Torsti Laine

Bluetooth Mesh -ohjainten taakanjako

Metropolia Ammattikorkeakoulu
Insinööri (AMK)
Tieto- ja viestintätekniikka
Insinöörityö
11.2.2022

Tiivistelmä

Tekijä:	Torsti Laine
Otsikko:	Bluetooth Mesh -ohjainten taakanjako
Sivumäärä:	42 sivua
Aika:	11.2.2022

Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Lehtori Simo Silander Ohjelmistoarkkitehti Oliver Jones

---

Insinöörityössä kehitettiin ohjelmistomoduuli, joka vastaa taakanjaosta Bluetooth Mesh -verkkoteknologiaan perustuvan valaisujärjestelmän yhteydessä toimivien ohjainlaitteiden välillä. Ohjain toimii yhdyskäytävänä valaisujärjestelmän ja pilvipalvelujen välillä.

Taakanjakomoduulin viestintää varten kehitettiin kaksitasoinen viestintäkerros, joka käyttää langallista lähiverkkoa. Alemman tason tehtävä on saattaa ohjaimet tietoisiksi toisistaan UDP-monilähetysviestejä käyttäen. Ylempi viestintäkerroksen taso vastaa varsinaisesta ohjainten välisestä viestinnästä, joka on toteutettu AMQ-viestintäprotokollaa käyttävällä RabbitMQ-viestijonolla. Kehitetty moduuli jakaa verkkosolmuja ohjainten välillä perustuen solmuilta vastaanotettavien viestien ’hops’-kentän arvoon, jolla mitataan viestin kulkemaa etäisyyttä verkossa. Periaatteena on, että ohjaimet saavat itseään lähinnä olevat solmut omistukseensa. Moduulia testattiin kahden ohjaimen välillä. Ohjainten tietokantoihin luotiin 200 lumesolmua, joille tuotettiin satunnaisesti ’hops’-arvo. Aluksi kaikki lumesolmut asetettiin toisen ohjaimen omistukseen, minkä jälkeen odotettiin taakanjako-operaation käynnistymistä. Toisessa testissä kullekin ohjaimelle annettiin 100 lumesolmua ja taakanjako-operaatio käynnistettiin samanaikaisesti kummassakin ohjaimessa.

Viestintäkerros luo yhteydet ohjainten välille automaattisesti, kykenee toipumaan virhetilanteista ja luo yhteydet uudelleen, mikäli yhteys katkeaa. Ohjainten välinen viestintä on luotettavaa, eikä testeissä havaittu yhtäkään tapausta, jossa viestit olisivat menneet sekaisin, tai viesti ei olisi saapunut määränpäähänsä.

Moduulin suorittama taakanjako toteutui odotetulla tavalla, ja jokaisessa toistossa noin puolet lumesolmuista siirrettiin toisen ohjaimen omistukseen. Samanaikaisten taakanjako-operaatioiden viestintä ei häiriintynyt, ja molemmissa tapauksissa lopputulos oli odotetun kaltainen.

Avainsanat:	Bluetooth Mesh, RabbitMQ, valaisujärjestelmä, yhdyskäytävä, taakanjako

Abstract

Author:	Torsti Laine
Title:	Load Balancing Between Controllers of Bluetooth Mesh Network
Number of Pages:	42 pages
Date:	11 February 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Software Engineering
Supervisors:	Simo Silander, Senior Lecturer Oliver Jones, Software Architect

The goal of the study was to create a load balancing module capable of distributing ownership of Bluetooth Mesh Network nodes of a wireless lighting solution between network control devices. The controller in question is a gateway device and transmits data between the Bluetooth Mesh based lighting system and cloud services.

A communication layer, based on Ethernet technology, was developed for controller-to-controller messaging. The lower communication layer uses UDP multicast messages and is responsible for controller discovery and providing the necessary information to build the upper layer. Actual messaging between the controllers is managed by the upper communication layer, which utilizes the RabbitMQ message broker in routing the messages. The load balancing module distributes the Bluetooth Mesh nodes among the connected controllers by assigning the nodes to the controllers nearest to them. The distance between the node and a controller is determined by how many times the message was relayed before reaching the controller, which is available in the ‘hops’ field that each Mesh message contains. The functionality of the module was tested on two connected controllers by generating 200 fake nodes in the controllers’ databases. Each node had a randomly generated ‘hops’ count with a value ranging from one to four and was assigned to one of the controllers. After the setup the communication layer was brought online, and the load balancing process was triggered.

The controllers were connected to each other automatically once the communication layer was brought up and was able to recover after a connection loss. All messages between the controllers were relayed successfully across multiple test instances and not a single case was observed where the messaging would have gone out of sync or otherwise disrupted.

The load balancing operation progressed as expected. Each time the test was run, approximately half of the fake nodes were transferred to the other controller according to the used load balancing rules. A minor deviation in the number of devices distributed was observed, but also expected due to randomly generated connection statistics.

Keywords: Bluetooth Mesh, Gateway, Lighting Solution, RabbitMQ, Load Balancing

Sisällys

Lyhenteet

1 Johdanto 1

2 Bluetooth Mesh -verkkoprotokolla 2

2.1 Solmu 3
2.2 Monesta moneen -yhteys 4
2.3 Verkon ulkopuolisten laitteiden yhteys 5
2.4 Viestit 6

3 ActiveAhead Generation 2 7

3.1 ActiveAhead-ohjainlaite 8
3.1.1 Docker-säiliöintialusta 8
3.1.2 Redis-tietokanta 9
3.1.3 RabbitMQ 10
3.2 ActiveAhead-ohjainohjelmisto 11
3.2.1 JavaScript ja Node.js 11
3.2.2 JSON-skeema ja sen validointi 12

4 ActiveAhead-ohjaimen toiminnallisuus 13

4.1 AA-ohjainten välinen viestintä 15
4.2 Tiedon kerääminen 18
4.3 Taakanjakomoduulin ominaisuudet 18

5 AA-ohjainten väliset yhteydet 19

5.1 RabbitMQ-konfiguraatiotiedosto 20
5.2 Ethernet-moottori ja UDP 23
5.3 Taakanjakomoduulin viestintä ja RabbitMQ-asiakasohjelma 27

6 Taakanjakoalgoritmi 31

6.1 Taakanjakoviestit 32
6.2 Viestinkäsittely ja toimenpiteet 35

7 Taakanjaon testaus ja loppusanat 38

7.1 Tulokset 39

7.2 Lopuksi 39

Lähteet 41

Lyhenteet

AA: *ActiveAhead*. Helvarin langaton valaisujärjestelmä.

AMQP: *Advanced Message Queueing Protocol*. Avoimeen standardiin perustuva viestiprotokolla, joka kykenee luomaan viestijonoja ja tukee salausta.

BLE: *Bluetooth Low Energy*. Likiverkkotekniikka, jonka tavoitteena on minimoida energian kulutus.

JSON: *JavaScript Object Notation*. Avoimeen standardiin perustuva tiedonvälityksessä käytettävä tiedostomuoto.

MQTT: *Message Queue Telemetry Transport.* Avoimeen standardiin perustuva viestiprotokolla, joka on erityisen kevyt. Ei tue salausta.

TCP: *Transmission Control Protocol.* Luotettava tietoliikenneprotokolla, jolla on kyky toipua virhetilanteista.

TLS: *Transport Layer Security.* Salausprotokolla, jota käytetään tietoliikenteen suojaamiseen IP-verkoissa.

TTL: *Time-To-Live.* Bluetooth Mesh -viestien sisältämä kenttä, joka määrittää sen, kuinka monta kertaa kyseinen viesti voidaan toistaa. Toisin sanoen kenttä määrittää, kuinka pitkälle viesti kulkee verkossa.

STOMP: *Streaming Text Oriented Messaging Protocol*. Yksinkertainen tekstipohjainen viestiprotokolla.

UDP: *User Datagram Protocol*. Kevyt ja yksinkertainen kuljetuskerroksen protokolla, joka ei sisällä virheentarkistusta.

1 Johdanto

Valaisujärjestelmien kehitys oli pitkään verrattain hidasta, koska itse valaisinteknologiassa ei tapahtunut paljoakaan muutoksia ja rakennusten valaisujärjestelmät tähtäsivät pääasiassa energiasäästöihin. Alan kehitys alkoi kiihtymään huomattavasti LED-valaisimien ilmestyessä markkinoille kaksituhattaluvun alussa. LED-teknologia avasi ovet valaisinjärjestelmien innovaatioille, koska ne tarjosivat huomattavasti laajemman toiminnallisuuksien kirjon ja kuluttivat samalla vähemmän energiaa kuin perinteisesti käytetyt loisteputket. Matalamman energiakulutuksen lisäksi LED-valaisimien tärkeimpiä ominaisuuksista on, että valaisimen laittaminen pois päältä ja takaisin päälle ei vähennä sen elinikää ja että valonvoimakkuutta voidaan säätää himmentämällä. Vaikka himmennettäviä valaisimia oli markkinoilla myös aiemmin, ne olivat kalliita eikä himmennystaso ollut lähelläkään sitä, mihin LED-valaisimet kykenevät. Yhdistämällä LED-valaisin valoanturiin pystyttiin mukauttamaan valaisimen toiminta tilaan tulevaan luonnolliseen valoon, jolloin energiatehokkuus kasvoi entisestään.

Toinen merkittävä askel valaisujärjestelmien kannalta on ollut älyrakennusten kehitys. Rakennuksissa toimivien järjestelmien avulla kerätään tietoa rakennuksen käytöstä ja mukautetaan järjestelmien toimintaa käyttötarpeiden mukaan. Uusi amerikkalainen pääasiallisesti rakennusalaan kohdistuva WELL-standardi vie rakennusautomaation, mukaan lukien valaisujärjestelmät, ja sen päämäärän vielä pidemmälle. Yksi standardin tavoitteista on kerätä ja analysoida tietoa rakennuksen käytöstä, ja sen pohjalta ohjata rakennusjärjestelmiä parantamaan rakennuksen käyttäjien hyvinvointia.

Helvar aloitti toimintansa 1920-luvulla ja siirtyi valonohjauskomponenttien valmistukseen 1960-luvun puolessa välissä. 1970-luvulla toiminta keskittyi valaisualalle. Tänä päivänä Helvar on Suomen johtava älykkäiden valaisujärjestelmien tuottaja ja on pitkään ollut yksi alan edelläkävijöistä.

ActiveAhead-tuoteperhe on Helvarin langaton valaisujärjestelmä, joka hyödyntää keskinäisessä viestinnässä Bluetooth Mesh -verkkoteknologiaa. Tämän

opinnäytetyön aiheena on kehittää ActiveAhead-järjestelmään kuuluvaan ohjain/yhdyskäytävä-laitteeseen taakanjako-ominaisuus, jonka avulla järjestelmän luotettavuutta ja tiedonkeruuominaisuuksia pystytään tehostamaan.

2 Bluetooth Mesh -verkkoprotokolla

Tämä luku alilukuineen perustuu Woolley Martinin (2020, 16–19) verkkojulkaisuun *Bluetooth Mesh Networking - An Introduction for Developers*. Bluetooth Mesh -verkkoprotokolla on Bluetooth Special Intrests Groupin (Bluetooth SIG) vuonna 2017 julkaisema protokolla, joka on kehitetty erityisesti rakennusautomaatiossa ja WELL-rakennuksissa käytettäväksi runkoverkoksi. Rakennusautomaatiolla tarkoitetaan lämmitys-, valo-, ilmanvaihto- ja hälytysjärjestelmien automaattista ohjausta, jolla pyritään vähentämään energian kulutusta ja tehostamaan rakennuksen toimintaa käyttötarpeisiin nähden.

WELL-standardin ja sitä noudattavan rakentamisen ydinajatus on rakennuksen käyttäjien hyvinvointi ja standardin kehitystä ohjaavat tutkimukset siitä, miten ympäristö vaikuttaa ihmisen hyvinvointiin. Rakennusautomaation ja WELL-standardin toteuttaminen käytännössä edellyttää rakennuksen kattavaa verkkoa, jolla voidaan kerätä tietoa tilojen käytöstä. (International WELL Building Institute. 2020: 1.)

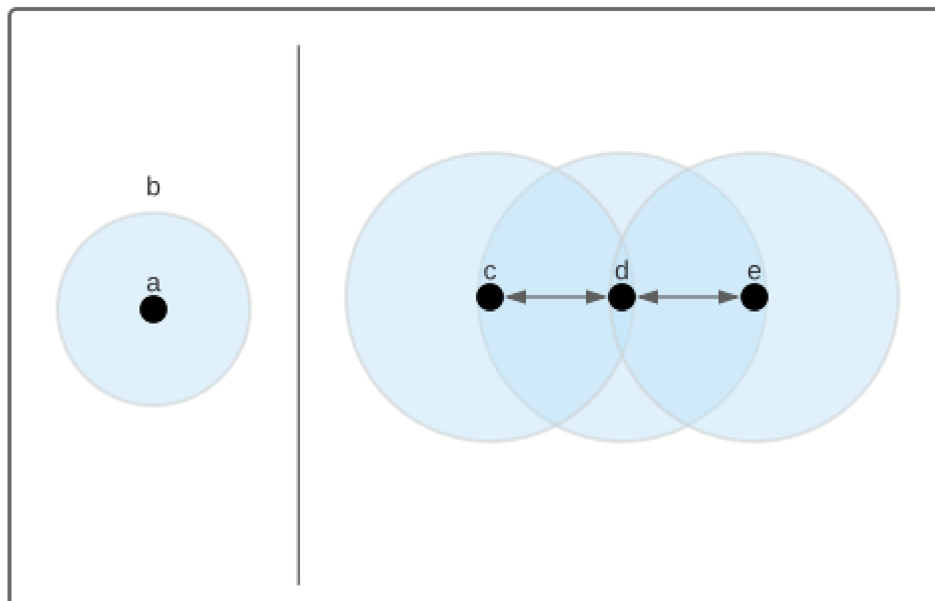
Bluetooth Mesh -verkkoprotokollan kehityksen tavoitteena oli, että verkko saataisiin sulautettua rakennuksen valaisujärjestelmään. Tällöin Bluetooth Mesh -verkkoa varten ei tarvittaisi erillistä infrastruktuuria, ja sen asennuskustannukset olisivat matalat uusissa ja vanhoissa rakennuksissa. Verkon ollessa osa rakennuksen valaisujärjestelmää se pystyy samalla kattamaan myös rakennuksen välittömässä läheisyydessä sijaitsevat ulkoiset tilat.

Bluetooth Mesh -verkkoteknologia perustuu solmulaitteisiin (eng. *node*), jotka pystyvät viestimään keskenään. Täten solmut luovat yhtenäisen verkoston, jota kutsutaan Bluetooth Mesh -verkoksi. Solmuihin on mahdollista kiinnittää muita laitteita, kuten esimerkiksi liiketunnistimia, hälyttimiä, hiilidioksidimittareita, joilla

voidaan kerätä tilojen käyttöön liittyvää dataa. Toisaalta solmuun voidaan liittää myös toiminnallisia laitteita kuten valaisimia, ilmastointilaitteita ja lämmittimiä, joiden toimintaa voidaan siten ohjata etänä. Bluetooth Mesh -verkkoprotokollan kehityksessä tavoitteena on, että tulevaisuudessa kaikki rakennuksessa toimivat järjestelmät voidaan liittää osaksi koko rakennuksen kattavaa Bluetooth Mesh -verkkoa, jonka saavutettavuuden perusta on koko rakennuksen kattava valaisujärjestelmä.

2.1 Solmu

Kuten edeltävässä luvussa asiaa sivuttiin, solmu on laite, joka kykenee vastaanottamaan ja lähettämään Bluetooth Mesh -verkossa kulkevia viestejä. Tämä lisäksi solmuilla voi olla myös verkon toiminnan kannalta oleellisia rooleja, joiden myötä ne pääsevät käsiksi lisätoiminnallisuuksiin, ja niiden merkitys osana Bluetooth Mesh -verkkoa muuttuu. Tällaisia rooleja ovat yhteyssolmu, välittäjäsolmu, ystäväsolmu ja energiansäästösolmu. Tämän opinnäytetyön osalta oleellisia ovat yhteys- ja välittäjäsolmut, joita käsitellään seuraavissa alaluvuissa tarkemmin.



Kuva 1. Bluetooth Mesh -solmuja (a, c, d ja e)

Bluetooth Mesh -verkko koostuu siis keskenään viestivistä solmuista, joita kuvassa 1 merkitään mustilla täplillä (a, c, d ja e). Solmun ’a’ signaalin kantama on merkitty kuvaa jakavan viivan vasemmalla puolella kirjaimella ’b’, ja sitä havainnollistaa vaalean sininen taustaväri. Mesh-verkon kantamaa kuvassa 1 ilmentää solmujen ’c, d, e’ signaalikantamien muodostama unioni, jonka sisällä olevat solmut pystyvät viestimään keskenään vapaasti. Solmu ’d’ pystyy viestimään suoraan solmuille ’c’ ja ’e’. Välittämällä edellä mainittujen solmujen viestejä ’d’ mahdollistaa myös solmujen ’c’ ja ’e’ välisen viestinnän.

2.2 Monesta moneen -yhteys

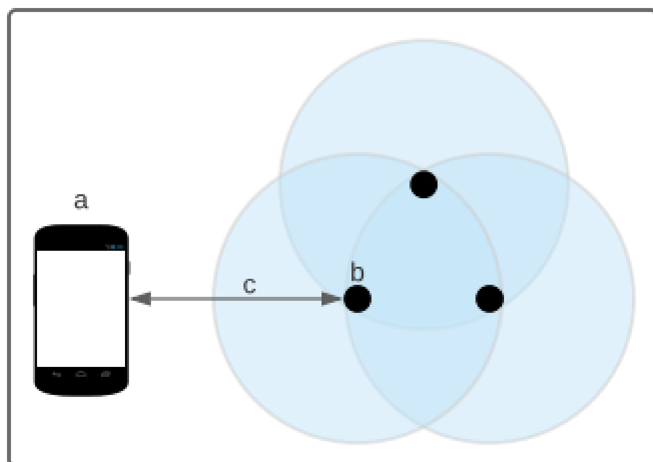
Aiemmin Bluetooth Low Energy (BLE) -viestintä on tapahtunut langattoman likiverkon kautta (*Wireless Personal Area Network, WPAN*), joka tunnetaan Bluetooth-protokollassa nimellä piconet. Tällaisessa verkossa voi olla samaan aikaan aktiivisena 8 laitetta, joista yksi on aina *master* ja muut 7 laitetta ovat *slave*-laitteita. Kaikki piconetissä lähetetyt viestit kulkevat *master*-laitteen kautta, eli kyseessä on pisteestä pisteeseen -periaatteen (*point-to-point*) mukainen viestintä. Bluetooth Mesh -verkossa solmut viestivät monesta moneen -periaatteen (*many-to-many*) mukaisesti julkaise/tilaa-periaatteen (*publish/subscribe*) mukaisesti.

Osa mesh-verkon solmuista saa välittäjäsolmun roolin (kuva 1 solmu ’d’), mikä tarkoittaa, että normaalien toimintojen lisäksi ne toistavat vastaanottamiaan viestejä. Toisin sanoen välittäjäsolmut toimivat ikään kuin reitittiminä, mikä mahdollistaa verkon lähes rajattoman laajentamisen. Toinen välittäjäsolmujen tuoma etu on, että viestit voivat kulkea määränpäähänsä useita eri reittejä, joten yksi viallinen solmu ei heikennä verkon toimintaa. Välittäjäsolmuroolin ansiosta Bluetooth Mesh -verkko pystyy kattamaan erittäin suuria alueita, mikä käytännön syistä ei ole välttämättä mahdollista muita verkkoteknologioita hyödyntämällä.

Bluetooth Mesh -verkossa viestin elinaikaa (*Time-To-Live, TTL*) mitataan yksiköllä nimeltään ’hops’, joka käytännössä tarkoittaa sitä, kuinka monen välittäjäsolmun kautta viesti on kulkenut. Jokainen mesh-verkossa kulkeva viesti sisältää TTL-kentän, jonka avulla voidaan ennalta määrittää, kuinka pitkälle viesti voi kulkea verkossa. Asettamalla kentän arvoksi neljä, määrätään, että kyseisen viestin voi välittää korkeintaan neljä kertaa, ja aina välittäjäsolmun toistaessa viestin kentän arvosta vähennetään yksi. TTL-kentän arvo 0 tarkoittaa sitä, ettei kyseistä viestiä enää toisteta. Tutkimalla vastaanotetun viestin TTL-kentän arvoa voidaan arvioida viestin lähettäjän ja vastaanottajan välistä etäisyyttä, tai kun tiedetään kahden solmun olevan lähellä toisiaan, mutta niiden välisten viestien TTL-kentän arvo on matala, on mahdollista, että verkossa on häiriö esimerkiksi viallinen välittäjäsolmu, minkä takia viesti joutuu kiertämään tavallista kauempaa. Jokaisella solmulla on myös lyhytaikainen välimuisti, joka estää sen, ettei välittäjäsolmu toista tai vastaanottava solmu vastaanota samaa viestiä kuin kerran.

2.3 Verkon ulkopuolisten laitteiden yhteys

Bluetooth Mesh -verkon ulkopuoliset laitteet voivat olla yhteydessä verkkoon muodostamalla yksi-yhteen-yhteyden johonkin verkon solmuun. Mesh-verkon ulkopuolisen laitteen muodostaessa yhteyttä solmuun kohteena oleva solmu omaksuu yhteyssolmun roolin.



Kuva 2. Ulkopuolisen laitteen yhteys Bluetooth Mesh -verkkoon

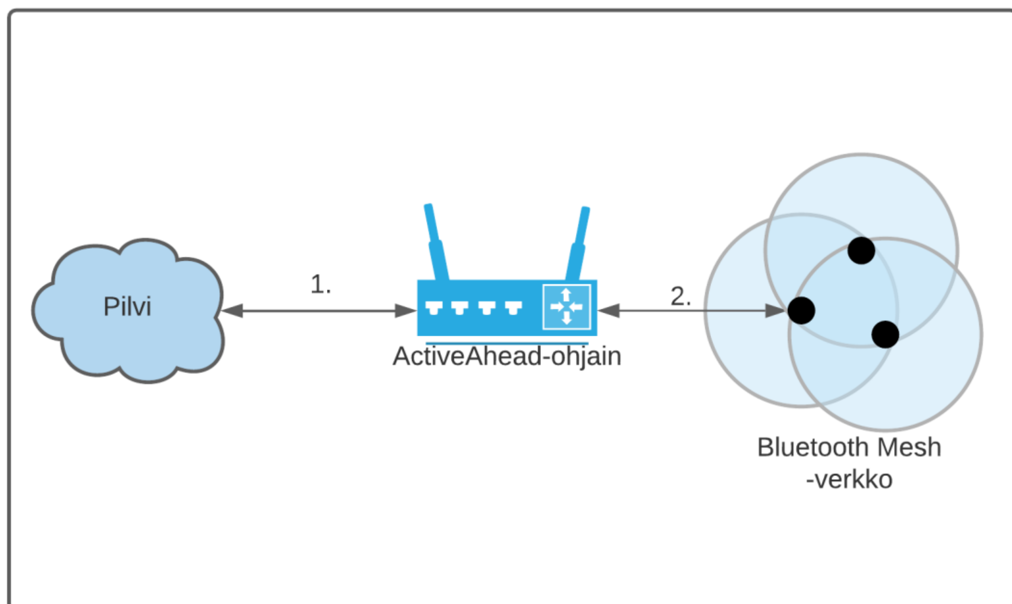
Kuvassa 2 Bluetooth Mesh -verkon ulkopuolinen laite ’a’ on muodostanut yksi-yhteen-yhteyden ’c’ verkon solmuun ’b’. On kuitenkin tärkeää ottaa huomioon se, että vaikka verkon ulkopuolinen laite pystyy muodostamaan yhteyden mesh-verkkoon, se ei kuitenkaan ole osa itse verkkoa. Käytännössä tämä tarkoittaa sitä, ettei ulkopuolisella laitteella ole omaa verkko-osoitetta, vaan se vastaanottaa ja lähettää viestejä yhteyssolmun osoitteella.

2.4 Viestit

Kuten aiemmin todettiin, solmujen välinen kommunikaatio perustuu viestien julkaisemiseen ja niiden kuunteluun. Viestityyppejä on kahdenlaisia: viestit, jotka edellyttävät vastausta, ja viestit, jotka eivät edellytä vastausta. Ensimmäisenä mainitusta esimerkkinä toimii viesti, jolla pyritään selvittämään solmun tilaa, jolloin vastauksena lähetetään solmun tila. Toinen esimerkki vastausta edellyttävästä viestistä on solmulle annettu komento, jonka vastaanotettuaan solmu tiedottaa lähettäjälle, että viesti on otettu vastaan onnistuneesti. Sellaiset viestit, jotka eivät edellytä vastausta, viestivät esimerkiksi laitteen tai solmun sen hetkisestä tilasta.

3 ActiveAhead Generation 2

ActiveAhead (AA) -tuoteperhe on Helvarin kehittämä langaton valaisujärjestelmä. Sen pääominaisuuksia ovat järjestelmän helppo skaalautuvuus, kyky kerätä tietoa toimintaympäristöstä ja mukautua automaattisesti tilojen käyttötarpeisiin. Tavoitteena on parantaa tilojen viihtyvyyttä ja samalla laskea valaisujärjestelmän energiakulutusta. (Helvar. ActiveAhead Generation 2.)



Kuva 3. ActiveAhead-valaisujärjestelmä

ActiveAhead-valaisujärjestelmän ohjain on Bluetooth Mesh -verkon ulkopuolinen laite, joten se luo yksi-yhteen-yhteyden lähimpään solmuun (kuva 3 kohta 2.), minkä lisäksi ohjain on myös yhteydessä Helvarin pilvipalveluihin internetyhteyden kautta. Ohjaimella on kolme päätehtävää. Ensinnäkin ohjain kuuntelee verkon solmujen julkaisemia viestejä, ja päivittää solmujen sekä niihin liitettyjen laitteiden kuten anturien ja valaisimien tilan paikalliseen tietokantaan sekä välittää vastaanottamansa tilapäivitykset eteenpäin Helvarin pilvipalveluihin, joissa kerätty data säilötään, analysoidaan ja mallinnetaan loppukäyttäjälle esimerkiksi sähkönkulutusgraafina. Toisekseen ohjain käsittelee vastaanottamaansa dataa

ja ilmoittaa pilvipalveluihin mahdollisista verkkohäiriöistä, kuten kuolleesta solmusta tai viallisesta valaisimesta. Kolmantena ohjain hallinnoi verkon solmuihin liitettyjä laitteita julkaisemalla komentoja verkkoon. Nämä komennot voivat olla paikallisia, eli ohjaimen automaatioon perustuvia tai ne saattavat olla loppukäyttäjän pilven kautta ohjaimelle lähetettyjä komentoja.

3.1 ActiveAhead-ohjainlaite

ActiveAhead-ohjain on SolidRun yrityksen SolidSense N6 -yhdyskäytävälaitteella, joka alkujaan suunniteltiin palvelemaan paikallista IoT-laiteverkkoa. Ohjainohjelmistoa ajetaan omassa säiliössä, ja sen toiminta edellyttää myös Redis-tietokannan ja RabbitMQ-viestijonopalvelimen asentamista. AA-ohjaimessa myös Redis-tietokanta ja RabbitMQ-viestijono-ohjelmisto ajetaan erillisissä säiliöissä.

3.1.1 Docker-säiliöintialusta

Säiliöiden käyttö ohjelmistokehityksessä on yleistymässä, koska ne eivät ole käyttöjärjestelmäriippuvaisia, minkä ansiosta ne eivät tarvitse erillistä virtuaalikonetta ohjelmiston ajamiseksi. Säiliö on normaalista käyttöjärjestelmästä erillinen ajoympäristö, minkä ansiosta siinä ajettavan ohjelman toiminta on helpommin ennakoitavissa, eivätkä esimerkiksi käyttöjärjestelmäpäivitysten aiheuttamat muutokset vaikuta ohjelmiston toimintaan. (Ghandi, Rajeev – Szmrecsanyi, Peter. 2019.) Säiliöiden käyttö lisää myös verkkoturvallisuutta, koska ohjelmiston saavutettavuutta verkon kautta voidaan säätää säiliön konfiguraatiotiedostossa, jolloin ne eivät ole riippuvaisia isäntäkäyttöjärjestelmästä.

Docker on avoimeen lähdekoodiin perustuva säiliöintialusta, joka helpottaa ohjelmiston käyttöönottoa, lähdekoodin kääntämistä ohjelmaksi, ajamista, päivittämistä ja pysäyttämistä. Käyttämällä *Docker Compose* -liitännäistä, on mahdollista luoda usean säiliön sisältävä ajoympäristö. (Docker, Inc. Docker overview.)

3.1.2 Redis-tietokanta

Bluetooth Mesh -verkossa voi sen koosta riippuen olla useita tuhansia solmuihin liitettyjä antureita, valaisimia tai muita laitteita, minkä takia solmut julkaisevat laitekohtaisia päivityksiä jatkuvasti. Lyhyellä aikavälillä vastaanotettavien viestien suuresta määrästä johtuen on erittäin tärkeää, että AA-ohjaimen paikallinen tietokanta toimii nopeasti ja luotettavasti. Huomioon on otettava myös se, että verkossa olevat laitteet, kuten valaisimet ja anturit saattavat muuttaa tilaansa taajaan. Edellä mainitun takia tietokantaan viedään suuri määrä tilapäistä tietoa, jota ei välttämättä tarvitse kirjoittaa levylle.

Redis on avoimeen lähdekoodiin perustuva tietorakennevarasto, jota voidaan käyttää välimuistina, viestinvälittäjänä tai relaatiomallista poikkeavana tietokantana. Se mahdollistaa atomisten operaatioiden käytön tukemilleen tietotyypeille, joita ovat merkkijonot, hajautustaulut, listat, joukot ja järjestetyt joukot. Redis säilyttää välimuistissa tietoaineistoa, jonka määrätyin väliajoin kirjoitetaan levylle. Tämä nopeuttaa tietokannan toimintaa tilanteissa, joissa tietokanta joutuu käsittelemään paljon dataan kohdistuvia päivityksiä ja tärkeää on säilöä viimeisimmät muutokset. (Redis ltd. Introduction to Redis.)

Redis-tietokanta on erinomainen valinta, kun tarvitaan erittäin nopeaa tietokantaan kirjoitusta, lukua tai päivitystä. Tiedot ovat olemassa muistissa, ja ne kirjoitetaan levylle määrätyin väliajoin, eli jokaista yksittäistä tietokantaoperaatiota ei tallenneta levylle. Tällöin tietojen jatkuva päivitys on huomattavasti nopeampaa kuin tietokannoissa, jotka suorittavat kaikki operaatiot levyllä. Samasta syystä vältytään myös relaatiotietokannoille tyypilliseltä haunkäsittely- ja parantamisprosessilta. Koska Redis hyödyntää toiminnassaan välimuistia, vältytään tilanteilta, joissa kiintolevylle viedään paljon väliaikaista tietoa. Tästä johtuen kiintolevyyn kohdistuvien tietokantaoperaatioiden määrä on huomattavasti matalampi kuin toiminnalta raskaammissa relaatiotietokannoissa. (Redis ltd. Introduction to Redis.)

3.1.3 RabbitMQ

AA-ohjaimen on kyettävä hoitamaan myös monia paikallisoperaatioita. Se käsittelee Bluetooth Mesh -verkkosolmujen julkaisemia viestejä, julkaisee verkkoon pyyntöjä ja komentoja yhdyssolmunsa kautta, pitää kirjaa verkon tapahtumista sekä jäsentää, lajittelee ja välittää verkon tapahtumia pilvipalveluihin. Jotta kaikki edellä mainitut tehtävät voidaan hoitaa tehokkaasti ja samanaikaisesti, pitää ohjelmakoodi laajennettavana sekä ylläpidettävänä, ohjelmisto on jaettu toimintokohtaisesti moottoreihin, joiden pienemmät ohjelmakokonaisuudet on edelleen jaettu moduuleihin. Moottorit ovat laajoja kokonaisuuksia ja joutuvat jakamaan tietoa toistensa välillä, mutta toisaalta viestinnän toteutus ei saisi lisätä niiden keskinäistä riippuvuutta. On myös huomioitava, että moottorin kaatuessa, sille lähetettyjä viestejä ei saa menettää, joten viestien täytyy olla tallennettuna muistissa tai kiintolevyllä. Jotta kaikki edellä mainittu kyetään toteuttamaan, viestin lähettäjä on erotettava vastaanottajasta, mikä voidaan saavuttaa käyttämällä viestinnässä erillistä ohjelmistoa.

Nykyään tällaisia ohjelmistoja on saatavilla markkinoilta, eikä sellaisen kehittämiseen tarvitse käyttää resursseja. Tarjolla on myös tehokkaita avoimeen lähdekoodiin perustuvia viestijonoja, joten maksullisten ratkaisujen käyttö ei myöskään ole pakollista.

RabbitMQ on Erlang-kielellä kirjoitettu avoimeen lähdekoodiin ja AMQP 0-9-1 -standardiin perustuva viestijono, jonka ensimmäinen versio julkaistiin vuonna 2007. Lähtökohtana RabbitMQ:n kehityksessä oli luoda avoimeen lähdekoodiin perustuva ja alustariippumaton viestijono kalliiden kaupallisten vaihtoehtojen rinnalle. Avoimen lähdekoodin lisäksi RabbitMQ:ta on helppo käyttää, asentaa ja siihen on saatavilla liitännäisiä, joiden avulla AMQP saadaan toimimaan myös muiden suosittujen viestintäprotokollien kuten MQTT:n ja STOMP:n kanssa. RabbitMQ:ta pidetään myös vakaampana kuin sen kanssa kilpailevat avoimen lähdekoodin ratkaisut. (Videla, Alvaro – Williams, Jason J.W. 2012. RabbitMQ in Action: Distributed Messaging for Everyone.)

3.2 ActiveAhead-ohjainohjelmisto

AA-ohjaimessa on käytetty useita eri teknologioita, joista osaa on jo aiemmin käsiteltykin, mutta säiliötä, joka sisältää itse ohjelmiston, ei vielä ole tarkemmin tuotu esiin. Tämän työn aihe eli AA-ohjaimeen toteutettava taakanjakomoduuli toteutetaan pääosin ohjelmiston sisältävään säiliöön, ja edellä käsiteltyjä Redis- ja RabbitMQ-säiliöitä muokataan konfiguraatiotiedostojen kautta. Tässä luvussa käydään läpi lähdekoodissa käytetty kieli ja muut kehitystyön kannalta oleelliset teknologiat.

3.2.1 JavaScript ja Node.js

AA-ohjainohjelmiston kehityksessä käytetään JavaScript-ohjelmointikieltä. Se on yksisäikeinen kieli, joka tukee niin deklaratiivista kuin imperatiivista ohjelmointiparadigmaa sekä funktionaalista ja olio-ohjelmointia. JavaScript on prototyyppipohjainen ohjelmointikieli, mikä tarkoittaa, että luokat ovat löyhästi määriteltyjä ja niiden ilmentymiin voidaan lisätä ominaisuuksia dynaamisesti. Muuttujien tyypitys tapahtuu JavaScriptissä ajonaikaisesti, mikä tuo kieleen joustavuutta, mutta edellyttää joissain ohjelmakoodin kohdissa tyyppitarkistuksia, koska tyypitykseen liittyviä virheitä ei havaita ohjelman käännösvaiheessa toisin kuin staattisen tyypityksen kanssa. (MDN Web Docs. JavaScript.)

Node.js on näennäisesti rinnakkaisuutta tukeva ajoympäristö, joka on suunniteltu skaalautuvien verkkojen rakentamiseen. Näennäisellä rinnakkaisuudella tarkoitetaan sitä, että operaatioita voidaan suorittaa rinnakkain, muttei silti aidosti yhdenaikaisesti. Edellä mainittu rajoitus ei kuitenkaan sulje pois moniytimisten prosessorien hyödyntämistä. On hyvin harvinaista, että Node.js päätyy estotilaan, mikä helpottaa skaalautuvien verkkojen rakentamista. Tämä johtuu siitä, ettei Node.js sisällä lukkoja, mikä puolestaan on seurausta siitä, että suoria siirto-operaatioita, kuten näppäimistön syötteen lukemista käyttäviä funktioita on erittäin vähän. (OpenJS Foundation. About node.js.)

ActiveAhead-ohjaimen kehityksessä käytetty node.js versio on Node v12.13.0 (LTS).

3.2.2 JSON-skeema ja sen validointi

AA-ohjaimen toiminnan kannalta on tärkeää, että moottorien välillä kulkevien JSON-tietopakettien sisältö on eheä ja sisältää aina kaikki tarvittavat kentät. Vaikka järjestelmä on täysin automatisoitu ja moottorit pystyvät virhetilanteessa käynnistämään itsensä uudelleen, ohjelmiston ajautuessa virhetilaan tietoa saatetaan menettää johtuen vastaanotettavan ja välitettävän tiedon volyymista. Vuorovaikutteiseen kanssakäymiseen liittyvät ongelmat eivät edellytä toimenpiteitä, koska valaisujärjestelmän käyttäjät eivät pysty vaikuttamaan välittömästi ohjaimen toimintaan.

JSON-skeema (*JavaScript Object Notation*) on tiedostomuoto, jonka avulla määritetään JSON-pohjaisen tiedon rakenne, validaatio, käsittely, dokumentaatio ja hyperlinkkien navigaatio. Sen internet-mediatyyppi on *application/schema+json* (Galiegue, Francis – Zyp, Kris – Court, Gary. 2013. JSON Schema: Core definitions and terminology).

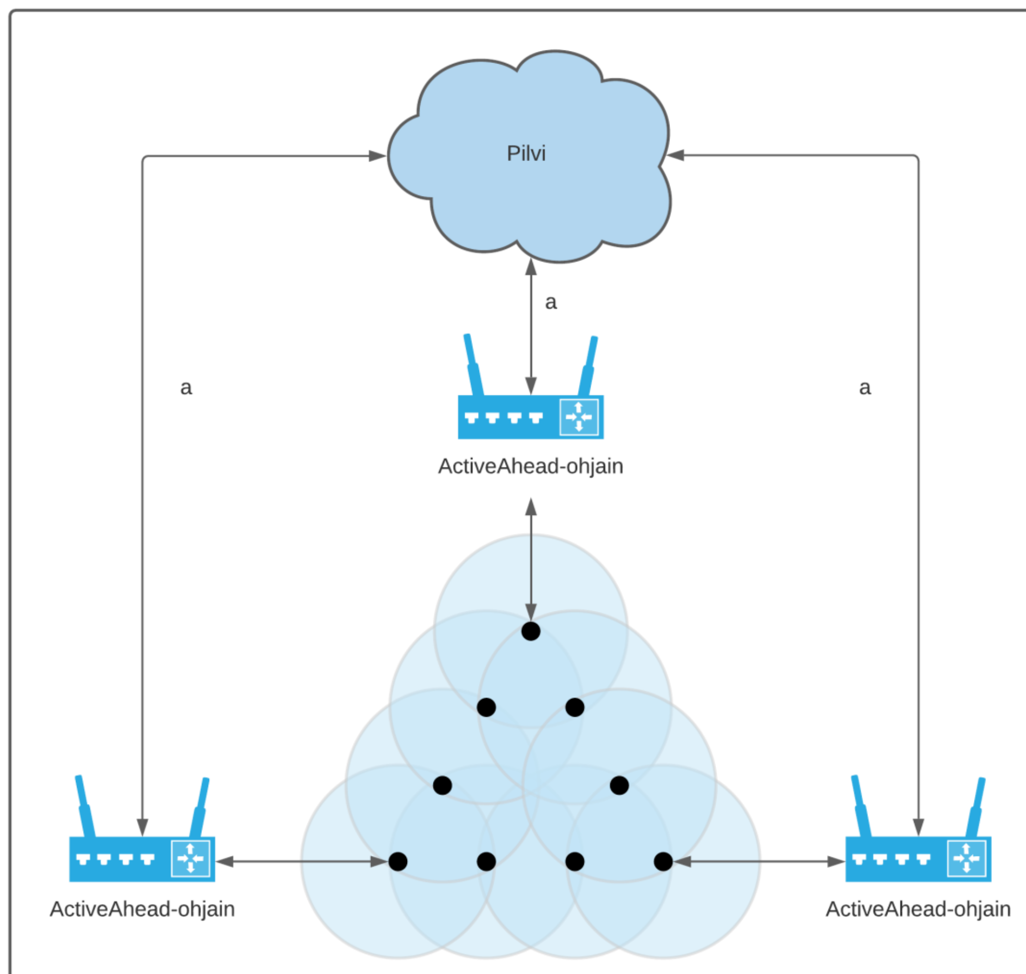
Ilmentymän validaatio on yksi JSON-skeeman monista käyttötarkoituksista ja sen avulla voidaan varmistaa, että JSON-ilmentymä täyttää ennalta määrätyt edellytykset ja sisältää tarvittavat tiedot (Galiegue, Francis – Zyp, Kris – Court, Gary. 2013. Json Schema: Interactive and non interactive validation). Validoimalla ilmentymä pyritään siis varmentamaan, että vastaanotetun JSON-viestin sisältö täyttää vastaanottavan järjestelmän sisältövaatimukset. Validaatioprosessi voi olla joko vuorovaikutteinen, eli luetaan käyttöliittymästä käyttäjän täyttämä lomake, tai vuorovaikutukseton, eli varmistetaan vastaanotetun JSON-ilmentymän sisältävän ennalta määrätyt tiedot (Galiegue, Francis ym. 2013).

JSON-tiedon rakenteen määrittäminen ja vahvistaminen on ohjelmiston kannalta tärkeää, koska tällöin vältytään tilanteilta, joissa järjestelmä odottaa tietyn kentän sisältämää tietoa ja päätyy ongelmatilanteeseen, kun vastaanotettu

JSON-ilmentymä ei sisälläkään kyseistä kenttää. JSON-skeeman avulla pystytään myös määrittämään kenttien tietotyypit, mikä vähentää vuorovaikutteisen tiedon käsittelyyn liittyviä ongelmia kuten väärin täytetyt kaavakkeet tai injektioyritykset.

4 ActiveAhead-ohjaimen toiminnallisuus

AA-ohjain ohjaa Bluetooth Mesh -verkossa olevien valaisinten toimintaa, tarkkailee verkon tapahtumia ja kuntoa sekä välittää verkossa olevien laitteiden tilapäivityksiä ja anturien keräämää tietoa eteenpäin Helvarin pilvipalveluihin. Kun ohjain otetaan käyttöön tilassa, jossa on olemassa AA-valaistusjärjestelmä, ensimmäisenä laite muodostaa yksi-yhteen-yhteyden lähimpään yhdyssolmuun, minkä jälkeen se alkaa etsimään mesh-verkon solmuja ja niihin liitettyjä laitteita. Löytäessään uuden solmun ohjain yrittää saada solmun omistukseensa. Ohjain kuuntelee omistamiinsa solmuihin liitettyjen laitteiden viestejä, ja välittää niitä pilvipalveluihin. Toisekseen ohjain tarkkailee kyseisten laitteiden kuntoa ja yhteyden laatua. Tarpeen vaatiessa ohjain lähettää pilveen varoituksen, mikäli se ei ole saanut viestejä omistamaltaan solmulta tai solmuun liitetyltä laitteelta, jolloin loppukäyttäjä tai tukihenkilö näkee ilmoituksen viallisesta laitteesta.



Kuva 4. Ohjaintenvälinen yhteydenpito ympäristössä, jossa on monta ohjainta

Helvarin pilvipalvelut vastaavat ohjainten yhteistoiminnan koordinaatiosta tiloissa, joissa mesh-verkon laajuus edellyttää useamman ohjaimen samanaikaista käyttöä. Kuvassa 4 ohjainten välinen viestintä kulkee 'a'-viivojen kautta. Kun yhdessä paikassa on useita ohjaimia, solmun omistajaksi määräytyy se ohjain, joka solmun ensimmäisenä havaitsee riippumatta etäisyydestä tai yhteyden laadusta. Solmun omistajuus voi muuttua, mikäli yhteys solmun omistavaan ohjaimeen katkeaa. Tämä saattaa tapahtua esimerkiksi tilanteessa, jossa yhteys solmun ja omistavan ohjaimen välillä on heikko, mikä saattaa johtua etäisyydestä, rakennuksen pohjapiirustuksesta tai rakennusmateriaaleista.

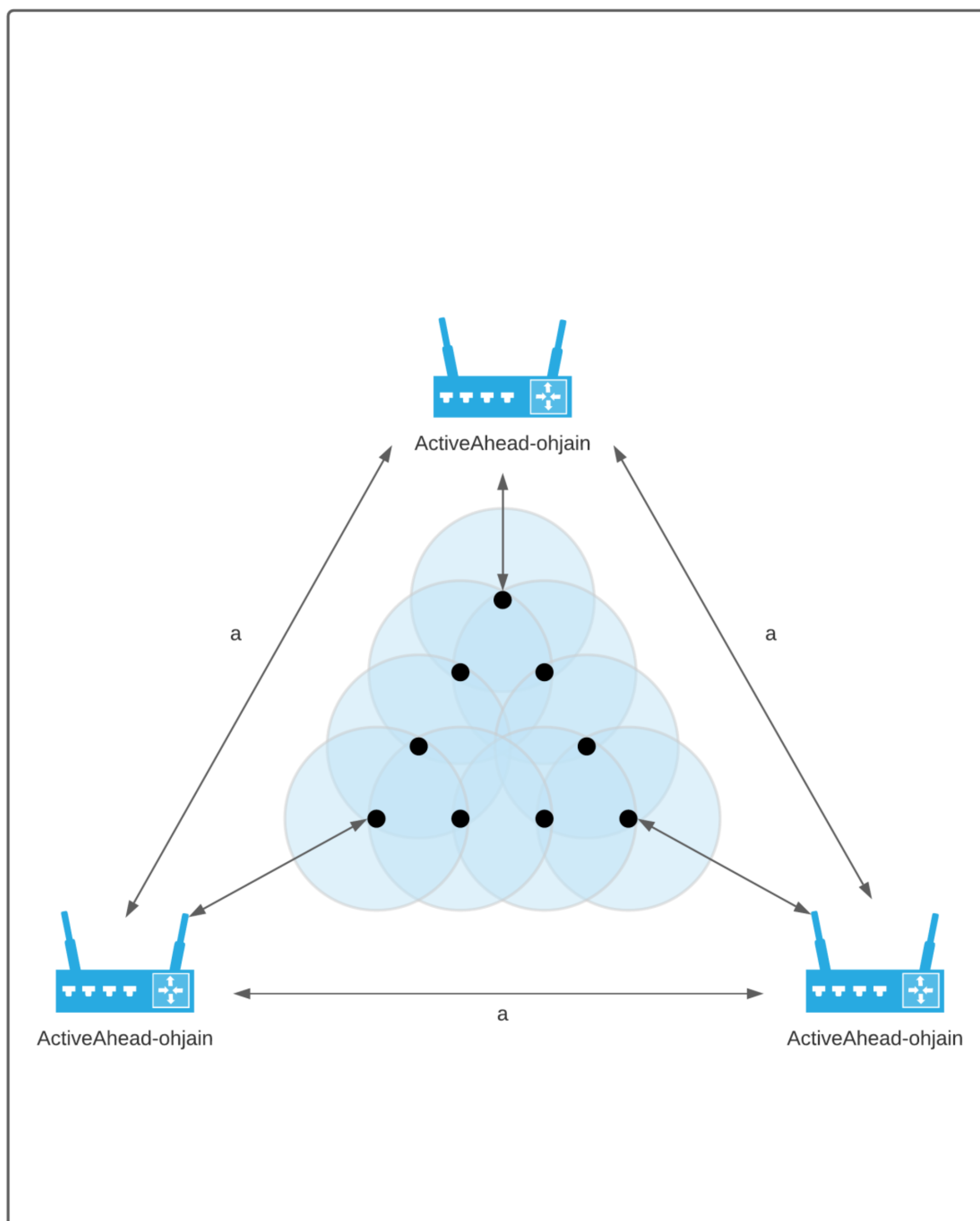
Nykyisen toimintamallin heikkous on, että se on täysin riippuvainen pilvipalveluiden saavutettavuudesta, mikä tarkoittaa, että kun internetyhteydet menetetään, ohjaimet menettävät kykynsä toimia koordinoidusti, ja edellä mainitussa tilanteessa solmu, johon sen omistava ohjain ei saa yhteyttä, ei siirry toisen ohjaimen omistukseen. Järjestelmän riippuvuutta pilvestä voidaan vähentää antamalla ohjaimille kyky koordinoida yhteistyötä pakallisesti. Kehittämällä ohjainten paikallista toiminnallisuutta järjestelmästä tulee itsenäisempi ja edukkaampi, koska ohjainten koordinaatioon liittyvät toiminnot eivät enää kohdenna tietovirtaa pilvipalveluihin.

Tämän opinnäytetyön aiheena olevalla taakanjakomoduulilla pyritään vastaamaan pääasiassa kolmeen haasteeseen: miten ohjainten välinen viestintä tulisi toteuttaa, miten ohjaimen toimintaa tiedon kerääjänä pystytään tehostamaan ja millä periaatteilla taakanjakoa tulisi toteuttaa.

4.1 AA-ohjainten välinen viestintä

AA-ohjainten välistä viestintää toteutettaessa ensimmäisenä nousee esille, minkä verkkoteknologian avulla se toteutetaan. AA-ohjainlaitteessa, SolidSense N6, käytettävissä olevat verkkoteknologiat ovat Ethernet, WiFi ja Bluetooth, joista kaksi ensin mainittua ovat lähiverkkoratkaisuja ja kaksi jälkimmäistä ovat langattomia verkkoratkaisuja. Vaikka AA-tuoteperheen perustavana ajatuksena on järjestelmän langattomuus, on siitä huolimatta otettava huomioon, että langattomien yhteyksien luonteesta johtuen ne eivät välttämättä ole paras mahdollinen valinta, kun edellytetään viestinnän varmuutta. Esimerkiksi WiFi-verkkoa asennettaessa yhteyden laatu voi olla erittäin hyvä, mutta muutokset rakennuksen pohjapiirroksessa tai jopa kalustuksessa saattavat heikentää verkkosignaalia, jolloin datapaketit eivät välttämättä saavuta vastaanottajaa. Viestinnän luotettavuutta langattomassa verkossa voidaan parantaa ohjelmallisesti, mutta luotettavuus ei siitä huolimatta koskaan saavuta langallisen yhteyden tasoa. Langallisissa yhteyksissä voidaan myös tarpeen vaatiessa käyttää häiriösuojattuja kaapeleita.

Verkkoteknologiaa valitessa on huomioitava, että AA-ohjaimet joutuvat ilmoittamaan solmujen omistussuhteiden muutoksista Helvarin pilvipalveluihin, minkä takia viestinnän luotettavuus on erittäin tärkeää. Halutaan välttää tilanne, jossa pilvipalveluihin tallennetut solmujen omistussuhteet eivät vastaisi paikallista järjestelmää. Koska paikallisen yhteyden laadulla on niin suuri vaikutus järjestelmän toimintaan, langallinen Ethernet on nykytilassa ylivoimaisesti järkevin. Ethernetillä on luotettavuuden lisäksi myös toinen etu langattomiin vaihtoehtoihin nähden: AA-ohjaimessa jo entuudestaan käytössä olevaa RabbitMQ:ta ei suositella käytettäväksi langattomissa verkoissa, koska sitä ei ole suunniteltu toimimaan langattomassa ympäristössä, minkä takia heikkolaatuisella yhteydellä saattaa olla arvaamattomia vaikutuksia. Koska AA-ohjainten välinen viestintä hoidetaan langallisesti, niiden välisessä viestinnässä voidaan käyttää RabbitMQ:ta, eikä siitä johtuen ole tarpeellista kehittää omaa viestijonoa.



Kuva 5. Ohjainten välinen paikallinen viestintä

Ohjainten välinen viestintä toteutetaan siis kuvan 5 mukaisesti. Tässä kuvassa ’a’-viivalla merkitään laitteiden välistä Ethernet-yhteyttä. Ohjaimet on kiinnitetty tavalliseen verkkokytkimeen, jonka kautta viestintä tapahtuu.

Bluetoothin käyttö ohjainten välisessä viestinnässä olisi kiinnostava vaihtoehto, mutta se ei ole tällä hetkellä mahdollista, koska AA-valaisujärjestelmässä käytettävää Bluetooth Mesh -verkkoa ei ole vielä optimoitu kuljettamaan viestejä luotettavasti verkon ulkopuolisten laitteiden välillä. Mesh-verkon kehittyessä on mahdollista, että tulevaisuudessa Bluetoothratkaisu syrjäyttää Ethernetin.

4.2 Tiedon kerääminen

Tilojen käyttötiedon kerääminen ja välittäminen Helvarin pilvipalveluihin on oleellinen osa ActiveAhead Generation 2 -järjestelmän toimintaa, minkä takia on pyrittävä siihen, että mahdollisimman moni Bluetooth Mesh -verkossa toimivien laitteiden viesteistä saadaan vastaanotettua ja välitettyä eteenpäin. Pohjimmiltaan taakanjaolla pyritään tehostamaan tiedon keräämistä mesh-verkosta. Ensinnäkin tavoitteeseen pyritään ehkäisemällä tilanteita, joissa yksi ohjain vastaisi kolmen neljäsosan solmujen viestinnästä, vaikka samassa tilassa olisi läsnä neljä muuta ohjainta. Toisekseen pyritään siihen, että ohjaimet omistaisivat vain sellaisia solmuja, joihin niillä on hyvä yhteys.

4.3 Taakanjakomoduulin ominaisuudet

Taakanjakomoduulilta odotetaan, että ohjain osaa määrittää itselleen sopivan määrän solmuja ja pyrkii luovuttamaan solmuja toisille ohjaimille, mikäli taakka on liian suuri. Nykyisellään ohjain säilyttää tietokannassaan kaikkien sen kantamalla olevien solmujen tietoja, joten tietokannassa olevien laitteiden määrästä voidaan päätellä suhteellinen osuus, joka ohjaimella kuuluisi olla. Tämän lisäksi voidaan asettaa kattoarvo, jota ohjaimen ei tulisi koskaan ylittää. Jos ohjaimen suhteellinen osuus taakasta on liian matala, se voisi ilmoittaa muille ohjaimille olevansa valmis ottamaan vastaan lisää laitteita. Toisaalta, jos suhteellinen osuus taakasta olisi liian korkea, ohjain voisi alkaa tarjoamaan solmuja muille ohjaimille.

Perustana sille, mitä solmuja ohjain tarjoaa muille, voidaan pitää sitä, kuinka kaukana solmu on ohjaimesta eli hyödynnetään solmuilta vastaanotettujen viestien TTL-kenttää – mitä suurempi kentän arvo sitä lähempänä solmu on ohjainta. On kuitenkin otettava huomioon, että Bluetooth Mesh -verkossa viestit eivät aina saavu perille samaa reittiä, mikä tarkoittaa sitä, että samalta solmulta vastaanotettujen viestien TTL-kentän arvo saattaa vaihdella viestikohtaisesti. Tästä syystä ohjaimen tietokantaan voidaan lisätä kenttiä erilaisille matemaattisille keskiluvuille kuten TTL-kentän moodi tai keskiarvo viimeisten 50 viestin ajalta. Toisaalta voidaan myös mitata sitä, kuinka usein ohjain saa solmulta viestejä, eli tietokantaan lisätään frekvenssikenttä, josta nähdään, kuinka pitkä aika viimeisen 50 viestin vastaanottamiseen on kulunut.

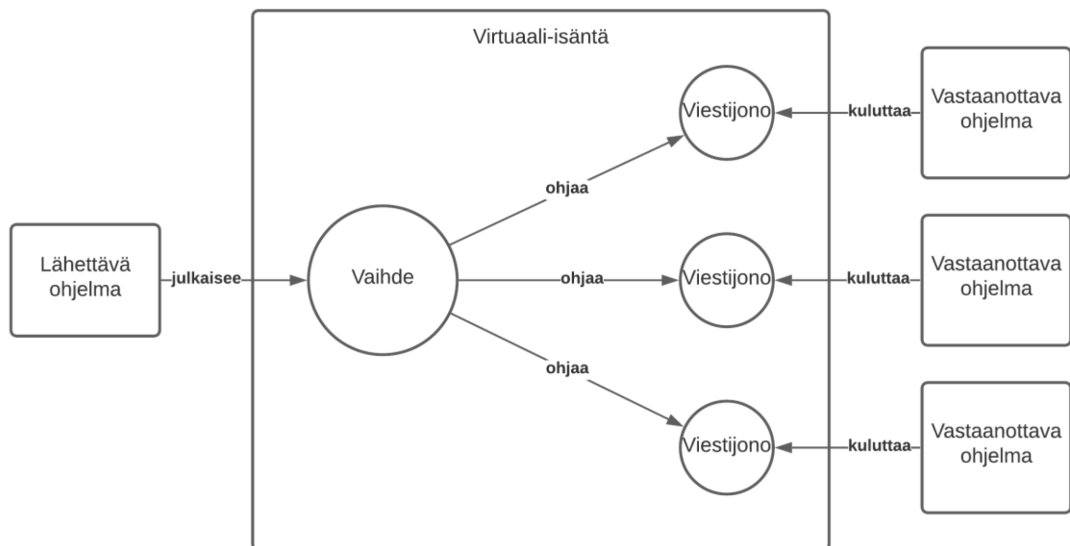
Kun ohjain päättää luovuttaa solmuja pois, se voi verrata edellä mainittuja arvoja ja päätellä siten, mitkä solmut ovat haluttuja ja mitkä eivät. Samaten ohjaimen havaitessaan taakkansa olevan liian matala, se pyytää toisilta ohjaimilta sellaisia solmuja, joihin sillä on hyvä yhteys. Tällaisen pyynnön ohessa ohjain voisi samalla välittää omat yhteyden laatua mittaavat parametrit, jolloin vastaanottaja voi verrata niitä omiin vastaaviin arvoihin ja päättää, kannattaako solmua luovuttaa toiselle.

5 AA-ohjainten väliset yhteydet

Jotta minkäänlaista taakanjakoa pystyttäisiin toteuttamaan, ohjainten on kyettävä automaattisesti luomaan yhteys toisiinsa ja kyettävä ratkaisemaan yhteyshäiriöihin liittyvät virheet. Ethernet-yhteyden muodostaminen on jaettu kahteen vaiheeseen, joista ensimmäiseen sisältyy ohjainten kyky tulla tietoisiksi toisistaan. Ensimmäistä vaihetta varten luodaan uusi Ethernet-moottori, joka lähettää ja vastaanottaa UDP-ryhmälähetyksiä, jotka ovat moottorin tasaisin väliajoin lähettämiä sydämenlyöntejä. Toisessa vaiheessa UDP-viestin vastaanottanut ohjain muodostaa yhteyden lähettäjän RabbitMQ-palvelimeen, jonka kautta lopullinen ohjaintenvälinen viestintä toteutuu. Nämä edellä mainitut vaiheet käydään läpi tarkemmin RabbitMQ:n konfiguraatiotiedostoa käsittelevän luvun jälkeen aliluvuissa 5.2 ja 5.3.

5.1 RabbitMQ-konfiguraatiotiedosto

Uuden Ethernet-moottorin yläluokka on Helvarin kehittämä *Engine*-luokka, joka vastaa kaikille moottoreille yhteisistä toimenpiteistä, kuten käynnistys, konfiguraatio ja lopetus. Jotta uusi Ethernet-moottori pystyy keskustelemaan toisten AA-ohjaimen moottorien kanssa, on välttämätöntä tehdä lisäyksiä moottorien välisestä viestinnästä vastaavan RabbitMQ:n konfiguraatiotiedostoon.



Kuva 6. Viestin kulku RabbitMQ-palvelimella

Jotta konfiguraatiotiedoston sisältöä olisi helpompi ymmärtää, kuvassa 6 on käyty tämän opinnäytetyön osalta tärkeimmät RabbitMQ-palvelimen osat, ja niiden tehtävät viestin perille ohjaamiseen. RabbitMQ-palvelin sisältää oletusvirtuaali-isännän (*default vhost*), jonka lisäksi palvelimelle voidaan luoda tarpeen mukaan uusia virtuaali-isäntiä (*vhost*). Jokainen virtuaali-isäntä sisältää oletusvaihteen (*default exchange*), jonka lisäksi voidaan luoda uusia vaihteita (*exchange*). Vaihteiden lisäksi virtuaali-isäntä sisältää viestijonoja. Vaihteen tehtävä on ohjata vastaanottamansa viestit eteenpäin oikeisiin jonoihin, mikä tapahtuu sidosten (*bindings*), ja niiden tunnistuksessa käytettävien reititysavainten (*routing keys*) avulla. Kun ohjelma haluaa lähettää viestin RabbitMQ-palveli-

men kautta, lähetyksen yhteydessä se nimeää käytettävän vaihteen sekä reititysavaimen, jonka avulla tiedetään, minkä sidonnan sääntöjä noudatetaan. Vaihde lukee viestin ohessa annetun reititysavaimen ja ohjaa viestin määränpäänä olevaan viestijonoon. Vaihteiden ja viestijonojen välisten yhteyksien määrittämisessä käytetään sidoksia, jotka ovat ikään kuin viestinvälityksessä käytettäviä sääntöjä. Sidosta luotaessa määritetään se, mistä virtuaali-isännän vaihteesta viestit ohjataan mihinkin virtuaali-isännän viestijonoon. Toisin sanoen sidoksessa määritetään alkuperänä oleva vaihde, kohteena olevan viestijono ja reititysavain, jonka avulla vaihde tunnistaa, mitä sidosta eli sääntöä minkäkin viestin yhteydessä tulee soveltaa.

```
{
  “rabbit_version”: 3.8,
  “vhosts”: [
     {
       “name”: “/sisäinen”,
     },
     {
       “name”: “/ulkoinen”,
     },
  ]
  ”queues”: [
     {
       “name”: “ethernet-moottori”,
       “vhost”: “/sisäinen”,
       “durable”: true,
       “auto_delete”: false,
       “arguments”: {},
```

```
    },
  ],
  "exchanges": [
    {
      "name": "sisäinen",
      "vhost": "/sisäinen",
      "type": "direct",
      "durable": true,
      "auto_delete": false,
      "arguments": {},
    },
  ],
  "bindings": [
     {
       "source": "sisäinen",
       "vhost": "/sisäinen",
       "destination": "ethernet-moottori",
       "destination_type": "queue",
       "routing_key": "ethernet-moottori",
       "arguments": {},
     }
}
```

Esimerkkikoodi 1. Palvelimen asetukset RabbitMQ-konfiguraatiotiedostossa

RabbitMQ-palvelimen asetuksia voidaan muuttaa sen ollessa käynnissä käyttäen palvelimen omaa komentokehotetta. Toisaalta asetukset voidaan lukea myös palvelimen käynnistyksen yhteydessä erillisestä konfiguraatiotiedostosta (esimerkkikoodi 1). AA-ohjaimessa RabbitMQ ajetaan Docker-säiliössä, minkä takia asetukset on määritettävä tiedostossa. Komentokehotteella tehdyt muutokset menetettäisiin, jos säiliö joudutaan käynnistämään uudestaan esimerkiksi sähkökatkoksen takia.

RabbitMQ-konfiguraatiotiedostossa voidaan määrittää erilaisista käyttötarkoituksista vastaavia virtuaali-isäntiä, jotka nimetään tiedoston *vhosts*-kohdassa (ks. esimerkkikoodi 1). Virtuaali-isännät ovat loogisia kokonaisuuksia, joilla on muun muassa omat käyttäjät, viestijonot, vaihteet ja sidokset. Niiden nimet aloitetaan etukenoviivalla ”/<isännänNimi>”. RabbitMQ:n oletusisännästä käytetään pelkkää etukenoviivaa, eikä sitä tarvitse erikseen nimetä konfiguraatiotiedostossa. Virtuaali-isäntiä käyttämällä voidaan erottaa esimerkiksi kahden eri ohjelman viestintä toisistaan tai laitteen sisäinen viestintä ulkoisesta kuten esimerkkikoodissa 1 on tehty.

Seuraavana esimerkkikoodissa 1 nähdään, kuinka *queues*-listassa luodaan Ethernet-moottorin käyttöön uusi viestijono, jolle annetaan nimi ja virtuaali-isäntä, jonka alaisuuteen jono luodaan. Avainsana *durable* on boolean-tyyppinen muuttuja, ja se määrittää, säilytetäänkö viestit RabbitMQ:n uudelleen käynnistyessä. Tässä käytetään arvoa *true*, joka tarkoittaa, että viestijonon metadata sekä *persistent*-asetusta käyttäen julkaistut viestit säilötään kiintolevylle välimuistin sijaan, minkä ansiosta palvelimen käynnistyessä uudelleen tietoa menetetään minimaalisesti. Avainsana *auto_delete* arvo on niin ikään boolean-tyyppinen, ja *false*-arvo tarkoittaa, ettei viestijonoa tuhota, vaikkei siihen olisi liitetty vastaanottajaa.

Esimerkkikoodissa 1 *exchanges*-listassa luodaan vaihteet palvelimen virtuaali-isännille. Vaihdetta luodessa on ilmaistava *type*-ominaisuuden kohdalla, minkälaisesta vaihteesta on kyse. Tässä tapauksessa vaihteen tyypiksi asetetaan *direct*, mikä tarkoittaa, että vaihteeseen julkaistut viestit ohjataan viestijonoihin reititysavainten (*routingkey*) perusteella. Muut vaihteen ominaisuudet toimivat samalla periaatteella kuin viestijonon yhteydessä.

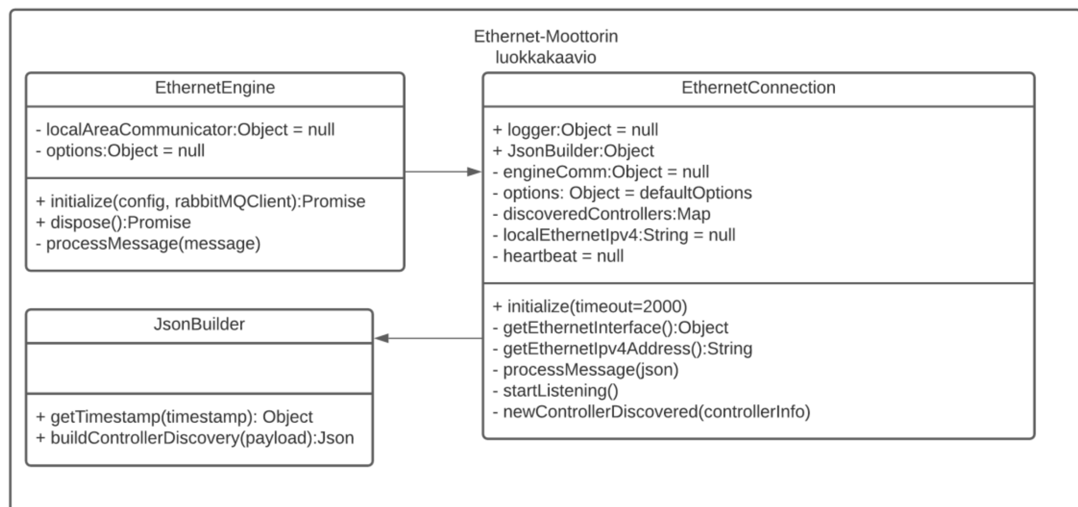
Viimeisenä konfiguraatiotiedostossa luodaan sidokset viestijonojen ja vaihteiden välille, toisin sanoen viestijono yhdistetään vaihteeseen. Sidoksen *source*-ominaisuus määrittää jonoon päätyvien viestien alkuperän eli vaihteen. *Destination*-ominaisuus puolestaan määrittää kohteen, johon vaihteeseen julkaistut viestit päätyvät. Ominaisuus *destination_type* on tärkeä, koska vaikka useimmiten sidoksen kohteena on viestijono, RabbitMQ:ssa on myös mahdollista yhdistää vaihteita toisiinsa. Tässä yhteydessä sidokselle annetaan myös oma reititysavain, jota käyttämällä voidaan lähettää viestejä suoraan kyseiseen jonoon.

5.2 Ethernet-moottori ja UDP

Ethernet-moottorin tarkoitus saada AA-ohjaimet tietoisiksi toistensa olemassaolosta. Tämä toteutetaan Ethernet-moottorin tasaisin aikavälein lähettäminä sydämenlyöntiviesteinä, joita toisten ohjainten Ethernet-moottorit kuuntelevat.

Vastaanotettujen viestien lähettäjistä pidetään kirjaa, ja mikäli kyseessä on entuudestaan tuntematon ohjain, Ethernet-moottori tiedottaa monitoring-engine -moottorissa sijaitsevalle taakanjakomoduulille uudesta ohjainlaitteesta. Ethernet-moottorin toiminta on kokonaisuuden kannalta tärkeä, koska sen keräämän tiedon pohjalta luodaan myöhemmässä vaiheessa ohjaintenväliset yhteydet ohjainten RabbitMQ-palvelimien välille.

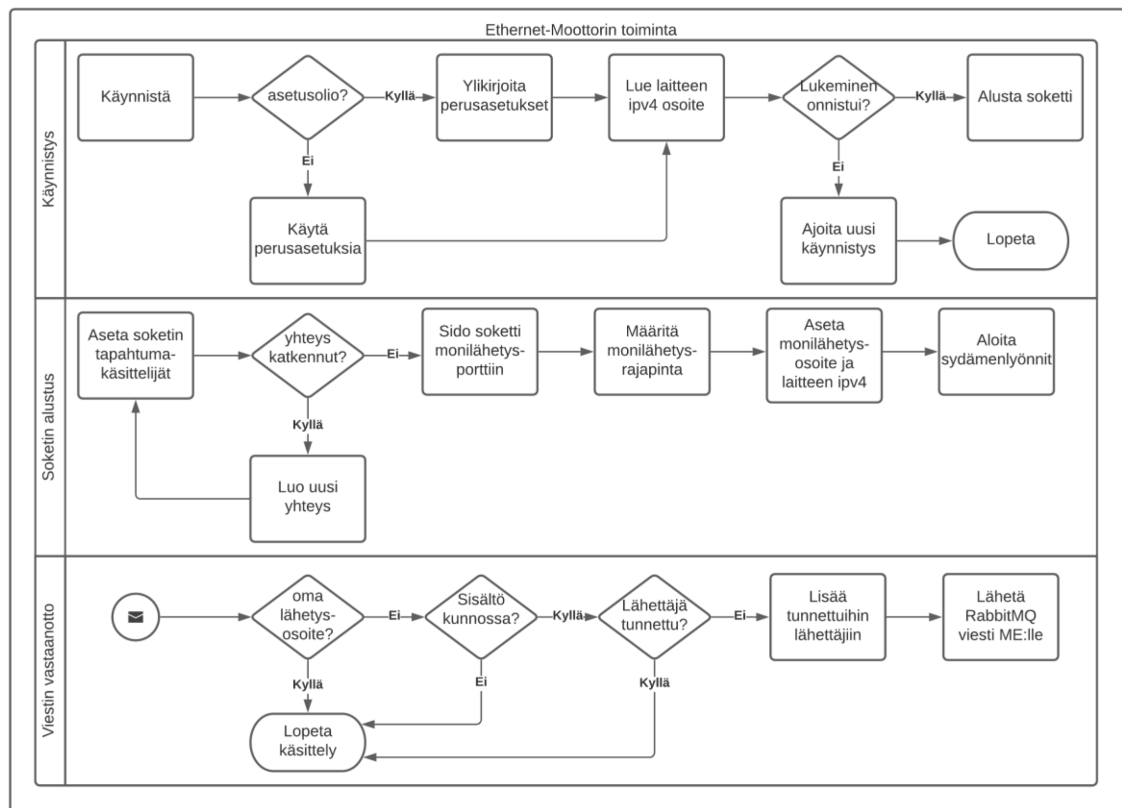
Ethernet-moottorin pääluokka, EthernetEngine käynnistetään index.js-tiedostossa, jossa ei tehdä muuta kuin luodaan uusi instanssi EthernetEnginestä, ja kutsutaan sen Engine-yläluokassa määriteltyä start-funktiota. Kyseinen funktio lukee ActiveAhead-ohjaimen oman konfiguraatiotiedoston sekä luo moottorikohtaisen lokitiedostoja kirjoittavan ohjelman, yhdistää moottorin RabbitMQ:n kautta muihin moottoreihin, ja kutsuu johdetun luokan, tässä tapauksessa EthernetEnginen initialize-funktiota, jolle se välittää kyseistä moottoria vastaavan pätkän ohjaimen omasta konfiguraatiotiedostosta.



Kuva 7. Ethernet-moottorin luokkakaavio

Ethernet-moottori koostuu kahdesta luokasta, EthernetEnginestä ja EthernetConnectionista. Kuvassa 7 näkyvä JsonBuider on enemmänkin kirjasto, jonka sisältämillä funktioilla varmistetaan, että moottorien välillä kulkevat viestit ovat rakenteeltaan oikeanlaisia. EthernetEngine-luokka on moottorin pääluokka,

joka vastaa moottorien välillä kulkevien viestien käsittelystä, moottorin käynnistyksestä ja sammutuksesta. EthernetConnection-luokka vastaa UDP-soketin luonnista, UPD-monilähetysten käsittelystä ja sydämenlyöntien lähetyksestä. Sydämenlyöntiviestien ansiosta AA-ohjaimet pystyvät löytämään toisensa. Viestit sisältävät tiedot lähettäjän sarjanumerosta ja ohjelmiston versiosta. Tämän lisäksi viestistä voidaan lukea lähettäjän ipv4-osoite, jota voidaan käyttää, kun halutaan muodostaa asiakasyhteys lähettäjän RabbitMQ-palvelimeen.



Kuva 8. Ethernet-moottorin pääoperaatiot: käynnistys, soketin alustus ja viestin vastaanotto

Ethernet-moottoria käynnistäessä sille välitetään asetusolio, joka sisältää monilähetysosoitteen ja sydämenlyöntien aikavälin millisekunneissa. Mikäli asetusoliota ei välitetä, moottori käyttää siihen ohjelmoituja oletusasetuksia. Kun EthernetEngine-luokka on käynnistynyt, se käynnistää EthernetConnection-luokan, joka yrittää lukea ohjaimen Ethernet-rajapinnan ipv4-soitteen. Mikäli tämä ei onnistu, ajoitetaan uusi yritys.

```
initialize(timeout=2000) {
  this.localEthernetIpv4 = this._getEthernetIpv4Address();
  if (!this.localEthernetIpv4 || !this.options.controllerInfo) {
     this.emit(‘error’, ‘Failed to initialize connection.’);

     setTimeout(() => {
       this.initialize(timeout * 2);
     }, timeout);
     return;
  }
  this._startListening();
}
```

Esimerkkikoodi 2. EthernetConnection-luokan käynnistysfunktio

Jos ipv4-osoitteen lukeminen ei syystä tai toisesta onnistu, uusien yritysten välistä aikaa kasvatetaan kertoimella 2, kuten esimerkkikoodssai 2 *setTimeout*-funktiokutsun yhteydessä näkyy: uuden kutsun ajastuksessa käytetään *initialize*-funktion *timeout*-parametria, jonka oletusarvo on 2 000 millisekuntia. Ensimmäisen kutsun yhteydessä funktiolle ei välitetä parametria, joten virhetilanteessa seuraava kutsu tehdään oletusarvon määräämän ajan kuluttua. Seuraavan kutsun parametri on tällöin 4 000 ms (2000 * 2), ja kolmannen 8000 ms (4000 * 2).

Jos *timeout*-parametri olisi vakioarvo, ajauduttaisiin ongelmiin, mikäli Ethernet-rajapinnan lukeminen epäonnistuisi ohjelmiston ulkopuolisista syistä, kuten laiteviasta tai vaurioituneesta kaapelista. Tällaisessa tapauksessa uudet yritykset käyttäisivät ohjaimen resursseja turhaan, koska yhteyttä ei kuitenkaan voitaisi muodostaa, ennen kuin laitevika on korjattu. Kasvattamalla aikaväliä eksponentiaalisesti taataan se, että normaaleissa olosuhteissa ohjain pystyy itse korjaamaan yhteyden, jos se katkeaa, muttei kuitenkaan tuhlaa resursseja, mikäli virhe aiheutuu ulkoisista syistä.

Kuvan 8 keskimmäisellä kaistalla, ’soketin alustus’, kuvataan UDP-soketin luonnin tärkeimmät vaiheet. Ensimmäiseksi asetetaan soketin tapahtumakäsittelijät. Kaiken kaikkiaan soketilla on neljä tapahtumaa, jotka ovat *error*, *listening*, *close* ja *message*. *Listening*-tapahtuma ilmoittaa, kun yhteys on valmiskäytettäväksi ja *error* puolestaan kirjaa lokiin käytönaikaiset virheet. *Close*-käsittelijä kattaa tilan-

teen, jossa yhteys katkeaa ja sen tarkoitus on pyrkiä avaamaan yhteys uudestaan heti sen katkettua. *Message*-käsittelijä määrittää sen, mitä vastaanotetulle viestille tehdään, eli tässä tapauksessa viesti välitetään seuraavassa alaluvussa käsiteltävään RabbitMQ-yhteyksistä ja niiden luonnista vastaavaan ohjelmanosaan.

Soketin alustuksen viimeisenä askeleena käynnistetään ohjaimen sydämen lyönnit, joita se lähettää joko oletusmonilähetysosoitteeseen tai EthernetConnection-luokan konstruktorille välitetyn asetusolion määräämään monilähetysosoitteeseen. Sydämenlyöntiviesti sisältää ohjaimen oman sarjanumeron sekä ohjelmiston version, ja se lähetetään määrätyin väliajoin, jotta ohjaimet löytävät toisensa.

Message-tapahtumakäsittelijän toiminta on kuvattu kuvassa 8 kohdassa viestin vastaanotto. Ohjaimen ei tule reagoida omiin sydämenlyönteihinsä, joten sen omasta ipv4-osoitteesta lähetettyjä viestejä ei käsitellä. Vastaavasti entuudestaan tunnettujen ohjainten viestejä ei myöskään käsitellä, koska monilähetystä käytetään ainoastaan siihen, että ohjaimet tulevat tietoisiksi toisistaan. Ohjainten välinen viestintä tapahtuu RabbitMQ:n kautta, ja nämä yhteydet luodaan sydämenlyöntiviesteistä kerättyjen tietojen pohjalta.

5.3 Taakanjakomoduulin viestintä ja RabbitMQ-asiakasohjelma

Taakanjakomoduuli vastaa AA-ohjainten välisestä taakanjaosta. Se määrittää, mitkä solmut voidaan luovuttaa ja mitä solmuja voidaan vastaanottaa toisilta ohjaimilta. Moduuli on liitetty osaksi MonitoringEngine-moottoria. Kun MonitoringEngine ottaa vastaan RabbitMQ-viestin Ethernet-moottorilta, viesti käy läpi validaatio-prosessin, jossa varmistetaan, että se sisältää JSON-skeemassa esitellyt tiedot. Mikäli vastaanotetun viestin rakenne ei vastaa ennalta määrättyä skeemaa, se hylätään ja käsittely lopetetaan. Validaatioprosessin läpäissyt viesti ohjataan eteenpäin taakanjakomoduulissa käytettävälle AmqpConnectionMana-

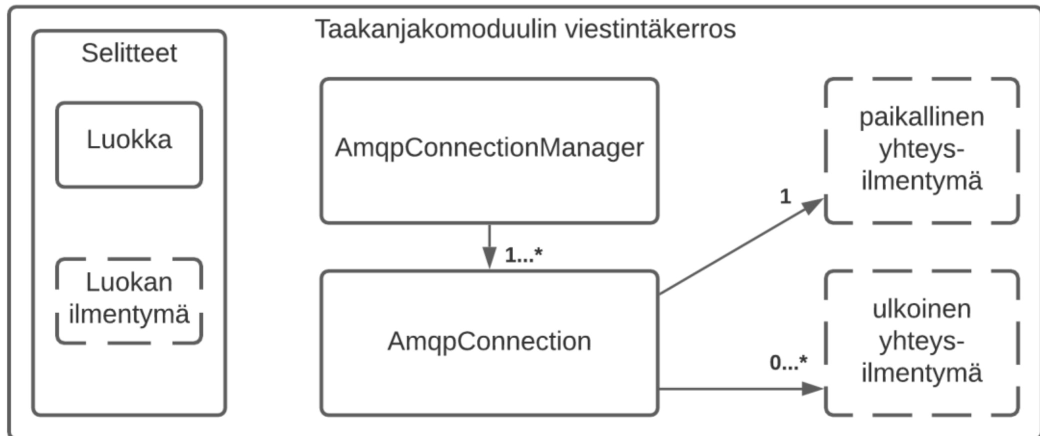
ger-luokalle, joka on vastuussa uusien AmqpConnection-yhteysolioiden luonnista, niiden viestinnästä ja viestien julkaisemisesta paikallisen RabbitMQ-palvelimen vaihteeseen.

```
{
  "rabbit_version": "3.8",
  "users": [
    {
      "name": "esimerkki_käyttäjä",
      "password_hash": "<salasanan_hash>",
      "hashing_algorithm":"rabbit_password_hashing_sha256",
      "tags": "administrator"
    }
  ],
  "permissions": [
    {
      "user": "esimerkki_käyttäjä",
      "vhost": "/ulkoinen",
      "configure": ".*",
      "write": ".*",
      "read":".*"
    },
  ],
}
```

Esimerkkikoodi 3. RabbitMQ-konfiguraatiotiedoston lisäykset

Jotta taakanjako voidaan toteuttaa ohjelmatasolla, on RabbitMQ-palvelimen konfiguraatiotiedostoon tehtävä vielä esimerkkikoodin 3 lisäykset. *Users*-listaan on lisättävä uusi käyttäjä, koska palvelimen oletuskäyttäjää, *guest* ei voida käyttää ulkoisten yhteyksien hallinnassa (eikä oletuskäyttäjää muutenkaan tulisi käyttää tuotannossa turvallisuussyistä). Huomattavaa on myös, että tässä esimerkissä käyttäjän *tags*-kentän arvo on *administrator*. Tätä sovelletaan kehityksen aikana, mutta tuotantoon siirryttäessä käyttäjän oikeudet lasketaan tehtävän edellyttämälle tasolle.

Lisäksi uudelle käyttäjälle on annettava oikeudet virtuaali-isäntään, joka vastaa ulkoisesta viestinnästä, eli *vhost*-kenttään annetaan isännän nimi, minkä jälkeen voidaan vielä tarkentaa käyttäjän oikeuksia isäntään.



Kuva 9. Taakanjakomoduulin viestintäkerros

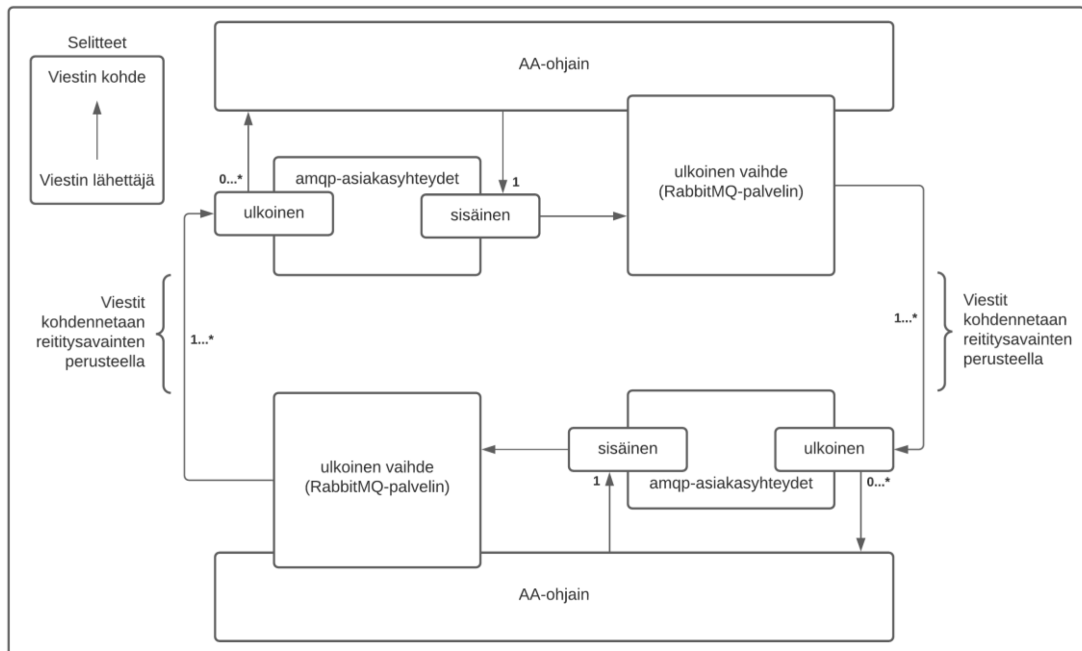
AmqpConnectionManager-luokan ilmentymä luodaan MonitoringEnginen käynnistyksen yhteydessä, minkä jälkeen ilmentymää ohjeistetaan luomaan yhteys AA-ohjaimen omaan RabbitMQ-palveluun (ks. kuva 9 paikallinen yhteysilmentymä). AmqpConnectionManager-luokka sisältää siis aina vähintään yhden ilmentymän AmpqConnection-luokasta, joka sisältää amqp-asiakasohjelman. Paikalliseen RabbitMQ-palvelimeen yhteydessä olevaa ilmentymää käytetään viestien julkaisussa ja reitityksessä toisille ohjaimille. Ulkoiset yhteysilmentymät luodaan Ethernet-moottorilta vastaanotettujen viestien pohjalta, ja ne kuuntelevat toisten ohjainten julkaisemia viestejä. Ulkoisia yhteysilmentymiä voi olla useita, mutta ne eivät ole välttämättömiä.

```
const defaultOptions = {
  protocol: 'amqp',
  hostname: 'localhost',
  username: 'esimerkki_käyttäjä',
  password: '<esimerkki_käyttäjän salasana>',
  heartbeat: 30,
};
```

Esimerkkikoodi 4. Oletusyhteysolio, ja sen sisältämät tiedot

AmqpConnection-luokka sisältää oletusasetukset sisältävän olion (esimerkkikoodi 4), jonka tietoihin perustuen luodaan asiakasohjelman yhteys RabbitMQ-palvelimeen. Edellä mainittu olio luo aina yhteyden ohjaimen omaan palvelimeen,

mutta mikä tahansa sen sisältämistä kentistä voidaan tarpeen vaatiessa korvata konstruktorikutsun yhteydessä. Korvaamalla *hostname*-kentän arvo ipv4-osoitteella otetaan yhteys etäpalvelimeen, tai jos halutaan vaihtaa amqp-viestintäprotokollaa, vaihdetaan *protocol*-kentän arvo. Pitää huomata, että protokollan vaihto edellyttää myös asetusten muuttamista yhteyden kohteena olevan RabbitMQ-palvelimen asetuksissa.



Kuva 10. Ohjainten muodostama verkosto

Ohjaimet muodostavat automaattisesti keskinäisen verkon luomalla asiakasyhteyksiä toistensa RabbitMQ-palvelimella olevaan vaihteeseen. Kuvassa 10 luodaan katsaus viestien kulkuun ohjainten välillä, vaikka siinä on kuvattuna vain kahden ohjaimen välinen viestintä. Samaa periaatetta voidaan soveltaa myös useamman ohjaimen välillä. Ohjain lähettää viestejä ulospäin käyttäen sen omaan palvelimeen yhdistettyä amqp-asiakasohjelmaa (kuva 10 ’sisäinen’-asiakas). Viestiä lähetettäessä taakanjakomoduuli ohjaa viestin sisäisestä yhteydestä vastaavalle AmqpConnection-luokan ilmentymälle, joka julkaisee viestin ohjaimen RabbitMQ-palvelimen vaihteeseen käyttäen taakanjakomoduulilta

saatua reititysavainta, joita on kahdenlaisia: täsmälähetysavain, joka käytännössä on kohteena olevan ohjaimen sarjanumero, ja yleislähetysavain, jonka kohteena ovat kaikki palvelimen vaihteeseen sidotut viestijonot, toisin sanoen muut verkostossa olevat ohjaimet.

AmqpConnectionManager-luokka sisältää ulkoiseen viestintään tarkoitettuja AmqpConnection-luokan ilmentymiä nolla tai enemmän. Nämä ilmentymät avaavat yhteyden ulkoiseen RabbitMQ-palvelimeen, kiinnittyvät palvelimen 'ulkoinen'-vaihteeseen ja luovat uuden viestijonon, jonka ne sitovat edellä mainittuun vaihteeseen käyttäen reititysavaimina omaa sarjanumeroaan ja yleislähetysavainta. Kun sidonta on saatettu loppuun, yhteysolio on valmis vastaanottamaan yhteyden kohteena olevan palvelimen vaihteeseen julkaistuja viestejä.

Kun ulkoisesta yhteydestä vastaava yhteysolio vastaanottaa viestin, se ilmoittaa vastaanotetusta viestistä AmqpConnectionManager-luokalle, jonka tapahtumakäsittelijä ohjaa viestin eteenpäin taakanjakomoduulin käsiteltäväksi. Määrittämällä viestityyppikohtaisia tapahtumakäsittelijöitä liikennettä voidaan ohjata myös tarpeen vaatiessa muihin moduuleihin, minkä ansiosta viestintäkerrosta voidaan käyttää, kun ohjainten välistä suoraa viestintää pyritään laajentamaan taakanjaon lisäksi myös muille ohjaimen toiminnan osa-aluille.

6 Taakanjakoalgoritmi

Varsinaisesta solmutaakan jaosta vastaava algoritmi on osa *LoadBalancer*-luokkaa, jossa märitetään toiminnon edellyttämien viestien rakenne, käsittelijät, järjestys sekä säännöt siirtopäätösten tekemiselle. Taakanjaossa käytettävä logiikka on kapseloitu *Task*-yläluokan periviin tehtäviin, joiden ilmentymiä luodaan *LoadBalancer*-luokassa toisilta ohjelmilta vastaanotettujen viestien pohjalta tai ajastettuina tehtävinä ohjaimen omasta toimesta. Uusi taakanjakotehtävä luodaan ohjaimessa ajastetusti joka neljäs tunti, ja se käynnistää ohjainten välisen taakanjaon. Tarpeen vaatiessa aikaväliä voidaan säätää asetusolion avulla, joko lyhemmäksi tai pidemmäksi. On kuitenkin hyvä huomioida, että koska taa-

kanjako voi olla verrattain raskas toimenpide laajojen mesh-verkkojen yhteydessä, sitä ei kannata toistaa kovinkaan tiuhaan. Toisaalta jatkuva solmutaakan tarkistaminen ja jatkuvat pienehköt siirrot eivät myöskään paranna ohjainten toimivuutta harvaan suoritettavaan toimintoon nähden.

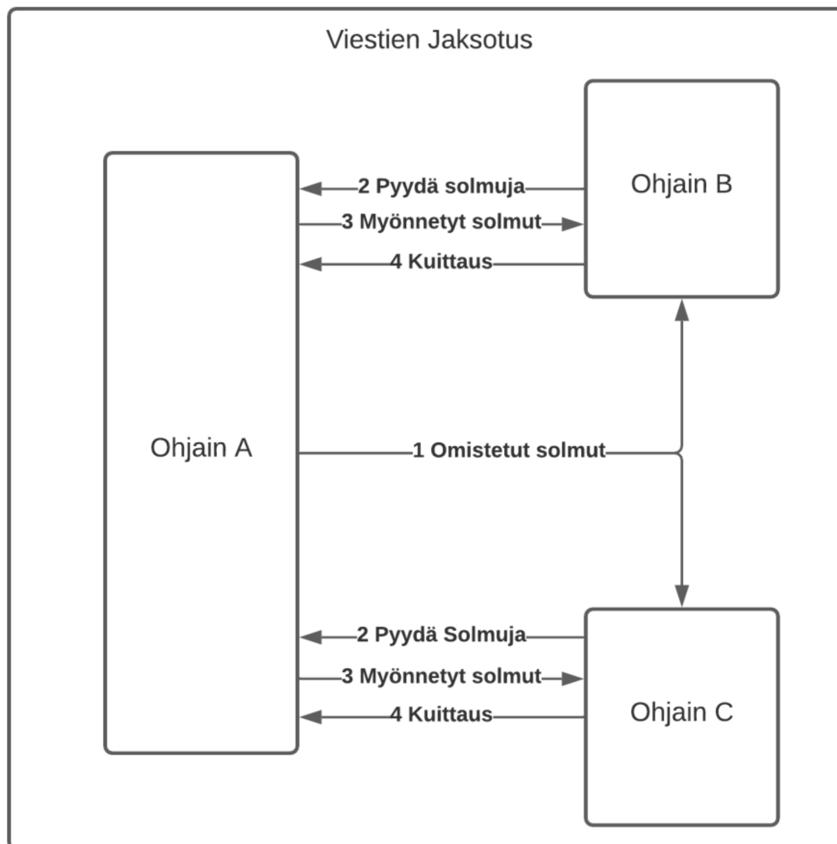
Tässä opinnäytetyössä luodaan taakanjakoalgoritmi, jonka tarkoitus on pyrkiä tasaamaan AA-ohjainten välistä solmutaakkaa ohjaamalla solmu sellaisen ohjaimen alaisuuteen, joilla on kyseiseen solmuun paras yhteys. Yhteyden laatua mitataan mesh-viestien sisältämän 'hops'-kentän lukuarvolla, joka mittaa sitä, kuinka monen välittäjäsolmun kautta viesti on kulkenut ennen kuin se on saapunut määränpäähänsä (mitä suurempi arvo sitä parempi). Mikäli tulevaisuudessa halutaan käyttää useampaa eri kriteeriä yhteyden laadun arvioinnissa tai ainoastaan korvata 'hops'-kenttä esimerkiksi vastaanotettujen viestien määrällä, se tapahtuu helposti, koska algoritmin logiikka on kapseloituna tehtäviin. Täten yhden tehtävän muokkaaminen ei välttämättä edellytä muutoksia toisten rakenteessa ja uusien tehtävien lisääminen onnistuu helposti luomalla uusi *Task*-luokan perivä alaluokka, eikä itse *LoadBalancer*-luokkaan tarvitse tehdä muuta kuin yhdistää viesti tai ajastettu tapahtuma uuden toiminnon luontiin.

Taakanjakoalgoritmi perustuu siihen, että kukin ohjain lähettää toisille ohjaimille tietyin aikavälein tiedot siitä, mitkä solmut ovat sen alaisuudessa ja minkä laatuinen yhteys ohjaimella on omistamiinsa solmuihin. Vastaanottavat ohjaimet puolestaan vastaavat lähettämällä tiedot niistä solmuista, joihin niillä on parempi yhteys ja pyytävät kyseisiä solmuja omistukseensa. Alkuperäisen viestin lähettäjä käsittelee saamansa vastaukset ja luovuttaa vastauksissa nimetyt solmut sille ohjaimelle, jolla on niihin paras yhteys.

6.1 Taakanjakoviestit

Jotta taakanjako AA-ohjainten välillä olisi mahdollista, on ensin määritettävä ohjainten välinen viestiliikenne ja viestien jaksotus algoritmin eri vaiheissa. Viestien ja taakanjaon jaksotus on tärkeää. Kun yhden mesh-verkon yhteydessä toi-

mii useampi AA-ohjain, on varmistuttava siitä, ettei samaa solmua tarjota useammalle eri ohjaimelle. Nimenomaan tästä syystä taakanjako käynnistyy siten, että ohjain tarjoaa omia solmujaan toisille. Jos ohjaimet voisivat pyytää yksittäisiä solmuja toisiltaan ilman omistajan aloitetta, saatettaisiin päätyä tilanteeseen, jossa omistaja käsittelee yhdenaikaisesti kahta samaan solmuun kohdistuvaa siirto-operaatiota. Koska ohjaimet tarjoavat omia solmuja toisille, ei synny riskiä siitä, että sama solmu olisi osa useamman eri ohjaimen siirtoja.



Kuva 11. Taakanjakoviestien jaksotus

Kuvassa 11 on kuvattu taakanjakoviestien jaksotus kolmen AA-ohjaimen (A, B, C) välillä. Taakanjako-operaatio on käynnistynyt ohjaimessa A, joka lähettää toisille ohjaimille viestin, joka sisältää ohjaimen omien solmuyhteyksien laatua kä-

sittelevät tiedot. Lähetyksen jälkeen ohjain A odottaa vastauksia toisilta ohjaimilta (oletusasetukselta 5 sekunnin ajan). Jos vastauksia ei saavu, toimenpide perutaan ja toistetaan uudestaan määritetyn aikavälin kuluttua.

Vastaanottavat ohjaimet (kuvassa 11 B ja C) vertaavat A:n lähettämän viestin sisältämiä yhteyden laatutietoja omiin tietokannasta haettuihin tietoihin kyseisten solmujen osalta. Mikäli vastaanottavalla ohjaimella on parempi yhteys yhteen tai useampaan A:n solmuista, se lähettää vastauksena omat laatutiedot niiden solmujen osalta, joihin sillä on parempi yhteys.

Ohjaimen A saadessa vastauksen viestin sisältö tallennetaan väliaikaisesti *LoadBalancer*-luokassa olevaan *replies*-hajautustaulukkoon. Jokaisesta vastauksen sisältämästä solmusta tehdään taulukkoon avain-arvopari, jossa solmun tunniste toimii avaimena ja arvona on lista yhteydenlaatutietoja. Tällöin, jos esimerkiksi B ja C pyytävät samaa solmua omistukseensa, kummankin ohjaimen tiedot yhteyden laadusta tallentuvat samaan listaan ja kun ohjain A alkaa käsittelemään saamiaan vastauksia, sen ei tarvitse hakea tietoja eri paikoista. Vastausajan umpeutuessa, ohjain A alkaa käsittelemään saamiaan vastauksia. Jokaista vastannutta ohjainta varten luodaan oma lista, johon lisätään sille lähetettävät solmut. Käydessään läpi vastauksia ohjain A valitsee kullekin solmulle ohjaimen, jolla ’hops’-kentän arvo on suotuisin eli mahdollisimman suuri. Kun kaikki vastausten sisältämät solmut on käsitelty, ohjain A lähettää ohjaimille B ja C kohdistetut viestit, jotka sisältävät listan kullekin ohjaimelle siirrettävien solmujen tunnisteista ja vaihtaa luovutettavien solmujen tilan ’omistetusta’ ’vieraaksi’.

Kun vastaanottava ohjain saa siirtoviestin (kuvassa 11 viesti 3), se muuttaa siirtoviestin sisältämien solmujen tilan vieraasta omaksi ja lähettää sen jälkeen kuittausviestin (kuvassa 11 viesti 4) takaisin ohjaimelle A. Mikäli ohjain A ei vastaanota kuittausviestiä ennalta määrätyn ajan sisällä, siirtotoimenpide perutaan.

6.2 Viestinkäsittely ja toimenpiteet

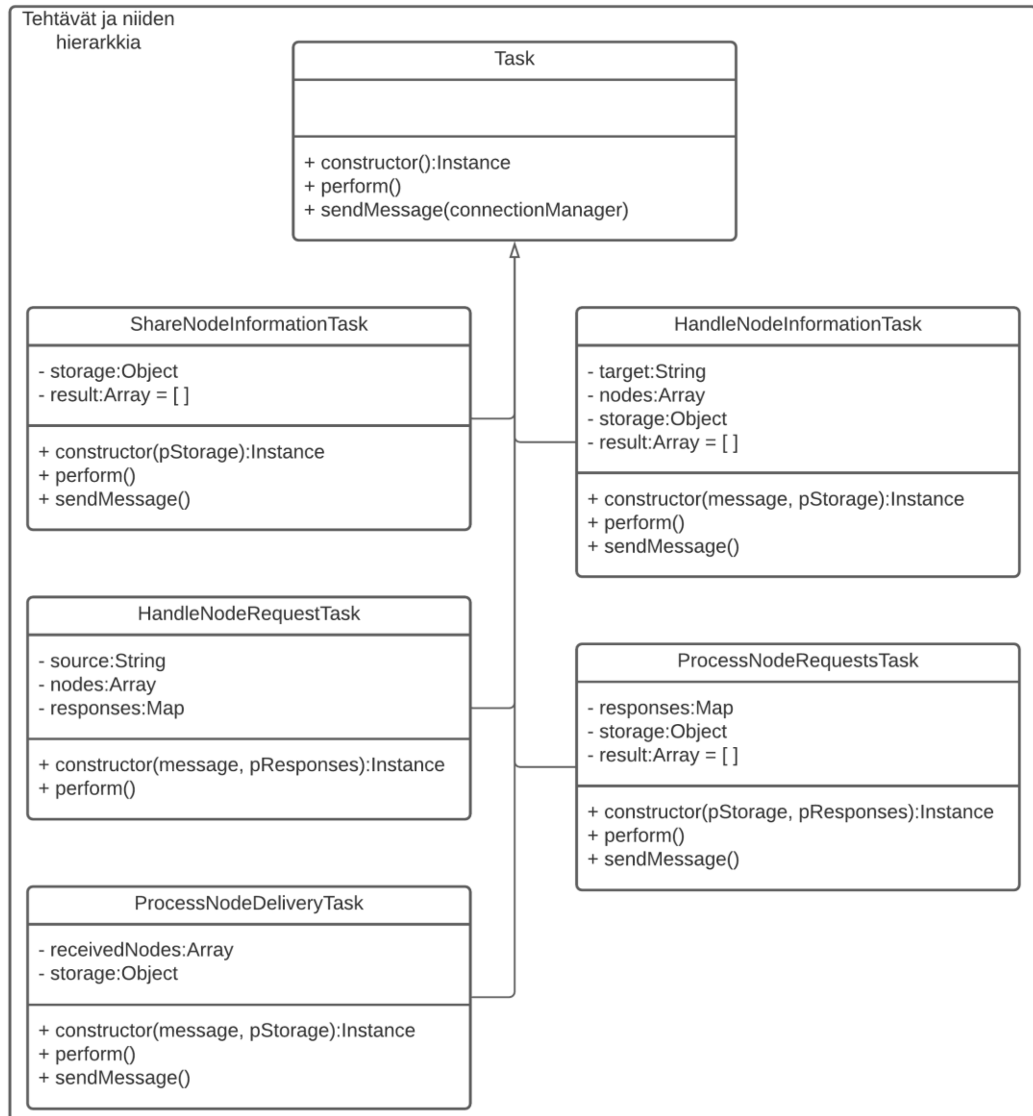
Taakanjaon viestinkäsittely perustuu jokaisen viestin sisältämään viestityyppi-kentän arvoon, minkä ansiosta moduulin toiminnallisuutta on helppo laajentaa yhdistämällä uusi viestityyppi joko uuteen tai jo olemassa olevan tehtävän luontiin. Tämä tapahtuu *LoadBalancer*-luokan *processControllerMessage*-metodin sisältämässä *tasks*-oliossa.

```
processControllerMessage(msg) {
    const type = msg.header.type;

    const tasks = {
      'owned-nodes': (msg) => {
        return new AnalyzeNodeOfferTask(msg, this.storage);
      },
      'node-request': (msg) => {
        if (!this.isWaitingForResponses) return new Task();
        return new HandleOverloadResponseTask(msg, this.responses);
      },
      'node-delivery': (msg) => {
        return new NodeDeliveryTask(msg, this.storage);
      },
    };
    const task = tasks[type](msg);
    task.perform();
    task.sendMessage(this.connectionManager);
}
```

Esimerkkikoodi 5. Ohjainviestien käsittely *LoadBalancer*-luokassa

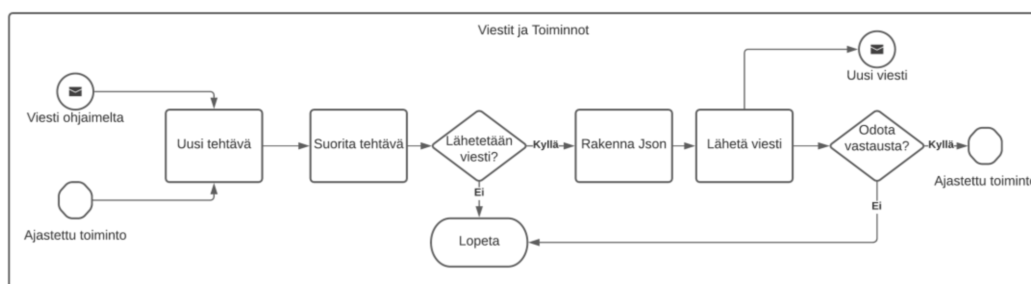
Vastaanotetun viestin sisällön tai tietokannasta haettujen tietojen esikäsittely voidaan hoitaa esimerkkikoodi 5:ssä esillä olevan *tasks*-olion ominaisuuksina toteutetuissa käsittelijäfunktioissa, jotka lopputuloksena palauttavat viestin edellyttämän tehtäväilmentymän.



Kuva 12. Tehtäväluokat

Kuvassa 12 on esitetty taakanjakomoduulin sisältämät tehtäväluokat ja niiden sisältö. Kuten aiemmin todettiin, jokainen tehtävä sisältää yhden taakanjaossa käytettävän operaation, ja kokonaisuudessaan algoritmi perustuu näiden yksittäisten tehtävien ketjuttamiseen. Koska operaatiot on kapseloitu tehtäviin, algoritmin osia on helppo muokata tarpeisiin sopivaksi, eikä tarvitse huolehtia siitä, että jokin toinen operaatio lakkaa toimimasta halutulla tavalla.

Tehtäville välitetään parametreina kaikki niiden suorittamisen edellyttämät resurssit, se mitä resursseilla tehdään (kuva 12 *Task*-luokan *perform*-metodi) ja tarvitseeko tehtävän suorituksen jälkeen lähettää viesti toisille AA-ohjaimille (kuva 12 *Task*-luokan *sendMessage*-metodi). *Task*-yläluokan metodit ovat tynkiä, jotka antavat vain ilmoituksen siitä, että kyseisellä metodilla ei ole toteutusta. Kyseisten metodien toiminta määritetään tarpeen mukaan konkreettisesti alaluokissa, jotka perivät *Task*-luokan.



Kuva 13. Viestinkäsittely *LoadBalancer*-luokassa

LoadBalancer-luokka vastaa taakanjakoviestien käsittelystä, ajastetuista tehtävistä ja tietokantayhteydestä. Kuvassa 13 esitetään viestinkäsittelyprosessi kokonaisuudessaan. Ensimmäisenä luodaan uusi tehtävä joko ajastettuna toimintona tai toiselta ohjaimelta vastaanotetun viesti pohjalta. Heti luonnin jälkeen tehtävä suoritetaan, minkä jälkeen tarkistetaan, lähetetäänkö toisille ohjaimille viesti saavutetuista lopputuloksista. Mikäli tarkoituksena on lähettää viesti, tehtävän lopputuloksien pohjalta rakennetaan olio, joka sisältää vastaanottavan ohjaimen kannalta oleelliset tiedot lopputuloksista. Tämän jälkeen olio liitetään osaksi JSON-muotoista viestiä, jonka *header*-osiossa on määritetty viestin määränpää, tyyppi, lähettäjä ja aikaleima. Jatkotoimenpiteitä ei tarvita, jos viesti lähetetään kohdistetusti jollekin tietylle ohjaimelle, sillä mikäli vastaanottava ohjain lähettää vastauksen, se käsitellään uutena tehtävänä. Jos viesti lähetetään yleislähetysavainta käyttäen ja sille odotetaan vastauksia, luodaan uusi ajastettu toiminto, joka luo uuden tehtävän vastausten käsittelyä varten määrätyn ajan kuluttua.

7 Taakanjaon testaus ja loppusanat

Tällä hetkellä moduulin toiminnallisuutta ei pystytä testaamaan oikean ActiveAhead-järjestelmän yhteydessä, koska pilvipalvelujen puolella ei ole vielä saatu toteutettua muutoksia, jotka mahdollistaisivat paikallisten toimintojen käytön. Algoritmin toimivuus todennetaan käyttämällä kahta ActiveAhead-ohjainta. Niiden tietokantoihin luodaan 200 lumesolmua, joiden tunnisteet on juoksevasti numeroitu yhdestä kahteen sataan. Täten varmistetaan, että jokaisella ohjaimella on tiedot samoista lumesolmuista. Seuraavaksi jokaiselle lumesolmulle luodaan myös ohjainkohtaiset yhteystilastot satunnaismuuttujaa hyödyntäen. Lopputuloksena jokaisella ohjaimella on saman lumesolmun osalta erilaiset tilastot, mikä kuvastaa tilannetta, jossa ohjaimet on asetettu eri puolille mesh-verkkoa.

Ensimmäisessä testissä simuloidaan yleistä käytännön tilannetta, jossa valtaosa solmuista on päätynyt ensimmäisen mesh-verkon yhteyteen tuodun AA-ohjaimen omistukseen. Olosuhteet luodakseen kaikki lumesolmut osoitetaan aluksi yhden ohjaimen omistukseen, minkä jälkeen odotetaan, että ajastettu taakanjako käynnistyy. Normaalisti solmutaakka tarkistetaan neljän tunnin välein, mutta tämän testin yhteydessä ensimmäinen tarkistuskutsu tehdään viiden minuutin päästä ohjaimen käynnistyksestä, jotta testin tulokset saadaan nopeasti tarkasteltavaksi. Tuloksista on tarkoitus varmentaa, että lumesolmuihin kohdistuvat siirrot tapahtuvat tavoitteiden mukaisesti eli solmut siirtyvät sen ohjaimen omistukseen, jolla ’hops’-kentän arvo on korkein.

Toisessa testissä simuloidaan tilanne, jossa kaksi ohjainta aloittavat taakanjaon samanaikaisesti. Tässä testissä lumelaitteiden omistus on jaettu siten, että sata ensimmäistä lumesolmua (tunnisteet 1–100) ovat ensimmäisen ohjaimen omistuksessa ja loput sata (tunnisteet 101–200) toisen ohjaimen omistuksessa. Testiä varten moduuliin luodaan väliaikainen viestityyppi, joka vastaanotettaessa käynnistää taakanjako-operaation. Ensimmäinen ohjain lähettää edellä mainitun viestin, jotta toinen ohjain käynnistää taakanjaon. Heti viestin lähetyksen jälkeen myös ensimmäinen ohjain käynnistää oman taakanjako-operaation. Tavoitteena

on todentaa, etteivät kahden samanaikaisen operaation taakanjakoviestit mene sekaisin keskenään.

7.1 Tulokset

Ensimmäisessä testi toistettiin kymmenen kertaa ja jokaisella kerralla lumesolmut jakautuivat suurin piirtein tasan ohjainten välillä. Vaihteluväli oli noin kymmenen lumelaitetta, mikä oli odotettavissa käytettäessä satunnaismuuttujaa hyödyntäen luotuja yhteystilastoja. Lumesolmujen siirrot tapahtuivat odotusten mukaisesti 'hops'-kentän arvoon perustuen, ja jokainen lumesolmu päätyi korkeimman arvon omaavan ohjaimen alaisuuteen.

Toisen testin tulokset olivat lumesolmujen jaon osalta samankaltaiset kuin ensimmäisessä testissä, eikä operaatioiden viestintä häiriintynyt samanaikaisuudesta huolimatta.

7.2 Lopuksi

Tässä opinnäytetyössä kehitetty taakanjakomoduuli ja sen käyttämä viestintäkerros toimivat odotetulla tavalla testeissä, joskin on otettava huomioon se, että moduulia ei vielä voitu testata oikean ActiveAhead-järjestelmän yhteydessä. Tästä syystä moduulin kehityksessä kiinnitettiin huomiota siihen, että moduuli on rakenteeltaan helposti muokattavissa, mikä näkyy muun muassa siinä, että algoritmin logiikka on toteutettu jaksottamalla viestejä, joiden pohjalta luodaan taakanjaon kannalta tärkeitä tehtäviä. Jos vastaisuudessa nousee tarve muuttaa jotakin algoritmin osaa tai lisätä moduuliin uusia tehtäviä, muutokset voidaan toteuttaa yhdessä paikassa moduulia ilman, että algoritmin muiden osien toiminta häiriintyy.

Työn tuloksena syntynyttä moduulia ei ole tarkoitus käyttää sellaisenaan tuotannossa, mutta sitä käytetään pohjana tuotantoversion kehityksessä. Tuotantoon

siirtäminen edellyttää muun muassa automatisoitujen testien kehitystä sekä laajaa käytännön testausta, eikä näiden toteuttaminen tämän opinnäytetyön rajoissa ollut mahdollista.

Taakanjakomoduulin jatkokehityksessä tullaan todennäköisesti lisäämään kriteereitä solmujen yhteyden laadun määrittämiseksi. Esimerkiksi ’hops’-kentän arvon rinnalle voidaan nostaa vastaanotettujen viestien määrä, jonka avulla pyritään arvioimaan sitä, kuinka suuri osa solmun lähettämistä paketeista saapuu perille määränpäähän. Jatkokehityksen kohteena tulee siis todennäköisesti olemaan myös yhteyksien laatutietojen keruu ja tarkempi analysointi. Myös moduulin kykyä selvitä itse mahdollisista yhteyksiin liittyvistä virhetilanteista tullaan testaamaan ja kehittämään tarpeen vaatiessa, koska tavoitteena on AA-ohjain, joka kykenee toimimaan automaattisesti.

Muita kehityksen kohteita saattavat olla myös se, että RabbitMQ-yhteyksissä käytetään *Transport Layer Security* (TLS) -salausprotokollaa salaamattoman *Transmission Control Protocol* (TCP) -protokollan sijasta.

Lähteet

Carlson, Josiah L. Redis in Action. Manning. Saatavilla osoitteessa: https://redis.com/ebook/redis-in-action/. Luettu 16.8.2021.

Docker, Inc. Docker overview. Verkkoaineisto. Osoitteessa: https://docs.docker.com/get-started/overview/. Luettu 8.11.2021.

Galiegue, Francis – Zyp, Kris – Court, Gary. 2013. Json Schema: Core definitions and terminology. Internet Engineering Task Force. Verkkoaineisto. Osoitteessa: http://json-schema.org/draft-04/json-schema-core.html. Luettu 20.10.2021.

Galiegue, Francis – Zyp, Kris – Court, Gary. 2013. Json Schema: Interactive and non interactive validation. Internet Engineering Task Force. Verkkoaineisto. Osoitteessa: https://json-schema.org/draft-04/json-schema-validation.html. Luettu 20.10.2021.

Helvar. ActiveAhead Generation 2. Verkkoaineisto. Osoitteessa: https://helvar.com/fi/ratkaisut/activeahead/. Luettu 8.11.2021

International Well Building Institute. WELL v2. 2020. Verkkoaineisto. Osoitteessa: https://a.storyblok.com/f/52232/x/eee22a8551/well-building-standard-v2-wellv2-q4-2020-wellapv2.pdf. Luettu 8.11.2021.

Kolderup, Ken. 2017. Introducing Bluetooth Mesh Networking. Bluetooth SIG. Verkkoaineisto. Osoitteessa: https://www.bluetooth.com/blog/introducing-bluetooth-mesh-networking/. Luettu 5.8.2021.

MDN Web Docs. JavaScript. Verkkoaineisto. Osoitteessa: https://developer.mozilla.org/en-US/docs/Web/JavaScript. Luettu 12.8.2021.

OpenJS Foundation. About node.js. Verkkoaineisto. Osoitteessa: https://nodejs.org/en/about/. Luettu 12.8.2021.

Ghandi, Rajeev – Szmrecsanyi, Peter. 2019. The Benefits of Containerization and What It Means for You. Verkkoaineisto. Osoitteessa: https://www.ibm.com/cloud/blog/the-benefits-of-containerization-and-what-it-means-for-you. Luettu 8.11.2021.

Redis Ltd. Introduction to Redis. Verkkoaineisto. Osoitteessa: https://redis.io/topics/introduction . Luettu 16.8.2021.

Videla, Alvaro – Williams, Jason J.W. 2012. RabbitMQ in Action: Distributed Messaging for Everyone. Saatavilla osoitteessa: https://livebook.manning.com/book/rabbitmq-in-action/about-this-book/. Luettu 20.8.2021.

Woolley, Martin. 2020. Bluetooth Mesh Networking - An Introduction for Developers. Bluetooth. Saatavilla osoitteessa: https://www.bluetooth.com/bluetooth-resources/bluetooth-mesh-networking-an-introduction-for-developers/. Luettu 8.11.2021.