Antero Juutinen

Esineiden rajapinta pilvipalveluihin

Tietojenkäsittely,
Tradenomi

Syksy 2016

KAJAANIN
AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

TIIVISTELMÄ

Tekijä: Juutinen, Antero

Työn nimi: Esineiden rajapinta pilvipalveluihin

Tutkintonimike: Tietojenkäsittely, tradenomi (AMK), Data center -ratkaisut

Asiasanat: Esineiden internet, World Wide Web, REST, MQTT, Tietokannat

Opinnäytetyö käsittelee esineiden internet -ilmiötä, joka on mullistamassa palveluiden tuottamisen ja kuluttamisen internetissä. Työssä tutkittiin ja tuotettiin erilaisia teknologioita, jotka tukevat ja mukautuvat esineiden internetin vaativiin haasteisiin. Tavoitteina oli prototypoida mahdollisimman monipuolisesti erilaisia ratkaisuja palveluiden tuottamiseen niin, että ne olisivat helposti sovellettavissa opetuskäyttöön. Työ tehdään Kajaanin ammattikorkeakoulun ICT-muuntokoulutuksen tarpeisiin sekä koulun opetusympäristöjen kehittämisen vuoksi.

Esineiden internetin konsepti mahdollistaa normaalista poikkeavia teknologioita tietoverkkojen ja palvelualustojen toteutuksessa. Älykkäiden esineiden potentiaalinen hyöty on suuri, mutta sitä ennen ratkaistavat ongelmat on monimutkaisia. Nykyaikaiset pilvipalvelut ovat rakennettu internetin päällä toimivaan World Wide Web -palvelualustaan. Tämä palvelualusta on kehittynyt vuosien aikana sovellusalustaksi ja mahdollistaa käytännössä kaiken fyysisten maailman ilmiöiden integroinnin internetiin. Integrointi toteutetaan kuvaamalla asian tai esineen sen hetkinen tila resurssina WWW-palvelualustassa. Resurssi voidaan tunnistaa URI-tunnuksien avulla, ja hypermedia kuvaa resurssien välistä linkitystä ja sen mahdollistamaa epälineaarista resurssien käyttöä.

Työn käytännön osuutena suunniteltiin ja toteutettiin älykkäitä esineitä tukevia pilvipalveluita. Toteutettavat pilvipalvelut rajattiin kahteen eri aiheeseen. Ensimmäinen on älykkäitä esineitä tukevan REST-arkkitehtuurin mukainen WWW:n rajapintapalvelu. Sen ideana on integroida esineitä mahdollisimman yleispätevällä tavalla nettiin ja mahdollistaa lisäpalveluiden tuottaminen kallisarvoisella sensoritiedolla.

Toinen prototyyppi rakennettiin älykkäiden laitteiden kommunikointiongelmien ratkaisemiseksi. MQTT on sovelluskerroksen protokolla, joka on erikoistunut sulautettujen laitteiden viestintään. Se on niin sanottu julkaise ja tilaa -protokolla, joka mahdollistaa viestinnän vähävirtaisissa tai muuten heikko tehoisissa yhteyksissä. Protokolla toteuttaa viestintäpalvelimen avulla asiakassovelluksen ja palvelinsovelluksen erottamisen niin, että ne voivat kehittyä toisistaan riippumatta. Palveluiden tueksi rakennettiin hajautettu NoSQL-tietokantajärjestelmä.

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Author: Juutinen, Antero

Title of the Publication: Cloud integration of Things

Degree Title: Bachelor of Business Information Technology

Keywords: Internet of things, World Wide Web, REST, MQTT, Databases

This thesis deals with a phenomenon called the internet of things. It is revolutionizing the way we consume and produce services over the internet. The purpose of this work is to study and produce different technologies that support and integrate embedded systems to the existing cloud and web infrastructure. The work was done for the needs of ICT studies in the Kajaani University of Applied Sciences, and to develop the functions of the school's different laboratories.

The theoretical part of the thesis introduces the concept of internet of things, and discovers how it can enable different divergent technological solutions in networking and service platforms. The potential of internet of things is huge, but so are the challenges that must be solved before that. Modern cloud services are built on top of the World Wide Web service platform. This service platform has developed over the years to an application platform and permits the abstraction of physical things or ideas, so that the information can be presented over the internet. These abstract representations are called resources and can be identified using global URI identification. Hypermedia is the concept of linking resources together and accessing them in a non-linear way.

The practical part of this thesis focuses on planning and producing prototype applications and infrastructures that support the internet of things integration to cloud services. Implementations were limited down to two different prototypes. First is a web service that implements the REST-architecture. Its idea is to make the integration of cloud services to embedded devices possible through a generalized interface and to access and store important sensor data in a general machine-to-machine way.

The second prototype focuses on the communications of embedded devices. The available energy and connection stability and speed is often very limited when it comes to internet of things. Thus, it is required that the software and communication protocols are specialized for this problem. MQTT is a publish and subscribe -protocol that decouples software with a centralized message broker. It is also designed with previously presented problems in mind and allows flexible communication between embedded devices and the cloud. Software implementations that support the protocol were made on Arduino Uno microcontroller and server-side operating system. A centralized message server was installed, which then connects these two implementations. To support these prototypes, a clustered and fault tolerant NoSQL database system was implemented.

SISÄLLYS

SYMBOLILUETTELO

BSON	Binäärisessä muodossa oleva JSON-merkintäkieli.
DNS	Internetin nimipalvelujärjestelmä.
HTML	Hypertekstin merkintäkieli.
HTTP	Hypertekstin siirtoprotokolla.
IAAS	Infrastructure as a Service. Palvelimien ja palvelinsalien ulkoistaminen.
M2M	Kahden tietokoneen välinen kommunikaatio.
PAAS	Platform as a Service. Palvelualustan ulkoistaminen.
P2P	Vertaisverkko.
REST	Representational state transfer. Sovelluskehitys arkkitehtuurityyli.
SAAS	Software as a Service. Ohjelmiston hakkimista palveluna.
URI	Merkkijono, jolla tunnistetaan resurssin nimi tai paikka.
WLAN	Langaton lähiverkkotekniikka.
YALM	YALM Ain’t Markup Language. Ihmiselle luettava merkintäkieli.

1 JOHDANTO

Tietokoneet ja laitteistot ympärillämme ovat jatkuvasti älykkäämpiä. Yhä enemmän fyysistä maailmaa yhdistetään virtuaaliseen, koska sen lupaamat hyödyt ovat suuret. Autot, rakennukset, teollisuuden koneistot, kodinhoidonlaitteet, kasvit, ihmiset ja eläimet ovat kaikki osa fyysistä maailmaa. Kun näihin asioihin lisätään tietoteknistä älyä, voidaan esimerkiksi paikantaa eksyneitä lemmikkejä, mitata ihmisen terveyttä ja ennaltaehkäistä sydänkohtaus, hallita kodin turvallisuutta sekä lämpötilaa etänä ja optimoida tuotantoprosesseja.

Tätä ilmiötä kutsutaan monella eri nimellä, mutta tunnetuin on esineiden internet, joka kääntyy englannin kielen sanoista ”Internet of things”. Esineen tai asian liittäminen internetiin tarjoaa todella suuren potentiaalisen hyödyn, mutta myös haasteet sen toteutuksessa ovat suuret. Haasteita ovat esimerkiksi massiivisen tiedonmäärän kuljettaminen, käsittely, tallentaminen sekä integroiminen moderneihin palvelinjärjestelmiin. Kahvinkeittimen täytyy pystyä jotenkin kommunikoimaan sen rajoittuneella suorituskyvyllä tietonsa palvelimelle, ja kun näitä laitteita on useita tuhansia, pitää palvelimien ja palveluiden pystyä skaalautumaan haasteeseen.

Tietojenkäsittelyn kontekstissa pilvi-käsite verhoaa internetin takana löytyvän monimutkaisen infrastruktuurin yhden termin alle. Pilvipalveluilla viitataan pilvessä toimiviin palveluihin, kuten tietokantoihin, nettisivuihin tai rajapintasovelluksiin.

Opinnäytetyö tehdään pääasiallisesti Kajaanin ammattikorkeakoulun ICT-muuntokoulutusprojektin opetusmateriaaliksi, mutta toimii myös koulun laboratorioiden toimintoja kehittävänä esimerkkinä. Työssä käydään läpi esineiden internet -käsitteeseen liittyviä teknologioita ja toteutetaan erilaisia rajapinta- ja kommunikointiratkaisuja, tavoitteena saada mahdollisimman monipuolinen kokemus eri vaihtoehdoista.

2 ESINEIDEN INTERNET

Esineiden internet sisältää useita monialaisia teknologioita ja on niiden yhteistyön summa (Kellmereit & Obodovski, 14). Sen perimmäinen tarkoitus on yhdistää fyysinen maailma virtuaaliseen, ja täten luoda yleistä lisäarvoa sekä hyötyä (Kellmereit & Obodovski, 10). Konseptilla on sen historian aikana ollut useita eri termejä kuten jokapaikan tietotekniikka, M2M-kommunikointi, sulautettu tietotekniikka ja älykkäät järjestelmät. Nimeämisen vaikeus kuvastaa sitä, kuinka haasteellinen konsepti on. (Kellmereit & Obodovski, 16.)

Tietokoneiden välinen kommunikointi mahdollistaa esineiden internetin. Esineet jakaantuvat tyhmiin sekä älykkäisiin. Jokainen laite, joka on yhdistettynä internetiin tai toisiin laitteisiin, on älykäs. Tämä älyn määrä kasvaa ympäristössämme jatkuvasti, ja se tapahtuu ihmisten periferiassa, eikä sitä tai sen lisääntymistä välttämättä huomioida jokapäiväisessä elämässä. Ihmisiä ei kiinnosta, jos heidän jääkaappinsa on yhdistettynä internetiin. Älykkään jääkaapin implikoitu hyöty on kuitenkin mahdollisesti todella suuri, ja täten haasteet piilevät arvon toimittamisessa loppukäyttäjälle eli siinä, mitä se voi meille tehdä. (Kellmereit & Obodovski, 18.)

M2M-kommunikaatio ja ekosysteemi voidaan jakaa kolmeen kategoriaan, jotka ovat datan hankinta, siirto ja analyysi. Datan hankinta kattaa älykkäät esineet ja kaiken laitteiston. Sensorit keräävät tietoa ja lähettävät sen tietoverkkoon. Tietoverkon tehtävä on siirtää tieto haluttuun paikkaan. Tietoverkko voi esimerkiksi olla WLAN, lyhyen matkan radiolähetys tai mobiiliverkko. Analysointi tapahtuu pilvessä, jossa sovellukset tai ihmiset hyödyntävät ja tulkitsevat kerätyn tiedon. (Kellmereit & Obodovski, 42.)

Uusien teknologioiden käyttöönotto suuressa mittakaavassa on hidasta, jos ympäristö ei ole siihen valmis. Esineiden internet vaatii ympäristöltään paljon. Keskeiset teknologiat, jotka ovat ajaneet konseptin kehitystä ja tehneet siitä realiteetin, ovat tietokoneiden miniatyrisointi, edullisuus ja langattomuus. Tietokoneet ovat tarpeeksi tehokkaita pienemmässä muodossaan, virranhallinta ja erilaiset akustot mahdollistavat laitteiden sulauttamisen esineisiin, kommunikointi on mahdollista

langattomasti ja kaikki tämä on paljon edullisempaa kuin aikaisemmin**.** (Kellmereit & Obodovski, 22.)

Tietoverkot

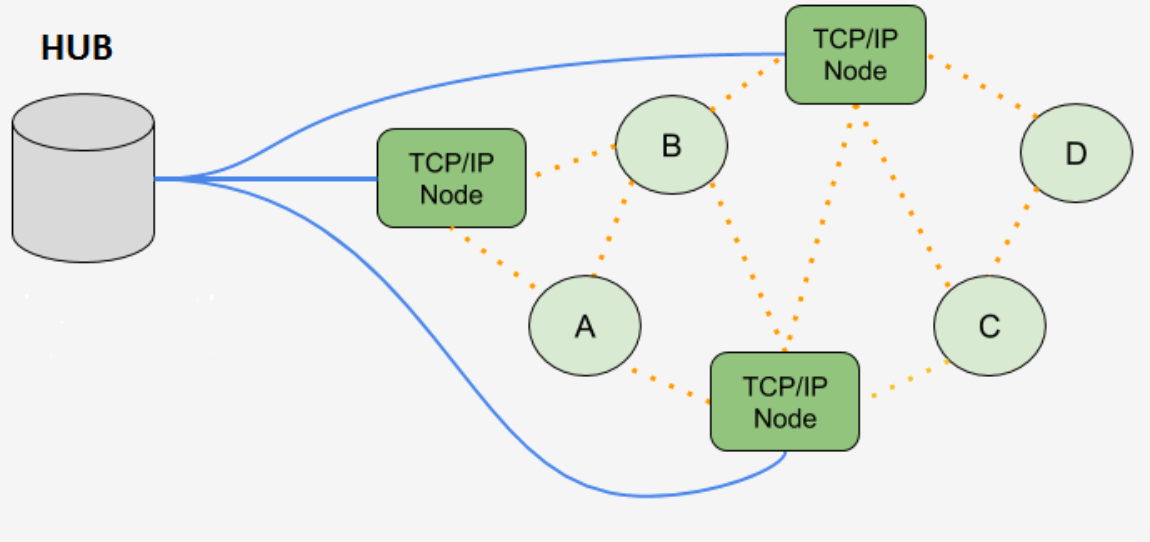
Tietoverkot on laaja konsepti, joka ei rajoitu vain tietokoneisiin, vaan voidaan yleisesti myös sanoa, että kaikki järjestelmät, joissa tapahtuu kommunikaatiota kahden tai useamman osapuolen toimesta, ovat tietoverkkoja (Davis, Turner & Yocom 2004, 19). Tietokoneiden kommunikoinnin mahdollistamiseen on useita eri teknologioita, mutta näistä yleisin on lähiverkko. Lähiverkolla tarkoitetaan tietoverkkoa, jossa vähintään kaksi tietokonetta on yhdistettynä toisiinsa suoraan, tai siihen tarkoitetun laitteen, kuten kytkimen, avulla. Erillisen laitteen käyttäminen tekee uusien tietokoneiden lisäämisestä tietoverkkoon yksinkertaisempaa ja mahdollistaa sen kasvun. Kun useampia tällainen verkko liitetään toisiinsa, muodostetaan teknologia nimeltään internet. (Davis ym. 2004, 21)

Nykyaikaiset tietoverkot on kaikki rakennettu tällä tavalla, ja konseptia kutsutaan tähti-topologiaksi. Tietoliikenne on keskitettyä niin konesaleissa kuin yksityisissäkin verkoissa. Esineiden internetkin on mukautunut suurimmalta osin noudattamaan tätä verkkotopologiaa, mutta ilmiö mahdollistaa myös vaihtoehtoisen ratkaisun. Sulautettujen laitteiden avulla myös suoritus on mahdollista hajauttaa eri laitteiden välillä. Tietoverkoissa tämä tarkoittaa P2P-arkkitehtuuria eli vertaisverkkoja (Levent-Levi 2015). Vertaisverkko tarkoittaa sitä, että kaikki verkon tietokoneet voivat toimia palvelimina ja asiakaskoneina itsenäisesti (Tietotekniikan termitalkoot a 2011).

P2P-verkon hyötyjä ovat sen yksityisyys, skaalautuvuus ja nopeus. Tietoliikenne pysyy esimerkiksi älytalosi sisällä, sensoreiden välisenä keskusteluna, jolloin tietoliikenteeseen kuluva aika on myös huomattavasti pienempi kuin internetin kautta. Selvänä heikkoutena on järjestelmän monimutkaisuus, sillä laitteiden täytyy keskenään tietää ja löytää toisensa jotenkin. Tällaista toteutusta voi olla vaikea hallita. (Levent-Levi 2015.)

Perinteisessä keskitetyssä mallissa hyötyjä sen tuttuuden lisäksi ovat hallinnan helppous ja se, että esineet voidaan jättää tyhmäksi sekä tiedon saatavuus yhdestä paikasta. Tieto pystytään näin hyödyntämään paremmin ja tehokkaammin eri palveluihin. Huonoja puolia mallissa on esineiden tietoliikenteen hidastuminen, koska se suoritetaan internetin ylitse. Palveluiden keskittäminen aiheuttaa usein huolia tietoturvasta. Kaikki tieto on saatavilla yhdestä paikasta myös mahdolliselle hyökkääjälle. (Levent-Levi 2015.)

Nämä teknologiat edustavat omaa ääripäätään hajautettuna ja keskitettynä ratkaisuna. Voimme yhdistää näiden parhaita puolia ja toteuttaa esineiden kommunikoinnin hybridimallilla, jossa liitämme paikalliseen sensoriverkkoon yhden keskitetyn viestintäpalvelimen. Tämä palvelin on internetin ja lähiverkon välissä kommunikoimassa tietoja internetiin, jossa niiden keräämää tietoa voidaan paremmin hyödyntää keskitetyillä ratkaisuilla, mutta myös varsinaiset sensorit pystyvät itsenäisesti kommunikoimaan keskenään. Tätä verkotusmallia sanotaan ”hub and spoke” -malliksi. Kuvassa 1 on verkotusmallin esimerkkitoteutus, jossa radion avulla kommunikoivia sulautettuja laitteita kuvataan ympyröillä ja TCP/IP-kommunikointiin kykeneviä laitteita nelikulmioilla. Laitteet kommunikoivat keskenään, ja TCP/IP-laitteet kommunikoivat hubin kanssa. (Levent-Levi 2015.)



Kuva 1. Hub and spoke -verkkotopologian esimerkki.

3 PILVIPALVELUT

Tietokoneet ovat viimeisten vuosikymmenien aikana menestyneet erittäin hyvin erilaisissa keskitetyissä ratkaisuissa, ja niiden kehitystä on ajanut tieteen ongelmat, joiden ratkaisemiseen tarvittiin laskentatehoa. Keskitettyä ympäristöä on vaikea hyödyntää maksimaalisesti, sillä palvelut usein tarvitsevat resursseja vain hetkellisesti ja näin merkittävä osa laskentavoimasta on joutilaana lopun aikaa. (Chandrasekaran, 21, 28.)

Ongelman ratkaisemiseksi kehitettiin tekniikoita, joiden avulla ylimääräistä tyhjäkäynnissä olevaa laskentakapasiteettia pystytään jakamaan toisille tarvitseville osapuolille saumattomasti, niin ettei se kuitenkaan haitannut alkuperäisen tarkoituksen toimintoja. Tällaista tietotekniikan resursseja paremmin hyödyntävää toteutusta kutsutaan ”grid computing” -malliksi, ja voidaan sanoa, että pilvipalvelut ovat tästä mallista kehittynyt, kaupallistettu versio. (Chandrasekaran, 28.)

Pilvipalvelut eroavat ”grid computing” -mallista siinä, että jälkeenpäin mainittu tarjoaa hajautettua rinnakkaislaskentakapasiteettiä tietyn ongelman ratkaisemiseen, missä taas pilvipalvelut hyödyntävät useita eri resursseja, esimerkiksi laskenta- ja tallennuskapasiteettia, tuottamaan loppukäyttäjälle yhtenäisen ja kokonaisen palvelun (Chandrasekaran, 29). Pilvilaskennan käsite voidaan jakaa neljään ominaisuuteen:

1. Kyky vastata asiakkaan tai käyttäjän tarpeisiin tarvittaessa ja mahdollistaa itsepalvelu.

2. Palveluiden laaja saatavuus. Ne toimivat monella eri alustalla ja käyttöönotto on standardisoitu.

3. Joustava, nopea ja keskeytymätön resurssien yhdistäminen ja skaalautuminen.

4. Resurssien automaattinen mittaaminen ja valvominen. (Chandrasekaran, 15.)

Jos näitä ominaisuuksia ei tueta tai niissä on jotain puutteita, ei voida puhua todellisesta pilvipalvelusta (Chandrasekaran, 15.)

Pilvi-termiä käytetään tietotekniikan kontekstissa kielikuvana internetille ja sen taakse verhoutuvalle infrastruktuurille. Kuitenkin yksinkertaisimmillaan se tarkoittaa tiedon hakemista ja tallentamista internetin ylitse etäsijaintiin. Tällä etäsijainnilla on useita erikoistuneita ominaisuuksia, kuten skaalautuvuus, joustavuus ja yhteensopivuus. (Chandrasekaran, 36.)

Pilvipalveluiden toteutuksen mallit usein jaetaan kolmeen eri malliin, jotka ovat SAAS, PAAS ja IAAS. Riippuen asiakkaiden tarpeista palveluita voidaan toteuttaa yksityisessä pilvessä, jossa infrastruktuuria tai palveluita tarjotaan vain yhden asiakkaan yksityiseen käyttöön. Asiakkaan organisaatio voi olla osittain vastuussa palveluiden ylläpitämisestä. Yksityisen pilven vastakohta on julkinen pilvi, jolloin pilvi on täysin palveluntarjoajan hallitsemissa ympäristöissä ja yleisesti saatavilla suurelle yleisölle. Kun näitä palveluita jakaa useampi organisaatio tai asiakas, puhutaan yhteisön pilvipalvelusta. Viimeinen eli hybridipilvi on jokin yhdistelmä näistä toteutuksista. (Chandrasekaran, 39.)

3.1 Tarve ja motivaatio pilvilaskennalle

Tärkeät pääasialliset syyt pilvipalveluiden suosiolle ovat niiden käytännöllisyys ja luotettavuus. Menneisyydessä, jos haluttiin siirtää tiedostoja tietokoneelta toiselle, jouduttiin ne ensin siirtämään jollekin fyysiselle medialle, kuten CD-levylle tai muistitikulle, ja sitten toimittamaan tiedostot manuaalisesti haluttuun paikkaan. Nykyaikaisilla pilvipalveluilla tiedostojen siirtäminen vaatii vain internetiin yhdistetyt tietokoneet. Myös tiedostojen menettäminen on erittäin epätodennäköistä. (Chandrasekaran, 36.)

Kääntöpuolena on huoli tietojen yksityisyydestä, sillä tietojen vieminen pilveen tarkoittaa, että loppukäyttäjä ei tiedä, missä ne fyysisesti ovat ja kuka niitä pystyy tarkastelemaan. Tietoturva ulkoistetaan suurelta osin pilvipalvelun tuottajalle. (Chandrasekaran, 37.)

Pilvipalvelut tarjoavat myös usein taloudellisemman ja ympäristöystävällisemmän ratkaisun. Organisaation omien tietokonelaitteistojen, sovellusten, verkotuksien ja energian hankkiminen on kallista, puhumattakaan niiden ylläpitämisestä. On siis yksinkertaisempaa ja halvempaa vuokrata nämä resurssit niihin erikoistuvalta ulkopuoliselta toimittajalta ja maksaa palveluista käyttötarpeiden mukaan. (Chandrasekaran, 34.)

3.2 SAAS

SAAS-termi tulee englannin kielen sanoista ”Software As a Service” ja on pilvipohjainen palveluntoteutusmalli. SAAS-malli on sovelluksien jakelumuoto ja käyttöpalvelu internetin ylitse. Palveluntarjoaja ylläpitää ja tuottaa palveluita pilvikapasiteetin avulla. Asiakkailla voi olla mahdollisuus konfiguroida sovelluskerros omien tarpeiden mukaisesti, mutta tämä on harvinaisempaa. Pilvipalveluiden skaalautuvat resurssit ovat tukena toiminnalle. (Chandrasekaran, 41.)

Nykyaikaiset pilvipalvelusovellukset toimivat käytännössä kaikki kevyen loppupäätteen päällä, kuten esimerkiksi nettiselaimella. Palveluita voivat esimerkiksi olla sähköposti ja projektinhallinta, ja usein tärkeät yritystoimintaa tukevat palvelut ovat suosituimmat. Palveluiden maksumallit ovat kuukausittainen tai käyttöön perustuva malli. Lisäksi palvelussa on usein palvelun laadusta ja ominaisuuksista riippuen, erilaisia tasoja joista kuluttaja voi halutessaan maksaa enemmän. (Glister 2014, 20.)

3.3 PAAS

”Platform As a Service” -pilvipalvelumalli mahdollistaa omien palveluiden rakentamisen ja niiden tai valmiiden palveluiden tarjoamisen käyttäen eri ohjelmointikieliä, kirjastoja ja työkaluja (Chandrasekaran, 41). Käytännössä PAAS-malli tarjoaa asiakkaalle virtuaalisia palvelimia ja käyttöjärjestelmiä, integroituja kehitysympäristöjä ja työkaluja. Malli tukee ohjelmistokehityksen testausta, käyttöönottoa ja hallintaa pilvipalveluilla. (Glister 2014, 21.)

3.4 IAAS

IAAS-termi tulee sanoista ”Infrastructure As a Service” ja tarkoittaa täydellisen tietokoneympäristön vuokraamista asiakkaan tarpeiden mukaan. IAAS-palvelu on hyödyllinen, jos kuluttaja ei halua rakennuttaa omaa infrastruktuuria ja maksaa jatkuvia kuluja oman tietokoneympäristön ylläpitämisestä (Glister 2014, 22). Asiakas tilaa palveluntarjoajalta haluamansa määrän laskenta-, tallennus-, verkotus- ja muita perusresursseja. Asiakas hallitsee resursseja sekä päällä ajettavia sovelluksia itsenäisesti. (Chandrasekaran, 41).

Suurin ero PAAS- ja IAAS-mallien välillä on hallinnan määrä. Mallit tarjoavat samat resurssit, mutta IAAS-mallissa asiakkaalla on suurempi vastuu hallinnasta. Yleisesti IAAS-malli on halutumpi, kun tarkoituksena ei ole kehittää uusia sovelluksia, vaan suorittaa valmista ohjelmistoa pilvessä. (Chandrasekaran, 41).

4 WORLD WIDE WEB

Internet on kehittynyt viime vuosikymmenien aikana ylivoimaiseksi palveluiden tarjonnan ja kulutuksen alustaksi. Sen läsnäolo ja yleisyys kaikkialla ovat yksinkertaisen arkkitehtuurin, ja sen toteutuksessa käytettyjen laajasti levenneiden teknologioiden, suora seuraus. Ne verkkopalvelut, jotka omaksuvat internetin yksinkertaiset ominaisuudet, nauttivat alustan skaalautuvuudesta, turvallisuudesta ja luotettavuudesta. (Webber, Paratatidis & Robinson 2010, 31.)

World Wide Web (jatkossa WWW) on informaation alusta ja palvelujärjestelmä internetin päällä (Tietotekniikan termitalkoot b, 2012). Tim Bernes-Lee suunnitteli WWW:n 1990-luvun alussa. Tarkoituksena oli kehittää helppokäyttöinen, hajautettu ja löyhästi linkitetty järjestelmä dokumenttien jakamiseen. Tuloksena oli alusta, jonka kanssa oli helppo julkaista ja kehittää eri sovelluksia sekä sisältöä. Muutama vuosi järjestelmän synnyn jälkeen se oli käytössä useassa akateemisessa sekä tieteellisessä laitoksessa niiden nettisivujen julkaisua varten. Kun liikemaailma huomasi alustan potentiaalin, teknologia kehittyi nopeasti ja nykyään WWW on sekoitus tieteellisiä, hallituksellisia, sosiaalisia ja yksityisiä palveluita. (Webber ym. 2010, 20.)

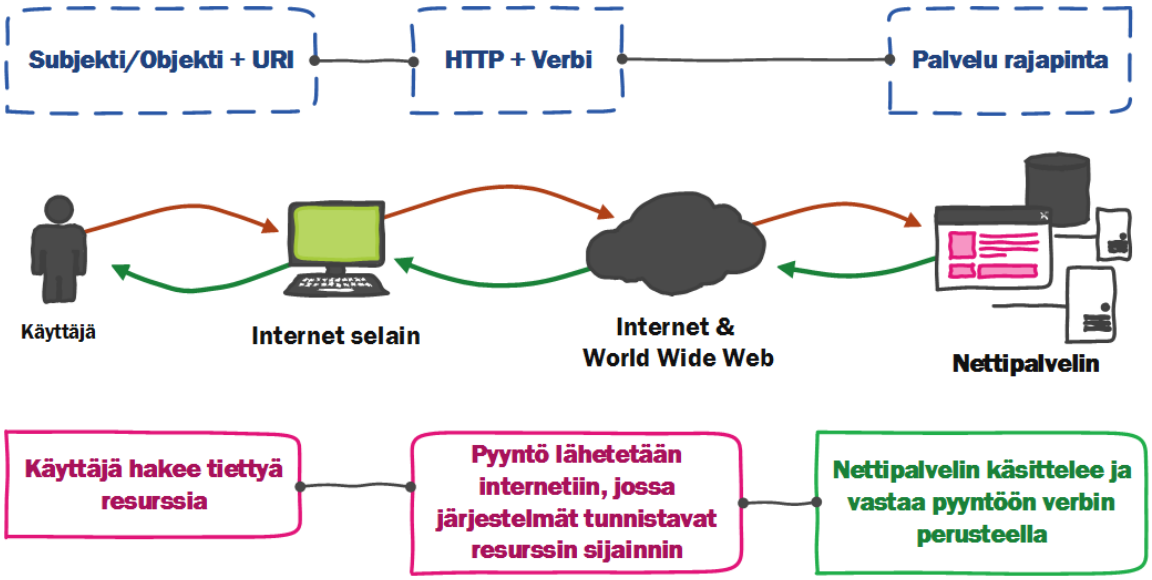
Jos nykyajan tilannetta katsotaan globaalissa mittakaavassa, niin tämä alusta vaikuttaa isolta informaation kaaokselta. Todellisuudessa kyseessä on kuitenkin kokoelma yksinkertaisia, pienen mittakaavan vuorovaikutuksia, ihmisten, ohjelmistojen ja resurssien välillä, jotka kaikki käyttävät internetin eri teknologioita. WWW koostuu laajalle levinneistä ja standardisoiduista palvelimista, jotka ylläpitävät resursseja sekä niiden käsittelyyn tarvittavaa laskentavoimaa, ja verkkoliikenteen mahdollistavista palveluista kuten välitys- ja välimuistipalvelimista. (Webber ym. 2010, 21).

Järjestelmässä mielenkiintoista tai haluttua informaatiota kutsutaan resurssiksi ja nämä ovat eroteltavissa yksilöivällä globaalilla URI-tunnuksella, joka on käytännössä vain resurssin sijainnin tai nimen kertova merkkijono. (Jacobs 2004.)

Resurssilla voidaan viitata esimerkiksi dokumenttiin, videotiedostoon, ohjelmisto-prosessiin tai laitteeseen, joka on asetettu näkyville internetiin. Useat fyysisen maailman resurssit voidaan tuoda internetiin saataville abstrahoimalla niihin liittyvät tiedot sekä ominaisuudet sopivaksi WWW:n palvelujärjestelmään. Esimerkiksi omena voi olla resurssi, mutta koska sen siirtäminen internetin ylitse ei vielä ole mahdollista, esitämme sen sijaan kyseisen omenan loogisen informaation ja nykyhetken tilan. Tämän konseptin yleispätevyys tarkoittaa, että kokonaisuus on heterogeeninen. Loppukäyttäjän näkökulmasta resurssi on vain jokin asia, jonka avulla he voivat edetä tavoitettaan kohti (Webber ym. 2010, 22).

Resurssin esittämisen muoto ei ole staattinen, ja WWW:n eri komponentit muokkaavatkin aina alkuperäistä määritelmää jotenkin ennen sen esittämistä palveluiden mahdollistamiseksi. Esimerkiksi eri käyttäjille resurssi täytyy esittää eri tavalla, riippuen heidän käyttöoikeuksista, mutta alkuperäinen resurssi pysyy kuitenkin ennallaan, kun URI-tunnukset yhdistävät ja assosioivat resurssin eri näkymiä eri palveluihin. (Webber ym. 2010, 8).

WWW on siis alusta, jossa ylläpidetään resursseja ja joiden kanssa haluamme olla vuorovaikutuksessa. Näitä resursseja kuvataan loogisina objekteina sekä subjekteina, mutta niiden käyttäminen on eri asia. HTTP on yksi monista tämän vuorovaikutuksen mahdollistavista protokollista. Siinä missä resurssit ovat subjekteja ja objekteja, WWW-protokollat ovat verbejä (Webber ym. 2010, 29). Kuvassa 2 on hahmotettuna prosessi, jossa käyttäjä hakee nettisivun WWW-palvelusta.



Kuva 2. Käyttäjä hakee nettisivun WWW-palvelujärjestelmästä.

4.1 Hypermedia

WWW-palvelualusta mahdollistaa monipuolisen median käyttämisen epälineaarisella tavalla. Tätä epälineaarisesti linkitettyä mediaa kutsutaan hypermediaksi ja se kuvaa WWW:n kaiken sisällön. Termiä käytetään yleisesti vain sovelluskehityksen yhteydessä, sillä puhekielessä muut termit, kuten vuorovaikutteinen media, kuvastavat samaa asiaa. (Huston 2016.)

Hypermedian tarkoitus on yhdistää WWW:n resursseja toisiinsa ja esittää niiden ominaisuuksia tietokoneen ymmärtämällä tavalla. Konsepti koostuu useasta eri teknologiasta, joten sillä ei ole yhtä standardia tai toteutusta. WWW-palvelussa asiakas löytää URL-tunnuksen avulla jonkin resurssin, mutta koska mahdollisia tunnuksia on käytännössä loputtomasti, asiakas tarvitsee jonkinlaisen tavan löytää resursseja ja niihin liittyviä resursseja. Hypermedian toteuttaminen ratkaisee tämän ongelman. (Amundsen & Richardson 2013, 45.)

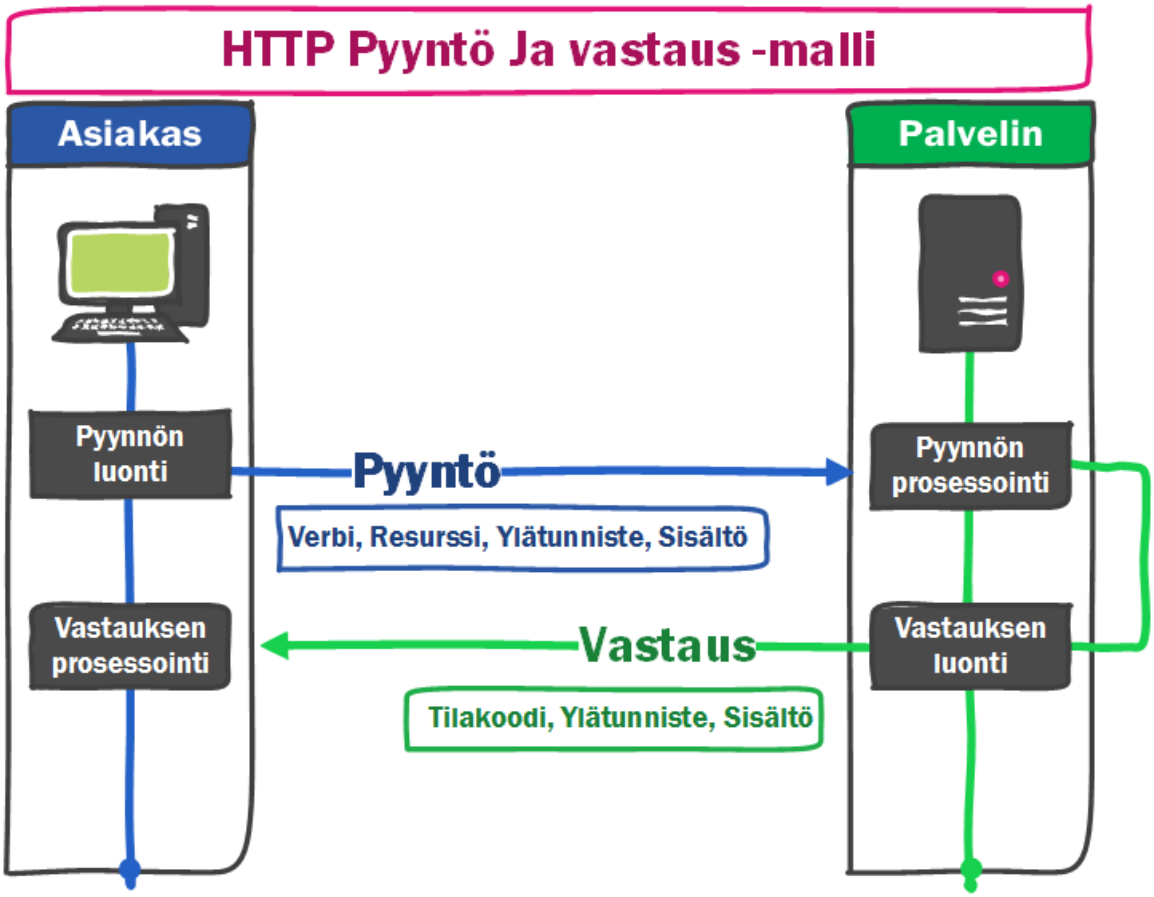
Ensimmäinen pyyntö tulee aina asiakkaalta palvelimelle. Tämän jälkeen palvelin voi arvata, mitä asiakas mahdollisesti tulee seuraavaksi haluamaan, ja dynaamisesti tai staattisesti tarjota linkkejä näihin resursseihin. Dynaaminen linkki muodostuu sovelluslogiikan ja käyttäjän vuorovaikutuksen perusteella. Staattinen

linkki on ennalta määritelty linkki, joka muodostettiin resurssin luonnin yhteydessä tai lisättiin siihen palvelimen toimesta. (Huston 2016.)

4.2 HTTP

HTTP on internetissä käytössä oleva sovelluskerroksen protokolla, jota käytetään esimerkiksi selaimen ja palvelimen kommunikoinnissa, kun halutaan hakea tietyn sivun sisältö. Protokolla on kehittynyt paljon tästä alkuperäisestä tarkoituksestaan, ja nykyään sitä käytetään esimerkiksi tietokoneiden väliseen kommunikointiin, automaatioon ja esineiden internetin mahdollistamiseen. (Waher 2015, 52.)

Protokolla noudattaa perinteistä pyyntö ja vastaus -palvelumallia (kuva 3). Asiakas lähettää pyynnön palvelimelle, joka vastaa siihen sen sisällön perusteella. Pyyntö koostuu metodista, resurssista, ylätunnisteista ja muista vapaammista optioista ja sisällöistä. Palvelimen vastaus taas koostuu kolmen numeron tilakoodista, ylätunnisteista ja valinnaisista vapaammista sisällöistä (Waher 2015, 53). Tilakoodit ovat joukko ennalta määrättyjä ja yhteisesti sovittuja numeroita, kuten esimerkiksi ”200 OK”, ”201 Created” ja ”404 Not Found”. Jokaisella koodilla on tietty merkitys, ja sillä kerrotaan pyynnön tilanne tai lopputulos sen asettajalle. (Webber ym. 2010, 29.)



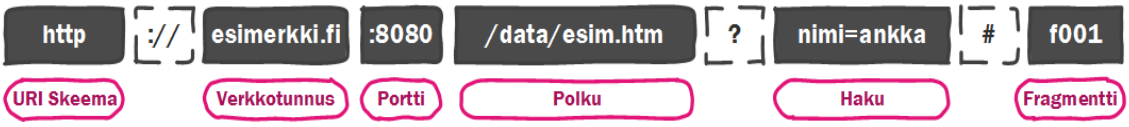
Kuva 3. HTTP-protokollan pyyntö ja vastaus -malli.

Ylätunnisteet sisältävät metatietoa verkon ylitse lähetettävästä pyynnöstä ja sitä mahdollisesti seuraavasta vastauksesta. Nämä ovat usein ihmiselle luettavassa muodossa olevia avain-arvopareina. Tämä tieto voi esimerkiksi koskea sisällön tyyliä, sisältöön liittyviä koodekkeja ja tiedon elinikää. (Waher 2015, 53).

Protokollan virallisessa standardissa, jossa määritellään sen tekniset sekä ideallisset puitteet, käytetään termiä metodi, mutta näille on toinen yleisempi termi: verbi. Verbit kuvaavat protokollan eri toimintoja, esimerkiksi GET-verbiä käytettäessä haetaan URI-tunnuksen määrittämää resurssia ja POST-verbillä taas halutaan lähettää jotain tietoa palvelimelle. Tällöin pyyntö sisältää myös lähetettävän tiedon. (Webber ym. 2010, 29.)

URI-tunnus on tunnuksien yläkäsite, ja kun haemme esimerkiksi nettisivua, käytämme ”Universal Resource Locator”- eli URL -tunnusta. Kaikki internetissä resurssin tunnistavat osoitteet ovat URL-tunnuksia. HTTP-protokollan URL-skeema muodostuu omistajan DNS-nimestä tai IP-osoitteesta ja palvelun sovellusportista,

joita seuraa resurssin tarkka polku. Polun jälkeen voidaan sisällyttää tietoa, esimerkiksi käyttäjän tekemästä hakuoperaatiosta, joka erotetaan kysymysmerkillä polusta. Vapaavalintainen risuaitasymboli erottelee fragmentti-tunnuksen, joka on aina viimeisenä, ja viittaa saman nimiseen elementtiin resurssissa tai nettisivussa. (kuva 4). (Waher 2015, 53.)



Kuva 4. URL-tunnuksen rakenne.

4.3 REST

Roy Fieldings yleisti WWW:n arkkitehtuurin periaatteet ja esitti tuloksensa väitöskirjassaan vuonna 2000. Tämän työn tuloksena syntyi REST-arkkitehtuurityyli, jonka ideana on mahdollistaa yhteentoimivuus hajautetuissa järjestelmissä, joissa eri osapuolet voivat kehittyä sekä muuttua itsenäisesti toisistaan riippumatta. Työssään hän esimerkiksi kuvasi WWW:n hajautettuna hypermediasovelluksena, johon linkitettyjen resurssien eri tiloja ja muotoja kommunikoidaan. Tämän seurauksena WWW kehittyi ja otettiin käyttöön sovellusalustana. (Webber ym. 2010, 30).

REST-arkkitehtuuri voidaan määritellä palveluiden ja rajapintojen kehitykseen asetettujen rajoitusten perusteella. Seuraavaksi käydään läpi muutama tärkein arkkitehtuurinen pääpiirre, jotka ohjaavat toteutusta.

Yhtäläinen rajapinta

Yhtäläisen rajapinnan tarkoitus on rajata kaikki palvelut ja palvelua käyttävät sovellukset yhden teknisen rajapinnan käyttöön. Rajapinnan täytyy olla sopiva erilaisiin palveluihin ja toteuttaa yleispäteviä ominaisuuksia. Yleisesti rajapinta muodostuu HTTP-protokollan verbeistä sekä muista yleisesti hyväksytyistä semantiikoista. (Webber ym. 2010, 30).

Käytännössä yhtäläinen rajapinta ja sen toteutus mahdollistavat järjestelmän komponenttien yksittäisen kehittymisen erottamalla ne toisistaan, mutta silti mahdollistaen niiden kommunikoinnin. Tämän haittapuolena on kommunikaation tehokkuuden alentuminen, sillä se täytyy standardisoida ja yleistää, joten sitä ei voida optimoida sovelluksen tarpeiden mukaisesti. (Amundsen ym. 2013, 382.)

Tilattomuus

REST-arkkitehtuuri suosittelee tilattoman kommunikoinnin asiakkaan ja palvelimen välillä. Asiakkaan resurssipyynnöt palvelimelle sisältävät itsessään kaiken tarvittavan tiedon pyynnön ymmärtämiseen ja käsittelyyn. Palvelun operaatiot eivät tarvitse mitään tallennettua tietoa asiakkaasta, tai kommunikaatiosta sen välillä, toimiakseen halutulla tavalla. Asiakkaan tila annettaan aina varsinaisessa pyynnössä ja tämä kasvattaa viestien kokoa, hidastaen kommunikaatiota. (Doglio 2015, 22.)

Tilattomuus on yksi arkkitehtuurin tärkeimmistä rajoituksista. Se tekee järjestelmistä helposti monitoroitavia, kun kaikki tieto on varsinaisessa pyynnössä. Kun tietoa ei tarvitse tallentaa pyyntöjen välissä, vapautuu kapasiteettia nopeammin asiakkaiden käsittelyyn ja täten palvelinpuolen skaalautuvuus on parempi. Ohjelmiston kehittäminen hajautettuun ympäristöön on helpompaa, koska tilatietoja ei tarvitse hallita ja jakaa useamman eri palvelimen kanssa. Samasta syystä myös vikasietoisuus on järjestelmässä parempi. (Doglio 2015, 22.)

Tilattomuudessa on yksi selkeä haavoittuvuus turvallisuuden suhteen, koska palvelinjärjestelmät luottavat asiakkaan pyyntöön ja olettavat sen sisältävän hyödyllistä tietoa. Yleinen ja helppo hyökkäys tällaisiin järjestelmiin on kirjoittaa omaa haitallista koodia pyyntöön mukaan niin, että palvelinjärjestelmät tulkitsevat sen normaalina ohjeistuksena olettaen, että se liittyy jotenkin palvelun resursseihin. (Doglio 2015, 22.)

Resurssien tunnistaminen

Klassiset hypertekstijärjestelmät käyttävät yksilöllistä tunnusta resurssien yksilöimiseen, joka vaihtuu aina kun, varsinainen tieto vaihtuu. Tämä teknologia nojaa erillisiin järjestelmiin, jotka ylläpitävät tunnuksien varsinaisia viittauksia. On vaikea kuvitella tällaista järjestelmää nykyaikana, sillä se on ongelmallinen tapa tunnistaa resurssit WWW -palvelualustassa laajan skaalan sekä monimutkaisuuden vuoksi. (Amundsen ym. 2013, 377.)

REST-arkkitehtuurissa resurssin tila voi vaihtua, mutta sen yksilöivä tunnus pysyy samana. Jos tunnus vaihtuu, niin palvelin voi hyödyntää hypermediajärjestelmiä pyynnön uudelleen ohjaamiseen. (Amundsen ym. 2013, 378.)

4.4 COAP

COAP on protokolla, joka on suunniteltu sulautettujen laitteiden kommunikointiin. Nämä laitteet ovat usein erittäin rajoittuneita verkon, virran ja laskentatehon suhteen, joten tarvitaan erikoistuneempi ja siihen ympäristöön optimoitu protokolla. COAP mukailee HTTP-protokollaa, tarkoittaen, että sitä voidaan käyttää REST-arkkitehtuurin mukaisesti hypermedian julkaisemiseen. (Amundsen ym. 2013, 287.)

Teknisesti protokollien välillä on paljon eroa. Pyyntöjen ja vastauksien koko on todella pieni COAP-protokollassa. Esimerkiksi vähävirtaisessa langattomassa internetyhteydessä yhden viestipaketin koko ei saa olla yli 80 tavua (Amundsen ym. 2013, 287).

Suurin tekninen ero näiden protokollien välillä löytyy varsinaisesta tiedonsiirtoon käytetystä protokollasta. HTTP ja COAP ovat sovelluskerroksen protokollia, mutta siinä missä HTTP on rakennettu TCP-tiedonsiirtoprotokollan päälle, COAP on rakennettu UDP -tiedonsiirtoprotokollan päälle. TCP-protokolla tukee yhteyksien muodostamista, mikä tarkoittaa sitä, että kun asiakas lähettää pyynnön palvelimelle, jää se odottamaan palvelimen vastausta ja pakettien perille saapuminen

varmistetaan. UDP-protokolla taas lähettää paketin ja jatkaa pakettien lähettämistä, varmistamatta pakettien perille saapumista. (Amundsen ym. 2013, 288.)

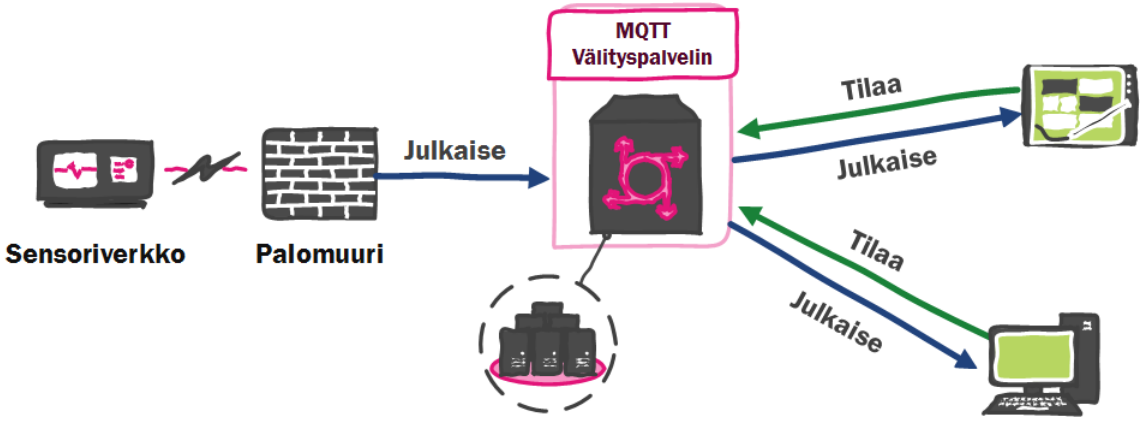
Koska kommunikaation alussa ei tehdä erillistä kättelyä, niin COAP-protokolla on paljon kevyempi, mutta varmuus tiedonsiirrosta ei ole taattu. COAP pystyy sovellusratkaisujen avulla kuitenkin ovelasti emuloimaan tiettyjä tiedonvarmuutta parantavia ominaisuuksia. Lisäksi sulautetuissa ympäristöissä tärkeintä ei välttämättä ole taata täydellistä viestinnän eheyttä. (Amundsen ym. 2013, 288.)

Verkotukseltaan COAP muistuttaa normaalia WWW-palvelualustaa, sillä sulautetut laitteet voivat olla täysin eri valmistajilta, eikä niiden tarvitse olla linkitettynä toisiinsa. Laitteiden täytyy kuitenkin pystyä löytämään ja ymmärtämään toisia laitteita verkon ylitse ilman ihmisen erillistä apua. (Amundsen ym. 2013, 287.)

5 MQTT

MQTT on julkaise ja tilaa -protokolla, joka on suunniteltu rajallisten resurssien laiteympäristöihin. Se rakennettiin ottamaan huomioon seuraavat ominaisuudet: helppo käyttöönotto ja toteutus, mukautuva tiedoneheyden varmistus, kevyt ja kaistatehokas kommunikointi, yleinen sopivuus eri järjestelmiin sekä jatkuva tilan ylläpitäminen. Protokolla on binäärinen, joka tarkoittaa, että tieto siirretään konekielellä ja sen pakettikoko on todella pieni. Nämä ominaisuudet palvelevat sulautettuja järjestelmiä hyvin. (MQTT Essentials: Part 1 2015.)

Toisin kuin HTTP- ja COAP-protokollissa, MQTT-protokollassa ei luoda pyyntöjä ja vastauksia, asiakkaan ja palvelimen puolesta, vaan sen sijaan käytetään tapahtumapohjaista julkaise ja tilaa -mallia. Tällaisessa mallissa on kolme erillistä osaa: julkaisia, tilaaja ja viestintäpalvelin. Julkaisijan rooli on muodostaa yhteys viestintäpalvelimeen ja julkaista tietoa. Tilaajan rooli on ottaa yhteys samaiseen viestintäpalvelimeen ja tilata sieltä haluamansa tiedot. Kolmas komponentti eli viestintäpalvelin on vastuussa viestien välittämisestä julkaisijoiden ja tilaajien välillä. Kuva 5 hahmottaa MQTT-protokollan toteutuksen rakenteen. Sensoriverkko julkaisee tiettyyn aiheotsikkoon tietoa yksityisestä verkosta ja välityspalvelin jakaa sen sitä tilaaville. (Waher 2015, 125.)



Kuva 5. MQTT-protokollan toteutuksen topologia.

Julkaise ja tilaa -malli erottaa viestin lähettäjän sen vastaanottajasta, mahdollistaen näiden erillisen kehittymisen ja kasvamisen. Tarkoittaa sitä, että lähettäjä ja vastaanottaja eivät ole tietoisia toisistaan, joten niiden ei myöskään tarvitse toimia

samassa verkossa, tilassa tai edes yhtäaikaisesti. Tämä toteutus mahdollistaa palveluiden paremman skaalautuvuuden, sillä keskitetty viestintäpalvelin voi esimerkiksi toimia hajautettuna pilvessä. (MQTT Essentials: Part 2 2015.)

5.1 Viestien hallinta

MQTT-protokolla jakaa ja tunnistaa sisältöä julkaisijan asettaman tekstipohjaisen aiheotsikon mukaan. Näiden aiheiden arvot toimivat hyvin samantapaisesti kuin esimerkiksi tietokoneiden tiedostojärjestelmät. Ne muodostavat hierarkkisen puurakenteen, jossa kauttaviiva merkitsee uuden haaran alkua. Tilaajat voivat muodostaa viestien tilaukset täsmällisesti johonkin tiettyyn haaraan, tai ne voivat käyttää erilaisia jokerimerkkejä tilatakseen useampia eri aiheita ja niiden arvoja. (Waher 2015, 125.)

Aihearvojen rakenteelle ei ole standardisoitua tapaa, mutta on hyvin yleistä käyttää jokaisen julkaisevan ja lähettävän laitteen kohdalla ”inTopic”- ja ”outTopic”-arvoja, jotka määräävät sisäänpäin ja ulospäin menevät yhteydet julkaisijan näkökulmasta. Lisäksi välilyönnit sekä isot ja pienet kirjaimet ovat merkitseviä. (MQTT Essentials: Part 5 2015.)

Kuvassa 6 on esimerkki monitasoisesta aihearvosta, jonka tarkoitus on tilata laitteelle tarkoitettua viestejä välityspalvelimelta. Aiheotsikon tasoista voidaan usein päätellä laitteen fyysinen sijainti sekä sen nimi, mutta aiheotsikoilla ei ole suoraa, taattua korrelaatiota oikeiden paikkojen tai nimien kanssa. Sijaintia kuvaavien aihetasojen tilalla voi olla mitä vain. (MQTT Essentials: Part 5 2015.)

Kuva 6. Esimerkki MQTT-protokollan aiheotsikosta ja sen rakenteesta.

5.2 Yhteys

Yhteys muodostetaan MQTT-protokollassa aina asiakkaan ja viestintäpalvelimen välille. Tätä ylläpidetään, kunnes asiakas erikseen lähettää sen katkaisuun viittaavan viestin tai muuten häviää verkosta. Kyseessä on sovelluskerroksen protokolla, ja se rakentuu TCP-protokollan päälle. Liikennettä on mahdollista suorittaa SSL/TLS -suojauksen avulla, jolloin käytetään 8883-porttia tai suojaamattomana 1883-portilla. (MQTT Essentials: Part 3 2015.)

Yleisesti kaikki välityspalvelimen asiakkaat ovat yksityisessä verkossa jonkin reitittimen takana. Tämä ei kuitenkaan ole ongelma, sillä yhteys aloitetaan aina asiakkaan toimesta, välityspalvelimella on julkinen osoite ja tätä yhteyttä ylläpidetään. Yhteyden avaaminen tehdään lähettämällä CONNECT-paketti viestintäpalvelimelle. Paketin muokattavissa olevat arvot ovat kuvattuna taulukossa 1. (MQTT Essentials: Part 3 2015.)

Taulukko 1. MQTT-protokollan CONNECT-paketti (MQTT Essentials: Part 3 2015).

Arvo	**Kuvaus**
ClientID	Yksittäisen asiakkaan muista erottava nimi. Tämä arvo on pakollinen ja sen on oltava yksilöivä.
CleanSession	Totuusarvo, joka määrää ovatko asiakkaan tilaukset pysyviä. Jos arvo on totta, tilauksia ei tallenneta ja myös aikaisempien yhteyksien tiedot tuhotaan.
KeepAlive	Aikaväli, jota käytetään ajoittamaan välityspalvelimen toimesta asiakaslaitteen saatavuuden tarkistamiseen.
QoS	Numeroarvo, joka määrää viestiliikenteen varmennuksen tason.
Username / Password	Mahdolliset käyttäjätunnukset.
LastWillTopic	Julkaisun aihe -arvo, jota palvelin käyttää, kun yhteys katkeaa tahattomasti.
LastWillQoS	Palvelun laatu -arvo, jota palvelin käyttää, kun yhteys katkeaa tahattomasti.
LastWillMessage	Viimeinen viesti, jonka palvelin julkaisee asiakkaan puolesta, kun yhteys katkeaa tahattomasti.

”Quality of Service” eli QoS-arvolla säädetään tiedonsiirron eheyttä, ja se voi olla yksi seuraavista arvoista ja niiden tarkoituksista:

- 0, jolloin minkäänlaisia varmistuksia tai kuittauksia viestien saapumisesta ei suoriteta takaisin julkaisijalle,
- 1, jolloin välittäjä kuittaa paketit niiden lähettäjälle,
- 2, jolloin välityspalvelin kuittaa viestin saapuessa sekä sitä lähettäessä alkuperäiselle julkaisijalle. (MQTT Essentials: Part 3 2015.)

Käyttäjätunnukset kulkevat protokollassa alkuperäisessä muodossaan, jolloin ne ovat erittäin haavoittuvaisia erilaisiin kaappauksiin. SSL/TSL -suojauksen käyttö kryptaa koko paketin, joten se on suositeltu tapa korjata tämä mahdollinen haavoittuvuus. (MQTT Essentials: Part 3 2015.)

6 PALVELINOHJELMOINTI

Palvelinohjelmoinnilla viitataan siihen ohjelmointityöhön, joka on tarpeellinen pilvipalveluiden ylläpitämiseen. Palvelinohjelmointi voidaan jakaa kahteen kategoriaan: palvelimia ja sitä tukevan infrastruktuurin ylläpitäminen ja automaatio, sekä varsinaisten palveluiden tuottaminen erilaisilla ohjelmointikielillä ja työkaluilla. Tätä kokonaisuutta kutsutaan myös backend-ohjelmoinniksi, ja termi viittaa palveluiden taustalla toimivien palvelimien ohjelmointiin. (Long 2012.)

Palveluita voidaan toteuttaa kaikilla ohjelmointikielillä, jotka tukevat kommunikointia jollain tavalla kahden sovelluksen välillä. Tällä ohjelmointikielellä, ja siitä luotavilla palveluilla, ei tarvitse olla mitään korrelaatiota sitä kuluttavien ohjelmointikielien kanssa. Tämä mahdollistetaan yhteisen standardisoidun kommunikointiprotokollan, kuten HTTP-protokollan avulla. Sovellusten ohjelmointia varten on palvelinpuolelle kehittynyt siihen erikoistuneita ohjelmointikieliä. (DuVander 2013.)

6.1 Palveluiden toteutuksen suunnittelu

Palveluiden ohjelmoinnin suunnittelussa täytyy huomioida useita asioita. Ensimmäisenä valitaan varsinaiseen kommunikointiin käytettävä siirtokerroksenprotokolla. Valinta ei välttämättä ole yksiselitteinen, sillä mikään ei estä palvelua hyödyntämästä useampia protokollia yhdessä sovelluksessa. Tämä kuitenkin monimutkaistaa toteutusta, ja siksi täytyy osata valita oikeat työkalut oikeaan tehtävään. (Davis ym. 2004, 170.)

Seuraavaksi suunnitellaan sovelluskerroksen kommunikointi. Oman sovelluskerrosprotokollan kehittäminen voi olla tarpeellista, jos laitteistot ja alustat ovat niin erikoistuneita, että se nähdään tarpeelliseksi. Usein kuitenkin valitaan yleinen ja laajasti käytössä oleva protokolla, kuten HTTP tai COAP, mutta valinta tarkoittaa, että on ymmärrettävä sen toiminnallisuus ja toteutettava se asetetun standartin mukaisesti. (Davis ym. 2004, 169.)

Sovelluskerroksen protokolla voi kommunikoida binäärisenä tai tekstipohjaisena, joilla kummallakin on omat hyvät ja huonot puolet. Tekstipohjaista ja ihmiselle luettavassa muodossa olevaa kommunikointia on helpompi korjata ja kehittää. Se on myös yleispätevä lähes kaikille alustoille, jonka vuoksi suurin osa internetin protokollista ovat tällaisia. Binäärisen muodon hyödyt tulevat esiin sen nopeassa ja kevyessä liikennöinnissä, sillä yksittäisten viestien koko on pienempi ja kommunikoivat osapuolet eivät tarvitse kykyä käsitellä tekstipohjaista dataa. (Davis ym. 2004, 170.)

Viimeiseksi päätetään missä ja miten palvelun logiikka sekä tieto tullaan käsittelemään ja tallentamaan. Tämä määrää palvelun palvelin ja asiakas -arkkitehtuurin. Perinteisesti toteutukset rakennettiin kaksitasoisella arkkitehtuurilla, joka koostuu asiakassovelluksesta ja yhdestä palvelimesta. Tässä mallissa suurin osa sovelluksen logiikasta on asiakas-sovelluksessa ja palvelin toimii vain yksinkertaisena resurssien ja tiedon antajana. Hyvä puoli on toteutuksen yksinkertaisuus, mutta skaalautuvuus muodostuu nopeasti ongelmaksi. Lisäksi aina, kun palvelun logiikka muuttuu jotenkin, täytyy uusi asiakas-sovellus jakaa käyttäjille uudestaan. (Davis ym. 2004, 172.)

Suurin osa sovelluksista tänä päivänä toteuttaa kolmen tason arkkitehtuuria, joka on luonnollinen seuraaja kaksitasoiselle arkkitehtuurille. Asiakkaan ja palvelimen lisäksi lisätään kolmas sovelluskomponentti, jossa palvelun logiikka sijaitsee. Näyttökerros eli asiakas-sovellus pyytää tietoa sovelluskerrokselta ja tämä tekee hakuoperaatiot varsinaiseen tietokantaan, ja tuottaa pyynnön ja resurssin perusteella palvelun. Erottamalla palvelua tuottavan logiikan omaksi komponentiksi, palvelut voivat toimia ja kehittyä paljon joustavammin ja skaalautuvuus on helpompaa. Lisäksi päivitysten yhteydessä asiakas-sovellusta ei välttämättä tarvitse aina muuttaa tai jakaa uudestaan. (Davis ym. 2004, 171.)

Siirtyminen neljän tason arkkitehtuurin on jo alkanut suurempien palvelutuottajien toimesta. Kolmentason arkkitehtuuri suunniteltiin, kun ainoa asiakas-sovellus palveluilla oli nettiselain. Mobiililaitteet ja esineiden internet tarjoavat aivan uuden haasteen, joka vaatii sovelluksilta kykyä erikoistua ja sopeutua moneen eri laitteeseen ja tarkoitukseen. Ongelma ei ole tasojen määrässä, vaan ohjelman tuottamisessa ja yhdistyneessä rakenteessa. (Nommensen 2015.)

Yhtenäinen sovellus sisällyttää kaikki sen tuottamat palvelut yhteen prosessiin ja skaalautuu monistamalla tätä prosessia useammalle palvelimelle. Tämä tekee palveluiden kehittämisestä tarpeettoman vaikean ja joustamattoman. Vaikka palvelua hajautetaankin modulaarisesti, pienetkin muutokset siihen tarkoittavat sen kääntämistä kokonaan uudestaan lähdekoodista, ja tämä uusi sovellus täytyy testata, ennen kuin se on valmis tuotantoon. Palveluiden täytyy siis toimia erillisinä prosesseina, jolloin niiden skaalautuminenkin on helpompaa, ja kyky mukautua asiakkaiden tarpeisiin on parempi. (Nommensen 2015.)

Ohjelmistojen hajottaminen ominaisuuksien perusteella modulaariseen rakenteeseen on pitkään ymmärretty hyväksi käytänteeksi, mutta vasta viime vuosien aikana erilaiset avoimen lähdekoodin teknologiat ovat omaksuneet tämän ominaisuuden kunnolla. (Nommensen 2015.)

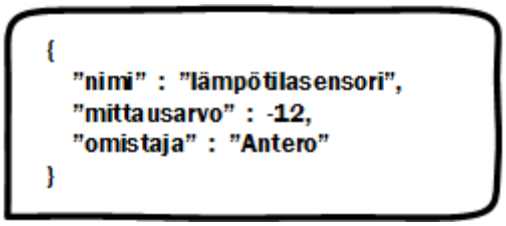
Modulaaristen palveluiden lisäksi tässä arkkitehtuurimallissa käytetään erillistä bisness- ja toimitustasoa, eli sovelluskerros jaetaan kahteen. Toimitustaso on vastuussa palveluiden toimittamisesta käyttäjille saatavilla olevien tietojen perusteella. Bisness-taso on ohjelmistototeutus, joka toimii keskuksena sisäisien ja ulkoisien palveluiden välillä, hyödyntäen reaaliaikaista kommunikaatiota. (Nommensen 2015.)

6.2 URI-tunnuksien ja resurssien suunnittelu

Internetin palveluiden kehityksessä täytyy aina huomioida ja suunnitella URI-tunnuksien rakenne ja toiminnollisuus. Yleisesti resurssit voidaan kuvata hierarkkisesti niiden välisten suhteiden perusteella, ja perusperiaate on, että kaikki resurssia tunnistavat URI-tunnukset ovat intuitiivisia ja mahdollisimman yksinkertaisia. (Foran 2014, 15.)

Jokaisella resurssilla on jonkinlainen muoto palvelussa, ja tätä muotoa esitetään tietyllä tavalla. Yleisimmät resurssien esitystavat ovat JSON- ja XML-tiedostomuodot. Sovellus voidaan ohjelmoida tukemaan useampaa esitysmuotoa, täten mahdollistaen tuen useammille asiakas-sovelluksille. Kun yhteys muodostetaan asiak-

kaalta palvelimelle, voi asiakas-sovellus voi ensin kommunikoida haluamansa tiedon esitystavan, tai palvelusovellus voi päätellä esitystavan lähetetyn pyynnön perusteella. Seuraavassa kuvassa on esimerkki JSON-tietomuodosta (kuva 7), joka kuvaa lämpötilaa mittaavan sensorilaitteen. (Pinna & Pintus 2016, 5.)



Kuva 7. JSON-tiedosto, joka kuvaa sensorilaitteen.

Resurssille yleensä annettaan erillinen ID-arvo ominaisuus, jonka avulla se voidaan yksilöidä muista resursseista. Samankaltaiset resurssit järjestetään erilaisiin listoihin, joista niitä on helpompi käsitellä. Resurssien nimet eivät saa sisältää verbejä, koska HTTP-protokolla varaa ne toimintoihin, ja kokoelmien nimet tulee olla monikossa, niin ne ovat helposti tunnistettavissa (Pinna ym. 2016, 6.)

Palvelun URL-tunnuksen tulee olla yksinkertainen, luettava ja helppo ymmärtää. Se heijastaa sen tunnistavan bisneksen konseptia ja alustaa. Tunnuksen alkuun on hyvä sisällyttää sovelluksen versionumero. Tämä käytäntö mahdollistaa useamman palveluversion ylläpitämisen, samassa asiakkaille tutussa, URL-osoitteessa. Pelkän toimialuenimen käyttäminen palvelun tarjoamiseen rajoittaa ja vaikeuttaa muiden palveluiden tarjoamista samasta toimialueesta. (Pinna ym. 2016, 8.)

6.3 Tietotekniikan automatisointi

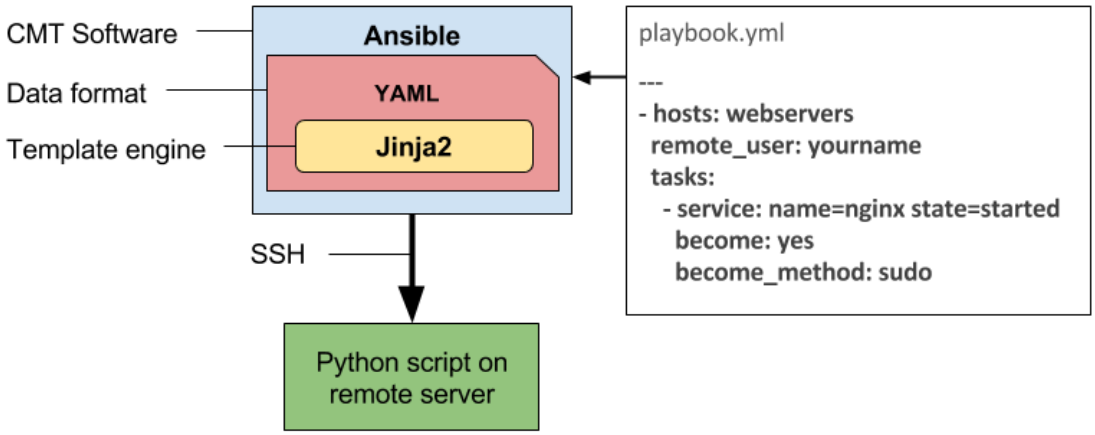
Tietotekniikan automaatio on ohjelmistojen ja palvelimien linkittämistä niin, että ne ovat itseään hallitsevia tai säätäviä. Automaatiota toteutetaan eri ohjelmilla, skripteillä tai muilla sovellusratkaisuilla. Se soveltuu moneen asiaan tietotekniikassa, ja muutamia esimerkkejä ovat uusien tuotantoympäristöjen pystyttäminen tai muuttaminen, auditointi, sovelluskehitys ja dokumentointi. Automaatio lupaa

suuren sijoitetun pääoman tuoton, mutta prosessin käyttöönotto ei välttämättä tule halvaksi tai tapahdu nopeasti. (Raghavenderrao 2015).

Palveluiden ja palvelimien tuottamisessa ja ylläpitämisessä tehdään paljon manuaalista ja toistuvaa työtä, jonka automatisoiminen vapauttaa resursseja varsinaisen bisneksen sekä palveluiden kehittämiseen. Automaatio nopeuttaa palveluiden tuottamista, ja pakottaa yhtäläisyyden testaamiseen ja eri tuotantoympäristöihin. (Raghavenderrao 2015).

Ansible

Ansible on yksinkertainen ATK-järjestelmien automaatio-, asennus- ja konfigurointisovellus. Sen toiminta perustuu ensin haluttujen toimintojen kuvaamiseen YAML-syntaksilla, joista sitten generoidaan Python-ohjelmointikielen sovellus. Tämä sovellus suoritetaan etäyhteyden avulla kohteisiin (kuva 8). (Hochstein 2015, 24.)



Kuva 8. Ansible ohjelmiston toiminta.

Ansiblen yksinkertaisuus tulee sen käyttämistä laajasti hyväksytyistä ja levinneistä teknologioista, sekä YAML-tiedostomuodosta, joka on suunniteltu erityisesti sen luettavuus mielessä. Etäyhteyden muodostamiseen käytetään SSH-protokollaa, jonka turvallisuus ja luotettavuus on todettu monen vuoden kehitystyöllä ja aktiivisella käytöllä. SSH-protokolla onkin yleisimmissä palvelinkäyttöjärjestelmissä tu-

ettuna oletuksena. Tämä tarkoittaa, että varsinaisille hallittavilla palvelimille ei välttämättä tarvitse asentaa mitään erillistä sovellusta. Nämä ominaisuudet erottavat ansiblen muista automaatiojärjestelmistä. (Hochstein 2015, 25.)

6.4 Node.js

Node.js on yksi suosituimmista JavaScript-pohjaisista teknologioista. Ryan Dahl loi ohjelmointialustan vuonna 2009 ja tämä mahdollisti JavaScript-ohjelmoinnin palvelinpuolella. Node.js-kehitysalustan kanssa käytetään ”npm” pakettienhallintajärjestelmää, jonka avulla kehittäjät jakavat hyödyllisiä ohjelmistomoduuleja toisilleen. Valmiiden moduulien hyödyntäminen onkin yksi Node.js-kehitysalustan suurimmista eduista, sillä kehitystyö on paljon nopeampaa niiden avulla. (Tsonev 2015, 1.)

Node.js-kehitysalusta suorittaa kaiken ohjelmakoodin yksisäikeisesti. Palveluiden skaalautuminen toteutetaan käynnistämällä useampi prosessi jokaiselle säikeelle tietokoneen keskusyksikössä, ja käyttämällä erillistä kuormantasaajaa jakamaan liikenteen näiden välillä. Suorituksessa käytetään tapahtumapohjaista silmukkaa, jonka toiminta perustuu Google-yhtiön kehittämään avoimen lähdekoodin JavaScript-moottoriin nimeltä ”V8”. (Tsonev 2015, 2.)

Express

Express on Node.js-kehitysalustalla toteutettu kirjasto WWW-palveluiden rakentamiseen. Kirjaston toiminnan takana on toinen kirjasto nimeltä ”connect”, joka mahdollistaa palveluiden helpon kehittämisen sen yksinkertaisella rakenteella ja työkaluilla.

Express tarjoaa yhdistyneen järjestelmän, jonka avulla on mahdollista hyödyntää lähes mitä tahansa HTML-merkintäkielen mallinnus työkaluja. Lisäksi järjestelmä mahdollistaa yksinkertaisten apuohjelmien käytön esimerkiksi eri tiedostomuotojen käsittelyyn, tiedostojen siirtämiseen ja URL-tunnuksien reitittämiseen. (Cantelon, Harter, Holowaychuk & Rajlich 2013, 176.)

Yleinen periaate Express-kirjaston toiminnallisuudessa on, että palveluiden riippuvaisuuden ja vaatimukset vaihtelevat paljon, joten kevyt rajapinta, joka mahdollistaa kehittäjän toteuttamaan vain palvelun tarvittavat ominaisuudet, eikä mitään ylimääräistä, on ideaalinen. Kirjastot Node.js-kehitysalustan ympärillä keskittyvät pieniin modulaarisiin ja itsenäisiin toiminnollisuuksiin. (Cantelon ym. 2013, 176.)

6.5 Python

Python on korkeantason tulkittu ja dynaaminen olio-ohjelmointikieli. Nämä ominaisuudet yhdessä sen sisään rakennettujen tietorakenteiden kanssa tekevät siitä hyvän ja nopean vaihtoehdon palveluiden prototypointiin sekä kehittämiseen. Tulkittu ohjelmointikieli tarkoittaa sitä, että koodia ei tarvitse erikseen kääntää konekielelle. Tämän seurauksena virheiden etsintä ja palveluiden testaus on yksinkertaisempaa. (Python Software Foundation.)

Ensimmäinen versio ohjelmointikielestä julkaistiin vuonna 1989, mutta vasta vuonna 2000 julkaistun Python 2.0-version avulla ohjelmointikieli alkoi saada huomiota ja yhteisöt sen ympärillä kasvoivat. Python 3.0 julkaistiin vuonna 2008 ja edustaa ohjelmointikielen tulevaisuutta, mutta 2.0-version kehitystä kuitenkin jatkettiin aina 2010 vuoteen asti, sillä versioiden välillä oli niin paljon eroavaisuuksia, että kaikki kirjastot ja ohjelmistot tulisi kirjoittaa uudestaan. Useat kirjastot ovat vieläkin kääntämättä python 3.0-versiolle, joten vanhan version käyttämiseen voi olla perusteltu syy. (The Python Wiki.)

7 TIETOKANNAT

Tietokanta on kokoelma reaalimaailmasta selkeästi rajattua tietoa, ja tietokantajärjestelmät ovat ne sovellukset, jotka mahdollistavat tämän tiedon käytön. Tietokantajärjestelmät ovat suunniteltu käsittelemään suuria määriä tärkeää tietoa, tavoitteena olla mahdollisimman luotettava, tehokas ja kätevä. (Silberschatz, Korth & Sudershan 2011, 28.)

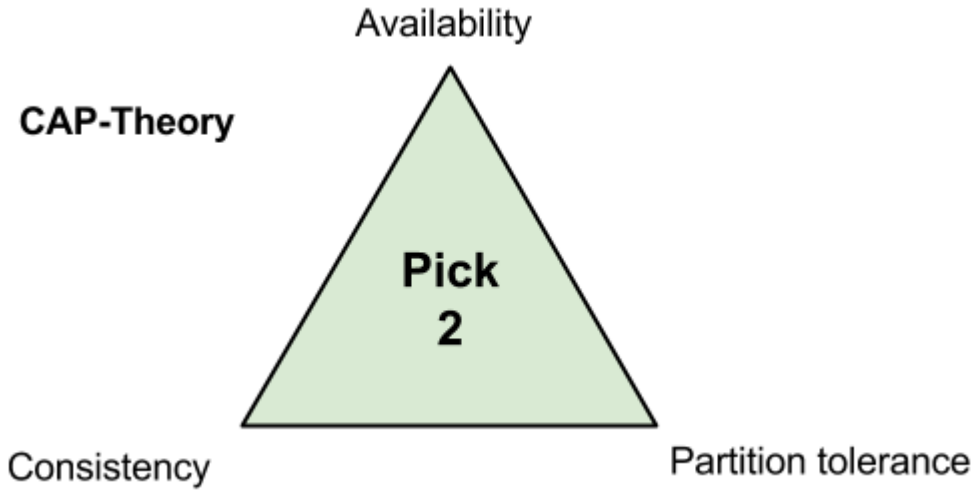
Informaatio on tärkeää organisaation palveluille ja toiminnalle. Tämän vuoksi tietokantajärjestelmiä on kehitetty jatkuvasti, ja ne ovat hyvin erikoistuneita erilaisiin tarkoituksiin. Tietokantojen perustana on jokin malli, jonka avulla järjestelmä kuvaa tiedon, sen sisäiset suhteet, semantiikan, konsistenssin sekä rajoitukset. (Silberschatz ym. 2011, 30.)

Tiedon skeema eli sen malli on omavarainen ja looginen määritelmä tiedon rakenteelle. Skeeman määrittelyllä mahdollistetaan tietokantajärjestelmän ja tiedon vuorovaikutus. Sen perusteella voidaan erotella ja kategorisoida tietokantajärjestelmiä. (Date 2015, 35.)

Tietokantajärjestelmät, sekä niiden käyttämät tiedon mallit, eroavat niiden käytännön ominaisuuksien priorisoinnissa. CAP-teoria väittää, että hajautetussa järjestelmässä, voidaan taata vain kaksi seuraavista luku- ja kirjoitusoperaatioiden ominaisuuksista:

- Johdonmukaisuus, niin että, lukuoperaatio palauttaa aina uusimman arvon sitä hakevalle.
- Tiedon saatavuus eli toimiva järjestelmä vastaa oletetulla tavalla kohtuullisessa ajassa.
- Osio toleranssi, eli järjestelmän toiminta ei katkea, vaikka tietoverkon kommunikaatio osittuisi. Osittautumisella viitataan kahden tietokantapalvelimen välisen verkkoliikenteen estymiseen. (Greiner 2014.)

Tämä tarkoittaa, että kun järjestelmä on hajautettu pilvessä, joudutaan tietokannan ja sitä käyttävien palveluiden valinnassa valitsemaan joko tiedon johdonmukaisuuden tai sen saatavuuden välillä (kuva 9).



Kuva 9. CAP-teoria.

Tämän erottelun perusteella pystytään myös kategorisoimaan tietokantoja ja tekemään valintoja perustuen tarpeiden mukaan. Loppuen lopuksi kuitenkin valinta näiden ominaisuuksien välillä on kehittäjän käsissä, sillä se voidaan tehdä tietokantajärjestelmistä irrallisena, sovelluspohjaisen ratkaisuna. (Greiner 2014.)

Mitä isompi tietokanta ja tallennetun tiedon määrä, sitä enemmän resursseja vaaditaan tietojärjestelmiltä. Suuret määrät yhtäaikaisia kutsuja voi tukkia tietokoneen prosessorin, ja isojen tietomäärien käsittely vaatii paljon keskusmuistilta, jonka ylikäyttäminen rasittaa ja hidastaa kovalevyjä. Seurauksena tietokantaa hyödyntävät palvelut hidastuvat merkittävästi. (MongoDB 2016 a.)

Palveluiden hidastumista voidaan ehkäistä kasvattamalla palvelinkapasiteettia horisontaalisesti tai vertikaalisesti. Vertikaalinen skaalautuminen tarkoittaa olemassa olevan laitteiston ja sovellusten parantamista, esimerkiksi palvelintietokoneen prosessorin vaihtaminen uudempaan ja tehokkaampaan. Horisontaalinen skaalautuminen tarkoittaa uusien laitteiden hankkimista ja käyttöönottoa, jonka seurauksena järjestelmän ei myöskään tarvitse olla yhden tietokoneen varassa, vaan voidaan toteuttaa korkean saatavuuden järjestelmiä. (MongoDB 2016 a.)

7.1 Replikointi

Replikoinnilla tarkoitetaan tietokantojen kontekstissa, että tiedot ovat kahdennetaan toiseen fyysiseen sijaintiin. Fyysisellä tasolla tämä usein tarkoittaa jonkinlaisen RAID-tekniikan toteuttamista, jolla tietokoneiden vikasietoisuutta parannetaan käyttämällä useita erillisiä kiintolevyjä, jotka yhdistetään yhdeksi loogiseksi levyksi. (Poulton 2013.) Sen lisäksi replikoinnissa usein käytetään erillisiä replikointipalvelimia, joiden toiminta rajoittuu vain varsinaisen tietokantapalvelimen kanssa kommunikoimiseen ja tiedon replikoitumiseen eri laitteistoon. (Poulton 2013.)

Riippuen replikoinnin aggressiivisuudesta, jatkuva tiedon siirron määrä kasvaa ja aiheuttaa suuria verkkoliikenteen kasvupiikkejä, etenkin kun tietoa siirretään fyysisesti pitkiä matkoja internetin yli. (Poulton 2013.)

7.2 Relaatiomalli

Tietokantajärjestelmien yleisin tietomalli on relaatiomalli. Se käyttää tiedon ja sen suhteiden esittämiseen toisiinsa linkitettyjä taulukkoja (Silberschatz ym. 2011, 64). Taulukot sisältävät eri kolumneja ja tieto asetetaan taulukon riveille, yksi tietue yhdellä rivillä. Taulukoilla on yksilöivä nimi, joka kuvaa siihen tallennettavaa samanlaista tietoa, kuten esimerkiksi käyttäjätiedot. Kolumneilla on myös uniikki nimi, ja niiden tarkoitus on kuvata tiedon tallennettavat ominaisuudet sekä mahdollisesti indeksoida tietueet. (Silberschatz ym. 2011, 66.)

Relaatiomalliset tietokannat tunnetaan ominaisuudestaan käsitellä tiedon eheyttä. Tiedon eheys viittaa tiedon huoltamiseen niin, että sen oikea muoto ja johdonmukaisuus varmistetaan. Käytännössä tämä toteutetaan pakottamalla skeemaa niin, että kirjoitettava tai luettava taulukko lukitaan muilta käyttäjiltä operaation ajaksi. Näin estetään yhtäaikaisista operaatioista syntyvät ongelmat eli tiedon päällekirjoittaminen tai vanhan tiedon lukeminen. Jokainen lukitus tarkoittaa asiakkaalle odottelua, sillä operaatiot tapahtuvat järjestyksessä eikä yhtäaikaisesti. (Halperin 2015.)

Kun tiedot ovat suhteutettu toisiinsa, on lukittava kerrallaan useita taulukkoja yhden tietokanta operaation vuoksi. Ongelma pahenee, kun käytössä on useampi, eri palvelimille hajautettu tietokanta. Lukitusaika usein kasvaa suoraan suhteessa käyttäjämäärään ja täten voi nopeasti hidastaa sovellusta. (Halperin 2015.)

7.3 NoSQL-malli

NoSQL-tietokannat eivät aseta tiedon tallennukseen mitään rajoituksia, vaan vastuu tiedon eheydestä ja sen rakenteesta tai skeemasta jää sitä käyttävälle sovellukselle. Rivien ja taulukkojen sijaan tallennetaan dokumentteja, jotka ovat pienempi osa kokoelmaa. Dokumentti voi sisältää BSON-muotoista tietoa, avain-arvo tietueita tai graafi tietoja. Kokoelmien ja dokumenttien skeema rakennetaan dynaamisesti sovelluksesta syötettyjen tietojen perusteella. (Halperin 2015.)

NoSQL-Tietokannoilla on muutama yleinen ominaisuus, jota ne kaikki tukevat enemmän tai vähemmän. Nämä ominaisuudet ovat:

- relaatiomallin puuttuminen,
- dynaaminen skeema,
- lopulta tapahtuva johdonmukaisuus,
- avoin lähdekoodi,
- helposti hajautettava. (Brust 2012.)

Pakotetun skeeman puuttuminen tarkoittaa, että järjestelmä on joustava. Minkään tietueen ei tarvitse olla rakenteeltaan samanlainen. Tietokantaa käyttävässä sovelluksessa on mahdollista lisätä avain-arvo pareja dynaamisesti dokumentin skeemaan sen luonnin jälkeen, ja tämän vuoksi myöskään suhteita tietojen välillä, ei tietokantaohjelmiston puolesta voida pakottaa. (Brust 2012.)

Kun tiedon eheyttä ei pakoteta, vaikuttaa se positiivisesti tietokannan nopeuteen. Kirjoitus operaation ajaksi dokumentti pitää lukita, mutta lukitus ei säteile muihin,

koska järjestelmässä ei ole pakotettua skeemaa tai suhteita, jota pitäisi ylläpitää. (Halperin 2015.)

MongoDB

MongoDB on NoSQL-tietokanta, joka on DB-engines sivuston mukaan markkinoiden viidenneksi suosituin tietokantajärjestelmä, ja täten myös suosituin omassa mallissaan. (DB-engines).

MongoDB on julkaistu vuonna 2009 ja on avoimen lähdekoodin tietokantajärjestelmä. Ohjelma on toteutettu C++-ohjelmointikielellä ja tukee Linux, OSX, Solaris ja Windows-käyttöjärjestelmiä. Tietokanta on skeema vapaa ja tukee useita ohjelmointikieliä, kuten C, JavaScript ja Python. (DB-engines).

Replikointia MongoDB toteuttaa replica set -nimisten prosessiryhmien avulla. Prosessiryhmät muodostetaan mongod-prosesseista, jotka ovat konfiguroitu replikointia varten. mongod-Prosessi on tietokantajärjestelmän pääkomponentti, ja on vastuussa tietokannan toiminnasta. (MongoDB b).

Replikoinnin konfiguroinnissa määritellään yksi päätietokanta, ja yksi tai useampi sekundaaritietokanta. Päätietokanta on ainoa, johon voidaan suorittaa sekä luku, että kirjoitusoperaatioita, ja sekundaaritietokannat tukevat vain lukuoperaatioita. Nämä tietokannat kommunikoivat päätietokannan kanssa ja ylläpitävät kopiota sen tiedoista (MongoDB b).

Jos päätietokannaksi säädetty prosessi tai palvelin vikaantuu, tai verkkoliikenne siihen estyy jostain syystä, niin sekundaaritietokannat järjestävät äänestyksen ja valitsevat joukostaan uuden päätietokannan. Näin toteutetaan tietokantapalvelun vikasietoisuus ja mahdollistetaan korkean saatavuuden palvelu. (MongoDB b).

Replikointiklusteriin voidaan lisätä yksi mongod-instanssi ja konfiguroida se arbiter-rooliin. Tällaiseen rooliin asetetut prosessit eivät ylläpidä mitään tietoa, vaan niiden tarkoitus on ylläpitää päätöskykyä vastaamalla ajoittaisiin yhteyden tarkistuksiin sekä äänestyspyyntöihin. Sen päätehtävä on siis ratkaista uuden päätietokannan valinta, jos äänestys menee muuten tasan. arbiter-Palvelu ei ylläpidä tietoa, joten se ei vaadi ympäristöltään paljoa resursseja. (MongoDB b).

8 PROTOTYYPPI: PILVIPALVELU ESINEIDEN INTERNETILLE

Opinnäytetyöni käytännön osuus suoritettiin Kajaanin ammattikorkeakoululle esisijaisesti Älykkäät järjestelmät -koulutuksen tarpeisiin. Tarkoituksena työssä oli toteuttaa erilaisia kommunikaatiopalveluita, joita voidaan hyödyntämään projektien sekä opetuksen kanssa, liittyen esineiden internet ilmiöön. Tuotettuja pilvipalveluiden prototyyppejä hyödynnetään koulun eri projekteissa ja koulutuksissa. Aikaisemmin koululla on käytetty vain vähän hyödyksi erilaisia palvelin puolen ratkaisuja älykkäissä järjestelmissä. Nämä ratkaisut ovat olleet valmiita kolmannen osapuolen tuotoksia, joten tämän opinnäytetyön käytännön tuotokset toimivat myös esimerkkinä palveluiden tuottamiseen itse.

Työssä toteutettiin useammalla ohjelmointikielillä erilaisia asiakas- ja palvelinsovelluksia, tarkoituksena prototypoida koko kommunikaatio ketju pilvipalveluista älykkäisiin esineisiin. Toinen tarkoitus oli tuottaa esimerkkejä, joiden pohjalta muut opiskelijat ja opettajat voivat helposti soveltaa omia vastaavia ratkaisujaan pilvipalveluiden tuottamiseen. Näiden perusteella opinnäytetyön käytännön toteutus rajattiin kolmeen kokonaisuuteen:

1. Hajautettu NoSQL-mallin tietokanta,
2. REST-arkkitehtuurin mukainen nettipalvelu, joka mahdollistaa helpon tiedon tallentamisen sekä hakemisen tietokannasta,
3. MQTT-viestintäpalvelin, ja sitä hyödyntäviä asiakas-sovelluksia.

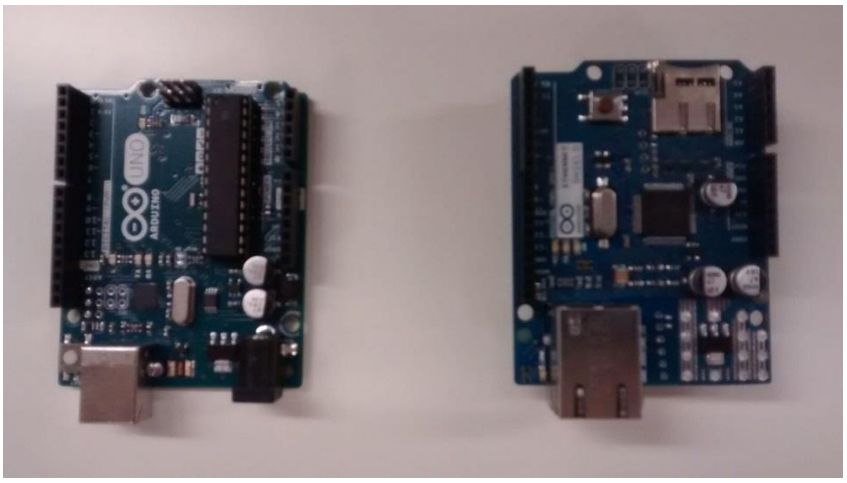
Toteutuksen tavoitteista karsittiin COAP-protokollaa hyödyntävä nettipalvelu, nettipalvelua käyttävät asiakas-sovellukset sekä palveluiden automaatio Ansible ohjelmistolla. Lisäksi palveluiden tietoturvan suunnittelu ja toteutus jätettiin tarkoituksella työn ulkopuolelle, ja prioriteetti asetettiin sen sijaan toiminnollisuuden esittämiseen ja todistamiseen.

Käytännön osuuden toteuttamiseen käytettiin Debian Jessie -palvelinkäyttöjärjestelmää, joka pohjautuu Linux-ytimeen. Sovelluksien ohjelmoinnissa käytettiin git-

versionhallintaohjelmaa, sekä Sublime Text 3 ja Arduino IDE-tekstinkäsittely ohjelmia. Palveluiden ohjelmointikielenä käytettiin JavaScript, Python ja Processing-kieliä.

8.1 Testiympäristö ja laitteet

Testiympäristö muodostui Kajaanin ammattikorkeakoulun, Data center -laboratorion virtualisointi- ja verkkoympäristöstä, sekä Arduino Uno -mikrokontrollereista, jotka toimivat prototyypeissä älykkäinä esineinä, keräämässä sensorikomponenttien avulla tietoa fyysisestä maailmasta. Arduino Uno -mikrokontrollereissa käytettiin erillistä Ethernet-kilpeä, joka mahdollistaa laitteen kommunikoinnin internetiin (kuva 10). Virtualisointiympäristöstä käytössä oli useampi palvelinkone.



Kuva 10. Kuvassa vasemmalla Arduino Uno -mikrokontrolleri ja oikealla yhteensopiva Ethernet-kilpi.

8.2 Tietokantajärjestelmän suunnittelu

Tietokantajärjestelmäksi valittiin MongoDB, sillä sen ominaisuudet täsmäävät palveluiden tarpeisiin. Esineiden internet tarvitsee tietokannan, joka kykenee nopeisiin ja useisiin yhtäaikaisiin operaatioihin, sekä kykenee hallitsemaan suuria määriä tietoja. Tämän lisäksi resurssien esitysmuodoksi valittiin prototyypeissä JSON-tiedostomuoto, jolloin ne voidaan sellaisenaan tallentaa MongoDB tietokantaan,

helpottaen ja nopeuttaen kehitystyötä. Tietokanta halutaan hajauttaa, koska se tuo palveluiden toiminnalle varmuutta sekä vikasietoisuutta. Tätä tietokantaa käytettiin hyödyksi kaikissa toteutetuissa prototyypeissä.

Suunnitelma oli hajauttaa tietokanta viiteen erilliseen virtualipalvelimeen. Nämä palvelimet tulevat olemaan verkotettuna toisiinsa, sekä varsinaista palvelua ylläpitävään virtualipalvelimeen, yksityisessä verkossa.

8.3 Tietokantajärjestelmän toteutus ja testaus

MongoDB-Tietokantajärjestelmän asentaminen ei eroa normaalista ohjelmisto-asennuksesta Linux-pohjaisessa käyttöjärjestelmässä. Järjestelmän kotisivuilla on yksityiskohtaiset ohjeet eri käyttöjärjestelmille, ja asennuspakettia tarjotaan julkisesti näkyvissä olevasta repositoriosta, jota ylläpitää järjestelmän kehitysorganisaatio. Asennuksen mahdollistamiseksi lisättiin viite tästä repositoriosta käytössä olevaan testiympäristön käyttöjärjestelmän pakettihallintaohjelmistoon, täten mahdollistaen asennuksen oletusohjelmistolla. Asennettu versio tietokannasta on 3.2.

Tietokantajärjestelmä voidaan säätää erillisen konfigurointitiedoston avulla, tai ottamalla yhteyden käynnissä olevaan järjestelmään ja syöttämällä asetukset prosessille konsolin kautta. Poikkeuksellisesti järjestelmän verkko ja replikointi asetukset täytyy syöttää jo käynnistyksen yhteydessä.

Kaikille tietokannoille asetettiin kuvan 11 esittämät asetukset, jotka tallennettiin erilliseen tekstitiedostoon, ohjelman oletusarvoisella nimellä ja sijainnilla levyjärjestelmässä. Tämä tarkoittaa, että käynnistyessään tietokanta kykenee löytämään asetukset automaattisesti ja ottamaan ne käyttöön. Tietokantaprosessit käynnistettiin Debian-käyttöjärjestelmän mukana tulevalla service-ohjelmistolla.

```
storage:
  dbPath: /data/db/
  journal:
    enabled: true

systemLog:
  destination: file
  logAppend: true
  path: /var/log/mongodb/mongod.log

net:
  port: 27017
  bindIp: 0.0.0.0

replication:
  replSetName: "r1"
```

Kuva 11. ”Mongod.conf”-tiedoston asetukset.

Tämän jälkeen valittiin pääpalvelin ja alustettiin replikointiklusteri sen kautta. Alustaminen tehdään vain yhdelle tietokannalle, ja siitä muodostuu päätietokanta. Tämän jälkeen muut palvelimet kuvattiin JSON-tiedostomuodossa ja annettiin parametrina tietokannan komentorivin ”rs.add” -funktiolle, joka yhdistää palvelimet ja aloittaa replikoinnin.

Tietokantaklusterin toiminnallisuus pystytään helposti todistamaan kirjoittamalla tietoa päätietokantapalvelimelle ja lukemalla sama tieto replikoivalta tietokantapalvelimelta. Klusterin vikasietoisuus pystyttiin todentamaan sammuttamalla päätietokantapalvelin, ja varmistaa, että uusi päätietokantapalvelin valitaan.

8.4 Esineiden rajapintapalvelun suunnittelu

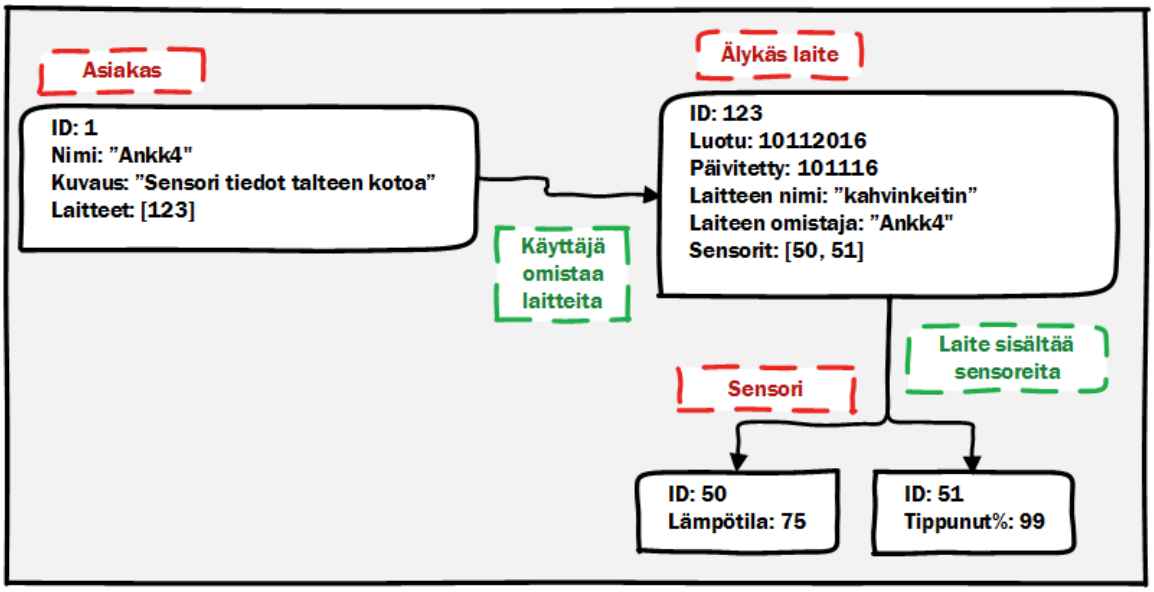
Palvelussa on tarkoituksena tuoda älykkäitä esineitä näkyville internettiin, sekä mahdollistaa niiden eri tilojen ja tietojen helppo ohjelmallinen lukeminen ja tallentaminen. Palvelun on tarkoitus noudattaa REST-arkkitehtuuria sekä kolmentason

asiakas- ja palvelin arkkitehtuuria. Resurssien tilat esitetään JSON-tiedostomuodossa ja kommunikointi hoidetaan tekstipohjaisena, HTTP-protokollaa hyödyntäen.

Tällaista REST-arkkitehtuuria noudattavan rajapinnan suunnittelu aloitetaan resurssien määrittelystä. Jokaiseen resurssiin liitetään palvelun toimesta yksilöivä ID-tunnus sekä luonti- ja muokkausajankohdat. Resurssit muodostuvat älykkäästä laitteesta, sen sensorikomponenteista ja palvelua käyttävästä asiakkaasta.

Laitteiden ominaisuuksista tallennetaan tarkka tuotenimi, laitteen omistaja sekä viittaukset sen sensorikomponenttien resursseihin. Laitteen omistaja-arvo on viittaus käyttäjästä muodostuvaan resurssiin. Sensorikomponentti resurssissa ominaisuuksiksi tallennetaan mitattavan suureen nimi, mittauksesta saatu arvo ja sen omistava laite.

Käyttäjä-resurssin ominaisuuksista tallennetaan käyttäjän nimi, kuvaus palvelun käyttötarpeesta sekä viittaukset sille kuuluviin laitteisiin. Asiakkaalla voi olla useampi laite, ja laitteella voi olla useampi sensorikomponentti. Resurssien tunnistukset tehdään aina ID-arvon perusteella. Suunniteltu resurssien rakenne sekä suhteet esitetään kuvassa 12.



Kuva 12. Asiakas, älykäs laite ja sensori-resurssit, sekä niiden väliset suhteet.

Alun perin suunnitelmani oli toteuttaa sensorikomponentit laitteiden aliresurssina, mutta prosessin aikana huomasin, että varsinainen mittaustieto tulee olemaan määrällisesti suuri, ja täten sitä ei ole hyvä sitoa suoraan toiseen resurssiin. Resurssia käsitellessä jouduttaisiin myös aina käymään läpi kaikki siihen liitetyt mittaustiedot, ja tämä hidastuttaisi palvelun käyttöä suorassa suhteessa mittaustietojen määrään. Täten on parempi erottaa jokainen sensorikomponentti omaan resurssiinsa, ja lisätä näihin viittaava viite jokaiseen laiteresurssiin, vaikkakin sen lisää hieman ohjelmointityötä.

8.5 URL-Polkujen suunnittelu

URL-Polkujen suunnittelu on helppoa, kun resurssit ovat määritelty hyvin. Polkujen perusteella päätetään myös palvelun tuetuista HTTP-ominaisuuksista. Peruspolku muodostuu aina sovelluksen versionumerosta, jota välittömästi seuraa resurssin nimi monikossa. Jos resurssin jälkeen huomataan URL-tunnuksessa ID-muuttuja, palvelu sen sijaan suorittaa operaation muuttujan määrittelemään yksittäiseen resurssiin, tunnistaen halutun resurssin vertaamalla tätä muuttujaa ID-tunnuksiin. Taulukko 2 on esimerkki käyttäjä resurssin muodostamista poluista, HTTP-verbeistä ja niiden toiminnallisuudesta.

Taulukko 2. Käyttäjä resurssin toteuttamat polut.

HTTP VERBI	**POLKU**	**TOIMINTO**
GET	/v1/users	Hae kaikki käyttäjät.
GET	/v1/users?id	Hae yksi käyttäjä ID-arvon perusteella.
POST	/v1/users	Luo uusi käyttäjä.
PATCH	/v1/users?id	Päivitä käyttäjä resurssi.
DELETE	/v1/users?id	Poista käyttäjä ID-arvon perusteella.

Jokaiselle resurssille toteutetaan taulukossa 2 esitetyt polut. ”Älykäs esine” -resurssi tunnistetaan englanninkielen device-nimellä, ja sensori resurssi taas sensor-nimellä. HTTP-verbien toiminnot tullaan toteuttamaan standardin mukaisesti. Poikkeuksena DELETE-verbin toiminnollisuus ei tule poistamaan mitään tietoa resurssien kokoelmista pysyvästi, vaan sen sijaan poistettavat tiedot siirretään toiseen tietokantaan talteen. Tämä tehdään vahinkojen välttämiseksi, ja poistettu tieto voi myöhemmin saada arvonsa takaisin, joten sitä ei ole järkeä poistaa välittömästi.

Jos pyyntöjä tehdään väärään polkuun, antaa palvelin HTTP-protokollan mukaisen, oikeaoppisen virheilmoituksen vastauksena. Onnistuneisiin pyytöihin vastataan joko pyydetyllä resurssilla, tai jälleen toiminnan tilaa kuvaavalla palvelimen tilakoodilla.

8.6 Rajapintapalvelun toteutus

Rajapintapalvelun prototyyppi ohjelmointiin Node.js -kirjastolla ja sen toteutuksessa hyödynnettiin suunniteltuja resursseja sekä tietokantapalvelua. Ohjelmoinnissa käytettiin seuraavia kolmannen osapuolen kirjastoja: express, path, body-parser ja mongoose. Lisäksi sovellusta varten ohjelmointiin seuraavat moduulit: device, user, sensor, deviceRouter, userRouter, sensorRouter, deviceController, userController ja sensorController.

Mongoose-kirjasto mahdollistaa yhteyden MongoDB-tietokantaan ja helpottaa dokumentti skeemojen luomista. Muut listatut kolmannen osapuolen kirjastot ovat pieniä apuohjelmia JSON-tiedostomuodon ja URI-tunnuksien hallitsemiseen.

Sovellusta varten ohjelmoidut moduulit jakaantuvat kolmeen osaan: tiedon skeema, URL-reititin ja reitittimen hallinta. Palvelun silmukka aloitetaan server.js -nimisestä tiedosta, josta express-kirjasto alustaa kaikille resursseille reititin moduulit. Reititin moduuli vastaanottaa pyynnöt käsittelyä varten ja kutsuu jotain funktiota reitittimen kontrollerista. Kontrolleri on vastuussa varsinaisista HTTP-protokollan verbeistä ja niiden ominaisuuksien toteuttamisesta. Koska rajapintapalvelun

koodirivien määrä on todella suuri, ei sitä nähty järkeväksi lisätä liitteeseen, vaan sovellus jää koulun sisäiseen versionhallintaan. Liitteessä 1 on luokkakaavio rajapinnan toteutuksesta.

8.7 MQTT-viestintäpalvelin ja asiakassovellukset

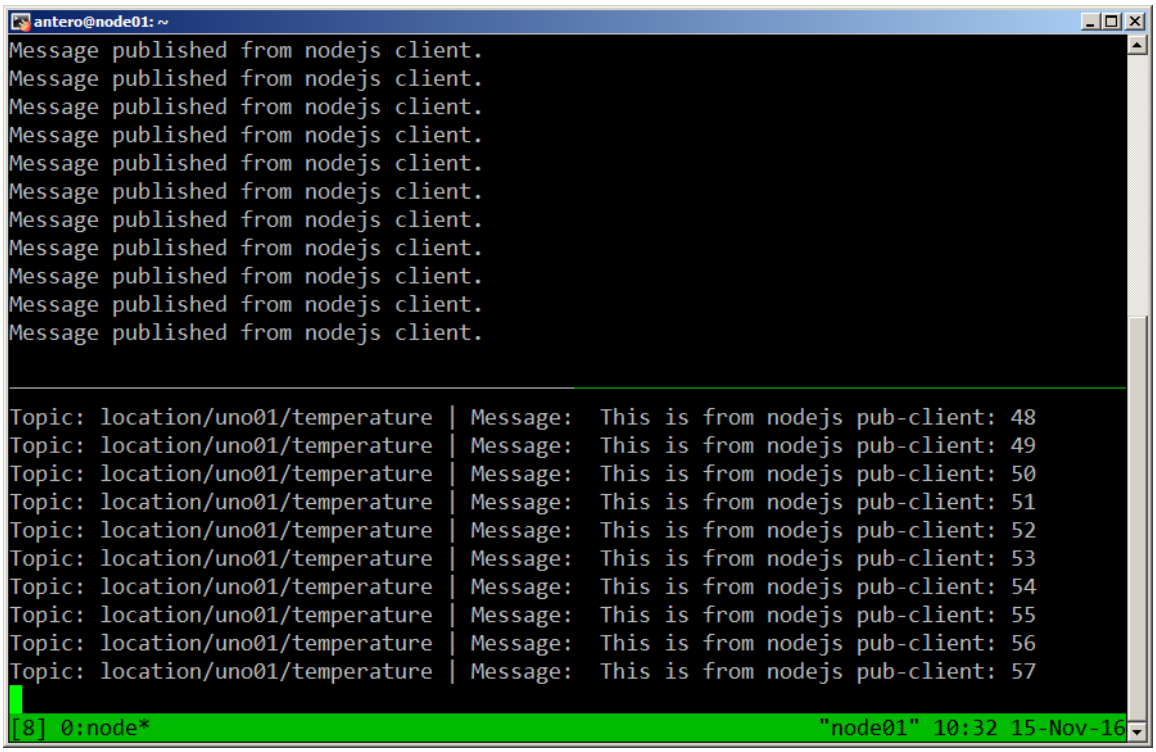
MQTT-protokollan viestintäpalvelimeksi valitsin RabbitMQ-sovelluksen. Se tukee virallisen lisäosan kautta MQTT-protokollan uusinta standardia. RabbitMQ-sovellus sisältää nettisivun päälle rakennetun hallintatoiminnollisuuden, joten se on hyvä visualisoimaan prototyypissä viestien kulkua.

Asiakassovellukset toteutettiin Node.js ja Python ohjelmilla palvelin puolella, sekä Processing-ohjelmointikielellä Arduino Uno -mikrokontrollerissa. Tarkoituksena on toteuttaa järjestelmä, jossa Arduino Uno julkaisee sensoridataa MQTT-protokollaa käyttäen ja erillinen palvelin sovellus tilaa datan, sekä tallentaa sen tietokantaan. Tietokantana hyödynnetään hajautettua MongoDB-tietokantaklusteria.

Toteutus ja testaus

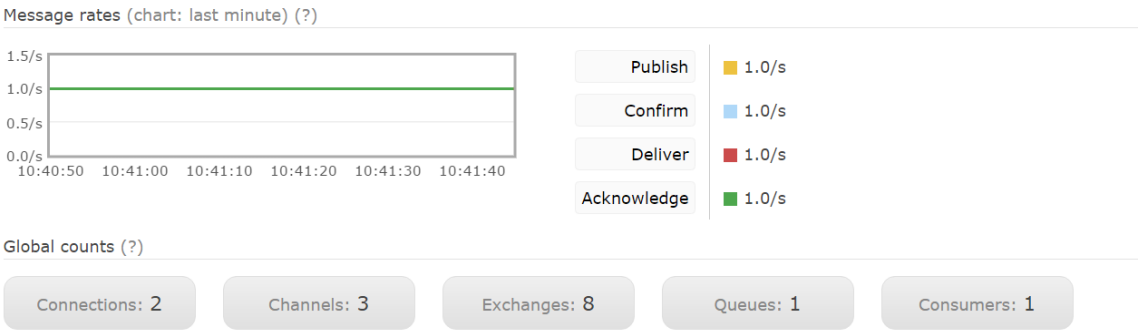
RabbitMQ-viestintäpalvelimen asennus testiympäristöön tarvitsi erillisen viittauksen repositorioon, että asennuspaketit pystyttiin asentamaan käyttöjärjestelmän asennustyökaluilla. Tämän jälkeen MQTT- ja hallintalisäosat täytyi aktivoida erikseen ennen kuin niitä pystyi käyttämään.

Protokollan ohjelmointiin löytyy avoimen lähdekoodin moduuleja usealle kielelle. Ensimmäisenä testattiin viestintäpalvelimen toimintaa yksinkertaisten Node.js -ohjelmien avulla. Kuvassa 13 yläpuolella ajettava Node.js -ohjelma julkaisee viestin ja alapuolella ajettavassa ohjelmassa se viesti tilataan. Julkaisija lähettää yhden viestin sekunnin välein, ja jatkaa lähettämistä ikuisesti. Molemmat ohjelmat käyttävät samaa aiheotsikkoa, että viesti voidaan välittää perille. MQTT-protokollan ohjelmoimiseen Node.js -sovelluksissa käytettiin ”node-mqtt”-kirjastoa.



Kuva 13. Node.js ohjelmistot kommunikoivat viestintäpalvelimen avulla.

RabbitMQ-hallintapaneelista voidaan todentaa yhteyksien muodostuminen. Kuvassa 14 on hallintapaneelin yhteenveto välitetyistä viesteistä, ja ne täsmäävät testin parametrien mukaisesti.



Kuva 14. RabbitMQ-hallintapaneelin yhteenveto yhteyksistä ja viesteistä.

Arduino Uno -kehitysalustaan on useita MQTT-kirjastoja, mutta vaihtoehtoja tutkiessa huomattiin, että yksi niistä oli paremmin dokumentoitu ja muutenkin sopiva tarkoitukseeni. Kirjasto on nimeltään ”pubsubclient”, ja on saatavilla Arduino IDE -tekstinkäsittelyohjelmiston kautta. Liitteessä 2. on tätä hyödyntävä Arduino Uno -

sovellus. Sovellus hyödyntää MQTT-kirjaston ominaisuuksia niin, että vaikka yhteys menetetäänkin väliaikaisesti, pystyy se yhteyden palautuessa jatkamaan lähetystä. Lisäksi koodissa tiedon muotoillaan JSON-tiedostomuotoon ennen niiden lähetystä.

Liittessä 3. on Python ohjelmakoodi, joka tilaa tietoa MQTT-palvelimelta, ja kirjoittaa sen MongoDB-tietokantaan. Ohjelman tarkoitus on mahdollistaa kaikkien viestintäpalvelimien läpi kulkevien viestien tallentaminen. Ohjelma tilaa kaikki viestit tietystä, päätetystä aiheotsikosta. Esimerkiksi jos sensorilaiteet julkaisevat tietoa ”paikka/laite/sensori”-skeemalla, voidaan ohjelmisto laittaa tilaamaan ja tallentamaan kaikki tietyn paikan sensorit yhteen tietokantaan hyödyntämällä jokerimerkkejä. Ohjelman toimivuus voidaan todeta asettamalla toinen sovellus julkaisemaan jotain dataa, ja kuvassa 15 voidaan nähdä, että tiedot ovat tallentuneet oikeassa muodossa, aiheotsikon mukaiseen kokoelmaan, ja sovellus toimii suunnitellulla tavalla.

```
r1:PRIMARY> show dbs
local  0.012GB
mqtt   0.000GB
test   0.000GB
r1:PRIMARY> show collections
location/arduino-uno-01/outTopic/temperature
r1:PRIMARY> db['location/arduino-uno-01/outTopic/temperature'].find()
{ "_id" : ObjectId("582b4685705cc82cf6e119d9"), "temperature" : 78.91 }
{ "_id" : ObjectId("582b4685705cc82cf6e119da"), "temperature" : 79.39 }
{ "_id" : ObjectId("582b4685705cc82cf6e119db"), "temperature" : 79.88 }
{ "_id" : ObjectId("582b4685705cc82cf6e119dc"), "temperature" : 80.37 }
{ "_id" : ObjectId("582b4685705cc82cf6e119dd"), "temperature" : 79.88 }
{ "_id" : ObjectId("582b4685705cc82cf6e119de"), "temperature" : 79.39 }
{ "_id" : ObjectId("582b4685705cc82cf6e119df"), "temperature" : 78.91 }
{ "_id" : ObjectId("582b4685705cc82cf6e119e0"), "temperature" : 78.42 }
{ "_id" : ObjectId("582b4685705cc82cf6e119e1"), "temperature" : 77.44 }
{ "_id" : ObjectId("582b4685705cc82cf6e119e2"), "temperature" : 76.95 }
{ "_id" : ObjectId("582b4685705cc82cf6e119e3"), "temperature" : 76.95 }
{ "_id" : ObjectId("582b4685705cc82cf6e119e4"), "temperature" : 77.44 }
{ "_id" : ObjectId("582b4685705cc82cf6e119e5"), "temperature" : 78.42 }
{ "_id" : ObjectId("582b4685705cc82cf6e119e6"), "temperature" : 79.39 }
{ "_id" : ObjectId("582b4685705cc82cf6e119e7"), "temperature" : 79.88 }
{ "_id" : ObjectId("582b4685705cc82cf6e119e8"), "temperature" : 79.88 }
{ "_id" : ObjectId("582b4685705cc82cf6e119e9"), "temperature" : 80.37 }
{ "_id" : ObjectId("582b4685705cc82cf6e119ea"), "temperature" : 80.37 }
{ "_id" : ObjectId("582b4685705cc82cf6e119eb"), "temperature" : 79.88 }
{ "_id" : ObjectId("582b4685705cc82cf6e119ec"), "temperature" : 78.91 }
Type "it" for more
r1:PRIMARY>
```

Kuva 15. MongoDB Tietokannan tarkistus testin jälkeen.

Kuvassa ensin tarkistetaan olemassa olevat tietokannat ”show dbs”-komennolla, jonka jälkeen todetaan, että sovellus on luonut mqtt-nimisen tietokannan. Tämän jälkeen tietokanta otetaan käyttöön ja sinne tallennetut kokoelmat tulostetaan konsoliin komennolla ”show collections”. Lopuksi voimme hakea kokoelman sisältämät tiedot ja todentaa sen sisältävän lämpötilamittauksia.

9 YHTEENVETO

Opinnäytetyössä tavoitteeksi asetettiin esineiden internet -ilmiötä tukevien palveluiden selvittäminen mahdollisimman monipuolisesti, sekä näiden eri palveluiden prototypointi ja toiminnan todentaminen. Pilvipalveluiden toimintaa ja rakennetta avattiin niiden syntyperän avulla, sekä purkamalla WWW-palvelualusta sen perusteknologioihin. Näitä teknologioita ja niiden toteuttamista tutkittiin lähempää, tarkoituksena mahdollistaa oman rajapintapalvelun tuottaminen.

Rajapintapalvelun lisäksi tutkittiin sulautettujen laitteiden kommunikointia internetin ylitse. Näiden laitteiden käytettävissä olevat resurssit ovat usein hyvin rajoitettuja, joten tarvitaan erikoistuneita ja tehokkaita työkaluja kommunikoinnin mahdollistamiseen. MQTT-sovelluskerroksenprotokolla on juuri tällainen työkalu. Sen toimintaperiaate on yksinkertainen ja ohjelmistokirjastoja on saatavilla laajasti.

Työssä toteutettiin kaksi erillistä prototyyppiä, joiden toiminnan tueksi rakennettiin hajautettu ja vikasietoinen NoSQL tietokanta. Älykkäiden esineiden rajapintasovellus toteutettiin Node.js -kehitysalustalla. Rajapinta abstrahoi älykkään esineen, sen omistajan eli palvelun käyttäjän ja varsinaisen sensoritiedon resursseiksi, jota pystytään tunnistamaan URL-tunnuksen avulla. Sovellus mahdollistaa M2M-kommunikoinnin ja toimii rajapintapalveluna muille sovelluksille, jotka haluavat hyödyntää sulautettujen esineiden informaatiota.

Toinen prototyyppi pilvipalvelu oli hub and spoke -verkotusmallin inspiroima MQTT-protokollan toteutus ja testaus. MQTT-protokollan topologia muodostuu kolmesta osapuolesta: julkaisia, tilaaja ja viestienvälityspalvelin. Työssä todettiin, että viestintäpalvelin mahdollistaa ja massiivisen tiedon tallentamisen kaikista protokollaa toteuttavista sulautetuista laitteista. Kaikki viestintäpalvelimen kautta kulkevat viestit voidaan tallentaa työssä luodulla sovelluksella tietokantaan, täten helpottaen viestejä tilaavien sovelluksien taakkaa.

Prototyyppien toteutuksessa ei huomioitu tietoturvaa tai viestinnän salausta millään tavalla, joten herkän tietojen tai laitteistojen käyttäminen niiden kanssa ei ole suositeltavaa.

Rajapintapalvelun käyttöönotto ei tule tapahtumaan välittömästi ja palvelua täytyy jatko kehittää, että siitä saataisiin Kajaanin ammattikorkeakoululle hyödyllinen ja toimiva rajapinta. Tavoitteet laboratorioympäristöjen kehittämisestä saavutettiin, sillä prototyypit tullaan ottamaan tulevaisuudessa laajemmin käyttöön erilaisissa projekteissa. Tämä ei tapahdu välittömästi, sillä prototyypit tarvitsevat jatkokehitystä ennen kunnollista käyttöönottoa. Kolmannen osapuolen pilvipalveluiden käyttöä ei siis korvata tuotettujen prototyyppien avulla vielä, mutta mahdollisuus oman pilvipalvelun muodostamiseen on lähempänä todellisuutta.

LÄHTEET

Amundsen, M. & Richardson, L. (2013). RESTful Web APIs. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O´Reilly Media, Inc.

DB-engines. Solid IT. Viitattu 9.11.2016. http://db-engines.com/en/ranking

Brust, A. (2012). Understanding NoSQL. [Videotiedosto]. Viitattu 15.11.2016. https://app.pluralsight.com/library/courses/understanding-nosql/table-of-contents

Chandrasekaran, K. (2014), Essentials of cloud computing (1st ed.). 6000 Broken Sound Parkway NW, Suite 300: Chapman and Hall/CRC.

Date, C. (2015). SQL and Relational Theory: How to Write Accurate SQL Code (3rd edition). 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc.

Davis, K., Turner, J., Yocom, N. (2004). The Definitive Guide to Linux Network Programming. 2560 Ninth Street, Suite 219, Berkeley, CA 94710: Apress, inc.

Doglio, F. (2015). Pro REST API Development with Node.js. 233 Spring Street, 6th Floor, New York, NY 10013: Apress, inc.

DuVander, A. (2013, 3. kesäkuuta). What Programming Language is Most Popular with APIs? Viitattu 7.11.2016. http://www.programmableweb.com/news/what-programming-language-most-popular-apis/2013/06/03

Foran, P. (2014). Your API is Bad. Viitattu 14.11.2015. https://lean-pub.com/yourapiisbad, Leanpub.

Glister, R. (2014). CompTIA cloud+ CV0-001 in depth. Boston, US: Cengage Learning PTR.

Greiner, R. (2014). CAP Theorem: Revisited. Viitattu 14.11.2016. http://robertgrei-ner.com/2014/08/cap-theorem-revisited/

Halperin, N. (2015). MongoDB Administration. [Videotiedosto]. Viitattu 15.11.2016. https://app.pluralsight.com/library/courses/mongodb-administration/table-of-contents

Hochstein, L. (2015). Ansible: Up and Running. 1005 Gravenstein Highway North, Sebastopol, CA 95472, USA: O'Reilly Media, Inc.

Huston, T. (2016). What is hypermedia? Viitattu 13.11.2016. https://smartbear.com/learn/api-design/what-is-hypermedia/

Jacobs, I. (2004). Architecture of the World Wide Web, Volume One. Viitattu 28.10.2016. https://www.w3.org/TR/webarch/#intro

Kellmereit, D., Obodovski, D. (2013). The Silent Intelligence: The Internet of Things. DND Ventures LLC.

Levent-Levi, T. (2015). IOT messaging - should we head for the cloud or P2P? Viitattu 15.10.2016. https://bloggeek.me/messaging-iot-p2p/

Long, J. (2012, 25. syyskuuta). I Don't Speak Your Language: Frontend vs. Backend. Treehouse. Viitattu 7.11.2016. http://blog.teamtreehouse.com/i-dont-speak-your-language-frontend-vs-backend

MongoDB Documentation. a. Sharding. Viitattu 9.11.2016. https://docs.mongodb.com/manual/sharding/

MongoDB Documentation. b. Replication. Viitattu 9.11.2016. https://docs.mongodb.com/manual/replication/

MQTT Essentials: Part 1 - Introducing MQTT. (2015). HiveMQ. Viitattu 7.11.2016. http://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt

MQTT Essentials Part 2: Publish & Subscribe. (2015). HiveMQ. Viitattu 7.11.2016. http://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe

MQTT Essentials Part 3: Client, Broker and Connection Establishment. (2015). HiveMQ. Viitattu 7.11.2016. http://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment

MQTT Essentials Part 5: MQTT Topic & Best Practices. (2015). HiveMQ. Viitattu 7.11.2016. http://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment

Nommensen, P. (2015, 6. helmikuuta). It's Time to Move to a Four-Tier Application Architecture. Viitattu 7.11.2016. https://www.nginx.com/blog/time-to-move-to-a-four-tier-application-architecture/

Pinna, F. & Pintus, A. The Web API Design guidelines for happy developers. Viitattu 14.11.2015. https://leanpub.com/thewebapinntux, Leanpub.

Poulton, N. (2013). CompTIA Storage+ Part 2: Network Storage & Data Replication. [Videotiedosto]. Viitattu 4.10.2016. https://app.pluralsight.com/library/courses/comptia-storage-plus-pt2-network-storage-data-replication/table-of-contents

Python Software Foundation. What is Python? Executive Summary. Viitattu 14.11.2016. https://www.python.org/doc/essays/blurb/

Raghavenderrao, R. (2015, 15. syyskuuta). IT Automation - What it means. Viitattu 14.11.2016. http://www.netenrich.com/it-automation-what-it-means/

Silberschatz, A., Korth, H., Sudarshan S. (2011). Database System Concepts, Sixth Edition. 1221 Avenue of the Americas, New York, NY 10020: The McGraw-Hill Companies, Inc.

Tietotekniikan termitalkoot a. (2011). Vertaisverkko. Viitattu 15.10.2016. http://www.tsk.fi/tsk/termitalkoot/node/267?page=get_id&id=ID0286&vocabulary_code=TSKTT

Tietotekniikan termitalkoot b. (2012). World Wide Web. Viitattu 13.11.2016. http://www.tsk.fi/tsk/termitalkoot/fi/node/266

The Python Wiki. Should I use Python 2 or Python 3 for my development activity? Viitattu 14.11.2016. https://wiki.python.org/moin/Python2orPython3

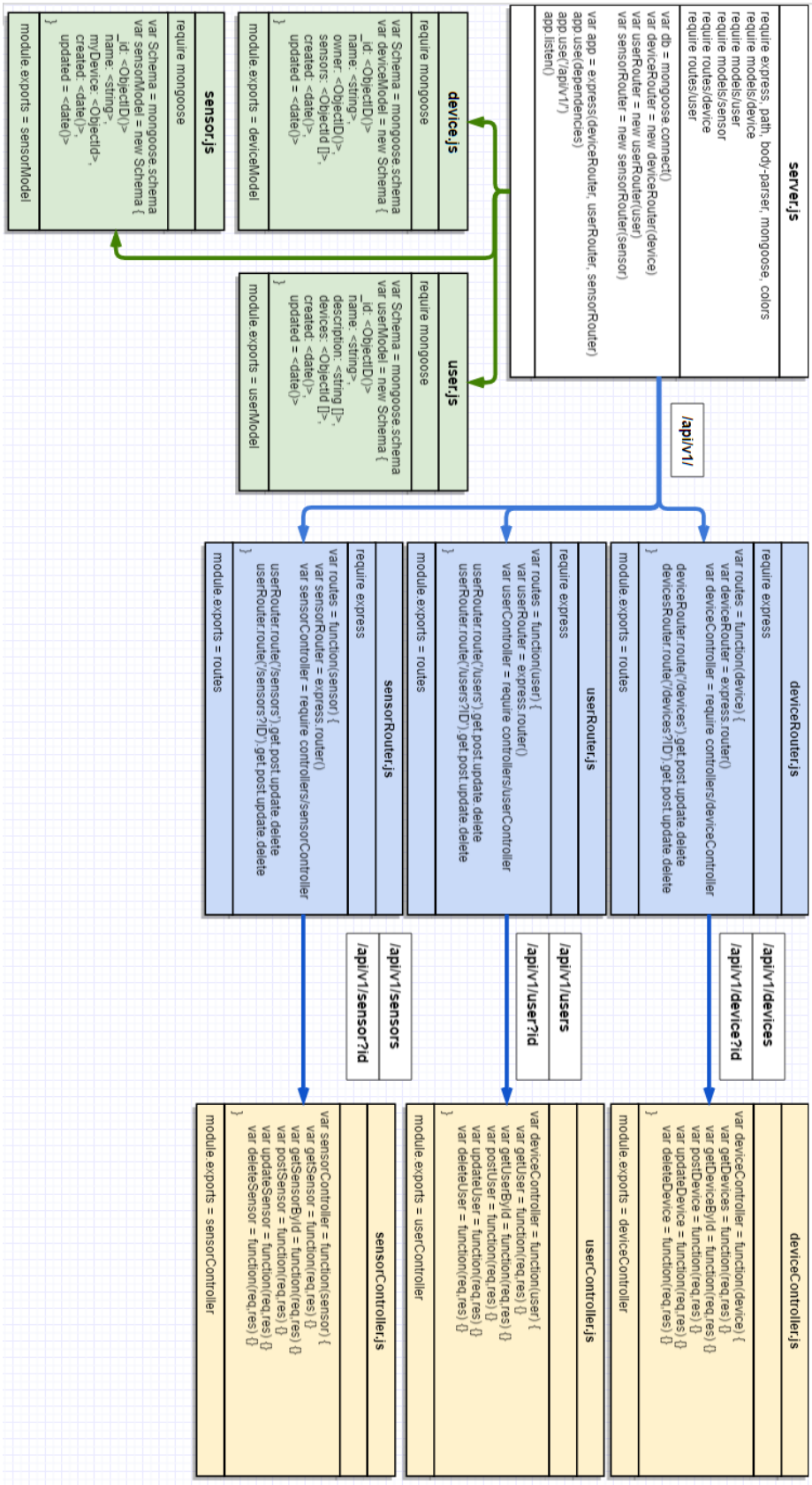
Tsonev, K. (2015). Node.js By Example. Livery Place, 35 Livery Street, Birmingham B3 2PB, UK: Packt Publishing.

Waher, P. (2015). Learning Internet of Things. Livery Place 35, Livery Street, Birmingham, B3 2PB, UK: Packt Publishing Ltd.

Webber, J., Paratatidis, S., & Robinson, I. (2010). REST in Practice. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O´Reilly Media, Inc.

LIITTEET

Liite 1. Rajapintapalvelun luokkakaavio.



Liite 2. MQTT –protokollaa hyödyntävä Arduino -sovellus.

```
#include <SPI.h>
#include <PubSubClient.h>
#include <Ethernet.h>

// MQTT library Initialization
PubSubClient mqttClient;

// MQTT Protocol options
const char* broker      = "192.168.202.83"; // test.mosquitto.org
const char* clientID    = "arduino-uno-01"; // Broker device ID
char* clientPath        = "location/arduino-uno-01"; // variable for topic
generation
byte qos                = 1;

const char* username    = "admin";
const char* password    = "Passw0rd";

byte willQoS            = 0;
boolean willRetain      = false;
const char* willTopic   = "willTopic";
const char* willMessage = "My Will Message";

//IP-settings & ethernet initialization - Change these!
EthernetClient ethernetClient;
byte mac[]        = {0xDE, 0xED, 0xBA, 0xFE, 0xFE, 0xED };
byte ip[]         = {172, 31, 14, 194};
byte nameserver[] = {172, 31, 14, 153};
byte gateway[]    = {172, 31, 14, 129};
byte subnet[]     = {255, 255, 255, 128};

const int temperaturePin = 0;
long lastReconnectAttempt = 0;
float voltage, degreesC;
char bigstring[40];  // enough room for all strings together (topic generation)

void setup()
{
    Serial.begin(9600);
    while (!Serial) {}
    Serial.println("Serial is online");

    // Mqtt client setup
    mqttClient.setClient(ethernetClient);
    mqttClient.setServer(broker, 1883); // 1883 - is the default port for mqtt
(no ssl)
    mqttClient.setCallback(callback); // Add subscription callback to get
messages

    // Function that connects to the network
    Serial.println("Starting ethernet connections");
    reconnect();
}
```

```
void startEthernet()
{
    //Start from scratch
    ethernetClient.stop();
    delay(5000);
    if (Ethernet.begin(mac) == 0) {
        Serial.println("Failed to configure Ethernet using DHCP");
        Serial.println("Try connecting using static network settings...");
        Ethernet.begin(mac, ip);
    }
    else {
        Serial.println("DHCP Connected successfully.");
        printIPAddress();
        delay(2000);
    }
}

boolean reconnect()
{
    // First restart the ethernet connection
    startEthernet();
    boolean rc = mqttClient.connect(clientID, username, password, willTopic,
willQoS, willRetain, willMessage);
    delay(5000);
    // Check if connection was successful
    if (rc)
    {
        // connection succeeded
        // Subscribe to the incoming messages of this device
        Serial.println("Connected to the broker. ");
        char* tempTopic = join2Strings(clientPath, "/inTopic/#");
        boolean rc = mqttClient.subscribe(tempTopic, qos);

        if(!rc) { Serial.println("Subscription failed for some reason."); }
        else { Serial.println("Subscription successful for incoming messages ");
}
    }
    else
    {
        // connection failed
        Serial.println("Connection failed to the broker.");
        Serial.println("Starting ethernet connections");
    }
    return mqttClient.connected();
}
```

```
// put your main code here, to run repeatedly:
void loop()
{
    // Maintain DHCP
    Ethernet.maintain();

    // Read analog pin voltage for temperature
    voltage = getVoltage(temperaturePin);
    degreesC = (voltage - 0.5) * 100.0;

    // Check if client is connected, if not try to connect again
    if (!mqttClient.connected()) {
        Serial.println("MQTT Client not connected! :(");
        long now = millis();
        if (now - lastReconnectAttempt > 5000) {
            lastReconnectAttempt = now;
            // Attempt to reconnect
            Serial.println("Attempting to reconnect...");
            if (reconnect()) {
                lastReconnectAttempt = 0;
            }
        }
    } else {
        // Client is connected
        // Convert topic and temperature to CHAR*
        char tempPayload[0];
        char* tempTopic;
        dtostrf(degreesC, 8, 2, tempPayload);
        tempTopic = join2Strings(clientPath, "/outTopic/temperature"); //
generate topic by joining two strings

        mqttClient.publish(tempTopic, tempPayload, qos);  // Publish message

        // Maintain MQTT client
        mqttClient.loop();
    }
}

void callback(char* topic, byte* payload, unsigned int length)
{
    // handle received message
    Serial.print("Message arrived [");
    Serial.print(topic);
    Serial.print("] ");

    for (int i=0;i<length;i++) {
      Serial.print((char)payload[i]);
    }
    Serial.println();
}

char* join2Strings(char* string1, char* string2) {
   bigstring[0] = 0;          // start with a null string:
   strcat(bigstring, string1);   // add first string
   strcat (bigstring, string2);
   return bigstring;
}
```

```
void printIPAddress()
{
  Serial.print("My IP address: ");
  for (byte thisByte = 0; thisByte < 4; thisByte++) {
    // print the value of each byte of the IP address:
    Serial.print(Ethernet.localIP()[thisByte], DEC);
    Serial.print(".");
  }
  Serial.println();
}

float getVoltage(int pin)
{
  return (analogRead(pin) * 0.004882814);
}
```

Liite 3. Python -sovellus, joka tallentaa viestienvälityspalvelimen viestit tietokantaan.

```
#!/usr/bin/python
#-*-coding: utf-8-*-
import json
from MongoDB import MongoDB
from Mqtt import Mqtt

run = True

# MongoDB Class configuration
mongo_url  = '192.168.202.2'
database   = 'mqtt'
collection = 'broker01'

# Luo uusi instanssi mongodb luokasta
mongoClient = MongoDB(mongo_url, database, collection)

# MQTT Class configuration
mqttAuth      = {'username':"admin", 'password':"Passw0rd"}
broker        = '192.168.202.83'
qos           = 1
clientID      = 'dclab-database'
clean_session = True
# Topic -arvo jonka mukaan tilaukset tehdään
subTopic = '#'

def on_connect(client, userdata, flags, rc):
    print("Connected with result code "+str(rc))
    mqttClient.subscribe(subTopic, qos)

# Callback kuuntelee tietoa palvelimelta subscriben perusteella, tulostaa
viestit.
def on_message(client, userdata, msg):
    #print(msg.topic+" "+str(msg.payload))
    # HUOM Tämä koodi olettaa, että topic rakenteen viimeisenä on varsinaista
arvoa kuvaava nimi!

    # Aseta kokoelma topic -arvon perusteella, mutta poista viimeinen taso
    mongoClient.set_collection(str(msg.topic))

    # Erottele ja muuta oikeaan muotoon viimeinen topic -taso ja arvo
    key = msg.topic.rpartition('/')[2]
    value = float(msg.payload)
    data = {key:value}

    print data
    # Syötä rakennettu arvo!
    mongoClient.insert_one(data)
    print "inserted"


# Luo uusi instanssi mqtt luokasta
mqttClient = Mqtt(broker, qos, clientID, clean_session, mqttAuth, subTopic,
on_connect, on_message)

while run:
  mqttClient.maintain()
```

```
#!/usr/bin/python
#-*-coding: utf-8-*-
import pymongo
import datetime

'''

Saves and queries data form and to MongoDB.

Made by Antero Juutinen

'''

class MongoDB:

  def __init__(self, mongodb_uri, database, collection):
    try:
      self._client = pymongo.MongoClient(mongodb_uri)
      self._database = self._client[database]
      self._collection = self._database[collection]
    except pymongo.errors.ConnectionFailure, e:
      print "Tietokantayhteys epäonnistui: %s" % e


  def client(self):
    return self._client


  def set_database(self, database):
    if self._database != database:
      self._database = self._client[database]


  def set_collection(self, collection):
    self._collection = self._database[collection]


  def find_one(self, query):
    return list(self._collection.find_one(query))


  def insert_one(self, data):
    return self._collection.insert_one(data)
```

```
#!/usr/bin/python
#-*-coding: utf-8-*-
'''
Model for serialport read and write.

Wrapperi pyserial moduulille. Näin funktiot voidaan laittaa eri tiedostoon ja
koodi on luettavampaa.

'''
import json
import paho.mqtt.client as mqtt


class Mqtt:

    '''konstruktori MQTT -clientille, jonka avulla saamme yhteyden brokeriin.'''
    def __init__(self, broker, qos, clientID, clean_session, userdata, subTopic,
on_connect, on_message):
        self._clientID      = clientID
        self._clean_session = clean_session
        self._userdata      = userdata
        self._broker        = broker
        self._subs          = set()
        self._qos           = qos
        self._subTopic      = subTopic

        self._client = mqtt.Client(clientID, clean_session, userdata)
        self._client.username_pw_set(userdata['username'], userdata['password'])

        self._client.on_connect = on_connect
        self._client.on_message = on_message
        self._client.connect(broker, 1883, 60)


    def subscribe(self, topic, qos):
        self._client.subscribe(topic, self._qos)

    def loop_start(self):
        self._client.loop_start()

    def loop_stop(self):
        self._client.loop_stop()

    def maintain(self):
        self._client.loop()
```