Paavo Rantala

GraphQL esineiden internetissä

Tietotekniikan
Pro gradu -tutkielma
21. maaliskuuta 2022

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Kokkolan yliopistokeskus Chydenius

Tekijä: Paavo Rantala
Yhteystiedot: paavorantala94@gmail.com
Puhelinnumero: 0405595377
Ohjaaja: Ismo Hakala
Työn nimi: GraphQL esineiden internetissä
Title in English: GraphQL in the Internet of Things
Työ: Tietotekniikan Pro gradu -tutkielma
Sivumäärä: 94
Tiivistelmä: Esineiden internetin laajentumisen yhteydessä käytettävissä olevien laitteiden ja teknologioiden lukumäärä on kasvanut. Samalla verkkoinfrastruktuurien kuormitus on lisääntynyt, mikä heijastuu väistämättöminä ongelmina resurssirajoittuneissa systeemeissä. Järjestelmällisellä standardoinnilla on onnistuttu parantamaan eri tekniikoiden yhteensopivuutta ja palvelukeskeinen arkkitehtuuri on edistänyt verkkojen skaalautuvuutta. Erityisesti web-palveluna tunnettu REST-arkkitehtuuri on saavuttanut suosiota esineiden internetissä ja mahdollistanut aikaisemmin opitun siirtämisen uuteen laiteympäristöön. HTTP:tä kevyempi CoAP-protokolla on ollut ominaisuuksiltaan järkevä vaihtoehto REST-arkkitehtuurin tuomiseksi esineiden internetiin. Vastaavasti kuten REST, GraphQL on web-palveluna kehittynyt spesifikaatio rajapinta-arkkitehtuurille, jonka tärkeimmät osat ovat kyselykieli ja ajonaikainen järjestelmä. Spesifikaation suunnittelua on lähestytty asiakkaan tarpeista, pyrkien RESTiä joustavampaan viestintään. GraphQL on varteen otettava vaihtoehto esineiden internetille, sillä sen ominaisuuksilla on edellytyksiä vähentää energiankulutusta ja parantaa järjestelmien integroimista keskenään. Lisäksi GraphQL:n kommunikaatiomallit tukevat RESTiä kattavammin muitakin IoT-protokollia kuin CoAPia. Tässä tutkielmassa selvitetään GraphQL:ää ja esineiden internetiä käsittelevissä tutkimuksissa tehtyjä havaintoja ja niiden perusteella kartoitetaan aiheen nykytilannetta. Tutkimuksista opitun perusteella toteutetaan yksinkertainen IoT-järjestelmä, jossa web-sovellukseen haetaan mittausdataa IoT-laitteelta ja toisen osapuolen hallitsemasta web-palvelusta hyödyntäen GraphQL:ää osana arkkitehtuuria.
Avainsanat: GraphQL, esineiden internet, palvelukeskeinen arkkitehtuuri, protokolla, rajapinta, REST, spesifikaatio, standardi, väliohjelmisto, web
Abstract: As the Internet of Things has expanded, the number of usable devices and technologies has increased as well. The network infrastructure has gained more stress that reflects inevitable problems in resource constrained systems. Well-

organized standardizing has accomplished to improve the integration of different techniques and service-oriented architecture has enhanced the network scalability. Especially known as a web service, REST has gained popularity in the Internet of Things and made it possible to reuse the learned knowledge in a new device environment. Regarding its features, CoAP protocol has been a reasonable lighter alternative to HTTP for bringing REST into the Internet of Things. As with REST, GraphQL is another interface architecture specification that has matured as a web service whose most important parts are a query language and an execution engine. The specification tries to achieve a more flexible communication than in REST by approaching the design from the needs of the clients. GraphQL is a considerable alternative to the Internet of Things because its features possess implications to reduce energy consumption and to achieve a better integration among systems. Additionally, the communication models of GraphQL have a more comprehensive support for other IoT protocols than only CoAP. In this study the results of the research made about GraphQL and IoT are examined and used to depict the present understanding about the subject. A simple IoT system is developed by what is learned from the research. GraphQL is used as a part of the system architecture to fetch measurement data from an IoT device and a web service owned by a different party to be shown in a web application.
Keywords: GraphQL, interface, middleware, protocol, REST, service-oriented architecture, specification, standard, the Internet of Things, web

Copyright © 2022 Paavo Rantala

All rights reserved.

Sanasto

6LoWPAN	IPv6 over Low-Power Wireless Personal Area Networks
API	Application Programming Interface
CoAP	Constrained Application Protocol
CoRE	Constrained RESTful Environments
DTLS	Datagram Transport Layer Security
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
IETF	The Internet Engineering Task Force
IoT	Internet of Things
IP	Internet Protocol
JSON	JavaScript Object Notation
M2M	Machine to Machine
MQTT	Message Queueing Telemetry Transport
REST	Representational State Transfer
RFC	Request for Comments
ROLL	Routing Over Low power and Lossy networks
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WWW	World Wide Web
XML	Extensible Markup Language

Sisällys

Sanasto i

1 Johdanto 1

2 Esineiden internet 3

2.1 Sensoriverkot 5

2.2 Standardit 6

2.2.1 6LoWPAN-työryhmä 7

2.2.2 ROLL-työryhmä 8

2.2.3 CoRE-työryhmä 8

3 Verkkoprotokollat 11

3.1 Asiakas-palvelin-arkkitehtuuri 11

3.2 HTTP 13

3.3 CoAP 16

3.3.1 Protokolla 17

3.3.2 CoAP verkon protokollana 19

3.3.3 CoAPin kehittäminen 21

3.4 Muut IoT-protokollat 23

3.4.1 MQTT 24

3.4.2 XMPP 25

3.4.3 AMQP 25

3.4.4 DDS 26

3.5 Protokollien vertailu 26

4 Palvelukeskeinen arkkitehtuuri 31

4.1 Web-palvelut 33

4.2 REST-arkkitehtuuri 34

4.3 Web-palvelut esineiden internetissä 35

5 GraphQL **40**
5.1 Historia ja motivaatio . . . 41
5.2 Kyselykielisyntaksi . . . 42
5.2.1 Tiedonhaku . . . 44
5.2.2 Tiedonmuokkaaminen . . . 46
5.2.3 Skeema . . . 47
5.3 Arkkitehtuuri . . . 48
5.4 Selvittäjäfunktiot . . . 50
5.5 Operaatioiden suoritus . . . 52
5.5.1 Validointi . . . 57
5.6 Introspektio . . . 57
5.7 GraphQL ja web-sovellukset . . . 59
5.7.1 GraphQL vs. REST . . . 60
5.8 GraphQL ja esineiden internet . . . 62
5.9 Puolesta ja vastaan . . . 65

6 IoT-järjestelmän demonstraatio **68**
6.1 Mittalaite . . . 70
6.2 Yhdyskäytävä . . . 71
6.3 Pilvipalvelu . . . 72
6.4 Web-sovellus . . . 75

7 Pohdinta **79**

8 Yhteenveto **82**

Lähteet **83**

1 Johdanto

Yli 20 vuotta sitten Kevin Ashton [6] toi ilmi käsitteen esineiden internet (Internet of Things, IoT). Tuosta hetkestä tähän päivään esineiden internet on kohdannut suurta mielenkiintoa, merkittävää kasvua ja eittämättä myös haasteita. Uusien toimijoiden tarttuessa aiheeseen, joko tieteellisestä tai kaupallisesta näkökulmasta, on syntynyt laaja valikoima erilaisia käytänteitä, teknologioita sekä laitteita. Samalla kun valinnanvapaus käytettävissä tekniikoissa on lisääntynyt, niin järjestelmien yhteensovittaminen keskenään on vaikeutunut. Yhteisiä standardeja suunnittelemalla on luotu hyviä käytänteitä laitteiden valmistajille ja sovellusten kehittäjille, mutta sääntöjä on kuitenkin tarpeen päivittää ja omaksua uusia.

Monet esineiden internetin sovelluksista on tuotu lähelle ihmisten jokapäiväistä elämää. Alkukustannusten pienentyminen ja oppimiskäyrän loiventuminen on tuonut IoT-järjestelmien kehityksen myös kuluttajien saataville. Edistystä on ajanut lisäksi järjestelmien käytön ja hallinnan helpottuminen yhdistämällä ne julkiseen internetiin. Järjestelmien hallinta ja valvonta web-sovelluksina on tehnyt niistä lähes kaikkialla saavutettavia laitteesta riippumatta. Esineiden internetin kasvu ja levittäytyminen on kuitenkin lisännyt tiedonsiirron tarvetta räjähdysmäisesti ja teknologian kehityksestä huolimatta on ollut syytä panostaa myös jo olemassa olevan suorituskyvyn ja infrastruktuurin optimointiin.

IoT-järjestelmät ovat integroitavissa osaksi laajempaa kokonaisuutta erilaisten rajapintojen kautta. Hyvin tyypillinen rajapintamalli sekä esineiden internetissä että web-sovelluksissa on REST-rajapinta. Vaikka REST-rajapinnat ovat sementoineet asemansa luotettavana arkkitehtuurina, niin myös niiden ongelmiin tulee suhtautua kriittisesti ja pohtia ratkaisuja ilmenneisiin haasteisiin. GraphQL on web-sovelluksissa suosiotaan kasvattanut spesifikaatio yhdistämään asiakkaat ja palvelimet keskenään ja monet sen ominaisuuksista voidaan nähdä hyödyllisinä myös esineiden internetissä. GraphQL:n ominaisuuksia tiedonsiirron optimoinnissa ja siitä saavutettavassa energiansäästössä on tutkittu vielä melko vähän esineiden internetin kontekstissa. Monet aiheesta julkaistut tutkimukset ovat laajuudeltaan suhteellisen pieniä, niiden testausasetelmien koostuessa alle kymmenestä laitteesta. Toteutetut tutkimukset ja suunnitelmat ovat kuitenkin osoittaneet GraphQL:n hyödylliseksi ta-

vaksi yhdistämään heterogeeniset laitteet ja järjestelmät keskenään. Tässä tutkielmassa nostetaan esille aiheesta julkaistuissa tutkimuksissa ja muissa julkaisuissa todettuja tuloksia ja havaintoja. Lopuksi demonstroidaan kerätystä aineistosta opittua suunnittelemalla ja toteuttamalla yksinkertainen IoT-järjestelmä.

Tämä tutkielma jakaantuu seuraaviin osiin. Luvussa kaksi tutustutaan lyhyesti esineiden internetiin ja langattomiin sensoriverkkoihin. Samassa luvussa selvitetään myös esineiden internetiä varten tehtyä standardointityötä painottaen kommunikaatioprotokollia. Sen jälkeen luvussa kolme syvennytään tarkemmin IoT-järjestelmissä suosittuihin kommunikaatioprotokolliin. Luvusta neljä lähtien käsiteltävä aihe siirtyy enemmän web-sovelluksiin aloittaen palvelukeskeisestä arkkitehtuurista ja web-palveluista. Aiheen käsittely jatkuu luvussa viisi mielenkiinnon suuntautuessa GraphQL:n pariin. Luvussa kuusi demonstroidaan GraphQL:n käyttöä yhdistämään esineiden internet ja web-sovellukset. Tutkielma päättyy luvussa seitsemän esitettyyn pohdintaan ja luvun kahdeksan yhteenvetoon.

2 Esineiden internet

Esineiden internet on kasvanut tärkeäksi ja suureksi teemaksi jokapäiväisessä elämässämme. Teknologian sovelluskohteet ovat monipuolisia ja kattavat elämän osa-alueita mm. kotiautomaatiosta logistiikkaan ja teollisuudesta opiskelun tehostamiseen. Tutkimuksessa [7] siirtyminen kohti yhä laajempaa esineiden internetin hyödyntämistä kuvaillaan tapahtuvan sijoittamalla ympäristöömme laitteita ja esineitä, jotka varustetaan riittävällä älykkyydellä ja tarvittavilla ominaisuuksilla pystyäkseen tuottamaan tietoa ja osallistumaan vuorovaikutukseen. Esineet voivat pyrkiä toimimaan itsenäisesti, mutta hyvin usein ne tekevät yhteistyötä viestimällä keskenään ja tunnistaen toisensa käyttötarkoitusta varten suunnitellun osoitejärjestelmän perusteella. Käyttökohteiden ohella myös laitevalikoima rikastuu jatkuvasti. Nykyään kuluttajille on tarjolla monenlaisia laitteita, kuten erilaisia sensoreita ja toimilaitteita usealta eri valmistajalta. Voimakas kysyntä, teknologian kehitys ja positiiviset tulevaisuudennäkymät ovat pitäneet esineiden internetin vahvana ehdokkaana ajamaan talouden kasvua. Verkkojen levittäytyessä ja nopeampien sekä tehokkaampien yhteyksien muodostuessa tietoturvallisuuteen kohdistuvat uhat ovat kuitenkin väistämättömiä joko tarkoituksellisina tietomurtoina tai tekniikassa ilmenneinä odottamattomina virheinä.

Esineiden internetin kuvaamalla mallilla on onnistuttu kehittämään suuri määrä sovelluksia elämänlaadun parantamiseksi tunnistamalla elinympäristöstämme esineitä, joilta ennestään puuttuu älykkäitä ominaisuuksia. Nämä esineet voidaan varustaa mikropiireillä ja muilla teknisillä laitteilla ottaen ne osaksi alati kasvavaa laiteverkkoa. Atzori et al. [7] nostavat yhdeksi haasteeksi kuitenkin sovelluskohteiden ja laitteiden rajoittuneiden resurssien yhteensovittamisen. Laskennallinen teho, tiedolle varattu tila ja käytettävä energia on tarve mukauttaa muuttuviin ympäristöihin ja olosuhteisiin saavuttaen laitteille riittävä älykkyys, jotta ne kykenisivät toimimaan yhdessä. Samalla tulee huomioida luottamus, yksityisyys ja turvallisuus sekä verkon laajentuessa varautua skaalautumiskyvyssä ilmeneviin ongelmiin. Lisäksi tämä kaikki tulee saavuttaa kustannustehokkaasti kilpailun kiristyessä arvokkaista raaka-aineista. On ennustettu, että tulevaisuudessa yhä useampi laite on yhdistetty keskenään internetin avulla, mikä johtaa väistämättä ongelmiin laitteiden keräämän

tiedon esityksessä, tallennuksessa ja hallinnassa.

Vaikka laitteet kattavat käyttötarkoituksillaan eri ympäristöjen, olosuhteiden ja tehtävien asettamia vaatimuksia, niin niiden laitteistoarkkitehtuurista pystytään komponenttien osalta tunnistamaan yhteneviä yksiköitä. Arkkitehtuurin tavoitteita ja komponenttien rooleja kuvaillaan kirjassa [59, s.3 – 4]. Arkkitehtuuri pyrkii mahdollistamaan yhteistyön, vuorovaikutuksen ja kommunikaation muiden verkkoon asennettujen laitteiden kanssa. Jokaisella komponentilla on oma tärkeä roolinsa ja niistä merkittävimpiin lukeutuvat mikroprosessori, kommunikaatioyksikkö, virtalähde, sensorit ja toimilaitteet. Mikroprosessorin vastuulla on suorittaa laskentaa, hallintaa ja käskytystä laitetasolla rajoittuneen kapasiteettinsa puitteissa. Kommunikaatioyksikkö mahdollistaa verkostoitumisen ja yhteistyön muiden laitteiden kanssa tapahtuvan viestinnän avulla. Kommunikaation välttämättömyys älykkäille laitteille on tunnistettu ja sitä pidetään edellytyksenä niistä saatavan todellisen hyödyn saavuttamiseksi. Eikä yhteydenpito rajoitu ainoastaan laitteiden kesken tapahtuvaksi. Tapahtumista, kuten havaituista liikkeistä vartioidulla alueella, voidaan ilmoittaa myös ihmisille. Virtalähde, joka voi olla akku, paristo tai myös verkkovirta, ylläpitää laitteen toimintoja. Sensoriensa avulla laitteet pystyvät aistimaan fyysistä maailmaa ja seuraamaan ympäristönsä tapahtumia havainnoimalla lämpötilaa, ilman kosteutta tai muita fysikaalisia ominaisuuksia. Toimilaitteet mahdollistavat reagoimisen ympäristössä tapahtuneeseen muutokseen: havaitusta liikkeestä sytytetään valo tai ilmalämpöpumppu käynnistetään hellerajan rikkoutuessa. Sensorit ja toimilaitteet tekevät laitteista vuorovaikutteisia ympäristönsä kanssa.

Kirjassaan [59, s. 15] Vasseur ja Dunkels toteavat, että IoT-järjestelmien suunnittelussa on havaittu monia haasteita ja ne voidaan hyvin karkeasti jakaa koko järjestelmän laajuisiksi tai laitekohtaisiksi. Joitakin erityisesti laitetasolla esille nousevia haasteita ovat virrankulutus, laitteen koko ja hinta. Virrankulutukseen voidaan vaikuttaa sekä laitteisto- että ohjelmistovalinnoilla. Vähän virtaa kuluttavat ja tehokkaat komponentit yhdessä edullisen lepotilan kanssa ovat suoraviivaisia laitteistotason menetelmiä virrankulutuksen minimoimiseksi. Ohjelmisto hallinnoi komponenttien toimintaa ja tiloja. Myös käytetyillä protokollilla ja verkkoarkkitehtuureilla voidaan vaikuttaa tehokkuuteen.

Tyypillisesti laitteiden toivotaan sulautuvan ympäristöönsä tai ne voivat sijaita paikoissa, joihin pääsy voi olla haastavaa. Teollisuudessa joitain mittalaitteita asennetaan usein prosessilinjaston koneiden sisälle keräämään tietoa tuotannon kulusta. Laitteen ja sen komponenttien koko saattavat asettua ratkaiseviksi tekijöiksi käy-

tettävyydelle tai lopputuloksen hyväksynnälle. Kokonaiskustannuksilla on tapana laskea tuotannon kehittyessä ja suurempia raaka-aineiden toimituseriä tilattaessa. Laitteiden lukumäärän kasvaessa sovellusjärjestelmän kustannukset ovat kuitenkin vaarassa kasvaa huomattaviin summiin. Myös suunnittelussa on suhteutettava käyttötarkoitukseen tehtävät investoinnit ja siitä saatava tuotto. Esineen älykkääksi muuttava mikropiiri saattaakin olla kauppahinnaltaan arvokkaampi kuin itse esine.

Esineiden internetin tulkinnasta ja sen merkityksestä voidaan esittää vaihtelevia näkökantoja. Artikkelissa [7] on koostettu erilaisia lähestymistapoja, joista erityisesti CASAGRAS-yhtymän hahmottelema kuvaus antaa laajemman perspektiivin tässä työssä käsiteltyyn tapaukseen: esineiden internet koostuu automatisoidusti toisilleen ja tietokoneille kommunikoivista laitteista, jotka tarjoavat palveluita ihmisten elämän helpottamiseksi. Esineiden internet on yleismaailmallinen infrastruktuuri, joka yhdistää virtuaaliset ja fyysiset esineet keskenään nykyisessä ja kehittyvässä tietoverkkorakenteessa ja julkisessa internetissä.

2.1 Sensoriverkot

Erääksi merkittäväksi kokonaisuudeksi esineiden internetissä on tunnistettu sensoriverkot, joiden rakennetta ja toimintaa kuvataan artikkelissa [7]. Sensoriverkot rakentuvat noodeiksi kutsutuista laitteista, jotka kykenevät esimerkiksi seuraamaan kappaleiden tilaa ja luomaan tietoisuutta vallitsevasta ympäristöstä sensoriensa avulla. Noodit viestivät keskenään usein langattomasti ja monihyppymenetelmällä keräten tietoa sinkiksi kutsutuille noodeille, jotka välittävät tiedon eteenpäin ulkomaailmaan raportoitavaksi. Sensoriverkoille on kehitetty lukuisia sovelluksia fyysisen ja digitaalisen todellisuuden yhdistämiseksi mm. ympäristönvalvonnassa, terveydenhuollossa, logistiikassa ja teollisuuden prosesseissa.

Atzori et al. [7] selventävät, että tavallisesti vastuu laitteiden toiminnallisuudessa on jaettu protokollapinoilla kuvattuihin arkkitehtuureihin. Yhtä jokaiseen sovellukseen sopivaa ratkaisua ei ole tunnistettu, vaan arkkitehtuurien on käyttökohteesta riippuen otettava kantaa ainakin energiatehokkuuteen, skaalautuvuuteen, luotettavuuteen ja järjestelmän kestävyyteen. Useimmat sensoriverkkojärjestelmät toteuttavat IEEE 802.15.4 -standardin mukaisen verkkoarkkitehtuurin. Standardi määrittelee fyysisen ja MAC-kerroksen ominaisuudet, mutta ei ota kantaa niitä ylempiin kerroksiin.

Koska sensoriverkkojärjestelmät koostuvat hyvin usein lukumäärältään suures-

ta ja monipuolisesta joukosta erilaisia esineitä, niin artikkelissa [7] esitetään tarve abstraktiokerrokselle. Esineiden vaihtelevat toiminnallisuudet ja ymmärtämä semantiikka yhdistetään abstraktiokerroksen avulla yhteiseksi kieleksi ja proseduureiksi. On myös tapauksia, joissa tarvitaan oma rajapinta- ja kommunikaatiokerroksensa laitteiden löytämiseksi.

Yhtenä menetelmänä abstraktiokerroksen toteuttamiselle Atzori et al. [7] esittävät väliohjelmistojen käytön. Väliohjelmistot kuvataan usein omana kerroksenaan sovelluskerroksen ja muiden kerroksien välille. Kerroksen tarkoituksena on yksinkertaistaa uusien ja vanhojen palveluiden sekä järjestelmien integroimista keskenään. Palvelukeskeinen arkkitehtuuri on osoittautunut suosituksi menetelmäksi väliohjelmistorakenteiden suunnittelussa. Sitä noudattamalla monimutkaiset ja monoliittiset järjestelmät voidaan jakaa pienempiin, yksinkertaisempiin ja paremmin määriteltyihin sovelluksiinsa. Sensoriverkkojärjestelmissä sovellukset sijoittuvat usein arkkitehtuurin ylimmälle kerrokselle. Väliohjelmistoja, standardoituja protokollia ja muita palveluteknologioita hyödyntäen sovellukset voivat muodostaa täydellisen integraation hajautettujen järjestelmien ja niihin sijoitettujen sovellusten kesken.

2.2 Standardit

Vasseur ja Dunkels kertovat kirjassaan [59, s. 183 – 186] ytimekkäästi standardisoinnin tärkeydestä ja standardointityön seurauksena syntyvien asiakirjojen merkityksestä. Standardisoinnilla halutaan taata avoimuuden ja yhteistoimivuuden turvaaminen eri valmistajien, kehittäjien, tuotteiden ja palveluiden välillä. Standardisointiprosessissa valmistuvat dokumentit kuvaavat käsiteltävän aiheen määritelmineen ja antavat ohjeita, joita noudattamalla järjestelmien sovittaminen keskenään voidaan varmistaa. Esineiden internetiä varten on kehitetty lukuisia yksityisten toimijoiden hallitsemia ratkaisuja, mutta avointen standardien omaksuminen on erittäin tärkeää sekä edellytys skaalautuvan arkkitehtuurin suunnittelulle.

Kirjassa [59, s. 183 – 186] huomioidaan Internet Engineering Task Force (IETF), joka on vuonna 1986 perustettu kansainvälinen ja avoin standardisoinnista vastaava organisaatio, jonka jäsenet ovat yksityishenkilöitä ilman yrityskytköksiä. IETF muodostuu useista työryhmistä, jotka keskittyvät eri verkkoarkkitehtuurikerroksille suunniteltuihin protokolliin. Organisaatiossa on omat työryhmänsä mm. sovellus-, kuljetus- ja reititysprotokollille. IETF on asettanut tehtäväkseen korkealaatuisten

ja asiaankuuluvien teknisten dokumenttien tuottamisen, jotka vaikuttavat siihen, kuinka ihmiset suunnittelevat, käyttävät ja hallinnoivat internetiä. Perimmäisenä päämääränä dokumenteilla on taata internetin hyvä toimivuus. Kirjalliset esitykset muotoillaan RFC-dokumenttien (Request for Comments) muotoon työryhmien toimesta.

IETF:n toimintaa ohjaamaan on valittu aatteita, jotka Vasseur ja Dunkels [59, s. 183 – 186] selostavat. Kuka tahansa on vapaa osallistumaan standardointiprosessiin ilman esiehtoja ja käsiteltäväksi otetaan ainoastaan asioita, joihin jäsenillä on riittävä tekninen osaaminen. Jokainen jäsen osallistuu työskentelyyn vapaaehtoisesti ja suunnittelussa tehtävät päätökset perustuvat toimiviin ratkaisuihin sekä jäsenten väliseen yhteisymmärrykseen. Standardisointiprosessin laadun takaamiseksi on asetettu joitakin tavoitteita. Menetelmissä halutaan painottaa teknistä erinomaisuutta, aikaista toteutusta ja testausta. Tuotetulta dokumentaatiolta toivotaan selkeyttä, järjestelmällisyyttä ja ajantasaisuutta. Lisäksi prosessin etenemisen tulee olla avointa ja reilua.

2.2.1 6LoWPAN-työryhmä

Ensimmäisenä merkittävänä työryhmänä esineiden internetiä varten kirjassa [59, s. 191 – 197] esitellään vuonna 2004 perustettu IPv6 over Low-power WPAN (6LoWPAN). Työryhmä vastaa protokollaspesifikaatioiden määrittämisestä IPv6-protokollan optimoimiseksi IEEE 802.15.4 -standardin mukaisille linkeille, jotka toimivat LoWPAN-verkoissa. LoWPAN-verkkojen oleellisimpiin ominaisuuksiin kuuluu pakettien pieni koko, 16-bittisten osoitteiden tuki ja linkkien vaatimaton kaistanleveys. 6LoWPAN-työryhmän tavoitteita ovat mm. pakettien paloittelun ja uudelleenkokoamisen suunnittelu, otsakkeiden tiivistys ja osoitteiden autokonfigurointi. Työryhmä ottaa kantaa myös verkonhallintaan ja muihin toteutuksessa huomioitaviin asioihin, kuten turvallisuuteen.

Heinäkuussa 2007 nostettiin esille tarve lisätä työskentelyä älykkäiden järjestelmien IP-pohjaisissa protokollissa. Tarkoituksena on aina ollut jo olemassa olevien IP-protokollien hyödyntäminen, kunhan se vain on mahdollista. Joitain protokollia, kuten UDP ja TCP, voitiin ottaa käyttöön ilman muutoksia, mutta myös uusien protokollien kehitykselle on ollut tarvetta. 6LoWPAN-työryhmän vastuulla oleva IPv6:n käyttäminen IEEE 802.15.4 -verkossa on vaatinut lisäyksiä ja sopeuttamista IPv6-pakettien lähettämisen optimoimiseksi.

2.2.2 ROLL-työryhmä

Reititys on aina ollut tärkeä osa verkkojen toimintaa ja useita IP-osoitteisiin perustuvia protokollia on suunniteltu tätä varten. Osa protokollista on tarkoitettu verkkojen sisäiseen reititykseen ja toiset verkkojen välillä tapahtuvaan reititykseen. Low-power and Lossy Networks (LLN) verkkojen ominaisuudet perustelivat tarpeen Routing Over Low Power and Lossy Networks (ROLL) työryhmän perustamiselle maaliskuussa 2008, mikä on toinen tärkeä kirjassa [59, s. 191 – 197] käsitelty rajoittuneiden verkkojen kehittämisestä vastaava työryhmä. Työryhmän alkuperäinen tarkoitus oli tuottaa yksityiskohtaisia reititysvaatimuksia ja arvioida jo olemassa olevien protokollien käytettävyyttä LLN-verkoissa. Vaikka LLN-verkkojen käyttökohteita on runsaasti, niin kehitystyön tarkentamiseksi ja vaatimusten vähentämiseksi protokollien suunnitellussa rajoituttiin neljään pääsovellukseen. Pääsovelluksiksi valikoitui teollinen-, koti- ja rakennusautomaatio sekä kaupunkiverkot. Mitään osa-aluetta kaikista mahdollisista käyttökohteista ei haluttu tietoisesti jättää pois, vaan keskittyä pääsovelluksiin, koska monet niiden ratkaisuista kattaisivat myös muiden sovellusalueiden ongelmat. ROLL asetti reititysprotokollille joitain vaatimuksia ja mittareita: skaalautuvuus tilojen, linkkien sekä noodien suhteen, kyky vastata paikallisesti häviöihin, hallintakustannuksista aiheutuva kuorma ja linkkien sekä noodien kustannusten arviointi reitityspäätöksissä. Arvioidessaan jo olemassa olevia protokollia työryhmä päätyi lopputulokseen, että mikään niistä ei täytä kaikkia vaatimuksia. Työryhmä järjestäytyi uudelleen ja muotoili alueet, joihin sen työskentely kohdistuu: uusien protokollien määritteleminen sekä vanhojen laajentaminen, reititysmetriikka, turvallisuus, hallinta, arkkitehtuurimalli ja käyttöönoton dokumentaatio.

6LoWPAN-työryhmä käyttää termiä LoWPAN, kun taas ROLL-työryhmä suosii yleisempää Low-power and Lossy Networks (LLNs) termiä, joiden tarkoitusta Vasseur ja Dunkels [59, s. 191 – 197] täsmentävät seuraavasti. Termit ovat jokseenkin vastaavia viitattaessa rajoittuneista laitteista muodostettuihin verkkoihin. Merkittävin ero on, että LLN ei rajoitu IEEE 802.15.4 -standardin mukaisiin linkkeihin, vaan se kattaa myös muut vähävirtaiset linkit esimerkiksi WiFi-verkoissa.

2.2.3 CoRE-työryhmä

Rajoittuneita RESTful-ympäristöjä varten IETF on perustanut Constrained Restful Environments (CoRE) työryhmän, jonka tärkein tehtävä on peruskirjassa [27] ku-

vatun kehysmallin kehittäminen. CoREn suunnittelema kehysmalli resurssiorientoituneille sovelluksille on tarkoitettu rajoittuneita IP-verkkoja varten. Rajoittuneissa IP-verkoissa pakettien koko on normaaliverkkoja pienempi, niiden katoamistodennäköisyys saattaa olla suuri ja laitteet voivat vaihdella aktiivisen tilan ja lepotilan välillä. Verkoille ja niiden noodeille on tunnusomaista rajoitettu suoritusteho ja käytettävä virta sekä sovellusten kompleksisuuden rajaaminen noodien muistin ja koodin viemän tilan mukaan. Rajoittuneet verkot voivat esiintyä osana koti- ja rakennusautomaatiota, energianhallintaa ja esineiden internetiä.

Peruskirja [27] kertoo työryhmän määrittelemän kehysmallin käyttökohteista seuraavasti. Kehysmalli on tarkoitettu rajoitetulle lukumäärälle sovelluksia. Sovellukset keskittyvät yksinkertaisten resurssien käsittelyyn rajoittuneissa verkoissa. Näihin sovelluksiin lukeutuvat yksinkertaisten sensorien, kuten lämpötila-anturien tai valokytkinten valvonta, toimilaitteiden käyttö sekä laitteiden hallinta.

Työryhmä [27] kuvailee kehysmallin arkkitehtuurin yleisellä tasolla koostuvan noodeista, eli laitteista, rajoittuneessa verkossa. Laite on vastuussa yhdestä tai useammasta resurssista, joka voi esittää sensoria, toimilaitetta, arvojen yhdistelmää tai jotain muuta tietoa. Laitteet lähettävät viestejä resurssienvaihtoa varten muille laitteille. Muuttuneista resursseista voidaan lähettää ilmoituksia niille laitteille, jotka ovat ilmoittaneet kiinnostuksensa kyseisestä resurssista. Laite voi myös julkaista tai vastaanottaa kyselyitä resursseistaan. Osana kehysmallia työryhmä on määritellyt Constrained Application Protocol (CoAP) protokollan resurssien käsittelyä varten.

Selventääkseen CoAPin käytettävyyttä peruskirja [27] kertoo lyhyesti, että CoAP on suunnattu käyttöympäristöihin, jotka on määritelty ROLL- ja 6lo-työryhmissä, mutta CoAPia voidaan käyttää myös perinteisissä IP-verkoissa. 6lo-työryhmä keskittyy IPv6-protokollan integroimiseen noodien rajoittuneissa verkoissa [29]. CoRE-työryhmä [27] ylläpitää neljää ensimmäistä standardisoimaansa spesifikaatiota, joihin lukeutuvat RFC-dokumentit 6690, 7252 ja 7641 sekä luonnos draft-ietf-core-block. Lisäksi CoRE jatkaa ryhmäkommunikaatiotuen kehittämistä RFC-dokumentissa 7390. Luotettavaa multicast-lähetysarkkitehtuuria työryhmä ei ole asettanut tavoitteekseen. CoRE-työryhmä on jatkanut aktiivisesti CoAPin ja REST-ympäristöjen kehitystyötä rajoittuneissa verkoissa, mikä on todistettavissa julkaistujen dokumenttien [28] perusteella. Kehityskohteina ovat olleet resurssien tunnistaminen ja semantiikan määrittely, mutta myös ryhmäkommunikaatiossa ja turvallisuuden parantamisessa on saavutettu edistystä.

Työryhmä on sitoutunut eri järjestöjen ja standardointiorganisaatioiden asetta-

miin vaatimuksiin. Työskentely tapahtuu läheisessä yhteistyössä muiden IETF-työryhmien kanssa, erityisesti rajoittuneisiin verkkoihin keskittyneiden, kuten 6lo- ja ROLL-työryhmien sekä asiaankuuluvien hallinnointi- ja turvallisuusryhmien kanssa.

3 Verkkoprotokollat

Tietoverkot ovat keskenään kommunikoivien laitteiden muodostamia verkkoja, jotka voivat kooltaan vaihdella pienistä kotitalouksien sisäisistä verkoista maailmanlaajuisiin verkkoihin, kuten julkiseen internetiin. Kommunikaation hallinnointia ja järjestämistä varten on tarve määritellä yhteiset säännöt, jotka Kurosen ja Rossin [31, s. 33 35] mukaan asettaa käytetyksi valittu protokolla. Yhteisen protokollan avulla verkon laitteet kykenevät ymmärtämään toisiaan. Protokolla määrittelee viesteissä käytetyn syntaksin ja semantiikan sekä sen, kuinka protokollaa käytetään. Viestinvälityksessä tapahtuva tiedonsiirto ja lähetyksestä sekä vastaanotosta seuraava toiminta ovat oleellisimpia protokollaa määritteleviä tekijöitä. Protokollan käyttäjiä ohjeistetaan mm. kommunikaatioyhteyden muodostamisessa, viestien järjestyksessä, vastaanottamisessa ja lähetyksessä, kuten myös siinä, kuinka viesteihin ja tapahtumiin tulisi reagoida. Semantiikka kertoo, kuinka viestit on tulkittava ja syntaksi määrää käytetyn merkistön ja viestien rakenteen.

Kurose ja Ross [31] kertovat, että erilaisia protokollia on ollut tarpeen suunnitella ja kehittää verkkoympäristöjen vaihtelevuudesta johtuen vastaamaan kommunikoivien osapuolien, viestinnän tarpeellisuuden ja suoritettavan tehtävän asettamiin haasteisiin. Jokaisella TCP/IP-kerrosarkkitehtuurin mukaisella kerroksella on lukuisia niille sijoitettuja protokollia, jotka voivat olla toteutettuna laite- tai ohjelmistotasolla.

3.1 Asiakas-palvelin-arkkitehtuuri

Nykyään tuntemallamme maailmanlaajuiselle internetille 1990-luku oli suurten läpimurrosten vuosikymmen, joista kirjassa [31, s. 92] nostetaan yhdeksi tärkeimmistä World Wide Webin eli WWW:n julkaiseminen. WWW:n ansiosta internetin suosio kasvoi merkittävästi sen saapuessa ihmisten koteihin ja työpaikoille, viitoittaen samalla tietä kohti internetin kaupallistumista. Käyttäjämäärien kasvaessa myös uusien sovellusten tarve lisääntyi ja alkunsa saivat monet edelleen hyvin tunnetut sovellukset, joiden olemassaolosta on tullut monelle meistä itsestäänselvyys. Monet hyödylliset ja viihdyttävät web-sovellukset ovat vahvistaneet ja ylläpitäneet ih-

misten tarvetta julkisista tietoverkoista ja korostaneet palveluiden saatavuuden ja niihin pääsyn tärkeyttä. Riittävä verkkoinfrastruktuuri ja käytännölliset protokollat ovat muodostuneet edellytyksiksi sovellusten käytettävyydelle ja niiden sovittamiseksi eri laitteille ja osaksi jokapäiväistä elämää.

Eräs tunnusomainen piirre verkossa käytettäville sovelluksille on kyky kommunikoida muiden käyttäjien kanssa, minkä toteuttamista voidaan lähestyä kirjassa [31, s. 112 – 119] esitellyn sovellusarkkitehtuurin kautta. Sovellusarkkitehtuuri kuvailee viitekehyksen, joka havainnollistaa, kuinka sovelluksen sisäinen viestintä ja tehtävänjako tapahtuu komponenttien kesken, mutta jättää teknologiaan liittyvät päätökset toteuttajan vastuulle. Kurose ja Ross [31] jatkavat syventymällä monissa internetin web-sovelluksissa noudatettuun asiakas-palvelin-arkkitehtuuriin. Asiakas-palvelin-arkkitehtuurissa yhteydenmuodostuksen aloittajaa kutsutaan asiakkaaksi ja vastakkaista osapuolta palvelimeksi. Aina käynnissä oleva palvelin kuuntelee asiakkaiden lähettämiä pyyntöjä ja vastaa yhteydenottoihin protokollan mukaisesti. Esimerkiksi web-sovelluksissa palvelintietokone toimii nimensä mukaisesti palvelimena ja asiakkaan osassa on käyttäjän päätelaitteessa toimiva ohjelma, joka on hyvin usein tavallinen verkkoselain. Koko sovellusjärjestelmän toiminta rakentuu näiden kahden eri osapuolen toisistaan erilleen asennettuihin sovelluksiin. Päätelaitteissa sijaitsevat ohjelmat kommunikoivat keskenään verkossa välitettävien viestien avulla ja sovellukset voidaan toteuttaa monilla eri ohjelmointikielillä. Sovellusten kehittäminen ja julkaisu nopeutuu ja yksinkertaistuu, koska ainoastaan päätelaitteet sisältävät sovelluskerroksen, missä otetaan kantaa sovelluksen toimintaan. TCP/IP-arkkitehtuurin mukaisella verkkokerroksella ja sitä alemmilla kerroksilla operoivien laitteiden, kuten reitittimien ja kytkinten ohjelmistoon ei tarvitse tehdä muutoksia.

Artikkelissa [8] WWW sijoitetaan TCP/IP-arkkitehtuurin sovelluskerrokselle ja määritellään resurssi, joka on yksi WWW:n keskeisimpiä käsitteitä. Resurssi on palvelimen hallitsema abstraktio ja tunnistettavissa URI-osoitteen kautta. Asiakas-palvelin-arkkitehtuurissa asiakkaat pyytävät resursseja palvelimilta pyyntö-vastausmenetelmällä. Palvelin omistaa resurssin alkuperäisen tilan ja resurssipyyntöjä voidaan hallita välimuistin, välityspalvelinten ja viestien uudelleenohjauksen avulla. Resurssit sisältävät usein linkkejä toisiin resursseihin, mistä muodostuu skaalautuva ja joustava verkko hajautettujen päätepisteiden kesken.

Asiakas-palvelin-arkkitehtuurissa tietoliikenne kulkee palvelimen kautta. Staattisen IP-osoitteen omaava palvelin hoitaa web-sovelluksessa tapahtuvaa tiedonväli-

tystä eivätkä asiakkaat tavallisesti kommunikoi suoraan vertaistensa kanssa. Asiakkaiden lukumäärän kasvaessa palvelin joutuu ylläpitämään useampaa yhteyttä ja käyttämään enemmän prosessointiaikaa ja muistia, mikä johtaa suorituskyvyn heikkenemiseen ja jossain vaiheessa palvelimen resurssit saattavat osoittautua riittämättömäksi. Tämän takia monet suositut web-sovellukset, esimerkiksi suuret sosiaalisen median alustat, on asennettu datakeskuksiin, jotka koostuvat useista palvelinlaitteista luoden tehokkaan virtuaalisen palvelimen asiakkaita varten. Välityspalvelimet ja kuormanjakajat reitittävät ja tasapainottavat yhteyksiä sekä kuormitusta palvelinten kesken toiminnan vakauttamiseksi.

3.2 HTTP

Tietoverkoissa käytettäviä sovelluksia on hyvin paljon. Sovellukset vaihtelevat käytön ja tarkoituksensa mukaan, mikä on luonut tarpeen tapauskohtaisesti valita tai kehittää sopiva protokolla vastaamaan asetettuja tarpeita. Protokollissa voidaan korostaa esimerkiksi tiedonsiirron luotettavuutta tai optimoida tietyn tiedostomuodon välittämistä. Osa protokollista on julkistettu vapaaseen käyttöön toisten säilyttäen yksityinen omistajuus. Kirjassa [31, s. 126 – 127] käsitellään kattavasti HTTP-protokollaa, joka on kenties tunnetuin sovelluskerroksen protokolla ja erityisen suosittu internetissä ja web-sovelluksissa.

HTTP:n, sen eri versioiden ja ominaisuuksien määrittely on jakaantunut lukuisiin RFC-dokumentteihin, mutta kirjassa [31, s. 129 – 142] annetaan myös karkea kuvaus tavallisesta vuorovaikutuksesta. Tyypillisessä käyttötapauksessa kommunikaatio asiakkaan ja palvelimen välillä tapahtuu HTTP-viestien avulla, missä protokolla määrittelee viestien rakenteen ja niiden lähetysjärjestyksen. Resurssia, kuten verkkosivua, pyytäessä asiakas lähettää palvelimelle pyyntöviestin sivun sisältämistä objekteista. Jos pyyntöä käsiteltäessä ei esiinny ongelmia, niin palvelin reagoi lähettämällä asiakkaalle vastausviestin pyynnön onnistumisesta. Muutoin palvelin palauttaa virheviestin. Palvelimella ajettavasta sovelluksesta riippuen vastausviesti voi sisältää koko resurssin tai sitä voi seurata useita vastauksia sisältäen verkkosivulle linkitettyjä muita resursseja, esimerkiksi teksti-, kuva- tai äänitiedostoja. Koska HTTP on ainoastaan protokolla, niin se ei ota lainkaan kantaa siihen, kuinka siirretyt resurssit esitetään asiakkaalla tai palvelimella.

TCP/IP-arkkitehtuurissa sovelluskerrokselle sijoittuva HTTP tarvitsee alempien kerroksien palveluita toimiakseen luotettavasti. Kurose ja Ross [31, s. 129 – 142] sel-

ventävät, että kuljetuskerroksella HTTP tukeutuu luotettavan tiedonsiirron takaavaan TCP-protokollaan ja hyödyntää monia sen tarjoamia palveluita. Aloittaakseen kommunikaation asiakas muodostaa TCP-yhteyden palvelimen kanssa. Kun yhteys on muodostettu kolmitiekättelyprosessin mukaisesti, niin osapuolten sovelluksille luodaan yhteyden käyttöä varten sokettirajapinnat, joiden kautta viestien lähetys ja vastaanotto tapahtuu. Internetin kerrosarkkitehtuuri helpottaa vastuunjakoa kommunikaatiosta. Jokainen kerros vastaa sille asetetuista tehtävistä hyödyntämällä alempien kerrosten palveluita, tarjoten samalla omia palveluitaan ylemmille kerroksille paljastamatta niiden yksityiskohtaista toteutusta. Sovelluskerroksella toimivan HTTP:n ei tarvitse huolehtia viestien katoamisesta tai kuinka siitä palaudutaan. Kun tietoliikennepaketti on siirtynyt soketin läpi kuljetuskerrokselle, TCP ja alempien kerrosten protokollat pitävät huolen luotettavasta tiedonsiirrosta.

Monissa internetin sovelluksissa asiakas ja palvelin kommunikoivat usean pyyntö-vastaus-syklin verran, kuten edellä kuvatussa tapauksessa, jossa yksittäinen verkkosivu koostuu useasta resurssista. Jokaista pyyntöä varten edellinen TCP-yhteys voidaan päättää ja muodostaa uusi yhteys. Toinen tapa on käyttää yhtä ja samaa yhteyttä koko istunnon ajan, minkä Kurose ja Ross [31, s. 129 – 142] esittävät yhtenä menetelmänä tiedonsiirron optimoimiseksi. Ensimmäisessä tapauksessa yhteyksiä kutsutaan ei-pysyviksi ja jälkimmäisessä pysyviksi. HTTP pystyy käyttämään molempia yhteysmalleja ja nykyään oletuksena käytettävä toimintatapa on pysyvät yhteydet. Jos verkkosivu sisältää useita resursseja, esimerkiksi HTML-sivun, kuvia ja tyylitiedostoja, niin ei-pysyviä yhteyksiä käytettäessä jokaista resurssia varten muodostetaan oma TCP-yhteys. Nykyään selaimet käyttävät tavallisesti 5 – 10 rinnakkaista TCP-yhteyttä, joten edellä kuvatussa esimerkissä resurssien vaatimat yhteydet muodostettaisiin rinnakkain sarjassa tapahtuvan yhteydenmuodostusprosessin sijasta. Rinnakkaisten yhteyksien käyttäminen lyhentää vastausaikaa, koska resursseja voidaan siirtää samanaikaisesti ilman odotusta edellisen tiedonsiirron päättymisestä. Vastausaikaa mitataan Round-trip time (RTT) ajalla. RTT on aika, joka kuluu pienen paketin lähettämiseen asiakkaalta palvelimelle ja takaisin asiakkaalle. RTT koostuu paketin etenemisviiveestä, jonottamisajasta linkkien verkkolaitteilla ja laitteilla tapahtuvasta paketin käsittelyajasta. Etenemisviive voidaan jakaa linkkien kaistanleveydestä riippuvaan tiedonsiirron nopeuteen ja signaalien välittämiseen, joka riippuu välittäjämateriaalin fysikaalisista ominaisuuksista.

Ei-pysyvien yhteyksien käyttäminen lisää resurssien tarvetta erityisesti palvelimella, johtuen ylimääräisestä työstä, joka aiheutuu peräkkäisten yhteyksien aloitta-

misesta ja päättämisestä. Kirjan [31, s. 129 – 142] mukaan HTTP:n eri versiot noudattavat oletuksena vaihtelevia yhteyskäytänteitä. Pysyviä yhteyksiä käytettäessä HTTP-versiossa 1.1 palvelin ei sulje TCP-yhteyttä lähetettyään vastauksen, vaan odottaa seuraavia pyyntöjä tapahtuvaksi saman yhteyden välityksellä. HTTP käyttää oletusasetuksillaan pysyviä yhteyksiä putkituksen kanssa. Putkituksessa seuraavien resurssien lähettäminen aloitetaan ennen kuin edeltäviä on kuitattu. HTTP:n toinen versio (RFC 7540) sallii pyyntöjen ja vastausten lomittamisen samassa yhteydessä ja tarjoaa menetelmän niiden priorisoimiseen.

Kurose ja Ross [31, s. 129 – 142] jakavat HTTP-viestit kahteen tyyppiin: pyyntöihin ja vastauksiin. Viestit kirjoitetaan tavallisella ASCII-merkistöllä, mikä tekee niistä myös ihmisten tulkittavia. Pyynnön aloittaa pyyntörivi, seuraavien rivien ollessa otsakerivejä. Pyyntörivillä esitellään käytetty HTTP-metodi, URL-osoite ja HTTP-versio. Metodeja on erilaisia, joista yleisimmät ovat GET, POST, PUT ja DELETE. Otsakeriveillä voidaan välittää erilaista metatietoa tai esittää lisäoptioita siitä, kuinka pyyntö halutaan käsiteltävän. Asiakas voi ilmoittaa esimerkiksi käyttämänsä selaimen ja neuvotella palautettavan sisällön kieliversiosta tai tiedostomuodosta. Pyyntöviesteissä on myös kolmas osio otsakerivien ja pakollisen tyhjän rivin jälkeen. Tämä osio on viestin runko ja se on tarkoitettu puhtaasti tiedon lähettämistä varten. Esimerkiksi verkkolomaketta täytettäessä asiakas lähettää tiedot POST-metodilla pyynnön rungossa. Tietoja voidaan lähettää myös muilla metodeilla, kuten URL-osoitteeseen lisättävillä kyselyparametreilla. Esimerkiksi GET-metodilla tehdyssä Google-haussa "https://www.google.com/search?q=http" kysymysmerkin jäljessä oleva "q" on parametrin nimi ja "http" sen arvo.

Vastausviesti sisältää myös otsaketiedot ja rungon, mutta kirjassa [31, s. 129 – 142] esitetään lisäksi sen eroja verrattaessa pyyntöviesteihin. Ensimmäinen rivi on tilarivi, joka vastaa pyynnön pyyntöriviä. Tilarivillä ilmoitetaan protokollaversio, statuskoodi ja sitä kuvaileva viesti. Otsakeriveillä on sama rooli kuin pyyntöviesteissä: ne kertovat metatietoa palvelimesta ja sen lähettämästä sisällöstä. Runko sisältää palvelimen lähettämän tiedon, joka voi olla esimerkiksi HTML-verkkosivu, jonka käyttäjän selain jäsentelee nähtäväksi. Tilarivin statuskoodi- ja viesti kertovat pyynnön onnistumisesta tai epäonnistumisesta. Statuskoodit ovat laajennettavissa, mutta yleisimmin käytetyt on määritelty RFC-dokumentissa 7231 [17]. Koodin ensimmäinen numero kertoo vastauksen luokan, esimerkiksi väillä 200 – 299 olevat koodit ilmoittavat onnistuneesta pyynnöstä ja välillä 400 – 499 asiakkaan virheestä.

Kurose ja Ross [31, s. 129 – 142] mainitsevat, että palvelinarkkitehtuurien suun-

nittelu yksinkertaisemmiksi ja tehokkaammiksi on helpottunut HTTP:n tilattomuuden ansiosta, mikä sallii jopa tuhansien samanaikaisten TCP-yhteyksien ylläpidon. HTTP on tilaton protokolla, mikä tarkoittaa, että palvelin ei ylläpidä tietoa asiakkaista. Palvelin vastaa asiakkaan pyyntöihin ottamatta lainkaan kantaa siihen, onko asiakas ollut aikaisemmin yhteydessä palvelimeen tai esittänyt vastaavaa pyyntöä menneisyydessä. Joissain tilanteissa asiakkaan tunnistaminen sisällön kohdistamiseksi tai aikaisemman istunnon palauttamiseksi on toivottavaa. Tätä varten monet verkkosivut käyttävät nykyään evästeitä, jotka välitetään viestien otsaketiedoissa. Evästeiden avulla voidaan luoda istuntokerros tilattoman HTTP:n päälle. Vaikka evästeet voivatkin parantaa käyttökokemusta ja helpottaa kiinnostavan tiedon löytämistä, niin ne ovat herättäneet myös paljon huolta. Kohdennettu markkinointi, informaatiokuplat ja käyttäjien seuranta ovat vain joitakin asioita, jotka ovat sytyttäneet keskustelua yksityisyydensuojasta.

3.3 CoAP

Ihmisten käyttämä WWW kuvataan Bormann et al. [8] toimesta karkeasti kolmella eri teknologialla: HTML, HTTP/REST ja URI-osoitteet. HTML vastaa tiedon esittämisestä, HTTP ja REST tiedonsiirrosta ja -käytöstä sekä URI-osoitteet resurssien tunnistamisesta ja yksilöimisestä. Laitteiden välisessä kommunikaatiossa ainoastaan kaksi viimeistä ovat käyttökelpoisia, koska tiedon esityksessä on olemassa paremmin laitteille soveltuvia muotoja. HTTP on tehokas ja paljon käytetty protokolla, mutta se on suhteellisen raskas tarvitsemansa kooditilan ja verkkoresurssien käytön suhteen.

REST-arkkitehtuurin ansioiksi artikkelissa [8] mainitaan resurssien saataviksi, jaettaviksi ja tunnistettaviksi tekeminen niitä yksilöivien URI-osoitteiden avulla. Sovellukset kommunikoivat ja vaihtavat julkaisemiaan resursseja yhdessä sovitun tiedonsiirtoprotokollan välityksellä. Dokumentissa [51] tunnistetaan REST-arkkitehtuurin tehokkuus ja helppo kehitettävyys, mitkä ovat tehneet siihen perustuvista verkkopalveluista, kuten API-rajapinnoista, hyvin yleisiä internetin sovelluksissa. Arkkitehtuurin edut ovat herättäneet kiinnostusta sen sovittamisesta rajoittuneita noodeja ja verkkoja varten. Bormann et al. [8] nostavat esille, että RESTin tuominen esineiden internetiin sallii kehittäjien aikaisemman tietotaidon siirtämisen uuteen laitteistoympäristöön. Vuosien saatossa tapahtunut kasvu ja kehitys ovat lisänneet HTTP:n ominaisuuksia ja niiden vaatimia resursseja. Tämä on kasvattanut pai-

netta käyttökelpoisempien ratkaisujen suunnitteluun rajoittuneille laitteille. IETF:n CoRE-työryhmä on pyrkinyt tuomaan REST-arkkitehtuurin käytettäväksi rajoittuneille laitteille ja verkoille, joille HTTP on liian raskas protokolla. Työryhmän tärkein tuotos on ollut CoAP-protokolla.

3.3.1 Protokolla

Dokumentissa [51] kerrotaan, että yleisluontoisen web-protokollan suunnittelussa rajoittuneisiin ympäristöihin ei haluttu ainoastaan tiivistää HTTP:n esitystä, vaan myös hahmottaa RESTin ja HTTP:n yhtenevät osat optimoituna M2M-sovelluksille. CoAP tukee menetelmää palveluiden ja resurssien etsinnälle sekä soveltaa webissä usein esiintyviä käsitteitä, URI-osoitteita ja internetissä käytettyjä mediatyyppejä. CoAP pyrkii myös vastaamaan rajoittuneiden ympäristöjen erikoisvaatimuksiin koskien multicast-lähetyksiä pienentäen protokollan yläpuolista kuormaa.

Yksi protokollan suunnittelutavoitteista on ollut viestien ylimääräisen kuorman minimoiminen välttäen pakettien pilkkomista, mikä heikentäisi viestien perille pääsyä [51]. Toisin kuin esikuvansa HTTP, CoAP tähtää tavoitteisiinsa huomattavasti yksinkertaisemmin ilman historian tuomaa velkaa ja siitä aiheutuvaa yhteensopivuuden ylläpitämisvastuuta [8].

Yksi silmiinpistävimmistä eroista CoAPin ja HTTP:n välillä on se, että standardissa [51] määritelty CoAP käyttää kuljetuskerroksella UDP:tä TCP:n sijasta. Kuljetusprotokollan yläpuolelle on suunniteltu kaksi loogista kerrosta. Viestikerros huolehtii UDP:stä, kadonneiden pakettien uudelleenlähettämisestä sekä asynkronisesta kommunikaatiosta. Pyyntö-vastaus-kerroksen vastuulla ovat viestityypit ja vastauskoodit. Viestityyppejä on neljä erilaista: Confirmable, Non-confirmable, Acknowledgement ja Reset. Kerrosten vastuunjaosta huolimatta, CoAP on kuitenkin ainoastaan yksi protokolla, missä edellä mainitut tiedot ovat CoAP-otsakkeissa ilmoitettuja ominaisuuksia. Kuljetuskerroksella käytetyn UDP:n takia CoAP tukeutuu viestinnän turvallisuudessa TLS- ja SSL-protokollien sijasta DTLS-protokollaan, mikä sallii avain- ja sertifikaattihallinnan [8].

CoAP-protokollan tavallisia käyttökohteita ovat M2M-sovellukset, mitä painotetaan RFC-dokumentissa 7252 [51]. Esimerkkejä tyypillisistä sovelluskohteista ovat älykäs energianhallinta ja rakennusautomaatio. Päätepisteiden välinen vuorovaikutus tapahtuu pyyntö-vastaus-menettelyllä. CoAP on osoittautunut M2M-sovellusten vaatimukset täyttäväksi protokollaksi rajoittuneisiin ympäristöihin. Asynkroninen viestintä onnistuu luotettavasti UDP-yhteyttä käyttäen, tukien unicast- ja multicast-

lähetyksiä, pientä otsakekokoa ja monimutkaistakin viestien jäsentelyä. Tilaton yhdistäminen HTTP:n ja CoAPin kesken on mahdollista samoin kuin URI-osoitteiden ja eri tietotyyppien käyttö.

Otsaketietoja koskien, artikkelissa [8] lisätään, että tavallisen pyynnön neljän tavun otsaketiedot ja lisäoptiot summautuvat noin 10 – 20 tavun viestikuormaksi tuottaen kompaktin ja helposti jäsenneltävän muodon. Optioissa sallittu vapaus mahdollistaa laajentumisen tulevaisuudessa ilman yksinkertaisten toteutusten kuormittamista. Viestikerroksen yläpuolella CoAP määrittelee neljä HTTP:stä tuttua pyyntötyyppiä: GET, PUT, POST ja DELETE. Myös vastausten statuskoodit on mallinnettu HTTP:stä, mutta tiivistetty yhteen tavuun. Esimerkiksi koodi 201 Created on CoAPissa 2.01.

Artikkelissa [8] todetaan, että CoAP toimii hyvin siirrettäessä pieniä määriä tietoa, kuten yksittäisiä lämpötilalukemia tai kytkinten tilatietoja, mutta silloin tällöin sovelluksilla on tarve suurempien tietomäärien siirtämiseen, esimerkiksi laiteohjelmistopäivityksiä varten. HTTP:tä käytettäessä alemmalla kerroksella toteutettu TCP pitää huolen pakettien pilkkomisesta ja järjestyksestä. Vaikka UDP tukee suurempia tietosisältöjä IP-pilkkomisella, niin se on rajoitettu 64 kB kokoisiin paketteihin ja pilkkominen ei sovellu hyvin rajoittuneisiin sovelluksiin tai verkkoihin. Tämän takia CoAP lisää lohko-optioita, joita käytetään resurssin siirtämiseen useassa tietolohkossa ja pyyntö-vastaus-parissa. Lohko-optiot sallivat palvelimen toimimisen täysin tilattomasti. Palvelin voi käsitellä jokaisen lohkon itsenäisesti ilman yhteydenmuodostusta tai aikaisempien lohkojen muistamista.

Laitteiden muodostamissa verkoissa osapuolien täytyy pystyä löytämään toisensa ja niiden tarjoamat resurssit, mikä on Bormann et al. [8] mukaan yksi CoRE-työryhmän käsittelemistä ongelmista. Työryhmän kehitystyö keskittyy autonomisiin laitteisiin ja sulautettuihin järjestelmiin, missä yhteneväinen ja järjestelmien välinen resurssien etsintä on merkittävämpää kuin nykyisissä web-ympäristöissä. Yhteistoimivuuden varmistamiseksi CoAP esittelee tekniikan resurssien ja niiden kuvausten löytämiseksi ja mainostamiseksi. Kuvausten muoto on ollut tarpeen standardoida, koska ne ovat laitteiden tulkitsemia. Etsinnän saavuttamiseksi CoAP-palvelimia kehotetaan mainostamaan kuvauksia ”/.well-known/core” URI-osoitteen kautta. Asiakkaat pääsevät kuvauksiin käsiksi tästä osoitepolusta ja polkua voidaan myös mainostaa muualle verkkoon.

3.3.2 CoAP verkon protokollana

Iglesias-Urkia et al. [24] väittävät, että CoAP on erottunut esineiden internetille suunnattujen kommunikaatioprotokollien joukossa erityisesti RESTiin perustuvan arkkitehtuurinsa ansiosta, mikä tekee HTTP:n korvaamisen sillä helpommaksi. CoAP on kevyt laite- ja verkkovaatimuksiltaan tehden siitä sopivan yleiskäyttöiseksi protokollaksi vaihtelevissa laiteympäristöissä. Protokolla on sopeutunut hyvin ominaisuuksiltaan ja vaatimuksiltaan poikkeavien resurssirajoittuneiden laitteiden ja verkkojen sovelluksiin.

Villaverde et al. [60] toteavat, että esineiden internetin kehitykselle on havaittu haasteeksi skaalautuvan ja yhteensopivan sovelluskerroksen sekä kehittäjille suunnitellun yhteisen ohjelmointimallin muodostaminen. Ratkaisuna näihin ongelmiin on esitetty REST-arkkitehtuuria, jossa CoAPilla on merkittävä rooli. CoAP pystyy hoitamaan sulautettujen verkkojen yhdistämisen olemassa olevien web-teknologioiden kanssa alustariippumattomasti ja valvomaan sekä ohjaamaan verkkoon liitettyjä resursseja ja hallintalaitteita.

Tutkimuksessa [60] todetaan REST-arkkitehtuurin käyttöönoton vähävirtaisissa verkoissa keränneen jo vuonna 2012 paljon kiinnostusta, mikä tuli silloin ilmi tutkijoiden ja kaupallisten tekijöiden panostuksena CoAPiin. CoAPia käsittelevä standardi oli tuolloin vielä kehitysvaiheessa, joten oli aiheellista nostaa keskusteluun tärkeitä kysymyksiä koskien arkkitehtuuria. Pohdinnassa oli mm. CoAPin rooli välittäjänä SOAP-palveluissa ja protokollolla tapahtuva M2M- ja Human-to-machine vuorovaikutus REST-sovelluksissa. Samalla todettiin spesifikaatiossa esiintyneitä ongelmia ja tarve protokollan lisätestaukselle M2M-järjestelmissä. Jo ennen CoAPia koskevan standardin julkaisua jotkut yritykset olivat alkaneet tarjoamaan yhdistettyjä CoAP-IPv6-palveluja ja tutkijat julkaisseet avoimen lähdekoodin toteutuksia. Samoin protokollan viiveen, energiankulutuksen ja kuormituksen ominaisuuksia oli selvitetty ja merkittävät tahot olivat ilmoittaneet aikomuksistaan CoAPin käyttöönotolle sulautetuissa laitteissaan. Ennen standardin julkaisua CoAPin turvallisuusominaisuuksista tunnistettiin ongelmia, jotka oli tarpeen korjata. Päästä päähän ulottuvan turvallisen kommunikaation saavuttamiseksi CoAPia ja HTTP:tä käyttävä viestiliikenne pitää pystyä kääntämään DTLS- ja TLS-protokollille välityspalvelimella ja DTLS-protokollan tuki nähtiin jo tuolloin tarpeelliseksi laajentaa multicast-lähetyksiin.

CoAP on osoittanut arvonsa kommunikaation tapahtuessa CoAP-protokollaa ymmärtävien päätelaitteiden kesken, mutta Bormann et al. [8] väittävät, että to-

dellinen hyöty saavutetaan, kun protokolla integroidaan toimivaksi HTTP:n kanssa. REST-arkkitehtuurin kuvailemat välityspalvelimet tekevät tämän mahdolliseksi. Välityspalvelin toimii tulkkina HTTP- ja CoAP-protokollien välillä ilman erityistä sovelluslogiikkaa ja riippumatta asiakkaasta tai palvelimesta. Koska CoAP on ottanut mallia paljolti HTTP:stä, niin protokollien välillä tehtävä kartoitus on melko suoraviivaista. Välityspalvelin voi myös naamioida CoAPin URI-osoitteet (coap://) HTTP:n mukaisiksi (http:// tai https://), jolloin asiakkaat voivat löytää resursseja tietämättä lainkaan, minkä protokollan saavutettavissa ne todellisuudessa ovat.

Kuten myös HTTP, CoAP tukee asiakas-palvelin-arkkitehtuuria kommunikaatiossaan [51], mutta myös muiden viestintämallien käyttö on mahdollista. HTTP:ssä viestintä alkaa aina asiakkaan aloitteesta ja pysyäkseen ajan tasalla asiakkaan tulee kysellä uutta tietoa palvelimelta säännöllisesti. Rajoittuneissa ympäristöissä tämä voi käydä resurssien kannalta kalliiksi. Ratkaisuna tähän artikkelissa [8] esitellään havainnointi, joka on valinnainen CoAPin käyttämä asynkroninen menetelmä, jolla asiakkaalle voidaan ilmoittaa palvelimella tapahtuneista muutoksista. Jos havainnointi on palvelimen tarjoama palvelu, niin asiakas voi ilmoittautua resurssin tilaajaksi, jolloin resurssin muutoksista ilmoitetaan asiakkaalle asynkronisesti lähetettävillä viesteillä. Suunnittelumalli on tehokas ja laajentaa RESTiä välttäen tarpeen monimutkaiselle publish-subscribe-arkkitehtuurille.

Villaverde et al. [60] väittävät, että CoAP on osoitettu käyttökelpoiseksi vähävirtaisessa, vuoroa vaihtelevassa 6LoWPAN-verkossa. Energiankulutuksen ja viiveen mittaus on osoittanut käytännössä toteutetun järjestelmän virrankulutukseltaan tehokkaaksi ilman sovelluskerroksella käytettäviä erikoistuneita virranhallintamenetelmiä. Toteutetuissa logistiikkasovelluksissa on todettu CoAPin parempi soveltuvuus verrattaessa muihin REST-arkkitehtuuria noudattaviin protokolliin, koska CoAP onnistui säilyttämään viestinnän luotettavuuden pienemmällä kuormalla kuin muut protokollat. Älykästä sähköverkkoa ja rakennusautomaatiota varten on esitetty joitakin menetelmiä suosittujen sovelluskerrosprotokollien yhdistämiseksi CoAPin kanssa. Vanhempien protokollien resurssit voidaan kuvata CoAPin vastaaviin ja niiden hyötykuorma voidaan kapseloida CoAP-viesteihin. Myös ryhmäviestintä voidaan integroida CoAPin multicast-lähetyksiin.

Vastaavasti kuten Naik [39] toteaa, niin tutkimuksessa [60] huomattiin, että HTTP:hen verrattaessa CoAPin viive, viestien yläpuolinen kuorma ja energiankulutus ovat pienempiä. CoAPin otsaketietoja on tiivistetty ja kuljetuskerroksella käytetään UDP:tä TCP:n sijasta. SOAP-over-CoAP-jäsentelyllä on mahdollista tuoda jo olemassa ole-

via SOAP-palveluja 6LoWPANia käyttävien langattomien sensoriverkkojen saataville. Tavallisia viestejä suurempia tietomääriä käsiteltäessä CoAPin välimuistiominaisuus on osoittautunut hyödylliseksi esimerkiksi ohjelmistopäivitysten lataamisessa noodeille. Päivitysten osia voidaan tallentaa verkon noodien muistiin vähentäen lähetysten tarvetta, kun noodit voivat jakaa asennuspaketteja keskenään.

3.3.3 CoAPin kehittäminen

Tutkimuksessaan [24] Iglesias-Urkia et al. vertailivat CoAPille toteutettuja avoimen lähdekoodin ohjelmointikirjastoja. Tarkasteluun joutuivat kirjastojen ominaisuudet, laajennettavuus, käyttöympäristö, ohjelmointikieli ja yhteensopivuus. Tutkimuksessa otettiin kantaa myös suorituskykyyn mittaamalla viivettä sekä muistin ja prosessorin käyttöä teolliseen ympäristöön suunnitellussa sovelluksessa. Ohjelmointikirjastoja tutkimalla ja vertailemalla pyrittiin helpottamaan sopivimman valintaa. Vaikka kirjastot poikkesivatkin ohjelmointikielen ja käytettyjen kääntäjien suhteen, niin vertailu suoritettiin samassa ympäristössä pyrkien mahdollisimman tasapuoliseen arvioon.

Vertailuun valittiin eri kielillä kirjoitettuja ohjelmointikirjastoja, joista keskitymme erityisesti Pythonilla kirjoitettuihin CoAPthon- ja CoAPy-kirjastoihin. CoAPya lukuun ottamatta kaikki toteutukset tukevat RFC-dokumentin 7252 spesifikaatiota, missä CoAP on määritelty. CoAPy perustuu sitä edeltäneeseen luonnokseen. Luonnosta päivittävissä määritelmissä muutettiin ja poistettiin CoAPyn käyttämä polkuosoite, mistä johtuen se ei ole yhteensopiva muiden kirjastojen kanssa.

Lähdekoodin rakenteessa havaittiin sekä yhtäläisyyksiä että eroavaisuuksia Python-kirjastojen kesken. CoAPthonissa jokaiselle resurssille luodaan oma luokka, joka sisältää sille sallituille operaatioille kirjoitetut metodit. Ohjelmointikirjasto huolehtii vastauskoodeista. CoAPyssa resurssit esitetään vastaavasti luokkarakenteina, mutta vastauskoodeista huolehtiminen jää sovelluskehittäjän vastuulle. Kirjastossa toteutettu vastauskoodien hallinta helpottaa ja nopeuttaa palvelinsovellusten kehitystyötä. Kehittäjien tarvitsee ainoastaan määritellä resurssit, metodit pyyntöjen käsittelylle ja yhdistää niiden toiminta järjestelmissään.

Käytännön testeissä verkkopakettien analysaattori havaitsi suorituskykyä haittaavan ongelman CoAPthon-kirjastossa. CoAP-viesteissä käytetään kahdenlaisia tunnisteita: tunnisteita viestien ja kuittausten yhdistämiseksi sekä mahdollisesti tyhjiä token-koodeja yleisempää käyttöä varten. Toisin kuin RFC-spesifikaatiossa on määritelty, niin CoAPthon-palvelin luo uuden tokenin, jos asiakas ei lähetyksessään sitä

välitä, mikä johtaa viestien hylkäämiseen joissakin asiakasohjelmissa.

Laitteistoläheisellä C-kielellä kirjoitettujen kirjastojen palvelinsovellukset olivat suorituskyvyltään parempia kuin muilla kielillä kirjoitetut, jotka vaativat ylimääräisen käännöskerroksen koodin tulkitsemiseksi. Palvelinsovelluksiin verratessa, muiden kielten asiakasohjelmat olivat tehokkuudeltaan lähempänä C-kielellä toteutettuja. Muiden kielten korkeamman abstraktiotason myötä tutkimus [24] suosittelee kuitenkin niiden käyttöä CoAP-asiakassovelluksissa. Jotkin kirjastoista, CoAPthon ja CoAPy mukaan lukien, tulee asentaa laitteelle ja osassa kaikki riippuvuudet sisältyvät suoritettavaan ohjelmaan. CoAPthon-kirjasto oli pienempi kuin C-kielen vastaavat, mutta suurempi kuin JavaScriptillä kirjoitetut. Asennettava tiedosto pienensi suoritettavien ohjelmien kokoa, mikä huomattiin vertailtaessa esimerkiksi Javalla toteutettuihin kirjastoihin, joissa koko oli merkittävästi suurempi. Tarvittavaan muistitilaan piti kuitenkin lisätä ohjelmointikielen tulkintaan tarvittava ohjelmisto. CPU- ja RAM-käyttöä arvioitaessa huomattiin, että C-kielen kirjastot olivat suoritukseltaan nopeimpia ja kevyimpiä. Java- ja JavaScript-kirjastot olivat RAM-muistin käytöltään noin kymmenkertaisia, kun CoAPthon oli nelinkertainen ja kevyempi CoAPy kolminkertainen. Prosessorin laskenta-ajassa mitattuna CoAPthon oli kaikista kirjastoista selkeästi hitain.

Esineiden ja laitteiden tunnistamista varten on esitetty Electronic Product Code (EPC) pohjaisia URI-osoitteita, mikä Villaverde et al. [60] mukaan laajentaisi CoAPin yhteistoimivuutta jo olemassa olevien tietoverkkojen kanssa. EPC-koodeja on käytetty esimerkiksi Twitter-käyttäjillä sekä iPad-sovelluksilla ja tehnyt suosittuja web-sovelluksia integroitavaksi CoAPin kanssa. Lisäksi CoAPilla toimivia laitteita voidaan hallita Copperin[1] kaltaisilla selainsovelluksilla, joissa CoAPia ja HTTP:tä ymmärtävän välityspalvelimen avulla verkon laitteilla tapahtuvat mittaukset voidaan visualisoida HTTP:tä ymmärtävässä selaimessa.

CoAPin useat konkreettiset toteutukset ovat Rahman et al. [43] mukaan osoitus sen tärkeästä ja entisestään korostuvasta roolista tulevaisuuden IoT-järjestelmissä, mikä on kasvattanut painetta protokollan turvallisuuden parantamiseksi. CoAPin turvallisuudesta on keskusteltu paljon, koska sillä ei ole luotettavia standardeja turvalliselle arkkitehtuurille. Tutkimuksessaan Rahman et al. [43] selvittivät CoAPin turvallisuutta hyödynnettäessä DTLS-protokollaa. He huomasivat, että CoAPin turvaaminen vaatii toisen, salatun protokollan käyttöönottoa, kuten perinteisissä XML-esityksissä ja protokollissa. UDP:tä käytettäessä suojaus saavutetaan tavallisesti DTLS:ää

[1]https://github.com/mkovatsc/Copper4Cr

ja joskus IPSeciä käyttämällä. Löydökset osoittivat huomionarvoisia asioita ja niihin pyrittiin löytämään ratkaisuja sekä valottamaan tulevaisuuden kysymyksiä.

CoAPissa on määritelty neljä turvallisuusluokkaa: NoSec, PresharedKey, RawPublicKey ja Certificates. Tutkimuksen [43] perusteella ne voidaan kuvata lyhyesti seuraavasti. NoSec-luokassa suojausta ei ole lainkaan. PresharedKey-asetuksella toimivat laitteet on esiohjelmoitu käyttämään symmetrisiä avaimia. Menetelmä on sopiva sovelluksille, joissa laitteet eivät pysty turvautumaan julkisen avaimen salaukseen. RawPublicKey on pakollinen luokka laitteille, jotka vaativat julkisen avaimen tunnistautumista. Certificates tukee julkisen avaimen tunnistautumista ja sovelluksia, jotka osallistuvat sertifikaattiketjuihin. Certificates-asetus olettaa, että järjestelmään on yhteensovitettu jo olemassa oleva suojausinfrastruktuuri. Elliptic Curve Cryptography (ECC) on toinen julkisen avaimen salaus, joka tukee Certificates- ja RawPublicKey-luokkia. Luokat vaativat toteutuksen salakirjoituskirjastolle, jota voidaan käyttää neuvotellessa verkkoyhteyden suojausasetuksista. CoAPin suojauksessa laskennan vaatimat korkeat kustannukset ja kuluttava kättelyprosessi viesteissä johtavat niiden paloitteluun. Avainten hallinta on toinen haittapuoli CoAPin suojauksessa, mikä on kuitenkin ongelma lähes kaikissa protokollissa.

Turvallisuutta koskevat haasteet ja keskustelut ovat lisänneet CoAPin tutkimusta. Suurin haaste Rahman et al. [43] mukaan on korkean suorituskyvyn ylläpitäminen säilyttäen samalla tietyt turvallisuusstandardit ja suojauspalvelut. DTLS on virallinen ja mukautettu sovellustason protokolla, jota voidaan käyttää CoAPin turvallisuuden parantamiseksi. DTLS ei ole kuitenkaan ongelmaton: suuri viestikoko, kättelyiden tiivistys ja yhteensopivuus CoAP-välityspalvelinten tilojen kanssa haastavat sen käytön. Päästä päähän tapahtuvassa kommunikaatiossa välityspalvelimen täytyy tulkita paketti tarkistamatta sen sisältöä haitallisen koodin varalta. DTLS ei myöskään tue multicast-viestejä ryhmäkommunikaatiossa. Välityspalvelin voi kääntää TLS:ää käyttävästä HTTP:stä CoAPiin, mutta sen pitää päättää, onko kyseessä multicast- vai unicast-viesti. Vaikka DTLS:llä on omat käyttökohteensa, niin siinä on edelleen heikkouksia, jotka voivat tulevaisuudessa esiintyä turvallisuusuhkina CoAPia käytettäessä.

3.4 Muut IoT-protokollat

Koska eri tapauksiin käyttökelpoisille protokollille on kysyntää, niin ei ole lainkaan yllättävää, että myös esineiden internetiä varten on suunniteltu useita pro-

tokollia. IETF ei ole ainoa organisaatio, joka on perustanut työryhmiä vastaamaan standardointia koskevasta määrittelytyöstä ja johtamaan prosessin etenemistä. Artikkelissaan [1] Al-Fuqaha et al. luetteloivat muita tunnettuja organisaatioita, kuten World Wide Web Consortium (W3C), EPCglobal, Institute of Electrical and Electronics Engineers (IEEE) ja European Telecommunication Standards Institute (ETSI). Työstetyt protokollat voidaan jakaa neljään kategoriaan: sovellusprotokollat, palveluiden etsintään tarkoitetut protokollat, infrastruktuuriprotokollat ja muut protokollat. CoAPin lisäksi monet muutkin esineiden internetin sovellusprotokollat ovat saaneet paljon huomiota ja panostusta kehitystyöhön. Seuraavat MQTT-, XMPP-, AMQP- ja DDS-protokollia käsittelevät alaluvut perustuvat Al-Fuqaha et al. artikkeliin [1], missä he esittelevät tiivistetysti kyseiset hyvin tunnetut sovellusprotokollat.

3.4.1 MQTT

Message Queue Telemetry Transport (MQTT) on vuonna 1999 esitelty, mutta kuitenkin vasta vuonna 2013 standardoitu viestintäprotokolla. Protokolla noudattaa publish-subscribe-arkkitehtuuria ja reititysmenetelmänsä avulla se pyrkii sulautettujen järjestelmien ja verkkojen yhdistämiseen sovellusten ja väliohjelmistojen kanssa. MQTT on osoittautunut soveltuvansa hyvin rajoittuneille laitteille ja epäluotettaviin sekä pienikaistaisiin verkkoihin. Sitä on kuvailtu ihanteelliseksi viestintäprotokollaksi IoT- ja M2M-kommunikaatioon ja useat sovellukset esimerkiksi terveydenhuollosta ja energianseurannasta luottavat siihen. Kuljetuskerroksella MQTT käyttää TCP:tä ja viestit voidaan toimittaa kolmella eri palvelunlaadulla. Julkaistuista spesifikaatioista yksi on suunnattu erityisesti sensoriverkkoja varten. Spesifikaatio lisää välittäjätuen publish-subscribe-arkkitehtuurin mukaisten aiheiden indeksointiin ja määrittelee UDP-kartoituksen MQTT:n tarpeisiin.

Kuten useimmissa publish-subscribe-arkkitehtuurin mukaisissa toteutuksissa, myös MQTT rakentuu kolmesta roolista: tilaaja, julkaisija ja välittäjä. Tilaaja ilmoittaa kiinnostuksena haluamastaan aiheesta ja julkaisijan tuottaessa jotain aiheeseen liittyvää, välittäjä ilmoittaa tästä tilaajalle. Välittäjä huolehtii julkaisijoiden ja tilaajien tunnistamisesta.

3.4.2 XMPP

Extensible Messaging and Presence Protocol (XMPP) on IETF:n välittömän viestinnän standardi, joka kehitettiin avoimen lähdekoodin yhteisön toimesta julkiseksi, turvalliseksi, roskapostivapaaksi ja hajautetuksi protokollaksi. XMPP:ä voidaan käyttää internetin välityksellä käyttöjärjestelmäriippumattomasti ryhmäkeskusteluihin, ääni- ja videopuheluihin sekä etäläsnäoloon. XMPP:n avulla välittömän viestinnän sovelluksiin voidaan tuoda tunnistautuminen, pääsynhallinta, yksityisyydenmittaaminen, linkkikohtainen ja päästä päähän salaus sekä yhteensopivuus muiden protokollien kanssa. Yhdyskäytävien avulla XMPP-verkko voidaan yhdistää toisia protokollia käyttäviin verkkoihin. Monien ominaisuuksiensa ansiosta XMPP:sta on kehittynyt varteenotettava vaihtoehto välittömän viestinnän sovelluksille ja esineiden internetiä varten. Protokolla on turvallinen, laajennettavissa uusilla sovelluksilla ja hajautettavissa. Kommunikaatiossa käytettävä XML-syntaksi tuottaa melko paljon ylimääräistä kuormaa, mutta esitys on tiivistettävissä muilla menetelmillä.

3.4.3 AMQP

Advanced Message Queuing Protocol (AMQP) on esineiden internetiä varten suunniteltu avoimen standardin sovellusprotokolla. Kommunikaation luotettavuudessa viestinlähetyksen onnistumiselta vaadittu taso voidaan valita kolmesta vaihtoehdosta vastaanotettujen viestien lukumäärän suhteen: enintään, vähintään tai täsmälleen kerran. Kuljetuskerroksella AMQP vaatii luotettavan protokollan, kuten TCP:n, palveluita. Viestinnässä vastuun ottaa kaksi pääkomponenttia: vaihto ja viestijono. Vaihdot huolehtivat viestien reitittämisestä jonoille säännöillä ja ehdoilla, jotka määritellään etukäteen. Viestit voidaan tallentaa jonoihin ja myöhemmin lähettää vastaanottajille. Pisteestä-pisteeseen-kommunikaation lisäksi AMQP tukee publish-subscribe-arkkitehtuuria. Viestintäominaisuuksia AMQP hallinnoi kuljetuskerroksensa yläpuolelle suunnitellun viestikerroksen avulla. Viestejä on kahdenlaisia: puhtaita ja merkittyjä. Kuljetuskerroksella kommunikaatio on kehysorientoitunutta ja kerros tarjoaa laajennuksia viestikerrokselle.

3.4.4 DDS

Data Distribution Service (DDS) on Object Management Groupin (OMG) kehittämä publish-subscribe-protokolla reaaliaikaiselle M2M-viestinnälle. DDS saavuttaa loistavan palvelunlaadun ja korkean luotettavuuden luopumalla välittäjistä ja käyttämällä multicast-lähetyksiä, mikä erottaa sen muista publish-subscribe-protokollista. Arkkitehtuuri sopii hyvin IoT- ja M2M-viestinnän tiukoille aikavaatimuksille. Useita kriteereitä, kuten turvallisuutta, kiireellisyyttä, tärkeyttä ja luotettavuutta, voidaan muuttaa 23 palvelunlaatuun vaikuttavalla linjauksella. Arkkitehtuurissa määritellään kaksi kerrosta: Data-Centric Publish-Subscribe (DCPS) ja Data-Local Reconstruction Layer (DLRL). DCPS toimittaa tiedon tilaajille ja DLRL on valinnainen kerros, joka toimii rajapintana DCPS:n toiminnoille ja hallitsee hajautetun tiedon jakamista hajautetuille kohteille.

3.5 Protokollien vertailu

MQTT, CoAP, AMQP ja HTTP ovat jo pitkälti vakiintuneita protokollia ja Naik [39] on vertaillut niitä vahvuuksien ja heikkouksien keräämiseksi, tarkoituksenaan käyttäjävaatimusten ja -tarpeiden mukaan helpottaa sopivimman protokollan valintaa. Naikin vertailevassa tutkimuksessa korostettiin, että tulokset saattavat vaihdella käytettyjen komponenttien mukaan. Komponentit olivat staattisia ja muussa kirjallisuudessa todettuja empiirisiä havaintoja. Myös verkon olosuhteiden muuttuminen järjestelmän elinkaaren aikana ja uudelleenlähetyksistä aiheutuva kuorma voivat vaikuttaa saatuihin tuloksiin.

Esineiden internetissä hyvin suosittujen protokollien, CoAPin ja MQTT:n, välillä Naik [39] havaitsi useita poikkeavia ominaisuuksia. Kaikista vertailuun valituista protokollista CoAP oli julkaisuvuodeltaan uusin. Ominaisuuksiltaan CoAP on rikkaampi kuin MQTT ja sallii asiakas- ja palvelinsovelluksien kehittyä riippumattomasti toisistaan. CoAP tukee sekä MQTT:n käyttämää publish-subscribe-arkkitehtuuria että HTTP:n asiakas-palvelin-arkkitehtuuria. Toisin kuin MQTT, aiheiden sijasta CoAP käyttää kuitenkin URI-osoitteita – samoin kuin HTTP. Esineiden internetin vaatimuksia varten CoAPiin on kehitetty myös useita parannelluja palveluita. Palveluihin kuuluu mm. tuki havainnoitsijoille, multicast-ryhmäkommunikaatiolle, resurssienlöytämiselle ja lohkoittain tapahtuvalle tiedonsiirrolle. Myös välimuistin ja välityspalvelinten käyttö on tuettu CoAPissa. CoAP on osa web-arkki-

tehtuuria ja sopii parhaiten UDP:tä tai UDP:n kaltaista protokollaa tukeville laitteille, mikä kuitenkin rajaa käyttökohteina olevia IoT-laitteita.

Energiankulutusta ja resurssivaatimuksia verrattaessa tutkimus [39] osoitti, että CoAP vaatii vähiten energiaa ja resursseja. Sekä CoAP että MQTT on suunniteltu pienen kaistanleveyden verkoissa toimiville resurssirajoittuneille laitteille. Naik [39] väittää myös useiden tutkimusten tukevan havaintoa, että CoAP on hieman edullisempi käytettyjen resurssien suhteen samoissa olosuhteissa sekä epäluotettavassa että luotettavassa verkossa olettaen, että pakettien katoamista ei ole tapahtunut. On kuitenkin huomioitava, että vertailu ei ota kantaa dynaamisesti muuttuviin verkko-olosuhteisiin tai pakettien uudelleenlähetyksiin ja siitä aiheutuvaan kuormaan.

Muut tutkitut protokollat käyttävät kuljetuskerroksella TCP:tä, mutta CoAP käyttää UDP:tä. Naikin [39] mukaan kuljetuskerroksella käytettävä protokolla on merkittävä syy sille, että CoAP käyttää pienintä kaistanleveyttä ja siinä esiintyy vähiten viivettä. TCP-yhteydessä käytettävä slow start -vaihe ei hyödynnä täyttä kaistanleveyttä yhteyden alussa. CoAPin käyttämä UDP tarvitsee vain kaksi datagrammia vuorovaikutteiseen kommunikaatioon, koska kuittausviestejä ei käytetä. MQTT:n on todettu käyttävän suurempaa kaistanleveyttä kuin CoAP samoissa olosuhteissa ja samalla viestikoolla. Vertailtaessa MQTT:n tason kaksi palvelunlaatua ja CoAPin vahvistettavia viestejä, MQTT käytti noin kaksi kertaa enemmän kaistaa, mikä johtui TCP:ssä tapahtuvasta nelitiekättelystä.

Otsaketietojen kokoa vertailtaessa CoAP sijoittuu tutkimukseen [39] valittujen IoT-protokollien keskikastiin neljällä tavulla, mikä edesauttaa sitä, että viestien koko on usein tarpeeksi pieni sopiakseen yhteen IP-datagrammiin. CoAP osoittautui viestien kooltaan ja niiden vaatimalta yläpuoliselta kuormalta pienimmäksi. Koska MQTT, AMQP ja HTTP käyttävät kuljetuskerroksen protokollana TCP:tä, niin yhteydenmuodostus ja -katkaisu aiheuttavat ylimääräistä kuormaa verrattaessa UDP:n päällä toimivaan CoAPiin. Protokollaero kuljetuskerroksella tuottaa selvän eron kokonaiskuormaan ja viestien kokoon.

Palvelunlaadulle CoAPissa on määritelty kaksi tasoa samoin kuin AMQP:ssä, kun vastaavia on MQTT:ssä kolme kappaletta. Vaikka MQTT tarjoaa parhaimman palvelunlaadun, niin sen yhteensopivuus muiden protokollien kanssa on Naikin [39] vertailun mukaan huonoin. TCP:ssä suurin etu on luotettava tiedonsiirto, mikä tulee UDP:hen perustuvassa CoAPissa jotenkin korvata. CoAP paikkaa UDP:n epäluotettavan viestinnän määrittelemällä uudelleenlähetysmekanismin ja tarjoten palvelun resurssien löytämiselle niiden kuvausten kautta. CoAPissa palvelunlaa-

duntaso valitaan käytettävällä viestityypillä, joita on kaksi erilaista: vahvistettavat (CON) ja vahvistamattomat (NON). Viestityypin valinnan perusteella CoAP mukautuu käyttämään kuittauksia sekä uudelleenlähetyksiä ja ne vastaavat pitkälti MQTT:n kahta alinta palvelunlaaduntasoa. Turvallisuustekijöitä ja lisäpalveluita vertailtaessa AMQP suoriutui parhaiten. MQTT:n varustelu näiden saralla oli puolestaan huonoin. Tunnistautumisen, eheyden ja salauksen täyttämiseksi CoAPissa voidaan käyttää DTLS- ja IPSec-protokollia.

Standardointia ja organisaatioiden tukea verrattaessa tutkimus [39] toteaa HTTP:n poikkeavan muista siinä, että se on yleisesti hyväksytty ja laajasti omaksuttu webstandardi, joka ei kuitenkaan ole suunniteltu esineiden internetiä varten. AMQP on menestynyt hyvin ja otettukin käyttöön joissakin poikkeuksellisen suurissa projekteissa. MQTT on vakiintunut M2M-protokolla, jonka takana seisoo moni teknologiaalan yritys ja yhteisö. Samoin CoAP on saanut tukea teknologiamaailmassa ja onkin kerryttänyt nopeasti suosiota. IETF on korostanut CoAPin standardoinnissa esineiden internetin ja webin yhdistämistä. MQTT ja AMQP ovat OASIS-järjestön ylläpitämiä standardeja, joista MQTT:tä on esitetty yleiseksi protokollaksi IoT-järjestelmille. CoAPin standardisoinnista vastaavat IETF ja Eclipse Foundation ja protokolla on lisensoitu avoimeksi lähdekoodiksi, kuten myös MQTT ja AMQP. Kaikilla IoT-protokollilla on monen suuren kaupallisen toimijan tuki.

Chen et al. [12] suorittivat lääketieteellisellä sovelluksella toteutetun vertailun MQTT-, CoAP- ja DDS-protokollille. He ottivat mukaan myös hyvin kevyen, UDP:n päälle toteutetun mukautetun protokollan vertailukohdan saamiseksi. Sovelluksessa emuloitiin erilaisia verkko-olosuhteita, kuten pientä kaistanleveyttä, korkeaa viivettä ja pakettien häviämistä langattomassa verkossa. Protokollien suorituskykyä mitattiin käytetyn kaistanleveyden sekä koetun viiveen ja pakettien katoamisen avulla. Tutkimuksen tavoitteena oli hankkia määrällistä tietoa havainnollistamaan protokollien suorituskykyä rajoittuneessa langattomassa verkossa. Potilaiden käyttämät lääketieteelliseen käyttöön suunnatut sensorit lähettivät tietoa paikallisen verkon laitteelle, joka toimi yhdyskäytävänä välittämään tiedot keskitetylle palvelimelle.

Ensimmäisenä huomiona tutkimus [12] osoitti, että TCP- ja UDP-pohjaiset protokollat poikkesivat toisistaan kaistanleveyden käytössä. Koska UDP ei ole luotettava protokolla, niin CoAP ja mukautettu UDP eivät lisänneet käyttämäänsä kaistanleveyttä viiveen tai pakettien katoamisen kasvaessa. TCP:n päälle toteutettuina protokollina MQTT ja DDS toimivat puolestaan päinvastoin vastaavissa muutoksis-

sa. Käytetyn kaistan suuruutta arvioitaessa vaihtelevin paketinkatoamisprosentein huomattiin, että CoAP oli selvästi parempi kuin DDS. Pienemmillä katoamisprosenteilla CoAP käytti vähemmän kaistaa kuin MQTT, mutta arvon ylittäessä 25% niiden järjestys vaihtui. CoAPin keveys oli nähtävissä, kun sitä verrattiin mukautettuun UDP-pohjaiseen protokollaan, joka käytti ainoastaan hieman vähemmän kaistaa kuin CoAP. Vaikka DDS käyttikin enemmän kaistaa kuin MQTT, niin sen loistava tehokkuus viivettä ja luotettavuutta arvioitaessa tekevät siitä varteenotettavan vaihtoehdon IoT-sovelluksille.

Muuttaessaan verkon viivettä Chen et al. [12] huomasivat, että CoAPin ja mukautetun UDP:n havaitsemat viiveet pysyivät hyvin lähellä järjestelmän sisäistä viivettä. Viiveen vertailussa protokollien kesken huomattiin, että CoAP ja mukautettu UDP-protokolla pärjäsivät parhaiten, mutta DDS ei ollut niihin verrattaessa paljon huonompi eri mittareilla arvioitaessa. MQTT puolestaan poikkesi selvästi muista ja pärjäsi tulosten perusteella huonoiten.

Kun verkossa emuloitiin pakettien katoamista, niin UDP-pohjaiset protokollat olivat tulosten [12] valossa jälleen hyvin lähellä testiympäristössä määriteltyä kadonneiden pakettien lukumäärää. MQTT ja DDS mahdollistivat puolestaan luotettavan viestinvälityksen, niiden käyttämän TCP-protokollan ansiosta, mikä takasi viestien onnistuneen lähettämisen katoamistenkin ilmetessä.

Mitattaessa protokollien hallintaan käyttämää kuormaa suhteutettuna välitettyyn hyötykuormaan, hallintatieto normalisoitiin vertailun tasapuolistamiseksi. Tulokset [12] osoittivat, että DDS tarvitsi selvästi eniten ohjaustietoa viestinnässään. MQTT selvisi hieman alle neljäsosalla DDS:n vaatimasta, kun CoAP ja mukautettu UDP-protokolla pärjäsivät alle puolella MQTT:n vastaavasta. CoAP käytti ohjaustiedolle tilaa ainoastaan hieman enemmän kuin mukautettu UDP-protokolla, mikä oli jälleen osoitus CoAPin keveydestä käytetyissä resursseissa mitattuna.

Tutkimus [12] osoitti, että TCP-pohjaiset protokollat, DDS ja MQTT, toimivat täysin luotettavasti verkon noin 25% pakettikatoamisissa ja 400 ms viiveellä. CoAP ja mukautettu UDP-protokolla ovat hyviä ehdokkaita sovelluksille, joilla on käytettävissä pieni kaistanleveys, mutta vaativat pientä viivettä. Käyttökohteiden lukumäärää rajoittaa kuitenkin niiden epäluotettavuus takaamaan viestien perille pääsy ja reagoiminen ennustamattomiin pakettien katoamisiin.

Thangavel et al. [53] suunnittelivat MQTT- ja CoAP-protokollia varten yhteisen väliohjelmiston, joka on molempien protokollien käytettävissä jaetun ohjelmointirajapinnan kautta. Väliohjelmistolta vaadittiin seuraavia ominaisuuksia: laajennet-

tavuus nykyisille ja tuleville protokollille, yhteinen rajapinta eri protokollille sekä autonominen sopeutuminen olosuhteiden ja rajoitteiden mukaan. Merkittävä tavoite yhteiselle väliohjelmistolle oli sopivimman publish-subscribe-protokollan valinta vallitsevien verkko-olosuhteiden perusteella.

Artikkelissa [53] keskityttiin yksittäisistä sensorilukemista kootun tiedon lähettämiseen gateway-noodilta palvelimelle tai välittäjälle. CoAPin ja MQTT:n suorituskykyä arvioitiin kokeellisesti eri verkko-olosuhteissa käyttäen edellä kuvattua väliohjelmistoa ja mittaamalla päästä päähän tapahtuvaa viivettä ja kaistanleveyden käyttöä. Siirretyllä tietomäärällä voitiin tulkita protokollan käyttämää kaistanleveyttä ja se suhteutettiin yhdessä viiveen kanssa lähetettyjen viestien lukumäärään. Viive mitattiin erotuksena mittaustiedon vastaanotosta ja sen julkaisun välillä kuluneesta ajasta. Tuloksissa havaittiin, että protokollien suorituskyky oli riippuvainen verkko-olosuhteista.

Pakettien katoamisen ollessa vähäistä mittaustulokset [53] osoittivat, että MQTT koki vähemmän viivettä. Kun katoamisten lukumäärä kasvoi, niin CoAP suoriutui viestinvälityksessä paremmin. Viestien koon ollessa pieni ja katoamisten enintään 25%, CoAP tuotti vähemmän ylimääräistä viestiliikennettä ja kuormaa kuin MQTT luotettavan kommunikaation varmistamiseksi. MQTT:n käyttämä TCP lisäsi enemmän ylimääräistä kuormaa UDP:hen verrattuna. Viestien koon kasvaessa tulokset muuttuivat päinvastaisiksi. Viestin katoamisen todennäköisyys UDP:tä käyttävällä CoAPilla kasvoi suuremmaksi kuin TCP:tä käyttävällä MQTT:llä. Katoaminen johti CoAPilla kokonaisten viestien uudelleenlähetyksiin useammin kuin MQTT:llä.

Testauksessa [53] molemmat protokollat onnistuivat välittämään kaikki viestit riippumatta pakettien katoamisen todennäköisyydestä. Sekä MQTT että CoAP omaavat hyvät uudelleenlähetyskäytänteet pakettien katoamiseen reagoimiseksi alemmilla kerroksilla. Sovelluskerroksella MQTT on yksikerroksinen, kun CoAP koostuu kahdesta kerroksesta: viesti- ja pyyntö-vastaus-kerroksesta. Pakettien katoamisesta seurasi uudelleenlähetyksiä, mikä johti pidempiin viiveisiin viestien vastaanotossa. Koska MQTT:llä on kolme palvelunlaaduntasoa ja CoAPilla vain kaksi, niin mahdollisimman tasapuolisen vertailun saamiseksi MQTT käytti gatewaylla ja tilaajalla palvelunlaadun tasoa yksi ja CoAP vahvistettavia viestejä. Nämä kaksi mallia ovat hyvin samankaltaisia uudelleenlähetysten ja kuittausten suhteen.

4 Palvelukeskeinen arkkitehtuuri

MacKenzie et al. määrittelivät artikkelissaan [36, s. 4 – 11] palvelukeskeisen arkkitehtuurin paradigmaksi, jolla kuvataan hajautettujen, mahdollisesti eri osapuolten omistamien taitojen ja osaamisten järjestämistä ja käyttöä. Kirjassa [44] palvelukeskeistä arkkitehtuuria kehutaan menestyneeksi ja järkeväksi lähestymistavaksi suurten ja hajautettujen seuraavan sukupolven järjestelmien suunnitteluun ja kehitykseen. Raj et al. [44] kuvailivat arkkitehtuurin perusidean monoliittisten järjestelmäkokonaisuuksien osittamiseksi dynaamiseen joukkoon helposti tunnistettavia, hallittavia, käytettäviä ja koottavia palveluita, joista jokaisella on yksi tai useampi rajapinta tarjoamilleen palveluille. Rajapintojen avulla palvelut voidaan tunnistaa ja integroida toimimaan yhdessä toistensa kanssa kooten suurempia palvelukokonaisuuksia ja ratkaisemaan toimintaympäristöissä esiintyviä ongelmia.

Artikkelissa [36, s. 4 – 11] palvelukeskeisen arkkitehtuurin kuvaaman referenssimallin tehtäväksi asetetaan vastuu määritellä korkean tason vaatimukset, jotka jokaisen sitä noudattavan toteutuksen tulisi täyttää. Referenssimalli on abstrakti kuvaus, jonka avulla pyritään ymmärtämään ympäristön ja sen komponenttien oleelliset suhteet. Johdonmukaisia standardeja ja spesifikaatioita noudattaen mallin esittämä konkreettinen arkkitehtuuri voidaan kehittää. Malli rakentuu vähimmäismäärästä yhteneviä tekijöitä, aksiomeja ja suhteita, ja on riippumaton standardeista, teknologioista, toteutuksista tai muista tarkoista yksityiskohdista. Referenssimalli ei ole niinkään ratkaisu minkään tietyn aihepiirin ongelmaan, vaan pyrkimys järjestää ja toimittaa palveluita kokonaisuuden arvon kasvattamiseksi.

MacKenzie et al. [36, s. 4 – 11] mainitsevat yhdeksi palvelukeskeisen arkkitehtuurin tärkeimmistä ominaisuuksista kyvyn hyödyntää muiden tarjoamia palveluita ilman tietämystä niiden toteutuksen yksityiskohdista. Järjestelmän muodostavat komponentit abstrahoivat tarjoamiensa palveluiden toteutuksen ja paljastavat ne muiden käytettäväksi rajapinnan kautta. Vastaavasti sovelluksiin ja eri tietolähteisiin on luotu rajapinnat yhteistoimintaa varten ja palvelukeskeisestä mallista on kasvanut Raj et al. [44] mukaan uusi normaali yritysten ja pilvipalveluiden tiedonhallintaympäristöihin. Toiminnallisten komponenttien kuvailu itsenäisinä ja itsessään riittävinä yksikköinä on parantanut niiden joustavuutta ja uudelleenkäyttöä moni-

mutkaisissakin järjestelmissä.

Artikkelissa [36, s. 4 – 11] kuvaillut näkyvyys, vuorovaikutus ja toiminta ovat palvelukeskeisen paradigman keskeisiä käsitteitä ja tarjoavat tehokkaan kehyksen tarpeiden ja palveluiden yhdistämiseksi. Näkyvyys sallii komponenttien löytää toisensa yhteisesti ymmärrettävän syntaksin ja semantiikan avulla. Vuorovaikutus tapahtuu tavallisesti viestinvaihdon välityksellä pyytämällä tai aloittamalla palveluita muilla komponenteilla. Palveluiden käytön tarkoituksena on saavuttaa jonkinlainen toiminta, joka voi olla tiedonsiirto komponenttien kesken tai muutos vuorovaikutukseen osallistuvien komponenttien tiloissa.

MacKenzie et al. [36, s. 4 – 11] lisäävät, että myös palvelut ovat keskeinen osa palvelukeskeistä arkkitehtuuria ja niihin yhdistyy monta eri ajatusta. Palvelu voi olla todellista toiminnan suorittamista toiselle osapuolelle. Se voi olla myös komponentin kyky tehdä, tarjoutua tai kuvata toiselle tehtävä työ. Palvelukeskeisessä arkkitehtuurissa palvelut ovat menetelmä komponenttien tarpeiden ja kykyjen kohtaamiselle. Yleisesti palvelun käyttöön osallistuvia osapuolia voidaan kutsua tarjoajiksi ja kuluttajiksi. Palvelukeskeistä lähestymistapaa noudattavat järjestelmät korostavat uudelleenkäyttöä, kasvua ja yhteistoimivuutta. Käsitteet näkyvyys, vuorovaikutus ja toiminta koskevat myös palveluita. Palvelun kuvaus luo näkyvyyden ja tekee sen saatavaksi kuvatun syötteen, tuloksen ja käytetyn semantiikan kautta. Kuvaus kertoo myös, mihin palvelua käytetään ja mitä esivaatimuksia sillä on.

Artikkelissa [36, s. 4 – 11] esitetään, että keskeisimmät motivaatiotekijät palvelukeskeiselle arkkitehtuurille ovat olleet suurten tietoliikennejärjestelmien kasvun parempi hallinta, internet-laajuisiin järjestelmiin varautuminen ja organisaatioiden välisen yhteistyön kustannusten pienentäminen. Artikkelin kirjoittajien mukaan palvelukeskeisen arkkitehtuurin todellinen arvo näyttäytyy yksinkertaisena ja skaalautuvana paradigmana laajojen verkkojärjestelmien hallinnassa, jossa suurin hyöty saavutetaan yhteistyön kautta. Arkkitehtuuri on skaalautuva, koska se olettaa mahdollisimman vähän käytetystä verkosta ja minimoi kaikki luottamusolettamat. Paradigma pystyy sopeutumaan myös erilaisiin laiteympäristöihin ja järjestelmiin. Kehitys on ketterämpää ja vuorovaikutteisempaa kuin suunnittelemalla ja toteuttamalla lukuisia parittaisia rajapintoja komponenttien välille. Tuloksena organisaatioiden laajentuminen rakentuu vakaammalle perustalle ja muutoksiin sopeutuminen on joustavampaa.

4.1 Web-palvelut

Vasseur ja Dunkels [59, s. 91 – 98] määrittelevät web-palvelut kehysmalliksi hajautettujen järjestelmäarkkitehtuurien suunnittelemiseksi ja monet web-palveluiden kuvaamista arkkitehtuureista ovat saavuttaneet merkittävää suosiota tietotekniikan sovelluksissa. Vaikka harva web-palvelu itsessään tai sen tarvitsemat teknologiat liittyvät suoraan WWW:hen, niin niiden käyttö on ollut poikkeuksellisen suosittua monissa WWW-sovelluksissa, kuten vuorovaikutteisissa selainsovelluksissa. Kirja [59, s. 91 – 98] kuvailee web-palvelut palvelinten väliseksi kommunikaatioksi, joka on saanut alkunsa käyttäjän aloittamasta toiminnasta. Esimerkiksi autovuokraamon liiketoimintapalvelin voi keskustella toisen osapuolen omistaman palvelimen kanssa web-palveluita käyttäen. Jotta web-palvelulla tapahtuva tiedonsiirto onnistuu ilman ylimääräisiä käännöksiä, molempien palvelimien tulee noudattaa käytetyn palvelun määrittelemää kehysmallia. Samoin eri valmistajien tuottamat älykkäät mittauslaitteet voivat lähettää säännöllisesti mittaustuloksiaan samalle palvelimelle, kunhan kaikki tukevat käytettyä web-palvelua kommunikaatiomenetelmänä ja sen määrittelemää spesifikaatiota.

Koska keskustelevat web-palvelut saattavat poiketa toisistaan esimerkiksi arkkitehtuurin, laitteiston, sovellusten tai jopa omistajan suhteen, niin Vasseur ja Dunkels [59, s. 91 – 98] esittävät edellytykseksi hajautetun arkkitehtuurin toiminnalle yhteiset järjestelmäriippumattomat tiedon esitysmuodot. Itse web-palvelun määritelmän toteuttaminen on harvoin riippuvainen minkään yksittäisen tiedon esitysmuodon tukemisesta. Hyvin yleisiä rakenteellisen tiedon esitysmuotoja web-palveluissa ovat XML-merkintäkieli ja sitä kevyempi JSON-tietomuoto.

Web-palveluiden toteuttamiseksi on suunniteltu vaihtelevia arkkitehtuurimalleja, jotka eroavat tehovaatimuksiltaan ja tiedonsiirrossa käyttämältään kaistanleveydeltä. Joitain tunnettuja malleja ovat Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL) ja Universal Discovery Description and Integration (UDDI). Vasseurin ja Dunkelsin [59, s. 91 – 98] mukaan älykkäiden järjestelmien resurssirajoitteet tekevät näistä kuitenkin epäkäytännöllisiä niiden tarvitseman resurssikapasiteetin takia. Paremmaksi vaihtoehdoksi he esittävät Representational State Transfer (REST) arkkitehtuurin, joka on osoittautunut hyvin sopivaksi ja kevyeksi menetelmäksi web-palveluiden toteuttamiseksi IoT-järjestelmiin.

4.2 REST-arkkitehtuuri

Rajapintojen hyödyntäminen on yleinen tapa paikallisten ja verkon kautta saavutettavien palveluiden yhdistämiseksi. Useat sovellus- ja palvelutyypit, kuten sulautetut järjestelmät, mobiili-, web- ja pilvisovellukset tai analyyttiset ja liiketoimintajärjestelmät, on onnistuttu Raj et al. [44, s. 106] mukaan yhdistämään saumattomasti toisiinsa. Erityisesti HTTP:n yli tapahtuva tiedonsiirto REST-arkkitehtuurissa on noussut esille IoT-järjestelmien suunnittelussa. REST on arkkitehtuurimalli resurssien yhtenäistä käyttöä ja muokkausta varten. Palvelimella on omistajuus resurssin sen hetkisestä tilasta ja muut verkkoon osallistuvat voivat pyytää resurssin kuvausta sekä luoda, muokata tai poistaa resursseja. Resurssit tunnistetaan URI-osoitteestaan. Resurssiin kohdistuva toiminta kuvataan HTTP-verbeillä ja hyvin usein kuvaus esitetään JSON-tietomuodossa.

RESTin mahdollistama resurssikeskeinen arkkitehtuuri suuren mittakaavan hajautetuille internetsovelluksille on saavuttanut paljon suosiota. Kirjassa [44, s. 106] resurssien merkitystä RESTissä kuvaillaan seuraavasti. Resurssit ovat keskeinen osa RESTiä ja niiden käyttö tapahtuu internetstandardien mukaisesti, noudattaen merkistöjä, osoitteistusta ja asetettuja käyttöoikeuksia. Resurssit ovat tunnistettavissa yksilöivästi yleisesti sovitun syntaksin kautta URL-osoitteilla, ne jakavat yhtenäisen rajapinnan viestimiseen ja ovat esitettävissä useassa eri muodossa. Järjestelmät ovat heikosti sidoksissa toisiinsa tehden niistä joustavia ja helpommin skaalautuvia. RESTin mukaisten palveluiden toteutus on yksinkertaisempaa ja kevytrakenteisempaa muihin vastaaviin arkkitehtuureihin verrattuna.

Vasseurin ja Dunkelsin [59] mielestä REST ei ole ainoastaan esitys web-palveluille, vaan myös arkkitehtuurimalli hajautettujen sovellusten kehittämiseen. REST-arkkitehtuurista on tunnistettavissa kolme keskeistä periaatetta: esitys (representation), tila (state) ja tiedonsiirto (transfer). URI-osoittein tunnistettavat resurssit esitetään halutussa muodossa ja niitä siirretään asiakkaiden ja palvelimien välillä. Kaikki pyynnön täyttämiseen tarvittava tilatieto tulee olla liitettynä pyynnön yhteyteen. Asiakas tai palvelin eivät voi tukeutua toisen osapuolen säilyttämiin tilatietoihin keskustelukumppanistaan, koska molemmat ovat lähtökohtaisesti tilattomia. Tilattomuus koskee ainoastaan muodostettua yhteyttä, ei sovellusten tallentamaa tietoa. Web-sovelluksissa REST on tehokas toteuttaa yhdistämällä HTTP-, TCP- ja IP-protokollien käyttö, jolloin tiedonsiirto on jaettavissa kolmeen vaiheeseen: TCP-yhteyden avaamiseen, RESTin mukaiseen tiedonsiirtoon ja TCP-yhteyden sulkemiseen.

4.3 Web-palvelut esineiden internetissä

Palvelukeskeisellä arkkitehtuurilla saavutetut hyvät tulokset ovat kirjassa [44] esitetyn perusteella olleet motivoivia tekijöitä mallin tuomiseksi myös esineiden internetiin. Esineitä erottavien teknologisten rajojen rikkomiseksi ja kommunikaation sallimiseksi niiden tarjoamat palvelut abstrahoidaan rajapintojen taakse kehittäen palvelukeskeinen laitearkkitehtuuri. Esineet voivat kommunikoida keskenään ja tehdä yhteistyötä osoittaen älykästä toimintaa, jolloin palveluiden jakaminen muiden kanssa mahdollistuu. Kun laitteet yhdistetään pilvessä tai muussa verkossa julkaistujen sovellusten ja tietolähteiden kanssa, pystytään ne aktivoimaan ja ottamaan hallintaan etänä internetin kautta. Kaksi yleisintä palvelumallia ovat REST-arkkitehtuuri, jossa resursseja muokataan tilattomin operaatioin, ja satunnaiset palvelut, jotka on toteutettu hajanaisin operaatioin ja SOAP-viestein. Palvelut mahdollistavat hyvin määritellyn ja yhteistoiminnallisen tavan laitteiden väliselle kommunikaatiolle esimerkiksi M2M-sovelluksissa. Esineiden internetin ja palvelukeskeisen arkkitehtuurin tuominen teollisuuden käyttämiin sovelluksiin ei ole ollut artikkelin [65] mukaan kuitenkaan täysin ongelmatonta. Teollisten ympäristöjen yhteisten piirteiden tunnistaminen ja sen perusteella sopivien laitteiden suunnittelu on haastanut mm. standardisoinnin ja turvallisuuden.

Valtaosa palvelukeskeisistä arkkitehtuureista on web-palveluita ja niiden käyttöönotto langattomissa sensoriverkoissa on harvoin täysin yksiselitteistä, koska ne on suunniteltu eri laitealustoille ja käyttöjärjestelmille [3]. Raj et al. [44] mielestä esineiden internetiä ajatellen RESTillä on useita etuja muihin palvelumalleihin, kuten SOAPiin, verrattuna. Yläpuolinen kuorma on pienempi, tiedonparsinta yksinkertaisempaa, viestintä on tilatonta ja sillä on vahvempi yhteys HTTP:hen. Lisäksi REST-arkkitehtuuria tukevat sovellukset ovat suorituskyvyltään parempia langattomissa resurssirajoittuneissa sensoriverkoissa.

Älykkäiden järjestelmien yhteensovittaminen jo olemassa olevien tietojärjestelmien ja yritysten liikejärjestelmien kanssa onnistuu Vasseurin ja Dunkelsin [59, s. 91 – 98] mukaan tehokkaasti web-palveluiden avulla. Eräänä hyötynä web-palveluiden käytöstä älykkäille laitteille he [59, s. 108 – 109] mainitsevat jo olemassa olevien palvelukeskeisten järjestelmien, ohjelmointikirjastojen ja tietämyksen uudelleenkäytön IoT-sovelluksissa. Näin saavutettuihin hyötyihin lukeutuu mm. integrointi jo olemassa oleviin järjestelmiin ja samojen rajapintojen käyttö. Kokonaisarkkitehtuurin monimutkaisuuden lisääntymistä saadaan kavennettua välttämällä uusien väliohjelmistojen tarvetta. Yhteensovittaminen takaa esineiden internetin todellisen yh-

distämisen muihin tietojärjestelmiin, kuten yritysten resurssienhallintajärjestelmiin.

Kirjassa [59, s. 91 – 98] huomautetaan, että web-palveluita pidetään usein raskaina ja niiden tehokkuus suurissa palvelinarkkitehtuureissa on kyseenalaistettu useasti, mikä herättää tarpeen menetelmän kriittiselle arvioinnille esineiden internetiä varten. Älykkäiden laitteiden resurssirajoittuneisuus energian, kaistanleveyden ja muistin suhteen asettaa tarpeen yksinkertaisille sovelluksille. Liian suuret resurssivaatimukset tekisivät web-palveluista epäsoveltuvia käytettäväksi rajoittuneissa verkoissa. Täten web-palveluilla tulisi olla pieni ohjelmallinen jalanjälki ja siirrettävän tietomuodon mahdollisimman hyvin tiivistetty. Havainnot ovat kuitenkin osoittaneet, että web-palvelut ovat sovitettavissa rajallisen muistin asettamiin rajoihin ja tutkimukset tukeneet väitettä niiden kevytrakenteisuudesta esineiden internetiä varten. On myös onnistuttu saavuttamaan tuloksia, jotka vahvistavat käsitystä, että web-palvelut sopeutuvat muihinkin rajoittuneiden laitteiden resurssirajoitteisiin ja tehokkuus saadaan ylläpidettyä tarpeeksi korkeana.

Web-palvelut esineiden internetissä voidaan suunnitella REST-mallin mukaisesti, jonka periaatteet ovat kirjan [59, s. 108 – 109] mukaan toteutettavissa kustannustehokkaasti ja yksinkertaisesti HTTP-yhteyden päällä älykkäittä laitteita varten. Palvelut ovat sovitettavissa myös vähävirtaisiin radioverkkoihin hyväksi todetuin tuloksin. Saavutetut yhteistoiminta- ja integraatioedut yhdistettynä pieniin resurssivaatimuksiin ja hyvään suorituskykyyn tekevät web-palveluista houkuttelevan vaihtoehdon älykkäiden laitteiden järjestelmiin.

Kyusakov et al. [32] totesivat langattomille sensoriverkoille suunnatun palvelukeskeisen arkkitehtuurin keskittyneen väliohjelmistojen toteuttamiseen gateway-laitteilla. Tutkimuksessa [32] arkkitehtuurin toteutukseen otettiin toinen näkökulma ja selvitettiin SOAP-palvelun käyttämistä noodeilla gateway-laitteen sijasta. Ratkaisun todettiin parantavan integraatiota edeltävien järjestelmien kanssa ja tukevan verkon heterogeenisyyttä alimmilta kerroksilta lähtien. Lähestymistapa lisäsi kuitenkin noodien kuormaa ja resurssien kulutuksen vähentäminen sekä viiveen vertailu tulivat aiheelliseksi selvittää. Sensorinoodien yhdistämiseksi suunniteltiin esimerkkisovellus, jossa pyrittiin parantamaan tehokkuutta, jotta SOAP-palveluiden toteuttaminen rajoittuneilla laitteilla olisi mahdollista. Ratkaisu sisälsi sensoreilla toimivan täyden SOAP-pohjaisen web-palvelun, mikä poisti tarpeen ylimääräisille väliohjelmistoille. Malli yhdisti kevyen version TCP/IP-arkkitehtuurista ja web-palveluille suunnitellun gSOAP-työkalun.

SOAP-palvelun suurin heikkous tutkimuksen [32] mukaan on XML-dokument-

tien jäsentely, kun muissa tietomuodoissa sama voitaisiin esittää tiivistetymmin. Heikkous ei kuitenkaan tarkoita, että XML olisi käyttökelvoton sensorinoodeille. Tulokset osoittivat, että viestien jäsentely muodosti suhteellisen vähän viivettä verrattaessa vertailukohteena olleeseen TCP-malliin. SOAP-palvelun toteutus noodeilla lisäsi kokonaisviivettä merkittävästi, mutta se koostui pääosin lähetysviiveestä. Tutkimus [32] paljasti, että standardoidun palvelukeskeisen arkkitehtuurin toteuttaminen rajoittuneilla laitteilla on mahdollista, mutta tehokaskin toteutus kasvattaa ylimääräistä kuormaa heikentäen suorituskykyä.

Xu et al. [65] mukaan palvelukeskeisessä laitearkkitehtuurissa erityisesti skaalautuvuuden hallinta on aiheuttanut ongelmia. Laitteiden, verkkojen ja kommunikaation monipuolisuuteen ei ole pystytty vastaaman yleisesti hyväksytyllä tavalla, joka piilottaisi ne helpommin käytettävän nimeämiskerroksen taakse. Yhteisen kuvailukielen puute vaikeuttaa palveluiden kehittämistä ja yhteensovittamista verkon fyysisten kappaleiden rinnalle. Uusien esineiden internetiin perustuvien järjestelmien integroiminen muiden tai vanhojen tietojärjestelmien kanssa ei ole suoraviivaista ja on vaatinut erilaisten väliohjelmistojen kehittämistä. Esineiden internetin yhdistäminen pilvilaskennan kanssa on tarjonnut suositun kehitys- ja tutkimuskohteen, mitä Xu et al. [65] ehdottavat yhtenä ratkaisumallina. Pilviteknologian avulla laitteet voivat muodostaa verkkoja ja internetiin yhdistettyihin laitteisiin pääsy helpottuu.

Shancang et al. [34] väittävät RESTin parantavan heikosti sidoksissa olevien palveluiden ja hajautettujen sovellusten yhteensovittamista. Palvelukerros on tavallisesti esittänyt yhteisesti käytettävän rajapinnan sovelluksille, mutta tutkimuksissa on osoitettu myös service provisioning process -menetelmän toimivan tehokkaasti vuorovaikutuksen välittäjänä. Menetelmä alkaa tyyppikyselyllä, missä pyydetään palveluita geneerisessä WSDL-muodossa. Kyselyyn vastataan hakemalla sovellusvaatimuksiin parhaiten sopiva palvelu perustuen sovelluksen käyttö- ja ongelmaympäristöön sekä palvelunlaatua koskeviin vaatimuksiin. Suorituskykyä koskevien haasteiden lisäksi palvelukeskeisen arkkitehtuurin automaattinen koostaminen sovellusvaatimusten perusteella ei ole vielä täysin ongelmatonta. Monimutkaisten ja heterogeenisten verkkojen hallinnalle ja kappaleiden tunnistamiselle ei ole osoitettu täydellistä mallia, mikä edelleen korostaa tarvetta laitteiden hallintaan, yhdistämiseen ja vuorovaikutukseen käytettävän yhteisen rajapinnan kehittämiseksi. Yhteisen kuvailukielen puute tekee palveluiden toteuttamisen yksiselitteisesti eri ympäristöihin mahdottomaksi. Rajapinnan lisäksi tarpeet huomioiva palvelukes-

keinen arkkitehtuuri tarvitsee myös tehokkaan mallin palveluiden löytämiseksi ja etsimiseksi esineiden internetin kehittymisen helpottamiseksi.

Väliohjelmistojen käyttöä langattomien sensoriverkkojen palvelukeskeisessä arkkitehtuurissa selvitettiin tutkimuksessa [3], johon valittujen referenssimallien haasteita ja ominaisuuksia, kuten kommunikaatiota, turvallisuutta ja tiedonkäsittelyä, pohdittiin. Referenssimallit pystyttiin luokittelemaan kolmeen kategoriaan: Ensimmäisen kategorian mallit käyttivät vaihtelevia väliohjelmistoarkkitehtuureja. Toisessa luokassa palvelukeskeinen arkkitehtuuri toteutettiin ilman väliohjelmistoja ja kolmannessa olivat loput vähemmän tutkitut hallinta- ja palvelukokonaisuuksien arkkitehtuurimallit. Artikkelin [3] tavoitteena on ollut väliohjelmistoja käyttävien mallien suunnittelua, toteutusta ja validointia koskevien tietojen yhteen keräämi-nen langattomien sensoriverkkojen sovelluksia ja ympäristöjä varten. Jotta valitut referenssimallit vastaisivat langattomien sensoriverkkojen vaatimuksiin, niin niille asetettiin joitain hakua suodattavia tekijöitä. Malleille asetettuja kriteerejä olivat mm. palveluiden löytäminen ja niiden paranneltu käyttö sekä verkkopalveluiden ja tietoresurssien jakaminen.

Alshinina et al. [3] tulivat johtopäätökseen, että turvallisuudelle ja tiedon tehokkaalle käsittelylle tarjottu tuki oli useimmissa malleista riittämätön ja ainoastaan yksi malleista mahdollisti usean palvelun yhdistämisen. Monimutkaistenkin palvelukokonaisuuksien rakentaminen todettiin kuitenkin mahdolliseksi hyvällä verkkosuunnittelulla. Vaikka tutkimukseen [3] valittuja väliohjelmistoja käyttäviä arkkitehtuureja resurssi- ja optimointiongelmiin vastaamiseksi oli monia, niin yksikään niistä ei kuitenkaan täyttänyt kaikkia langattomien sensoriverkkojen tarpeita. Tiedonkäsittelyn ja palveluiden yhdistämisellä yritettiin pienentää energiankulutusta ja verkolle aiheutuvaa kuormaa, mutta laitteiston ja kommunikaation heterogeenisyys haastoivat verkkojen yhteistyön ja osa ratkaisumalleista oli täysin riippuvaisia väliohjelmistoista.

Artikkelissa [23] tutkittiin ympäristötietoisia palvelukeskeisiä arkkitehtuureja esineiden internetin palveluiden julkaisemiseksi, löytämiseksi ja jakamiseksi. Jotta laite voitaisiin löytää ja tehdä muiden käytettäväksi, niin sen omistajan tulee tehdä laitetta koskevat tiedot julkisiksi. Käyttäjän vaatimuksia ja tarpeita parhaiten vastaavan laitteen ja sen palvelun löytämiseksi arvioinnissa pitää ottaa huomioon konteksti. Ympäristötietoisuus voi ratkaista sopivien parien löytämisen haasteet skaalautuvuudelle. Tuloksena saadaan ympäristötietoinen palvelukeskeinen arkkitehtuuri, joka yhdistää käyttäjät ja laitteet kontekstin ja vaatimusten perusteella.

Konteksti on Ibrahim et al. määritelmän [23] mukaan tietoa, jolla voidaan karakterisoida esineen tai kappaleen tila. Kappale voi olla henkilö, paikka tai objekti ja sen merkitys on korostunut esineiden internetin kasvaessa. Kontekstia määriteltäessä voidaan esittää kysymyksiä: kuka on käyttäjä, miksi ja missä hän käyttää sitä sekä mitä hän käyttää. Kontekstiin liitetään usein myös tarkempia määritelmiä, kuten paikka, aika, lämpötila, käyttäjän tunteet tai mieltymykset.

Artikkelissaan [23] Ibrahim et al. esittelivät arkkitehtuurin, joka täyttää seitsemän kriteeriä, jotka heidän mielestään ovat tarpeellisia. Käyttäjän ja laitteen konteksteja tulee tulkita yhdessä eikä erillään toisistaan. Arkkitehtuurin pitää mahdollistaa kontekstin muuttuminen ajan kuluessa. Laitteiden lukumäärän kasvaessa skaalautuvuuteen tulee pystyä vastaamaan. Palvelut tulee olla löydettävissä ja sen jälkeen käyttäjien etsittävissä tarpeidensa mukaisesti. Etsinnän tulokset pitää pystyä järjestämään yhteisen kontekstin perusteella parantaen haun tehokkuutta. Viimeiseksi arkkitehtuurin tulee olla formaalisti määriteltävissä sallien toimintojen validointi ja oikeellisuuden vahvistaminen.

5 GraphQL

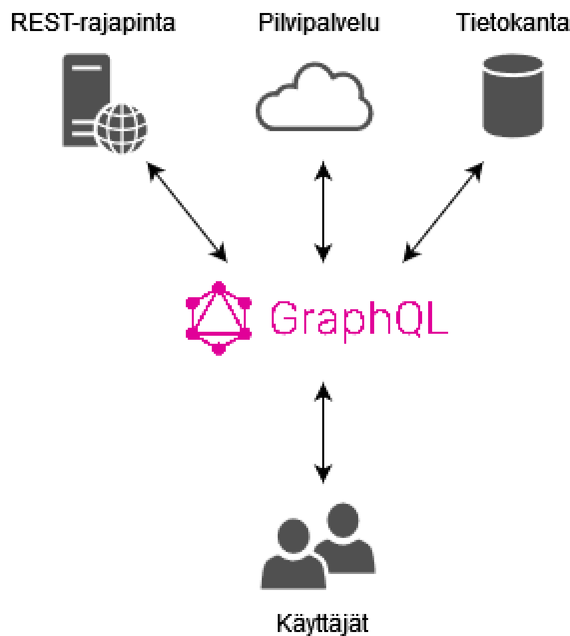
Internetin välityksellä saavutettavien palveluiden käytöstä on tullut arkipäivää nykymaailmassa ja staattiset verkkosivut ovat yhä useammin vaihtuneet dynaamisesti sisältöään muuttaviin web-sovelluksiin. Sovelluksille asetetut odotukset ovat kohonneet ja niiltä toivotaan tasalaatuista suorituskykyä vaihtelevissa ympäristöissä, liikkuessa ja eri laitteita käytettäessä. Stemmler [52] kuvaa ongelmaksi sen, että kaikkien asiakasvaatimusten täyttämiseksi ei riitä enää pelkkä tietokantaan kohdistuva hakuoperaatio ja tulosten esittäminen verkkosivulla, vaan käyttöliittymistä halutaan rikkaita ja rakenteeltaan monimutkaisia luoden uusia käyttökokemuksia.

Vain ja ainoastaan tarvittavan tiedon siirtäminen, säilyttäen samalla ohjelmistoarkkitehtuurin modulaarisuus on haastava ongelma, jonka suhteen joudutaan yleensä tyytymään kompromissiratkaisuihin. Nykyaikaiset sovellukset tarvitsevat usein poikkeavia käyttöliittymiä eri alustoille, kuten tietokoneiden ja matkapuhelinten verkkoselaimille ja niiden sovelluksille tai palvelimella saattaa olla tarve koostaa data useasta eri lähteestä. Tämä kaikki on Porcellon ja Banksin [40, s. 11] mukaan johtanut tiedonsiirrolta vaadittavien resurssien kasvuun ja halutun informaation esittämiseksi joudutaan sitä suodattamaan ja muokkaamaan erikseen sovellusten ohjelmakoodissa.

Maldonado [37] mainitsee API-rajapinnoissa siirtymisen SOAP-arkkitehtuurista REST-arkkitehtuuriin lisänneen kehitystyön joustavuutta, mutta Stemmler [52] huomauttaa kehittämisen ja ylläpidon vaikeutuneen monimutkaisuuden yhä lisääntyessä. Joitain esille nousseita ongelmia on listattu verkkosivulla [41]. Kasvanut mobiililaitteiden käyttö on lisännyt tarvetta tiedonsiirron optimoimiselle virran säästämiseksi ja epäluotettavampien verkkojen kuormittamisen vähentämiseksi. Laitealustojen ja käyttöliittymien monipuolisuus on tehnyt yhteisen API-rajapinnan ylläpidon ja kehittämisen vaikeaksi vastaamaan kaikkien käyttäjien tarpeita. Lyhyemmät kehityssyklit ja nopeammat päivitysjulkaisut tekevät versioinnista ja olemassa olevien toiminnallisuuksien säilyttämisestä ongelmallista.

GraphQL™ on kyselykieli ja ajonaikainen järjestelmä, jonka tavoitteena on edellä kuvattujen ongelmien ratkaiseminen helposti ymmärrettävällä syntaksilla ja käyttäjävaatimuksiin sopeutumalla. GraphQL on kokonaisvaltainen ratkaisu ongelmaan

nykyaikaisten sovellusten palveluihin yhdistämiseksi muodostamalla perustan uudelle ja tärkeälle kerrokselle ohjelmistokehityksessä: graafille [49]. Uusi kerros kokoaa datan ja palvelut yhteen paikkaan ja mahdollistaa niiden käytön yhtenäisen, turvallisen ja helppokäyttöisen rajapinnan kautta (Kuva 5.1).



Kuva 5.1: GraphQL-kerros datapalveluiden ja käyttäjien välillä.

Kyselykielen suunnitteluun valittiin joitakin periaatteita, joita Buna [10, s. 12] käsittelee kirjassaan. Pyyntöjen tulee olla hierarkkisia kuvastaen vastauksena saatavan datan muotoa. Palvelimella määriteltävä tietomalli on vahvasti tyypitetty, mikä helpottaa pyyntöjen syntaksin ja semantiikan tarkistamista vähentäen virheiden esiintymistä. Asiakas määrittelee palautettavan datan ja rajapintojen suunnittelua tulisikin lähestyä käyttöliittymän vaatimuksista alkaen. Palvelun tulee lisäksi pystyä itsehavainnointiin eli tutkimaan omaa määrittelyään ja rakennettaan. Näiden lisäksi GraphQL on alusta asti halunnut huomioida mobiilikäyttäjien tarpeet koskien rajoittuneempia laiteresursseja ja liikkuvuuden sallimista.

5.1 Historia ja motivaatio

GraphQL sai alkunsa, kun Facebook aloitti spesifikaation kirjoittamisen vuonna 2012 asiakas-palvelin-arkkitehtuurin mukaiselle kommunikaatiolle. Vuonna 2015

spesifikaatio julkaistiin kaikkien saatavaksi [40, s. 17], minkä jälkeen sitä on päivitetty vuosien saatossa ja aktiivinen kehitystyö jatkuu edelleen. Byron [11] kirjoittaa, että lähtölaukauksena GraphQL:n kehittämiselle oli Facebookin puhelinsovelluksissa todetut suorituskyvyn ongelmat ja tarve kerätä yhteen dataa laajasta valikoimasta yrityksen tuotteita ja palveluita. Olemassa olevat ratkaisut kasvattivat turhautumista joko palvelimella tai asiakaslaitteella tapahtuvaan datan muokkaukseen, mikä lisäsi kirjoitettavan ja ylläpidettävän ohjelmakoodin määrää. Kyselykielen omaksuminen on kasvanut merkittävästi, kun se julkaistiin avoimena lähdekoodina eikä tulevaisuus ole näyttänyt merkkejä siitä luopumisesta [52].

GraphQL-kyselykielellä on paljon yhtäläisyyksiä sitä edeltäneiden kyselykielien kanssa, joista kirjassa [40, s. 57] nostetaan esille erityisesti SQL-kieli. Vaikka GraphQL ja SQL ovat molemmat kyselykieliä, niin ne poikkeavat täydellisesti käyttöympäristönsä suhteen: SQL on tarkoitettu käytettäväksi relaatiotietokannoissa ja GraphQL kyselykielenä API-rajapinnassa. GraphQL-kyselyt eivät myöskään kohdistu ainoastaan yhteen ja tiettyyn tietolähteeseen, vaan kyselyt voidaan osoittaa moneen eri kohteeseen, kuten tietokantaan, mikropalveluun tai REST-rajapintaan, ja koostaa yhteen GraphQL-palvelimella.

5.2 Kyselykielisyntaksi

GraphQL-palveluissa tuetut tyypit ja ominaisuudet määritellään tätä tarkoitusta varten suunnitellulla kyselykielisyntaksilla, joka tunnetaan nimellä Schema Definition Language (SDL). GraphQL-spesifikaatio kuvailee kyselykielen syntaksia intuitiiviseksi ja joustavaksi asiakkaiden kanssa tapahtuvaan vuorovaikutukseen. GraphQL ei ole laskennalliseen työhön tarkoitettu ohjelmointikieli, vaan kieli kyselyiden tekemiseksi palveluille, jotka täyttävät spesifikaatiossa määritellyt kriteerit.

Syntaksin avulla kuvattu Kirjailija-tyyppi olisi määriteltävissä seuraavasti:

```
type Kirjailija {
    id: ID!
    nimi: String!
    kansalaisuus: String
}
```

Tyypillä Kirjailija on kolme kenttää, joista ensimmäinen, id, on tyypiltään GraphQL-spesifikaatiossa määritelty identiteettiarvo. Kaksi muuta kenttää, nimi ja kansalaisuus, ovat merkkijonoja. Tietotyypin perässä oleva huutomerkki määrittelee kentän pakolliseksi. Toinen esimerkki syntaksin mukaisesta tyypistä olisi Kirja-tyyppi:

```
type Kirja {
    id: ID!
    nimi: String!
    julkaisuvuosi: Int
    tekija: Kirjailija!
}
```

Kirjan pakollisen tekija-kentän tietotyyppi on Kirjailija, mikä määrittelee viittaussuhteen kirjailijan ja kirjan välillä. Tällöin on tarpeen lisätä viittaus myös Kirjailija-tyypille

```
type Kirjailija {
    id: ID!
    nimi: String!
    kansalaisuus: String
    kirjat: [Kirja]!
}
```

Koska kirjailija on saattanut tuottaa useita kirjoja, niin hakasulkumerkinnällä osoitetaan kirjat-kentän tietotyyppi taulukoksi, mikä sisältää Kirja-tyyppejä.

5.2.1 Tiedonhaku

HTTP-protokollaa käyttävissä sovelluksissa kyselyt välitetään tavallisesti yksinkertaisina merkkijonoina POST-pyynnöissä GraphQL-palvelimen julkaisemaan päätepisteeseen [40, s. 58]. Koska GraphQL-rajapinnoissa päätepisteitä on tavallisesti ainoastaan yksi, niin asiakas määrää kyselyssään, mitä dataa hän haluaa palvelimen palauttavan, toisin kuin REST-arkkitehtuurissa, missä päätepisteeseen yhdistetyt resurssit määrittelevät palautettavan datan muodon. Kyselyt esitetään kuvan 5.2 muodossa, mikä muistuttaa palvelimen palauttaman tiedon rakennetta.

```
query kaikkiKirjailijat {
    nimi
}
```

Kuva 5.2: Asiakkaan lähettämä kysely kirjailijoiden nimistä.

Valinnainen avainsana query osoittaa kyselyn tiedonhakuoperaatioksi. Kyselyn nimi on ”kaikkiKirjailijat” ja siinä jokaiselta kirjailijalta pyydetään tietoa hänen nimestään. Kyselyyn palautettaisiin vastauksena seuraavan kaltainen lista kirjailijoista:

```
"data": {
    "kaikkiKirjailijat": [
        {"nimi": "George R. R. Martin"},
        {"nimi": "Haruki Murakami"},
        {"nimi": "Agatha Christie"}
    ]
}
```

Vastauksessa palautetaan tieto ainoastaan jokaisen kirjailijan nimestä, koska se oli ainoa kyselyssä pyydetty kenttä. Kansallisuustiedon saamiseksi riittää, että sille varattu kenttä lisätään kyselyyn

```
query kaikkiKirjailijat {
    nimi
    kansallisuus
}
```

GraphQL-rajapinnan kyselyille ja kentille voidaan määritellä myös argumentteja. Esimerkiksi seuraava kysely hakee kirjailijan, jonka identiteettiarvo on seitsemän

```
query kirjailija(id: 7) {
    nimi
    kansallisuus
}
```

Viitattujen tyyppien tietojen hakeminen kyselyssä on yksinkertaista niiden hierarkkisen rakenteen ansiosta. Lisäämällä edelliseen kyselyyn kirjat-kenttä, palvelin palauttaa dataa myös haetun kirjailijan teoksista.

```
query kirjailija(id: 7) {
  nimi
  kansallisuus
  kirjat {
    nimi
  }
}
```

5.2.2 Tiedonmuokkaaminen

Tiedon muokkaamista varten GraphQL-spesifikaatiossa on määritelty kyselytyyppi mutation.

```
mutation lisaaKirjailija(nimi: "Mika Waltari", kansallisuus: "Suomi") {
  id
}
```

Kyselylle "lisaaKirjailija" annetaan argumentteina kirjailijan nimi ja kansallisuus. Tietoa muokkaavissa operaatioissa asiakas voi samalla pyytää tarvitsemaansa dataa palvelimelta sen vastauksessa, mikä vähentää edestakaista viestiliikennettä. Edellä uuden kirjailijan lisäämisen jälkeen palvelin palauttaa kirjailijan identiteettiarvon asiakkaalle.

Kolmantena kyselytyyppinä GraphQL-spesifikaatiossa on määritelty subscription eli tilaus. Kysely on syntaksiselta rakenteeltaan vastaavan kaltainen edellisten kanssa, mutta avainsana query tai mutation on korvattu termillä subscription.

Subscription-kyselyillä tilataan tietoa palvelimelta, jolloin asiakkaalle välitetään tietoa tilauksen kohteeseen kohdistuneista tapahtumista. Tapahtuma voi olla esimerkiksi sosiaalisen median julkaisulle saapunut uusi kommentti.

5.2.3 Skeema

Kaikki edellä kuvattu määritellään GraphQL-rajapinnalle kirjoitetussa skeemassa. Skeema kuvailee rajapinnan ominaisuudet ja toimii sopimuksena asiakkaalle takaamaan, mitä kyselyjä ja tietotyyppejä rajapinta tukee. Skeema on kokoelma tyyppejä, joista juurityypeillä Query, Mutation ja Subscription on erityisasema. Juurityypeistä ainoastaan Query on pakollinen, eikä spesifikaatiota täyttäviltä rajapinnoilta vaadita tukea Mutation- ja Subscription-tyyppisille kyselyille. Juurityypit ovat kyselyiden alkupisteitä ja sisältävät kyselytyyppinsä edustajat.

```
type Query {
    kaikkiKirjailijat: [Kirjailija]!
}
```

Tiedonhakukysely "kaikkiKirjailijat" sijoitetaan juurityyppinsä sisään ja sille määritellään sen palauttama tietotyyppi, mikä edellä on taulukko Kirjailija-tyyppejä. Määrittelyyn kirjoitetaan myös kyselyn tukemat parametrit, jos niitä on:

```
type Mutation {
    lisaaKirjailija(nimi: String!, kansallisuus: String): Kirjailija!
}
```

Skeemassa määritellyt tietotyypit sisältävät tiedon niihin kuuluvista kentistä ja kenttien tietotyypeistä. Tyypit ovat olennainen osa palvelulle luotua skeemaa. Spesifikaatio määrittelee yhteensä kuusi nimettyä tyyppiä: skalaari, objekti, rajapinta, yhdiste, lueteltu tyyppi ja objektiargumentti. Näiden lisäksi tyypeistä voidaan muodostaa kokoelmia taulukoiden avulla ja non-null tyypillä varmistaa tiedon olemassaolo.

Pyynnöissä haettavat kentät ovat aina jonkin tyypin ilmentymiä. Objektikenttien avulla pyynnön suorituksesta rakentuu puu, jossa objektit vastaavat puun välisolmuja. Objektien käyttö mahdollistaa toisiinsa viittaavien kohteiden tiedon hakemisen tai muokkaamisen samassa pyynnössä. Jokaiselta objektityypiltä tulee pyytää vähintään yhtä sille määriteltyä kenttää, koska puussa rakentunut polku ei voi päättyä objektiin. Polun päättävää kenttää kutsutaan lehdeksi ja ne voivat olla ainoastaan skalaarityyppiä. Spesifikaatiossa määriteltyjä skalaarityyppejä ovat kokonais- sekä liukuluvut, merkkijonot, totuusarvot ja identiteettiarvot. Identiteettiarvojen tehtävänä on toimia yksilöivänä tunnisteena objekteille tai välimuistin avaimille. Skalaareilla ei tarvitse aina olla arvoa, vaan ne voivat saada myös arvon null.

Vastaavasti kuten kyselyoperaatiot, myös kentät voidaan tulkita operaatioiksi, jotka palauttavat niille määritellyn muotoista tietoa ja niille voidaan välittää argumentteja [10, s. 44 – 45]. Tämä on perusteltavissa kenttien funktiomaisen luonteen avulla.

5.3 Arkkitehtuuri

GraphQL-rajapintojen verkkomainen arkkitehtuuri luo uuden tavan hakea tietoa viitatuista resursseista. Hierarkkinen syntaksi ja verkkorakenne ovat käytännöllisiä ominaisuuksia siirryttäessä resurssista toiseen. Matemaattinen verkko on solmuista ja niitä yhdistävistä linkeistä muodostuva malli, missä linkit kuvastavat solmujen suhdetta toisiinsa. GraphQL:n tietomallia havainnollistavassa verkossa solmuna kuvatusta objektista ei tarvitse olla linkkiä muihin objekteihin, mutta linkkejä voi olla myös yksi tai useampi [2]. Solmujen väliset linkit ovat suuntaamattomia, jolloin linkillä yhdistetyillä objekteilla on viittaus toisiinsa ja tekee objektien välisistä suhteista syklisiä

GraphQL-rajapinta on asiakassovellusten ja tietolähteiden väliin luotu kerros, josta kaikki data on kerralla saatavissa. Tietolähteiden yhdistäminen abstrahoivan rajapinnan kautta vähentää asiakassovelluksella tehtävien pyyntöjen lukumäärää kaiken tarvittavan datan saamiseksi. Vaikka tyypillisessä GraphQL-rajapinnassa kaikki asiakaspyynnöt tapahtuvat yhden URI-osoitteen kautta [37], niin GraphQL-palvelu voi kuitenkin käyttää datan hakemiseen ja koostamiseen useita päätepisteitä. Matkailupalvelu voisi esimerkiksi hakea hotellitietoja tietokannasta ja paikkatietoa Google Maps -rajapinnasta. Tietolähteen vaihtuminen tai sen version muuttuminen ei vaikuta asiakkaalle näyttäytyvään GraphQL-rajapintaan, mikä helpottaa asiakas- ja

palvelinsovellusten kehittämistä itsenäisinä komponentteina [5]. Koska asiakassovellukset määräävät palautettavan datan, niin GraphQL-rajapinta voi kehittyä ilman versioinnin haasteita. Uusien kenttien lisääminen jo olemassa olevaan tietomalliin ei riko asiakassovelluksia. Kerrosmaisen arkkitehtuurin ja riippumattomuutensa ansiosta GraphQL on sovellettavissa jo olemassa olevien tietojärjestelmien sekä täysin uusien kanssa, kunhan ne kykenevät täyttämään GraphQL:n spesifikaatiossa yhtenäiseksi tarkoitetun määrittelyn [56].

Toinen merkittävä ominaisuus, mikä vähentää kaistankäyttöä, on GraphQL-rajapintojen mahdollisuus välittää vain ja ainoastaan tarvittavaa dataa [37]. Resurssikohtaisten päätepisteiden sijasta GraphQL-rajapinnat rakentuvat hyvin määriteltyjen tyyppien ja kenttien avulla. Jos sovellus tarvitsee tiedon ainoastaan kirjailijan kansallisuudesta, niin REST-rajapinnoissa joudutaan kirjoittamaan ylimääräistä ohjelmalogiikkaa palvelimelle tai siirtämään koko kirjailijaobjekti (Kuva 5.3) asiakkaalle, missä suoritetaan datan suodatus.

```
"data": {
    "kirjailija": {
        "id": 7,
        "nimi": "Haruki Murakami",
        "kansalaisuus": "Japani"
    }
}
```

Kuva 5.3: Rajapinnan palauttama kirjailijaobjekti kokonaisuudessaan.

GraphQL suunniteltiin tehostamaan ja parantamaan kehittäjäkokemusta tiedonkäsittelyssä asiakas-palvelin-arkkitehtuuria noudattavissa sovelluksissa [52]. Tiedonhakuprosessi haluttiin suunnitella uudestaan aloittaen käyttöliittymäsuunnittelijoiden ja -kehittäjien tarpeista [11]. Jos käyttöliittymässä tarvitaan tieto ainoastaan kirjailijan kansalaisuudesta, niin koko kirjailijaobjektin sijasta GraphQL-rajapinnasta voidaan pyytää vain kansalaisuustieto:

```
"data": {
    "kirjailija": {
        "kansalaisuus": "Japani"
    }
}
```

Suunnittelussa tehtiin paljon teknologiariippumattomia valintoja, esimerkiksi ohjelmointikielten ja tietokantajärjestelmien suhteen. Käytettäviä protokollia ei myöskään määritelty, mutta tyypillisimmät toteutukset web-sovellusympäristöissä rakentuvat HTTP-protokollan päälle. Spesifikaatioon pohjautuvia ohjelmointi- ja sovelluskirjastoja on kehitetty runsaasti, minkä edellytyksenä Porcello ja Banks [40, s. 31] pitävät toteutuksen suhteen annettuja vapauksia. Tunnettuja ja suosittuja ohjelmointikirjastoja ovat mm. Facebookin kehittämä Relay ja Meteorin Apollo.

5.4 Selvittäjäfunktiot

GraphQL-arkkitehtuurissa tehty päätös [55] alustariippumattomuudesta on saavutettu ajonaikaisella järjestelmällä, missä tiedonhausta vastaa tarkoitusta varten kirjoitettu ohjelmakoodi. Ohjelmakoodin suorittamat operaatiot vastaavat asiakaskyselyihin datalla, joka vastaa palvelimella määriteltyjä tyyppejä. Jokainen tyyppi ja kenttä määritellään ympäristössä käytetyllä ohjelmointikielellä. GraphQL-rajapintojen rakenteesta vastaa edellä kuvattu skeema. Toiminnalliset ominaisuudet on puolestaan annettu selvittäjäfunktioiden vastuulle [41]. Selvittäjäfunktioiden tehtävänä on kenttien ja operaatioiden arvojen hakeminen ja laskeminen. Edellä kuvatun kirjailija-hakuoperaation ja Kirjailija-tyypin nimen selvittäjäfunktioina voisi JavaScript-kielisessä ympäristössä toimia funktiot

```
function Query_kirjailija(parent, args, context_info, schema_info) {
  return kirjailijat.getById(kirjailija => kirjailija.id === args.id);
}
```

ja

```
function kirjailija_nimi(parent, args, context_info, schema_info) {
  return parent.nimi;
}
```

Selvittäjäfunktiot ovat GraphQL-palvelimelta vaadittu ominaisuus tiedon evaluoimiseksi ja jokaisen tietotyypin kaikilla kentillä tulee olla selvittäjäfunktio, jota kutsutaan kentän arvoa pyydettäessä [40, s. 135 – 144]. Kentät voidaan tulkita myös isäntätyyppinsä funktioiksi, jotka palauttavat kentässä esitetyn tyypin mukaisen arvon. Monissa GraphQL-ohjelmointikirjastoissa on haluttu helpottaa kehittäjien työtä sallimalla yksinkertaisten selvittäjäfunktioiden poisjättäminen. Ohjelmointikirjastoissa saattaa olla apufunktioita tai abstrahoivia tietorakenteita, jotka automatisoivat selvittäjäfunktioiden toteuttamisen. Käytetystä ohjelmointikielestä riippuen kentän nimen voidaan myös olettaa vastaavan jotain kyseisen kielen tietorakenteen ominaisuusarvoa.

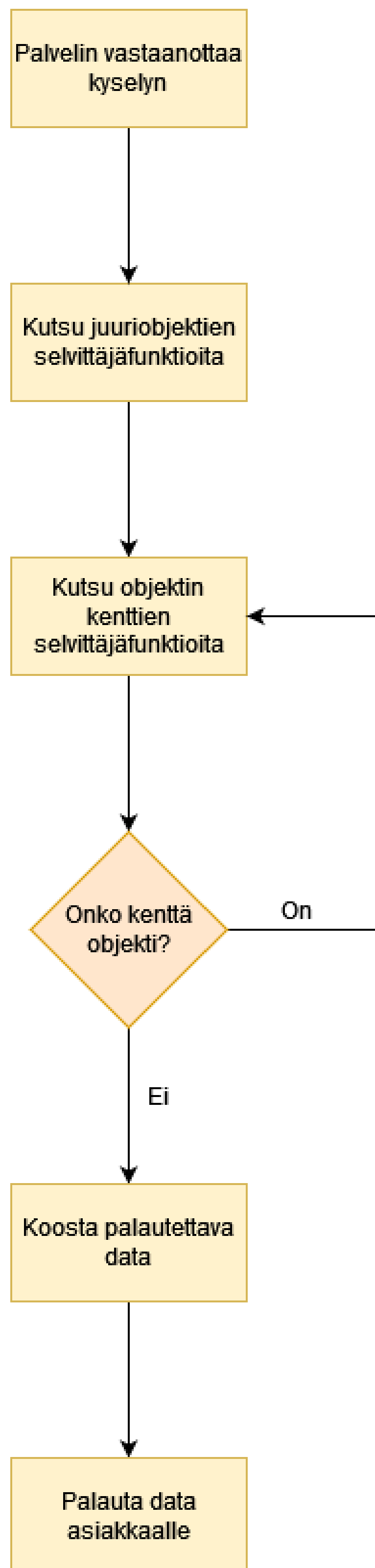
Spesifikaatiossa [56] ja kirjassa [40] on kirjoitettu selvittäjäfunktioille välitettävistä neljästä argumentista. Ensimmäinen argumentti on kenttää edeltävä objekti puurakenteessa. Toisena argumenttina selvittäjäfunktio saa pyynnölle annetut argumentit. Selvittäjäfunktion kaksi viimeistä argumenttia ovat vahvasti sidoksissa rajapinnan toteutukseen. Kolmas argumentti on kontekstiriippuvaista informaatiota, mihin voidaan sisällyttää esimerkiksi tietoa kirjautuneesta käyttäjästä tai käytetystä tietokantayhteydestä. Viimeisessä argumentissa on skeemaan ja kenttään liittyvää tietoa.

5.5 Operaatioiden suoritus

Vastaanotettuaan asiakaspyynnössä välitetyn kyselyn, GraphQL-palvelu suorittaa tarkistuksen, jotta varmistutaan, että viitatut tyypit ja kentät ovat hyvin määriteltyjä. Tarkistuksen jälkeen selvittäjäfunktiot toteuttavat tulosten koostamisen. GraphQL-rajapintaan kohdistuneiden asiakaspyyntöjen suorittamista havainnollistetaan usein verkko- tai puurakenteina [40][55]. Asiakaspyynnössä lähetetty dokumentti toimii puun juurena haarautuen siinä viitattuihin objekteihin ja kenttiin. Objektien välille muodostuu linkkejä, jotka päättyvät lopulta puun lehtiin ja palautettaviin arvoihin.

GraphQL sallii usean kyselyn esittämisen samassa dokumentissa, mikä on tehnyt aiheelliseksi määrätä joitakin käytänteitä kyselyiden suoritusjärjestykselle. Koska tiedon hakeminen ei aiheuta muutoksia olemassa olevaan dataan, niin lukuoperaatiot voidaan suorittaa yleensä rinnakkain [56]. Tiedon muokkaamisen tarkoitettujen Mutation-kyselyiden ajaminen toteutetaan tavallisesti sarjassa eli seuraava kyselyoperaatio voi alkaa vasta edellisen päätyttyä. Näin varmistutaan, ettei käyttäjälle kehity itseään vastaan kohdistuvaa kilpailutilannetta, jossa yhteen tietorakenteeseen kohdistuvat operaatiot yritetään suorittaa samanaikaisesti. Jos asiakaspyynnössä lähetetty dokumentti sisältää saman kyselyn useaan kertaan, niin esiintymät tulee nimetä yksilöivillä tunnisteilla. Yksi dokumentti ei voi myöskään käyttää sekä nimeämättömiä että nimettyjä kyselyitä [10, s. 85].

Yksittäisen pyynnön selvittäminen toteutetaan rekursiivisesti ja spesifikaatio [56] tarkentaa suoritusta algoritmilla. Prosessia voidaan havainnollistaa kuvan 5.4 vuokaaviolla. Jokaisen polun tulee aina päättyä skalaariarvoon, mikä merkitsee polun saapumista puun lehteen ja polussa syntyneen ketjun päätymistä. Objektien tapauksessa polku haarautuu jokaiseen objektilta haettavaan kenttään. Koska kenttien arvot selvitetään yleensä rinnakkain, niin tyypillisesti aktiivisia polkuja on samanaikaisesti useita. Lopullinen polulta palautettava arvo saadaan sen päättävältä kentältä. Arvo palautetaan siirtymällä puussa ylöspäin aina juureen asti. Palautettava tietorakenne heijastaa esitetyn pyynnön rakennetta, missä haettu data on kuvattuna avain-arvo-pareina. Tiedon esitysmuotoa ei ole määrätty, mutta hyvin usein käytännön toteutukset tukeutuvat JSON-tietomuotoon.



Kuva 5.4: Vuokaavio kyselyn selvittämisestä palvelimella.

Kyselyn suoritus voidaan jakaa taulukon 5.1 vaiheisiin, joita on havainnollistettu kuvissa 5.5, 5.6, 5.7, 5.8 ja 5.9.

Taulukko 5.1: Kyselyn suorituksen vaiheet.

Vaihe	Selitys
1	Asiakas lähettää kyselyn palvelimelle.
2	Palvelimella kutsutaan Kirjailija-tyypille määriteltyä selvittäjäfunktiota. Selvittäjäfunktiossa voidaan esimerkiksi edelleen kutsua relaatiotietokantaan kohdistuvista operaatioista vastaavaa palvelua, joka palauttaa asiakaskyselyn argumenttina annettua identiteettiarvoa vastaavan kirjailijaobjektin.
3 i	Palvelin kutsuu kyselyssä haettujen kenttien selvittäjäfunktioita, joiden suoritus voi tapahtua rinnakkain. Kenttien selvittäjäfunktioiden ensimmäisenä argumenttina annetaan edellisen kohdan kirjailijaobjekti.
3 ii	Nimen ja kansallisuuden selvittäjäfunktiot.
3 iii	Kirjat-kentälle toimittaisiin vastaavasti kuin kohdassa 2. Aluksi haetaan kirjaobjektit, minkä jälkeen jokaisella kirjalla kutsuttaisiin nimen selvittäjäfunktiota. Kirjan nimen selvittäjäfunktio saisi ensimmäisenä argumenttinaan kirjaobjektin, koska se on kentän vanhempi puurakenteessa. Kirjojen hakemiseen voitaisiin käyttää esimerkiksi joko toista tietokantaa tai REST-rajapintaa.
4	Selvitysprosessi päättyy ja data palautetaan asiakkaalle.

```
query kirjailija(id: 7) {
  nimi
  kansallisuus
  kirjat {
    nimi
  }
}
```

Kuva 5.5: Taulukon 5.1 vaihe 1.

```
function Query_kirjailija(parent, args, context_info, schema_info) {
  return kirjailijat.getById(kirjailija => kirjailija.id === args.id);
}
```

Kuva 5.6: Taulukon 5.1 vaihe 2.

```
function kirjailija_nimi(parent, args, context_info, schema_info) {
  return parent.nimi;
}

function kirjailija_kansallisuus(parent, args, context_info, schema_info) {
  return parent.kansallisuus;
}
```

Kuva 5.7: Taulukon 5.1 vaihe 3ii.

```
function kirjailija_kirjat(parent, args, context_info, schema_info) {
    return parent.kirjat;
}

function kirja_nimi(parent, args, context_info, schema_info) {
    return parent.nimi;
}
```

Kuva 5.8: Taulukon 5.1 vaihe 3iii.

```
"data": {
    "kirjailija": {
        "nimi": "Haruki Murakami",
        ”kansallisuus”: ”Japani”,
        "kirjat": [
            {"nimi": "Suuri lammasseikkailu"},
            {"nimi": "Kafka rannalla"},
                        .
                        .
                        .
            {"nimi": "1Q84"}
        ]
    }
}
```

Kuva 5.9: Taulukon 5.1 vaihe 4.

5.5.1 Validointi

Syntaksin virheettömyyden lisäksi lähetettyjä dokumentteja tulkittaessa tulee varmistua käsiteltävien kenttien yksikäsitteisyydestä ja ymmärrettävyydestä ympäröivässä kontekstissa. Virheelliset pyynnöt saattavat olla suoritettavissa, mutta niiden palauttamat tulokset voivat poiketa virheettömien pyyntöjen tuloksista [56]. Virheiden ilmaantuessa, palvelin palauttaa vastauksessaan listan niistä ja vastaus voi sisältää myös lähetetyssä dokumentissa pyydettyä tietoa [56]. Jos palautettavaa tietoa ei ole, niin virheet ilmoittavat syyn sen puuttumiselle. Jos virheitä ei lainkaan esiinny pyynnön suorituksen yhteydessä, niin mitään virhettä ei tule myöskään asiakkaalle palauttaa.

Tavallisesti dokumentin kyselyt validoidaan juuri ennen niiden suoritusta. On kuitenkin mahdollista, että validointi ohitetaan [56], jos palvelu muistaa hyväksyneensä täsmälleen saman pyynnön aikaisemmin. Optimointi ei tosin ole aina kannattavaa. Aikaisemmin hyväksytty pyyntö voi muuttua kelpaamattomaksi skeemassa tapahtuneiden muutosten seurauksena. Tämän kaltaisia järjestelmän toimintaan vaikuttavia muutoksia suositellaan vältettäväksi kehitys- ja ylläpito-ongelmien minimoimiseksi.

Tyyppijärjestelmän vahva tyypitys tukee validointia. Monet yleiset virheet, kuten objektiin kuulumattomien kenttien pyytäminen tai skalaari- ja objektikenttien sekoittaminen, ovat vältettävissä järjestelmän avustuksella [10]. Validoinnin palauttamilla virheillä voidaan estää myös vakavampien ongelmatilanteiden kehittyminen. Koska kyselyiden selvittämisestä rakentuu objektien muodostama verkko, niin riskinä on syklin kehittyminen ja päättymättömään laskentaan ajautuminen. Erilaisilla muokattavissa olevilla validointisäännöillä voidaan kuitenkin estää tämän tapahtuminen.

5.6 Introspektio

GraphQL-palveluiden introspektio-ominaisuus antaa käyttäjille mahdollisuuden pyytää skeemaan liittyvää metatietoa palvelimelta (Kuva 5.10). Kyselystä riippuen vastaus sisältää tietoa skeeman rakenteesta, tyyppien määrittelystä tai käytetyistä nimistä ja argumenteista [10, s. 59 – 64] (Kuva 5.11). Tyypeille ja niiden kentille voidaan kirjoittaa kuvauksia [56], jotka ovat siirrettävissä automaattisesti dokumentaatioon. Kentät voidaan merkitä myös vanhentuneiksi, jolloin käyttäjät osaavat varau-

tua pyyntöjen rakenteen muuttamiseen. Kahdella alaviivalla alkavat tyypit ja kentät ovat osa introspektiojärjestelmää ja sen paljastamaa rajapintaa. Ratkaisulla on pyritty välttämään nimeämisissä ilmaantuvia ristiriitoja. Esimerkiksi kaikilta spesifikaatiota noudattavilta toteutuksilta vaaditaan ainakin __type ja __schema kenttien tukeminen.

```
query __type(name: "Kirjailija") {
    name
    kind
    description
}
```

Kuva 5.10: Kysely Kirjailija-tyypin metatiedoista.

```
"data": {
    "__type": {
        "name": "Kirjailija",
        "kind": "OBJECT",
        "description": "Henkilö, joka kirjoittaa kirjoja."
    }
}
```

Kuva 5.11: Metatietoa Kirjailija-tyypistä.

Introspektiolla on merkittävä rooli palveluiden dokumentaatiossa sekä niiden kehitystyöhön ja käyttöön suunnattujen työkalujen ja sovellusten tuottamisessa. Käyttäjille suunnattuihin asiakassovelluksiin on pystytty kehittämään muun muassa automaattiseen täydennykseen ja ennakoiviin varoituksiin kykeneviä ominaisuuksia.

5.7 GraphQL ja web-sovellukset

Bunan [10, s. 6 – 7] mukaan GraphQL-palveluiden käyttö web-ympäristöissä rakentuu kyselykielen ja ajonaikaisen järjestelmän ympärille. Yhteisen kyselykielen avulla asiakas- ja palvelinsovellukset pystyvät kommunikoimaan ja siirtämään tietoa keskenään. Ajonaikaisena järjestelmänä GraphQL on kokonaisarkkitehtuuriin sisältyvä kerros, joka mahdollistaa kyselyjen kääntämisen, ymmärtämisen ja niihin vastaamisen. Tämä tekee ajonaikaisesta järjestelmästä oleellisen rakenteen GraphQL:n käyttöönotossa. Koska GraphQL on piittaamaton käytetyn teknologian suhteen, niin ajonaikainen järjestelmä on riippumaton muusta sovellus- tai laitteistoympäristöstä. GraphQL-spesifikaatio on antanut monelle kehittäjälle ja yhteisölle innoitusta siirtää sen kuvaama arkkitehtuuri käytäntöön. Seurauksena on syntynyt useita eri ohjelmointikielille tai niiden kirjastoille suunnattuja konkreettisia toteutuksia.

Vogel et al. [61] selvittivät älykotijärjestelmässä käytettävän REST-rajapinnan vaihtamista GraphQL-rajapintaan ja vertailivat ratkaisuja keskenään. He toteavat, että GraphQL voi toimia kerroksena joko osana tai erillään muusta palvelinrakenteesta. Edeltävässä ratkaisussa GraphQL-palvelulla on suora pääsy järjestelmän muihin palveluihin, jolloin ylimääräisiä verkon välityksellä tapahtuvia pyyntöjä ei tarvita. Jälkimmäisessä tapauksessa GraphQL toimii viestiliikennettä välittävänä rajapintana muihin palveluihin. Tämän kaltaista mallia on käytetty esimerkiksi mikropalveluarkkitehtuuria noudattavissa järjestelmissä muodostamaan yhtenäinen rajapinta käyttäjille.

Koska GraphQL-skeema voidaan muodostaa myös osista, niin mikropalveluarkkitehtuurissa jokainen palvelu voi määritellä oman skeemansa, jotka ovat yhdistettävissä yleiseksi skeemaksi yhteen rajapintaan [14]. Näin asiakkaat voivat tietämättään kohdistaa kyselyitä useaan palveluun samassa dokumentissa rajapinnan toimiessa välityspalvelimena. Yhteisen GraphQL-skeeman käyttö asettaa suurtenkin järjestelmäarkkitehtuurien tietomallille yksikäsitteisen totuuden (single source of truth) ja jaetun rajapinnan kautta voidaan hallita yritysten ja organisaatioiden kaikkia järjestelmiä [48]. Koska GraphQL ei sanele tiettyä sovellusarkkitehtuuria, niin se on sovitettavissa jo olemassa oleviin järjestelmiin ja hyödynnettävissä aikaisempien rajapintatyökalujen kanssa.

Web-sovelluksia kehitettäessä yhdeksi haasteeksi kirjassa [10, s. 86 – 87] esitetään versiointi. Tyypillisesti jotkut käyttäjistä ovat riippuvaisia vielä jatkossakin vanhoista ominaisuuksista, jolloin vähintään siirtymävaiheen ajan tulee ylläpitää vanhoja toimintoja yhdessä uusien kanssa. Rajapinnan rakenne tulee pystyä organisoimaan

siten, että pyynnöt pystytään kohdistamaan haluttuihin versioihin. GraphQL:n kannalta versiointi on tarpeetonta ja se siirtää vastuun haettavista resursseista ja niiden kentistä käyttäjälle. Uusia kenttiä määriteltäessä riittää, että niitä tarvitsevien asiakassovelluksien pyyntöjä muokataan uusien kenttien osalta. Poistettaessa skeeman määrittelemien tietotyyppien kenttiä, GraphQL tarjoaa vanhentumisesta ilmoittavan direktiivin käyttämisen. Tällöin käyttäjille välittyy tieto tulevasta muutoksesta ja heille annetaan mahdollisuus reagoida päivittämällä pyyntöjään.

5.7.1 GraphQL vs. REST

Julkaisustaan lähtien GraphQL:ää on verrattu RESTiin ja esitetty ennusteita REST-rajapintojen korvautumisella GraphQL-rajapinnoilla, mitä Porcello ja Banks [40, s. 23] eivät kuitenkaan pidä täysin suoraviivaisena kehityksenä. REST ja GraphQL eivät ole toisiaan poissulkevia teknologioita, vaan ne pystyvät toimimaan yhteistyössä. Esimerkiksi GraphQL-palvelu voi käyttää tietolähteenään REST-rajapinnasta saatavaa dataa.

RESTissä on ajan saatossa havaittu heikkouksia, joiden on arvioitu ratkeavan siirtymällä GraphQL:n käyttöön, mutta täydellisen RESTful-arkkitehtuurin[1] toteuttaminen nähdään kuitenkin harvoin järkeväksi todellisissa sovelluksissa. Useat REST-rajapinnat tarjoavat asiakkailleen mukautettuja päätepisteitä tiedonhakua tai -muokkausta varten. Tämä parantaa usein suorituskykyä, mutta lisää samalla sovelluksen ja rajapinnan välistä riippuvuutta toisistaan.

Esimerkiksi kirjailijoista ja heidän teoksistaan tietoa tarjoava rajapinta voisi täydellisessä RESTful-arkkitehtuurissa toimia seuraavasti. Päätepisteeseen "/api/kirjailijat/6" tehtävä pyyntö palauttaa tiedon kirjailijasta tunnistearvolla kuusi ja listan hänen kirjoittamiensa teosten tunnisteista. Lisätietojen hakeminen jokaisesta teoksesta vaatisi uuden kyselyn päätepisteeseen "/api/teokset/TEOS_ID", missä TEOS_ID on teoksen tunniste. Poikkeamalla RESTful-arkkitehtuurin periaatteista kirjailijan hakemiseksi tehtävässä pyynnössä voitaisiin palauttaa myös kaikkien hänen teostensa tiedot, mikä tosin kasvattaisi siirrettävän datan määrää.

Tutkimuksessa [61] tehtiin kaksi koetta REST- ja GraphQL-rajapintojen suorituskykyjen vertailemiseksi. Yksittäistä resurssia tai ainoastaan yhtä sen kentistä haettaessa GraphQL ei antanut mainittavaa hyötyä viivettä arvioitaessa. Toisessa kokeessa haettiin kolme erillistä resurssia. Käyttäessään kolmea erillistä pyyntöä RES-

[1] https://martinfowler.com/articles/richardsonMaturityModel.html

Tillä aiheutuva viive oli huomattavasti suurempi kuin GraphQL:n yhdellä pyynnöllä. REST-rajapintaa muokkaamalla voitiin välittää kaikki resurssit yhdessä pyynnössä, mikä pienensi viiveissä havaittua eroa. REST-arkkitehtuurissa ongelmana ei ole kuitenkaan vain kaiken tiedon kerääminen yhteen useammalla pyynnöllä ja siitä aiheutuva suurempi viive. Pidempään kestävä tiedonsiirto kasvattaa energiankulutusta, mikä tulee huomioida akku- ja paristokäyttöisillä laitteilla. Myös ohjelmalogiikan uudelleenkäyttö hiemankaan poikkeavan datan käsittelyssä on nähty vaikeaksi [5].

Koska GraphQL on riippumaton käytetystä protokollasta, niin spesifikaatio kuvaa toiminnan abstraktilla tasolla esitetyllä kommunikaatiolla. Konkreettiset toteutukset, erityisesti web-sovelluksissa, tukeutuvat tavallisesti HTTP-protokollaan. Buna [10, s. 11] selventää, että mikään HTTP-protokollan ominaisuus ei kuitenkaan ole edellytys GraphQL:lle toisin kuin RESTille. REST-arkkitehtuurin fundamentaalisiin tekijöihin kuuluu HTTP-protokollan ominaisuuksien, kuten pyyntötyyppien, tilakoodien ja otsakkeiden soveltaminen. Yhtenä haittana HTTP:n ominaisuuksien hyödyntämättä jättämisessä GraphQL:lle on Vogel et al. [61] mukaan uusien tietoturvakäytäntöjen omaksuminen ja HTTP-protokollan tukeman välimuistin puuttuminen. Koska GraphQL-palvelimelta haettavia resursseja ei voida tunnistaa URI-osoitteilla, niin HTTP-otsakkeilla määrättyjä välimuistiominaisuuksia ei voida käyttää. Välimuistin käytöllä voidaan pienentää viivettä tulevissa pyynnöissä silloin kun aikaisempi vastaus on palautettavissa muuttumattomana. GraphQL:ssä välimuistin vastaava hyödyntäminen on kuitenkin monimutkaista, koska yhden päätepisteen takia kyselyissä tarvittaisiin kenttäkohtaista välimuistin käyttöä [14]. GraphQL:n ympärille muodostunut yhteisö on tunnistanut ongelman ja työskennellyt sen ratkaisemiseksi tuottamalla ohjelmakirjastoja tarkoitusta varten. Vaivannäöstä huolimatta kaikkia haasteita ei ole kuitenkaan onnistuttu ratkaisemaan esimerkiksi eri selain- ja mobiilialustoilla.

Luis Weir on kirjassaan [62, s. 164 – 165, 194 – 203] todennut aikaisempien rajapinta-arkkitehtuurien korvautuneen yhä useammin suosiotaan edelleen kasvattavilla REST-, GraphQL- ja gRPC-arkkitehtuureilla. Hän vertaili edellä mainittuja arkkitehtuureja eri kriteerein ja arvioitavaksi joutuivat mm. kehityskokemus, ylläpidettävyys ja pääsynhallinta. GraphQL sai kehuja soveltuvuudestaan käyttöliittymäkehitykseen, eri rajapintojen yhdistämisestä ja versiointikäytänteistä. Moitittaviksi tekijöiksi Weir [62] esitti palvelinsovellusten kehittämistyön ja pääsynhallinnan, joissa puolestaan REST pärjäsi hyvin. Kokonaistuloksissa GraphQL ja REST saivat suun-

nilleen yhtä hyvän arvion hyötyjen ja haittojen tasapainottaessa molempien käytettävyyttä. gRPC sai selvästi huonomman kokonaisarvion, mitä on perusteltu sen pääasiallisella käytöllä palveluiden välisessä kommunikaatiossa, eikä niinkään selainsovelluksissa.

5.8 GraphQL ja esineiden internet

Koska esineiden internetin yleistyminen eri käyttötarkoituksissa on kasvattanut tiedonsiirron tarvetta, niin kirjassa [10, s. 10] on GraphQL:ää pidetty varteenotettavana vaihtoehtona IoT-järjestelmille. GraphQL-spesifikaatiota noudattavan abstraktiokerroksen lisäämistä IoT-laitteiden kommunikaatioon on selvitetty Khanin ja Mianin artikkelissa [30], jossa he pyrkivät vähentämään pyyntöjen lukumäärää ja pienentämään energiankulutusta. Tavoitteen saavuttaminen voi vaikuttaa merkittävästi IoT-sovellusarkkitehtuuriin ja laitteiden elinikään, sillä tiedonvälitys on sensorimoduuleille suurimpia virrankuluttajia.

Khan ja Mian käyttivät Zolertian Z1 kehitysalustoja, joiden vähävirtainen mikrokontrolleri sisälsi aktiivisessa tilassa 16 MHz kellotaajuudella toimivan prosessorin, 8 kB RAM-muistia ja 92 kB sisäistä muistia. Alustoilla oli asennettuna Contiki OS käyttöjärjestelmä ja kommunikaatio tapahtui asiakas-palvelin-arkkitehtuurin mukaisesti CoAP-protokollalla. GraphQL-kyselyiden kääntäminen tapahtui pilvipalvelussa sijaitsevan funktiosovelluksen avulla.

Tutkimuksen [30] tuloksena saavutettu kommunikaation optimointi todettiin hyödylliseksi sekä harvoissa että tiheissä IoT-verkoissa. Kokeellisissa mittauksissa käytettiin ainoastaan neljää laitetta, mutta energiatehokkuudessa havaittiin huomattava, noin 50% paraneminen. Useita resursseja pyydettäessä GraphQL-toteutuksen viive oli myös merkittävästi pienempi kuin REST-arkkitehtuuria noudattavassa. Testeissä otettiin kantaa myös törmäysten vähentymiseen kommunikaation harvetessa, missä ei kuitenkaan saavutettu merkittävää poikkeamaa arkkitehtuurien välillä. Yhtenä tutkimuksen saavutuksena Khan ja Mian mainitsevat GraphQL-kyselykielen käytön IoT-laitteilla havainnollistaneen uuden tavan kommunikoida API-rajapintojen kanssa.

Lisäksi Khan ja Mian esittivät vaihtoehtoisen ratkaisun usean hypyn kommunikaatiolle, mikä aiheuttaa ylimääräistä kuormittumista enemmän tietoliikennettä välittävillä, lähellä sinkkiä sijaitsevilla noodeilla. GraphQL:n mahdollistama energiansäästäminen pystyttiin hyödyntämään suurempana lähetystehona ja useampi

lyhyen kantaman lähetys oli korvattavissa yhdellä pidemmän kantaman lähetyksellä. Tulokset vaikuttivat lupaavilta: viiveen ja kokonaisenergiankulutuksen havaittiin pienentyneen ja vastaustiheyden kasvaneen suurempaa lähetystehoa käytettäessä. Suurempi vastaustiheys pienensi kommunikaatiolle tarvittavaa aikaa, mikä vaikutti myös kokonaisenergiankulutukseen suotuisasti.

GraphQL-palvelun toteuttaminen säännöllisesti lepotilan ja aktiivisen tilan välillä vaihtavilla noodeilla on antanut samansuuntaisia tuloksia pienemmästä kommunikaatioviiveestä ja parantuneesta energiansäästöstä. Tutkimuksessa [45] huomioitiin yhdellä kommunikaatiolinkillä tapahtuva tiedonsiirto IoT-verkossa, jota testattiin sekä CoAP- että HTTP-protokollalla. Molempia protokollia käytettäessä viestien hyötykuorman syntaksi oli GraphQL-kyselykielen mukaista. Jotta kommunikaatio laitteiden ja pilvitietokannan kanssa onnistuisi, niin samaan pilvipalveluun oli julkaistu funktiosovellus vastaamaan GraphQL:n ajonaikaisesta järjestelmästä ja suorittamaan kyselyiden kääntämistä muiden järjestelmäkomponenttien välillä. IoT-laitteet pystyivät olemaan pidempään lepotilassa, koska tiedonsiirtoa saatiin tehostettua paremmalla kaistanleveyden hyödyntämisellä ja lyhentyneellä tiedonsiirtoajalla pyyntöjen lukumäärän vähentyessä.

GraphQL-kerros IoT-järjestelmässä helpottaa myös laitteiden yhteensovittamista keskenään ja muun järjestelmän kanssa. Yhteinen, hyvin määritelty GraphQL-skeema tyyppimäärittelyineen ja tarvittavine kyselyineen on osaltaan mahdollistamassa tämän, koska rajapintoihin ei tarvita uusia päätepisteitä eikä tiedonprosessointiin tai -hakuun tarvitse tehdä muutoksia palvelimella [66]. Artikkelissa [50] IoT-protokollilla kommunikoivat laitteet yhdistettiin yhdyskäytävän kautta API-rajapinnoista vastaavaan yhdyskäytävään, joka oli toteutettu GraphQL:llä. Usea palvelu oli käytettävissä GraphQL-rajapinnan kautta ja se toimi väylänä sekä IoT-laitteille että asiakassovelluksille palveluihin pääsemiseksi.

OPC UA on protokolla teollisuuden koneiden valvontaa ja hallintaa varten. Protokollaa ymmärtävän palvelimen kanssa tapahtuva kommunikaatio vaatii syvällistä perehtymistä OPC UA -arkkitehtuuriin ja kommunikaatio on riippuvainen ohjelmointikielestä ja käyttöympäristöstä. Hietala et al. [22] suunnittelivat toiminnasta vastaavan palvelimen rajapinnalle GraphQL-kerroksen, jotta nosturilaitteiston käyttäminen tehostuisi ja helpottuisi hallintasovelluksen kautta. Sovelluksen tuoman hyödyn lisäksi tutkimuksessa haluttiin nopeuttaa sovelluskehitystä Industry 4.0 ja Cyber-Physical Systemia varten hyödyntämällä web-kehityksessä tunnettua teknologiaa. Opinnäytetyössä [21] todettiin GraphQL:n tarjoavan samassa käyttöym-

päristössä helpon alustan mukautettujen palveluiden suunnitteluun, osaltaan siksi, että resurssit tulkitaan lähes vastaavasti OPC UA -arkkitehtuurin kanssa. GraphQL-rajapinta asennettiin Raspbian-käyttöjärjestelmää käyttävälle Raspberry Pi -tietokoneelle, joka toimi erillisenä palvelimena samassa verkossa OPC UA -palvelimen kanssa. Rajapinnan asentaminen kaikille alustoille nähtiin kuitenkin mahdolliseksi, kunhan ne täyttävät muut ohjelmistovaatimukset. Rajapinnan kautta pystyttiin lukemaan sensorien mittaamia arvoja ja lähettämään ohjauskäskyjä nosturilaitteille.

Käskynhallinnan helpottamisen lisäksi GraphQL on nähty mahdollisuutena automatisaation kasvattamiseen. Deksne et al. [13] hahmottelivat konseptiluonnoksen automatisoimaan ravintolatoimintaa. Sensorien, RFID-tagien, robottien ja muiden IoT-laitteiden käytön nähtiin parantavan varastonhallintaa, asiakkaiden pöytiinohjausta ja tilausten vastaanottamista. Konseptissa kuvailtiin GraphQL-kerroksen käyttämistä väliohjelmistona yhdistämään järjestelmän monipuolinen rakenne. Yhteisen rajapinnan lisäksi GraphQL koettiin merkittävänä tekijänä edistämään järjestelmän suorituskykyä ja palvelunlaatua. Erityisesti mobiiliverkkoympäristöissä pyyntöjen lukumäärän vähentymisen ennustettiin nopeuttavan tietoliikennettä.

Artikkelissa [64] puolestaan väitettiin, että aikaisemmat tutkimukset eivät ole onnistuneet täysin vastaamaan poikkeavuuksiin laitevalmistajien välillä olevissa standardeissa ja teknologia-arkkitehtuureissa. Laitteiden kesken vallitseva kommunikaatioero johtuu niiden resurssikapasiteettien poikkeavuuksista. Kaikki laitteet eivät tue esim. CoAP- tai MQTT-protokollia, mikä estää niiden kommunikaation joidenkin väliohjelmistojen kanssa. Yhdenmukainen pääsy estyy osaltaan myös laitteiden välillä vallitsevien poikkeavien ja keskenään yhteensopimattomien M2M-standardien käytöstä. Ratkaisuna artikkelissa selvitettiin väliohjelmiston käyttöä IoT-laitteiden ja -järjestelmien yhteensovittamiseksi yhteisen kommunikaatiohallintajärjestelmän mahdollistamiseksi. Tuloksena esitettiin GraphQL:n innoittama linkitettyyn dataan perustuva tietograafi, jota testattiin yhtenäistämällä maaseudulla sijaitsevan jäteveden käsittelylaitoksen etävalvontaa.

Vaikka GraphQL tulkitaan usein kokonaisarkkitehtuuriin vaikuttavana toiminnallisuutena, niin pelkän GraphQL-syntaksin omaksuminen on nähty varteenotettavana vaihtoehtona. Amazon Web Servicen [4] IoT Things Graph Data Model (TDM) on esitetty GraphQL-kielen mukaisella syntaksilla. Vaikka TDM-malli onkin yhteensopiva GraphQL-spesifikaation kanssa, niin se ei kuitenkaan käytä GraphQL:ää kyselykielenä tai ajonaikaisena järjestelmänä. GraphQL-spesifikaation mukaisella syntaksilla määritellään käytetty tyyppijärjestelmä ja muut tietorakenteet, jotta IoT-

järjestelmän kuvaaminen olisi käytännöllisempää ja havainnollisempaa. Syntaksi toimii keinona jäsentämään dataa eri laitteiden välillä, jolloin arkkitehtuuriltaan poikkeavat laitteet pystytään integroimaan samaan järjestelmään niiden kyetessä kommunikoimaan yhteisen kielen avulla.

Kerätyn aineiston perusteella vaikuttaa siltä, että GraphQL:n käyttö IoT-järjestelmissä on karkeasti jaettavissa kahteen osa-alueeseen: Ensimmäinen painottaa tiedonsiirron ja energiankulutuksen optimointia spesifikaation ominaisuuksilla tiedonvälitysoperaatioiden parantamiseksi. Toisessa tavoitellaan heterogeenisten järjestelmien ja laitearkkitehtuurien yhteensovittamista joko rajapinnan tai väliohjelmiston avulla. Vaikka edeltävissä tutkimuksissa esitetyt tulokset perustuivat suhteellisen pieniin järjestelmäkokonaisuuksiin tai konsepteihin, niin ne silti osoittivat, että GraphQL:n mahdollisella käyttöönotolla IoT-maailman millään sovellusalueella ei ole mainittavaa estettä. GraphQL:n käyttö vaatii ajonaikaisen järjestelmän tai poikkeavien syntaksien välillä tapahtuvaa jäsentelyä, mikä kasvattaa sekä laitteiston resurssivaatimuksia että energiankulutusta. Ymmärrettävästi useimmissa sovelluksissa ei ole nähty järkeväksi tai edes mahdolliseksi näiden ominaisuuksien tuomista IoT-laitteille, niiden rajoittaessa kyselykielellä tavoiteltuja hyötyjä. Joissain sovelluksissa GraphQL on saatettu sisällyttää laitteelle, mutta silloin kyseessä on jo usein Raspberry Pin kaltainen pienoistietokone, joka on tyypillisesti kytkettynä verkkovirtaan. Edellä kuvattujen seikkojen valossa GraphQL-kerros tai -jäsentely on toteutettu tavallisesti paikallisessa verkossa tai pilvipalvelussa sijaitsevalla palvelinkoneella tai funktiosovelluksella.

5.9 Puolesta ja vastaan

GraphQL on nähty kiinnostavana ratkaisuna muuttuvassa web-kehityksessä ja useiden suurten toimijoiden, kuten GitHubin ja IBM:n, antama tuki on tehnyt siitä Porcellon ja Banksin [40, s. 30] mielestä uskottavan ratkaisun muillekin palveluntarjoajille. Erityisesti mobiililaitteiden voimakas yleistyminen ja niiden erityistarpeiden, kuten rajoittuneemman laskentatehon ja verkkoyhteyksien vaihtuvuuden, huomioiminen on lähteen [10, s. 12] mukaan herättänyt tarpeen optimoida tiedonvälitystä verrattuna aikaisempiin web-palveluihin siirtämällä vain ja ainoastaan tarpeellista tietoa. Tapauskohtaista harkintaa ei tule kuitenkaan ohittaa ja kirjassa [10, s. 7] onkin arvioitu, että uuden GraphQL-palvelua käyttävän sovelluksen toteuttaminen on helpompaa kuin vanhan arkkitehtuurin konvertointi. Tutkimuksessa [61] puo-

lestaan väitetään, että merkittävin haaste siirryttäessä REST-palvelusta GraphQL-palveluun on hyvien piirteiden, kuten joustavuuden, skaalautuvuuden ja kestävyyden, säilyttäminen.

Tiedonvälityksen optimoinnista Porcello ja Banks [40, s. 24 – 28] tekevät olettamuksen, että asiakaspyynnöistä saadaan nopeampia siirrettävän tiedon määrän pienentyessä. Palveluiden nopeutta parantaa entisestään pyyntöjen lukumäärän vähentyminen haettaessa resurssissa viitattuja muita resursseja. GraphQL mahdollistaa viitattaviin resursseihin kuuluvan tiedon hakemisen samassa pyynnössä toisin kuin RESTful-arkkitehtuurissa, jossa jokaiselle resurssille tulisi suorittaa oma pyyntönsä. REST-rajapinnoissa on usein kierretty edeltävä ongelma kirjoittamalla palvelimelle ylimääräistä ohjelmalogiikkaa tapauksiin, joissa asiakas tarvitsee tietoa myös viitatuista resursseista, mutta GraphQL-rajapinnassa tämä on tarpeetonta.

Vaikka GraphQL on nähty ratkaisuna tiedonsiirron suorituskyvyn parantamiseksi, niin spesifikaatio ei kuitenkaan ole täysin ongelmaton. Joitakin yleisesti tiedostettuja ongelmia on pyritty ratkaisemaan vaihtelevilla lähestymistavoilla, sillä mikään yksittäinen keino ei ole toiminut aukottomasti kaikkien tapauksien käsittelyssä. Wittern et al. [63] keräsivät laajan kattauksen avoimia GraphQL-rajapintoja ja selvittivät niiden skeemojen rakennetta. Tutkimuksessaan he havaitsivat, että valtaosalla rajapinnoista huonoimman tapauksen vastauskoko saattaa kasvaa merkittävän suureksi, mikä voi muodostaa turvallisuusuhan rajapintojen altistuessa ylikuormittumiselle tai mahdollisesti jopa palvelunestohyökkäyksille. GraphQL suosittelee dokumentaatiossaan [55] sivutuksen käyttöönottoa, mutta Wittern et al. [63] huomasivat, että useimmissa rajapinnoissa on tarve hyväksi koettujen kehityskäytänteiden laajemmalle omaksunnalle.

Tutkimuksessa [20] puolestaan otettiin resurssi-intensiivisiin pyyntöihin hyvinkin formaali lähestymistapa. GraphQL-spesifikaation kuvaamalle tietomallille pyrittiin antamaan matemaattinen määritelmä tutkimuksen tavoitetta ajatellen. Suorituskykytarkastelu kohdistettiin GitHubin tarjoamaan GraphQL-rajapintaan, jossa pyyntöjen rakenteen syvyyttä on pyritty rajoittamaan. Tämän ja muiden rajoitteiden määrittely ei kuitenkaan ole osoittautunut riittäväksi suodattamaan pois liian kuormittavia pyyntöjä palvelimen suorituksesta. Erääksi keinoksi on ehdotettu pyyntöjen kompleksisuuden ja kustannusten arviointia ennen kuin ne hyväksytään suoritettavaksi. Ongelmaksi saattaa kuitenkin muodostua oikeiden arvioiden ja mittasuhteiden löytäminen. Hartig ja Pérez [20] toteavat, että kustannukset alittava pyyntö saatetaan hylätä ja ylittävä hyväksyä suoritettavaksi. He esittävät rat-

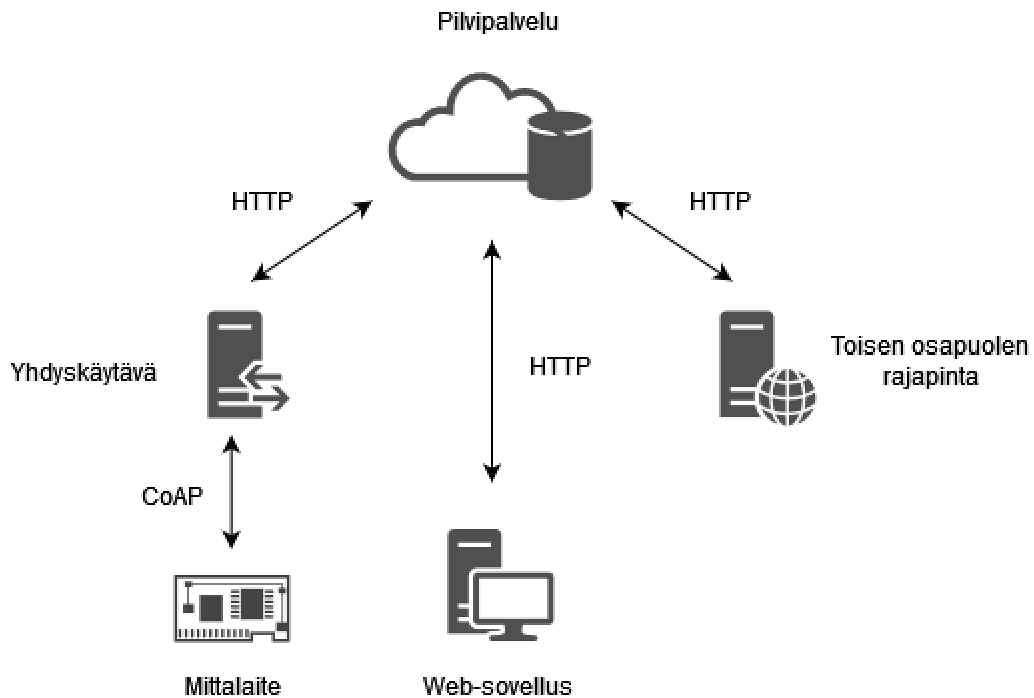
kaisunaan edellä mainittua syvyyden tai samaa linkkiä useasti kulkevien polkujen lukumäärän rajoittamista.

Palvelin- ja asiakassovellusten kehittämisestä ja niiden keskinäisestä suhteesta Buna [10, s. 12 – 13] nostaa esiin niiden riippumattomuuden toisistaan. Sovellusten toiminnalliset ratkaisut voivat olla hyvin löyhästi yhteydessä toisiinsa, jolloin modulaaristen ja ylläpidon kannalta joustavien järjestelmien rakentaminen helpottuu. Koska asiakas ja palvelin keskustelevat rakenteeltaan toisiaan muistuttavalla kielellä, niin ongelmien selvittäminen yksinkertaistuu. Epäsäännöllisyyksien havaitseminen on nopeampaa, mikä antaa arvokasta tietoa aloitettaessa virhetilanteiden korjausta. Vahva tyypitys paljastaa mahdolliset väärinkäytöt ja niistä raportoidaan jo hyvissä ajoin. Vahvan tyypityksen ja introspektiojärjestelmän yhteistyö ohjeistaa käyttäjiä ja antaa erityisesti kehittäjille tärkeää tietoa helposti ymmärrettävien yleisten ohjelmointitermien muodossa.

Vapaus, jonka abstraktilla tasolla kirjoitettu GraphQL-spesifikaatio ja riippumattomuus muista teknologioista antaa, saattaa myös rajoittaa niiden ominaisuuksien hyödyntämistä tai vaikeuttaa yhteensovittamista. Koska GraphQL painottaa tiedonhaussa enemmän palvelimella ajettavaa ohjelmakoodia, niin sen on nähty kasvattavan palvelinsovellusten kehittämisen monimutkaisuutta [48]. Toteutuksesta riippuen GraphQL voi vaatia REST-rajapintoihin verrattaessa eri lähestymistä rajapintojen hallintaan infrastruktuuria ja kustannuksia pohdittaessa.

6 IoT-järjestelmän demonstraatio

GraphQL:n käyttöä IoT-järjestelmässä demonstroidaan yksinkertaisen sovelluksen avulla (Kuva 6.1). Sovelluksessa näytetään sensorilaitteen varastotilasta mittaamia lämpötila- ja ilmankosteusarvoja sekä mahdollisimman reaaliaikaisia ulkoilmasta mitattuja vastaavia arvoja. Ulkoilmasta mitatut tulokset haetaan Ilmatieteenlaitoksen avoimesta rajapinnasta. Järjestelmäarkkitehtuuri muodostuu viidestä komponentista: mittalaitteesta, yhdyskäytävästä, pilvipalvelusta, web-sovelluksesta ja toisen osapuolen rajapinnasta. Mittalaite suorittaa mittauksia kymmenen minuutin välein, minkä jälkeen se lähettää datan pilvitietokantaan yhdyskäytävän kautta. Kommunikaatio mittalaitteen ja yhdyskäytävän välillä tapahtuu CoAP-protokollalla, mutta viestintä yhdyskäytävän ja pilvipalvelun kesken tapahtuu HTTP:n välityksellä. CoAPissa välitettävä hyötykuorma on GraphQL-syntaksin mukaista ja yhdyskäytävä toimii GraphQL-asiakkaana, joka lähettää kyselyt palvelimelle ajonaikaisen järjestelmän suoritettavaksi. Pilvipalveluun on määritelty tapahtumankuuntelija reagoimaan tietokantaan lisättäviin uusiin tiedostoihin. Tällöin tapahtumankuuntelija kutsuu samassa palvelussa julkaistua funktiosovellusta, joka hakee Ilmatieteenlaitoksen rajapinnasta viimeisen tunnin ajalta mitatut lämpötila- ja ilmankosteusarvot varastotilan sijaintikunnan mittauspisteeltä. Tämän jälkeen funktiosovellus jäsentelee rajapinnasta vastaanotetut mittaustulokset ja tallentaa ne tietokantaan aikaleimalla lisättynä.



Kuva 6.1: Kaavio suunnitellusta järjestelmästä.

Syntaksia lukuun ottamatta GraphQL:n ominaisuuksien tuomista mittalaitteelle ei nähty mielekkääksi resurssirajoitteiden takia. Jo luvussa 5.8 esitetyt tulokset tukevat väitettä, että resursseilta ja erityisesti tiedon prosessoinnilta vaadittavat lisäkustannukset eivät ole käytännöllisyyden kannalta järkeviä. Lisäksi tarvittavien ohjelmakirjastojen asentaminen mittalaitteelle varasi jo paljon käytettävissä olevasta muistikapasiteetista. Yhdyskäytävällä ei kuitenkaan ollut vastaavia rajoitteita. Sen suorituskyky on suurempi ja verkkovirtaan kytkettynä energiankulutuksen minimoiminen ei ole tavoitteellista järjestelmän elinajan kannalta. Järjestelmä on laajennettavissa uusia laitteita liittämällä eikä niille ole tunnistettavissa erityisiä rajoitteita. Koska kommunikaatio pilvipalvelun kanssa tapahtuu yhdyskäytävän kautta, niin laitteen yhteensovittamiseksi järjestelmään riittää, että se kykenee tiedonsiirtoon yhdyskäytävän kanssa. Joitain varastotilaan realistisia sensorilaitteita ovat mm. kulunvalvontaan suunnitellut laitteet ja liiketunnistinsensorit. Järjestelmään voidaan lisätä myös uusia sensoreita mittaamaan tilan fysikaalisia olosuhteita.

6.1 Mittalaite

Mittalaitteena toimii ESP32-mikrokontrolleri, joka on Espressif Systemsin kehittämä vähävirtainen ja monilla virrankulutukseen vaikuttavilla ominaisuuksilla varustettu laite. ESP32 on suunniteltu käytettäväksi erityisesti mobiililaitteita, puettavaa elektroniikkaa ja IoT-sovelluksia varten [16]. Mikrokontrolleri on varustettu kahdella mikroprosessorilla, joista päävastuun laskennassa ja komponenttien käskynhallinnassa ottaa Tensilican 32-bittinen Xtensa LX6 -mikroprosessori, mikä useimmissa malleissa sisältää kaksi ydintä [15]. Prosessorin kellotaajuus on parhaimmillaan 240 MHz ja käskyjen suorituksessa se saavuttaa arvon 600 DMIPS. Toisena mikroprosessorina toimii todella vähävirtainen prosessori, jonka tehtävänä on ylläpitää järjestelmätoimintoja lepotilan aikana. Käytettävissä olevat muistiyksiköt on jaettu tarkoituksensa mukaan eri komponentteihin. Käynnistyksestä ja ydintoiminnoista vastaa 448 kB ROM-muisti, tietoa ja käskyjä varten on varattu 520 kB SRAM-muistia ja sisäänrakennettu flash-muisti, joka on kooltaan enimmillään 4 MB. Langattomissa yhteyksissä mikrokontrollerilla on sisäänrakennettu tuki IEEE 802.11 -standardin mukaisia WiFi-yhteyksiä varten 2,4 GHz taajuusalueella sekä Bluetooth-tuki, joka kattaa neljännen sukupolven version 4.2 ja vähävirtaisen Bluetooth Low Energy (BLE) version. Yhteyksien suojaus- ja turvallisuusominaisuudet kattavat IEEE 802.11 -standardin määrittelemiä suojausvaatimuksia esimerkiksi WPA-salaukselle.

ESP32-mikrokontrollerille on ollut alun perin asennettuna valmistajan kehittämä laiteohjelmisto C-kielelle. Toteutettua järjestelmää varten sille asennettiin Micropython-laiteohjelmisto, joka on rajoittuneille laitteille suunniteltu Python-ohjelmointikielen kolmannen version toteutus [18]. Laiteohjelmisto on optimoitu sopimaan paremmin rajoittuneille laitteille jättämällä pois osa standardikirjaston moduuleista tilan säästämiseksi. Sen sijaan Micropythoniin on lisätty joitain erityisesti mikrokontrollereille tarkoitettuja ominaisuuksia. Lopputuloksena on saavutettu tehokas ja monipuolinen ohjelmointikirjasto, joka on käytettävissä ainoastaan 256 kB ohjelmamuistilla ja 16 kB RAM-muistilla, säilyttäen samalla mahdollisimman hyvä yhteensopivuus alkuperäisen Python 3 -version kanssa.

Micropythonin lisäksi mittalaitteelle tallennettiin tarvittavat tiedostot MicroCoAPy-kirjaston ja BME280-sensorin käyttämistä varten. MicroCoAPy on Micropythonia käyttäville laitteille kirjoitettu Python-ohjelmointikielen toteutus CoAP-protokollalle [26]. Mikrokontrollereille sovitettuna kirjastona kokonaisuus on pienempi ja yksinkertaisempi verrattaessa moniin muihin CoAPia varten suunniteltuihin kirjastoihin. Haittapuolena MicroCoAPy kuitenkin häviää monipuolisuudessa, mutta sen pieni

koko ja ominaisuudet koskien asiakas- ja palvelinsovelluksia sekä metodeja ja kommunikaationhallintaa tekevät siitä riittävän tämän työn tarkoitusta varten.

Lämpötilan ja ilmankosteuden mittaamiseen käytetään Boschin valmistamaa BME280-sensoria. Sensori on suunniteltu erityisesti mobiililaitteita ja puettavaa teknologiaa varten sovelluksissa, joissa komponentin koko ja pieni energiankulutus ovat suunnittelun kannalta oleellisia tekijöitä [9]. Lämpötilan ja ilmankosteuden lisäksi sensori kykenee myös ilmanpaineen mittaamiseen. Sensorin keskimääräinen virrankulutus lepotilassa on 0,1 µA ja pinta-alaltaan se kattaa 6,25 mm^2, korkeuden ollessa alle 1 mm. Koska BME280-sensorin käyttöä helpottava ohjelmakirjasto ei ole osa Micropythonin ESP32-versiota, niin myös se tulee asentaa mikrokontrollerille erillisenä ohjelmana[1].

6.2 Yhdyskäytävä

Yhdyskäytävänä käytetään pienikokoista Raspberry Pi -tietokonetta, jonka valmistajan mallisto kattaa kohtuuhintaisia ja tehokkaitta laitteita mikrokontrollereista tietokoneisiin ja edelleen niiden lisätarvikkeisiin ja komponentteihin [46]. Tietokoneisiin voidaan niille varatusta muistista riippuen asentaa haluttu käyttöjärjestelmä, joista osa sisältää laajan valikoiman valmiita ohjelmistoja ja kehitysympäristöjä. Toteutettuun IoT-järjestelmään valittiin kesäkuussa 2019 julkaistu Raspberry Pin malli 4 Model B [57]. Sille asennettu Debian Linuxiin pohjautuva Raspbian-käyttöliittymä sisältää monia hyötyohjelmia, kuten Pythonille tarkoitettuja ohjelmointiympäristöjä [35].

Teknisestä dokumentaatiosta [47] selviää, että Raspberry Pi 4 Model B sisältää hyvän suorituskyvyn omaavan 64-bittisen Cortex-A72 -neliydinprosessorin 1,5 GHz kellotaajuudella ja valinnasta riippuen 2, 4 tai 8 GB RAM-muistia. Langattomista yhteyksistä tuettuina ovat 2,4 ja 5,0 GHz IEEE 802.11 -standardin mukaiset lähiverkkoyhteydet, Bluetooth 5.0 ja Gigabit Ethernet. Langattomat lähiverkko- ja Bluetooth-yhteydet ovat yhteensopivuussertifioituja, mikä tekee laitteiston helpommin integroitavaksi laajempien järjestelmien kanssa testaustarvetta ja muita kustannuksia vähentämällä. Käyttöjärjestelmä ja käyttäjien tiedot voidaan tallentaa Micro SD -muistikortille, jolle on varattu oma paikkansa. Suorituskyvyltään laitteistoa voidaan verrata jopa joihinkin yleisiin pöytätietokonejärjestelmiin. Aikaisempiin mal-

[1] `https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/WiFi/HTTP_Client_IFTTT_BME280/BME280.py`

leihin verrattuna Raspberry Pi 4 Model B -mallia on kuvattu edistysaskeleeksi laskentanopeutta, multimedian käsittelyä, muistia ja yhteyksiä arvioitaessa. Kapasiteettien kehittymisestä huolimatta virrankulutus on onnistuttu säilyttämään lähes samana ja yhteensopivuus aikaisempien mallien kanssa luo jatkumon helpottamaan seuraavan sukupolven malliin siirtymistä.

Jotta Raspberry Pi kykenisi kommunikoimaan pilvipalvelussa sijaitsevan GraphQL-rajapinnan kanssa, niin sille asennetaan gql-ohjelmointikirjasto. Kirjasto on Python-ohjelmointikielellä kirjoitettu GraphQL-spesifikaation mukainen asiakassovellus [19]. Pythonille on kirjoitettu lukuisia GraphQL -palvelin- ja asiakaskirjastoja [54], mutta gql valittiin sen sisältämän tuen Pythonin kolmannelle versiolle, hyvän dokumentaation sekä aktiivisen kehitys- ja ylläpitotyön takia.

6.3 Pilvipalvelu

Tiedon tallentamiseen käytetään MongoDB Atlasia, mikä voidaan yhdistää pilvipalveluna toimivaan MongoDB Realm -sovellukseen, joka on käytettävissä GraphQL-rajapintana ja ajonaikaisena järjestelmänä. MongoDB Atlas on NoSQL-tietokantoihin kuuluva dokumenttitietokanta, missä tieto esitetään denormalisoimattomassa muodossa [33, s. 120 – 127]. Dokumenttitietokannat on suunniteltu osittain rakenteisen tiedon hallintaan, missä kokoelmiin tallennettavat objektit, dokumentit, ovat tiedon tallentamisen perusyksikköjä ja verrattavissa relaatiotietokantojen riveihin [58, s. 29 – 31]. Vaikka relaatiotietokannat ovat ajan saatossa vakiinnuttaneet asemansa merkittävänä sovelluskehityksen tiedonhallintaratkaisuna [33, s. 99 – 100], niin niiden vahvuus tiedon eheyden turvaamisessa on näyttäytynyt haasteena skaalautumisessa ja suurten web-sovellusten yleistyessä tiedonhallintaa laajemmassa mittakaavassa on ollut tarpeen tutkia ja kehittää [58, s. 7 – 11]. Lake ja Crowther [33, s. 99 – 100] pitävät NoSQL-tietokantoja skaalautuvuutensa ja suorituskykynsä puolesta parempana vaihtoehtona relaatiotietokannoille, sillä ne kykenevät nopeampiin tietokantaoperaatioihin luopumalla yhtä korkeasta tiedon eheyden varmistamisesta. Dokumenttitietokannat ovat osoittautuneet joustavaksi tiedonhallintaratkaisuksi, mitä Vaish [58, s. 29 – 31] pitää yhtenä syynä niiden suosiolle NoSQL-tietokantamallien joukossa. Tietokantakyselyt voidaan kohdistaa koko tietokantaan eikä pelkästään kokoelmiin, kuten relaatiotietokannoissa tauluihin. Osittain rakenteisella tiedolla ei ole relaatiotietokantoihin verrattaessa yhtä tarkkaan määrättyä rakennetta ja se voi muuttua ajan saatossa, mikä on koettu hyödylliseksi web-sovelluksissa.

MongoDB Realm on kehitysalusta datalähtöisten sovellusten kehittämiseksi [38], jotka ovat helposti yhdistettävissä MongoDB Atlaksessa ylläpidettyyn tietokantaan. Realmin sovellusvalikoimaan lukeutuu monia valmiita sovelluskehyksiä mm. tapahtumien seurauksena ajettavia funktiosovelluksia ja HTTP-osoitteina julkaistavia Webhook-palveluita. Tarjotuista palveluista otetaan käyttöön GraphQL-rajapinta, joka ajonaikaisena järjestelmänä mahdollistaa asiakassovelluksissa käytettävien GraphQL-kyselyjen kääntämisen ja suorittamisen vasten Atlaksessa tallennettua tietoa. Kun GraphQL-rajapinta on yhdistetty Atlakseen, niin se generoi GraphQL-spesifikaation mukaisen skeeman tallennetuista kokoelmista. Samalla sovellus luo käytettäväksi monipuolisen kattauksen valmiita GraphQL-kyselyitä, joiden avulla voidaan suorittaa tiedon hakua ja muokkausta. Mittalaitteen suorittamat mittaukset tallennettaan kokoelmaan "Measurements", jonka dokumenteista sovellus kehitti kuvan 6.2 tyypin. Varastotilasta mitattiin myös ilmanpaine, joka tallennettiin muiden suureiden yhteydessä tietokantaan. Tämän tarkoituksena on havainnollistaa web-sovelluksessa vain tarpeellisen tiedon pyytämistä GraphQL-rajapinnasta.

```
type Measurement {
  _id: ObjectId
  measurementTime: String
  temperature: Float
  humidity: Float
  pressure: Float
}
```

Kuva 6.2: Realm-sovelluksen kehittämä Measurement-tyyppi.

Ilmatieteenlaitoksen avoimen datan rajapinta on toteutettu OGC Web Feature Service 2.0 (WFS) rajapinnan avulla [25], joten se ei ole suoraan asiakkaiden käytettävissä GraphQL-kyselyinä. Realmin funktiosovellus hakee HTTP-kyselyllä dataa rajapinnasta jäsenneltäväksi ja tallennettavaksi Atlas-tietokantaan tehtyyn toiseen kokoelmaan "External_Measurements". Myös näistä dokumenteista generoitiin val-

mis tyyppi (Kuva 6.3).

```
type External_Measurement {
    _id: ObjectId
    time: String
    temperature: Float
    humidity: Float
}
```

Kuva 6.3: Realm-sovelluksen kehittämä External_Measurement-tyyppi.

Tyyppeihin perustuen Realm-sovellus loi valmiit kyselyt tiedonhakemiseksi ja muokkaamiseksi. Esimerkiksi mittalaite pystyi tallentamaan dataa ”Measurements” kokoelmaan seuraavalla kyselyllä, missä data objektissa on esimerkinomaisesti tallennettavat avain-arvo-parit.

```
mutation {
    insertOneMeasurement(data: {
        humidity: 10,
        pressure: 943,
        temperature: 21,
        measurementTime: "2022-03-01T18:40:43.501Z"}) {
            _id
    }
}
```

Kyselyille kehitettiin myös valinnaisesti käytettävät argumentit ja tyyppien jokaista kenttää varten hakua suodattavat parametrit. Suodatus voidaan tehdä, joko täsmällisin ehdoin tai esimerkiksi suurempi kuin ja pienempi kuin -relaatioin. Lisäksi haun palauttamien dokumenttien lukumäärää voitiin rajoittaa.

6.4 Web-sovellus

Mittaustulosten näyttämistä varten kehitettiin web-sovellus, joka on kirjoitettu JavaScriptillä ja Node.js:llä. Käyttöliittymässä näytettävä tieto haetaan MongoDB Realmin GraphQL-rajapinnasta ja kyselyiden tekemiseksi käytetään JavaScriptillä kirjoitettua ohjelmakirjastoa GraphQL-asiakassovellukselle. Ohjelmakirjastoksi valittiin graphql-request. Monet JavaScriptillä kirjoitetut GraphQL-asiakassovellukset on kehitetty jotain tiettyä sovelluskehystä varten, mutta graphql-request on yleisemmin Node.js:ää varten kirjoitettu ja pienestä koostaan huolimatta sisältää kaiken tässä sovelluksessa tarvittavan [42].

Realm-sovelluksena julkaistu GraphQL-rajapinta abstrahoi eri datalähteet yhteen rajapintaan. Web-sovelluksena toimivan asiakkaan ei tarvitse olla lainkaan tietoinen datalähteiden alkuperäisestä tarjoajasta. Sekä IoT-laitteella mitatut että WFS-rajapinnasta haetut tulokset ovat kaikki käytettävissä GraphQL-rajapinnan kautta. Web-sovelluksessa hyödynnettiin tietokannan kokoelmista kehitettyjä tiedonhakukyselyitä "measurements" ja "external_Measurements". Molemmat kyselyt voitiin lähettää kerralla GraphQL-spesifikaation ominaisuuden ansiosta, jossa yhdessä pyynnössä voidaan esittää useita kyselyitä (Kuva 6.4). Kyselyissä haettiin dataa ajan mukaan laskevassa järjestyksessä ja ainoastaan viisi viimeisintä mittaustulosta.

```
query {
  measurements(limit: 5, sortBy: MEASUREMENTTIME_DESC) {
    measurementTime
    temperature
    humidity
  }
  external_Measurements(limit: 5, sortBy: TIME_DESC) {
    temperature
    humidity
  }
}
```

Kuva 6.4: Web-sovelluksessa käytetyt tiedonhakukyselyt.

Vaikka mittalaite tallensi myös mitatun ilmanpaineen, niin sitä ei kuitenkaan pyydetty käyttöliittymässä, koska se ei ollut tarpeellinen tieto. Vastaavasti toimittiin ulkoisesta rajapinnasta haettujen tulosten aikaleimoissa. Web-sovelluksessa riitti, että ulkoa mitatut tulokset ovat viimeisimmät mahdolliset verrattaessa mittalaitteella suoritettuihin. Kuvassa 6.5 on tiedonhakukyselyyn vastaanotettu data ja kuvassa 6.6 web-sovelluksen käyttöliittymä, jossa data on esitettynä taulukoissa.

```
{
  "data": {
    "external_Measurements": [
      {
        "humidity": 79,
        "temperature": 1.3
      },
      {
        "humidity": 81,
        "temperature": 0.9
      },
      {
        "humidity": 84,
        "temperature": 0.4
      },
      {
        "humidity": 86,
        "temperature": 0
      },
      {
        "humidity": 88,
        "temperature": 0.3
      }
    ],
    "measurements": [
      {
        "humidity": 13,
        "measurementTime": "2022-03-12T10:22:43.511Z",
        "temperature": 22
      },
      {
        "humidity": 13,
        "measurementTime": "2022-03-12T10:12:43.541Z",
        "temperature": 21
      },
      {
        "humidity": 12,
        "measurementTime": "2022-03-12T10:02:43.491Z",
        "temperature": 21
      },
      {
        "humidity": 12,
        "measurementTime": "2022-03-12T09:52:43.501Z",
        "temperature": 22
      },
      {
        "humidity": 11,
        "measurementTime": "2022-03-12T09:42:43.501Z",
        "temperature": 21
      }
    ]
  }
}
```


Kuva 6.5: Web-sovelluksen vastaanottama data JSON-tietomuodossa.

Varasto #123

Ulkomittauspiste: Santahaka, Kokkola

Lämpötila

Aika	Varastossa	Ulkona
12.03.2022 10:22	22 °C	1.3 °C
12.03.2022 10:12	21 °C	0.9 °C
12.03.2022 10:02	21 °C	0.4 °C
12.03.2022 09:52	22 °C	0 °C
12.03.2022 09:42	21 °C	0.3 °C

Ilmankosteus

Aika	Varastossa	Ulkona
12.03.2022 10:22	13 %	79 %
12.03.2022 10:12	13 %	81 %
12.03.2022 10:02	12 %	84 %
12.03.2022 09:52	12 %	86 %
12.03.2022 09:42	11 %	88 %

Kuva 6.6: Web-sovelluksen käyttöliittymä.

7 Pohdinta

Standardointityö ja siihen käytetty aika on todettu olevan merkittävä tekijä uusien teknologioiden hyväksynnässä ja vaikuttamassa niiden levittäytymiseen osana omaksuttuja ratkaisumalleja. Vaikka GraphQL saikin alkunsa yksityisen omistajan hallitsemana ideana, niin se on siirtynyt avoimen yhteisön jatkokehitykseen ja vapaina dokumentteina julkaistavaksi spesifikaatioksi. Peittelemätön työskentely spesifikaation edistämiseksi tukee GraphQL:n ottamista mukaan osaksi standardointiorganisaatioiden keskusteluja.

Tavallisesti GraphQL:ää on käytetty yhdessä HTTP:n kanssa, mutta siitä ei ole tunnistettavissa mitään, mikä tekisi siitä sopimattoman muille protokollille. Monissa esineiden internetiä käsittelevissä tutkimuksissa GraphQL on sovitettu toimimaan CoAPin kanssa, mikä oletettavasti seuraa CoAPin ja HTTP:n välillä vallitsevista yhtäläisyyksistä. Kyselykielen syntaksi on kuitenkin eittämättä käytettävissä myös esimerkiksi MQTT:n kanssa. Laajentamista muihin IoT-protokolliin on syytä selvittää myös GraphQL:n tukemien viestintäarkkitehtuurien johdosta. Asiakas-palvelin-arkkitehtuuri on jo todettu toimivaksi, mutta publish-subscribe- ja havainnointiarkkitehtuurien käyttö on vielä jokseenkin selvittämätöntä.

GraphQL on nähtävissä osana palvelukeskeistä arkkitehtuuria ja täyttävän monia sen toivotuista piirteistä. Palvelukeskeisten arkkitehtuurien, kuten RESTin, tuominen esineiden internetiin on näytetty tehokkaaksi ratkaisuksi järjestelmien organisoimiseen jo aikaisemmin. GraphQL tuskin tuottaa poikkeusta tähän historiaan. Lisäksi GraphQL:n menneisyys jakaa RESTin kanssa samoja vaiheita teknologian omaksunnassa. Molemmat ovat kehittyneet pitkälti web-sovelluksien käytössä ennen kuin niiden tuomista esineiden internetiin on pohdittu. Myös GraphQL:n tilanteessa on hyvä mahdollisuus hyödyntää jo aikaisemmin opittua sekä GraphQL:stä itsestään että myös web-teknologioiden siirtämisessä laiteympäristöstä toiseen. Poikkeuksena aikaisempiin web-palveluihin GraphQL on teoriassa täysin riippumaton ympäristöstään, mikä tekisi siitä yhteensopivan kaikkien nykyisten ja tulevien teknologioiden ja järjestelmien kanssa.

GraphQL:n käyttämä kyselykieli on ainoastaan merkintäsyntaksi ja siten riippumaton laitteista tai tietomuodoista. Jo pelkästään spesifikaatiossa kuvatulla kysely-

kielellä on edellytykset yhtenäistää järjestelmäkomponenttien toimintaa. Amazonin IoT Things Graph Data Model on oiva esimerkki, kuinka kyselykieli ja sille tehtävä jäsentely riittää heterogeenisen laitejoukon yhdistämiseksi keskenään ilman ajonaikaista järjestelmää. GraphQL-palvelun toteuttaminen kokonaisuudessaan vaatii ajonaikaisen järjestelmän käyttöä, mikä voidaan sijoittaa paikallisen verkon palvelimelle, mutta valmiista pilvipalveluista ja pilvessä ajettavista funktiosovelluksista on kerrytetty jo hyviä kokemuksia. Väliohjelmistot ja yhdyskäytävät on havaittu myös tärkeäksi osaksi järjestelmää abstrahoimaan palveluita ja laitteita yhden rajapinnan taakse. Konseptuaalisia ideoita jo hyvin älykkäistä ohjelmistoista on esitetty, kuten ympäristöön ja verkko-olosuhteisiin mukautuva palveluarkkitehtuuri.

Esineiden internetin kannalta ei voida sivuuttaa energiansäästössä annettuja lupauksia. On kuitenkin tarpeen kriittisesti tarkastella GraphQL:n vaikutusta kokonaisvirrankulutukseen. Tutkimukset ovat antaneet todisteita pienemmästä virrankulutuksesta, mutta niissä käytetyt testiasetelmat ovat olleet mittaluokaltaan suhteellisen pieniä. Kuinka tämä heijastuu yleisesti esineiden internetissä kuvattuihin jopa tuhansien laitteiden järjestelmiin? Entä onko tiedonsiirrossa tavoiteltu pyyntöjen vähentäminen ja optimointi kuitenkaan edullista, jos asiakkaat voivat pyytää siirrettäväksi poikkeuksellisenkin suuria tietomääriä kerralla kyselykielen objektiviittausten avulla? Virrankäytössä tulee huomioida myös palvelimelle siirtyvä vastuu asiakkaiden toiveiden täyttämisestä. Rajan selvittäminen järjestelmän mittaluokan ja palvelimen liiallisen kuormittumisen välillä on tarpeen tutkia. Ajonaikaisen järjestelmän lisäämän virrankulutuksen arviointi voisi antaa suuntaa sen sijoittamisen kannattavuudesta akku- tai paristokäyttöiselle laitteelle ja ennusteita laitteen eliniälle. Samalla on aiheellista selvittää ajonaikaisen järjestelmän vaatimukset laskennalliselle suorituskyvylle ja muistille sekä verrata sitä IoT-laitteiden kapasiteetteihin.

Tässä tutkielmassa toteutettu IoT-järjestelmä havainnollistaa erästä tapaa GraphQL:n käytölle esineiden internetissä. Järjestelmä osoittaa, että jo hyvin lyhyessä ajassa voidaan kehittää yksinkertainen GraphQL:ää käyttävä IoT-sovellus hyödyntämällä olemassa olevia palveluita ja ohjelmakirjastoja. Web-sovelluksessa pystyttiin käyttämään joitakin GraphQL:n ominaisuuksia, joita käytiin läpi aikaisemmissa luvuissa. Asiakassovellus voi pyytää rajapinnasta vain tarvitsemansa tiedon välttyen käyttöliittymäsovelluksessa tehtävältä suodatukselta ja tiedon muokkaukselta. Samalla vältyttiin kahdelta pyynnöltä eri palveluihin kun kaikki data oli saatavilla yhdestä rajapinnasta. WFS-rajapinnan käyttäminen web-sovelluksessa helpottui, kun

GraphQL-rajapinta abstrahoi sieltä saatavan tiedon yhtenäisen rajapinnan taakse IoT-mittalaitteen kanssa. Tämä kuitenkin lisäsi palvelimella, eli pilvipalvelussa, tehtävää työtä tiedon hakemisessa, jäsentelyssä ja julkaisemisessa.

8 Yhteenveto

Esineiden internetin kasvaminen on tuonut mukanaan odotettuja haasteita, joita ei voida sivuuttaa. Tietoliikenteen vilkastuminen asettaa tarpeen optimoida jo olemassa olevia malleja ja reagoida skaalautuvilla arkkitehtuureilla. Jo aikaisemmat teknologian edistysaskeleet ovat osoittaneet hyvien standardien, spesifikaatioiden ja avoimuuden tärkeyden päätettäessä tulevaisuuden suunnitelmista.

Ongelman lähestyminen palvelukeskeisen arkkitehtuurin suunnasta on havaittu hyväksi tavaksi suurienkin järjestelmien muodostamiseksi. Palvelukeskeiseen arkkitehtuuriin kuuluu useita referenssimalleja, jotka kuvailevat skaalautuvan ja hajautetun järjestelmän, missä vastuu on jaettu itsenäisesti kehittyville komponenteille. Komponenttien välistä yhteensopivuutta parannetaan rajapintojen ja väliohjelmistojen avulla. Monet tunnetut malli ovat kehittyneet ja todettu käytännössäkin toimiviksi web-palveluina ja niitä on onnistuneesti siirretty esineiden internetiin. Erityisesti REST-arkkitehtuuri on web-palveluna keräämänsä suosion jälkeen omaksuttu myös IoT-järjestelmissä.

Vastaavasti kuten edeltäjänsä, GraphQL on web-palveluna kypsynyt arkkitehtuuri, jonka monet ominaisuudet ovat ihanteellisia esineiden internetiä ajatellen. Tiedonsiirron ja virrankulutuksen pienentäminen lukumäärältään vähemmillä lähetyksillä sekä alustariippumaton yhteensopivuus herättävät lupauksia paremmista IoT-järjestelmistä. Erilaisten viestintämallien tuki ei rajoita GraphQL:ää ainoastaan HTTP:n kaltaisen CoAPin kanssa toimivaksi, vaan sallii myös MQTT:n ja muiden publish-subscribe-protokollien käyttämisen.

GraphQL ja sen ajonaikainen järjestelmä vaativat kuitenkin ylimääräisiä resursseja joltain järjestelmän komponentilta. GraphQL:ää ja esineiden internetiä käsittelevät tutkimukset tuntuvatkin vahvasti painottuneet kyselykielen ja yhdyskäytävien käyttöön, siirtäen laskennallisen vastuun tehokkaammille laitteille. GraphQL:n siirtäminen IoT-laitteisiin asiakas- ja palvelinsovelluksina vaatii lisää järjestelmällistä ja tarkkaa tutkimusta verkon koon ja laitteiden tehon riippuvuudesta.

Lähteet

[1] AL-FUQAHA, A., GUIZANI, M., MOHAMMADI, M., ALEDHARI, M., JA AYYASH, M. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys Tutorials 17*, 4 (2015), 2347–2376.

[2] ALAM-NAYLOR, S. An introduction to GraphQL and how to use GraphQL APIs. URL `https://www.contentful.com/blog/2021/12/13/what-is-graphql/`, viitattu 18.2.2022.

[3] ALSHININA, R., JA ELLEITHY, K. Performance and Challenges of Service-Oriented Architecture for Wireless Sensor Networks. *Sensors 17*, 3 (2017).

[4] AMAZON WEB SERVICES, INC. What is the AWS IoT Things Graph Data Model? URL `https://docs.aws.amazon.com/thingsgraph/latest/ug/iot-tg-whatis-tdm.html`, viitattu 18.2.2022.

[5] APOLLO GRAPH, INC. Why adopt GraphQL? URL `https://www.apollographql.com/docs/intro/benefits/`, viitattu 4.2.2022.

[6] ASHTON, K., ET AL. That "Internet of Things" Thing. *RFID journal 22*, 7 (2009), 97–114.

[7] ATZORI, L., IERA, A., JA MORABITO, G. The Internet of Things: A survey. *Computer Networks 54*, 15 (2010), 2787–2805.

[8] BORMANN, C., CASTELLANI, A. P., JA SHELBY, Z. CoAP: An Application Protocol for Billions of Tiny Internet Nodes. *IEEE Internet Computing 16*, 2 (2012), 62–67.

[9] BOSCH SENSORTEC. BME280. URL `https://www.bosch-sensortec.com/products/environmental-sensors/humidity-sensors-bme280/`, viitattu 18.1.2022.

[10] BUNA, S. *Learning GraphQL and Relay*. Community Experience Distilled. Packt Publishing, 2016.

[11] BYRON, L. GraphQL: A data query language. URL https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/, viitattu 12.2.2022.

[12] CHEN, Y., JA KUNZ, T. Performance evaluation of IoT protocols under a constrained wireless access network. Julkaisusarjassa *2016 International Conference on Selected Topics in Mobile Wireless Networking (MoWNeT)* (2016), 1–7.

[13] DEKSNE, L., KEMPELIS, A., SNIEDZINS, T., JA KOZLOVSKIS, A. Automated System for Restaurant Services. *Information Technology and Management Science 24*, 1 (2021), 15–25.

[14] EKENE, P. What is GraphQL. URL https://www.digitalocean.com/community/conceptual_articles/what-is-graphql, viitattu 18.2.2022.

[15] ESP32.NET. The Internet of Things with ESP32. URL http://esp32.net/, viitattu 18.1.2022.

[16] ESPRESSIF SYSTEMS. ESP32. URL https://www.espressif.com/en/products/socs/esp32, viitattu 18.1.2022.

[17] FIELDING, R. T., JA RESCHKE, J. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC-7231, Kesäkuu 2014.

[18] GEORGE ROBOTICS LIMITED. MicroPython. URL https://micropython.org/, viitattu 18.1.2022.

[19] GRAPHENE PYTHON. GQL. URL https://github.com/graphql-python/gql, viitattu 18.1.2022.

[20] HARTIG, O., JA PÉREZ, J. Semantics and Complexity of GraphQL. Julkaisusarjassa *Proceedings of the 2018 World Wide Web Conference* (Republic and Canton of Geneva, CHE, 2018), WWW '18, International World Wide Web Conferences Steering Committee, 11551164.

[21] HIETALA, J. Real-time two-way data transfer with a digital twin via web interface. Master's thesis, Aalto University. School of Engineering, 2020.

[22] HIETALA, J., ALA-LAURINAHO, R., AUTIOSALO, J., JA LAAKI, H. GraphQL Interface for OPC UA. Julkaisusarjassa *2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS)* (2020), vol. 1, 149–155.

[23] IBRAHIM, N., JA BENCH, B. Service-Oriented Architecture for the Internet of Things. Julkaisusarjassa *2017 International Conference on Computational Science and Computational Intelligence (CSCI)* (2017), 1004–1009.

[24] IGLESIAS-URKIA, M., ORIVE, A., JA URBIETA, A. Analysis of CoAP Implementations for Industrial Internet of Things: A Survey. *Procedia Computer Science 109* (2017), 188–195. 8th International Conference on Ambient Systems, Networks and Technologies, ANT-2017 and the 7th International Conference on Sustainable Energy Information Technology, SEIT 2017, 16-19 May 2017, Madeira, Portugal.

[25] ILMATIETEEN LAITOS. Ilmatieteen laitoksen avoin data - Pikaohje. URL `https://www.ilmatieteenlaitos.fi/avoin-data-pikaohje`, viitattu 11.3.2022.

[26] INSIGH.IO. microCoAPy. URL `https://github.com/insighio/microCoAPy`, viitattu 18.1.2022.

[27] INTERNET ENGINEERING TASK FORCE. Constrained RESTful Environments, Charter. URL `https://datatracker.ietf.org/doc/charter-ietf-core/`, viitattu 27.11.2021.

[28] INTERNET ENGINEERING TASK FORCE. Constrained RESTful Environments (core), Documents. URL `https://datatracker.ietf.org/wg/core/documents/`, viitattu 27.11.2021.

[29] INTERNET ENGINEERING TASK FORCE. IPv6 over networks of resource-constrained nodes, Charter. URL `https://datatracker.ietf.org/doc/charter-ietf-6lo/`, viitattu 3.3.2022.

[30] KHAN, R., JA NOOR MIAN, A. Sustainable IoT Sensing Applications Development through GraphQL-Based Abstraction Layer. *Electronics 9*, 4 (2020).

[31] KUROSE, J. F., JA ROSS, K. W. *Computer Networking: A Top-Down Approach (6th Edition)*, 6th ed. Pearson, 2012.

[32] KYUSAKOV, R., ELIASSON, J., DELSING, J., VAN DEVENTER, J., JA GUSTAFSSON, J. Integration of wireless sensor and actuator nodes with IT infrastructure using service-oriented architecture. *IEEE Transactions on industrial informatics 9*, 1 (2012), 43–51.

[33] LAKE, P., JA CROWTHER, P. NoSQL Databases. Kirjassa *Concise Guide to Databases*. Springer, 2013, ss. 97–134.

[34] LI, S., XU, L. D., JA ZHAO, S. The internet of things: a survey. *Information Systems Frontiers 17* (2015), 243–259.

[35] LONG, S. Buster the new version of Raspbian. URL https://www.raspberrypi.com/news/buster-the-new-version-of-raspbian/, viitattu 18.1.2022.

[36] MACKENZIE, C. M., LASKEY, K., MCCABE, F., BROWN, P. F., METZ, R., JA HAMILTON, B. A. Reference model for service oriented architecture 1.0. *OASIS standard 12*, S 18 (2006).

[37] MALDONADO, L. Why GraphQL is the future of APIs. URL https://www.freecodecamp.org/news/why-graphql-is-the-future-of-apis-6a900fb0bc81/, viitattu 4.2.2022.

[38] MONGODB, INC. MongoDB Realm documentation. URL https://docs.mongodb.com/realm/, viitattu 16.1.2022.

[39] NAIK, N. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. Julkaisusarjassa *2017 IEEE International Systems Engineering Symposium (ISSE)* (2017), 1–7.

[40] PORCELLO, E., JA BANKS, A. *Learning GraphQL: Declarative Data Fetching for Modern Web Apps*, 1st ed. O'Reilly Media, Inc., 2018.

[41] PRISMA. The Fullstack Tutorial for GraphQL. URL https://www.howtographql.com/, viitattu 18.2.2022.

[42] PRISMA LABS. graphql-request. URL https://github.com/prisma-labs/graphql-request, viitattu 2.3.2022.

[43] RAHMAN, R. A., JA SHAH, B. Security analysis of IoT protocols: A focus in CoAP. Julkaisusarjassa *2016 3rd MEC International Conference on Big Data and Smart City (ICBDSC)* (2016), 1–7.

[44] RAJ, P., JA RAMAN, A. C. *The Internet of Things: Enabling Technologies, Platforms, and Use Cases*, 1st ed. Auerbach Publications, USA, 2017.

[45] Rasool, S., Khan, R., ja Mian, A. N. GraphQL and DC-WSN-Based Cloud of Things. *IT Professional 21*, 1 (2019), 59–66.

[46] Raspberry Pi Foundation. About us. URL `https://www.raspberrypi.org/about/`, viitattu 18.1.2022.

[47] Raspberry Pi Trading Ltd. Raspberry Pi 4 Computer Model B. URL `https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-product-brief.pdf`, viitattu 18.1.2022.

[48] Red Hat, Inc. What is GraphQL? URL `https://www.redhat.com/en/topics/api/what-is-graphql`, viitattu 18.2.2022.

[49] Schmidt, G., ja DeBergalis, M. Principled GraphQL. URL `https://principledgraphql.com/`, viitattu 4.2.2022.

[50] Seda, P., Masek, P., Sedova, J., Seda, M., Krejci, J., ja Hosek, J. Efficient Architecture Design for Software as a Service in Cloud Environments. Julkaisusarjassa *2018 10th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)* (2018), 1–6.

[51] Shelby, Z., Hartke, K., ja Bormann, C. *The Constrained Application Protocol (CoAP)*. RFC-7252, Kesäkuu 2014.

[52] Stemmler, K. What is GraphQL? GraphQL introduction. URL `https://www.apollographql.com/blog/graphql/basics/what-is-graphql-introduction/`, viitattu 4.2.2022.

[53] Thangavel, D., Ma, X., Valera, A., Tan, H.-X., ja Tan, C. K.-Y. Performance evaluation of MQTT and CoAP via a common middleware. Julkaisusarjassa *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)* (2014), 1–6.

[54] The GraphQL Foundation. Code using GraphQL. URL `https://graphql.org/code/`, viitattu 18.1.2022.

[55] The GraphQL Foundation. Learn. URL `https://graphql.org/`, viitattu 6.10.2021.

[56] The GraphQL Foundation. *GraphQL June 2018 Edition*, 2018.

[57] UPTON, E. Raspberry Pi 4 on sale now from $35. URL https://www.raspberrypi.com/news/raspberry-pi-4-on-sale-now-from-35/, viitattu 18.1.2022.

[58] VAISH, G. *Getting Started with NoSQL: Your Guide to the World and Technology of NoSQL*. Community experience distilled. Packt Publishing, 2013.

[59] VASSEUR, J.-P., JA DUNKELS, A. *Interconnecting Smart Objects with IP: The Next Internet*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[60] VILLAVERDE, B. C., PESCH, D., DE PAZ ALBEROLA, R., FEDOR, S., JA BOUBEKEUR, M. Constrained Application Protocol for Low Power Embedded Networks: A Survey. Julkaisusarjassa *2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing* (2012), 702–707.

[61] VOGEL, M., WEBER, S., JA ZIRPINS, C. Experiences on migrating RESTful web services to GraphQL. Julkaisusarjassa *International Conference on Service-Oriented Computing* (2017), Springer, 283–295.

[62] WEIR, L. *Enterprise API Management: Design and deliver valuable business APIs*. Packt Publishing Ltd, 2019.

[63] WITTERN, E., CHA, A., DAVIS, J. C., BAUDART, G., JA MANDEL, L. An empirical study of GraphQL schemas. Julkaisusarjassa *International Conference on Service-Oriented Computing* (2019), Springer, 3–19.

[64] XIE, C., YU, B., ZENG, Z., YANG, Y., JA LIU, Q. Multilayer internet-of-things middleware based on knowledge graph. *IEEE Internet of Things Journal 8*, 4 (2020), 2635–2648.

[65] XU, L. D., HE, W., JA LI, S. Internet of Things in Industries: A Survey. *IEEE Transactions on Industrial Informatics 10*, 4 (2014), 2233–2243.

[66] YUSUF, S. Better IoT with GraphQL and AppSync. URL https://blog.thundra.io/better-iot-with-graphql-and-appsync, viitattu 18.2.2022.