Oheislaitteiden liittäminen punnituspalveluun

LAB-ammattikorkeakoulu
Insinööri (AMK), Tieto- ja viestintätekniikka
2024
Eetu Leppinen

Tiivistelmä

Tekijä(t)	Julkaisun laji	Valmistumisaika
Eetu Leppinen	Opinnäytetyö, AMK	2024
	Sivumäärä 64	

Työn nimi

Oheislaitteiden liittäminen punnituspalveluun

Tutkinto ja koulutusala

Insinööri (AMK), tieto- ja viestintätekniikan koulutus

Toimeksiantajaorganisaatio (jos opinnäytetyöllä on toimeksiantaja)

Tamtron Precision Oy

Tiivistelmä

Oheislaitteiden liittäminen punnituspalveluun vie kehittäjiltä paljon resursseja laitteiden eroavaisuuksien takia. Opinnäytetyön tavoitteena oli rakentaa POC-versio oheislaitteiden liittämisestä punnituspalveluun käyttämällä low-code työkalua. Työn toimeksi antajana toimii Tamtron Precision Oy.

Opinnäytetyössä rakennettiin POC-versio oheislaitteiden liittämisestä punnituspalveluun käyttämällä työkalua Node-RED. Liitettyjä oheislaitteita ovat liikennevalot, kulunohjauslaitteet ja lisänäytöt. Node-RED käynnistyy Docker-kontin sisällä, joka lähetetään laitteelle AWS Greengrass -palvelun avulla. Node-RED virrat hyödynsivät kommunikaatiossa IPC-protokolla solmuja ja laitteiden ohjaamisessa I2C-protokollaa sekä ASCII-tiedonsiirto standardi solmuja. Liikennevaloja ja kulunohjauslaitteita ohjattiin digitaalilähtö solmuilla, jotka käyttävät I2C-protokollaa ja lisänäytöille lähetettiin painolukema ASCII-muodossa sarjaliikennesolmun avulla.

Node-RED asennettiin kahdelle asiakkaalle Suomessa. Oheislaitteet ja Node-RED sovellus ovat toimineet virheettömästi.

Asiasanat

Node-RED, AWS, Greengrass, Docker, I2C, ASCII

Abstract

Author(s)

Eetu Leppinen

Type of Publication

Thesis, UAS

Number of Pages

64

Published

2024

Title of Publication

Connecting peripheral devices to the weighing service

Degree, Field of Study

Engineer (UAS), Information and communications technologies

Organisation of the client (if the thesis work is commissioned by another party)

Tamtron Precision Oy

Abstract

Connecting peripheral devices to a weighing service takes a lot of resources from developers due to device differences. The aim of the thesis was to build a POC version of connecting peripheral devices to a weighing service using a low-code tool. The client of the work is Tamtron Precision Oy.

In the thesis, a POC version of connecting peripheral devices to the weighing service was built using the tool Node-RED. Connected peripheral devices include traffic lights, access control devices and additional displays. Node-RED starts inside a Docker container that is sent to the device using AWS Greengrass. Node-RED uses IPC protocol nodes for communication and I2C protocol and ASCII communication standard nodes for controlling devices. Traffic lights and access control devices were controlled by digital output nodes that use I2C protocol. Weighing readings were sent to the additional displays in ASCII format using a serial communication node.

Node-RED was installed for two customers in Finland. The peripheral devices and the Node-RED app have worked flawlessly.

Keywords

Node-RED, AWS, Greengrass, Docker, I2C, ASCII

Sisällys

1 Johdanto 1
2 Laitteisto 2
2.1 Vaaka tyypit ja protokollat 2
2.2 Esineiden internet 3
3 Suoritusympäristö 4
3.1 Suorittamis- ja kehitysympäristö 4
3.2 Docker-ympäristö 4
3.2.1 Docker-kuva 4
3.2.2 Docker-kontin käynnistäminen 7
4 Amazon-verkkopalvelut 10
4.1 Palvelut ja käyttökohteet 10
4.2 IoT-Core 11
4.3 AWS IoT Greengrass 13
4.4 IPC kommunikaation perusteet 14
4.4.1 IPC-aiheet 14
4.4.2 IPC-Viesti 15
5 Node-RED 17
5.1 Node-RED historia 17
5.2 Käyttöliittymä 18
5.3 Solmut 19
5.4 Viestit ja virrat 23
6 Digitaalitulo ja -lähtöportti 26
6.1 Digitaalitulo ja -lähtöportti yleisesti 26
6.2 I2C-perusteet ja toiminta 26
6.3 Arkkitehtuuri 26
6.3.1 Isäntä ja orja 27
6.3.2 Datan siirto 27
6.4 Rekisteri ja viestin rakenne 28
6.5 I2C työkalut Linuxilla 28
7 Sarjaliikenneportti 31
7.1 Sarjaliikenneportin toiminta 31
7.2 ASCII-perusteet 31
7.3 Laajennettu ASCII 32
7.4 ASCII-taulukko 33

8 Toteutus 34
8.1 Laitteet 34
8.2 Docker ympäristö 34
8.3 Pilvipalveluiden Integrointi 36
8.4 Käytetyt solmut..37
8.4.1 Käytetyt Node-RED valmiit ja ladatut solmut..38
8.4.2 Omien Node-RED solmujen rakentaminen 39
8.5 Virtojen rakentaminen..50
8.6 Ongelmat..55
8.7 Tulokset..55
8.8 Jatkokehitys ja ylläpito..56
9 Yhteenveto ja pohdinta 57
Lähteet 59

Lyhenneluettelo

ASCII	American Standard Code for Information Interchange. Merkintästandardi, jota laitteet käyttävät binäärin ja tekstin muuntajana.
AWS	Amazon Web Services. Amazonin tarjoama pilvipalvelu.
DIO	Digital Input/Output. Laitteiden ohjaamiseen tarkoitettu liitin yhdyskäytävässä.
ECR	Elastic Container Registry. AWS tarjoama tietokanta.
I2C	Inter-Integrated Circuit. Sarjaliikenneprotokolla kaksisuuntaisella tiedonsiirtoväylällä.
IaaS	Infrastructure as a Service. Pilvipalvelumalli, jossa käyttäjä saa käyttöönsä virtuaalisen IT-infrastruktuurin.
IPC	Inter-Process Communication. Laitteen sisäinen kommunikaatio tapa AWS Greengrass -ytimen avulla.
IoT	Internet of Things. Esineiden internet on verkossa olevia laitteita.
JSON	JavaScript Object Notation. Datan formatointi tapa.
PaaS	Platform as a Service. Pilvipalvelumalli, joka tarjoaa kehittäjille alustan sovellusten kehittämiseen ja hallintaan.
RPC	Remote Procedure Call. Protokolla, jonka avulla tietokoneohjelma voi suorittaa toiminnon toisessa tietokoneessa.
SaaS	Software as a Service. Pilvipalvelumalli, jossa ohjelmistot tarjotaan verkkopalveluna.
SDK	Software Development Kit. Ohjelmistokehittäjille tarkoitettu paketti, jossa on valmiita funktioita.
TLS	Transport Layer Security. Salausprotokolla, joka varmistaa verkkoliikenteen turvallisuuden.
POC	Proof of Consept. Testi versio siitä että onko konsepti mahdollinen.

1 Johdanto

Maailman materiaalivirroissa käytetään vaakoja materiaalien seurannassa. Maailmassa on erilaisia vaakoja eri tarkoituksiin, kuten pöytävaakoja, laboratoriovaakoja ja ajoneuvovaakoja. Vaakoja käytetään normaalin ihmisen elämässä useissa eri tarkoituksissa, kuten ruokakaupoissa tai jäteasemilla. Punnitsemisen ilmoittamiseen käytetään erilaisia järjestelmiä, kuten paperilappuja, jotka liimataan esineeseen tai punnituspalveluita, jotka toteuttavat punnitsemisen automaattisesti verkossa. Punnituspalvelut helpottavat suurien ja pienien yritysten materiaalivirtojen seuraamista ja automatisointia.

Samaan aikaan kun vaa'at toimivat, ne hyödyntävät erilaisia oheislaitteita, jotka toimivat vaa'an ja punnituspalvelun kanssa yhteistyössä punnituksen suorittamisessa. Yleisiä oheislaitteita autovaaoissa ovat liikennevalot, kulunohjauslaitteet ja lisänäytöt. Oheislaitteet eroavat toisistaan liitännöissä ja käytetyissä protokollissa. Jopa samanlaiset laitteet kuten lisänäytöt eroavat toisistaan valmistajan mukaan, koska laitteistoille ei ole tiettyä standardia toimintaan. Oheislaitteet ja vaa'at ovat yhdistettynä tietokoneeseen, jota kutsutaan yhdyskäytäväksi. Yhdyskäytävän tehtävä on toimia vaa'an, oheislaitteiden ja punnituspalvelun yhdistäjänä. Oheislaitteet liitetään yhdyskäytävän ulkoisiin liitäntäportteihin. Käytettävä liitäntäportti vaihtelee oheislaitteen mukaan. Yleisiin käytettyihin portteihin kuuluu digitaalitulo/lähtöportti ja sarjaliikenneportti. Oheislaitteita ohjataan portista riippuvalla protokollalla. Oheislaitteiden yhdistäminen punnituspalveluun vie suurien eroavaisuuksien takia paljon resursseja ohjelmistokehittäjiltä.

Opinnäytetyön tavoitteena on rakentaa POC (Proof of Consept) versio oheislaitteiden liittämisestä punnituspalveluun käyttämällä low-code työkalua. Työkalu täytyy integroida pilvipalveluun, josta se voidaan lähettää pilven kautta yhdyskäytävälle etänä ja työkalun täytyy toimia omassa ympäristössään erossa yhdyskäytävän isäntäkäyttöliittymästä. Oheislaitteiden liitäntä täytyy olla mukautettava ja uusia laitteita täytyy pystyä liittämään ilman suurta resurssitarvetta. Opinnäytetyö toteutetaan Tamtron Precision Oy:lle

Tamtron Precision Oy on Tamtron Oy:n tytäryhtiö. Tamtron Precisionilla on yli sadan vuoden historia punnitusalan erilaisista palveluista. Yrityksen entinen nimi on Lahti Precision, mutta nimi muutettiin vuonna 2023 kun Tamtron Oy osti yrityksen. Tamtron Precision Oy toimii punnitusalalla vaakojen valmistajana ja asentajana sekä punnituspalvelusovelluksen kehittäjänä. Vuonna 2023 yrityksellä oli 101 työntekijää ja 20 miljoonan euron liikevaihto.

2 Laitteisto

2.1 Vaaka tyypit ja protokollat

Tamtron Precision tukee erilaisia vaaka tyyppejä kuten auto-, lattia- ja siltavaakoja. Vaa’at yhdistetään yrityksen punnituspalveluun, josta voi verkon kautta suorittaa punnituksia ja materiaalivirtojen seurantaa. (Tamtron.) Kuvassa 1 on Tamtron Precisionin toimittama autovaaka.

Kuva 1. Toimitettu autovaaka (Tamtron)

Jotta yhteys punnituspalvelun ja vaa’an välillä onnistuu, tarvitaan yhdyskäytävä. Yhdyskäytävän tehtävä on yhdistää punnituspalvelu ja vaaka sekä kontrolloida oheislaitteita. Yhdyskäytävät ovat teollisuuskäyttöön suunniteltuja tietokoneita. Punnituksen tehostamista varten vaa’an ympäristöön asennetaan useita erilaisia oheislaitteita. Oheislaitteita kontrolloidaan eri liitäntäporttien kautta ja jokaisella liitäntäportilla on oma protokollansa, jota käytetään viestimisessä. Yleisimmät oheislaitteet ovat liikennevalot, kulunohjaislaitteet ja lisänäytöt. Liikennevaloja ja kulunohjauslaitteita kontrolloidaan digitaalitulo ja -lähtöportin kautta. Yksi tai useampi digitaalilähtö ohjaa oheislaitetta. Lisänäytöt toimivat sarjaportin kautta, josta lähetetään viesti näytölle. Yhdyskäytävät toimivat esineiden internetti ympäristössä.

2.2 Esineiden internet

Esineiden internet lyhenteeltään IoT (Internet of Things) tarkoittaa fyysisten esineiden yhdistämistä internettiin, jolloin ne voivat vuorovaikuttaa keskenään ja muiden järjestelmien kanssa. IoT-laitteet voivat kerätä dataa ympäristöstään, lähettää sitä pilvipalveluihin analysoitavaksi tai niitä voidaan ohjata ja seurata etänä internetin kautta. Tämä teknologia mahdollistaa teollisuuslaitteiden digitalisoinnin ja automatisoinnin (Emirica.)

IoT-laitteet voivat vaihdella yksinkertaisten antureiden ja monimutkaisten järjestelmien, kuten itseajavien autojen, välillä. Teknisesti IoT-laitteet koostuvat fyysisistä datan keruu komponenteista tai langattoman viestinnnän komponenteista, jotka mahdollistavat datan lähettämisen ja vastaanottamisen. IoT-laitteet voivat käyttää erilaisia langattomia yhteysteknologioita, kuten WI-FI ja 5G tai langallisia, kuten Ethernet. IoT kehityksessä keskeistä on pilvipalveluiden käyttö, jonne laitteet voivat tallentaa dataa analysoitavaksi. Pilvipalvelut tarjoavat laskentatehoa, tallennustilaa ja etäkäyttöönotto mahdollisuuksia. (English 2024.) IoT-järjestelmissä oheislaitteet käyttävät usein yhdyskäytävä-protokollia eivätkä ole itsenäisesti yhteydessä verkkoon. Yhdyskäytävä-protokolliin kuuluu IPC (Inter-Process Communication), MQTT ja I2C (Inter-Integrated Circuit). (Li 2024; Cambell.)

3 Suoritusympäristö

3.1 Suorittamis- ja kehitysympäristö

Oheislaitteiden ohjaaminen suoritetaan Docker-kontin sisällä, koska kontittamalla voidaan erottaa ohjelmisto muusta yhdyskäytävästä ja mahdollistaa samanlainen ympäristön jokaiselle laitteelle. Tämän avulla mahdollistetaan luotettavuus, että ohjelmisto toimii tuotantoympäristössä aina samalla tavalla.

3.2 Docker-ympäristö

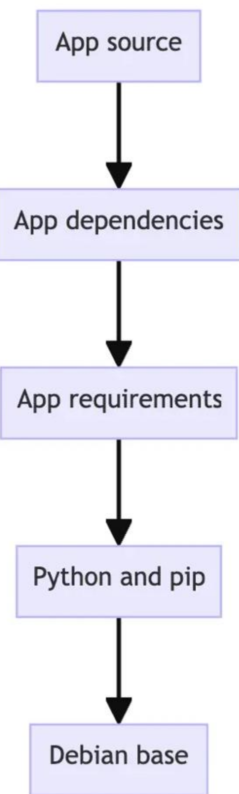
Docker on avoimen lähdekoodin ohjelmisto, joka toimii sekä Linux- että Windows-käyttöjärjestelmissä. Docker perustuu konttiteknologiaan ja on konttiteknologian yleistäjä teknologia alalla. Tätä myötä Docker on yksi maailman käytetyimmistä ohjelmistoista sovelluskehityksessä. Docker-kontit luovat abstraktion sovellustasolla eli pakkaavat sovelluksen ja sen tarvitsemat riippuvuudet yhteen eristettyyn tilaan muusta tietokoneen infrastruktuurista. Erottaminen tekee sovelluksen hallinnasta tehokkaampaa ja turvallisempaa. Kontitus pienentää kehitys- ja tuotantoympäristön eroavaisuutta ja varmistaa että jokaisella kehittäjällä on samanlainen ympäristö. (Walker 2024.)

Kontit ovat kevyempiä versioita virtuaalikoneista ja toimivat kätevämpinä työkaluina sovelluskehitykseen kuin virtuaalikoneet. Virtuaalikoneet ovat yleiskäyttöisiä työkaluja, jotka on suunniteltu tukemaan kaikenlaisia työkuormia samanaikaisesti. Sen sijaan kontit ovat kevyitä ja suunniteltu suorittamaan vain tiettyä tehtävää. Docker-kontit suunnitellaan tiettyä käyttötapausta varten sisältäen vain tarpeelliset työkalut. Kontit jakavat isäntäjärjestelmän käyttöjärjestelmän. Jakamalla käyttöjärjestelmän kontitus vapauttaa järjestelmäresursseja muille toiminnoille. Docker-konteille luodaan aina ohjeet, joilla kontti alustetaan käynnistämistä varten. Näitä kutsutaan kuviksi. (Walker 2024.)

3.2.1 Docker-kuva

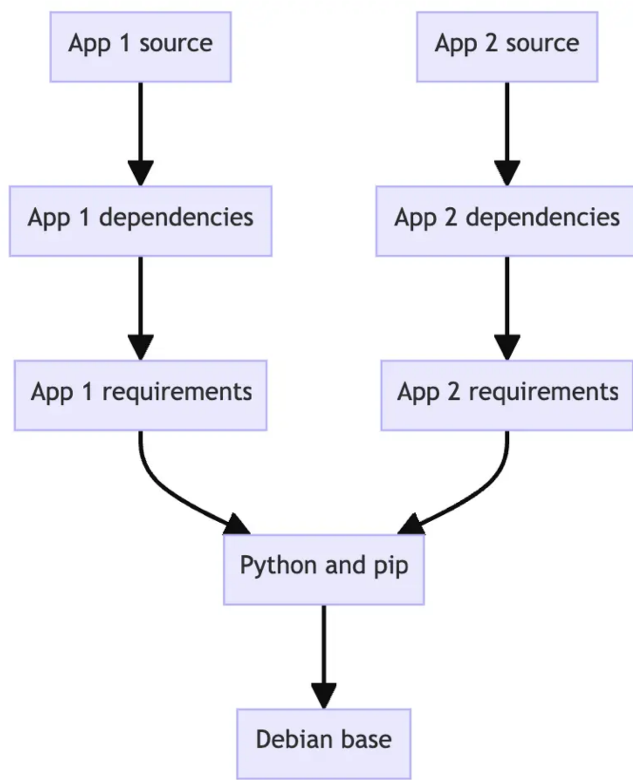
Docker-kuva on tiedosto ohjeita millainen kontin kuuluisi olla. Docker-kuvan virallinen nimeämismalli on Dockerfile. Docker-kuva on standardoitu paketti, joka sisältää kaikki tiedostot, kirjastot ja asetukset kontin toimintaa varten ja toimii Docker-konttien alustuksessa. Docker-kuvat ovat muuttumattomia, kun kuvat on luotu. Ainoa tapa muuttaa Docker-kuvaa on luoda uusi kuva tai lisätä muutoksia sen päälle. Konttikuvat koostuvat kerroksista, jotka näkyvät kuvassa 2. Jokainen kerros edustaa tiedostojärjestelmän muutoksia, jotka lisäävät, poistavat tai muokkaavat tiedostoja. Ensimmäisessä kerroksessa lisätään peruskomennot ja paketinhallintaohjelma, kuten apt. Toisessa kerroksessa asennetaan ajoympäristö ja

työkalut riippuvuuksien hallintaan. Kolmannessa kerroksessa kopioidaan sovelluksen requirements.txt-tiedosto, joka määrittelee sovelluksen tarvitsemat riippuvuudet. Neljäs kerros asentaa nämä riippuvuudet sovelluksen käyttöä varten. Viidennessä kerroksessa kopioidaan sovelluksen varsinainen lähdekoodi, jolloin kontti on valmis käynnistykseen. (Docker Inc f.)



Kuva 2. Docker-kuvan rakennus vaiheiden kerrokset järjestyksessä (Docker Inc d)

Kerrostus mahdollistaa kerrosten uudelleenkäytön eri kuvien välillä, jos luodaan toinen sovellus vanhan sovelluksen pohjalta, kuten kuvassa 3 on esitetty. Kuvassa esitetään kahden Docker-kuvan rakentaminen ja miten kerroksia voidaan uudelleen käyttää. Tämä nopeuttaa rakennus prosessia ja vähentää tarvittavan tallennustilan ja kaistanleveyden käyttöä. (Docker Inc d.)



Kuva 3. Docker-kuva tasojen uudelleenkäyttö (Docker Inc d)

Dockerfile on tekstipohjainen tiedosto, jolla on oma kirjoitus formaattinsa (Docker Inc g). Ensimmäinen asia Dockerfilessä on peruskuva. Peruskuva on Docker-kontin aloitusympäristö. Peruskuva ilmoitetaan FROM-komennolla. Peruskuva voi olla käyttöjärjestelmä, kuten Ubuntu tai ohjelmisto, kuten Python. Tyhjän peruskuvan saa SCRATCH-komennolla. (Docker Inc a.)

Peruskuvien koko riippuu kuvan versiosta. Alpine on pienin mahdollinen versio Docker-kuvalle ja sisältää vain käyttöjärjestelmän olennaisimmat osat. Dockerfilessä asetetaan kotihakemisto komentojen suorittamiselle WORKDIR-komennolla. Tiedostojen siirtäminen isäntälaitteen tiedostohakemistosta Docker-kontin sisälle tehdään COPY-komennolla. Komennossa ilmoitetaan mistä isäntälaitteen hakemistosta ja minne Docker-kontin sisälle siirretään kansiot ja tiedostot. Docker-kuvan rakennusvaiheessa ajetaan komentorivikomentoja RUN-komennon avulla. Komento luo aina uuden kerroksen Docker-kuvaan rakennusvaiheessa. Docker-kontin sisällä toimivat komentorivikomennot riippuvat peruskuvasta. Docker-kontin porttinumero ilmoitetaan EXPOSE-komennolla. EXPOSE-komento dokumentoi mitkä portit ovat käytettävissä ja mihin portteihin sovellukset odottavat liikennettä. Komento toimii vain tiedotteena käytössä olevista porteista eikä avaa portteja. Docker-kontille asetetaan käyttäjä USER-komennolla, joka määrittää suoritettavien komentojen käyttäjäoikeudet. Käyttäjän määrittämisellä kasvatetaan tietoturvaa kontin sisällä. (FOSS TechNix 2020.) Kuvassa 4 on esitetty Dockerfile, jossa asennetaan Python 3.12 ja riippuvuudet

requirements.txt tiedostosta. Kun Docker-kontti käynnistyy, käynnistetään Python projekti ja avaa kontille portin numero 8080.

```
FROM python:3.12
WORKDIR /usr/local/app

# Install the application dependencies
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

# Copy in the source code
COPY src ./src
EXPOSE 5000

# Setup an app user so the container doesn't run as the root user
RUN useradd app
USER app

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8080"]
```

Kuva 4. Dockerfile Python 3.12 pohjalta (Docker Inc g)

Docker-kontin aloittamiskomento asetetaan CMD-komennolla. Komento kertoo kontille, mitä ohjelmistoa tai komentoja kontissa tulisi suorittaa, kun kontti käynnistetään. Docker-kuvat rakennetaan komennolla docker build. (FOSS TechNix 2020.)

3.2.2 Docker-kontin käynnistäminen

Kun Docker-kuva on rakennettu, Docker-kontti käynnistetään ajamiskomennolla eli docker run. Docker ajokomennon yhteydessä voi ajaa useita asetuskomentoja samalla. Asetus kohdassa asetetaan –p komennolla porttinumerot, –e asetetaan ympäristömuuttujat ja –v volyymit. Kuvassa liitetään rakennettu Docker-kuva nimellä tai id avulla. Komennot ja argumentit ovat valinnaisia ja yliajavat Docker-kuvan aloituskomennon. Argumentteja käytetään komennon muokkauksessa. (Linuxize 2020.) Docker-kontin portit mahdollistavat tietoliikenteen konttien ja ulkomaailman välillä. Usealla Docker-kontilla on omia avoimia portteja, jotka ovat käytössä vain kyseisessä kontissa. Docker-kontin sisälle pääsee ulkomaailmasta asettamalla kontin portit isäntäjärjestelmän porteiksi. Portitus tehdään yleisesti kontin käynnistyksen yhteydessä. Käyttämällä -p asetusta Docker-kontin portti uudelleenohjataan näkymään isäntälaitteen portissa kuten kuvassa 5 on esitetty (Docker Inc c.) Kuvassa käynnistetään docker-kontti my_container kuvasta ja uudelleenohjataan kontin portti numero 80 isäntälaitteen porttiin 8080.

```
docker run -p 8080:80 my_container
```

Kuva 5. Docker-kontin käynnistys komento

Kun portit on julkaistu, Docker ohjaa automaattisesti liikenteen isäntäjärjestelmän portista oikeaan kontin porttiin. Tämä mahdollistaa verkkosovellusten käytön, jotka suoritetaan Docker-kontissa. (Docker Inc c.)

Useiden Docker-konttien kanssa käytetään Docker-yhdistelmää Docker-kuvien rakentamisessa ja käynnistämisessä Docker ajokomennon sijasta. Docker-yhdistelmä mahdollistaa monikonttisten sovellusten määrittämisen ja ajamisen yhden docker-compose.yaml tiedoston avulla. Yml loppuinen tiedosto on Docker-yhdistelmän käyttämä vanhempi tiedostotyyppi. Tiedosto sisältää kaikki Docker ajokomennon argumentit yhdessä tiedostossa. Docker-yhdistelmä korvaa rakentamisen- ja ajamisenkomennot omilla docker compose build/up komennoilla. (Docker Inc b.)

Volyymit ovat Docker-kontin ulkopuolella olevia tallennustiloja, joihin kontit voivat siirtää dataa. Volyymit ovat täysin Dockerin hallinnassa ja tämä tekee niistä helppoja varmuuskopioida ja siirtää. Volyymit toimivat sekä Linux- että Windows-konteissa, ja niiden käyttö on turvallista ja luotettavaa useiden konttien välillä. Uuden volyymin voi alustaa valmiiksi kontin sisältämällä datalla. Volyymit ovat turvallisempi tapa tallentaa dataa kuin kontin sisälle, koska volyymin sisältö säilyy riippumatta kontin elinkaaresta. (Docker Inc e.) Oletusasetukseltaan Docker-kontit ovat aina omia ympäristöjään eivätkä kykene käyttämään isäntälaitteen toimintoja, mutta privileged-komennolla voidaan ohittaa tämä esto. Privileged-komento antaa kontille juurioikeudet isäntälaitteen käyttöjärjestelmään mahdollistan kontille oikeudet tehdä mitä vain. (Simic 2020.) Kuvassa 6 on näkyvissä docker-compose.yaml tiedosto. Kuvassa on docker-compose.yaml tiedosto, jossa käynnistetään kolme Docker-konttia.

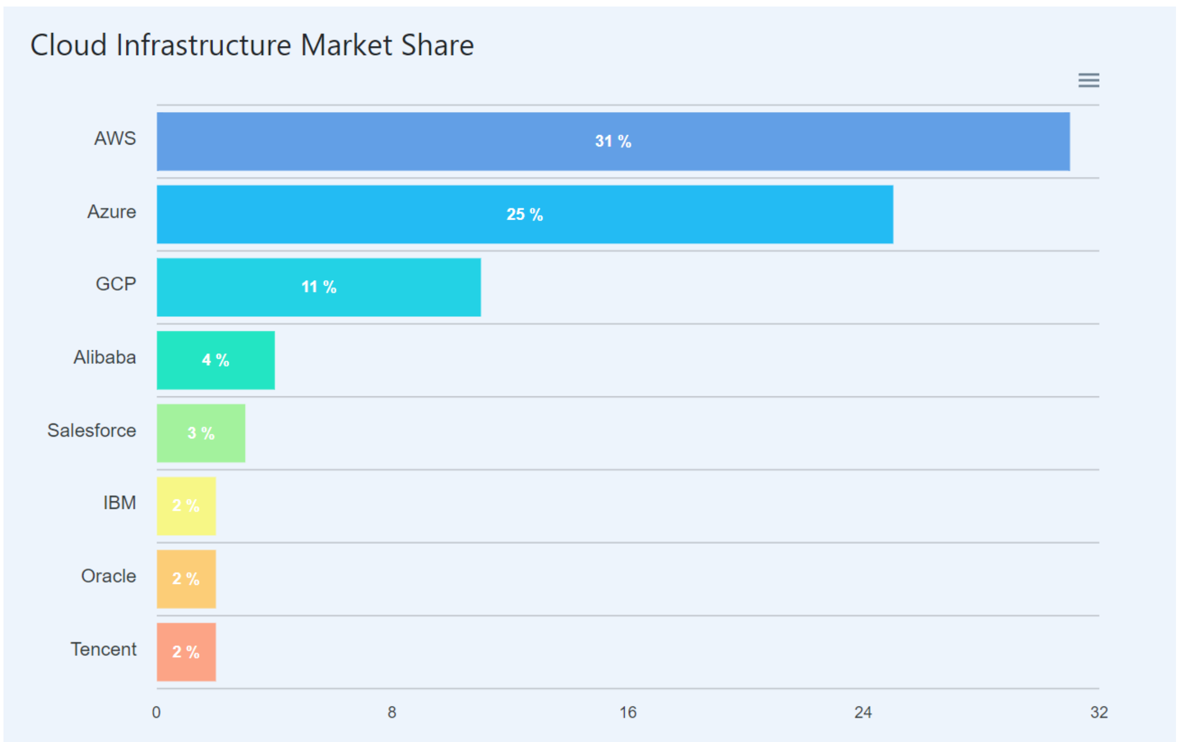
```yaml
version: "3.9" # specify the version of the compose file format

services: # define the services or containers that make up your application

  web: # name of the first service
    image: nginx # name of the image to use for this service
    ports: # list of ports to expose on the host machine
      - "80:80"
    depends_on: # list of services that this service depends on
      - app

  app: # name of the second service
    build: ./app # path to the directory containing the Dockerfile for this
    environment: # list of environment variables to pass to this service
      - DB_HOST=db
      - DB_USER=root
      - DB_PASS=secret
      - DB_NAME=todos
    volumes: # list of volumes to mount on this service
      - ./app:/app

  db: # name of the third service
    image: mysql # name of the image to use for this service
    environment: # list of environment variables to pass to this service
      - MYSQL_ROOT_PASSWORD=secret
      - MYSQL_DATABASE=todos

volumes: # define any named volumes used by the services
  db_data: # name of the volume for persisting database data

networks: # define any custom networks used by the services
  default: # name of the default network
    driver: bridge # driver to use for this network
```

Kuva 6. Docker-compose.yaml tiedosto (Fawade 2023)

4 Amazon-verkkopalvelut

4.1 Palvelut ja käyttökohteet

Amazon-verkkopalvelut eli AWS (Amazon Web Services) on Amazonin omistama pilvipalvelu alusta (Page V 2024). AWS aloitti pilvipalvelu tarjonnan vuonna 2006 ja vuonna 2024 tarjoaa palveluaan 190 eri maahan (Amazon Web Services 2024). Vuonna 2024 AWS on maailman suurin ja kattavin pilvipalvelualusta. Maailmanlaajuisista pilvipalvelumarkkinoista AWS edustaa 31%:a. Kuvassa 7 on nähtävissä maailmanlaajuisten markkinoiden jako vuodelta 2024. (Wawira, M. 2024.)

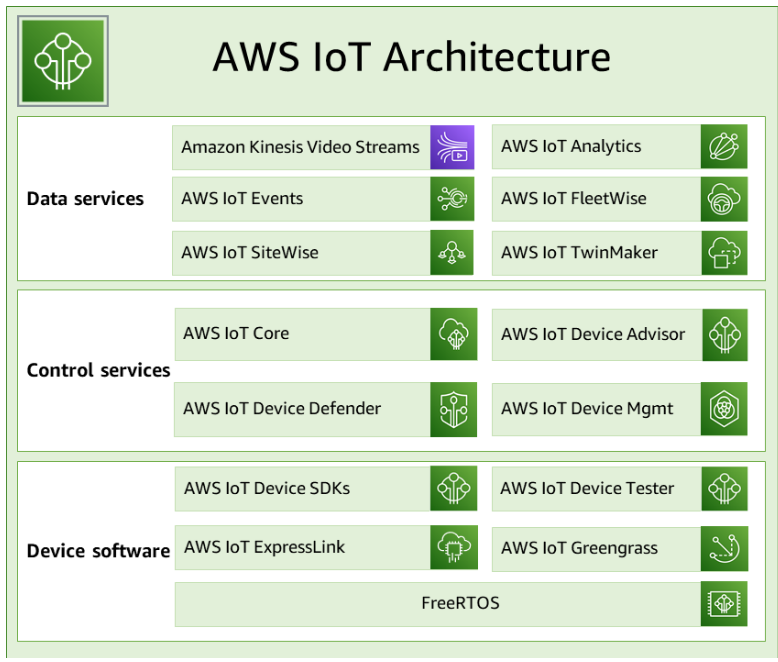


Kuva 7. Pilvipalvelu markkina jako (Kiruri 2024)

AWS tarjoaa kolmea eri hinnoittelumallia IaaS (Infrastructure as a Service), SaaS (Software as a Service) ja PaaS (Platform as a Service) (Amazon Web Services 2024). IaaS tarjoaa perusinfrastruktuurin, kuten palvelimet, verkot, tallennustilan ja virtuaalikoneet, joita käyttäjät hallitsevat itse. PaaS tarjoaa kehittäjille ympäristön sovellusten kehittämistä varten. SaaS tarjoaa valmiita sovelluksia ja ohjelmistoja, joita käyttäjät voivat käyttää suoraan verkossa, ilman tarvetta asentaa niitä omille laitteilleen. (Raza 2024.)

Käyttäjät voivat skaalata resursseja kysynnän mukaan, mikä varmistaa resurssien tehokkaan käytön ja kustannustehokkuuden. AWS tarjoamiin pilvipalveluihin kuuluu IoT, tietokanta, analytiikka ja tietoturvallisuus palveluita. (Amazon Web Services 2024.) AWS IoT

tarjoamat palvelut voidaan jakaa kolmeen eri osaan: data- ja kontrollipalveluihin ja laitesovelluksiin, kuten kuvassa 8 esitetään (Amazon Web Services c).

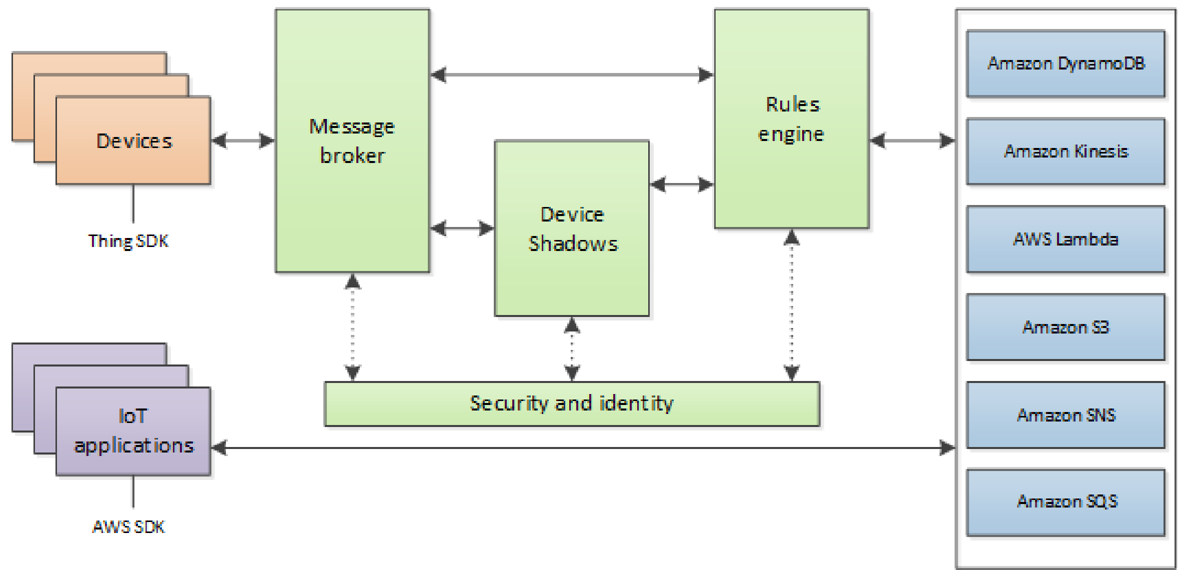


Kuva 8. AWS IoT palvelut (Amazon Web Services c)

AWS IoT-Core mahdollistaa laitteen yhdistämisen AWS pilveen ja laiteen hallitsemisen (Amazon Web Services c). AWS IoT SDK (Software Development Kit) on avoimenlähdekoodin ohjelmistokehitystyökalu, jolla käytetään AWS IoT palveluita. AWS IoT SDK tarjoaa valmiita funktioita sovelluskehittäjille, joita hyödyntämällä sovellukset ja pilvipalvelut voivat kommunikoida keskenään sujuvasti. SDK tarjoaa kirjastoja ja työkaluja eri ohjelmointikielille, kuten C, Python ja JavaScript. (To 2024.) AWS IoT Greengrass avulla sovelluksia voi lähettää verkon kautta laitteille riippumatta aikataulusta. Tämä mahdollistaa IoT-laitteiden päivityksen verkon ylitse. (Amazon Web Services c.)

4.2 IoT-Core

IoT-Core on AWS tarjoama IoT-palvelu (Vadim S 2024). IoT-Coren tehtävä on yhdistää fyysiset laitteet AWS pilvipalveluun sekä reitittää suuria määriä viestejä laitteiden tai AWS välillä luotettavasti ja turvallisesti (Emqx 2023). IoT-Coren palveluihin kuuluu viestien välittäjä, laitevarjot, sääntömoottori ja tunnistautuminen (Amazon Web Services c). IoT-Coren palveluita havainnollistaa kuva 9, joka esittää eri palveluiden riippuvuuksia toisiinsa.



Kuva 9. IoT-Coren palvelut (Amazon Web Services c)

IoT-Core tukee HTTP, WebSocket ja MQTT tiedonsiirtoprotokollia. MQTT tiedonsiirtoprotokolla on suunniteltu sietämään ajoittaisia ja hitaita yhteyksiä. Kaikki nämä tiedonsiirtoprotokollat voivat olla yhteydessä IoT-Coren viestien välittäjään, joka takaa turvallisen tiedonsiirron, koska kaikki IoT-Core viestiliikenne hyödyntää TLS yhteyden suojausta. Laitevarjo on JSON (JavaScript Object Notation) tiedosto, johon tallennetaan laitteiden nykyinen tilannetieto. Tilannetiedon määrittelee käyttäjä itse ja se voi muuttua milloin vain. Laitevarjo auttaa laitteiden parametroimisessa. Sääntömoottori on AWS tietoturvatyökalu, jolla voi luoda sääntöjä, joita laitteen täytyy noudattaa AWS ympäristössä. Turvallisuuteen ja tunnistukseen AWS IoT-Core käyttää X.509-varmenetta. Laite, joka haluaa käyttää AWS IoT palveluita, täytyy olla rekisteröitynyt IoT-Coreen. (Amazon Web Services c.) Laitteet muodostavat yhteyden IoT-Coreen rekisteröimällä itsensä IoT-laitteen alle. IoT-laite on representaatio laitteesta tai loogisesta kokonaisuudesta. Laite voi olla fyysinen kokonaisuus tai anturi. Looginen kokonaisuus tarkoittaa sovellusta, joka ei itsessään muodosta yhteyttä AWS IoT-ytimeen vaan muihin laitteisiin, jotka muodostavat yhteyden. (Amazon Web Services e.)

IoT-laitteen rekisteröiminen vaatii käytännön, X.509-sertifikaatin ja suojatun TLS-yhteyden käyttämisen. Käytännöt ovat sääntöjä, jotka määrittelevät mitä toimintoja laite saa suorittaa AWS ympäristössä. Näitä sääntöjä luodaan sääntömoottorin kautta. IoT-sääntöihin kuuluu iot:Publish, iot:Subscribe ja iot:Connect. Iot:Publish sääntö sallii laitteen MQTT-viestien julkaisemisen AWS IoT Coren kautta. Iot:Subscribe sallii MQTT-aiheiden tilaamisen IoT-Coresta. Molemmilla säännöillä voi rajata viestien julkaisemisen ja tilaamisen aiheita. Iot:Connect sallii laitteiden yhdistämisen AWS IoT-Coreen. X.509-sertifikaatti on digitaalinen varmenne, jota käytetään laitteen autentikoimisessa. Varmenne toimii henkilöllisyystodistuksena, joka varmistaa, että laite on luotettava ja oikea. TLS-yhteys on salausprotokolla, joka

varmistaa turvallisen tiedonsiirron laitteen ja IoT-Coren välillä. TLS yhteys suojataan kolmella tavalla, joita ovat viestien salakirjoitus, viestien eheyden varmistus ja autentikointi, jossa käytetään X.509-sertifikaattia molempien AWS ja laitteen henkilöllisyyksien varmistamisessa. (Amazon Web Services e.)

IoT-laitteet merkitään IoT-Coressa laiterekisterin alle yksilöllisellä nimellä. Kun laitteen rekisteröiminen on valmis, laite voi käyttää AWS IoT muita työkaluja, kuten Greengrassiä ja viestin välittäjää. (Vadim 2024.)

4.3 AWS IoT Greengrass

AWS IoT Greengrass on AWS tarjoama ohjelmistoalusta, jolla sovelluskomponentteja voi lähettää pilvestä laitteelle. Komponentin lähettäminen mahdollistaa sovellusten päivittämisen verkon ylitse laitteille. Greengrass mahdollistaa komponenttien keskustelun keskenään paikallisesti ilman, että laite on jatkuvasti yhteydessä verkkoon tai pilvipalveluun IPC-protokollalla. Aina kun laite saa yhteyden pilveen laite voi synkronoida tiedot pilven kanssa ja jatkaa toimintaa verkon ollessa poikki tai viiveiden aikana. Pilven kanssa tietojen synkronointi tapahtuu MQTT-protokollon avulla. (Amazon Web Services g.)

Kaikki Greengrass-ohjelmistot ovat komponentteja. Komponentteja tarjotaan valmiina ja ne sisältävät toimintoja sovelluksille. Komponentteihin kuuluu ydin, joka on ainoa pakollinen komponentti. Ydin-komponentti tarjoaa minimaalitoiminnot Greengrassin toiminnalle laitteella. Ydin hallitsee muiden komponenttien käyttöönottoa ja elinkaarta sekä mahdollistaa IPC-kommunikaation. Vaihtoehtoisiin valmiiksi rakennettuihin komponentteihin kuuluu Docker-sovellusten hallinta, joka mahdollistaa Docker-kuvien lataamista AWS ECR (Elastic Container Registry) tietokannasta. Tämä komponentti on pakollinen, jos hyödyntää Docker-kontitusta. Greengrass sallii omien komponenttien luomisen. (Amazon Web Services e.)

Omatekoinen komponentti vaatii reseptitiedoston komponentin lähetystä varten Greengrassiin. Reseptitiedoston täytyy olla joko YAML tai JSON tiedosto. Tiedostossa määritellään yksityiskohtaiset ohjeet, miten komponenttia hallitaan, ajetaan ja käytetään. Reseptiin määritetään komponentin versio, riippuvuudet, konfiguraatiot sekä suoritettavat toiminnot käynnistyessä. Komponentille voi asettaa riippuvaisuuksia, joilla tarkoitetaan riippuvuutta toiseen Greengrass-komponenttiin. Riippuvuuksia on kahdenlaisia, kovia ja pehmeitä. Kovassa riippuvuudessa komponentti on pakollinen eikä riippuvainen komponentti voi käynnistyä tai päivittyä ilman tätä komponenttia. Pehmeässä riippuvuudessa komponentti ei ole pakollinen ja riippuvainen komponentti voi silti toimia, mutta toiminnot saattavat olla rajoitettuja. (Amazon Web Services e.)

Ennen kuin komponentti lähetetään, komponentista luodaan artefakti reseptin perusteella. Artefakti on varsinainen ohjelmisto. Ohjelmisto voi olla lähdekoodi, skripti tai Docker-kontti. Jos komponentti on Docker-kontti, artefakti on tässä tapauksessa Docker-kuva ja reseptiin asetetaan Dockerin ajokomento. IoT-laite täytyy yhdistää Greengrass -palveluun luomalla Greengrass-asia tai lisäämällä se Greengrass-ryhmään. Kun komponentti on lähetetty, komponentin voi käyttöönottaa millä vain yhdistetyllä IoT-laitteella. (Amazon Web Services e.)

4.4 IPC kommunikaation perusteet

IPC on paikallinen viestintäkerros, joka mahdollistaa komponenttien välisen turvallisen ja tehokkaan kommunikoinnin IoT-laitteissa. IPC merkitys korostuu kyvystä vaihtaa tietoja ja tehdä yhteistyötä ilman tarvetta jatkuvalle pilviyhteydelle. Greengrass-ydinkomponentti on keskeinen vaatimustekijä IPC-kommunikaatiolle, ja sen käyttö tapahtuu AWS IoT SDK tarjoamien funktioiden avulla. IPC tarjoaa rajapinnan, jonka avulla viestien vaihto on mahdollista samalla rajoittaen pääsyä vain sallittujen komponenttien välille. IPC-rajapinnan käyttöoikeudet määritellään reseptissä, mikä lisää turvallisuutta. IPC-kommunikaatio erottuu muista IoT-laitteille suunnatuista protokollista kuten MQTT siten, että viestit eivät kulje välittäjän kautta, vaan menevät suoraan tilaajalle. Tätä viestintä tapaa kutsutaan julkaisija-tilaaja (publisher-subscriber) periaatteeksi. Viestien aihe määrittää, mihin viesti julkaistaan ja kuka sen vastaanottaa tilaamalla kyseisen aiheen. AWS IoT IPC-aiheissa käytetään samoja sääntöjä kuin MQTT-aiheissa, mikä mahdollistaa joustavan viestinnän. IPC-kommunikaatio hyödyntää etäproseduurikutsua eli RPC (Remote Procedure Call) viestinnässä. (Amazon Web Services f.)

RPC avulla Docker-kontin ohjelma voi kutsua toisen kontin aliohjelmaa ikään kuin se olisi paikallinen aliohjelma. IPC-viestit muodostavat aina yhteyden Greengrass-ydinkomponenttiin RPC kautta, jonka avulla hyödynnetään AWS IoT SDK toimintoja (Amazon Web Services h). Toimintaa varten IPC vaatii ympäristömuuttujia, kuten IoT-laite nimen, polun IPC-sockettiin ja salaisen tunnuksen, joka muodostaa yhteyden IPC-sockettiin (Amazon Web Services a). Socket on ohjelmistorajapinta, joka yhdistää sovellusten tietoliikenteen ja mahdollistaa tietojen lähettämisen ja vastaanottamisen (Kalin 2019).

4.4.1 IPC-aiheet

Aiheet toimivat yhteyksien suodattimina. Aiheita voi luoda vapaasti julkaisijat ja tilaajat. Aiheet ovat pelkkiä tekstikenttiä ja toimivat hierarkkisesti samalla tavalla kuin tietokoneiden tiedostojärjestelmät käyttämällä hierarkiassa erottimena kauttaviivaa “/” kuten kuvassa 10 on esitetty. Aiheissa käytetään kauttaviivaa aihetasojen nimien erottamisessa. Aiheiden

nimet toimivat eri tasojen tunnisteina ja auttavat viestien reitityksessä. Jotta aiheita pidetään kelvollisina, on aiheen nimessä oltava vähintään yksi merkki ja käytetään UTF-8-merkkejä. Aiheet ovat kirjainkokoherkkiä, merkit plus ja risuaita ovat tarkoitettu kuvastamaan villejä kortteja. (Light 2023.)

Kuva 10. IPC-aihe (HiveMQ Team 2024)

Villeillä korteilla kerrotaan, että tilaaja haluaa yhdistää kaikkiin aiheisiin, joilla on samanlainen aiherakenne kuin tilatulla aiheella. Villejä kortteja voi olla aiheessa vain kaksi ja niitä käytetään vain tilaamisessa. Plus-merkki on villi korttina yksitasoinen. Eli tilattavien aiheiden on oltava aivan samanlaisia tasojen määrissä mutta plus-merkin kohdalla voi olla mitä tahansa. Risuaita-merkkiä käytettäessä tilataan kaikki aiheet, joilla on samanlainen rakenne risuaitaan saakka ja risuaidasta eteenpäin tilataan kaikki aihetasot. Villejä kortteja käytettäessä suositellaan kirjoittamaan mahdollisimman monta tarkkaa aihetasoa, jotta tilaaja ei kuormitu monista samanaikaisista viesteistä. (Light 2023.)

4.4.2 IPC-Viesti

IPC-viestit koostuvat useista osista, joista jokaisella on oma roolinsa tiedon välittämisessä. Keskeinen osa viestien rakennetta on tilatessa SubscriptionResponseMessage ja lähetyksessä publishMessage. Nämä sisältävät useita viestejä. Viestit voivat olla JSON-muotoisia tai binaarimuotoisia. Viestin muoto riippuu viestin luonteesta ja käyttötarkoituksesta. JSON-viestit tarjoavat helpon ja luotettavan tavan välittää tietoa eri laitteiden välillä. JSON-viesti sisältää useita kenttiä, joista tärkeimmät ovat viesti ja konteksti. Viesti kenttä sisältää varsinaisen JSON-datan. JSON-rakenne mahdollistaa tietojen organisoinnin ja käsittelyn helposti. Konteksti kenttä tarjoaa lisätietoa viestin kontekstista, kuten aiheen, johon viesti on lähetetty. Tämä tieto auttaa vastaanottavaa laitetta ymmärtämään, mistä viestissä on kyse, ja miten siihen tulisi reagoida. Binaariviestit tarjoavat vaihtoehtoisen tavan tiedon välittämiseen. Ne ovat erityisesti hyödyllisiä, kun siirretään suuria määriä dataa tai kun dataa ei ole helppo esittää JSON-muodossa. Binaariviesti sisältää kentät viesti ja konteksti. Viesti sisältää datan, joka voi olla mitä tahansa binääridataa, kuten kuvia tai muita tiedostoja. Konteksti toimii samalla tavalla kuin JSON-muodossa. (Amazon Web Services d.) Binäärimuotoista viestiä esitetään kuvassa 11. Binäärimuotoinen viesti on message avaimen arvona.

```
{
  "binary_message": {
    "message": "QltbIm5vZGVfcmVkIl1d",
    "context": {
      "topic": "test/node_red/update_parameters"
    }
  }
}
```

Kuva 11. IPC binäärimuotoisen viestin rakenne

5 Node-RED

5.1 Node-RED historia

Node-RED on IBM kehittämä low-code tietovirtoihin perustuva ohjelmointityökalu avoimella lähdekoodilla. Node-RED on osa OpenJS säätiötä. Nick O'Learyn ja Dave Conway-Jones aloittivat Node-RED kehittämisen vuonna 2013 sivuprojektina. Projektin alkuperäinen tarkoitus oli yrittää todistaa konsepti MQTT-aiheiden kartoituksesta, visualisoinnista ja käsittelystä. Node-RED laajeni työkaluksi, joka soveltui moniin eri käyttötapauksiin. Lokakuussa vuonna 2016 Node-RED liittyi yhdeksi JS säätiön perustusprojekteista. (OpenJS Foundation j.) OpenJS säätiö perustettiin vuonna 2019 kun Node.js ja JS säätiöt yhdistyivät saman säätiön alle. OpenJS säätiön projekteihin kuuluu yli 30 erillaista Javascript-pohjaista projektia ja tuki yli 30 eri suuryritykseltä, kuten Google ja Microsoft. (OpenJS Foundation a.) Syyskuussa 2023 Node-REDistä tuli avoimen lähdekoodin projekti. Node-RED on Node.js kielestä rakennettu low-code työkalu, joka on suunniteltu toimimaan vähä resurssisilla laitteilla ja erilaisissa ympäristöissä, kuten paikallisella tietokoneella, Rasberry Pi, Dockerilla ja AWS pilvipalvelulla. (OpenJS Foundation g.)

Node-RED ohjelmoiminen tapahtuu selaimessa käyttöliittymän kautta (OpenJS Foundation c). Node-RED sisältää laajan määrän erilaisia valmiita solmuja perus IoT toimintoja varten. Kun tarvitsee tehdä toiminto, joka ei toteudu valmiilla solmulla, Node-RED mahdollistaa omien solmujen rakentamisen käyttämällä ohjelmointikieliä HTML ja Node.js. Omatekoiset solmut voi julkaista Node-RED kirjastoon, josta kehittäjät voivat ladata solmun omaan käyttöönsä. Node-RED konseptit ja käyttäjädata tallennetaan tiedostoihin kansiossa ~/data. Tiedostoja päivittetään käyttöliittymästä tallentamalla tai tiedostoja muokkaamalla manuaalisesti. Node-RED tiedostoihin kuuluu settings.js, johon tallentuu käytössä olevat asetukset ja flows.json, johon tallennetaan kaikki tiedot virroista, välilehditä ja solmuista. (OpenJS Foundation e; OpenJS Foundation m.)

Low-code ja tietovirtoihin perustuva ohjelmoiminen

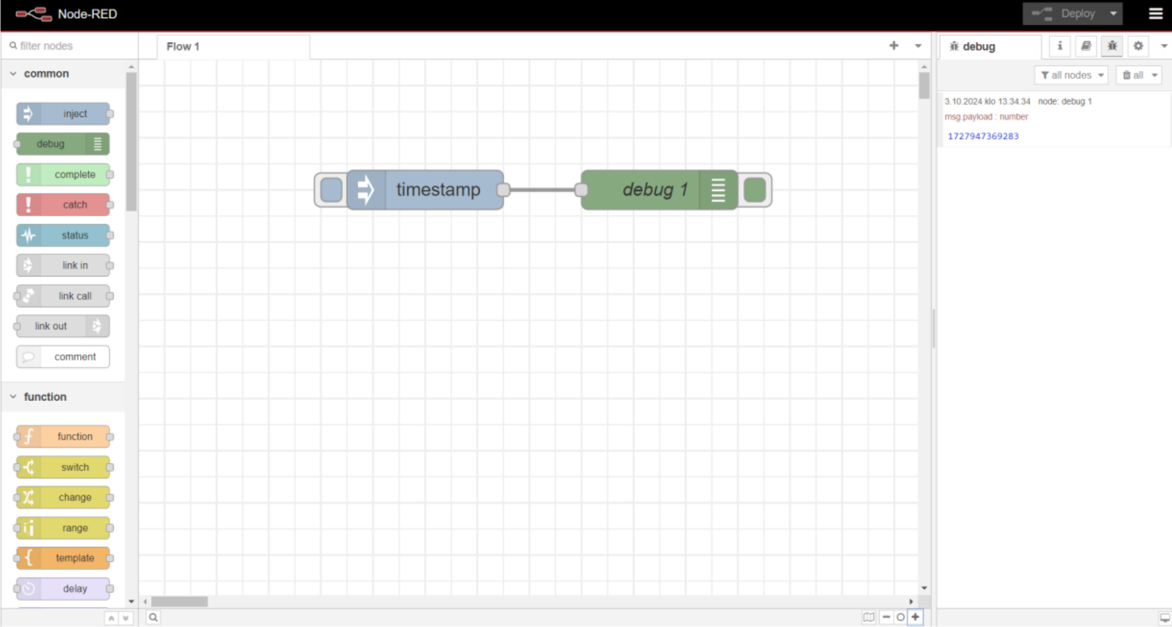
Node-RED perustuu low-code ohjelmoimiseen eli solmu ohjelmoimiseen. Solmu ohjelmoimisessa yhdistetään erilaisia toimintoja toisiinsa. Jokainen solmu kuvastaa erilaista toimintoa, kuten datan muuttamista, viestin lähettämistä tai datan filtteröintiä. Low-code ohjelmoinnin ja perinteisen ohjelmoinnin ero on että low-code ohjelmoijan ei tarvitse luoda toimintoja tyhjästä vaan hän yhdistelee valmiiksi luotuja solmuja. Solmujen välillä kulkee tietovirtoja jotka vastaavat viestin kulusta ja solmun käynnistymisestä oikeaan aikaan. Yleisimmät low-code ohjelmistokehityksen työkalut ovat Node-RED ja Microsoft PowerApp.

Low-code toimii hyvin kun ei haluta käyttää paljoa resursseja sovelluskehitykseen. (Ambientia 2023.)

Tietovirtoihin perustuvan ohjelmoinnin kehitti J. Paul Morrisonin 1970-luvulla. Tietovirtoihin perustuva ohjelmointi on tapa kuvata sovelluksen toimintaa verkostona, joka koostuu solmuista. Jokaisella solmulla on oma tehtävänsä virrassa datan käsittelyssä. Tietovirta ja low-code ohjelmointi ovat molemmat hyvin visuaalisesti esitettyjä. Visuaalisuudella luodaan matalakynnys ohjelmoimiselle. (OpenJS Foundation b.)

5.2 Käyttöliittymä

Node-RED käyttöliittymä avautuu osoitteesta http://localhost:1880/ kun verkkoselain ja Node-RED ovat molemmat samalla laitteella. Jos Node-RED on käynnissä ulkoisella laitteella niin käyttöliittymään saa yhteyden korvaamalla localhost sanan laitteen IP-osoitteella. (OpenJS Foundation m.) Kuvassa 12 on Node-RED käyttöliittymä. Kuvassa on rakennettu syöttö- ja debugaus-solmuilla virta.



Kuva 12. Node-RED käyttöliittymä

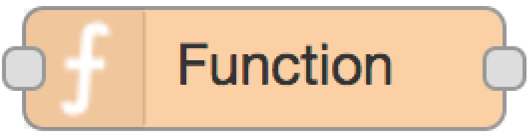
Node-RED käyttöliittymän voi jakaa neljään eri komponenttiin. Ylätunniste on käyttöliitttymän yläosa ja sisältää lähetys-painikkeen, päävalikon ja käyttäjävalikon. Lähetys-painike toimii Node-RED virtojen aloituksena ja tallennuspainikkeena. Painikkella on kolme eri muotoa, joissa kaikissa tallennetaan solmujen ja virtojen tiedot flows.json tiedostoon, mutta oletusasetuksena lähetys-painikkeella on että kaikki virrat käynnistetään uudelleen, toinen muoto on että virrat, joissa on muutoksia, käynnistetään uudelleen ja viimeinen muoto on, että vain muutetut solmut käynnistetään uudelleen. Muutokset tulevat käyttöön

uudellen käynnistyksen jälkeen. Samasta valikosta, josta lähetysmuoto valitaan, voi myös käynnistää kaikki virrat uudelleen. (OpenJS Foundation f; Steve 2022.)

Päävalikosta muutetaan käyttöliittymää ja asetuksia (FlowFuse 2024b). Käyttöliittymässä on paneeli, joka sisältää kaikki käytettävissä olevat solmut. Paneelissa solmut on jaettu omiin kategorioihinsa, joita ovat yleiset-, toiminto-, verkko-, syöttö-, lähtö-, sosiaaliset-, tallennus- ja analytiikkasolmut. Luokkia luodaan lisää solmuissa ilmoittamalla kategorian nimi. Käyttöliitymän keskellä on työalue, jossa luodaan virtoja yhdistetämällä tarvittavat solmut tietovirtojen avulla ja alueen yläpuolella on välilehti valikko. Käyttöliittymän oikealla on sivupalkki, joka sisältää virroista info-, apu- debugaus- ja konfiguraatiovälilehdet. (OpenJS Foundation l; OpenJS Foundation n.)

5.3 Solmut

Solmut toteuttavat aina tietyn toiminnon. Solmut on nimetty Node-RED käyttöliittymässä kuvastavalla nimellä, kuten kuvassa 13 on esitetty. (OpenJS Foundation k.) Kuvassa on Node-RED funktiosolmu.



Kuva 13. Node-RED funktiosolmu (OpenJS Foundation o)

Jokainen solmu toimii omana instanssinaan eikä jaa parametrejaan muille solmuille normaalisti. Vaikka solmut toimivat omissa instansseissaan osa solmuista käyttää konfiguraatiosolmuja, jotka ovat uudelleen käytettäviä parametreja, kuten MQTT-välittäjän määritteleminen. Solmuja yhdistetään tietovirroilla solmujen tulo- ja lähtöporttien kautta. Tietovirralla kerrotaan solmulle minne viesti lähetetään. Solmulla voi olla useita lähtöportteja, mutta vain yksi tuloportti. Kaikissa porteissa voi olla useita johtoja kiinni samanaikaisesti ja viesti lähetetään jokaista johtoa pitkin aina. (OpenJS Foundation k.)

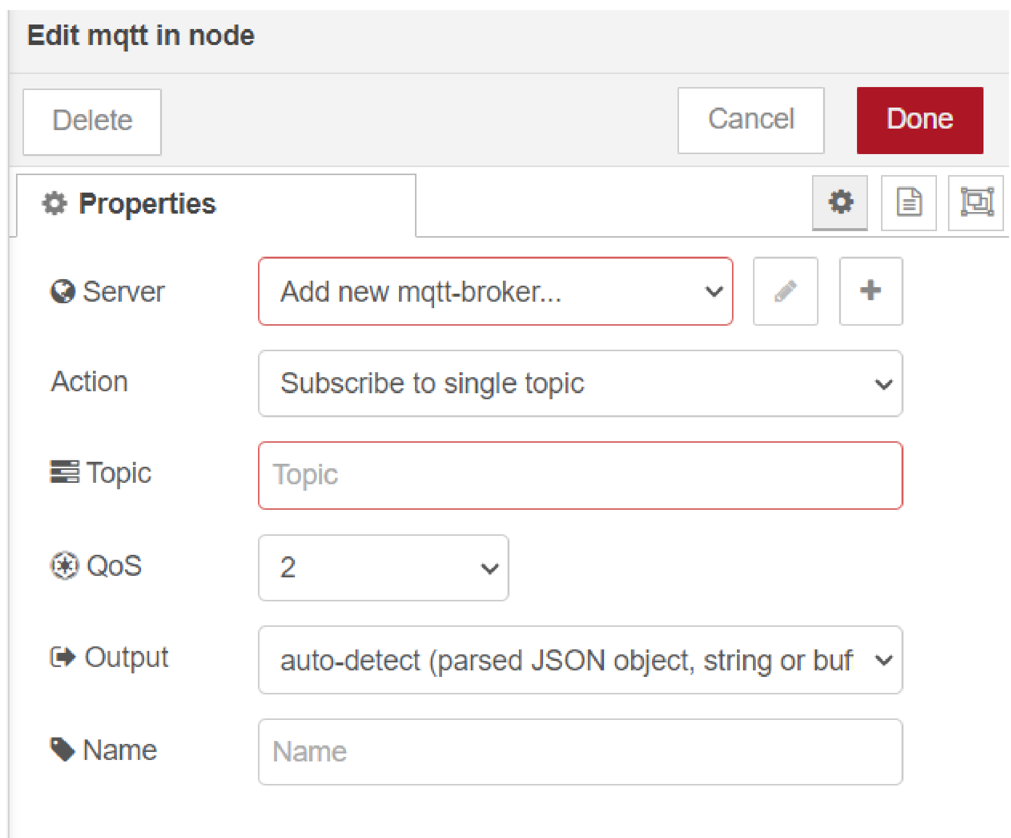
Node-RED:in eniten käytettyjä solmuja kutsutaan termillä ydinjoukko. Ydinjoukon solmuihin kuuluu:

- syöttösolmu
- debugaus-solmu
- funktiosolmu
- muutossolmu
- ehtosolmu

- mallinesolmu. (OpenJS Foundation o.)

Syöttösolmu toimii virran manuaalisena tai automaattisena aloittajana lähettämällä solmuun konfiguroidun viestin. Debugaus-solmu tulostaa saadut viestit debugausvälilehdelle. Funktio-solmu sallii JavaScript-koodin käyttämisen virrassa. Muutossolmulla voi asettaa, muuttaa, liikuttaa tai poistaa viestin tai viestin kontekstin. Ehtosolmu reitittää viestin eri haaroihin virrassa käyttämällä sääntöjä. Säännöt ovat asetetun arvon mukaan, tulevan viestisarjan mukaan, JSON-viestin mukaan ja muuten aina, jota käytetään, kun muut säännöt eivät täsmää. Mallinesolmu luo tekstin käyttämällä viestin arvoja mallin täyttämiseen (OpenJS Foundation o.)

FlowFuse (2024a) kertoo, että solmut esitetään käyttöliittymässä paneelin sisällä kategorioiden alla. Käytössä olevaa solmua konfiguroidaan kaksoisklikkaamalla solmun päältä. Solmujen konfiguraatioita on kahdenlaisia. Omat konfiguraatiot, jotka pätevät vain kyseisessä solmussa ja konfiguraatiosolmut, joita voidaan käyttää jokaisessa samanlaisessa solmussa. Kuvassa 14 on MQTT-solmun konfiguraatiovälilehti. MQTT-solmun konfiguraatiossa kaikki muut ovat solmun omia konfiguraatioita paitsi server osio, johon voi lisätä konfiguraatiosolmun, joka sisältää MQTT-välittäjän tiedot.



Kuva 14. MQTT-tilaajasolmun konfiguraatiovälilehti

Tiedostotasolla solmulla on kaksi osa HTML-käyttöliittymään ja JavaScript-taustajärjestelmään. JavaScript-tiedosto yhdistetään packeges.json tiedostoon, jota käytetään solmun asentamisessa. HTML on jaettu kolmeen osaan ja jokaista osaa ympäröi skripti tagi. Ensimmäinen osa on pääosa ja sisältää solmun kategorian, ulkoasun, nimen ja missä muodossa solmun käyttöliittymän parametrit ilmoitetaan taustajärjestelmälle. Tämä osio sisältää myös solmun ominaisuudet määriteltynä oletusobjektin alle. Seuraava osa muokkauspohja sisältää mitä HTML osuus tekee solmussa, kuten napit tai syöttökentät. Viimeinen osa on avustusteksti. Kun solmu on valittuna, teksti näkyy käyttöliittymän info paneelissa. (OpenJS Foundation h.) Kuvassa 15 on my-node solmun HTML-tiedosto. Kuvassa on esitettynä kaikki kolme HTML-tiedoston osuutta. Solmulle asetetaan syöttökenttä konfiguraatiovälilehdelle.

```
<script src="red/red.js"></script>
<script>
    RED.nodes.registerType('my-node', {
        category: 'function',
        color: '#a6bbcf',
        inputs: 1,
        outputs: 1,
        icon: 'node.png',
        label: function () {
            return this.name || 'My Node';
        },
        defaults: {
            userInput: { value: "" },
        }
    });
</script>
<script type="text/x-red" data-template-name="my-node">
    <div class="form-row">
        <label for="node-input-userInput"><i class="fa fa-tag"></i> User Input</label>
        <input type="text" id="node-input-userInput" placeholder="Enter your input" />
    </div>
</script>
<script type="text/x-red" data-help-name="my-node">
    <p>This node appends user input to the incoming message payload.</p>
</script>
```

Kuva 15. Solmun HTML-tiedosto

Node-RED solmun ajonaikaisen toiminnan määrittää sen JavaScript-taustajärjestelmä. Tiedoston alkuun määritetään moduulin julkaiseminen RED-objektilla. Objekti on pakollinen Node-RED toimintaa varten. Kaikki solmun toiminnot määritellään konstruktori-funktion sisällä, jossa solmu luo uuden instanssin ja ottaa vastaan konfiguraatiot. RED.nodes.createNode -funktio alustaa solmun. Alustamisen jälkeen solmuun liitetään sen toiminnallisuus, kuten tapahtumakuuntelija. Tapahtumakuuntelijaan asetetaan tapahtuma tyyppi, kuten tulo ja millainen viesti otetaan vastaan. Viestit kulkevat usein msg-nimisen objektin sisällä. Viesti

lähetetään eteenpäin kutsumalla Node-RED lähetysfunktiota. Suositeltu lähetettävien viestien muoto on msg.payload sisällä. (OpenJS Foundation i.)

Datan muodolla ei ole merkitystä, sillä lähettämisfunktio toimii aina, kun lähetettävä viesti ei ole tyhjä. Jos solmulla on useita uloslähtöpisteitä, on suositeltu käyttämään taulukkoa, jossa taulukon indeksit vastaavat lähtöporttien numeroita. Tämä mahdollistaa viestien ohjaamisen oikeisiin lähtöihin tehokkaasti. Ennen solmun uudelleenkäynnistämistä on tärkeää käyttää sulkutapahtumaa, jossa solmu siistii itsensä ja sulkee kaikki aktiiviset yhteydet, kuten sarjaportin kuuntelut. Tiedoston lopussa määritetään HTML-tiedostoon yhdistettävä nimi. (OpenJS Foundation i.) Kuvassa 16 on my-node solmun JavaScript-tiedosto. Kuvan tiedostossa yhdistetään käyttöliittymässä annetun syöttökentän arvo ja tietovirran viestin arvo.

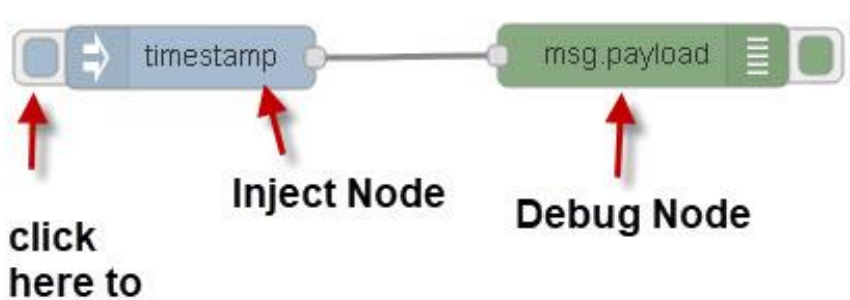
```
module.exports = function(RED) {
  function MyNode(config) {
      RED.nodes.createNode(this, config);
      const node = this;

      node.on('input', function(msg) {
          const userInput = config.userInput || "";
          if (msg.payload) {
              msg.payload += ' ' + userInput;
          } else {
              msg.payload = userInput;
          }
          node.send(msg);
      });
      node.on('close', function(done) {
        done();
    });
  }
  RED.nodes.registerType("my-node", MyNode);
}
```

Kuva 16. Solmun JavaScript-tiedosto

Packages.json tiedosto kertoo ohjelmistolle, mitä solmuja Node-RED käyttää osoittamalla tapahtuma-avaimen solmun JavaScript-tiedostoon. Yksi packages.json tiedosto voi sisältää useita solmuja. Packages.json sisälle merkitään myös solmuissa käytetyt kirjastot. Npm lataamis komennolla saadaan solmut käyttöön ja npm linkitys komennolla muutokset otetaan käyttöön. Komennot on ajettava packages.json tiedoston kanssa samassa kansiossa. (OpenJS Foundation d.)

5.4 Viestit ja virrat

Virta kuvaa yhtä solmujen yhdistelmää välilehdellä. Välilehdellä voi olla useita virtoja ja kaikki virtoja välilehdellä kuvastetaan sanalla virtavälilehti (OpenJS Foundation k). Kuvassa 17 on esitetty virta, jossa käytettään syöttö- ja debugaus-solmuja. Solmut on yhdistetty tietovirran avulla.



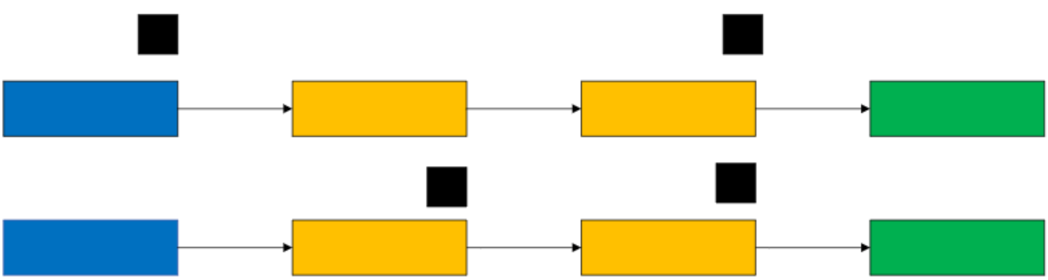
Kuva 17. Node-RED virta (Steve 2023)

Viestin näkyvyyttä hallitaan kontekstin avulla. Konteksteja on kolme erilaista. Solmu kontekstin arvo näkyy vain arvon käsittelijälle, virta arvo näkyy kaikille välilehdessä oleville solmuille ja globaali arvo näkyy kaikille solmuille välilehdestä riippumatta. Konteksti ilmoitetaan viestin nimen edessä. Oletusasetuksena kaikki kontekstit katoavat Node-RED uudelleen käynnistäessä. Arvot voi säilyttää konfiguroimalla asetuksiin oman tallennustavan. Kaikki Node-RED viestit lähetetään JavaScript-objektina ja viestiin viitataan sanalla msg solmuissa. (OpenJS Foundation k.)

Alussa Node-RED-virrat toimivat synkronisesti, mutta versiossa 1.0, joka julkaistiin vuonna 2019, virtojen toiminta muutettiin asynkroniseksi. Tämän muutoksen tarkoituksena oli parantaa Node-RED tehokkuutta ja suorituskykyä. Synkronisessa versiossa jokainen viesti kulki solmusta toiseen lineaarisesti, niin että edellinen viesti käsiteltiin kokonaan ennen uuden viestin lähettämistä. Tämä johti viiveisiin, koska uuden viestin käsittely alkoi vasta, kun edellinen viesti oli saavuttanut viimeisen solmun. Ainoa tapa nopeuttaa Node-RED toimintaa oli käyttää solmuja, jotka sisälsivät asynkronista koodia, kuten HTTP-pyyntösolmuja. Kun viesti kohtasi tällaisen solmun, solmu käsitteli viestin taustalla ja uusi viesti voitiin lähettää eteenpäin viiveettä. Tämä toi jonkin verran parannusta suorituskykyyn, mutta asynkronisen toiminnan vaikutus oli rajoitettu vain tiettyihin solmuihin. (O'Leary 2019.)

Vuonna 2019 siirtyminen kokonaan asynkroniseen käsittelyyn muutti merkittävästi viestien kulkua ja nopeutta. Nyt uusi viesti voidaan lähettää heti, kun edellinen viesti on siirtynyt seuraavaan solmuun riippumatta solmusta. Jos viestit kohtaavat solmun, jonka prosessissa kestää pitkään, muut viestit voivat jatkaa käsittelyä samaan aikaan. Näin koko virta ei enää pysähdy yhden viestin odottamisen vuoksi, mikä mahdollistaa tehokkaamman ja nopeamman viestien käsittelyn. (O'Leary 2019.)

Kuvassa 18 on esitetty viestien kulku synkronisessa ja asynkronisessa mallissa. Kuvassa suorakulmiot ovat solmuja ja neliöt viestejä. Synkroninen malli näkyy ylemmässä virrassa ja uusi viesti voidaan lähettää vasta kun edellinen on saapunut virran loppuun. Asynkroninen on alapuolella ja uusi viestit lähetetään eteenpäin aina kun edellinen on vaihtanut solmua.



Kuva 18. Synkroninen ja asynkroninen Node-RED

Node-RED viestit kulkevat JSON-muodossa ja voivat sisältää mitä vain JSON-sallittuja datamuotoja, kuten tekstiä, totuusarvoja tai matriiseja. Viestien JSON dataan lisätään aina _msgid niminen avain, jonka arvo on id numero, jota voidaan käyttää viestin reitin selvittämisessä. Node-RED viesteihin lisätään otsikko ja viestin data. Otsikkoa voi käyttää viestien erottelemisessa. (OpenJS Foundation d.)

OpenJS Foundation (q) suosittelee virtojen rakentamisessa käyttämään tiettyjä käytäntöjä. Virrat suositellaan jakamaan saman toiminnon tai ympäristön perusteella. Tämä pitää virrat luettavina ja välttää välilehtien täyttymistä useista virroista. Virroissa on usein uudelleen käytettäviä solmuosuuksia, joita käytetään useassa paikassa samaan aikaan. Virrat kannattaa tällöin jakaa joko linkkisolmuja hyödyntämällä tai alisolmujen avulla.

Linkkisolmut mahdollistavat virtojen viestien kulkemisen toisiin välilehtiin sisääntulo ja lähtö linkkisolmuilla, joiden välille lisätään virtuaalijohto. Linkit toimivat aina aloitus tai lopetus solmuina ja ne voidaan yhdistää useampaan kuin yhteen muuhun linkkisolmuun. Näin viestejä voi välittää useisiin muihin virtoihin tai antaa useiden virtojen välittää viestejä yhteen virtaan. (OpenJS Foundation q.)

Alivirtojen avulla voi luoda palettiin uuden solmun, jonka sisällä on oma virta. Alivirran voi aina lisätä minne vain virran sisälle. Linkitetyn virran ja alivirran ero on, että alivirta on aina oma instanssinsa, kun taas linkkivirta on sama instanssi jokaiselle yhteydelle. Jokaiseen virtaan suositellaan lisäämään kommenttisolmu, jossa kerrotaan virran toiminto. (OpenJS Foundation q.)

6 Digitaalitulo ja -lähtöportti

6.1 Digitaalitulo ja -lähtöportti yleisesti

DIO (Digital Input/Output) on tietokoneissa oleva digitaalitulo ja -lähtöportti, joka voi vastaanottaa ja lähettää digitaalisia signaaleja IoT-laitteisiin tai antureihin ja saada ne toimimaan halutulla tavalla. Digitaalitulot havaitsevat ulkoisten laitteiden signaalit ja digitaalilähdöt lähettävät signaaleja muiden komponenttien aktivoimiseksi. (Premio Inc 2022.) DIO-järjestelmiä käsitellään viestintäprotokollilla, kuten I2C (LabJack 2022).

6.2 I2C-perusteet ja toiminta

I2C-protokolla on suunniteltu useiden piirien väliseen kommunikaatioon ja lyhyen etäisyyden viestintään laitteen sisällä. Protokolla kehitettiin alun perin Philipsin siruille vuonna 1982. Alkuperäisessä versiossa rajoitettiin nopeutta ja osoitteita. Vuonna 1992 julkaistu versio laajensi nopeusaluetta ja osoitteiden määrää. Tämä paransi protokollan joustavuutta ja soveltuvuutta erilaisiin käyttötarkoituksiin. I2C on saavuttanut suosiota monissa sulautetuissa järjestelmissä ja elektroniikkalaitteissa sen yksinkertaisuuden ja tehokkuuden ansiosta. (Sfuptownmaker.)

I2C-protokolla mahdollistaa tehokkaan kaksisuuntaisen ohjauksen ja viestinnän useiden laitteiden välillä. I2C-kommunikaatiossa käytetään osoitteita, jotka mahdollistavat useiden laitteiden ohjaamisen isäntälaitteelta. Tämä mahdollistaa monipuolisen ja joustavan järjestelmän rakentamisen, jossa erilaiset laitteet voivat kommunikoida keskenään ja toimia yhteistyössä isäntälaitteen kanssa. I2C:n osoitejärjestelmä tarjoaa selkeän tavan erottaa ja hallita eri laitteita. (Pini 2020.)

6.3 Arkkitehtuuri

I2C-protokolla perustuu isäntä/orjahierarkiaan, jossa vähintään yksi laite toimii isäntänä ja muut laitteet ovat orjia. Viestintä tapahtuu sarjadatalinjan ja sarjakellolinjan avulla. Kun datansiirtoa ei ole, väylä on lepotilassa, jolloin se on loogisessa tilassa 1. (Afzal 2016.) I2C:n datansiirtonopeudet vaihtelevat eri tilojen välillä. Standarditila toimii 100 kbit/s, nopea tila toimii 400 kbit/s, nopea tila plus toimii 1 Mbit/s, ja todella nopea tila toimii 3,4 Mbit/s. (Pini 2020.) I2C-protokollan laitekapasiteetin määrittelee osoitteiden määrä. Maksimi osoitteiden määrä on 128. Data siirretään tavuina, tällöin viestin pienin yksikkö on 8-bittinen tavu. Viestit koostuvat:

- aloitusbitistä
- lopetusbitistä

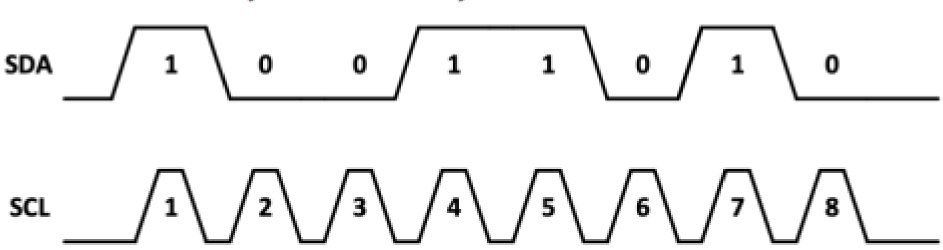
- laiteosoitteesta
- kirjoittamisbitistä
- lukemisbitistä
- datasta
- hyväksytty- tai hylättykuittausbitistä. (Afzal 2016.)

6.3.1 Isäntä ja orja

I2C-protokollan isäntälaitteena toimii pääasiassa mikro-ohjain tai prosessori, ja se kontrolloi väylää lähettämällä kellosignaaleja. Isäntälaite vastaa väylän hallinnasta, aloittaen datansiirron ja ohjaten tiedonsiirtoa. Orjalaitteet eivät voi itsenäisesti aloittaa datan siirtoa. Niiden tehtävä on vastata isännän pyyntöihin ja toimia niiden mukaisesti, kirjoittaen tai lukien dataa rekistereistään isännän ohjeiden mukaisesti. Tällainen hierarkia mahdollistaa tehokkaan ja hallitun tiedonsiirron useiden laitteiden välillä, jossa isäntälaitteella on päärooli tiedonsiirron aloittajana ja ohjaajana. Orjalaitteet puolestaan toimivat passiivisesti odottaen isäntälaitteen ohjeita ja vastaten niihin tarpeen mukaan. (Afzal 2016.)

6.3.2 Datan siirto

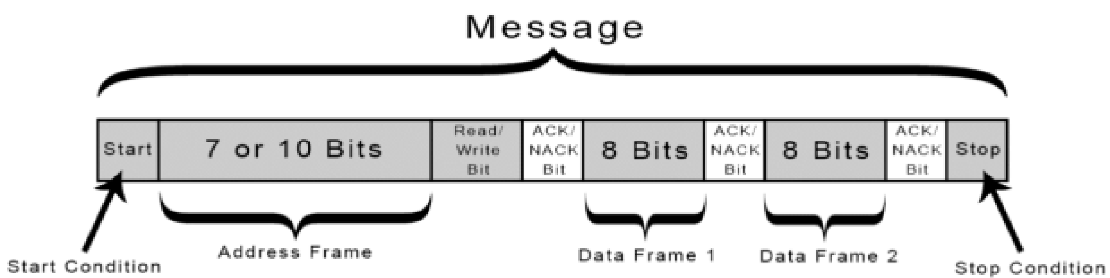
Datan siirrossa käytetään kahta eri linjaa, datalinjaa ja kellolinjaa. Nollabitti ja yksibitti erotetaan datassa sillä, että molemmat linjat näyttävät samanaikaisesti olevan ylhäällä eli yksi asennossa. Tällöin datassa luetaan looginen yksi muussa tapauksessa looginen nolla. Kun isäntälaitteelta halutaan siirtää dataa orjalaitteelle, se aloittaa kommunikaation lähettämällä aloitusbitin. Tämän jälkeen isäntälaite valitsee halutun orjalaitteen rekisterin ja lähettää datan laitteelle tavuina käyttämällä laskevaa reunaa. Lopuksi isäntälaitteen tulee lähettää lopetusbitti, jolla kommunikaatio päätetään. Kun taas isäntälaite haluaa lukea orjalaitteelta dataa, toiminta tapahtuu samalla tavalla, paitsi että isäntälaitteen ei tarvitse lähettää dataa orjalaitteelle. Sen sijaan orjalaitteelta lähetetään datan isäntälaitteelle ja tämä luetaan nousevalla reunalla. Kuvassa 19 kuvataan datan siirtoa, kun sarjadatalinjan ja sarjakellolinjan ollessa molemmat loogisessa yksi tilassa ja datan bitti luetaan loogisena yhtenä. Jos taas data tai kellolinja on looginen nolla, data bitti luetaan loogisena nollana. Tämä kommunikaatiotapa on olennainen osa I2C-protokollan toimintaa ja mahdollistaa tehokkaan ja luotettavan tiedonsiirron isäntälaitteen ja orjalaitteiden välillä. (ensatellite.)



Kuva 19. 10011010 datan siirto (ensatellite)

6.4 Rekisteri ja viestin rakenne

I2C-protokollan rekisterien lukemis- ja kirjoitusprotokollaan sisältyy useita olennaisia viestiehtoja. Viestin alussa ja lopussa on aloitus- ja lopetusbitit, jotka määrittävät viestin rajat. Aloitusbitti toistetaan aina ennen varsinaisen datan lähettämistä. Viestiin sisältyy myös osoitetavut, jotka ovat 7–10 bitin pituisia uniikkeja orjalaitteen osoitteita. Datan siirto aloitetaan aina nousevaa reunaa käyttämällä. (Afzal 2016.) Viestissä tulee aina yksi bitti, joka merkitsee, tuleeko viesti olemaan lukemista vai kirjoitusta varten. Lukemista varten bitti on nolla ja kirjoittamista varten yksi. Jokaisen tavun jälkeen tulee hyväksytty- tai hylättykuittausbitti. Mikäli tiedonsiirto on onnistunut, lähetetään hyväksymiskuittausbitti, ilmoittaen että viestin vastaanotto oli onnistunut. Viestiä havainnollistetaan kuvassa 20. (Cambell.) Kuva esittää I2C-viestin sisällön ja missä järjestyksessä viestin arvot ilmoitetaan.



Kuva 20. I2C viestin rakenne lukemista ja kirjoittamista varten (Cambell)

6.5 I2C työkalut Linuxilla

I2C-rekistereitä voidaan lukea Linuxin komentorivin kautta käyttämällä I2C-tools ohjelmaa. Tämä työkalupaketti tarjoaa useita hyödyllisiä komentoja, joista käytetyimpiä ovat i2cdetect, i2cdump, i2cget ja i2cset. I2C-toolsin pystyy asentamaan Linux-laitteelle, käyttämällä komentoa apt-get install i2c-tools. Tämä lataa ja asentaa I2C-tools-ohjelmiston, jonka avulla voit hallita I2C-väyliä ja lukea laitteiden rekistereitä. I2C-tools luo yleensä yhdeksän väylää, joita laitteet voivat käyttää. Näiden väylien olemassaolon voi tarkistaa /dev/ kansiosta, jossa on tiedostoja i2c-0, i2c-1… i2c-9. Nämä tiedostot vastaavat I2C-väyliä, joiden kautta on

mahdollista kommunikoida liitettyjen laitteiden kanssa. Kuvan 21 komennon i2cdetect avulla voidaan listata kaikki tiedot I2C väylästä. (Rathore 2022.)

```
:/dev$ sudo i2cdetect -y
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:                         08 -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: 30 31 -- -- 34 35 36 -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- 4a -- -- -- -- --
50: UU -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Kuva 21. I2cdetect komento ja sen tuloste

Lisäksi komentoa i2cdump voidaan käyttää laiterekistereiden koko datan tarkasteluun, kuten kuvassa 22 on esitetty. Tämä antavat kattavan yleiskuvan väylään yhdistettyjen laitteiden tiedoista. (Rathore 2022.)

```
:/dev$ sudo i2cdump -y
No size specified (using byte-data access)
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
00: 00 00 00 03 00 02 00 00 00 00 00 00 00 00 00 10    ...?.?.........?
10: 03 07 00 01 00 ff 00 00 00 00 00 00 00 00 00 00    ??.?............
20: ff 00 00 01 00 41 00 00 00 a1 00 10 00 00 20 00    ...?.A...?.?.. .
30: 00 00 00 00 00 00 19 00 73 3f 09 77 00 00 00 00    ......?.s??w....
40: 32 08 00 3e 00 00 39 17 5a 0b 32 09 32 0d 55 0b    2?.>..9?Z?2?2?U?
50: 32 0a 00 00 28 00 00 00 00 00 00 ff 00 00 00 00    2?..(...........
60: 03 00 00 14 00 00 03 00 00 00 03 00 0a 00 00 1e    ?..?..?...?.?..?
70: 00 00 1e 00 00 ff 00 00 00 04 00 05 00 00 ff 00    ..?......?.?....
80: 00 04 ff 00 03 00 00 00 00 00 00 00 00 00 00 00    .?..?...........
90: 00 01 00 03 00 00 00 00 00 00 00 00 00 00 00 00    .?.?............
a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
b0: b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf    ????????????????
c0: c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf    ????????????????
d0: 38 52 76 d6 54 2c 78 00 d8 d9 da db dc dd de df    8Rv?T,x.????????
e0: 0d e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef    ????????????????
f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
```

Kuva 22. I2cdump komento ja sen tuloste

Yksittäisen väylän rekisterin tietoja voi tutkia käyttämällä komentoa i2cget. Tätä käytetään tarkastamaan digitaalitulon tai -lähdön arvot kuten kuvassa 23 esitetään. (Rathore 2022.)

Kuva 23. I2cget komento ja sen tuloste

Käyttämällä i2cset komentoa isäntälaitteella voi kirjoittaa halutun arvon rekisteriin. Komento käyttää samoja parametreja kuin hakukomento, mutta loppuun lisätään asetettava arvo

kuten kuvassa 24 on asetettu. Tämä prosessi on tärkeä, koska se antaa isäntälaitteelle mahdollisuuden hallita orjalaitetta ja sen toimintoja. Nämä komentoesimerkit osoittavat, miten I2C-toolsia voidaan käyttää Linux-käyttöjärjestelmässä väylien ja laitteiden hallintaan. (Rathore 2022.)

Kuva 24. I2cset komento

7 Sarjaliikenneportti

7.1 Sarjaliikenneportin toiminta

Sarjaliikenneportti lähettää tavuja laitteelta toiselle. Näiden tavujen avulla siirretään merkki viestejä laitteiden välillä. Viestien lähettämiseen käytetään linjoja TX ja RX. (Hutasu 2017.) Sarjaliikenne kommunikointi standardeja ovat RS-232, RS-422 ja RS-485. RS-232 on yleisin standardi. RS-422 on suunniteltu pidempiin siirtoetäisyyksiin ja RS-485 on suunniteltu useiden laitteiden kommunikaatiolle. (Keeney 2020.)

Sarjaliikenne tarvitsee parametreja. Näiden parametrien täytyy täsmätä lähettävässä ja vastaanottavassa laitteessa, jotta viestit ymmärretään oikein. Viestit koostuvat useista merkeistä. Merkki sisältää 5–9 databittiä. 8 bittiä on yleisin määrä, koska 8 bittiä vastaa yhtä tavua. Merkin virheentarkistusmenetelmän käytetään pariteettibittiä. Pariteetti tarkistaa, onko merkissä ykkösiä parillinen vai pariton määrä. Merkin alkuun ja loppuun lisätään aloitus- ja lopetusbitit, jotka kertovat milloin merkki alkaa ja loppuu. Sarjaliikenteessä on erilaisia tiedonsiirtonopeuksia. (Hutasu 2017.) Kuvassa 25 on esitetty sarjaliikenne merkkien formaatti. Kuva alkaa aloitusbitistä, jonka jälkeen tulee data-, pariteetti- ja lopuksi lopetusbitti.

Kuva 25. Vietin rakenne (Jimblom)

Punnitus ympäristössä yleisin tiedonsiirtonopeus on 9600bit/s. Srinam (2023) kertoo, että ASCII:ta (American Standard Code for Information Interchange) voidaan käyttää sarjaliikenne viestien muuttamisessa oikeiksi viesteiksi.

7.2 ASCII-perusteet

ASCII on standardi tietojen vaihtoon tietokoneiden välillä. ASCII toimii amerikkalaisen merkkijärjestelmän perusmerkeillä. Järjestelmässä jokaisella merkillä on oma numeronsa välillä 0–127. Esimerkiksi iso A-kirjain vastaa numeroa 65 ja pieni a numeroa 97. Isot ja pienet kirjaimet erottavat toisistaan aina 32 yksikön ero. ASCII käyttää seitsemän bittistä koodausjärjestelmää, mikä tarkoittaa, että sillä voidaan esittää yhteensä 128 eri merkkiä. (Srinam 2023.)

Ennen ASCII-järjestelmää tietokoneilla oli omat, toisistaan poikkeavat tapansa esittää merkkejä. Tämä aiheutti suuria haasteita tiedonvaihdossa eri tietokoneiden välillä, koska laitteet eivät kääntäneet merkkejä oikein. 1960-luvulla, kun tietokoneet alkoivat yleistyä,

ASCII kehitettiin yhtenäiseksi ja universaaliksi merkkien koodausjärjestelmäksi, mikä mahdollisti sujuvamman tiedonsiirron ja yhteensopivuuden. Järjestelmä varmistaa, että tietty merkki tarkoittaa samaa kaikilla tietokoneilla, mikä on ollut tärkeää digitaalisen viestinnän kehitykselle. (Srinam 2023.) Kuvassa 26 on esitettynä tekstin muuttaminen ASCII muotoon ja binäärimuotoon.

```
Character: M | ASCII: 77 | Binary: 1001101
Character: o | ASCII: 111 | Binary: 1101111
Character: i | ASCII: 105 | Binary: 1101001
Character:   | ASCII: 32 | Binary: 0100000
Character: l | ASCII: 108 | Binary: 1101100
Character: u | ASCII: 117 | Binary: 1110101
Character: k | ASCII: 107 | Binary: 1101011
Character: i | ASCII: 105 | Binary: 1101001
Character: a | ASCII: 97 | Binary: 1100001
Character: ! | ASCII: 33 | Binary: 0100001
```

Kuvassa 26. Moi lukia! Kääntämin ASCII ja binäärimuotoon

Vaikka ASCII on ollut merkittävä edistysaskel, sen yksinkertaisuudesta on ajan myötä paljastunut rajoituksia. Koska järjestelmässä on vain 128 merkkiä, se ei kykene käsittelemään monia englannin ulkopuolisia kieliä tai erikoismerkkejä. Tämän puutteen vuoksi kehitettiin laajennettu ASCII, joka lisää kahdeksannen bitin ja näin mahdollistaa 256 merkin käytön, mukaan lukien aksenttimerkit ja erikoismerkit. ASCII on yhä keskeinen osa digitaalista viestintää, mutta sen heikkoudet ovat avanneet tien monipuolisemmille merkistöille, kuten Unicode, joka tukee laajempaa joukkoa kieliä ja symboleja. (Injosoft AB b.)

7.3 Laajennettu ASCII

Laajennettu ASCII kehitettiin mahdollistamaan eri maiden erikoismerkkien, kuten suomenkielisten Ä ja Ö, käytön. Laajennetussa ASCII-järjestelmässä merkkien määrä tuplattiin, jolloin niitä on yhteensä 256. Tämä saavutettiin muuttamalla ASCII kahdeksan bittiseksi järjestelmäksi. On tärkeää huomata, että laajennettu ASCII ei ole yksi yhtenäinen standardi, vaan kokoelma erilaisia merkkikoodausjärjestelmiä, jotka määrittävät eri merkkejä koodiväliin 128–255. Tämä tarkoittaa, että eri laajennetut ASCII-versiot voivat käyttää samoja koodeja eri merkeille. Tunnetuimpia näistä järjestelmistä ovat ISO-8859-1 ja Windows-1252, joita käytetään laajalti länsimaisissa kielissä. Laajennetun ASCII avulla voitiin käsitellä kielten erikoismerkkejä, mutta se ei riittänyt kattamaan kaikkia maailman kieliä. Nykyisin

käytetään laajempia merkistöjä, kuten Unicodea, joka tukee satojatuhansia merkkejä. (Injosoft AB a.)

7.4 ASCII-taulukko

ASCII-taulukko on järjestelmä, joka esittää kaikki merkit ja niiden vastaavat ASCII-koodit mahdollistaen käyttäjän tarkistaa kunkin merkin numerokoodin. Perus 128-merkkinen ASCII-taulukko jaetaan kahteen pääluokkaan: ei-tulostettavat ja tulostettavat merkit. Ei-tulostettavat merkit kattavat ASCII-koodit 0–31 sekä koodin 127. Näitä merkkejä käytettiin alun perin tietokonejärjestelmissä laitteiden ja ohjelmistojen ohjaamiseen. Ne eivät näy käyttäjälle, vaan toimivat ohjauskomentoina. Kuten merkkijonon päättymismerkki nolla ja äänimerkki seitsemän, joka voi antaa laitteelle käskyn tuottaa äänen. Tulostettavat merkit sijaitsevat koodeissa 32–126. Nämä merkit näkyvät käyttäjälle. Merkkeihin kuuluvat isot ja pienet kirjaimet, numerot ja erilaiset symbolit, kuten välimerkit ja erikoismerkit. ASCII-taulukko mahdollistaa tietokoneiden ja ihmisten välisen viestinnän standardoidulla tavalla, jossa jokaisella merkillä on selkeä, yhdenmukainen numeerinen esitys. Kuvassa 27 on esitetty normaali seitsemän bitin 0–127 ASCII-taulukko. (Hymel S.)

Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	0	000	NULL	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SoH	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	SoTxt	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	EoTxt	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EoT	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	Enq	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	Ack	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	Bell	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	Bsp	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	HTab	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LFeed	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VTab	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FFeed	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SOut	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SIn	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAck	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	Syn	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	EoTB	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	Can	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EoM	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	Sub	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	Esc	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FSep	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GSep	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RSep	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	USep	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		Delete

charstable.com

Kuva 27. 0–127 seitsemän bittinen ASCII-taulukko (NetCorp)

8 Toteutus

8.1 Laitteet

Vaaoilla on loputtomasti erilaisia oheislaitteita, joita hyödynnetään vaa'an toiminnassa tai punnitusten tekemisessä. Yleisimpiä oheislaitteita punnitusalueilla ovat liikennevalot, lisänäytöt ja kulunohjauslaitteet, kuten puomit ja portit. Liikennevaloilla ilmoitetaan vaa'an käyttäjälle, milloin vaaka on vapaa käytettäväksi ja milloin punnitus on valmis. Kulunohjauslaitteita käytetään lisävarmisteena ajoneuvojen kulkemiselle. Molempia ohjataan digitaalilähdöillä yhdyskäytävän kautta. Liikennevaloissa ohjataan valon vaihtuminen ja kulunohjauslaitteilla ohjataan avautuminen. Kulunohjauslaitteille annetaan digitaalilähdöllä signaali, joka toimii avautumiskomentona. Lähtöä pidetään aktiivisena yleisesti viisi sekuntia, jonka jälkeen lähtö palautetaan ei aktiiviseen tilaan. Kulunohjauslaitteet toteuttavat sulkeutumisen itse. Kulunohjauslaitteissa on yleisesti liikkeentunnistuskamera, joka sulkee laitteen, kun auto on poistunut vaa'alta. Liikennevalo väriä kontrolloidaan digitaalilähdöllä. Yhtä oheislaitetta ohjaa yksi yhdyskäytävän digitaalilähtö.

Opinnäytetyössä käytetään kahta erilaista yhdyskäytävää. Yhdyskäytävät eroavat tehokuudessa ja IO-pisteiden määrässä. Molemmilla yhdyskäytävillä on omat digitaalirekisterit oheislaitteiden ohjaamista varten. Rekisterit selviävät I2C-tools komentojen avulla. Lisänäyttö on vaakaan tai yhdyskäytävään yhdistettävä näyttö, joka näyttää jokaisen vaakasillan yhteispainon koko ajan. Lisänäyttöä tarvitaan, kun punnitussovellus ei ole käytettävissä ja ajoneuvon kokonaispainoa tarvitaan. Lisänäyttö liitetään yhdyskäytävän sarjaporttiin. Lisänäytöt käyttävät ASCII-järjestelmää paino viestin vastaanottamisessa

8.2 Docker ympäristö

Node-RED käynnistyy Docker-kontissa. Kontituksella pidetään tuotanto ja kehitysympäristö samanlaisena. Docker-kontti alustetaan Kuvan 28 Dockerfilen avulla. Kuva alustetaan node.js 22.2.0-slim versio aloituskuvalla, jonka sisälle asennetaan riippuvuudet ja Node-RED versio 4.0.2, i2c-bus ja AWS IoT SDK:n JavaScript-versio. Docker-kuva alustetaan paikallisen tietokoneen Node-RED käyttäjädatatiedostoilla, jotka kopioidaan Docker-kontin sisälle kopiointi komennolla. Kontin portti 1880 uudelleenohjataan näkymään yhdyskäytävän portissa 1880. Docker-kuvassa asetetaan kontille aloituskomento, jossa ladataan package.json tiedoston paketit ja käynnistetään Node-RED.

```
FROM node:22.2.0-slim

COPY ./mounts/ /usr/src/node-red

WORKDIR /usr/src/node-red

USER root

RUN apt-get update && \
    apt-get install -y cmake && \
    apt-get install -y g++ && \
    apt-get install -y jq && \
    apt-get install -y make && \
    apt-get install -y python3 && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* && \
    npm install -g --unsafe-perm node-red@4.0.2 && \
    npm install --save i2c-bus aws-iot-device-sdk-v2 aws-greengrass-ipc-sdk-js aws-greengrass-common-js axios && \
    npm link

EXPOSE 1880

CMD ["sh", "-c", "npm install && node-red"]
```

Kuva 28. Dockerfile Node-RED sovellukselle.

Paikallisesti Node-RED käynnistetään kuvan 29 docker-compose.yml tiedostolla. Docker-kontille asetetaan privileged-komento. Komento sallii yhdyskäytävän digitaali- ja sarjaportin kontrolloimisen kontista. Kontti ottaa käyttöön portin numeron 1880, kuten Docker-kuvassa ilmoitettiin. Kontille asetetaan ympäristömuuttujia AWS IPC kommunikaatiota varten IoT-laitteen nimi ja IPC-socket. Paikallisessa kehityksessä ilmoitetaan NODE_ENV muuttujalla, että käytetään paikallista ympäristöä. Volyymeihin asetetaan Node-RED käyttäjädatatiedostot ja paikallisesti isäntälaitteen IPC-kansio. Tämä tehdään, koska IPC-kansiosta haetaan IPC-socketin salainen tunnus. Tämä tunnus tulee automaattisesti Greengrass lähetyksessä. Käyttäjädatatiedostot säilytetään volyymin sisällä, koska käytössä olevia virtoja ei haluta kadottaa kontin uusissa versioissa. Ohjelmistokehityksessä kontin paikallinen käynnistäminen nopeuttaa ohjelmistokehitystä.

```yaml
node_red:
  build:
    context: components/node_red/
  container_name: iot_node_red
  environment:
    - NODE_ENV=local
    - AWS_IOT_THING_NAME=${THINGNAME}
    - AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT=/ipc/ipc.socket
  depends_on:
    - greengrass
  volumes:
    - ipc-socket:/ipc/
    - ./components/node_red/mounts/:/usr/src/node-red
    - node-red-mounts:/components/node_red/mounts
  ports:
    - 1880:1880
  privileged: true
```

Kuva 29. paikallisen kehityksen docker-compose.yml tiedosto

8.3 Pilvipalveluiden Integrointi

Tuotantoympäristössä Node-RED Docker-kuva lähetetään Greengrassin avulla yhdyskäytävälle ja kuva käynnistetään reseptitiedoston Docker-ajokomennolla. Greengrass valvoo kontin elossa pysymistä ja käynnistää kontin uudelleen, jos se sammuu.

Docker-kuva tallennetaan AWS ECR tietokantaan, josta se lähetetään Greengrassin avulla yhdyskäytävälle. Docker-kuvien lähettämisessä tietokantaan käytetään JSON-muodossa olevaa GDK-tiedostoa, shell-skriptiä ja reseptiä. Kuvan 30 GDK-tiedostossa kerrotaan mitä tehdään GDK rakennus ja julkaisu komennoilla. Rakennus komennossa ajetaan shell-skripti, jossa annetaan node_red ja julkaisuversio parametreina. Julkaisu komennossa kerrotaan ECR-ämpäri ja AWS-alue. Shell-skripti yhdistää AWS IoT-palvelimelle ja rakentaa Docker-kuvan, joka lähetetään AWS ECR-ämpäriin.

```
{
  "component": {
    "node_red": {
      "author": "Eetu Leppinen",
      "version": "NEXT_PATCH",
      "build": {
        "build_system": "custom",
        "custom_build_command": [
          "bash",
          "build-custom.sh",
          "node_red",
          "NEXT_PATCH"
        ]
      },
      "publish": {
        "bucket": "                                   ",
        "region": "            "
      }
    }
  },
  "gdk_version": "1.1.0"
}
```

Kuva 30. GRK-tiedosto rakennus ja julkaisukomennoilla AWS-pilvipalveluun

Resepti kopioidaan greengrass-build kansioon, joka lähetetään AWS IoT-palvelimelle julkaisu komennolla. Reseptissä asetetaan AWS IoT Core ja ECR-avaimet, jotka kertovat minne julkaistaan. Reseptissä asetetaan oikeudet IoT Core ja IPC-kommunikaatiolle. Kuvan 31 esittämä Docker-ajokomento asetetaan reseptitiedoston loppuun. Docker-ajokomento tekee samat asiat kuin docker-compose paikallisessa kehityksessä.

```
docker run --rm --privileged \
-p 1880:1880 \
-v $AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT:$AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT \
-v node-red-mounts:/usr/src/node-red \
-e SVCUID \
-e AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT \
-e AWS_IOT_THING_NAME \
--name $CONTAINER_NAME <IMAGE_URL_PLACEHOLDER>
```

Kuva 31. Reseptin Docker-ajokomento

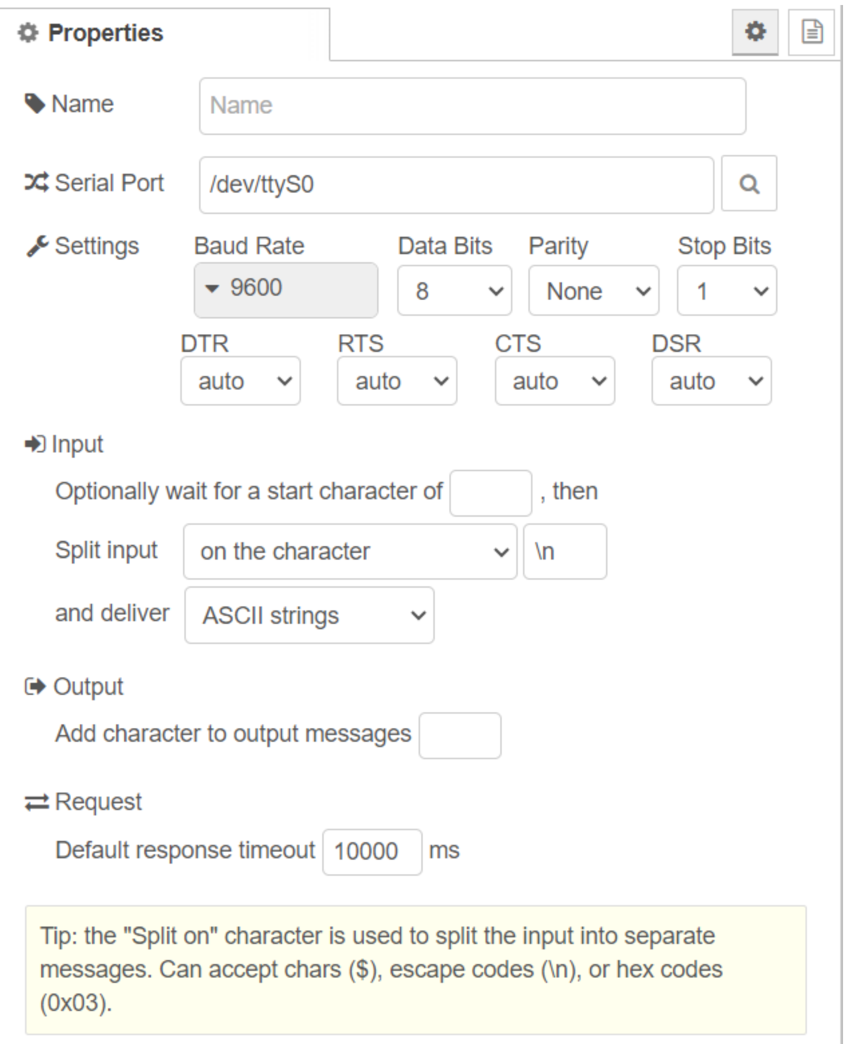
8.4 Käytetyt solmut

Opinnäytetyössä käytetään monipuolisesti Node-RED solmuja, ladattuja solmuja sekä itsetehtyjä solmuja. Node-RED valmiit solmut eivät riittäneet virtojen rakentamiseen, joten turvaudutaan muiden ja itsetehtyihin solmuihin.

8.4.1 Käytetyt Node-RED valmiit ja ladatut solmut

Osa virroista aloitetaan käyttämällä syöttösolmuja. Datan käsittelyssä käytetään ehto, muutos- ja funktiosolmuja. Viestejä lähetetään ja otetaan vastaan muilta virroille linkkisolmujen avulla ja viestejä ajoitetaan hidaste- ja laukaisinsolmuilla. Molemmilla ajoitussolmuilla voi muuttaa viestien lähetys aikaa. Virrat kommentoidaan kommenttisolmuilla. Sarjaliikenneportteja varten käytetään ladattavaa solmupakettia node-red-node-serialport.

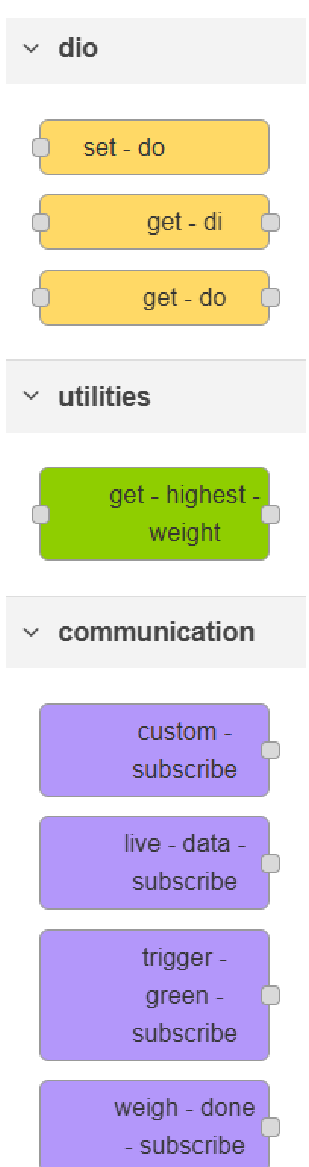
Solmupaketti mahdollistaa sarjaliikenneportin käyttämisen erilaisilla solmuilla (Dceejay & Knolleary 2024). Solmupaketista käytetään lähetys sarjaliikennesolmua. Solmulla lähetään haluttuun sarjaliikenneporttiin dataa. Solmu vaatii konfiguraatiosolmun sarjaliikenneportin toimintaan, joka on esitettynä kuvassa 32. Solmulle konfiguroidaan 9600 tiedonsiirtonopeus ja /dev/ttyS0 sarjaliikenneportti. Muuten konfiguraatio pidetään oletusasetuksissa.



Kuva 32. Sarjaliikenneportin konfiguraatiosolmu

8.4.2 Omien Node-RED solmujen rakentaminen

Omatekoisia solmuja tarvitaan datojen käsittelyä varten, IPC-kommunikaatiota varten ja digitaalitulo ja -lähtöjen kontrolloimista varten. Solmut asetettiin omiin kategorioihin käyttöliittymässä, kuten kuvassa 33 on asetettu. Kuvassa omatekoisia DIO-solmuja, IPC-solmuja ja aputoimintosolmuja.



Kuva 33. Omatekoiset solmut

Yhdellä vaa'alla voi olla useita vaakasiltoja, joista jokaiselta luetaan painolukemat. Liikennevaloja varten tarkkaillaan suurinta vaa'an painolukemaa, jotta valoa voidaan vaihtaa painolukeman ylityttyä. Painolukemat tarjotaan solmuille JSON-muodossa ja Node-RED ei tarjoa solmua monen eri JSON-arvon lukemiselle ja vertailulle. Tätä varten omatekoisen solmun rakentaminen oli tarpeellista.

Solmun nimi on get_highest_weight. Get_highest_weight solmun JavaScript-tiedosto käy lävitse jokaisen vaakasillan painolukeman JSON-datasta ja etsii suurimman. Tämä esitetään kuvassa 34. Suurin painolukema asetetaan msg.payload avaimen arvoksi ja lähetetään solmusta eteenpäin. Solmu vertailee kokonaispainoa, nettopainoa ja taarapainoa.

```
/**
 * Checks all current_weight and values from json
 * then checks what is the max_value and sends that out
 * @param {JSON} msg value passed from another node
 */
module.exports = function(RED) {
    function GetHighestWeight(config) {
        RED.nodes.createNode(this, config);
        const node = this;

        node.on('input', function(msg) {
            let values = [];
            let payload = msg.payload;

            payload.forEach(scale => {
                if (scale.current_weight) {
                    if (scale.current_weight.net_weight !== null) values.push(scale.current_weight.net_weight);
                    if (scale.current_weight.gross_weight !== null) values.push(scale.current_weight.gross_weight);
                    if (scale.current_weight.tare_weight !== null) values.push(scale.current_weight.tare_weight);
                }
            });

            let maxValue = null;
            values.forEach(value => {
                if (typeof value === 'number' && (maxValue === null || value > maxValue)) {
                    maxValue = value;
                }
            });
            node.send({ payload: maxValue });
        });
    }
    RED.nodes.registerType("get-highest-weight", GetHighestWeight);
}
```

Kuva 34. get-highet-weight solmun JavaScript-tiedosto

Kuvan 35 get_highest_weight HTML-tiedoston pääosassa lisätään Node-RED solmupalettiin apuohjelmat valikon alle ja solmulle asetetaan yksi datan tulo- ja lähtöportti. Solmun HTML-muokkauspohjassa on teksti, joka kertoo mitä solmu tekee.

```
<script type="text/javascript">
    RED.nodes.registerType('get-highest-weight', {
        category: 'utilities',
        color: '#8fce00',
        defaults: {
            topic: { value: "" }
        },
        inputs: 1,
        outputs: 1,
        label: function() {
            return this.name || "get-highest-weight";
        }
    });
</script>
<script type="text/x-red" data-template-name="get-highest-weight">
    <div class="form-row">
        <label>Static Text</label>
        <div id="static-text">Get highest scale weight.</div>
    </div>
</script>
```

Kuva 35. HTML-tiedosto get_highest_weight funktiolle

Kommunikaatiota varten Node-RED ei tarjoa valmista IPC-solmua, joten omien solmujen tekeminen on tarpeellista. Solmut käyttävät AWS IoT SDK JavaScript-kirjastoa. Solmuja käytetään vain laitteen sisäisessä kommunikaatiossa Greengrass IPC-kommunikaation avulla. Solmuja on neljä kappaletta. Kolmella solmulla on oletusasetuksena aihe ja yksi solmu on mukautettavalla aiheella. Aiheellisten solmujen nimet ovat live-data-subscribe, joka tilaa nykyisen vaaka painon. Trigger-green-subscribe tilaa pakotetun liikennevalon vaihtamisen aiheen. Weigh-done-subscribe tilaa aiheen, jolla ilmoitetaan, kun punnitus on valmis. Mukautetun solmun nimi on custom-subscribe ja aiheeksi voi valita mitä vain.

Kaikilla solmuilla on muut samanlaista paitsi aihe ja mihin asti viestiä parsitaan. Jokainen solmu toimii virran aloittamissolmuna ja yhdistää automaattisesti aiheeseen, kun Node-RED on käynnistynyt. Solmujen JavaScript-tiedosto alustaa aina uuden instanssin ja tarkistaa onko NODE_ENV ympäristömuuttuja asetettu paikalliselle kehitykselle. Tämä tehdään siksi, jotta voidaan alustaa paikallista kehittämistä varten IPC-kommunikaatio. Instanssin luominen ja ympäristömuuttujan tarkistaminen esitetään kuvassa 36 olevassa JavaScript-tiedostossa.

```
const fs = require('fs');
const pkg = require('aws-iot-device-sdk-v2');
const { Client } = require('aws-iot-device-sdk-v2/dist/greengrasscoreipc');
const { greengrasscoreipc } = pkg;

/**
 * Subscribes to ipc topic that user gives
 * and forwards the data under key message this comes from mgtt
 */
module.exports = function(RED) {
    function CustomSubscribe(config) {
        RED.nodes.createNode(this, config);
        const node = this;
        const environment = process.env['NODE_ENV'];

        if (environment == "local") {
            setEnvironment();
        }
```

Kuva 36. IPC-kommunikaation JavaScript-tiedoston alku

Jos Node-RED on käynnissä paikallisesti, solmu ajaa setEnvironment funktion, joka on esitetty kuvassa 37. Funktiossa asetetaan paikallista IPC-kommunikaatiota varten ympäristömuuttuja SVCUID, jonka arvoksi asetetaan Docker-rakennuksen aikana siirretty IPC-kansiossa olevan SVCUID-tiedoston sisältö, joka on IPC-socketin salainen tunnus.

```
/**
 * Allow mqtt communication locally
 * Env variable for NODE_ENV needs to be local
 */
function setEnvironment() {
    try {
        const content = fs.readFileSync('/ipc/SVCUID', 'utf8');
        process.env['SVCUID'] = content;
    } catch (err) {
        if (err.code === 'ENOENT') {
            console.log('There is no SVCUID file.');
            setTimeout(setEnvironment, 5000);
        } else {
            console.error('An error occurred:', err);
        }
    }
}
```

Kuva 37. Funktio setEnvironment

Kun Node-RED käynnistysjärjestelmä on selvitetty, solmu ajaa funktion getIpcClient, joka on esitetty kuvassa 38. Funktiossa käytetään AWS IoT SDK funktiota createClient. Funktio

createClient luo uuden Greengrass RPC-asiakkaan käyttämällä rakennus vaiheessa asetettuja ympäristömuuttujia. Funktio palauttaa asiakasmuuttujan, jolla voi kutsua muita SDK funktioita. Funktiossa getIpcClient asiakkaan luonnin jälkeen suoritetaan SDK funktio connect. Connect funktio avaa yhteyden. Jos yhteys saadaan luotua getIpcClient funktio palauttaa yhteyden.

```
        /**
         * Creates ipc mqtt connection and connects to it
         * @returns {Client}
         */
        async function getIpcClient() {
            try {
                const ipcClient = greengrasscoreipc.createClient();
                await ipcClient.connect().catch(error => {
                    console.error("Connection error:", error);
                });
                return ipcClient;
            } catch (err) {
                console.error("Error creating IPC client:", err);
            }
        }
    }

    RED.nodes.registerType("custom-subscribe", CustomSubscribe);
};
```

Kuva 38. Funktio getIpcClient

Kun yhteys on saatu, käytetään kuvan 39 toimintoja. Ensin asetetaan IPC-aihe subscribeToTopicRequest objektin sisällä. Aihe saadaan config.topic muuttujasta tai aihe merkitään tekstimuodossa topic arvoksi. Solmu tilaa annetun aiheen AWS IoT SDK funktiolla subscribeToTopic. Kun solmu saa IPC-viestin, viestistä parsitaan vähintään message.binaryMessage.message osuudet pois jokaisessa solmussa. Parsittu viesti asetetaan msg.payload avaimen arvoksi ja lähetetään eteenpäin. Aihe pysyy tilattuna koko ajan.

```
(async () => {
    try {
        this.ipcClient = await getIpcClient();
        const subscribeToTopicRequest = {
            topic: config.topic,
        };
        console.log('subscribe to topic', config.topic)
        const streamingOperation = this.ipcClient.subscribeToTopic(subscribeToTopicRequest);
        if (!streamingOperation) {
            throw new Error("subscribeToTopic method is not available on the IPC client.");
        }

        streamingOperation.on("message", (message) => {
            if (message.binaryMessage) {
                const messageString = message.binaryMessage.message.toString();;
                const data = JSON.parse(messageString);
                node.send({ payload: data });
            }
        });

        streamingOperation.on("streamError", (error) => {
            console.error("Stream error:", error);
        });

        streamingOperation.on("ended", () => {
            console.log("Stream ended");
        });

        streamingOperation.activate();

        await new Promise((resolve) => setTimeout(resolve, 10000));
    } catch (e) {
        console.error("Error:", e);
    }
})();
```

Kuva 39. IPC-viestin tilaaminen

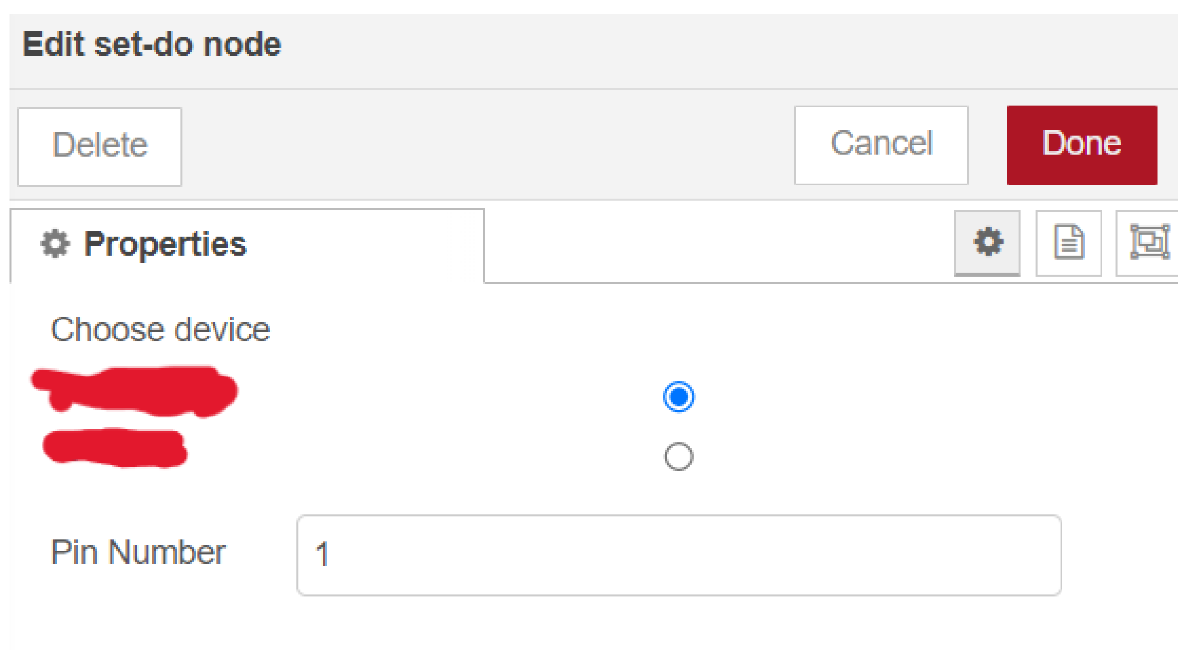
Kuvan 40 HTML-tiedosto on melkein samanlainen jokaisella IPC-kommunikaatio solmulla. Pääosassa solmulle asetetaan kategoriaksi kommunikaatio ja yksi lähtöportti. Muokkaus-pohjassa kerrotaan mitä solmu tekee ja custom-subscribe solmulle asetetaan syöttökenttä, johon aihe kirjoitetaan.

```html
<script type="text/javascript">
  RED.nodes.registerType('custom-subscribe', {
      category: 'communication',
      color: '#b397fa',
      defaults: {
          topic: { value: "" }
      },
      inputs: 0,
      outputs: 1,
      label: function() {
          return this.name || "custom-subscribe";
      }
  });
</script>
<script type="text/x-red" data-template-name="custom-subscribe">
  <div class="form-row">
      <div id="static-text">Give custom topic you wish to subscribe to.</div>
      <label for="node-input-topic">Topic</label>
      <input type="text" id="node-input-topic" placeholder="Topic">
  </div>
</script>
```

Kuva 40. Mukautetun tilauksen HTML-tiedosto

Liikennevaloja ja kulunohjauslaitteita ohjataan digitaalilähdön kautta. Node-RED ei tarjoa valmiita solmuja digitaalitulojen ja -lähtöjen ohjaamista varten. Digitaalitulo/lähdöille on kolme erilaista solmua. Digitaalilähtöjen tilan vaihtaminen set-do solmu ja lukeminen get-do sekä digitaalitulon lukeminen get-di. Jokainen solmu käyttää i2c-bus JavaScript-kirjastoa, joka ladataan Docker-kuvassa. Kirjasto tarjoaa valmiita funktioita I2C-kommunikaatiota varten.

Set-do Solmu aloittaa digitaalilähdön tilan vaihtamisen, kun solmu saa viestin. Viestin msg.payload arvona täytyy olla numero nolla tai yksi. Tämä ilmoittaa mihin digitaalilähdön tila halutaan vaihtaa. Digitaalilähdön tilan vaihtaminen vaatii väylänumeron, laiterekisterin, digitaalilähtörekisterin ja kontrolloitavan digitaalilähdön numeron. Kaikki I2C-rekisterit valitaan radionapilla solmun käyttöliittymästä kuten kuvassa 41 on esitetty. Radionapista valitaan käytössä oleva yhdyskäytävä ja digitaalilähdön numero asetetaan syöttökenttään. Tämä parametrien valinta on käytössä kaikissa DIO-solmuissa.



Kuva 41. Set-do solmun konfiguraatio välilehti

Kun solmu on saanut parametrinsa, solmu suorittaa funktion setDo. Kuvassa 42 näkyy Set-do solmun JavaScript-tiedoston ensimmäinen osa, jossa luodaan solmulle oma instanssi, asetetaan laite parametrit ja suoritetaan setDo funktio.

```
const i2c = require('i2c-bus');

/**
 * Change state of digital output
 * Doesn't return anything
 * @param {Number} msg value passed from another node
 */

module.exports = function(RED) {
    function SetDo(config) {
        RED.nodes.createNode(this, config);
        const node = this;
        let busNumber;
        let deviceRegister;
        let doRegister;
        let pin = config.pin;

        // Set values based on the device type
        if (config.device === '') {
            busNumber =
            deviceRegister =
            doRegister =
        } else if (config.device === '') {
            busNumber = ;
            deviceRegister = ;
            doRegister = ;
        }

        node.on('input', async function(msg) {
            let insertValue = msg.payload;
            try {
                await setDo(busNumber, deviceRegister, doRegister, pin, insertValue);
            } catch (error) {
                node.error(error.message, msg);
            }
        });
```

Kuva 42. Set-do solmun ensimmäinen osa

SetDo funktiossa avataan I2C-väylän yhteys käyttämällä i2c-bus kirjaston valmista funktiota openPromisefied. Yhdistämisen jälkeen funktio lukee valitun nastan nykyisen tilan i2c-bus funktiolla readByte. Funktio palauttaa nykyisen digitaalilähtörekisterin bittimaskin “10000011”, josta valitaan halutun digitaalilähdön bitti ja verrataan ovatko haluttu arvo ja nykyinen bitti saman arvoiset. Jos arvot ovat erilaiset funktio aloittaa valitun digitaalilähdön vaihtamisen bittiä muuttamalla.

Tilan vaihtaminen nollaksi ja yhdeksi tapahtuvat melkein samalla tavalla. Ensin luodaan bittimaski, jossa halutun nastan bittinumero on muutettu yhdeksi “00000001”. Ero nollan ja yhden vaihtamisessa tulee käytettävässä operaatiossa, kun verrataan nykyiseen rekisterin tilaan. Nolla arvossa bittimaski käännetään, jotta kaikki nollat muuttuvat yhdeksi ja ykköset nollaksi “11111110”. Tämän jälkeen käytetään JA-operaatiota nykyisen digitaalilähdön

bittimaskin ja tehdyn bittimaskin kanssa. Numeroksi yksi muuttaessa ei käännetä bittimaskia ja käytetään TAI-operaatiota.

Operaatioiden jälkeen käytetään i2c-bus funktiota writebyte. Funktio kirjoittaa uuden bittimaskin vanhan digitaalilähtörekisteri arvon tilalle. Lopuksi väylän yhteys suljetaan ja solmu loppuu. Koko setDo funktio on kuvassa 43. Kuvassa on I2C-väylän yhdistäminen ja digitaalilähdön rekisterin lukeminen ja muuttaminen.

```
        /**
         * @param {Number} busNumber
         * @param {Number} deviceRegister
         * @param {Number} doRegister
         * @param {Number} pin Number of the pin that you want to interact with
         * @param {Number} value Value you want to give the pin 1 or 0
         */
        async function setDo(busNumber, deviceRegister, doRegister, pin, value) {
            const bus = await i2c.openPromisified(busNumber);
            try {
                const status = await bus.readByte(deviceRegister, doRegister);
                const pinIsOn = (status & (1 << pin)) !== 0;
                if ((value === 0 && pinIsOn) || (value === 1 && !pinIsOn)) {
                    if (value === 0) {
                        const newStatus = status & ~(1 << pin);
                        await bus.writeByte(deviceRegister, doRegister, newStatus);
                    } else {
                        const newStatus = status | (1 << pin);
                        await bus.writeByte(deviceRegister, doRegister, newStatus);
                    }
                } else {
                    console.log('Error with pin value')
                }
            }catch (e) {
                console.error("Error:", e);
            } finally {
                await bus.close();
            }
        }
    }

    RED.nodes.registerType("set-do", SetDo);
};
```

Kuva 43. Digitaalilähdön muuttaminen

Digitaalitulon ja -lähdön lukeminen tapahtuu eri solmuissa nimeltään get-di ja get-do. Molemmilla solmuilla on samanlainen konfiguraatio välilehti kuin set-do solmulla. Solmut eivät ota vastaan dataa viesteistä, mutta vaativat viestin ottamisen vastaan ennen aloittamista ja lähettävät valitun digitaalitulon tai -lähdön tilan solmusta eteenpäin. Lukemis-solmut eroavat JavaScript-tiedostossa rekisterinumeroilla. Lukemis-solmuilla on samanlainen JavaScript-tiedosto rakenne kuin set-do solmulla mutta toteuttavat vain I2C väylän yhdistämisen, laiterekisterin lukemisen ja I2C väylän yhteyden katkaisemisen. Tilan saannin jälkeen tila

lähetetään viestissä eteenpäin seuraavalle solmulle Node-RED lähettämisfunktiolla. Kuvassa 44 on get-do solmun digitaalilähdön tilan lukemisfunktio ja tilan lähettäminen eteenpäin. Get-di solmulla on samanlainen JavaScript-tiedosto.

```
node.on('input', async function(msg) {
    try {
        const data = await getDo(busNumber, deviceRegister, doRegister, pin);
        node.send({ payload: data });
    } catch (error) {
        node.error(error.message, msg);
    }
});

/**
 * @param {Number} busNumber
 * @param {Number} deviceRegister
 * @param {Number} doRegister
 * @param {Number} pin Number of the pin that you want to interact with
 */
async function getDo(busNumber, deviceRegister, doRegister, pin) {
    const bus = await i2c.openPromisified(busNumber);
    try {
        const status = await bus.readByte(deviceRegister, doRegister);
        const value = (status & (1 << pin)) >> pin;
        return value;
    } finally {
        await bus.close();
    }
}
```

Kuva 44. Get-do solmun JavaScript-tiedosto

Kaikkien DIO-solmujen HTML-tiedostot ovat melkein samanlaisia. Tiedoston pääosassa asetetaan solmu dio kategorian alle, yksi tuloportti ja radionapin valinta tallennetaan kuten kuvassa 45 on toteutettu. Solmun muokkauspohjassa asetetaan radionapit laite valintaa varten ja syöttökenttä digitaalilähdön valintaa varten. Get-di ja get-do solmulla on myös yhdet lähtöportit solmuissa, jotta tilan voi lähettää eteenpäin.

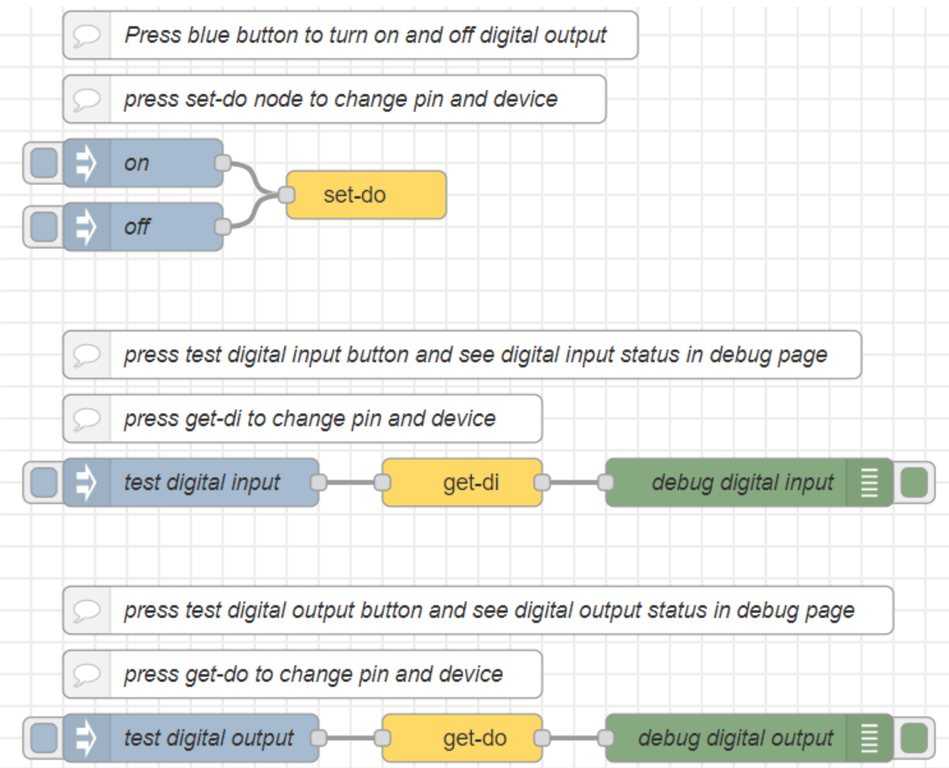
```
<script type="text/javascript">
    RED.nodes.registerType('set-do', {
        category: 'dio',
        color: '#ffd966',
        defaults: {
            device: { value: "" },
            pin: { value: "" }
        },
        inputs: 1,
        outputs: 0,
        label: function() {
            return this.name || "set-do";
        },
        oneditprepare: function() {
            // Load the device value
            if (this.device === "[illegible]") {
                $("#node-input-device-[illegible]").prop("checked", true);
            } else if (this.device === "[illegible]") {
                $("#node-input-device-[illegible]").prop("checked", true);
            }
        },
        oneditsave: function() {
            // Save the device value
            if ($("#node-input-device[illegible]").is(":checked")) {
                this.device = "[illegible]";
            } else if ($("#node-input-device-[illegible]").is(":checked")) {
                this.device = "[illegible]";
            } else {
                this.device = "";
            }
        }
    });
</script>
```

Kuva 45. DIO-solmujen HTML-tiedoston pääosa

8.5 Virtojen rakentaminen

Node-RED virroilla toteutetaan lisänäytön, liikennevalojen ja kulunohjauslaitteiden ohjaaminen ja digitaalitulojen ja -lähtöjen testaaminen. Jokainen virta on luotu mukautettavissa olevaksi, koska oheislaitteen kytkentä voi olla erilainen asennuspaikan mukaan. Ensimmäiset virrat ovat digitaalitulojen ja -lähtöjen testaamista varten. Virrat näkyvät kuvassa 46. Virta-välilehdellä on kolme erilaista virtaa. Yksi toteuttaa digitaalilähtöjen tilan vaihtamisen käyttämällä set-do solmua. Set-do funktio käynnistetään kahdella syöttösolmulla. Syöttösolmuilla lähetetään set-do solmulle haluttu digitaalilähdön tila. Set-do virtaa käytetään oheislaitteiden asennuksessa digitaalilähtöjen toiminnan varmistamiseksi. Kaksi muuta virtaa toteuttavat digitaalitulojen ja -lähtöjen tilan lukemisen. Virrat käyttävät syöttösolmuja,

joiden viesteillä ei ole väliä, koska digitaalitilojen lukemiseen käytettään get-di ja get-do solmuja, jotka eivät ota vastaan mitään viesti dataa. Solmut palauttavat digitaalitulon tai -lähdön tilan ja viestit lähetetään omiin debugaus-solmuihin, jotka tulostavat viestit debugaus välilehdelle.



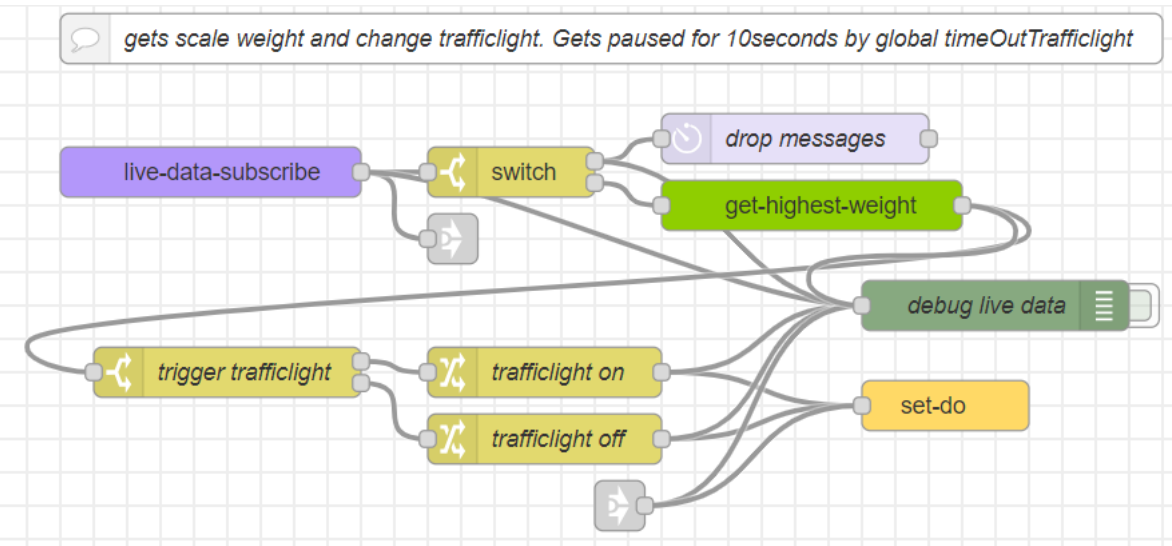
Kuva 46. DIO testaus virtoja

Liikennevaloja ohjataan kolmen erilaisen virran kautta ja jokaisella virralla on oma välilehti Node-RED käyttöliittymässä. Liikennevaloja ohjataan eniten live data nimiseltä välilehdeltä. Virran päätehtävänä on muuttaa digitaalilähdön tilaa nykyisen vaa'an painolukeman mukaisesti. Virta käyttää live-data-subscribe solmua, joka lähettää vastaanotetusta viestistä jokaisen vaa'an painolukeman JSON-muodossa ehtosolmulle ja lähtö linkkisolmulle. Ehtosolmu tarkistaa, onko globaali konteksti timeOutTrafficlight arvo tosi. Jos viestin arvo on tosi, viesti poistetaan viivesolmulla, koska live data virta on toimintakatkolla muiden virtojen toimesta. Muussa tapauksessa viesti kulkee get-highest-weight solmuun, joka lähettää suurimman painolukeman eteenpäin ehtosolmulle.

Ehtosolmun tehtävä on jakaa viesti painolukeman mukaan kahdelle muutossolmulle. Ehtosolmulla selviää, onko ajoneuvo vaa'alla. Muutossolmut muuttavat viestin arvon numeroksi yksi tai nolla riippuen kumpaan solmuun viesti lähetetään. Lopuksi viesti lähetetään set-do solmulle, joka vaihtaa digitaalilähdön tilan. Set-do solmuun on yhdistetty sisääntulo linkkisolmu, jonka kautta digitaalilähtöä voidaan vaihtaa tarpeen mukaan toisesta virrasta.

Virralla on myös debugaus-solmu, joka on yhdistetty tärkeimpiin solmuihin virran toiminnan kannalta.

Digitaalilähtöön on yleisesti yhdistetty liikennevalo ja valo muuttuu punaiseksi, kun tietty painolukema ylitetään. Virta on esitettynä kuvassa 47. Kuvassa on virta, joka vaihtaa digitaalilähdön tilaa ja lähettää vaa'an painoa linkkisolmulla toiselle virralle ja vastaanottaa digitaalilähdön muutos viestejä linkkisolmulla muilta virroilta.

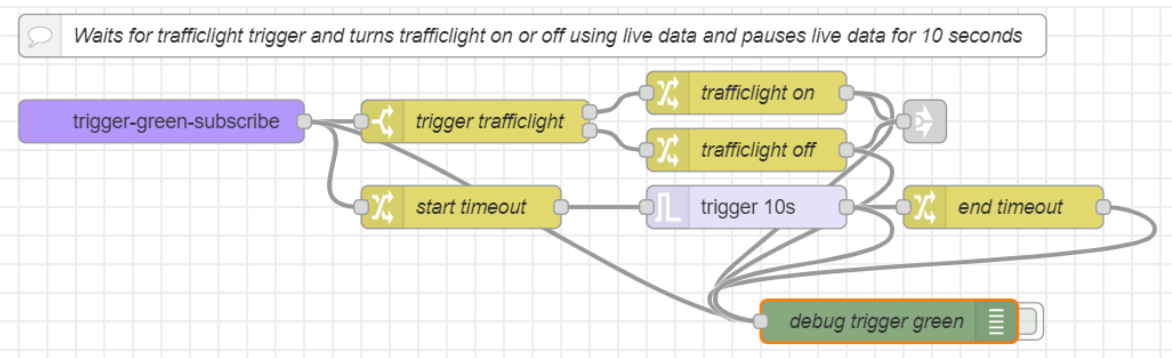


Kuva 47. liikennevalojen ohjaamisen live data virta

Seuraava virta on Trigger trafficlight. Virran tehtävä on vaihtaa liikennevalojen väriä pakotus komennolla. Virta on tarkoitettu liikennevalojen etäohjaamiseen punnituspalvelun kautta. Virta alkaa trigger-green-subscribe solmusta. Solmu vastaanottaa IPC-viestejä, jotka sisältävät komennon Green tai Red. Tämä viesti määrää vaihtuuko valo vihreäksi vai punaiseksi. Viesti lähetetään muutossolmulle ja ehtosolmulle. Muutossolmu asettaa globaalin kontekstin timeOutTrafficlight arvon todeksi. Tätä arvoa tarkkaillaan live datan virrassa live-data-subscribe solmun jälkeisellä ehtosolmulla. Muutossolmu lähettää viestit laukaisinsolmulle, joka odottaa 10 sekuntia ennen, kuin solmu lähettää viestin eteenpäin. Viesti lähetetään uuteen ehtosolmuun, joka muuttaa globaalin kontekstin epätodeksi ja poistaa viestin. Tämä live datan toimintakatko tehdään, koska muuten liikennevalojen muutos yliajettaisiin heti.

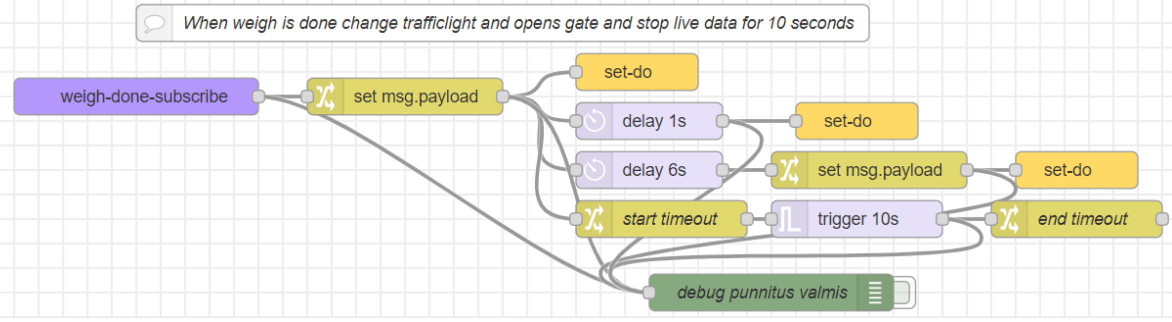
Trigger-green-subscribe solmun jälkeinen ehtosolmu jakaa viestin kahden muutossolmun välillä samalla tavalla kuin live data virrassa, mutta ehtosolmu jakaa viestin tekstiarvon mukaan. Kun arvo on Green, lähetetään viesti muutossolmulle trafficlight off, joka muuttaa arvon nollaksi. Red arvolla viesti lähetetään trafficlight on ehtosolmulle, joka vaihtaa arvon yhdeksi. Ehtosolmut lähettävät arvot lähtö linkkisolmulle, joka on linkitetty live datan sisääntulo linkkisolmuun. Virrassa on myös debugaus-solmu, joka on yhdistetty kriittisiin virran solmuihin. Koko virta esitetään kuvassa 48. Kuvan virrassa kuunnellaan IPC-aihetta ja

lähettää viestin linkkisolmulle live data virralle, jossa tehdään digitaalilähdön vaihto. Virta toimii myös live datan toiminnankatkaisijana. Toimintakatko kestää 10 sekuntia.



Kuva 48. Liikennevalojen etävaihtamista varten rakennettu virta

Viimeinen virta, joka käyttää digitaalilähdön tilan muuttamista, on punnitus valmis välilehdellä oleva virta, joka esitetään kuvassa 49. Virran tarkoitus on muuttaa liikennevalo vihreäksi ja avata kulunohjauslaite, kun punnitus on valmistunut. Virta alkaa weigh-done-subscribe solmusta. Solmu lähettää viestin vastaanottamisen jälkeen ehtosolmulle, joka muuttaa viestin numeroksi yksi. Ehtosolmu lähettää viestin neljälle eri solmulle. Kaksi solmua on set-do solmuja. Set-do solmuja käytetään liikennevalon vaihtamiseen ja kulunohjauslaitteen avaamiseen. Kolmas solmu on kuuden sekunnin ajastin, jonka jälkeen muutossolmussa viestin arvo muutetaan nollaksi ja lähetetään set-do solmulle, jossa lopetetaan kulunohjauslaitteen avaamisen ohjaaminen. Virta toteuttaa myös saman live datan toimintakatkon kuin Trigger trafficlight. Virtaan on lisätty debugaus-solmu, joka on yhdistetty kriittisiin kohtiin virran toiminnan seurantaa varten.



Kuva 49. Punnitus valmis komentoa varten rakennettu virta

Viimeinen rakennettu virta on lisänäytön toimintaa varten. Virran tehtävä on ottaa vastaan vaa'an jokaisen sillan kokonaispaino ja summata nämä yhteen ja ottaa yhteys yhdyskäytävän sarjaliikenneporttiin, josta lähetetään ASCII-muodossa oleva summapaino lisänäytölle. Lisänäyttö muuttaa ASCII-viestin numeroiksi ja tulostaa sen näytölle. Lisänäytöltä saadaan kokonaispainolukema otettua ylös, jos punnituspalvelulla on käyttökatkos. Virta alkaa sisääntulo linkkisolmusta. Solmu on yhdistetty live data virran lähtö linkkisolmuun, joka

sijaitsee virran IPC-solmun jälkeen. Sisääntulo linkkisolmu ottaa vastaan live datasta saatua vaakasiltojen painolukemaa.

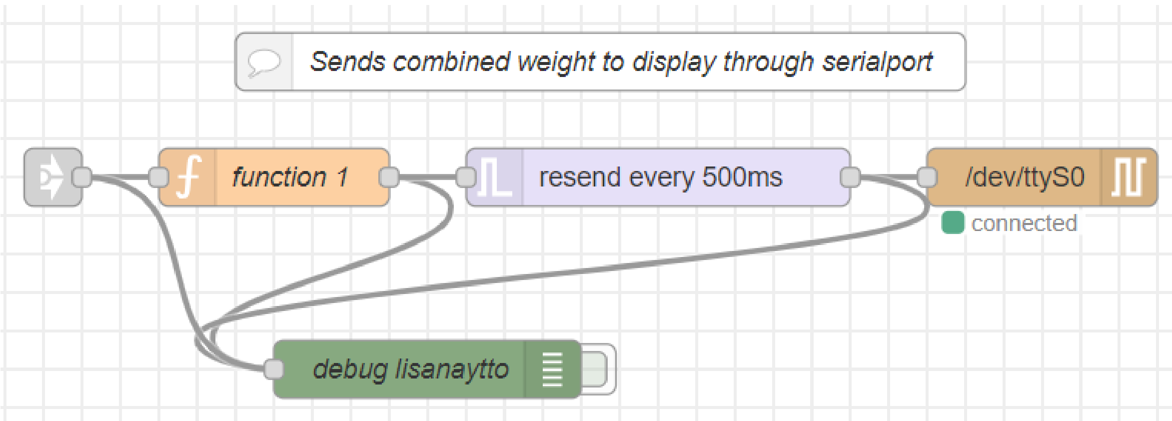
Solmu lähettää viestin eteenpäin funktiosolmulle, jossa jokaisen vaakasillan kokonaispainot summataan yhteen ja muutetaan ASCII-muotoon. ASCII-viestiin lisätään aloitus- ja lopetusmerkit, jotta lisänäyttö tietää milloin viesti alkaa ja loppuu. Lisänäytöillä yleinen aloitusmerkki on STX (02) ja lopetusmerkki CR (13). ASCII-viesti lähetetään funktio solmusta msg.payload muuttujan arvona. Funktio on esitetty kuvassa 50. Kuvan koodissa summataan yhteen JSON datan kokonaispainot ja painolukema muutetaan ASCII-muotoon.

```
1   let totalGrossWeight = 0;  // Initialize the sum
2
3   msg.payload.forEach(scale => {
4       if (scale.current_weight && scale.current_weight.gross_weight !== null) {
5           // Add gross_weight to totalGrossWeight
6           totalGrossWeight += scale.current_weight.gross_weight;
7       }
8   });
9
10  // Convert totalGrossWeight to its ASCII string representation
11  const weightString = totalGrossWeight.toString();  // Convert the number to a string
12  const weightBuffer = Buffer.from(weightString, 'ascii');  // Create a buffer from the ASCII string
13
14  // Define start and end markers
15  const startMarker = Buffer.from([2]);  // Start marker (STX)
16  const endMarker = Buffer.from([13]);   // End marker (Carriage Return, CR)
17
18  // Combine startMarker, weightBuffer, and endMarker into one final buffer
19  const finalBuffer = Buffer.concat([startMarker, weightBuffer, endMarker]);
20
21  // Set the finalBuffer as the payload to be sent
22  msg.payload = finalBuffer;
23
24  // Return the modified message to send it to the next node in the flow
25  return msg;
```

Kuva 50. Funktiosolmun sisältö

Solmu lähettää viestin laukaisinsolmulle, joka lähettää samaa viestiä puolensekunnin välein uudelleen, kunnes uusi viesti korvaa nykyisen. Viestin uudelleen lähettäminen on vaadittua, koska lisänäytöt eivät tallenna viestejä vaan näyttävät viestin ja unohtavat sen heti. Laukaisin solmu lähettää viestiä lähetys sarjaliikenneporttisolmulle, johon on luotu konfiguraatiosolmu.

Konfiguraatiosolmu on parametroitu ottamaan yhteys yhdyskäytävän sarjaliikenneporttiin. Sarjaliikennesolmu aloittaa yhteyden sarjaliikenneportin kanssa aina virran aloituksessa. Kun solmu saa viestin, data lähetetään sarjaliikenneportille, joka lähettää viestin lisänäytölle. Debugaus-solmu on yhdistetty jokaisen solmun lähetysporttiin. Kuvassa 51 on esitetty lisänäytön toimintaa varten rakennettu virta, jossa lähetetään dataa puolensekunnin välein sarjaliikenneporttiin.



Kuva 51. Virta sarjaliikenneporttia varten

8.6 Ongelmat

Oheislaitteiden, vaakojen ja yhdyskäytävien kanssa ongelmaksi tulee eroavaisuudet laitteiden välillä. Aina kun uudenlainen yhdyskäytävä otetaan käyttöön, tarvitsee I2C-toolsilla selvittää väylänumero ja rekisterinumerot. Oheislaitteissa ei ole standardia, jonka mukaan laitteita valmistetaan. Tämä johtaa siihen, että erivalmistajien oheislaitteet toimivat eri tavoilla liitäntöjen ja protokollien suhteen. Vaa'at lähettävät eri protokollilla tai toimintamalleilla painolukemaa, johon täytyy sovellus muotoilla joko koodin tai Node-RED:in avulla.

Ongelmaksi tulee myös asiakkaiden näkemykset, miten he haluavat laitteiden toimivan ja näiden näkemyksien toteuttaminen. Kaikki tämä eroavaisuus luo mukautetun ympäristön, kun vaaka asennetaan asiakkaalle. Node-RED helpottaa tämän ratkaisemisessa tietovirtoihin perustuvalla ohjelmistokehityksellä.

Node-RED:in käyttöliittymän ongelmana on, kun asettaa solmun tai datan väärin aiheuttaen error tilanteen Node-RED sovellus ja kontti sammuu. Tämän korjaa Greengrass automaattisella uudella Docker-kuvan rakentamisella, mutta olisi parempi, jos Node-RED menisi vain error tilaan eikä vaatisi uudelleen käynnistämistä.

8.7 Tulokset

Node-RED ympäristö asennetiin kahteen eri kohteeseen Suomessa. Kohteissa oli lisänäyttö, johon täytyi saada monen vaa'an yhteinen summapaino, liikennevalot, jotka vaihtoivat valoa painon tai komennon mukaan sekä kulunohjauslaite, joka aukaistiin punnituksen valmistumisen jälkeen. Asiakkaan yhdyskäytävälle lähetettiin Node-RED komponentti Greengrass lähettämisen avulla. Virtoihin täytyi valita jokaiselle DIO-solmulle oikea yhdyskäytävä ja ohjattava digitaalitulo tai -lähtö. Lisänäyttöä varten täytyi asettaa oikea määrä aloitusmerkkejä. Lisänäyttö oli konfiguroitu kolmella aloitusmerkillä. Node-RED on toimivassa käytössä asiakkaalla eikä toiminnassa ole huomattu ongelmia.

8.8 Jatkokehitys ja ylläpito

Node-RED sovelluksella rakennetaan tulevaisuudessa aina yhteydet oheislaitteisiin. Kehitysprosessin helpottaminen erilaisilla alivirroilla on tärkeää. Itsetehdyille solmuille voisi rakentaa testit toiminnallisuuden varmistamiseksi. Työpaikalle voisi rakentaa oheislaitteiden testaamisympäristön, jossa olisi mahdollista aina testata kaikkien virtojen toiminta oheislaitteilla ennen virran julkaisemista tuotantokäyttöön. Etäparametrointi Node-RED virroille punnituspalvelun kautta poistaisi tarpeen etäyhteyden ottamiselle yhdyskäytävään aina kun Node-RED virta tarvitsee muutoksen.

9 Yhteenveto ja pohdinta

Opinnäytetyön tavoitteena oli rakentaa POC versio oheislaitteiden liittämisestä punnituspalveluun käyttämällä low-code työkalua. Työkalu täytyi integroida pilvipalveluun, josta se voidaan lähettää pilven kautta yhdyskäytävälle etänä ja työkalun täytyy toimia omassa ympäristössään erossa yhdyskäytävän isäntäkäyttöliittymästä. Oheislaitteiden liitäntä täytyy olla mukautettavaa ja uusia laitteita täytyy pystyä liittämään ilman suurta resurssitarvetta.

Työssä rakennettiin työkalut lisänäyttöjä, kulunohjauslaitteita ja liikennevaloja varten Node-RED low-code sovelluksen avulla. Node-RED on tarkoitettu IoT-laitteiden kanssa työskentelyä varten. Node-RED virtoihin käytettiin valmiita, ladattavia ja itsetehtyjä solmuja virtojen toimintaa varten. Valmiita solmuja käytettiin viestien reitittämisessä, muokkaamisessa ja ajoittamisessa. Lisänäyttöjen sarjaliikennekommunikaatiota varten ladattiin solmupaketti. Sarjaliikenteessä käytettiin ASCII-standardia viestien formatoimisessa lisänäyttöä varten. Itsetehtyjä solmuja käytettiin datan formatoimisessa, IPC-kommunikaatiossa ja digitaalitulojen ja -lähtöjen lukemisessa ja muuttamisessa. Digitaalituloja ja -lähtöjä ohjattiin I2C-protokollalla. Digitaalilähtöjä tarvittiin liikennevalojen vaihtamiseen ja kulunohjauslaitteiden avaamiseen. Node-RED sovellus suoritettiin Docker-kontin sisällä, joka lähetetään AWS pilvipalvelusta Greengrass -palvelun avulla. AWS Greengrass -ydinkomponentti mahdollisti IPC-kommunikaation Node-RED Docker-kontin ja muiden Docker-konttien välillä yhdyskäytävän sisällä.

Rakennettujen Node-RED virtojen toiminnot ovat:

- Digitaalitulojen ja -lähtöjen testaaminen.
- Vaa'an painon seuraaminen, jotta liikennevalot vaihtuvat painon mukaan.
- Liikennevalon pakko vaihtaminen IPC-viestillä.
- Valmiin punnituksen jälkeen liikennevalon väri vaihtaminen vihreäksi ja kulunohjauslaitteen avaaminen.
- Vaakasiltojen summatun painon lähettäminen ASCII-muodossa sarjaliikenneportin kauttaa lisänäytölle.

Node-RED virrat luotiin mahdollisimman mukautettaviksi, koska oheislaitteiden toiminta muuttuu valmistajan ja konfiguraation mukaan. Asiakkailla voi olla omia toiveita, miten oheislaitteiden kuuluisi toimia.

Virrat ovat käytössä kahdella vaa'alla Suomessa. Vaaoilla ei ole tullut ilmi ongelmia Node-RED virtojen kanssa. Node-RED on hyvin suunniteltu työkalu IoT-laitteille sen laajalla dokumentaatiolla ja käyttäjävapaudella tehdä mitä vain. Jatkotoimenpiteinä osan virtojen

solmuista voisi yhdistää alivirroiksi uudelleen käytettävyyttä varten ja punnituspalvelun kautta etäparametroinnin mahdollistaminen, jotta ei tarvitsisi aina avata Node-RED käyttöliittymää.

Lähteet

Afzal, S. 2016. I2C Primer: What is I2C? (Part 1). Analog devices. Viitattu 5.8.2024. Saatavissa https://www.analog.com/en/resources/technical-articles/i2c-primer-what-is-i2c-part1.html

Ambientia. 2023. Mitä tarkoittaa Low-code –ohjelmistokehitys. Viitattu 21.9.2024. Saatavissa https://www.ambientia.fi/ajankohtaista/mita-tarkoittaa-low-code-ohjelmistokehitys

Amazon Web Services. 2024. Overview of Amazon Web Services. Viitattu 7.8.2024. Saatavissa https://docs.aws.amazon.com/whitepapers/latest/aws-overview/introduction.html

Amazon Web Services. a. Component environment variable reference. Viitattu 12.8.2024. Saatavissa https://docs.aws.amazon.com/greengrass/v2/developerguide/component-environment-variables.html

Amazon Web Services. b. Device provisioning. Viitattu 8.8.2024. Saatavissa https://docs.aws.amazon.com/iot/latest/developerguide/iot-provision.html

Amazon Web Services. c. How AWS IoT works. Viitattu 7.8.2024. Saatavissa https://docs.aws.amazon.com/iot/latest/developerguide/aws-iot-how-it-works.html

Amazon Web Services. d. Publish/subscribe local messages. Viitattu 13.8.2024. Saatavissa https://docs.aws.amazon.com/greengrass/v2/developerguide/ipc-publish-subscribe.html

Amazon Web Services. e. Security best practices in AWS IoT Core. Viitattu 7.8.2024. Saatavissa https://docs.aws.amazon.com/iot/latest/developerguide/security-best-practices.html

Amazon Web Services. f. Use the AWS IoT Device SDK to communicate with the Greengrass nucleus, other components, and AWS IoT Core. Viitattu 10.9.2024. Saatavissa https://docs.aws.amazon.com/greengrass/v2/developerguide/interprocess-communication.html

Amazon Web Services. g. What is AWS IoT Greengrass. Viitattu 10.8.2024. Saatavissa https://docs.aws.amazon.com/greengrass/v2/developerguide/what-is-iot-greengrass.html

Amazon Web Services. h. What’s the Difference Between RPC and REST. Viitattu 10.8.2024. Saatavissa https://aws.amazon.com/compare/the-difference-between-rpc-and-rest/

Cambell, S. BASICS OF THE I2C COMMUNICATION PROTOCOL. Circuit Basics. Viitattu 2.8.2024. Saatavissa https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/

Dceejay., Knolleary. 2024. node-red-node-serialport. OpenJS Foundation. Viitattu 4.10.2024 Saatavissa https://flows.nodered.org/node/node-red-node-serialport

Docker Inc. a. Glossary. Viitattu 11.9.2024. Saatavissa https://docs.docker.com/reference/glossary/#base-image

Docker Inc. b. How Compose works. Viitattu 14.9.2024. Saatavissa https://docs.docker.com/compose/intro/compose-application-model/#key-commands

Docker Inc. c. Publishing and exposing ports. Viitattu 14.9.2024. Saatavissa https://docs.docker.com/get-started/docker-concepts/running-containers/publishing-ports/

Docker Inc. d. Understanding the image layers. Viitattu 11.9.2024. Saatavissa https://docs.docker.com/get-started/docker-concepts/building-images/understanding-image-layers/

Docker Inc. e. Volumes. Viitattu 15.9.2024. Saatavissa https://docs.docker.com/engine/storage/volumes/

Docker Inc. f. What is an image. Viitattu 9.9.2024. Saatavissa https://docs.docker.com/guides/docker-concepts/the-basics/what-is-an-image/

Docker Inc. g. Writing a Dockerfile. Viitattu 11.9.2024. Saatavissa https://docs.docker.com/get-started/docker-concepts/building-images/writing-a-dockerfile/

Emirica. Esineiden internet yksinkertaisesti selitettynä. Viitattu 3.10.2024. Saatavissa https://www.empirica.fi/iot.html

Emqx. 2023. Understanding AWS IoT Core: Features, Use Cases & Quick Tutorial. Viitattu 7.8.2024 https://www.emqx.com/en/blog/understanding-aws-iot-core

English, D. 2024. What Is IoT And Cloud Computing. ROBOT.NET. Viitattu 3.10.2024. Saatavissa https://robots.net/tech/what-is-iot-and-cloud-computing/

Ensatellite. I2C: How it works and how to use it. Ensatellite. Viitattu 2.8.2024. Saatavissa https://ensatellite.com/i2c/

Fawade, A. 2023. How to Use Docker Compose and YAML for Multi-Container Applications | Day 18 of 90 Days Of DevOps. Medium. Viitattu 15.9.2024 Saatavissa

https://ajitfawade.medium.com/how-to-use-docker-compose-and-yaml-for-multi-container-applications-day-18-of-90-days-of-devops-78261fbd7b37

FlowFuse. 2024a. MQTT. Viitattu 25.9.2024. Saatavissa https://flowfuse.com/node-red/core-nodes/mqtt/

FlowFuse. 2024b. Node-RED Editor Header component. Viitattu 21.9.2024. Saatavissa https://flowfuse.com/node-red/getting-started/editor/header/

FOSS TechNix. 2020. 19 Dockerfile Instructions with Examples | Complete Guide. Viitattu 13.9.2024. Saatavissa https://www.fosstechnix.com/dockerfile-instructions/

HiveMQ Team. 2024. MQTT Topics, Wildcards, & Best Practices – MQTT Essentials: Part 5. HiveMQ. Viitattu 3.10.2024. Saatavissa https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices/

Hutasu. 2017. Sarjaliikenne ja sarjaportti. Viitattu 29.9.2024. Saatavissa https://www.hutasu.net/elektroniikka/sulautettu-elektroniikka/sarjaliikenne-ja-sarjaportti/

Hymel S. ASCII. SparkFun Electronics. Viitattu 3.10.2024. Saatavissa https://learn.sparkfun.com/tutorials/ascii/all

Injosoft AB. a. Extended ASCII. Viitattu 3.10.2024. Saatavissa https://www.ascii-code.com/glossary/extended-ascii

Injosoft AB. b. The Beginner's Guide to ASCII. Viitattu 3.10.2024. Saatavissa https://www.ascii-code.com/articles/Beginners-Guide-to-ASCII

Jimblom. Serial Communication. SparkFun Electronics. Viitattu 29.9.2024. Saatavissa https://learn.sparkfun.com/tutorials/serial-communication/all

Kalin, M. 2019. Inter-process communication in Linux: Sockets and signals. Red Hat. Viitattu 5.10.2024. Saatavissa https://opensource.com/article/19/4/interprocess-communication-linux-networking

Keeney, B. 2020. Serial Ports Explained. TEGUAR. Viitattu 29.9.2024. Saatavissa https://teguar.com/what-is-serial-rs232-port/

Kiruri, K. 2024. 28 Cloud Computing Trends: Adoption, Market, Technology, Services, Security and Cost Optimization. Cloudwards. Viitattu 8.8.2024. Saatavissa https://www.cloudwards.net/cloud-computing-trends/

LabJack. 2022. Digital I/O Overview (T-Series Devices). Viitattu 17.9.2024. Saatavissa https://labjack.com/blogs/faq/digital-i-o-overview-t-series-devices

Li, G. 2024. 8 IoT Protocols and Standards Worth Exploring in 2024. EMQ Technologies Inc. Viitattu 3.10.2024. Saatavissa https://www.emqx.com/en/blog/iot-protocols-mqtt-coap-lwm2m

Light, R. 2023. Essential Guide to MQTT Topics and Wildcards. Viitattu 10.9.2024. Saatavissa https://cedalo.com/blog/mqtt-topics-and-mqtt-wildcards-explained/#What_are_MQTT_Topics

Linuxize. 2020. Docker Run Command with Examples. Viitattu 13.9.2024. Saatavissa https://linuxize.com/post/docker-run-command/

NetCorp. Character Table image. Viitattu 3.10.2024. Saatavissa https://www.charstable.com/ascii/

OpenJS Foundation. a. About the OpenJS Foundation. Viitattu 18.9.2024. Saatavissa https://openjsf.org/about

OpenJS Foundation. b. About. Viitattu 19.9.2024. Saatavissa https://nodered.org/about/

OpenJS Foundation. c. Creating your first flow. Viitattu 18.9.2024. Saatavissa https://nodered.org/docs/tutorials/first-flow

OpenJS Foundation. d. Creating your first node. Viitattu 23.9.2024. Saatavissa https://nodered.org/docs/creating-nodes/first-node

OpenJS Foundation. e. Creating Nodes Nodes. Viitattu 19.9.2024. Saatavissa https://nodered.org/docs/creating-nodes/

OpenJS Foundation. f. Editor Guide. Viitattu 20.9.2024. Saatavissa https://nodered.org/docs/user-guide/editor/

OpenJS Foundation. g. Getting Started. Viitattu 18.9.2024. Saatavissa https://nodered.org/docs/getting-started/

OpenJS Foundation. h. HTML File. Viitattu 22.9.2024. Saatavissa https://nodered.org/docs/creating-nodes/node-html

OpenJS Foundation. i. JavaScript file. Viitattu 22.9.2024. Saatavissa https://nodered.org/docs/creating-nodes/node-js

OpenJS Foundation. j. Node-RED. Viitattu 18.9.2024. Saatavissa https://nodered.org/

OpenJS Foundation. k. Node-RED Concepts. Viitattu 19.9.2024. Saatavissa https://nodered.org/docs/user-guide/concepts

OpenJS Foundation. l. Palette. Viitattu 20.9.2024. Saatavissa https://nodered.org/docs/user-guide/editor/palette/

OpenJS Foundation. m. Running under Docker. Viitattu 19.9.2024. Saatavissa https://nodered.org/docs/getting-started/docker

OpenJS Foundation. n. Sidebar. Viitattu 20.9.2024. Saatavissa https://nodered.org/docs/user-guide/editor/sidebar/

OpenJS Foundation. o. The Core Nodes. Viitattu 19.9.2024. Saatavissa https://nodered.org/docs/user-guide/nodes

O'Leary, N. 2019. Making flows asynchronous by default. OpenJS Foundation. Viitattu 23.9.2024 Saatavissa https://nodered.org/blog/2019/08/16/going-async

Page V. 2024. What Is Amazon Web Services, and Why Is It So Successful. Viitattu 8.8.2024. Saatavissa https://www.investopedia.com/articles/investing/011316/what-amazon-web-services-and-why-it-so-successful.asp

Pini, A. 2020. Miten Inter-Integrated Circuit (I2C) -väylä helpottaa mikropiirien yhdistämistä – ja kuinka sitä käytetään. DigiKey. Viitattu 2.8.2024. Saatavissa https://www.digikey.fi/fi/articles/why-the-inter-integrated-circuit-bus-makes-connecting-icsso-easy

Premio Inc. 2022. Explaining DIO & GPIO in Industrial PCs. Viitattu 17.9.2024. Saatavissa https://premioinc.com/blogs/blog/explaining-dio-and-gpio-in-industrial-pcs

Rathore, S. 2022. I2C Utilities in Linux. Linuxhint. Viitattu 2.8.2024. Saatavissa https://linuxhint.com/i2c-linux-utilities/

Raza, M. 2024. SaaS vs. PaaS vs. IaaS: What's the Difference and How to Choose. BMC. Viitattu 9.8.2024. Saatavissa https://www.bmc.com/blogs/saas-vs-paas-vs-iaas-whats-the-difference-and-how-to-choose/#ref1

Simic S. 2020. Docker Privileged - Should You Run Privileged Docker Containers. Viitattu 18.8.2024. Saatavissa https://phoenixnap.com/kb/docker-privileged

Srinam. 2023. What is ASCII – A Complete Guide to Generating ASCII Code. GeeksforGeeks. Viitattu 3.10.2024. Saatavissa https://www.geeksforgeeks.org/what-is-ascii-a-complete-guide-to-generating-ascii-code/

Steve. 2022. Deploying Node-Red Flows. Viitattu 20.9.2024. Saatavissa https://stevesnoderedguide.com/deploying-node-red-flows

Steve. 2023. Beginners Guide to Node-Red Flows. Viitattu 20.9.2024. Saatavissa https://stevesnoderedguide.com/node-red-flows

Tamtron. Vuokravaaka. Viitattu 20.8.2024. Saatavissa https://tamtrongroup.com/fi/palvelut/vuokravaaka/

To, W. 2024. AWS IoT Core Explained: Features, Benefits, and Use Cases. Imply. Viitattu 19.8.2024. Saatavissa https://imply.io/blog/aws-iot-core/

Vadim S. 2024. How to Use AWS IoT Core: Your 2024 Guide. Viitattu 8.8.2024. Saatavissa https://relevant.software/blog/aws-iot-core-guide/

Walker, J. 2024. Docker for Beginners: Everything You Need to Know. Viitattu 9.9.2024. Saatavissa https://www.howtogeek.com/733522/docker-for-beginners-everything-you-need-to-know/

Wawira, M. 2024. 11 Popular Cloud Computing Platforms. Viitattu 8.8.2024. Saatavissa https://www.cloudwards.net/cloud-computing-platforms/