipaleiden lohkoille etsitään myös inter-ennusteita liikekompensoidulla ennusteella. Erona P-viipaleen lohkoihin on se, että tässä viipaleessa lohkolla voi olla kaksi liikevektoria, eikä se voi toimia referenssiviipaleena.

Edellä mainittujen lisäksi H.264:ssa tunnetaan SP-viipale eli ”Switching P” ja SI-viipale eli ”Switching I”, jota ei käsitellä tässä työssä. (Richardson, 2010, s. 117)

2.1.4 Joustava makrolohkojen järjestäminen

Joustava makrolohkojen järjestäminen (flexible macroblock ordering, FMO) on H.264:ssa käytetty tekniikka, joka mahdollistaa viipaleiden jaon useisiin ryhmiin. Standardissa määritellään kuusi erilaista tapaa järjestää viipaleet ja seitsemäs tapa on käyttäjän itse määriteltävissä. Kuvassa 4 on esitelty määritellyt tyypit viipaloinneille.



Kuva 4. Makrolohkojen jako viipaleisiin. Kukin väri edustaa eri viipaletta.

Tyyppi 0 on samanlaisena toistuva koko kuvan täyttävä sarja viipaleita alkaen vasemmalta ylhäältä edetessä oikealle alas. Dekooderille ei tarvitse siis toimittaa muuta tietoa kuin viipaleiden pituudet. I-tyypin referenssikehykset ovat tätä tyyppiä.

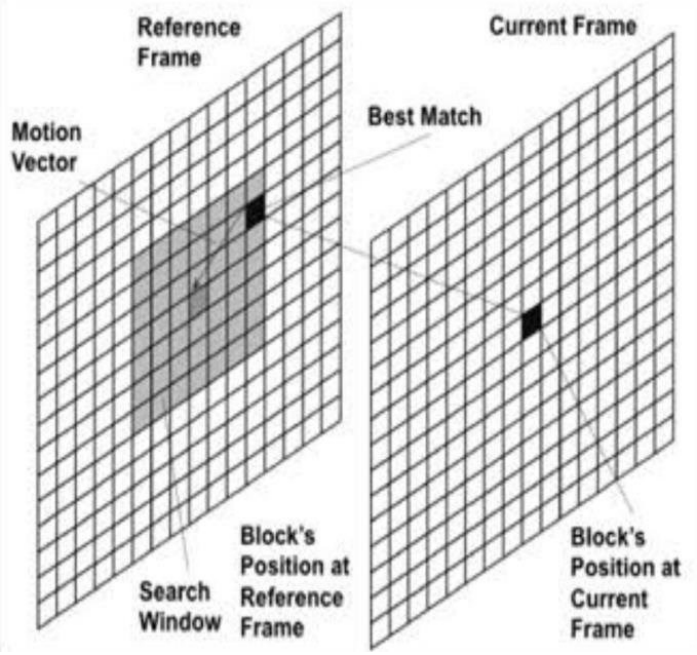
Tyyppi 1 on jaettu kahteen viipaleryhmään niin että joka toinen makrolohko on eriviipaleessa. Tässä on se etu, että esimerkiksi videokonferenssissa, jossa näkyy usein vain yhden henkilön naama tai ylävartalo, voidaan virheelliset lohkot korvata kokonaan viereisellä lohkolla, jolloin näkyvä virhe jää minimaaliseksi.

Tyyppi 2 on tarkoitettu tilanteisiin, jossa halutaan erottaa kuvasta tietyt mielenkiintoiset kohdat, jotka sisältävät yksityiskohtia tai liikettä. Taustalle jäävä osa pystytään täten pakkaamaan tiiviimmin. Dekooderi tarvitsee ainoastaan etualalla olevien viipaleiden vasemman ylänurkan ja oikean alanurkan makrolohkojen osoitteet pystyäkseen dekoodaamaan viipaleet. (Mazataud, 2009)

Tyypit 3, 4 ja 5 käyttävät dynaamisia viipaleryhmiä, jotka muuttuvat kooltaan koko videon ajan ennalta määrätyn mallin mukaan.
Tyyppi 6 on käytännössä täysin käyttäjän määriteltävissä, joten sitä ei ole esitetty sivulla 6 kuvassa 4. (Richardson, 2010, s. 238)

2.1.5 Liikkeen estimointi

Liikevektoreiden muodostaminen eli kuvien välisen liikkeen estimointi on laskennallisesti ylivoimaisesti työläin vaihe videon pakkaamisessa, koska liikevektoreita etsitään jokaisesta pikselin positiosta. Tässä vaiheessa haetaan esimerkiksi timantti tai kuusikulmio algoritmilla makrolohkojen välisiä yhteyksiä referenssiviipaleiden välillä ja tallennetaan niiden välinen liikevektoriksi kutsuttu yhteys. Hakualgoritmin tehokkuus riippuu paljolti siitä, millaista liikettä videossa esiintyy. Sivulla 8 kuvassa 5 on havainnoitu, kuinka lohkolle löydetään vastaavuus referenssikuvasta ja muodostetaan liikevektori Block Matching-algoritmilla lohkon vasemmasta yläkulmasta kohdepaikan vasempaan yläkulmaan. (Mahaboob, 2018)



Kuva 5. Liikevektorin muodostaminen referenssikuvasta (Mahaboob, 2018).

2.2 **Videotyökalut**

Työssä pääasiallisin työkalu videotiedostojen käsittelyyn on ffmpeg. Se on vapaan lähdekoodin komentorivityökalu, johon on koottu yhteen ohjelmaan useita vapaita ohjelmistoja. Sen vahvuutena moniin muihin työkaluihin ilmenee sen laajasta tuesta useille arkkitehtuureilla ja käyttöjärjestelmille. Työssä käytettävä työkalu täytyi kuitenkin olla käännettävissä ARM-prosessoriarkkitehtuurin omaavalle linuxille. Lisäksi se sisältää libavcodec-kirjaston joka on käytettävissä LGPL v2.1+ lisenssillä. FFmpeg:llä pystyy vaivattomasti muun muassa dekoodata, enkoodata, transkoodata, suodattaa ja striimata digitaalista ääntä ja videota (FFmpeg, 2019).

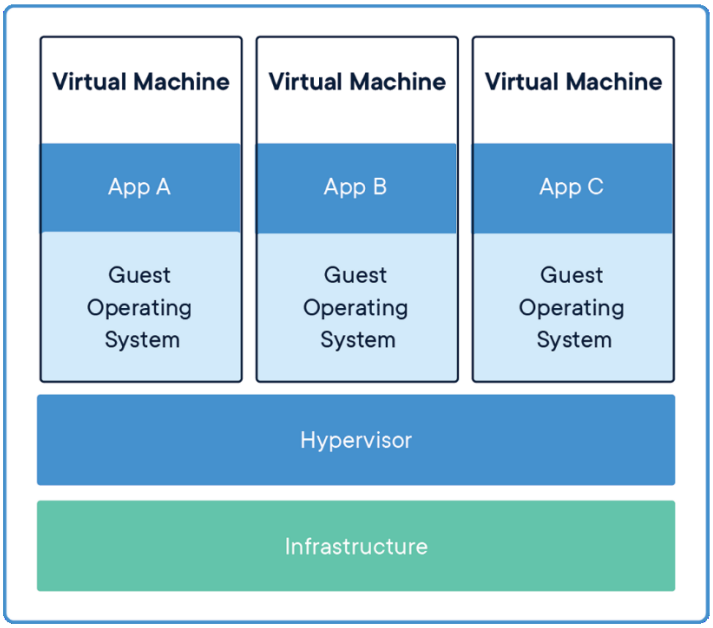
3 VIRTUALISOINTI

Virtualisointi tietotekniikassa tarkoittaa lyhyesti rakennetta, jossa jaetaan tietokoneen resurssit useaan ajoympäristöön eli jaetaan fyysinen laskentateho useaksi loogiseksi resurssiksi. Virtualisointi toimii myös toisinpäin eli useat fyysiset resurssit voidaan näyttää yhtenä kokonaisena loogisena resurssina.

Virtualisointi on kehitetty siksi, että prosessoreiden laskentateho saataisiin tehokkaammin ja joustavammin hyödynnettyä. Palvelinvirtualisoinnissa käytettäviä tekniikoita ovat täys-, para- ja käyttöjärjestelmätason virtualisointi.

Täysvirtualisointi tarkoittaa sitä, että virtualisoiduilla sovelluksilla on hallintasovelluksen avulla yhteys fyysisiin resursseihin käyttäen laite-emulointia. Täysvirtualisoinnissa ajettava virtuaalinen käyttöjärjestelmä ei havaitse, että sitä ajetaan virtuaalisessa ympäristössä. Täysvirtualisoinnissa voidaan ajaa eri käyttöjärjestelmiä päällekkäin.

Paravirtualisointi eroaa täysvirtualisoinnista siten, että siinä ajettavan käyttöjärjestelmän ytimen eli kernelin oleellisimmat kutsut on korvattava hyperkutsuilla, jotka kommunikoivat hypervisorin kanssa. Kuvassa 6 on havainnoitu hypervisorin rajapinta alla olevan infrastruktuurin ja sen päällä pyörivien virtuaalikoneiden välillä.



Kuva 6. Virtuaalikoneet sijoittuvat hypervisorin päälle (Docker Inc, 2019a).

Käyttöjärjestelmätason virtualisoinnissa taasen isäntäkone jakaa oman ytimensä siinä ajettavien vieraskäyttöjärjestelmien kanssa. Tämä tekniikka siis vaatii, että virtualisoitu käyttöjärjestelmä on sama kuin isäntäkoneella. (Graziano, 2011)

3.1 **Säiliönti**

Säiliönti eli edellä mainittu käyttöjärjestelmätason virtualisointi on kevyempi tapa toteuttaa virtualisointi, koska siihen ei tarvita erillistä virtuaalikonemonitoria eli hypervisoria. Kaikki säiliöt, jotka sisältävät usein vain sovelluksen ja sen tarvittavat kirjastot, voidaan ajaa alla olevan saman käyttöjärjestelmän päällä.

Alla oleva käyttöjärjestelmä voi olla joko fyysisen raudan päällä tai se voidaan ajaa myös virtuaaliympäristössä. Säiliöinnissä on useita muitakin hyötyjä kuten sovellusten liikutettavuus ja skaalattavuus. Säiliöt voidaan siirtää järjestelmästä toiseen, kunhan säiliöstä löytyy vastaava levykuva, jossa on sama järjestelmä- ja prosessoriarkkitehtuuri kuin ajettavassa järjestelmässä. (Estes, 2017)

3.2 **Docker**

Docker on suosittu avoimen lähdekoodin hallintajärjestelmä säiliöille, jolla voidaan luoda, ottaa käyttöön ja ajaa sovelluksia käyttäen säiliöitä. Sivulla 11 kuvassa 7 on kuvattu kuinka docker rakentuu palvelimen ja käyttöjärjestelmän päälle. (Red Hat Inc, n.d.)

Kuva 7. Docker säiliöt havainnoitu järjestelmässä.

Säiliöt ovat itsenäisiä yksiköitä eroteltuina toisistaan ja käyttöjärjestelmästä. Tämän hyötyjä ovat muun muassa selkeämpi ohjelmistorakenne ja se sallii joustavammin määritellä kuinka sovellus käyttää järjestelmäresursseja kuten prosessorien laskentatehoa, muistia, I/O-laitteita ja verkkokapasiteettia. Rakenne selkeyttää myös tiedon ja ohjelman eriyttämistä, koska se pakottaa kehittäjät erottamaan ohjelmakoodi ja sovelluksen data toisistaan. Dockerin arkkitehtuuri koostuu palveluprosessista, asiakasohjelmasta, rekistereistä ja objekteista.

Dockerin palveluprosessi ottaa vastaan pyyntöjä Docker Engine rajapinnan kautta ja se hallinnoi objekteja kuten levykuvia, säiliöitä ja verkkoja.
Käyttäjä pääsee käyttämään rajapintaa CLI asiakasohjelmalla nimeltään *docker*.

Rekisteriin tallennetaan dockerin levykuvat. Dockerilla on oma keskitetty rekisteri *Docker Hub*, joka on kaikille vapaassa käytössä ja docker onkin vakiona konfiguroitu etsimään levykuvia julkisesta rekisteristä.

Objektit ovat erinäisiä kokonaisuuksia, joista sovellukset rakennetaan. Objektien pääluokat ovat levykuvat, säiliöt ja palvelut. Levykuva luodaan kirjoittamalla tiedosto *Dockerfile* (Liite 3) johon kirjoitetaan ohjeet, millainen levykuva halutaan luoda. Levykuvan voi luoda täysin tyhjästä antamalla ohjeet kaikkien käyttöjärjestelmään tarvittavien tiedostojen kääntämiseksi Linux from scratch -tyylisesti, mutta nopeampaa on käyttää jotakin jakelupakettia levykuvan pohjana ja lisätä siihen halutut komponentit. Kun levy-

kuva luodaan tiedostosta niin docker luo jokaisesta ohjeesta oman kerroksensa, näin ollen muutokset *Dockerfileen* ovat kevyitä, koska jälkeenpäin pystytään muuttamaan ainoastaan niitä osia joihin muutokset kohdistuvat. (Docker Inc, 2019b)

Säiliöt käynnistetään ja ajetaan levykuvasta, joka tarkoittaa sitä, että säiliö on luontinsa jälkeen muuttumaton. Säiliöstä ei tallenneta tilatietoja itsessään minnekään, vaan tilatieto on halutessaan tallennettava jonnekin muualle esimerkiksi tietokantaan. (InfoWorld, 2018)

3.3 **Säiliöiden hallinta**

Säiliöiden määrän kasvaessa syntyy myös tarve hallita niitä. Tunnetuimmat säiliöiden hallintaan suunnitellut järjestelmät (engl. Container Orchestration Engines) ovat Kubernetes ja Docker Swarm. Molemmilla järjestelmillä on omat ominaisuutensa ja tarkoituksensa.

Docker Swarm on Dockeriin sisäänrakennettu hallintajärjestelmä ja se on kevyempi ja nopeampi ottaa käyttöön kuin Kubernetes. Swarm on suosittu kehittäjien keskuudessa, jotka arvostavat nopeaa käyttöönottoa ja yksinkertaisuutta.

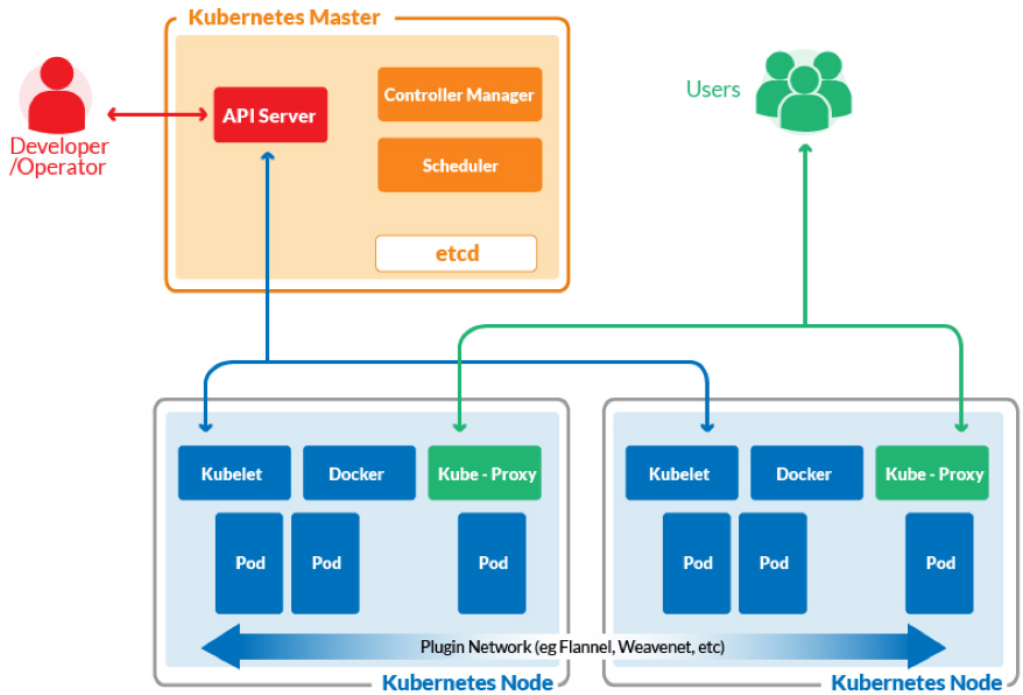
Kubernetes on taas hieman monimutkaisempi oppia ja käyttää kuin Swarm, mutta se on laajemmin käytössä tuotantoympäristöissä. Työssä tehdystä järjestelmästä haluttiin luoda mahdollisimman dynaamisesti skaalattava, joten Kubernetes oli luontevampi valinta tämän työn hallintajärjestelmäksi. (The New Stack, 2018)

3.4 **Kubernetes**

Kubernetes on Google-yhtiön kehittämä avoimen lähdekoodin järjestelmä konttipohjaisten sovellusten käyttöönottoon, skaalaukseen ja hallinnointiin. Sen edeltäjinä ovat järjestelmät Borg ja Omega. Borg on edelleen Googlella käytössä ensisijaisena säiliönhallintajärjestelmänä, vaikkakin se on näistä vanhin. Google tavoitteena oli ratkaista palvelinpuolen ongelmia palveluiden saatavuudessa ja eräajojen ajamisessa. (ACM Inc., 2016)

Kubernetes koostuu komponenteista ja ne on jaettu kolmeen eri ryhmään master- ja node-komponentit sekä lisäosat. Ryhmät ovat johdattelevasti nimetyt niin, että master-komponentit ajetaan klusterin masterilla ja node-komponentit nodeissa. Lisäosia voidaan tarvita molemmissa. (The Linux Foundation, 2018)

Kuvassa 8 on esitelty Kuberneteksen arkkitehtuuri ja miten eri osat kommunikoivat keskenään. Nodeissa esiintyvät podit sisältävät ajettavat säiliöt ja ne ovat itsessään hallittavia kokonaisuuksia.



Kuva 8. Kubernetes klusterin node arkkitehtuuri (Velez, 2019).

3.4.1 Master komponentit

Master komponentit eli Kubernetesin tarjoamat palvelut, joita ajetaan klusterin pääpalvelimella. Nämä palvelut luovat klusterille ohjaustason, johon linkittyy loppu kaikki klusterissa olevat laitteet.

Kube-apiserver on kubernetesin ohjelmointirajapinta sisäisen ja ulkoisen liittymän kubernetekseen. Sen tehtävänä on toimia kommunikointikanavana kehittäjän sekä Kuberneteksen komponenttien välillä. Lisäksi se varmistaa, että masterille tallennetut tiedot ja palvelukuvaukset käyttöönotetuista säiliöstä ovat kunnollisia.

Etcd on yhtenäinen avain-arvo tallennusjärjestelmä, johon Kubernetes tallentaa tietojaan klusterista. Palvelun toiminnan kannalta on oleellista, että kaikista Kubernetekseen liitetyistä koneista on pääsy palveluun, koska palvelun kautta jaetaan tietoja käynnissä olevista palveluista ja asetuksista.
Kube-scheduler valvoo luotuja podeja joita ei ole vielä määritelty ajamaan ja määrittelee noodin jossa podia ajetaan.

Kube-controller-manager ohjaa nimensä mukaisesti kontrollereita. Kontrollerit ovat taasen ohjelmasilmukoita, jotka valvovat kube-apiserverin kautta klusterin tilaa. Ne huomioivat halutut muutokset ja tekevät tarvittavat järjestelmämuutokset kohti haluttua tilaa. (The Linux Foundation, 2018)

3.4.2 Node komponentit

Node-komponentit eli Kubernetesin tarjoamat palvelut, joita ajetaan jokaisessa nodessa ja ne ovat vastuussa nodeissa ajettavista podeista. Kubelet on agentti, joka ajetaan jokaisessa nodessa. Sen tehtävänä on huolehtia siitä, että säiliöt toimivat podin sisällä. Se lukee tietonsa *PodSpec*:sta ja huolehtii että niissä mainitut säiliöt ovat kunnossa ja toimintakelpoisia.
Kube-proxy on jokaisen noden oma verkon välityspalvelin. Se huolehtii podien kaikesta verkon yli tapahtuvasta tiedonsiirrosta klusterin sisällä ja sen ulkopuolella.
Container Runtime on ohjelma, jonka vastuulla on käynnistää säiliöt. (The Linux Foundation, 2018)

3.4.3 Lisäosat

Lisäosat kuten DaemonSet ja Deployment tarjoavat käytettäviä ominaisuuksia klusterille. Lisäosia ajetaan koko klusterin tasolla niin ne kuuluvat koko *kube-system* nimiavaruuteen. Lisäosat eivät ole välttämättömiä klusterin toiminnalle, mutta ne helpottavat konfiguroimista ja vianetsintää huomattavasti. Niihin kuuluu muun muassa verkkotoiminnot, DNS, Web käyttöliittymä, resurssimonitorointi ja klusteritason lokien tallennus. (The Linux Foundation, 2018)

3.4.4 Konfigurointi

Kubernetesin konfigurointi onnistuu helposti luomalla YAML-tiedostoja (Yer Another Markup Language) ja ottamalla ne käyttöön kubectl -hallintaohjelman avulla komennolla *kubectl apply -f konfiguraatio.yaml*. Tiedostojen avulla on huomattavasti selkeämpää konfiguroida klusterin eri osia ja ylläpitää niitä, lisäksi hallintaohjelma tunnistaa automaattisesti jo käytössä olevat konfiguraatiot ja se ei yritä ajaa olemassa olevaa konfigurointia uudestaan, johon ei ole luotu muutoksia. (The Linux Foundation, 2018)

4 SUUNNITTELU JA TOTEUTUS

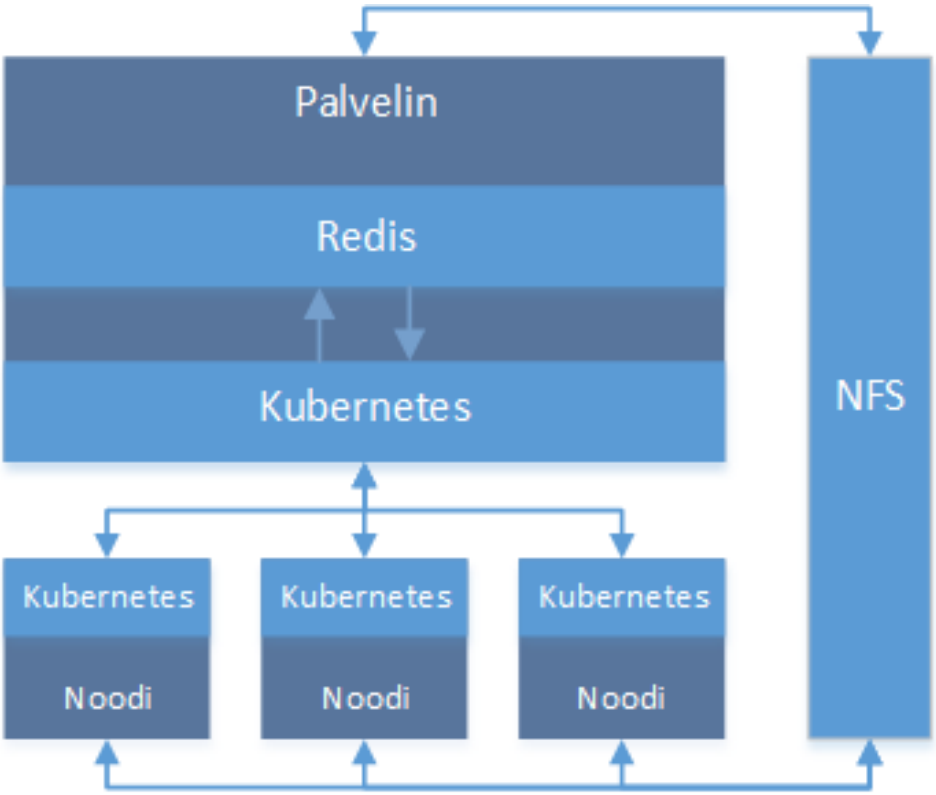
Klusterista täytyi saada toimiva ns. ProofOfConcept-tyyppinen toimiva malli, jolla pystyy osoittamaan, että pienellä määrällä Raspberry Pi –laitteilla pystyy suorittamaan laskennallisesti vaativan videoiden pakkaus- ja tallennusmuodon muutoksen nopeammin kuin yleisesti käytetty yksittäinen videotyöasema.

4.1 Ympäristö

Klusteri koostuu solmuista eli nodeista ja yhdestä palvelintietokoneesta, jonka päätehtävänä on ohjata klusterin noodeja. Toissijaisena tehtävänä sama palvelin toimii verkkolevy- ja tietokantapalvelimena.

Nodet koostuvat tässä työssä Raspberry Pi –minitietokoneista ja klusteri on suunniteltu niin, että nodeja voi lisätä tai poistaa dynaamisesti klusteriin. Node voi olla käytännössä mikä tahansa muukin ympäristö, jossa pystyy ajamaan Dockeria ja Kubernetesia.

Palvelintietokonetta ajettiin linux-virtuaalikoneella, johon asennettiin myös Docker, Kubernetes sekä Redis-tietokantaohjelmisto.
Tiedontallennusta varten samaan verkkoon liitettiin NFS-palvelin, joka toimi verkkolevynä nodeille ja palvelimelle. Sivulla 16 kuvassa 9 on kuvattu ympäristön sisäinen rakenne.

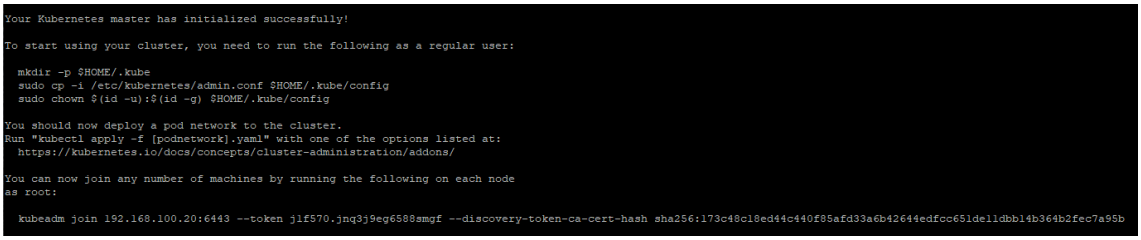


Kuva 9. Ympäristön sisäinen rakenne.

4.2 **Palvelin**

Palvelinkoneella on kaksi päätehtävää. Kubernetes masterin roolissa toimiminen ja Redis-tietokannan saavutettavuuden varmistaminen. Klusterin jokaisen minitietokoneen käyttöjärjestelmän päälle asennettiin Docker ja Kubernetes. Lisäksi hallinnoinnin helpottamiseksi luotiin ssh-avaimet testausvaiheen hallinnoinnin helpottamiseksi, koska usein nopeammaksi ratkaisuksi osoittautui resetoida koko klusteri, kuin lähteä diagnosoimaan tarkemmin virhetilanteita.

Kubernetesin alustus suoritetaan komennolla *kubeadm init* pääkäyttäjän oikeuksilla, joka luo tarvittavat sertifikaatit ja konfiguroinnit järjestelmään. Kun komento on suoritettu onnistuneesti, tulee sivun 17 kuvan 10 mukainen ilmoitus.



Kuva 10. Onnistuneesti suoritettu kubeadm alustus masterilla.

Nodejen liittämiseksi Kubernetes vaatii myös verkkotuen masterin ja nodien välille, joka toteutetaan asentamalla vaadittava lisäosa. Tässä tapauksessa verkko-lisäosaksi valittiin Weave Net, koska siinä on kubeproxyn kautta tuki Layer 3 reititykselle ja tuki salaukselle. Huonona puolena Weave Netissä on, että se toimii mesh-verkon tavoin ja jos klusteriin liitetään useita satoja nodeja niin reititykseen syntyy pullonkauloja. Tässä tapauksessa se ei kuitenkaan vielä koidu ongelmaksi. Nodet saadaan liitettyä klusteriin ohjelman tulostamalla komennolla ajamalla se jokaisessa noodissa erikseen. Kun noodit on liitetty, voidaan tarkistaa masterissa komennolla *kubectl get nodes* että kaikki noodit näkyvät palvelimelle.

4.2.1 NFS

Jotta jokainen klusterissa oleva laite pääsee käsiksi tiedostopalvelimelle, on Kubernetesiin konfiguroitava oma levytila. Tätä varten on tiedettävä tiedostopalvelimen osoite ja tiedostopolku. Liitteessä 1 on työssä käytetty konfiguraatio levypalvelimelle. Siinä konfiguroidaan ”data” -niminen loogi-nen levy, jonka kapasiteetiksi määritellään 20Gib. Nodet pääsevät tämän kautta liittämään omaan järjestelmäänsä tämän levytilan, josta ne käyvät lukemassa työstettävän tiedostonsa.

4.2.2 DaemonSet

Kubernetesin DaemonSet pitää huolen siitä, että nodessa ajetaan podin kopoita. Tätä varten luotiin daemonset.yaml -tiedosto (Liite 2), joka lataa levykuvan *menturio/clusnpi* dockerin hub-palvelusta osoitteesta https://hub.docker.com/. Levykuva on luotu palveluun käyttäjän ”*menturio*” -alle ja luomisprosessi on esitelty Dockerfile-tiedostossa (Liite 3).

Lisäksi konfiguroinnissa määritellään säiliöön liitettävä levytila polkuun */data* sekä VideoCore-laitetiedosto */dev/vchiq.* Kubernetes hoitaa levytilan jakamisen säiliölle ja sen ei tarvitse olla olemassa isäntäjärjestelmässä, mutta laitetiedosto on rajapinta Raspberry Pi:n fyysiselle prosessorille, joten se on oltava olemassa.

4.2.3 Tietokanta

Klusterin suorittamat työtehtävät tallennettaan tietokantaan ja tietokantaohjelmistoksi valittiin redis. Redis on avain-arvo –pari tietokanta ja sen vahvuutena on nopeus joka perustuu siihen, että se on suunniteltu niin että tietokantaan muokattavaa ja luettavaa dataa käsitellään palvelinkoneen RAM-muistista.

Redis voitaisiin myös ajaa hajautettuna palveluna saman klusterin rinnalla, mutta kokeilun yksinkertaistamiseksi sitä ajettiin itsenäisenä palveluna pääpalvelimella.

4.3 **Nodet**

Klusterissa käytettäville noodeille asennettiin käyttöjärjestelmäksi Rasbian, joka Raspberry Pi Foundationin kehittämä ja julkaisema Debian-pohjainen linux-jakelu. Käyttöjärjestelmän päälle asennettiin näihinkin vielä tarvittavat ohjelmistot Docker ja Kubernetes.

Konfiguroinnissa täytyi ottaa huomioon ohjelmistojen vaatimukset, sekä että enkoodaus ja dekoodaus toteutetaan koodekilla, joka käyttää hyödykseen käytettävän järjestelmäpiirin BCM2835 sisäänrakennettua multimediaprosessoria, jota kutsutaan nimellä VideoCore. Järjestelmäpiiri jakaa RAM-muistinsa omiin osoiteavaruuksiina ARM- sekä multimediaprosessorille (Broadcom Europe Ltd, n.d.). VideoCorelle määriteltiin muistialueen kooksi 256MB lisäämällä käynnistysasetustiedostoon /boot/config.txt parametri *'gpu_mem_256'*.

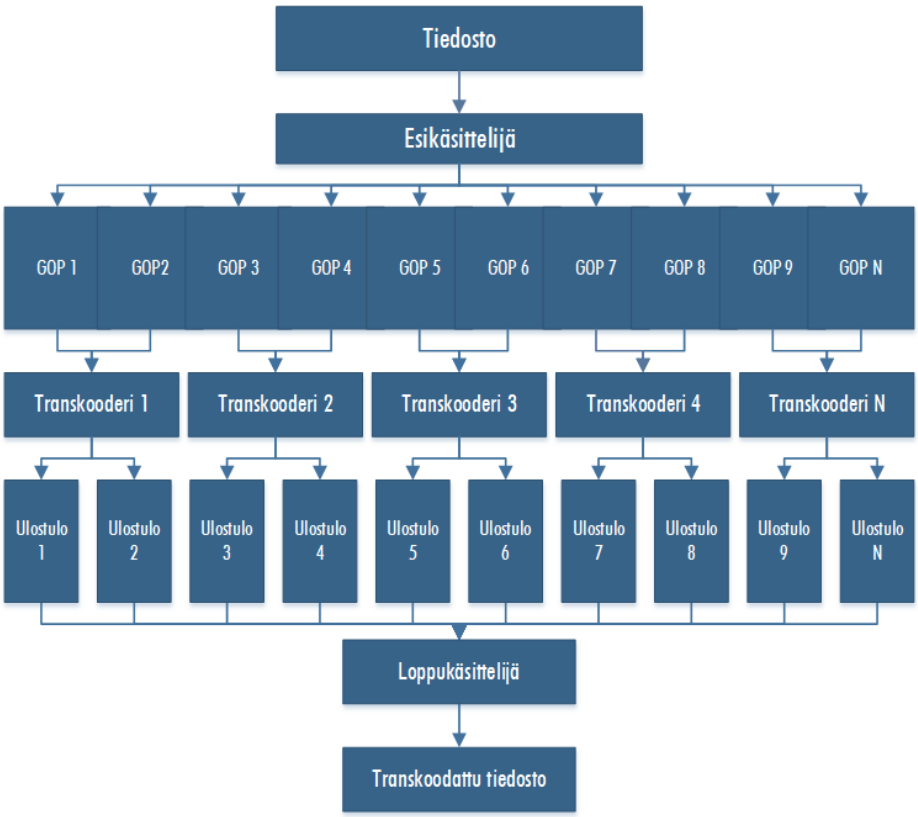
Toinen muistiin liittyvä konfiguraatio oli heittomuistin eli swap-muistitilan poistaminen, koska Kubernetes ei itsessään tue heittomuistia ja sitä käytettäessä muodostuisi muutenkin pullonkaula, koska heittomuisti tallentuisi muistikortille ja sen lukeminen ja kirjoittaminen olisi huomattavasti hitaampaa verrattuna kuin pelkän RAM-muistin käyttöön.

Linuxissa ajettavat prosessit asentavat Linux Standard Base -standardin mukaisesti oman hallintaohjelmansa niille varattuihin polkuihin *'/etc/init.d'* tai *'/etc/rc.d/rcN.d'*. Sieltä löytyy myös heittomuistin käyttöön tarvittava hallintaohjelma *'dphys-swapfile'*. Muistitila otettiin pois käytöstä ja poistettiin myös hallintaohjelma suorittamalla komentosarja: *'sudo dphys-swapfile swapoff && sudo dphys-swapfile uninstall && sudo update-rc.d dphys-swapfile remove'.* (The Linux Foundation, 2015)
Kun ohjelmat ajetaan Dockerin konttiympäristössä niin virtualisoidut kontit tarvitsevat pääsyn fyysiseen suorittimeen, muistiin, I/O- ja verkkolaitteisiin. Näitä resursseja hallinnoidaan linux-ytimen Control group eli cgroups ominaisuudella, joka hallinnoi suorittimen, muistin, levyn I/O ja verkon

käyttöä yhdelle tai useammalle prosessille. Docker vaatii toimiakseen cgroup:n sallivan pääsyn jakamaan saatavilla olevia muisti- ja suoritinresursseja. Nämä ominaisuudet otetiin käyttöön järjestelmän käynnistyessä lisäämällä raspin /boot –osiossa olevaan cmdline.txt -tiedostoon parametrit: *cgroup_enable=cpuset, cgroup_enable=memory* ja *swapaccount=1*.

4.4 Käyttö

Kun klusteri on käynnistynyt niin jokaiselle noodille jää yksi säiliö käyntiin, jossa ajettava ohjelma odottaa, että Redis-tietokantaan tulee '*jobs*'-nimiseen listaan tieto käsiteltävästä tiedostosta. Tieto lisätään tietokantaan CSV-muotoisena ja se sisältää tiedon käsiteltävästä tiedostosta, kohderesoluutiosta sekä kohdebittivirrasta. Kuvassa 11 on esitelty yleinen työnkulku aloitustiedostosta valmiiseen transkoodattuun tiedostoon.



Kuva 11. Transkooderin työnkulku.

4.5 Esikäsittelijä

Sivulla 19 kuvassa 11 työnkulku alkaa tiedostosta, josta annetaan tieto esikäsittelijälle. Esikäsittelijäksi luotiin python-ohjelma (Liite 4), jolle annetaan komentoriviparametreina käsiteltävän tiedoston nimi, tiedostopolku (johon luodaan väliaikaishakemisto) sekä tieto minne valmistunut tiedosto

tallennetaan. Lisäksi ohjelmalle annetaan parametreina halutut asetukset transkooderille.

Ohjelmalla on kaksi tarkoitusta, segmentoida videotiedosto niin että jokainen segmentti alkaa intrakoodatulla kehyksellä ja tallentaa työnimike asetuksineen tietokantaan. Samalla ohjelma toimii järjestelmän käyttöliittymänä. Videotiedosto segmentoidaan FFmpeg:llä ja sille annetut parametrit on esitelty taulukossa 1.

Taulukko 1. Segmentoinnissa annettavat parametrit.

Parametri	Arvo	Selite
-i	<tiedoston nimi>	Sisään luettavan tiedoston polku.
-vcodec	copy	Käytetään samaa videokoodekkia kuin lähdetiedostossa.
-acodec	copy	Käytetään samaa audiokoodekkia kuin lähdetiedostossa.
-map	0	Luetaan lähdetiedostosta mukaan kaikki sen sisältämät datavirrat.
-f	segment	Valitaan formaatiksi ’segment’
-flags	+cgop	Asetetaan ’closed GOP’ eli asetetaan jokainen segmentti alkamaan I-kehyksellä.
-segment_format_options	movflags=+faststart	Siirretään luodun tiedoston metadata tiedoston lopusta tiedoston alkuun.
-segment_list	<listan nimi>	Listatiedoston polku johon tallennetaan jokaisen segmenttitiedoston nimi.

Kun edellinen komento on suoritettu loppuun, ohjelma lähettää segmentoiduista tiedostoista lista –tyyppisen tietueen redis-tietokantaan rakenteella: <käsiteltävän tiedoston nimi>:<resoluutio>:<bittinopeus>:<tallennuspolku>:<enkooderi>.

4.6 Group Of Pictures

Sivulla 19 kuvassa 11 esiintyvä Group Of Pictures eli GOP on tiedosto verkkolevyllä, johon on tallennettu muunnettavasta tiedostosta yksi kuvaryhmä. Se alkaa kehyksellä, joka sisältää ainoastaan I-makroblokkeja ja sisältää vain yhden tällaisen kehyksen. Tiedostojen koko ja määrä riippuvat täysin lähdevideosta.

4.7 Transkooderi

Transkooderit ovat Raspberry Pi –laitteita, joissa jokaisessa ajetaan yhtä jatkuvasti päällä olevaa konttia. Kontissa on käynnissä pythonilla kirjoitettu ohjelma (Liite 2), joka ottaa tehtäviä töitä vastaan Redis-tietokannasta.

Konfigurointi päätettiin toteuttaa juurikin niin että Kubernetesin daemon-set-controller pitää huolen siitä, että jokaisessa noodissa pyörii jatkuvasti yksi kopio säiliöstä. Yksi säiliö sen takia, koska noodin fyysisen VideoCore-prosessorin rajapintalaitetta */dev/vchiq* pääsee käyttämään vain yksi prosessi kerrallaan. Kun uusi työ on otettu vastaan ohjelma alkaa suorittamaan muunnostyötä. Kun viimeinen työ on otettu työn alle tietokannasta, niin python-ohjelma jää odottamaan uutta suoritettavaa tehtävää.

4.8 Ulostulo

Ulostulo on muunnostyössä tehty transkoodattu Group Of Pictures eli GOP ja sen kääreen valitsee enkooderi. Vakiona tiedoston kääre luetaan transkooderille saapuvan tiedoston tiedostopäätteestä. Ulostulojen määrä on täsmälleen sama kuin lähdetiedostojen määrä.

4.9 Loppukäsittelijä

Loppukäsittelijä on käytännössä FFmpeg-ohjelmalla ajettava operaatio, jolla yhdistetään kaikki ulostulleet käsitellyt tiedostot yhdeksi tiedostopaketiksi. Esikäsittelijä luo *segments.lst* -nimisen tiedoston samaan tallennussijaintiin kuin käsiteltävät tiedostot. Tiedostossa on listattu kaikki tiedostonimet, jotka ohjelma luo. Samaan kansioon luodaan alikansio, joka nimetään resoluution mukaan ja loppukäsittelijä lukee *segments.lst* -tiedoston ja yhdistää uudet videotiedostot yhdeksi kokonaiseksi tiedostoksi komennolla *ffmpeg -f concat -safe 0 -i segments.lst -c copy full.mp4*.

5 VERTAILU

Järjestelmän nopeutta testattiin kuuden noodin kokoisella klusterilla. Kaikki noodit olivat Raspberry Pi 3 Model B -minitietokoneita ja ne liitettiin samaan verkkoon yhdellä 1GB nopeuksisella kytkimellä. Testattavana tiedostona käytettiin full hd-laatuista *Big Buck Bunny* -nimistä videoteosta joka on vapaasti käytettävissä Creative Commons Attribution 3.0 lisenssillä. Video on vapaasti ladattavissa osoitteesta https://peach.blender.org/download/.

Testityöasemana toimi nykyaikainen mediatyöasema varustettuna Intel Core i7-6700 3.40Ghz prosessorilla, 32GB RAM muistilla ja Geforce GTX 1080 näytönohjaimella. Enkooderina työasemassa käytettiin libx264-enkooderia. Transkoodaamisen mitattu aika otettiin FFmpeg:n *becnhmark* -optiolla joka tulostaa tiedoston käsittelyn jälkeen käytetyn ajan.

Taulukossa 2 on verrattuna tehotyöaseman ja klusterin käyttämät ajat saman lähdevideon transkoodaamiseen. Klusteriin käytetyssä ajassa ei ole otettu huomioon aikaa, joka kuluu tiedoston siirtämiseen verkon yli, mutta esimerkkitiedoston kohdalla lähiverkossa viive kasvaa joitakin millisekunteja. Lisäksi muita mahdollisia ulkoisia ongelmia saattaisi syntyä levypalvelimen luku/kirjoitusnopeudesta. Klusterin aika on 3 kokonaisen suorituskerran keskiarvo ja niihin käytetyt ajat olivat 172.991s, 171.275s ja 171.408s.

Taulukko 2. Transkoodaukseen käytettyjen aikojen vertailu.

	Käytetty kokonaisaika
Tehotyöasema	276.753s
Klusteri	173.891s
Erotus	**102.862s**

Yksinkertaisella vertailulla huomaa jo, että eroa suorituskyvyssä on noin 45%. Prosentuaalinen ero kahden luvun välillä lasketaan kaavan 2 mukaisesti.

$$\frac{|\Delta V|}{(\frac{\Sigma V}{2})} \times 100 \qquad (2)$$

Sivulla 22 kaavassa 2 $|\Delta V|$ on aikojen erotuksen itseisarvo ja ΣV on aikojen summa.

Klusterin virrankulutus mitattiin Paget Trading 9149 virtamittarilla, jonka tarkkuus virran ja jännitteen osalta oli +/-3 % mitatusta arvosta. Kun kaikki klusterin kuusi minitietokonetta kytkettiin sarjaan virtamittarin kanssa,

keskiarvoiseksi lukemaksi saatiin 11 000 mA, joka 5 V jännitteellä tarkoittaa 55 000 mW eli 55 W tehonkulutusta. Vertailuarvona testityöaseman pelkän prosessorin tehonkulutus on 65 W (Intel Corporation, n.d.).

6 YHTEENVETO

Tässä opinnäytetyössä tehty työ tuotti siinä mielessä tulosta, että kyettiin luomaan tavoitteen mukaisen järjestelmän, joka on toteutettu vähävirtaisilla ja kustannustehokkailla minitietokoneilla ja kykenee suoriutumaan nopeammin raskaita laskentatehtäviä vaativista prosesseista kuin yksittäinen työasemalla.

Työtä tehdessä tuli vastaan useita erilaisia teknisiä ongelmatilanteita, jotka ratkaistiin aina kuitenkin jollain tavalla. Suurimmat ongelmat johtuivat kuitenkin liiasta tavoitteellisuudesta. Esimerkiksi alkuperäisesti suunnitelmassa oli toteuttaa järjestelmä ilman ulkopuolista tallennustilaa niin, että tiedostot olisi striimattu asiakaslaitteelta palvelimelle ja lähetetty käsiteltyinä takaisin. Tämä osoittautui kuitenkin suhteellisen haastavaksi tehtäväksi useastakin syystä ja toimivan järjestelmän aikaan saamiseksi päädyttiin lopulta kuitenkin lopulta toimivaan ratkaisumalliin. Työ antoi kuitenkin erinomaisen tilaisuuden opetella ja ottaa selvää en- ja dekoodereista, virtualisoinnista, konttiteknologian käytöstä sekä hieman syvennystä python-ohjelmointiinkin. Uuden oppiminen loi raamit työlle ja se antoi enemmän kuin se otti.

Jatkokehitystä ajatellen tämän työn jälkeen on huomattavasti helpompi ja pelottomampi olo lähestyä muitakin paljon laskentaa vaativia tehtäviä ja nähdä enemmän ennalta ongelman ratkaisua rajoittavia ja mahdollistavia tekijöitä.

LÄHTEET

ACM Inc. (2016). Borg, Omega and Kubernetes. *acmQueue,* 2016 (1), s. 70 – 93

Adda, C. & Benyamina, A. (2017) Design of the H264 application and implementation on heterogeneous architectures. *International Journal of Computer Applications.* 2017 (7), s. 23 - 31

Broadcom Europe Ltd. (n.d.). *BCM2835 ARM Peripherals*. Haettu 20.7.2019 osoitteesta https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf

Carranza, C., Kober, V. & Hidalgo-Silva, H. (2010). *Proceedings of SPIE - The International Society for Optical Engineering.* Haettu 20.7.2019 osoitteesta https://www.researchgate.net/publication/253682881_Image_restoration_with_local_adaptive_methods

Chen, R. (2011). International Conference in Electrics, Communication and Automatic Control Proceedings. New York: Springer.

Docker Inc. (2019a). What is a Container? Haettu 20.7.2019 osoitteesta https://www.docker.com/resources/what-container

Docker Inc. (2019b). Docker overview. Haettu 20.7.2019 osoitteesta https://docs.docker.com/engine/docker-overview/

Encoding.com. (2019) 2019 Global Media Formats Report. Haettu 10.8.2019 osoitteesta https://www.encoding.com/resources/

Estes, P. (2017) *Multi-arch All the Things*. Haettu 28.12.2018 osoitteesta https://blog.docker.com/2017/11/multi-arch-all-the-things/

FFmpeg. 2019. About FFmpeg. Haettu 30.7.2019 osoitteesta https://ffmpeg.org/about.html

Graziano, C. (2011). A performance analysis of Xen and KVM hypervisors for hosting the Xen Worlds Project. Haettu 10.12.2018 osoitteesta https://lib.dr.iastate.edu/cgi/viewcontent.cgi?referer=&httpsredir=1&article=3243&context=etd

Hanhijärvi, J. (2009). Tutkielma eräästä videokoodausstandardista. Helsinki: Jussi Hanhijärvi Haettu 10.2.2019 osoitteesta https://users.aalto.fi/~hanhi/publications/artikkelit/h_264.pdf

InfoWorld. (2019). What is Docker? Docker containers explained. Haettu 20.4.2019 osoitteesta https://www.infoworld.com/article/3204171/docker/what-is-docker-docker-containers-explained.html

Intel Corporation. (n.d.). Intel® Core™ i7-6700 Processor. Haettu 30.8.2019 osoitteesta https://ark.intel.com/content/www/us/en/ark/products/88196/intel-core-i7-6700-processor-8m-cache-up-to-4-00-ghz.html

The Linux Foundation. (2018). Kubernetes dokumentaatio. Haettu 30.7.2018 osoitteesta https://kubernetes.io/docs/

The Linux Foundation. (2015). LSB Core Specification 5.0 . /etc: Host-specific system configuration. Haettu 10.1.2019 osoitteesta http://refspecs.linuxfoundation.org/LSB_5.0.0/LSB-Core-generic/LSB-Core-generic/etc.html

Mahaboob, B. & Kannan, M. (2018). *Literature survey on motion estimation techniques*. International Journal of Engineering & Technology. Haettu 20.7.2019 osoitteesta https://www.researchgate.net/publication/332665573_Literature_survey_on_motion_estimation_techniques

Malvar, H., Sullivan, G. & Srinivasan, S. (2008).
Lifting-based reversible color transformations for image compression. Haettu 20.11.2018 osoitteesta https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/2008_ColorTransforms_MalvarSullivanSrinivasan.pdf

Mazataud, C. (2009). *Error concealment for H.264 video transmission*. Diplomityö. Georgia Institute of Technology. Haettu 31.12.2018 osoitteesta https://smartech.gatech.edu/bitstream/handle/1853/34715/mazataud_camille_200908_mast.pdf

The New Stack. (2018) *Kubernetes vs. Docker Swarm: What's the Difference?* Haettu 15.6.2019 osoitteesta https://thenewstack.io/kubernetes-vs-docker-swarm-whats-the-difference/

O'Reilly Media, Inc. (n.d.). Fundamentals of Video Compression. Haettu 30.8.2019 osoitteesta https://learning.oreilly.com/library/view/visual-media-coding/9780470740576/chap02-sec003.html

Red Hat Inc. (n.d.). *What is Docker?* Haettu 13.12.2018 osoitteesta https://opensource.com/resources/what-docker

Richardson, I. (2010). *The H.264 advanced video compression standard.* Iso-Britannia: John Wiley & Sons, Ltd.

Velez, G. (2019). Kubernetes's diagram. Haettu 9.9.2019 osoitteesta https://containerjournal.com/topics/container-ecosystems/kubernetes-vs-docker-a-primer/

Winkler, S., Lambrecht, C. & Kunt, M. (2001). *Vision models and applications to image and video processing*. Sveitsi: Springer Science & Business Media.

Liite 1

Persistent volume YAML -konfiguraatio

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: data
spec:
  capacity:
    storage: 20Gi
  accessModes:
    - ReadWriteMany
  mountOptions:
    - nfsvers=3
  nfs:
    # NOTE: use the nfs server address
    server: 192.168.100.19
    path: "/data"
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: data
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 20Gi
```

Liite 2

DaemonSet YAML -konfiguraatio

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: clusnpi
spec:
  selector:
    matchLabels:
      name: hw-accelerated-encoder
  template:
    metadata:
      labels:
        name: hw-accelerated-encoder
    spec:
      containers:
      - name: c1
        image: menturio/clusnpi
        volumeMounts:
        - name: test-nfs
          mountPath: "/nfs"
        - name: vchiq
          mountPath: "/dev/vchiq"
        securityContext:
          privileged: true
      restartPolicy: Always
      volumes:
      - name: test-nfs
        persistentVolumeClaim:
          claimName: test-nfs
      - name: vchiq
        hostPath:
          path: "/dev/vchiq"
```

Liite 3

Dockerfile, menturio/clusnpi

```
FROM raspbian/stretch
MAINTAINER Petri Pakkanen "petri.pakkanen@student.hamk.fi"
RUN  apt-get update \
   && apt-get upgrade -y \
   && apt-get install -y \
      python3 \
      python3-pip \
      dpkg \
      ffmpeg \
      libomxil-bellagio-bin \
      libomxil-bellagio-dev \
   && pip3 install redis \
   && echo gpu_mem=256 >> /boot/config.txt

COPY ./worker.py /worker.py
COPY ./rediswq.py /rediswq.py

CMD python3 worker.py
```

Liite 4

Esikäsittelijä, Python-ohjelma

```
#!/usr/bin/python
import sys, getopt, subprocess, uuid, redis, string
from subprocess import Popen, PIPE

def print_help():
   print('split_video.py -i <inputfile>\n \
              -o <outputfilename>               (default: generated)\n \
              -r <resolution>                   (default: 1920x1080)\n \
              -b <bitrate>                      (default: 10485760)\n \
              -s <redis hostname | ip>          (default: 127.0.0.1)\n \
              -t <redis port>                   (default: 6379)\n \
              -p <output_path>\n \              (NFS path ON TARGET) \
              -d <decode, h264 | mpeg2 | mpeg4 | vc1>  (default: h264)\n \
              -l <logging, 0 | 1>               (default: 0)')

def main(argv):
   if not argv:
      print('Script requires arguments. Use -h parameter.')
      sys.exit(2)

   #Generate id for job and for tmp files
   jobID = uuid.uuid4()
   #Variables are re-assigned with values from database
   res = '1920x1080'
   bitrate = '10485760'
   decoder = 'h264'
   input_file = ''
   logging = 1
   output_path = ''
   #Assign default value for outputfile
   output_file = str(jobID)
   redis_host_addr = '127.0.0.1'
   redis_port = 6379

   try:
      opts,    args    =    getopt.getopt(argv,"hi:or:b:s:p:dlt",["input_file=","out-
put_file","resolution=","bitrate=","redis-host", "output-path=", "decoder", "logging",
"port"])
      print(opts)
   except getopt.GetoptError:
      print_help()
      sys.exit(2)
   for opt, arg in opts:
      if opt == '-h':
         print_help()
         sys.exit()
      elif opt in ("-i", "--ifile"):
         input_file = arg
      elif opt in ("-o", "--ofile"):
         output_file = arg
      elif opt in ("-r", "--resolution"):
         res = arg
      elif opt in ("-b", "--bitrate"):
         bitrate = arg
      elif opt in ("-s", "--redis-host"):
         redis_host_addr = arg
      elif opt in ("-p", "--output-path"):
         output_path = arg
      elif opt in ("-d", "--decoder"):
```

```
         decoder = arg
      elif opt in ("-l", "--logging"):
         logging = arg
      elif opt in ("-t", "--port"):
         redis_port = arg
   p0 = Popen(["/bin/mkdir", "-p", output_path + "/" + str(jobID)])
   p0.wait()
   p01 = Popen(["ffmpeg", "-i", input_file, \
                  "-acodec", "copy", \
                  "-vcodec", "copy", \
                  "-f", "segment", \
                  "-segment_format_options", "movflags=+faststart", \
                  "-flags", "+cgop", \
                  "-segment_list", str(jobID), \
                  "-segment_list", output_path + '/' + str(jobID) + '/segments.lst',
\
                  "-map", "0", output_path + "/" + str(jobID) + "/" + output_file +
"_%d." + input_file.rsplit('.',1)[1]])
   p01.wait()
   print("redis_port = " + str(redis_port))
   r = redis.Redis(host=redis_host_addr, port=32666, db='0')
   with open(output_path + '/' + str(jobID) + '/segments.lst') as f:
      for line in f:
         r.rpush('jobs', line.rstrip('\n') + ':' + res + ':' + bitrate + ':' + out-
put_path + ':' + decoder + ':' + str(logging))
         print(line.rstrip('\n') + ':' + res + ':' + bitrate + ':' + output_path + ':'
+ decoder + ':' + str(logging))
if __name__ == "__main__":
   main(sys.argv[1:])
```

Liite 5

Transkooderi, Python-ohjelma

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import time
import rediswq
import subprocess
import resource
from subprocess import Popen, PIPE
from subprocess import STDOUT, check_output
from threading import Timer

host = 'redis'
rdy_dir_name = 'rdy'
encoder = 'h264_omx'
package = 'mp4'
logging = 0

q = rediswq.RedisWQ(name='jobs', host=host)
print('Worker with sessionID: ' + q.sessionID())
while not q.empty():
   mainItem = q.lease(lease_secs=10, block=True, timeout=2)
   if mainItem is not None:
      mainItemStr = mainItem.decode('utf=8')
      print('Working on ' + str(mainItemStr))
      itemLst = mainItemStr.split(':')
      filename = itemLst[0]
      resolution = itemLst[1]
      bitrate = itemLst[2]
      output_path = itemLst[3]
      decoder = itemLst[4] + '_mmal'      # only for _mmal codecs!!!
      logging = itemLst[5]
      baseName = itemLst[0].split('_')[0]
      rdy_dir = output_path + '/' + baseName
      rdy_sub_dir = '/' + rdy_dir_name
      if logging:
         l = Popen(['export', 'FFREPORT=file=' + rdy_dir + '/' + filename +
'.log:level=32'])
         l.wait()
      else:
         l = Popen(['unset', 'FFREPORT'])
         l.wait()
      d = Popen(['mkdir', '-p', rdy_dir])
      d.wait()
      p = Popen(['ffmpeg', '-benchmark', \
                  '-c:v', decoder, \
                  '-i', rdy_dir + '/' + filename, \
                  '-c:v', encoder, \
                  '-s', resolution, \
                  '-b:v', bitrate, \
                  '-c:a', 'copy', \
                  '-f', package, rdy_dir + rdy_sub_dir + '/' + filename], \
                  stdout=PIPE, stderr=PIPE)
      while 1:
         output = p.stderr.readline()
```

```
        stdout = p.stdout.readline()
        if stdout:
           print(stdout)
        if output == '' and p.poll() is not None:
           break
        if output:
           print(output)
           if b'muxing overhead' in output.rstrip():
              q.complete(mainItem)
              break
   else:
      print('Waiting for work')
print('Queue empty, exiting')
```