Konsta Siekkeli

Django-pohjaisen verkko- ja sarjaliikenne-palvelimen kehitys ja testaus

Insinööri (AMK)

Tieto- ja viestintätekniikka

Kevät 2021

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä: Siekkeli Konsta

Työn nimi: Django-pohjaisen verkko- ja sarjaliikennepalvelimen kehitys ja testaus

Tutkintonimike: Insinööri (AMK), Tieto- ja viestintätekniikka

Asiasanat: Linux, i.MX 8M Nano, Django

Tämän työn oli tilannut Exens Development Oy. Työn tarkoituksena oli kehittää i.MX 8M Nano EVK -kehitysalustalla toimiva verkkosovellus, joka toteutettaisiin Python-pohjaisella Django-kehysympäristöllä. Verkkosovelluksen kautta voisi lähettää viestejä sarjaliikenneväylään yhdistetylle laitteelle. Kehitettävä sovellutus tahdottiin toteuttaa Debian-käyttöjärjestelmän päälle tämän ominaisuuksien vuoksi. Verkkosovelluksen kehityksessä tuli selvittää tehokkain tapa viestin välittämiseksi verkkosovellukselta sarjaliikennelaitteelle.

Työssä käydään läpi, millä tavoilla GNU/Linux-pohjainen Debian-käyttöjärjestelmä voidaan asentaa i.MX 8M Nano -pohjaiselle kehitysalustalle. i.MX 8M Nano on NXP Semiconductorsin kehittämä ARM-arkkitehtuuriin perustuva sovellussuoritin. NXP tarjoaa suoritinta hyödyntävien järjestelmien kehityksen tueksi ohjelmistoja ja dokumentteja, kuten Debianin asennuksessa käytetyn Binary Support Package (BSP) -paketin. BSP-paketti sisältää Embedded Linux -kehityksessä tarvittavia ohjelmia, kuten Yocto-projektilla luodun GNU/Linux-käyttöjärjestelmäkuvan. Tätä kuvaa ja sen osia hyödynnettiin asennettaessa Debiania i.MX-kehitysalustalle. Vaihdettaessa käyttöjärjestelmäversio Debianiin, ilmeni sarjaliikenteen toiminnassa ongelma, joka johtui Simple Direct Memory Access (SDMA) -ominaisuudesta. Ongelman aiheutti SDMA:n tarvitsema laiteohjelmisto, jonka lataaminen epäonnistui Debianin käynnistyessä. Tämän takia Universal Asynchronous Receiver Transmitter (UART) -laitteiston vastaanottamat viestit eivät välittyneet käyttöjärjestelmälle. Ongelma ratkesi korjaamalla latausprosessissa ilmenneet ongelmat.

Työssä käsiteltiin Pythonin pySerial-moduulin vaatiman ”Polling”-toteutuksen aiheuttamia rajoitteita, kun kehitetään mahdollisimman nopeaa sarjaliikennekommunikaatiota Djangolla. Toteutuksessa päädyttiin hyödyntämään mahdollisimman yksinkertaista toteutusversiota ja multiprocessing-moduulia hyödyntävää versiota. Kehityksen ja testausten aikana havaittiin, että kaikista tehokkain ratkaisu saatiin käyttämällä isäntä-orja-tyyppistä tiedonvälitystä laitteiden välillä ja pitämään sarjaliikennekommunikaatio mahdollisimman yksinkertaisena. Verkkosovelluspuolen kehityksessä hyödynnettiin Asynchronous Javascript and XML (AJAX) -menetelmiä, jotta verkkosivulla olevaa tietoa voitaisiin päivittää tehokkaasti selaimen ja palvelimen välillä.

Prototyypin toteutusversioita testattaessa selvisi, että merkittävin vaikutus Djangon suorituskykyyn oli käytettävällä tietokannalla ja muistikortin ominaisuuksilla. Käyttämällä vanhan tekniikan muistikorttia ja SQLite-tietokantaa saattoi verkkosovelluksen suoritus olla jumiutuneena useita sekunteja tietokantaan kirjoittamisessa. Djangon suorituskykyä pystyi siis parantamaan käyttämällä suorituskykyistä muistia ja tietokannan versiota.

KAMK • University of Applied Sciences

Abstract

Author: Siekkeli Konsta

Title of the Publication: Development and testing of a Django based Web and Serial Device Server

Degree Title: Bachelor of Engineering, Information Engineering

Keywords: Linux, i.MX 8M Nano, Django

The objective in this Bachelor's thesis was to develop a Web application on the i.MX 8M Nano Evaluation Kit. The Web application was created using the Python based Django Web framework and used to manage the sending of commands from the web-application's frontend to the serial device connected to the i.MX 8M Nano Evaluation Kit. GNU/Linux based Debian distribution was used as the base operating system in the i.MX 8M Nano Evaluation Kit. Debian was chosen because of its useful features and easy usage.

This thesis instructs how to install a Debian operating system to the i.MX 8M Nano Evaluation Kit. i.MX 8M Nano is an application processor developed by the NXP Semiconductors. i.MX 8M Nano is based on the ARM-processor architecture. NXP provides, among other software and documents, a Binary Support Package (BSP) that holds useful software and applications to use in Embedded Linux development. The BSP holds a GNU/Linux operating system image that was created with the Yocto Project. This operating system image and its parts were used when installing Debian to the i.MX 8M Nano Evaluation Kit. Changing the operating system to Debian introduced a problem with the serial communication. This problem was due to the failure when loading the firmware of the Simple Direct Memory Access (SDMA) at Debian boot-up. Because of the unfunctional SDMA, messages received by the UART-hardware were never transferred to the operating system and there-fore were not accessible by any application. The serial communication fault was repaired by fixing the problems in the Debian's firmware loading process.

With the web-application development the focus was to find the fastest way to route messages from the web-application's frontend to the serial device. Python module pySerial was used when creating the communication with the serial device. This thesis addresses the different restrictions caused by the polling implementation that needs to be used with the pySerial-module. Two different versions were created from the serial communication to test, which implementation is better. One using Python's multiprocessing module or the other minimalistic approach. The best way to implement the serial communication was to use Master-slave type interaction and keeping the communication implementation as simple as possible. On the Web application's frontend side Asynchronous JavaScript and XML (AJAX) was used to update data between the application's backend and frontend. Using AJAX to update the data between frontend and backend is much more efficient than by the traditional way of sending the whole webpage again.

When testing the basic operation of the different serial communication's implementations, no clear differences were found between the multiprocessing-based and the minimalistic one. However, when the serial communication object is needed to be created in the middle of the request handling the minimalistic approach works much faster than the one using multiprocessing. The interesting discovery was that the database type and used SD-card's features had a much greater impact to the whole system performance than the different serial communication versions.

Sisällys

Termit ja määritelmät

AJAX	Asynchronous JavaScript and XML, tietotekniikan menetelmien joukko, jonka avulla verkkosivun yksittäisiä tietoja voidaan päivittää ilman koko sivun uudelleen lähetystä.
ARM	Advanced RISC Machines, RISC-arkkitehtuuriin perustuvia saman nimisiä suoritinytimiä suunnitteleva ja lisensoiva yhtiö.
ASGI	Asynchronous Server Gateway Interface, nimitys Pythonin API-standardille, joka määrittelee synkronoimattoman kommunikoinnin toteuttamisen kehysympäristön ja verkkopalvelimen välillä.
API	Application Programming Interface, sovellusrajapinta, ohjelmointirajapinta, standardoitu tapa välittää tietoa laitteiden, sovellusten tai käyttäjien välillä.
Bootloader	Käynnistyslataaja, sulautetuissa järjestelmissä käytetty ohjelma, jonka tehtävänä on hoitaa alkuvalmistelut ja ladata itse käyttöjärjestelmä.
BSP	Board Support Package, ohjelmakokoelma, joka sisältää piirivalmistajan ajureita ja ohjelmia heidän tuotteensa käyttämiseen.
CSS	Cascading Style Sheets, WWW-sovelluksen kehityksessä käytettävä tiedosto, jolla voidaan muokata verkkosivun ulkoasua ja muokata sen ominaisuuksia.
Django	Django Web Framework, Python-pohjainen verkkosovellusten toteuttamiseen suunniteltu kehysympäristö.
DMA	Dynamic Memory Access, suora muistihaku, laitteisto-ominaisuus, jossa tietoa voidaan siirtää suoraan I/O-laitteen ja työmuistin välillä ilman tarvetta CPU:lle.

DNS	Domain Name System, internetin nimipalvelujärjestelmä, jonka tehtävänä on muuttaa domain-nimi, kuten ”www.google.fi”, tätä vastaavaksi IP-osoitteeksi.
GIL	Global Interpreter Lock, Pythonin lukko-ominaisuus, joka pakottaa Pythontulkin toimimaan vain yhdellä säikeellä kerrallaan.
IOPS	Input/Output operations Per Second, kuinka monta 4 KB kirjoitus- ja lukuoperaatiota voidaan suorittaa yhden sekunnin aikana.
IPC	Inter-Process Communication, käyttöjärjestelmän prosessien välisen kommunikoinnin ja synkronoinnin mahdollistava mekanismi.
MMC	MultiMediaCard, Flash-muistitekniikkaan perustuva muistikorttistandardi.
EVK	Evaluation Kit, Development kit, kehitysalusta.
Ethernet	IEEE 802.3, LAN- ja WAN-verkoissa käytetyn sarjamuotoisen tiedonsiirron määrittelevä standardi.
HTML	Hypertext Markup Language, verkkosivujen kehityksessä käytettävä kuvauskieli, jolla määritellään verkkosivun rakenne ja sen sisältö.
HTTP	Hypertext Tranfare Protocol, hypertekstin yhteyskäytäntö, TCP/IP:tä hyödyntävä yhteyskäytäntö, jonka avulla siirretään HTML-kielisiä tiedostoja laitteiden välillä.
LAN	Local Area Network eli lähiverkko on nimitys tilanteelle, jossa kodin, toimiston tai vaikka koulun tietokoneet ja tietotekniset laitteet ovat yhteydessä toisiinsa Ethernet-protokollaa käyttävillä fyysisillä yhteyksillä.
MMC	MultiMediaCard, Flash-lukumuistitekniikkaan pohjautuva muistityyppi.

NXP	NXP Semiconductors, alankomaalainen puolijohteita valmistava yritys.
RISC	Reduced Instruction Set Computer, tietokonesuorittimien suunnittelufilosofia, jossa tietokoneen toiminta pyritään toteuttamaan mahdollisimman pienellä määrällä konekäskyjä.
Rootfs	Root File System, Juuritiedostojärjestelmä, nimitys UNIX-tyyppisen käyttöjärjestelmän tiedostorakenteelle, johon käyttöjärjestelmän muodostavat tiedostot on ryhmitelty.
Sarjallistaminen	(Engl. serializing) on tietorakenteen muuttamista toiseen muotoon (yleensä teksti), jossa se voidaan tallentaa esimerkiksi tekstitiedostoon tai lähettää toiselle ohjelmalle tai tietokoneelle käsiteltäväksi.
SD	Secure Digital, Flash-lukumuistitekniikkaan perustuva muistikorttityyppi.
SDMA	Simple Dynamic Memory Access, DMA-ominaisuuden toteutusversio.
Sovelluskehys	Framework, Software Framework: keskeneräiseksi jätetty ohjelmistorunko, jota täydentämällä voidaan rakentaa tehokkaasti määritellyn tyyppinen sovellus.
Tarball	Tervapallo, nimitys tar-ohjelmalla tehdylle tiedostoarkistolle.
UART	Universal Asynchronous Receiver Transmitter, laitetason osa, joka muuttaa tietoa rinnakkais- tai sarjamuotoiseen formaattiin.
URL	Uniform Resource Locator, URL-osoite, internetissä olevan resurssin sijainnin ja sen käyttöön tarvittavan yhteyskäytännön yksilöivä tunnus.
WAN	Wide Area Network, ulko- tai laajaverkko, nimitys yksityiselle verkolle, jolla yhdistetään tietokoneita ja verkkolaitteita pitkien välimatkojen päästä.

WSGI	Web Server Gateway Interface, nimitys Python API-standardille, joka määrittelee synkronoidun tiedon välitystä kehysympäristön ja verkkopalvelimen välillä.

1 Johdanto

Tämän insinöörityön tarkoituksena on kehittää verkkopalvelimella varustettu sarjaliikennepalvelin (engl. Serial Device Server) Debian-käyttöjärjestelmälle. Sarjaliikennepalvelimen tarkoitus on tarjota siihen yhdistetyn sulautetun laitteen ja käyttäjän välille ohjausta helpottava käyttöliittymä, joka on toteutettu Djangon avulla. Samalla tulee selvittää i.MX 8M Nano -sovellussuorittimen soveltuvuutta tällaisen järjestelmän toteuttamiseen.

Työn tilaajana toimi Exens Development Oy, joka on kajaanilainen tuotekehitysyritys. Exens toteuttaa asiakkailleen hyvin laajasti erilaisia tietotekniikan ja mekaniikan ratkaisuja. Heillä oli noussut tarve toteuttaa talon sisäinen sarjaliikennepalvelin, koska markkinoilta ei löytynyt heidän vaatimuksensa täyttävää valmista sovellutusta.

Markkinoilla on tarjolla paljon muuntimia, jotka muuttavat sarjaliikenteen välityksellä siirrettävät viestit Ethernet-verkossa siirrettävään muotoon, mutta verkkopalvelimen ylläpitoon kykeneviä tai tämän ylipäätään mahdollistavia laitteita on tarjolla vähän. Tarjolla olevat valmiit ratkaisut ovat yleensä ominaisuuksiltaan rajallisia, mikä aiheuttaa hankaluuksia kehitettävään tuotteeseen. Toisena ääripäänä ovat markkinoille tulleet valmiit yhden levyn tietokoneratkaisut, jotka tarjoavat jo liikaakin ominaisuuksia eivätkä fyysisiltä ominaisuuksiltaan, kuten iskun- tai kosteudenkestävyys, ole soveltuvia käytettäväksi osana kehitettäviä tuotteita. Tämän takia on nähty etuna lähteä suunnittelemaan omaa laitteistoa, jolla tuotteiden asettamat tiukimmatkin vaatimukset voitaisiin täyttää.

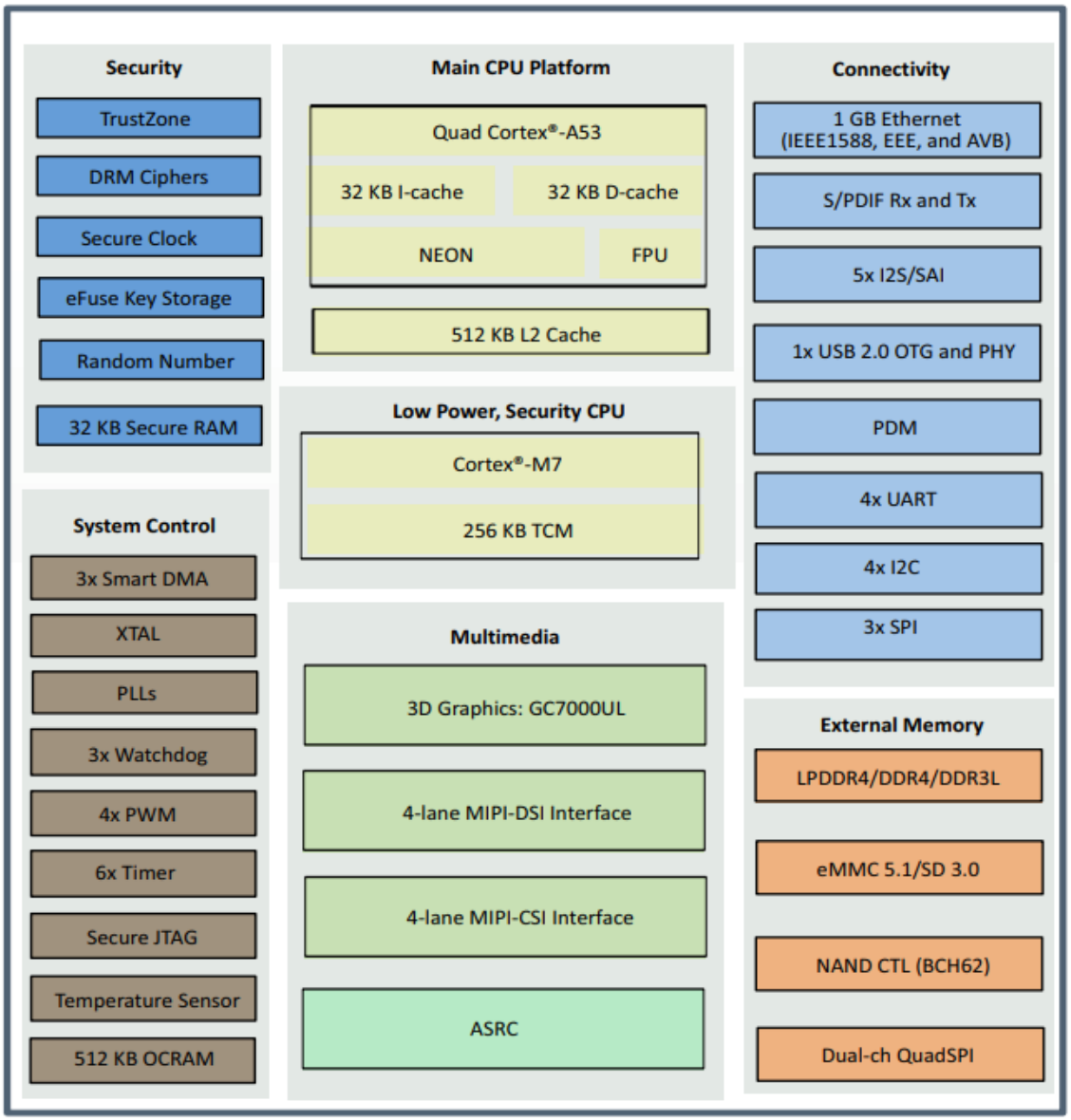
Prototyyppilaitteen toteutukseen yritys oli valinnut NXP:n i.MX 8M Nano -sovellussuorittimen, joka pohjautuu ARMv8-suoritinarkkitehtuuriin. Työn toteutuksessa käytettiin NXP:n tarjoamaa i.MX 8M Nano EVK -kehitysalustaa, joka on suorittimen tuotekehitykseen tarjottava valmis ympäristö.

Verkkosovelluksen kehityksessä haluttiin hyödyntää Djangoa, koska tämä on yritykselle jo ennestään tuttu ja paljon käyttämä kehysympäristö.

Sarjaliikennepalvelimen käyttöjärjestelmäksi haluttiin Debian-niminen GNU/Linux-jakelu. Linux-pohjaiset käyttöjärjestelmät ovat nykyään suosittuja sulautetuissa järjestelmissä Linuxin tarjoaman avoimuuden ja notkean laitteiston tuen sekä tehokkaan konfiguroitavuuden takia. Debian-jakelua haluttiin käyttää sen suuren suosion sekä helpon päivitettävyyden tarjoavan paketinhallintaohjelmiston takia.

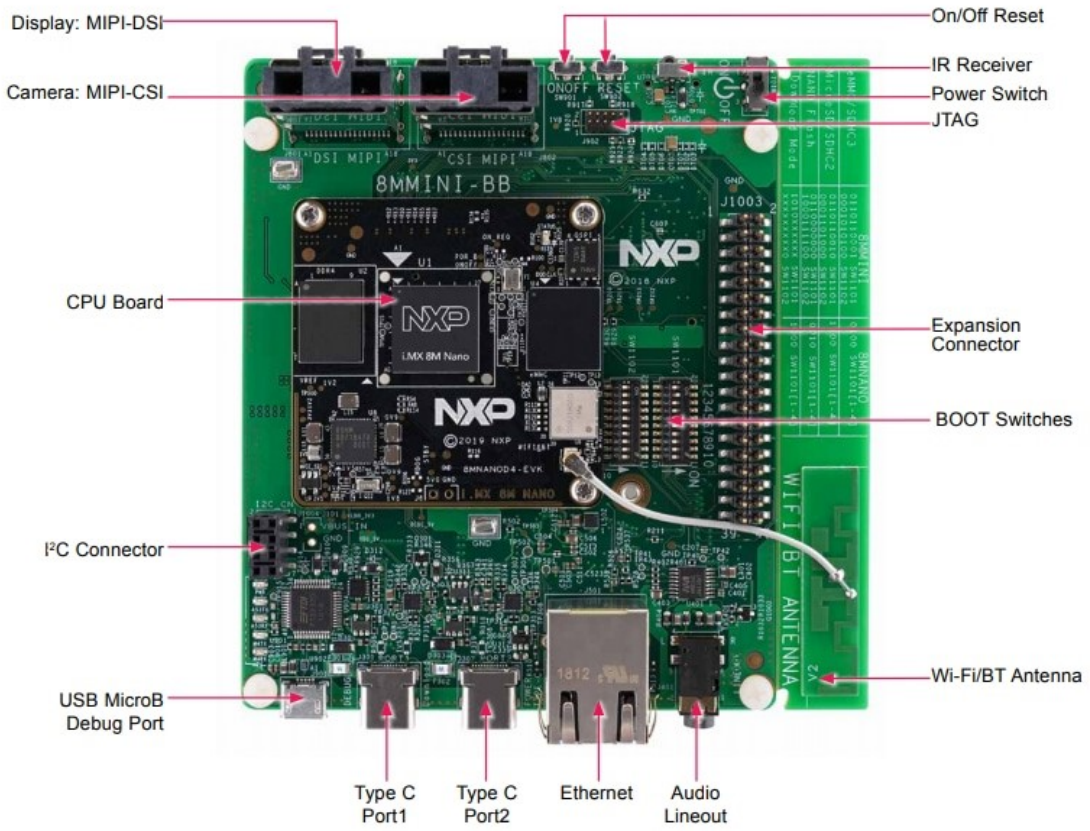
2 i.MX 8M Nano EVK

i.MX 8M Nano EVK on NXP:n tarjoama kehitysalusta ARMv8-arkkitehtuuriin pohjautuvalle i.MX 8M Nano -sovellussuorittimelle. Projektissa kehitysalustaa käytettiin apuna selvittämään ja kehittämään i.MX-suorittimen kanssa yhteensopivaa sarjaliikennepalvelin-ohjelmistoa. i.MX 8M Nano -sovellussuorittimen sisältämiä ominaisuuksia on esitelty seuraavassa lohkokaaviossa (kuva 1).

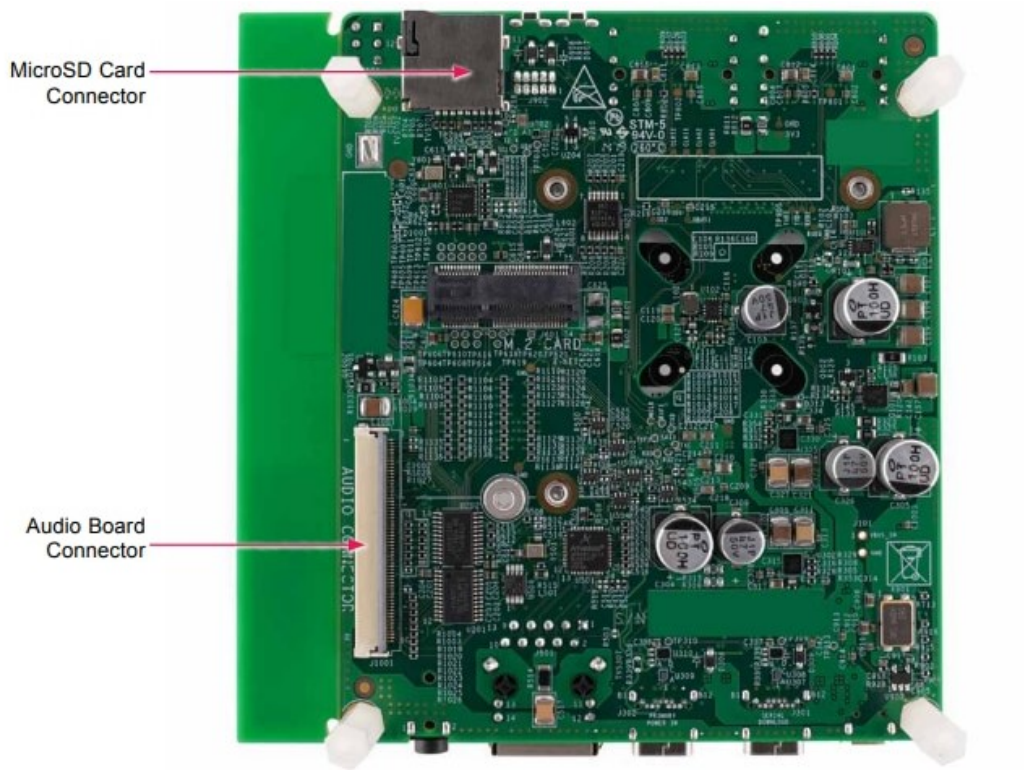


Kuva 1. i.MX 8M Nano -sovellussuoritin lohkokaavio [1].

Kehitysalustan rakennetta ja liitäntöjä on esitelty kuvissa 2 ja 3.



Kuva 2. i.MX 8M Nano kehitysalusta [2].



Kuva 3. i.MX 8M Nano -kehitysalustan pohjapuoli [2].

3 Käyttöjärjestelmä sulautetussa laitteistossa

Normaalisti sulautettuihin laitteisiin on toteutettu ohjelmisto matalan tason C- tai C++-kielisenä ohjelmistona, jonka tehtävänä on ollut yksinkertaisten operaatioiden, kuten releen tilan muuttaminen, tai jonkin suurearvon mittaaminen. Tällaisten sovellusten toteuttaminen on ollut suhteellisen yksinkertaista ja nopeaa rekisteritasolla toimivalla ohjelmalla toteutettuna.

Laskentatehon ja käytettävän muistin kasvamisen myötä voidaan yhä useampia ja monimutkaisempia operaatioita siirtää suoritettavaksi lähellä toimilaitetta sijaitsevaa sulautettua järjestelmää. Tällaisia operaatioita ovat esimerkiksi LAN-verkossa suoritettava HTTP-pohjainen tiedonsiirto, joka käsittää paljon laitteistosta ja ohjelmistosta riippuvia kerroksia kuten Ethernet, IP, TCP ja itse HTTP. Kaikkien näiden kerrosten toteuttaminen ja lisääminen uuteen ohjelmoitavaan järjestelmään on hyvin työlästä. Tämän takia yhä monimutkaistuvien järjestelmien ja toimintojen hallintaan on kehitetty käyttöjärjestelmä.

Käyttöjärjestelmä tarjoaa kerroksen laitteiston ja sovelluksen väliin, jonka tehtävänä on välittää sovellustason pyynnöt laitteistolle. Käyttöjärjestelmät sisältävät myös toiminnallisuuksia, joiden avulla useampia sovelluksia voidaan ajaa samanaikaisesti. Esimerkkinä tällaisista toiminnallisuuksista ovat prosessit, muistinhallinta ja vaiheittainen ohjaus (engl. Scheduler). Tämä mahdollistaa entistä monimutkaisempien järjestelmäkokonaisuuksien rakentamisen ja hallinnan, koska laitteistotason toiminnallisuudet on piilotettu käyttöjärjestelmän taakse. Ohjelmoijan ei siis tarvitse tietää, miten kaikki laitteen toimintaan tarvittavat kerrokset tulisi toteuttaa ja liittää yhteen halutun sovellutuksen luomiseksi. Käyttöjärjestelmän avulla riittää, että ohjelmoija tietää käyttöjärjestelmän tarjoamat kahvat kyseisten toimintojen kutsumiseksi ja voi tällöin käyttää niitä sovelluksessaan halutun alemman tason toiminnallisuuden toteuttamiseksi. Käyttöjärjestelmä siis tarjoaa modulaarisen tavan luoda ja hallita uusia ominaisuuksia järjestelmässä sovellusten avulla. Sovellusten avulla kehitystyötä voidaan pilkkoa ja hallita helpommin kuin yhden valtavan ohjelmakokonaisuuden tapauksessa. [3.] [4.]

Tällä hetkellä yksi suosituimmista tavoista toteuttaa Linux-pohjaisia käyttöjärjestelmiä sulautettuihin laitteisiin on Yocto-projekti, joka tunnettiin aikaisemmin nimellä openEmbedded. Tämän lisäksi on olemassa myös muita tapoja sulautetun Linux-käyttöjärjestelmän luomiseen, mutta Yocton ideana on tarjota tällaisessa kehityksessä käytettävät työkalut yhdessä paketissa. Tässä projektissa ei kuitenkaan käytetty Yoctoa ajansäästösyistä, vaan hyödynnettiin jo valmista Debian-nimistä GNU/Linux-jakelua. Debian tarjosi myös Yoctolla luotavaan jakeluun verrattuna muutaman keskeisen edun. [5.] [6.]

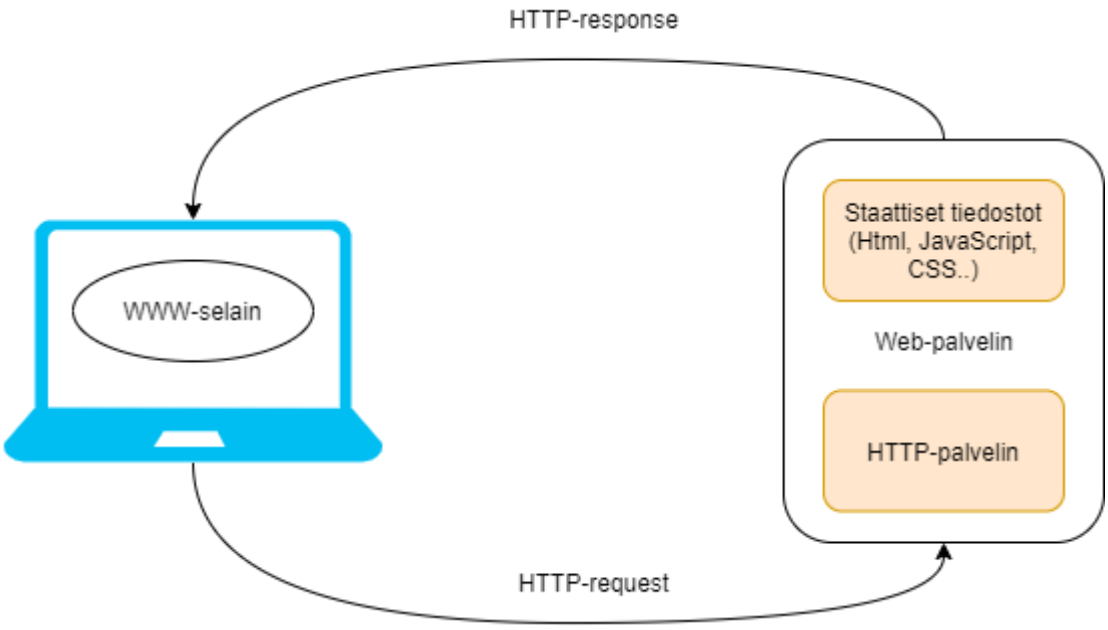
Koska työssä haluttiin käytettävän Djangoa, päädyttiin hyödyntämään Debian-jakelua, jonka sisältämä apt-get-pakettienhallintaohjelmisto mahdollistaa tällaisten osien, kuten Python, lisäämisen ja poistamisen järjestelmästä helposti ja nopeammin kuin Yocton tapauksessa. [6.] [7.]

4 Katsaus Django Web Framework -kokonaisuuteen

Kehysympäristöt (engl. Framework) ovat eräänlaisia keskeneräiseksi jätettyjä sovelluksia tai sovelluskokonaisuuksia, jotka pyrkivät nopeuttamaan ja tehostamaan sovelluksen kehitystä tarjoamalla kehityksen aloittamiseen valmiin pohjan ja valmiita toteutusratkaisuja yleisimpiin toimintoihin. Yleisin ja tunnetuin kehysympäristötyyppi on käyttöliittymien toteutukseen suunnitellut kehykset, jotka tarjoavat laajan API:n esimerkiksi sovellusikkunoiden hallintaan, tapahtumien käsittelyyn ja käyttäjän syötteiden rekisteröintiin. [8.] [9.] [10.]

Tärkeää Djangon kohdalla on huomata, että se ei ole verkkopalvelin, vaikka tarjoaa kehitykseen oman verkkopalvelimen. Tätä ei kuitenkaan tulisi käyttää lopullisessa tuotteessa, vaan hyödyntää oikeaa HTTP- ja/tai verkkopalvelinta, kuten Gunicorn, uWSGI, mod_wsgi, NGINX tai Apache. [11.]

Djangon tarkoitus on helpointa ymmärtää aloittamalla perinteisestä verkkopalvelimen rakenteesta, joka koostuu staattisista tiedostoista, jotka rakentavat perinteisen verkkosivun, sekä HTTP-palvelimesta, jonka tehtävänä on käsitellä selaimelta tulevia HTTP-pyyntöjä ja lähettää takaisin URL-osoitteella määritelty tiedosto (kuva 4).



Kuva 4. Web-palvelimen rakenne ja toiminta [12].

Djangon tarkoitus on korvata osa näistä staattisista tiedostoista mahdollistaen samalla dynaamisten verkkosivujen luonnin. Näin selaimelle lähetettävien HTML-dokumenttien rakennetta ja sisältöä voidaan tarpeen vaatiessa muokata. Django siis luo aina pyyntökohtaisesti selaimelle lähetettävän HTML-dokumentin, joka voi sisältää esimerkiksi tietokannasta haettua tietoa.

4.1 Django-projektin rakenne

Djangolla toteutettu verkkosovellus rakentuu yleensä kahdesta osasta: Django-projektista ja yhdestä tai useammasta Django-sovelluksesta (engl. Django Apps). Näistä Django-projekti sisältää kaikki tulevaan verkkosovellukseen liittyvät asetukset ja konfiguraatiot. Django-sovellus on taas kuvaus luotavasta verkkosovelluksesta tai sen ominaisuudesta. Djangolla luotu verkkosovellus voi siis koostua useammasta Django-sovelluksesta tai koko verkkosovelluksen toiminnan voi sisällyttää yhteen ainoaan Django-sovellukseen. Näistä ensimmäistä tapaa kuitenkin suositellaan käytettäväksi, jolloin luotuja Django-sovelluksen osia voi hyödyntää myös tulevissa sovelluksissa. [13.]

Aloitettaessa kehitys Djangolla luodaan ensin projekti Djangon tarjoamalla komennolla. Komento luo kaikki Django-projektin tarvitsemat tiedostot ja rakentaa projektirakenteen valmiiksi. Tämän jälkeen voidaan projektiin lisätä uusia sovelluspohjia, joihin sovelluksen toiminnallisuus toteutetaan. Oheisessa kuvassa on esimerkki Django-sovellusprojektin rakenteesta (kuva 5). [13.]

```
manage.py
django-projekti/
        __init__.py
        settings.py
        urls.py
        asgi.py
        wsgi.py
django-sovellus/
        __init__.py
        admin.py
        apps.py
        migrations/
        models.py
        tests.py
        views.py
```

Kuva 5. Django-sovellusprojektin rakenne.

Djangon verkkosivut sisältävät hyvin kattavan ohjeistuksen, jonka kautta vähemmänkin Python-kieltä osaava pääsee nopeasti sisälle Djangon toimintaan ja käyttöön. Seuraavissa luvuissa kuitenkin esiteltynä Djangon keskeisiä ominaisuuksia.

4.2 Toiminnallisuuden luonti Djangolla

Näkymät (engl. Views) ovat ensimmäinen ja tärkein Djangon tarjoama ominaisuus. Yksinkertaisuudessaan näkymän käsittelijäfunktioilla toteutetaan se, miten selaimelta saapuviin HTTP-pyyntöihin vastataan. Normaalissa tapauksessa vastaus olisi Djangolla luotu dynaaminen HTML-dokumentti. Näkymä-funktion toteutuksessa on ainoana rajoituksena, että sen tulee palauttaa aina Djangon HttpResponse-olio. Nämä funktiot määritellään views.py-nimiseen Python-moduuliin, joka on osa Django-sovellusta. Toinen tärkeä osa Djangon toimintaa on urls.py-niminen tiedosto, joka oletuksena luodaan Django-projektiin, mutta täytyy manuaalisesti lisätä Django-sovellukseen. Urls.py-tiedostoon määritellään, millä URL-osoitteella kutsutaan mitäkin näkymäfunktiota. [13.] [14.]

4.3 Django ja tietokannat

Djangon tarjoama Models-konsepti on erittäin tehokas API-tietokantojen lisäämiseen ja käyttämiseen Djangossa. Models-konseptin ansiosta sovelluksen tietokannan luomiseksi ei tarvitse muuta kuin määritellä uusi Model-luokka Django-sovelluksen models.py-tiedostoon. Tällaisen tietokantamallin luomisesta on ohessa havainnollistava esimerkki (kuva 6). Kun Model-luokka on luotu, hoitaa Django-kehys kaiken lopun tietokannan taulujen ja kenttien luomiseksi, jotta tietoa voidaan tallentaa tietokantaan. Djangon Models-ominaisuus tukee tällä hetkellä suoraan vain SQL-tyyppisiä tietokantoja, mutta NoSQL-tyyppisten tietokantojen käytön mahdollistavia moduuleja on tarjolla. [13.] [14.] [15.]

polls/models.py

```
from django.db import models


class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')


class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

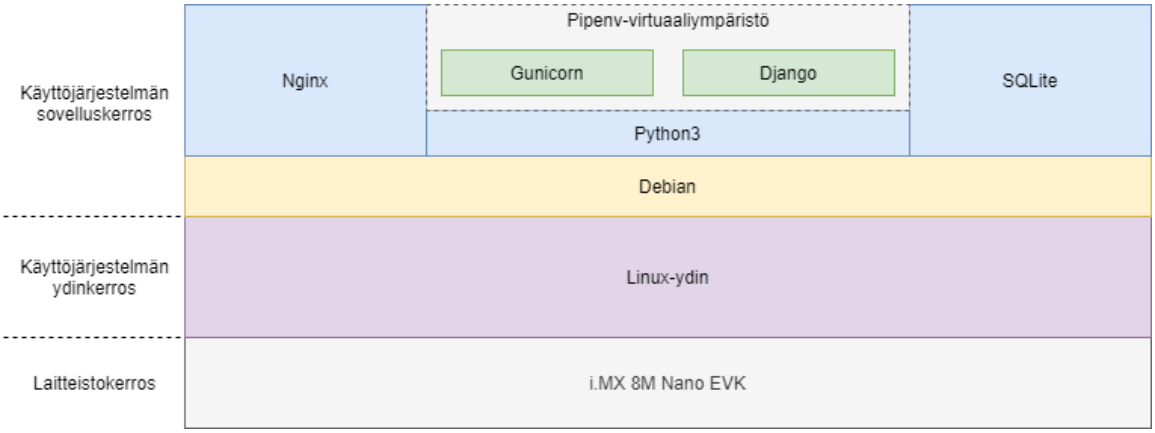
Kuva 6. Esimerkki Djangon models-luokkien luonnista [15].

4.4 Djangon käyttö verkkopalvelimen kanssa

Django ei ole tarkoitettu käytettäväksi verkkopalvelimena, vaikka se sisältää oman kehityksen nopeuttamiseksi tarkoitetun palvelimen. Djangon ainoa tarkoitus on luoda dynaamisesti HTML-dokumentteja, joita voi sitten tarjota oikealle verkkopalvelimelle WSGI-liittymän avulla. WSGI on perusolemukseltaan synkroninen, eli jokainen saapuva HTTP-pyyntö käsitellään alusta loppuun yksi kerrallaan. Synkronisen palvelintyypin vastakohta on synkronoimaton palvelin, joka voi jo aloittamansa pyynnön käsittelyn aikana aloittaa seuraavan pyynnön käsittelyä ja tarpeen mukaan hyppiä näiden välillä. WSGI ei tue tällaista toteutusta, mutta tähän tarkoitukseen on kehitetty ASGI-liittymästandardi, jonka perimmäisenä tarkoituksena on mahdollistaa verkkokehysympäristön ja palvelimen synkronoimattoman kommunikaation. [11.] [16.] [17.]

5 Työn toteutus

Työn toteutuksessa ensimmäinen vaihe oli asentaa i.MX 8M Nano -kehitysalustaan Debian-käyttöjärjestelmä, jonka päälle kehitettäisiin verkkopalvelin. Lopullinen verkkopalvelin tullaan toteuttamaan NGINX-, Gunicorn-, Django- ja SQLite-pinolla, mutta kehitysvaiheessa hyödynnettiin Djangon tarjoamaa Web-palvelinta, joka ei kuitenkaan ole soveltuva käytettäväksi lopullisessa jakeluun suunnatussa versiossa. Oheisessa kuvassa on havainnollistettuna järjestelmän osien sijoittuminen kokonaisuudessa (kuva 7).



Kuva 7. Kuvaus järjestelmäpinosta.

5.1 NXP BSP-kuvan ja Debianin asennus

NXP:n yhteyshenkilöltä oli saatu karkeahko ohjeistus (liite 1) siitä, miten kehitysalustalle saadaan asennettua Debian-jakelu. Ohjeistus alkoi NXP:n tarjoaman BSP:n lataamisesta heidän tukisivuiltaan. Kyseinen tukiohjelmisto on valmiiksi käännetty ohjelmistokuva, joka sisälsi U-Boot-käynnistyslataajan, esikäännetyn Linux-ytimen (engl. Linux Kernel) ja juuritiedostorakenteen tarvittavilla aputiedostoilla varustettuna. Tästä kokonaisuudesta tärkeitä ovat juuritiedostorakenteen sisältämät ladattavat ytimen moduulit ja laiteohjelmistot, U-Boot-käynnistyslataaja ja itse Linux-ydin, johon on käännetty osaksi i.MX kehitysalustan käytettävyyden kannalta olennaisia ajureita.

Ohjeistus kiteytyi NXP:n sivuilta ladatun BSP:n kopioimiseen muistikortille, jota myöhemmin käytetään i.MX:n käynnistysmuistina. Ohjelmistokuvan kopioimisen jälkeen muistikortille piti vielä lisätä uusi muistilohko, jolle kopioitiin qemu-debootstrap-ohjelmalla ladattu ja luotu minimalistinen Debian-juuritiedostojärjestelmä.

5.1.1 Asennuksen alkuvalmistelut

Kaikki alun toimenpiteet suoritettiin Ubuntu 18.04 LTE -käyttöjärjestelmällä varustetulla tietokoneella. Tietokoneen tuli sisältää SD-kortinlukija tai mahdollisuus kytkeä tällainen isäntäjärjestelmään. Debianin juuritiedostojärjestelmän luontiin tarvitaan vähintään Debian-jakeluun perustuva käyttöjärjestelmä. Windows-pohjaisessa järjestelmässä samojen operaatioiden suorittaminen vaatii virtuaaliympäristön, jossa voidaan ajaa Debian-pohjaista jakelua ja päästään käyttämään muistikortinlukijaa.

Ensimmäisenä vaiheena oli ladata ja asentaa Debian-juuritiedostojärjestelmän asennuksessa vaadittavat ohjelmistot osaksi Ubuntua. Suurin osa kyseisistä toimenpiteistä onnistui apt-get-työkalun avulla, jolla asennettiin paketit **unzip**, **debian-archive-keyring**, **debootstrap**, **qemu-user-static** ja **schroot**. Poikkeuksena oli NXP:n BSP-paketti, joka ladattiin manuaalisesti NXP:n nettisivuilta. Tämän työn toteuttamisessa käytettiin BSP-paketin versiota L5.4.24_2.1.0_MX8MN [18].

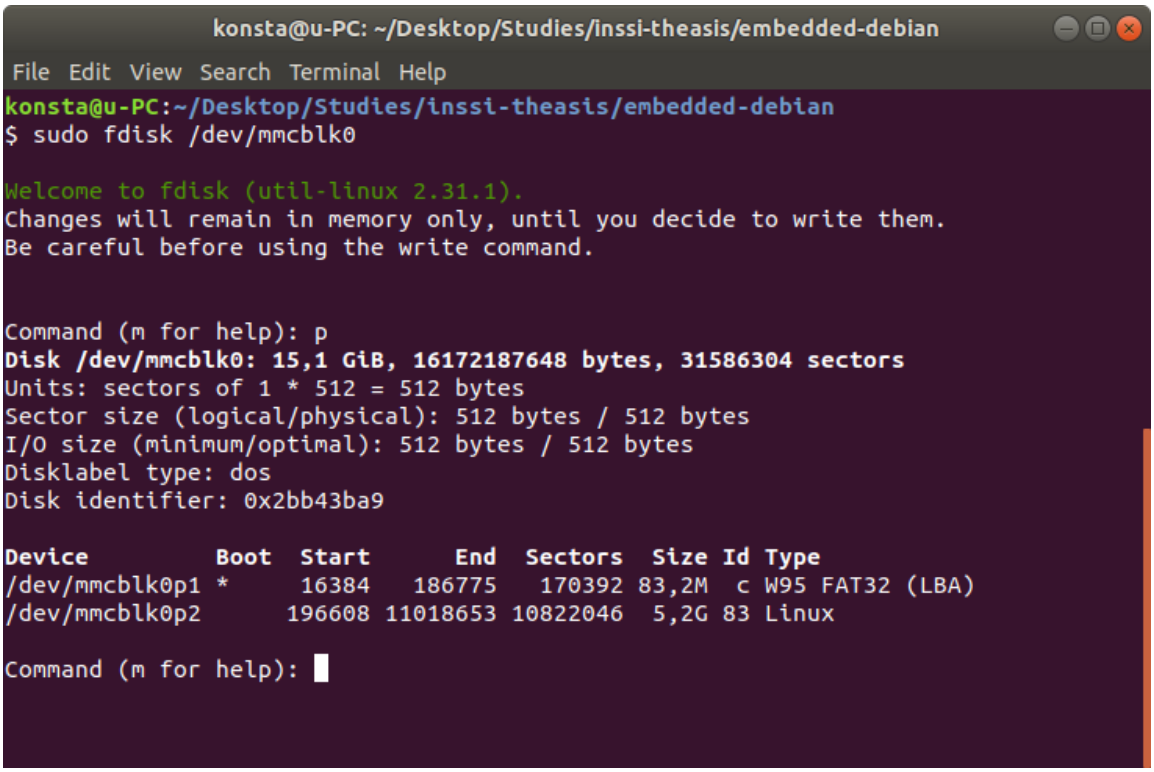
”Debian-archive-keyring”-asennuksen yhteydessä tuli vielä ajaa komento ”sudo apt-key add /usr/share/keyrings/debian-archive-keyring.gpg”. Kyseisellä avainketjulla mahdollistetaan, että debootstrap pystyy lataamaan Debian-juuritiedoston luonnissa tarvittavia paketteja. Avainketjun asentamisen jälkeen pystyi määritellyn Debian-jakelun lataamaan komennolla ”**sudo qemu-debootstrap --arch=arm64 --keyring /usr/share/keyrings/debian-archive-keyring.gpg --variant=buildd --exclude=debfoster buster ~/debian-arm64 http://ftp.debian.org/debian**”. Komennon tärkeitä osia ovat valitsin ”**--arch=arm64**”, jolla määritellään, että ladattavan juuritiedostojärjestelmän tulee olla yhteensopiva 64-bittisen ARM-arkkitehtuurin kanssa. Tämä siksi, että i.MX 8M Nano perustuu 64-bittiseen ARMv8-suoritinarkkitehtuuriversioon. Lisäksi käyttämällä valitsinta ”buster” voitiin määritellä ladattavan juuritiedostojärjestelmän tyypiksi Debian 10 -versio. [19.] [20.] [21.] [22.]

Lopuksi NXP:n sivuilta ladattu BSP-paketti purettiin komennolla ”**unzip L5.4.24-2.1.0_images_MX8MNEVK.zip imx-image-full-imx8mnevk.wic**”. Unzip-komennossa ensimmäinen ”**L5.4.24-2.1.0_images_MX8MNEVK.zip**”-osa on polku ladattuun ZIP-pakettiin. Toisella osalla ”**imx-image-full-imx8mnevk.wic**” määritellään purettavaksi vain kyseinen tiedosto ZIP-paketista [23]. Jos yksittäisen tiedoston määrittelyn jättää pois komennosta, puretaan tällöin koko ZIP-paketti.

5.1.2 Muistikortin alustus ja asennus

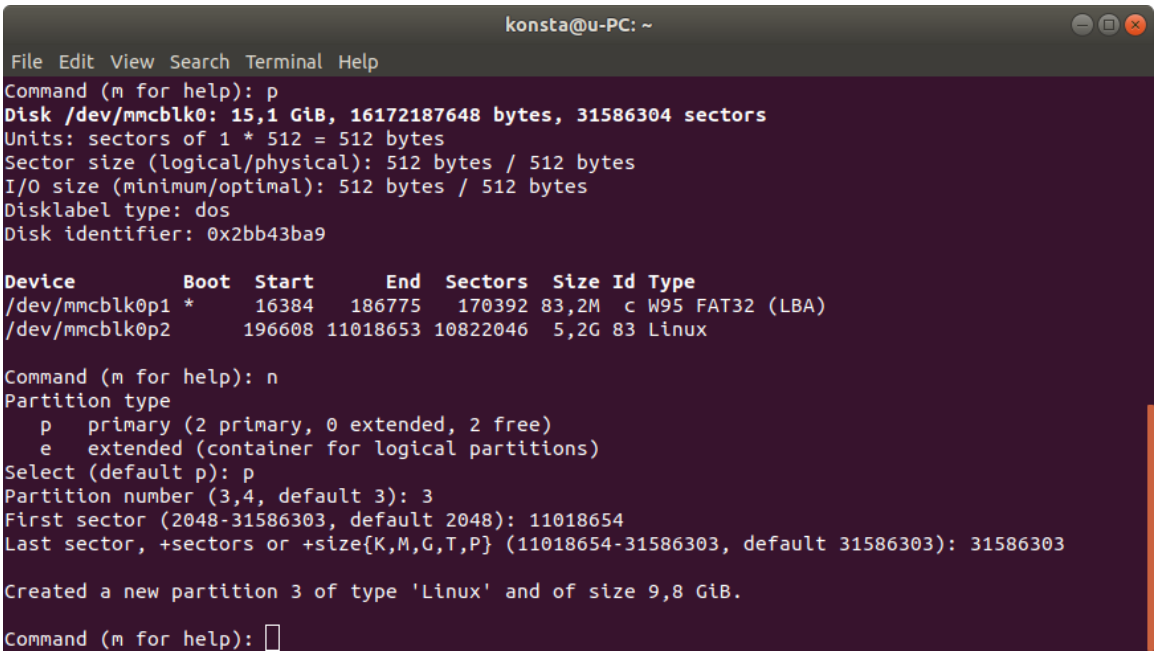
Ohjelmistokuvan kopiointiin ja hallintaan käytettiin Ubuntu-käyttöjärjestelmän sisältämiä dd- ja fdisk-komentoriviohjelmia. Ensimmäiseksi muistikortti liitettiin isäntäjärjestelmään, jonka jälkeen tälle kopioitiin BSP-paketista purettu järjestelmäkuva komennolla ”**sudo dd if=imx-image-full-imx8mnevk.wic of=/dev/mmcblk0 bs=1M status=progress**”. Koska kehitysympäristöstä oli käytössä DDR4-muistilla varustettu versio, täytyi BSP-paketista purkaa kyseisen muistityypin kanssa yhteensopiva versio U-boot-käynnistyslataajasta. Purkamiseen käytettiin Unzip-ohjelmaa komennolla ”**unzip L5.4.24-2.1.0_images_MX8MNEVK.zip imx-boot-imx8mnddr4evk-sd.bin-flash_ddr4_evk**”. Tämän jälkeen ajettiin komento ”**sudo dd if=imx-boot-imx8mnddr4evk-sd.bin-flash_ddr4_evk of=/dev/sdX bs=1k seek=32 conv=fsync**”, jolla kirjoitettiin kortille DDR4-muistityypin kanssa yhteensopiva versio U-boot-käynnistyslataajasta. [24, s. 8 ja s. 9] [25.]

Kopioinnin jälkeen muistikorttia hallinnoitiin fdisk-ohjelmalla, jolla luotiin uusi mmcblk0p3-niminen muistin osiointi. Fdisk-työkaluun pääsi ajamalla komennon ”**sudo fdisk /dev/mmcblk0**”, jossa ”/dev/mmcblk0” on isäntälaitteeseen yhdistettyyn muistikorttiin osoitettava laitetiedosto. Antamalla fdisk-ohjelmassa komennon ”p” saadaan listattua muistikortilla jo olevat osiot, jotka tulisi dd-komennon jäljiltä olla ”/dev/mmcblk0p1” ja ”/dev/mmcblk0p2” (kuva 8). [26.]



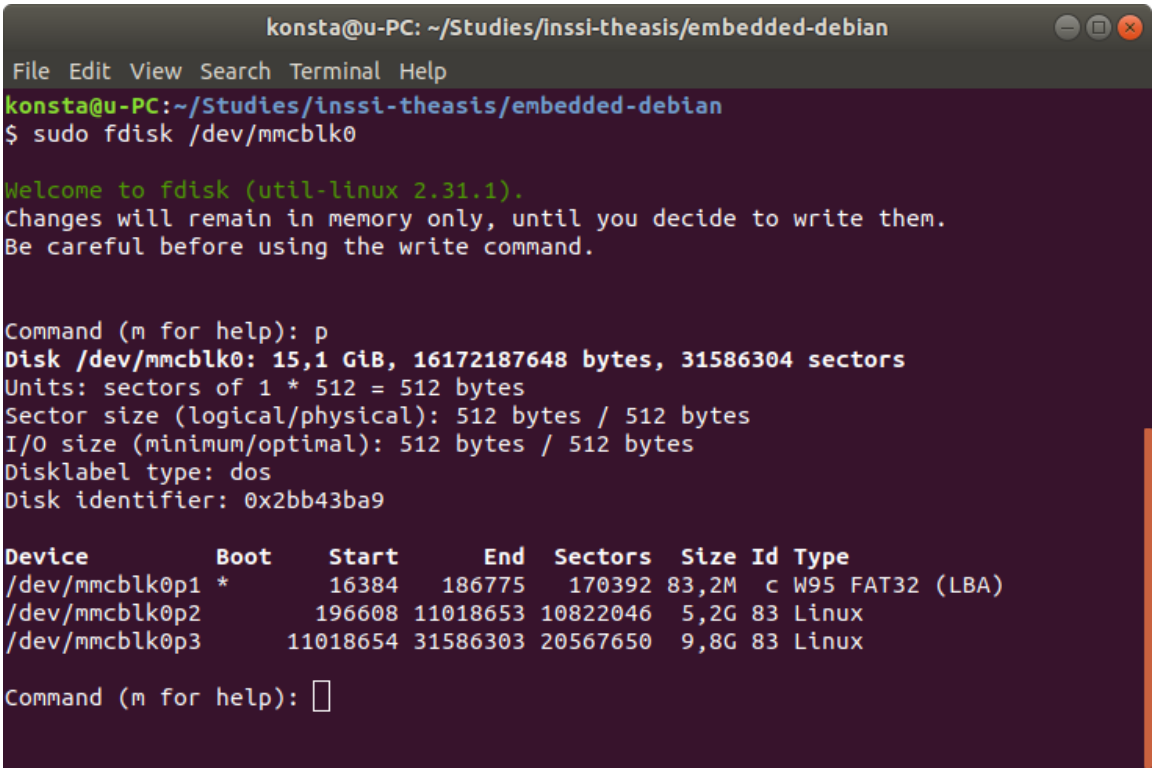
Kuva 8. Muistikortin sisältämät osiot.

Seuraavaksi fdisk-ohjelmassa ajettiin komento ”n”, joka aloittaa uuden osion luomisen. Uuden muistiosion tyypiksi valittiin ”primary” ja osion numeroksi luku kolme. Viimeisissä vaiheissa määritellään osion fyysinen koko osoittamalla fdiskille muistiosion aloitus- ja lopetussektorit (engl. Sector). Kolmas osiointi haluttiin alkavaksi heti toisen (mmcblk0p2) jälkeen. Tästä syystä fdisk-ohjelmassa annetaan kolmannen osion aloitussektoriksi oletusarvon (2048) sijasta sektori 11018654, koska aikaisemman osion alue päättyy sektoriin 11018653. Koska muistikortille ei tarvitse jättää tilaa tuleville osioille, käytetään lopetuspisteenä sektoria 31586303. Uuden osion luonnin vaiheet on esitelty kuvassa 9. [26.]



Kuva 9. Uuden muistiosion luonti.

Ajamalla komento ”p” uudestaan voitiin nähdä, että uusi osiointi oli nyt valmiina lisättäväksi muistikortille (kuva 10). Osiorakenteen muutosten tallentamiseksi tulee fdisk-ohjelmasta poistua komennolla ”w”, jonka jälkeen tehdyt muutokset astuvat vasta voimaan. Jos ohjelmasta poistuttaisiin komennolla ”q”, ei tehtyjä muutoksia tallennettaisi, vaan muistikortin aikaisemmat osiot pysyisivät ennallaan. [26.]



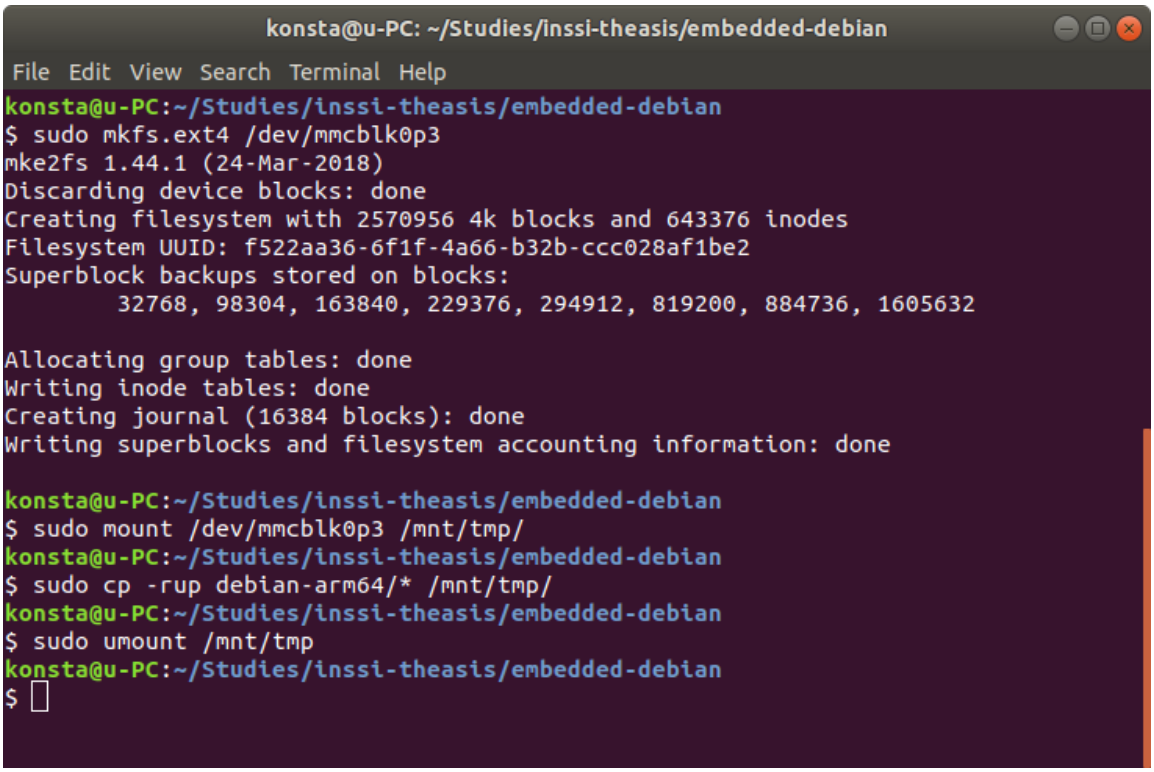
Kuva 10. Muistikortille lisätty uusi osiointi.

Viimeisenä vaiheena tuli alustaa tuore osiointi komennolla ”**sudo mkfs.ext4 /dev/mmcblk0p3**”. Kyseinen komento alustaa muistiosion EXT4-tyypin tiedostojärjestelmäksi, joka on viimeisin ja suorituskykyisin Linux-jakeluissa käytettävä tiedostojärjestelmätyyppi. Tämä sisältää, EXT3-tyypin ohessa, ominaisuuksia, jotka lisäävät järjestelmän selviämisen todennäköisyyttä mahdollisista vikatilanteista. Mahdollisuuksien mukaan on Linux-järjestelmissä suositeltavaa käyttää EXT3- tai EXT4-tiedostojärjestelmää. [27.] [28.] [29.]

Uuden muistiosion alustamisen jälkeen kopiotiin tähän Debian-juuritiedostorakenne. Kopioitavaksi tiedostorakenteeksi ei kuitenkaan kelpaa mikä tahansa Debianin versio, vaan sen täytyy olla ARMv8-arkkitehtuurin kanssa yhteensopiva. Tämä varmistettiin aikaisemmassa vaiheessa käyttämällä qemu-debootstrap-komennon kanssa komentoriviargumenttia ”--arch=arm64”.

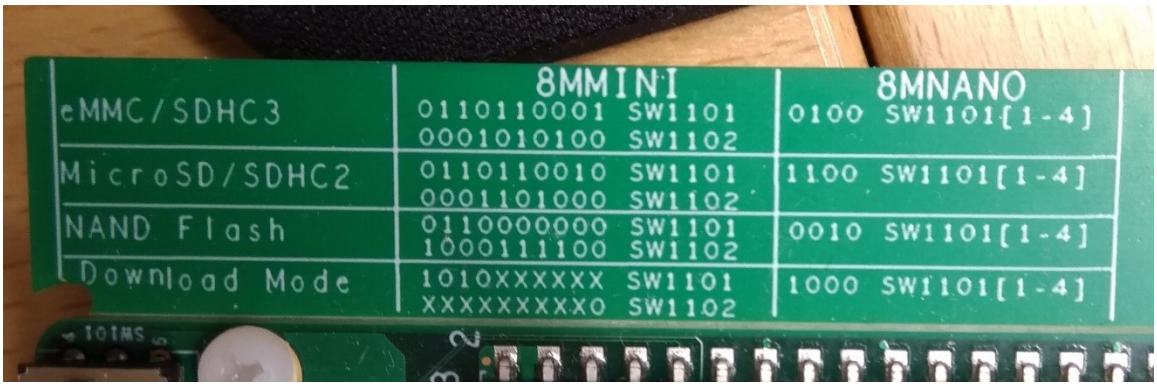
Juuritiedostorakenteen kopioinnissa käytettiin komentorivityökaluja mount ja cp. Näistä mount-komennolla voidaan liittää ulkoisen laitteen, kuten muistikortin tai kiintolevyn, tiedostojärjestelmä osaksi Linux-jakelun tiedostojärjestelmää, jonka jälkeen tiedostoja voidaan siirtää näiden välillä [30].

”/dev/mmcblk0p3”-muistiosio liitettiin ”/mnt/tmp”-sijaintiin komennolla ”sudo mount /dev/mmcblk0p3 /mnt/tmp”. Tämän jälkeen Debian-juuritiedostojärjestelmä voitiin kopioida luodulle osiolle komennolla ”**sudo cp –rup debian-arm64/* /mnt/tmp**”. Kopioinnissa kesti hetki, eikä tämän edistymisestä tarjottu mitään tulostetta. Edistymisestä kertovat tulosteet saisi kuitenkin päälle lisäämällä komentoon vivun ”-v”. Lopuksi liitetty tiedostojärjestelmä irrotettiin komennolla ”**sudo umount /mnt/tmp**” (kuva 11). [31.] [32.]

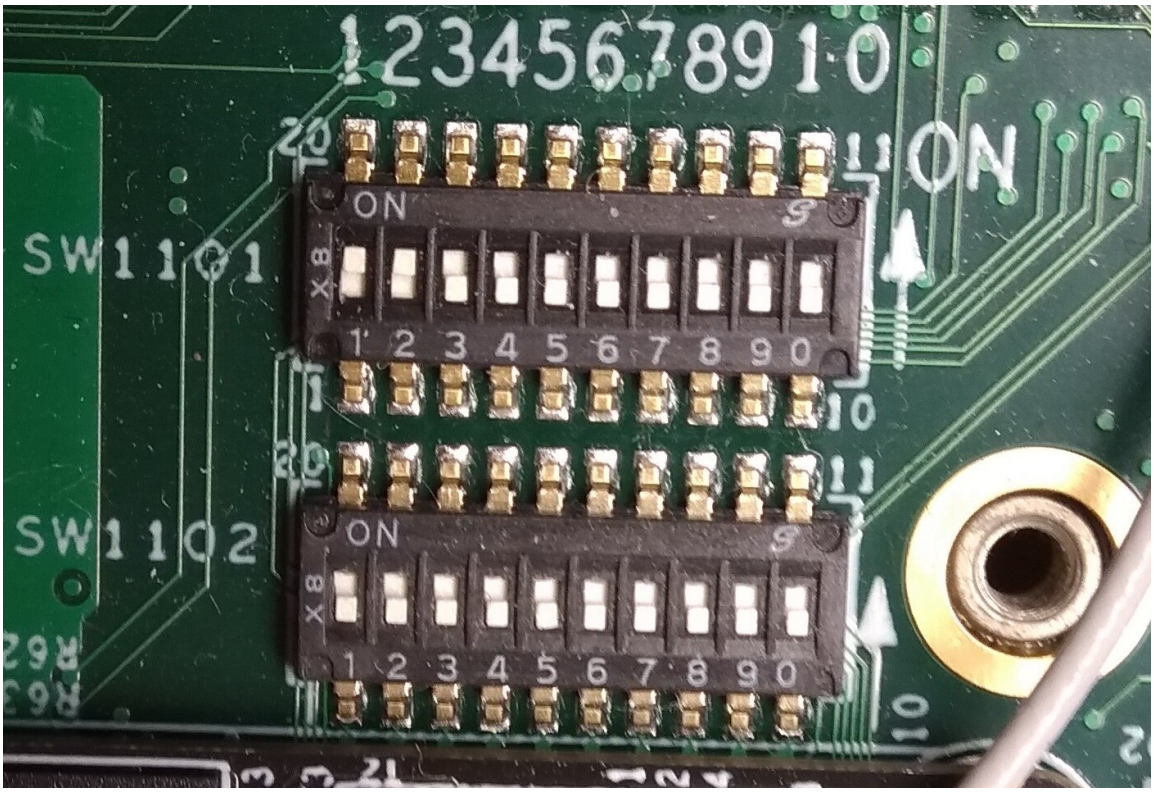


Kuva 11. mkfs-, mount- ja cp-komennot.

Debianin kopioimisen jälkeen muistikortti asetettiin i.MX-kehitysalustan muistikortinlukijaan. Tässä vaiheessa tuli myös muuttaa kehitysalustan ”BOOT Switches” -kytkinten asentoa, joilla valitaan, mistä muistityypistä käynnistys suoritetaan. Kytkinkombinaatioita vastaavat muistilähteet on merkitty kehitysalustan piirilevyyn, josta tässä tapauksessa haluttiin valita muistilaitteeksi MicroSD/SDHC2 (kuva 12). Tämä tapahtui kytkemällä SW1101-kytkinrivistön kaksi ensimmäistä kytkintä ON-asentoon (kuva 13).



Kuva 12. Käynnistysvaihtoehdot.



Kuva 13. ”Boot switch”-valintakytkimet, joissa valittuna MicroSD-muistilähde.

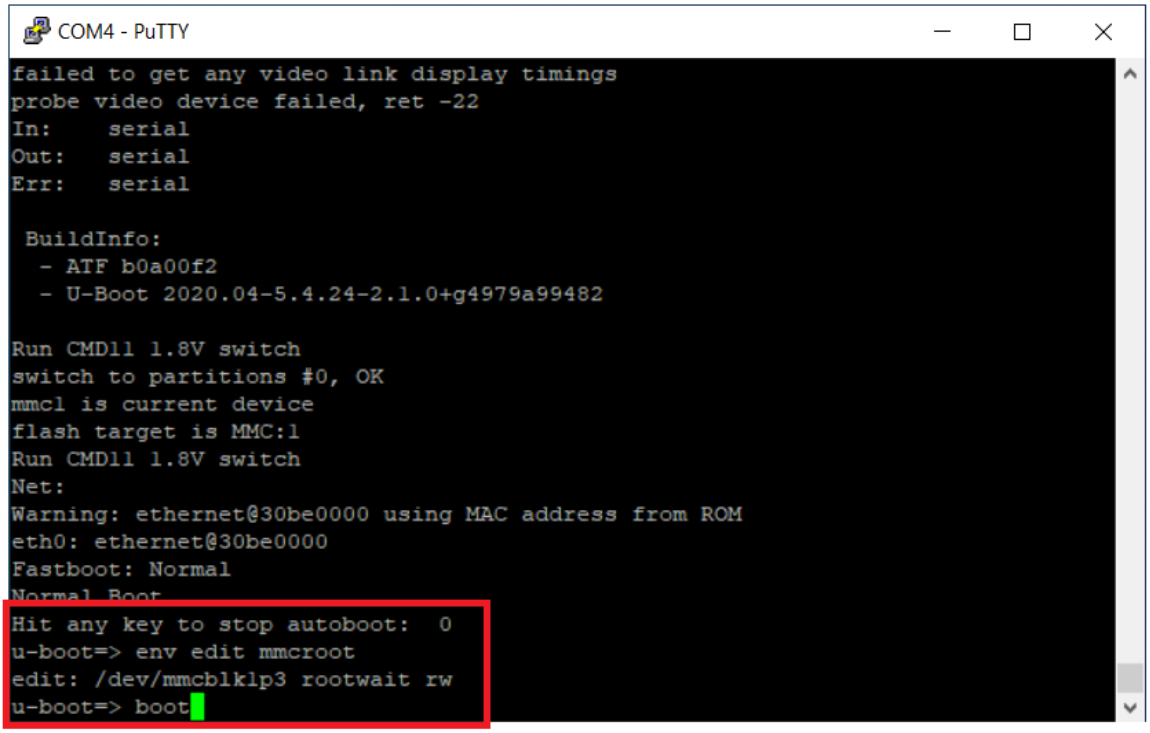
5.1.3 Kehitysalustan käynnistäminen ja loppuvalmistelu

i.MX-kehitysalusta sisältää kaksi USB C -tyypin liitäntää. Näistä toinen, RJ-45-liitintä lähinnä, on tarkoitettu virtalähteen liittämiseksi kehitysalustaan. Virtalähdettä ei tule liittää muihin USB C -tyypin liittimiin. Käytetty isäntätietokone kytkettiin i.MX-kehitysalustan Micro USB Debug -sarjaliikenneporttiin, koska tätä kautta saa tietoa kehitysalustan käynnistyksestä sekä pystyy hallitsemaan käynnistyvää järjestelmää. Tietokoneella täytyi olla asennettuna sarjaliikenneikkunaohjelmisto, jotta i.MX:n käynnistystä pystyi seuraamaan ja hallitsemaan. Tässä tapauksessa käytettiin Putty-nimistä ohjelmistoa Windows 10 -käyttöjärjestelmässä. Linux ympäristössä vastaavassa tehtävässä toimisi esimerkiksi minicom-niminen ohjelma. [33.]

Kehitysympäristö käynnistyi oletuksena Yocto-projektilla luotuun Linux-jakeluun. Sisäänkirjautuminen tapahtui kirjoittamalla käyttäjätunnukseksi ”root”. Tämän jälkeen ”/dev/mmclk1p3”-niminen muistiosio liitettiin käyttöjärjestelmän ”/mnt”-sijaintiin mount-komennolla. Kyseisen osion tulisi sisältää aikaisemmin valmisteltu Debian-jakelu, jonne seuraavaksi kopioitiin Yocton sijainnit /lib/modules ja /lib/firmware ”cp”-työkalulla.

Kun kehitysalustan toimintaan tarvittavat tiedostot oli siirretty Debianin tiedostorakenteeseen, ohjeen viimeisenä vaiheena oli siirtyä Yocton tiedostojärjestelmästä Debianiin ajamalla komento ”**chroot /mnt**”. Tämän jälkeen pystyttiin ajamaan Debianin tiedostojärjestelmän sisältämiä komentoja ja muokkaamaan tämän sisältämiä tiedostoja. Tässä vaiheessa tehtiin myös loput konfiguroinnit ja ohjelmien asentamiset, jotta Debianin kaikki ominaisuudet olisivat käytettävissä, kun se seuraavaksi käynnistetään. Tärkeimpinä oli **ifupdown**-, **net-tools**-, **network-manager**-, **udev**-, **sudo**- ja **ssh**-pakettien asentaminen apt-get-ohjelmalla. Ennen Debianin tiedostojärjestelmästä poistumista konfiguroitiin vielä järjestelmän isäntänimi ja ifupdown-verkkoliittymän asetukset. Ohjeistus ei sisältänyt käyttäjän luontia, mutta tämä kannatti myös tehdä ennen Debianin tiedostojärjestelmästä poistumista adduser-komennolla.

Kun Debianista puuttuneet ohjelmat ja asetukset oli asennettu, poistuttiin chroot-tilasta takaisin Yocton puolelle ajamalla komento ”**exit**” tai antamalla näppäinyhdistelmä Ctrl + D. Yocton puolella ajettiin komento ”**reboot**” järjestelmän uudelleenkäynnistämiseksi ja siirtymiseksi Debian-puolelle. Uudelleenkäynnistys pysäytettiin U-Boot-käynnistyslataajaan painamalla satunnaista näppäimistön merkkiä (kuva 14). Tämän jälkeen avattiin U-bootin mmcroot-ympäristömuuttuja muokattavaksi komennolla ”**env edit mmcroot**”. Muuttamalla juuritiedoston muistilohko osiosta /dev/mmcb1k1p2 /dev/mmcb1k1p3-nimiseksi osioksi voidaan muuttaa U-Boot käynnistämään Debian-käyttöjärjestelmä Yocton sijasta. Lopuksi ajettiin komento ”**boot**” käynnistyksen jatkamiseksi Debianiin



Kuva 14. Käynnistyslohkon valinta U-boot käynnistyslataajassa.

Tulee huomioida, että kyseinen käynnistysosion valintatoimenpide on mahdollista vain i.MX 8M Nano EVK -kehitysalustan Debug-sarjaliikenneportin kautta, joka yhdistetään tietokoneeseen micro-USB-johdolla. Samainen terminaalinäkymä soveltuu myöhemmin myös itse käyttöjärjestelmän ohjaamiseen.

5.2 Debian-käyttöjärjestelmän asentaminen ilman Yoctoa

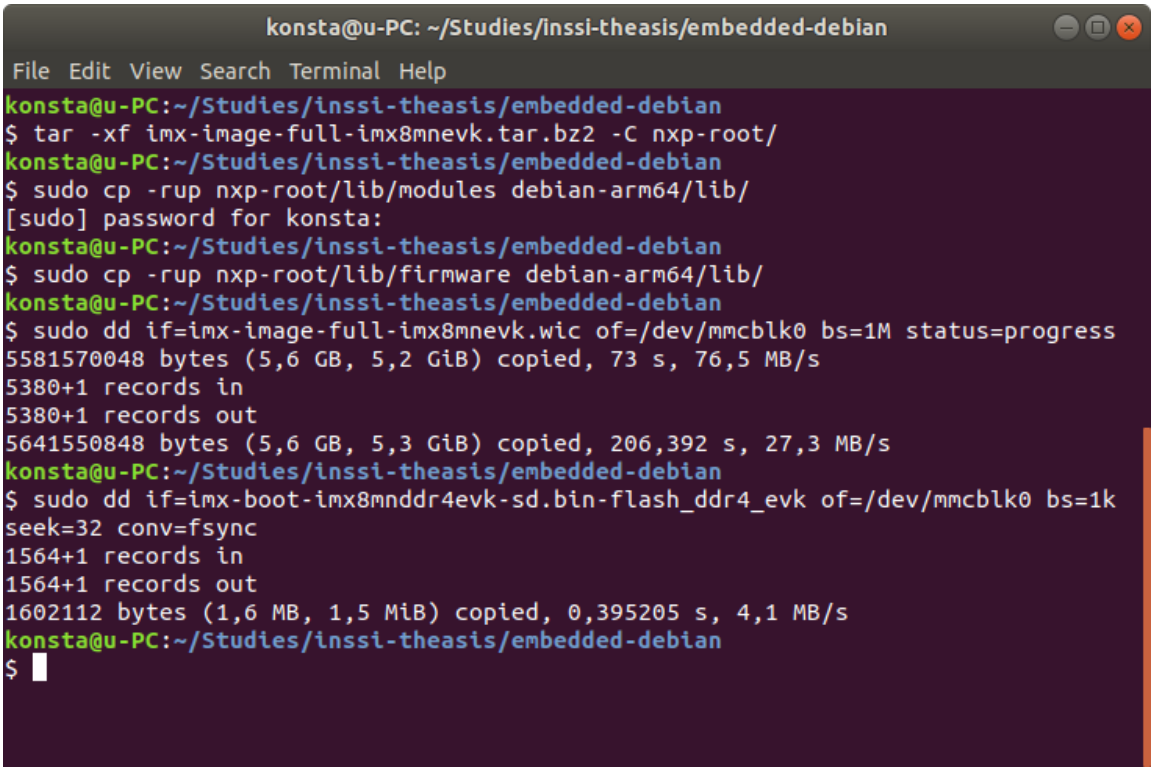
Lopullisessa versiossa on käytännöllisempää asettaa i.MX käynnistymään suoraan Debian-käyttöjärjestelmään ilman tarvetta muokkauksille kesken käynnistysprosessin. Yksi tapa kyseisen toiminnallisuuden aikaansaamiseksi olisi muuttaa U-Boot-käynnistyslataajan oletusarvoja. Tämä kuitenkin vaatisi U-Bootin uudelleenkääntämistä ja asentamista uudestaan muistikortille, mikä ei ole kovin käytännöllistä. Yleensä tuotteen kohdalla ei ole järkevää sisällyttää järjestelmän muistiin kahta erilaista versiota käyttöjärjestelmästä ilman pätevää syytä. Tämän takia kannattaa Yocto-version asentaminen jättää kokonaan pois, jolloin vapaaksi jäävälle muistin osiolle voidaan asentaa valmiiksi konfiguroitu Debian-jakelu. Näin U-Boot lataa oletusarvoisesti Debian-käyttöjärjestelmän käynnistyksessä Yocto-pohjaisen sijasta. Debian kuitenkin tarvitsee Yocton juuritiedostojärjestelmän sisältämiä tiedostoja käyttääkseen joitakin kehitysalustan tarjoamia ominaisuuksia.

Valmiiksi konfiguroidun Debianin saamiseksi löytyi muutama ratkaisu. Ensimmäisenä ratkaisuna voi Yocto-puolella konfiguroidusta Debianista tehdä kopion dd-komennolla, tämän jälkeen uudelleen järjestellä osioita fdisk-ohjelmalla ja asentaa Debian myöhemmin Yocton tilalle oikeaan muistin osioon. Jos käyttöjärjestelmästä halutaan toteuttaa uudempi versio, on edellä mainittu kuitenkin hidas ja työläs menetelmä Debianin päivittämiseksi. Myöhemmässä vaiheessa havaittiin, että ohjeessa ilmoitetut Debian järjestelmän konfiguraatiot voi suorittaa jo isäntäjärjestelmässä, jolloin Yocto-pohjaisen jakelun asennusta muistikortille ei periaatteessa tarvita. Debianin juuritiedostoon täytyy kuitenkin kopioida tärkeät /lib/modules- ja /lib/firmware-sijainnit tiedostoineen. Tarvittavat tiedostot voi kuitenkin erotella BSP-paketin sisältämästä ”imx-image-full-imx8mnevk.tar.bz2” tervapallosta (engl. Tarball), joka sisältää pakatussa muodossa Yocto-pohjaisen jakelun juuritiedostot.

5.2.1 Debianin muokkaus ja konfigurointi isäntäjärjestelmässä

Alkuvalmisteluina täytyi erotella luvussa 5.1 listattujen ” imx-image-full-imx8mnevk.wic”- ja ”imx-boot-imx8mnddr4evk-sd.bin-flash_ddr4_evk”-tiedostojen lisäksi ”imx-image-full-imx8mnevk.tar.bz2” niminen tervapallo ZIP-paketista. Tervapallon sisältämä juuritiedosto-rakenne purettiin sijaintiin ”nxp-root/” komennolla ”**tar –xf imx-image-full-imx8mnevk.tar.bz2 –C nxp-root/**”.

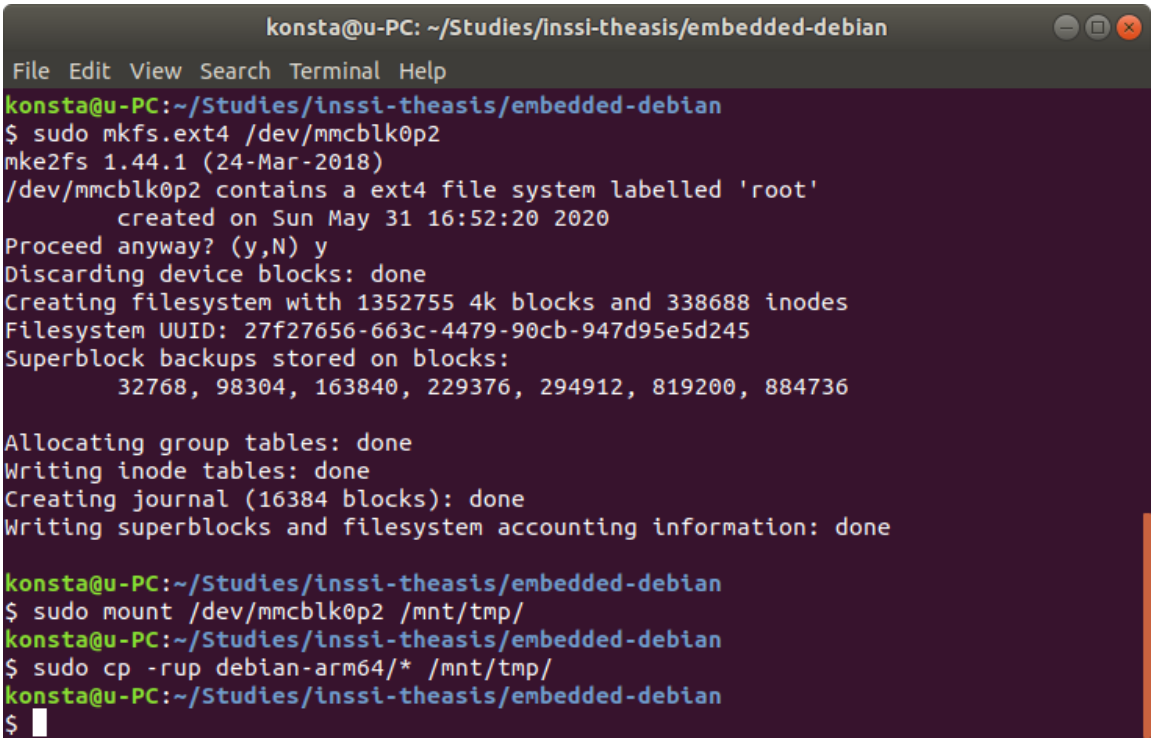
Nxp-root-sijaintiin puretusta tiedostorakenteesta kopioitiin sijaintien ”nxp-root/lib/modules/” ja ”nxp-root/lib/firmware/” sisältämät tiedot näitä vastaaviin sijainteihin ”debian-arm64/lib/”-sijainnin alle cp-komennolla. Tämän jälkeen voitiin valmistella muistikortti kopioimalla tälle dd-komennolla BSP-paketin sisältämä kokonaiskuvatiedosto ja DDR4-muistin kanssa yhteensopiva U-boot-versio (kuva 15).



Kuva 15. Debian-rootfs ja muistikortin esivalmistelu.

5.2.2 Debianin asentaminen ja asennuksen viimeistely isäntäjärjestelmässä

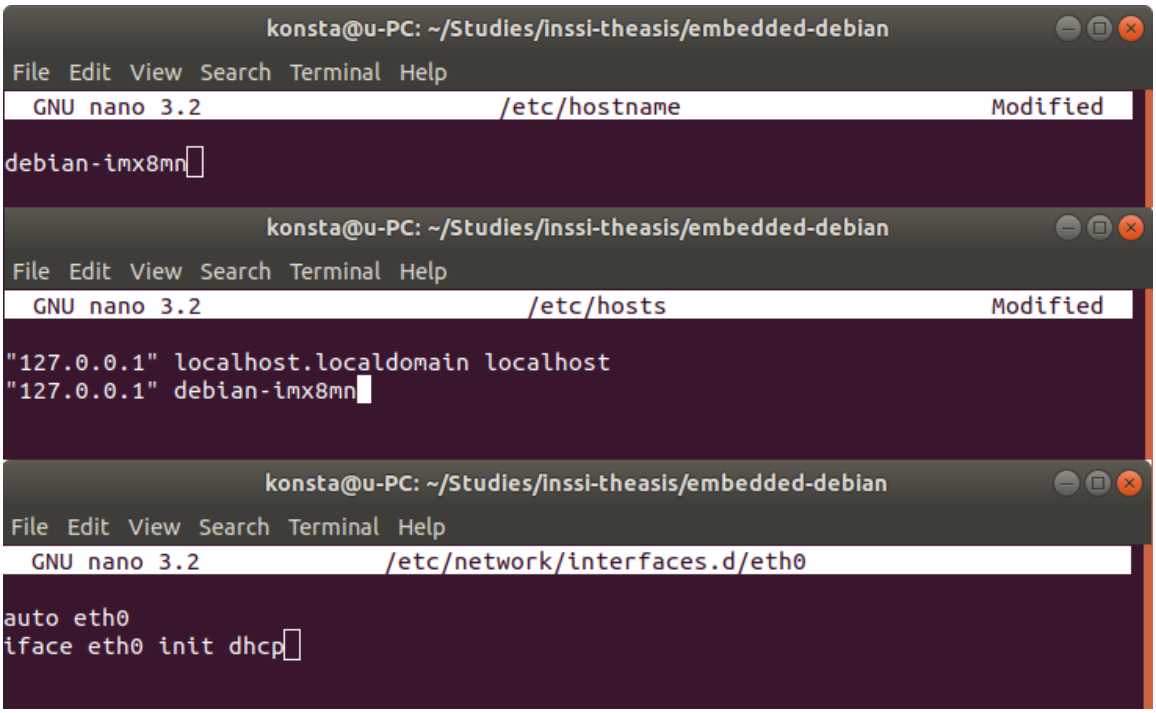
Seuraavaksi dd-komennon ajamisen jälkeen ”/dev/mmcblk0p2” osio alustettiin ext4-tyyppiseksi tiedostojärjestelmäksi mkfs.ext4-komennolla. Tiedostojärjestelmän tyypin muutoksen yhteydessä myös dd-komennolla asentunut Yocton juuritiedostojärjestelmä poistetaan kyseisestä osiosta. Alustuksen jälkeen muokattu Debian-juuritiedostojärjestelmä kopioitiin muistikortin mmcblk0p2-osiolle mount- ja cp-komentoja käyttäen (kuva 16). Kun tiedostot oli kopioitu, irrotettiin liitetty osion tiedostojärjestelmä isäntäkoneen juuritiedostojärjestelmästä umount-komennolla.



Kuva 16. Muistiosion alustus ja Debian-tiedostojärjestelmän kopiointi.

Ajamalla isäntäjärjestelmässä komento ”**chroot**” voidaan myös siirtyä luodun Debian-käyttöjärjestelmän sisälle ja tällä tavalla suorittaa samat asennukset, jotka aikaisemmin jouduttiin tekemään Yocton puolella. Jotta pakettien lataaminen olisi mahdollista apt-get-komennolla, täytyi ensimmäisenä määritellä DNS, jotta komennon pyynnöt osattaisiin reitittää oikein. Tämä tapahtui ajamalla komento ”**echo nameserver 8.8.8.8 >> /etc/resolv.conf**”.

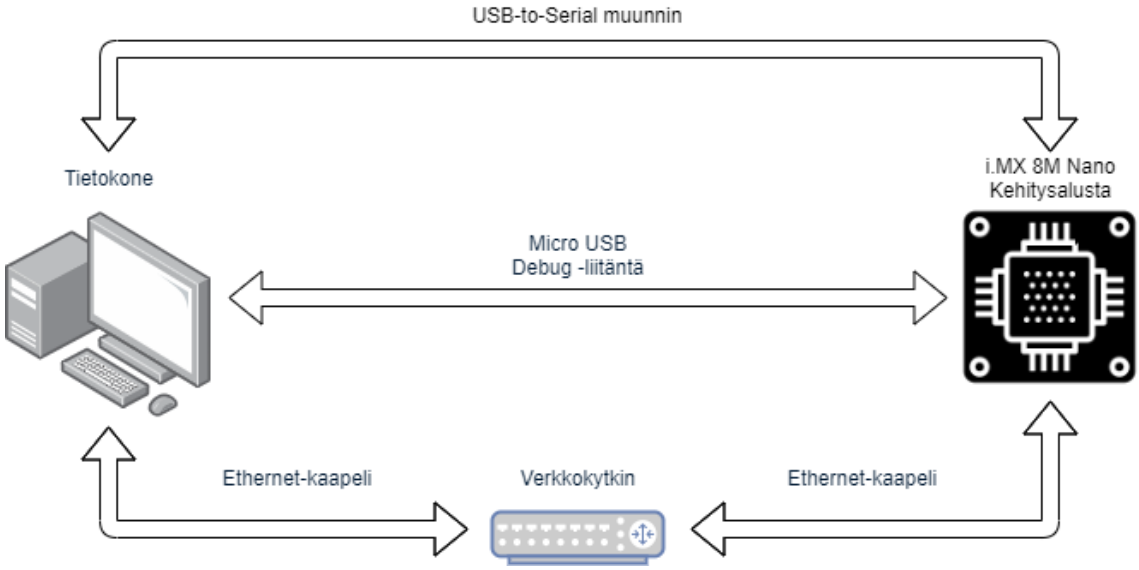
Ennen tarvittavien pakettien asentamista ajettiin vielä ”**apt-get update && apt-get upgrade”**-komento, jonka jälkeen ”**apt-get installa**”-komennolla asennettiin paketit **ifupdown**, **net-tools**, **network-manager**, **udev**, **sudo**, **ssh** ja **nano**. Asennuksen viimeistelyksi määriteltiin vielä järjestelmän isäntänimi- (engl. Hostname) ja verkkoliittymäasetukset tiedostoihin ”/etc/hostname”, ”etc/hosts” ja ”/etc/network/interfaces.d/eth0” nano-sovelluksella (kuva 17). Lopuksi luotiin uusi käyttäjä adduser-komennolla. Chroot-tilasta pääsi poistumaan komennolla ”**exit**” tai näppäinyhdistelmällä Ctrl + D, jonka jälkeen muistikortilla oli i.MX-kehitysalustassa toimiva Debian-jakelu.



Kuva 17. hostname, hosts ja eth0 konfigurointi.

5.3 Kehitysympäristö

Onnistuneen Debian-käyttöjärjestelmän asennuksen jälkeen toteutettiin Djangon kehityksessä hyödynnettävä kehitysympäristö. Kehityksessä keskeisessä osassa oli sarjaliikenteen käyttäminen, jossa hyödynnettiin USB-to-Serial-muunninta mahdollistamaan sarjaliikennekommunikaatio tietokoneen ja i.MX-kehitysalustan välille. Toisena tärkeänä osana oli LAN-verkkoyhteys laitteiden välille, joka toteutettiin kytkemällä i.MX-kehitysalusta samaan verkkokytkimeen kehitystietokoneen kanssa. Lisäksi laitteiden välillä oli vielä toinen Debug-sarjaväyläyhteys, jota käytettiin kehitysalustan ohjaamiseen (kuva 18).



Kuva 18. Kehityksessä käytetyt kytkennät.

Tietokoneelle toteutetussa kehitysympäristössä hyödynnettiin Visual Studio Code -tekstinkäsittelyohjelmaa, Git-Bash Linux-terminaaliemulaattoria, Putty-sarjaliikennemonitoria ja itsetoteutettua Shell-skriptiä tiedostojen päivittämiseksi laitteiden välillä. Skripti hyödyntää scp- tai rsynk-ohjelmia projektitiedostojen siirtämiseen tietokoneen ja kehitysalustan välillä. Skripti vaatii toimiakseen LAN-yhteyden laitteiden välille ja ssh-avainparin luomista. Muuten skriptissä käytetyt ohjelmat, kuten scp, ovat käytännössä Linux-jakeluista löytyviä standardiohjelmia. Git-Bash-terminaaliemulaattorin mukana tulee suurin osa samoista tarvittavista ohjelmista poisluettuna rsync, jonka voi tarvittaessa asentaa osaksi emulaattoria.

5.4 Django-sovellus

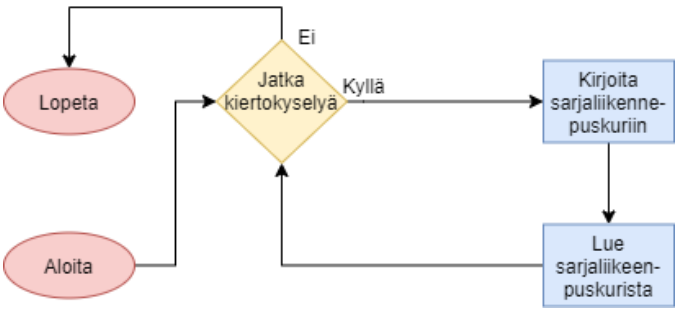
Django-pohjainen verkkosovellus oli koko kehitettävän sarjaliikennepalvelimen ydin. Verkkosovelluksen käyttöliittymän ylläpidon lisäksi tuli Djangoon kehittää tehokas tapa käsitellä sarjaliikenteen yli tapahtuvaa kommunikaatiota. Sarjaliikennetoteutuksen tuli olla mahdollisimman nopea ja pystyä käsittelemään useamman käyttäjän yhtäaikaisia pyyntöjä Djangolle.

5.4.1 Sarjaliikennekommunikaation kehitys

Django-sovelluksella oli tarkoitus toteuttaa sarjaliikennepalvelimelle ominainen sarjaliikennemuotoisen tiedon muuttaminen formaatista toiseen. Toisena tärkeä osa oli käyttöliittymän luominen, jonka kautta sarjaliikenneväylään liitettyä laitetta voisi hallita. Sarjaliikennekommunikaation tapauksessa käytettiin RS-232-protokollaan perustuvaa TTL-tasossa toimivaa varianttia, jolloin oli käytännöllistä hyödyntää pySerial-nimistä Python-moduulia, joka häivyttää kätevästi RS-232-protokollan toteutuksen Serial-olion ja tämän read- sekä write-metodikutsujen taakse. [34.]

Rakennettaessa sarjaliikennekommunikaatiota pySerial-moduulin avulla on hyvä huomioida muutama moduulin aiheuttama rajoitus. Moduuli ei tarjoa signaaleja, joita voisi hyödyntää alemman tason rauta-ajureiden kanssa viestimiseen. Tämän takia tiedon lukeminen ja kirjoittaminen sarjaliikennepuskuriin on toteutettava niin kutsutun ”polling”- eli kiertokyselyperiaatteen avulla.

Kiertokyselyssä haluttuja toimintoja ajetaan toistorakenteessa käyden vuorotelleen kaikki toimintovaihtoehdot läpi (kuva 19). Sarjaliikenteen tapauksessa näitä ovat tiedon lukeminen sarjaliikennepuskurista ohjelman käyttöön sekä tiedon kirjoittaminen sarjaliikennepuskuriin tiedon välittämiseksi väylän takana odottavalle laitteelle.

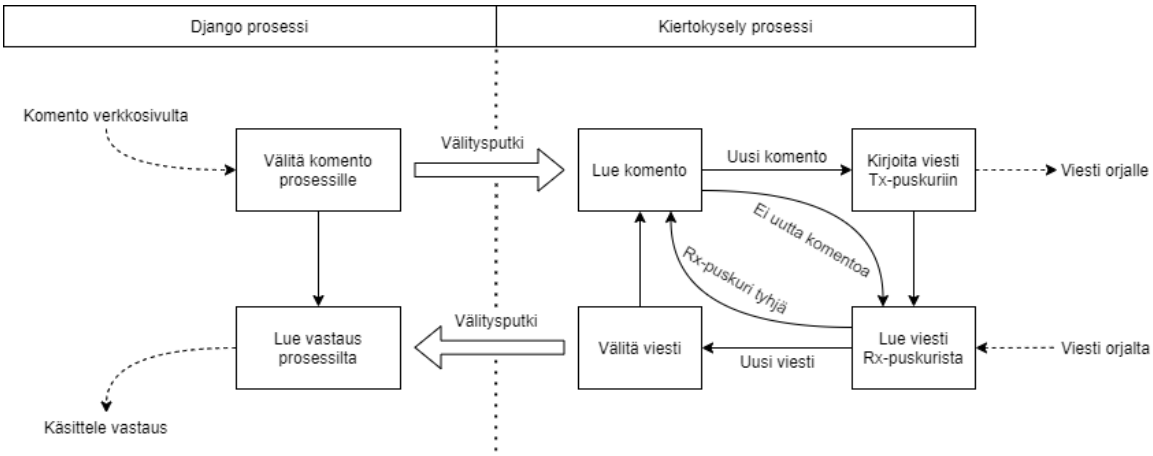


Kuva 19. Sarjaliikenteen kiertokysely.

Jos tiedon välittämistä sarjaliikenneväylän yli ei synkronoida mitenkään eli kommunikaatio toteutetaan niin sanotusti synkronoimattomasti (engl. Asynchronic), täytyy saapuvien viestien sarjaliikennepuskuria kiertokysellä jatkuvasti, jotta saapunut tieto saadaan mahdollisimman nopeasti pääohjelman käsiteltäväksi. Kiertokyselyn ongelmana on kuitenkin sen suuri prosessoriajan vaatimus, joka näkyy nopeasti muiden prosessisäikeiden hidastumisena. Tämän takia synkronoimattomissa toteutuksissa tulisi pyrkiä hyödyntämään niin sanottua tapahtumakäsittelyä, jossa ei aktiivisesti ajeta toistorakenteen sisälle määriteltyjä toimintoja vuoron perään, vaan ohjelma odottaa tapahtumia (engl. event), jotka käsitellään niiden sattuessa. Etenkin monimutkaisissa ohjelmatoteutuksissa kyseinen lähestyminen on suositeltavaa, koska tällöin suorittimen aikaa käytetään toiminnon suorittamiseen vain sen ollessa välttämätöntä. Tällaisessa sarjaliikennekommunikaation toteutuksessa laitteiston, ajureiden ja ohjelmiston tulisi siis tukea keskeytyksiä tai signaaleita, joilla tällainen tapahtumapohjainen lähestyminen voitaisiin toteuttaa. Tehokkuudestaan huolimatta kyseisen tyyppisen toteutuksen kehittäminen vaatii huomattavasti enemmän aikaa ja vaivaa.

Django-sovelluksessa sarjaliikennekommunikaatiota ohjaava sarjaliikennekäsittelijä yritettiin aluksi toteuttaa synkronoimattomana toteutuksena. Jotta kiertokysely ei söisi prosessoriaikaa Djangon omalta prosessilta, päätettiin kiertokysely toteuttaa omana lapsiprosessinaan käyttämällä hyödyksi Pythonin standardikirjastoon kuuluvaa Multiprocessing-moduulia. Multiprocessing-moduulia käyttämällä voidaan Pythonilla luoda uusia prosesseja käyttöjärjestelmään.

Python-prosessin etuna säikeisiin on se, että luotu aliohjelma toimii omana uutena Python-ohjelmana, jolloin käyttöjärjestelmän vaiheittainen ohjaus voi asettaa kyseisen prosessin toimimaan omassa prosessoriytimessään [35]. Pythonin säikeiden tapauksessa kuorman jakaminen useammalle prosessoriytimelle ei ole mahdollista, toisin kuin aitojen säikeiden kanssa, niin kutsutun GIL mekanismin takia [36]. GIL-mekanismista johtuen Pythonohjelman säikeitä ajetaan ”aidon säikeen” tai prosessin sisällä toiminto kerrallaan, jotta vältyttäisiin mahdollisilta ongelmilta, jotka johtuvat Pythonin toiminnasta ja toteutustavasta [37] [38]. Kuvassa 20 on pyritty esittämään ensimmäisen sarjaliikennekäsittelijän toimintaa ja toteutusta.

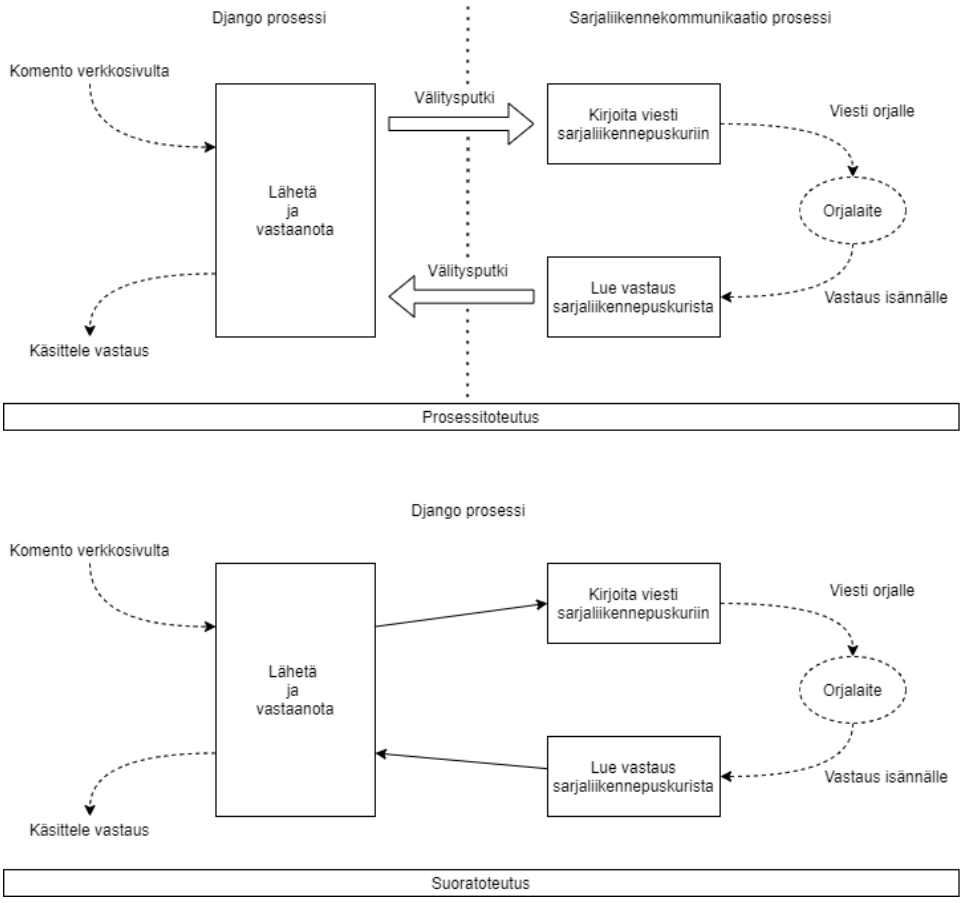


Kuva 20. Synkronoimaton toteutusversio sarjaliikennekäsittelijästä.

Lapsiprosessin käytön yhtenä haittana on sen säikeitä selvästi hitaampi luontinopeus. Toisin kuin säikeillä, aliohjelmat eivät jaa samaa muistialuetta isäntäprosessin kanssa, vaan koko isäntäprosessin muistialueesta tehdään kopio, jossa aliprosessia sitten suoritetaan. Tämä ohjelmaprosessin uudelleen kopioituminen siis aiheuttaa aliprosessin luonnin hitauden verrattuna säikeeseen. Lisäksi erilliset muistialueet rajoittavat, miten tietoa pystytään jakamaan kahden prosessin välillä. Säikeillä tietoa voidaan jakaa suoraan samoja muuttujia käyttämällä, koska säikeet toimivat samassa varatussa tilassa muistia. Kuitenkin prosessien välisessä kommunikaatiossa joudutaan hyödyntämään IPC-toteutuksia tiedon välittämiseksi isäntä- ja lapsiprosessin välillä.

Kehityksen myöhemmässä vaiheessa todettiin, että synkronoimattomalle kommunikaatiolle ei ole tarvetta, jolloin sarjaväylän laitteiden välinen kommunikaatio kannatti toteuttaa Master-Slave-tyyppisesti. Tämä poisti myös aikaisemman ongelman, joka aiheutui sarjaliikenteen kiertokyselystä. Master-Slave-tyyppisessä toteutuksessa isäntälaite, tässä tapauksessa sarjaliikennepalvelin, lähettää sarjaväylän päässä odottavalle orjalaitteelle pyynnön, johon laitteen tulee aina vastata saatuaan isännältä viestin. Kyseisellä menetelmällä sarjaliikennekäsittelijä on aktiivisena vain, kun viesti täytyy välittää isäntälaitteelta orjalaitteelle. Tällaisella toteutuksella voidaan välttää tilanne, jossa koko ajan aktiivisena toimiva kiertokysely söisi suoritusaikaa muilta prosesseilta.

Loppuvaiheessa päädyttiin luomaan sarjaliikennekommunikaatiosta kaksi versiota, joista ensimmäistä kutsutaan suoratoteutukseksi ja toista prosessitoteutukseksi. Suoratoteutuksessa sarjaliikennekommunikaatio suoritetaan samassa prosessisäikeessä Djangon pääprosessin kanssa, kun taas prosessitoteutuksessa itse viestin välitys ja vastaanottaminen toteutetaan erillisessä prosessissa. Yleisesti toteutusversiot muistuttavat toisiaan perustoiminnaltaan. Kuvassa 21 on pyritty esittämään näiden kahden toteutuksen toimintaa ja eroja.



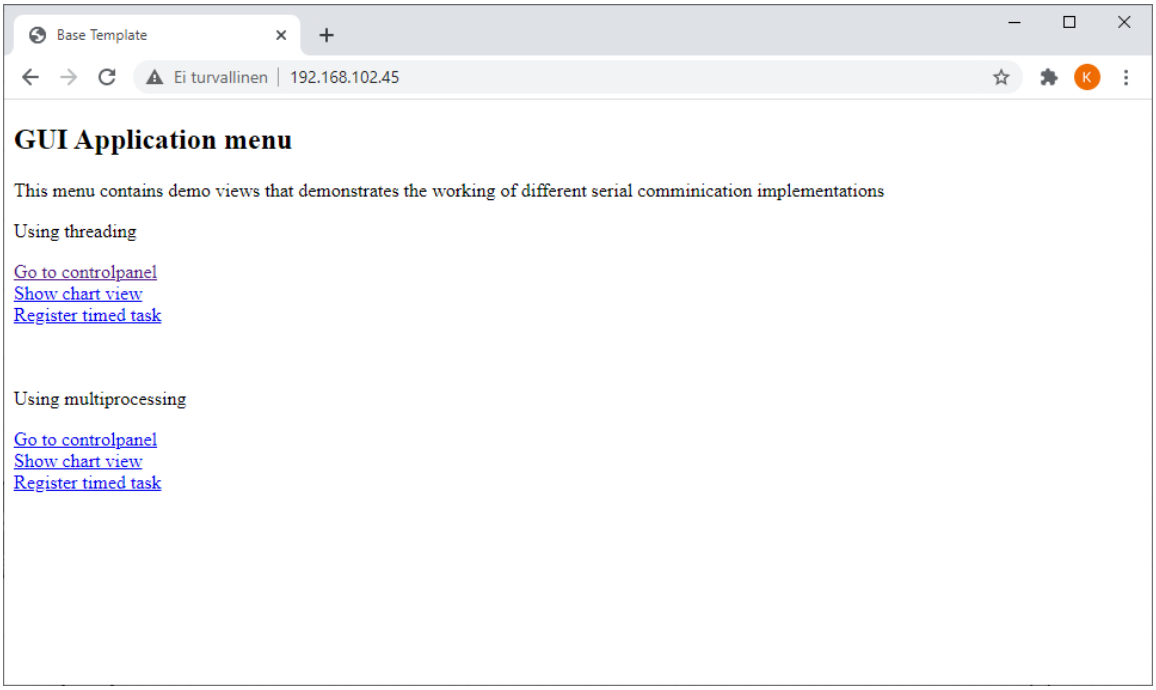
Kuva 21. Eri toteutusversioiden toiminta.

5.4.2 Sovellusnäkymien kehitys

Työn pääasiallinen toiminta perustui lähinnä palvelimen taustaohjelmaan (engl. Back-end) liittyvien ominaisuuksien kehittämiseen. Kuitenkin verkkosovelluksen kehityksessä isona osana on myös käyttöliittymä, jonka avulla koko järjestelmäketjun toimintaa pystyy testaamaan.

Projektin kehittyessä käyttöliittymätoteutukset muokkaantuivat selaimessa toimivasta sarjaliikennemonitorista nykyiseen palvelimen suorituskykyä mittaavaan ja ominaisuuksia esittelevään edustaan (engl. Front-end) prototyyppiin. Näkymissä ei käytetty CSS-koristelua, vaan ne ovat toteutukseltaan puhtaita HTML-dokumentteja, joihin lisätään toiminnallisuutta Djangon ja JavaScriptin avulla.

Koska toteutusvaihtoehtoja oli kaksi, päädyttiin seuraavanlaiseen rakenteeseen, jossa index.html-sivu sisälsi linkit kuuteen eri näkymään. Näkymät on lajiteltu kahteen ryhmään, joista ensimmäisen ryhmän näkymät hyödyntävät suoratoteutusta ja jälkimmäisen ryhmän linkit hyödyntävät prosessitoteutusta (kuva 22).

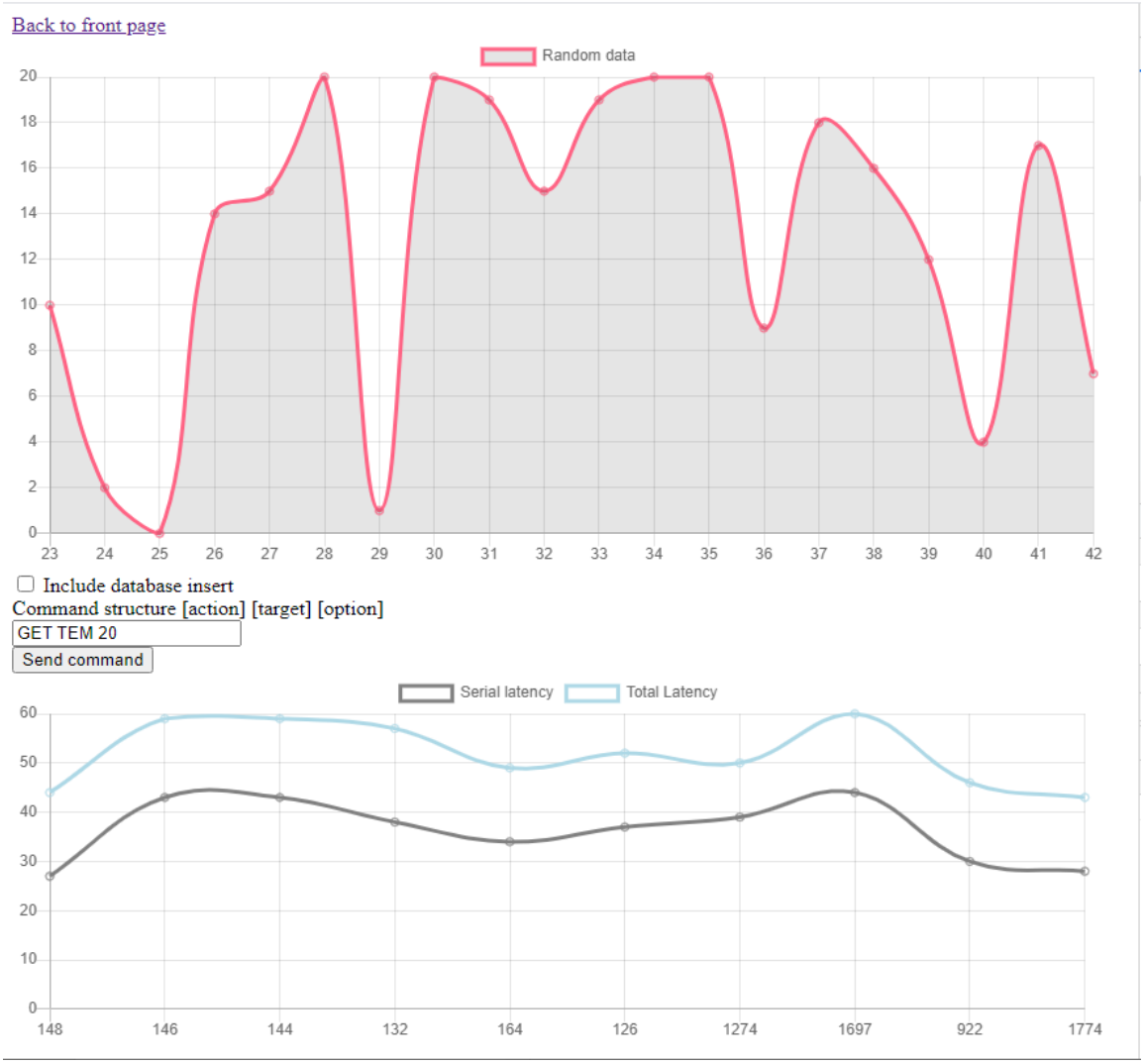


Kuva 22. Kuva index.html-näkymästä.

Kukin kuudesta näkymästä pystyttiin toteuttamaan käyttämällä kolmea uutta HTML-pohjaa. Ensimmäinen näkymistä oli ”Controlpanel”, joka sisälsi suurimman osan mittausjärjestelmän ominaisuuksista. Toinen näkymistä oli ”Chart view”, jolla pystyi esittelemään sarjaliikennelaitteelta haettavaa tietoaineistoa. Kolmantena näkymänä oli ajastetun toiminnon asettamiseen tarkoitettu ”Register timed task”, jonka tarkoituksena oli asettaa palvelimelle komento, jota lähetettäisiin mikrokontrollerille määrätyin väliajoin.

Tiedon välitykseen selainnäkymän ja palvelimen välillä käytettiin AJAX-tyyppistä tiedonsiirtoa. AJAX mahdollistaa verkkosivun tiedon päivittämisen ilman tarvetta koko sivun lataamiseen uudestaan. Tällä tavalla voidaan toteuttaa yhden sivun sovelluksia (engl. Single Page Apps), joissa verkkoselaimelle lähetetään kerralla kaikki tarvittava sivun esittämiseen. Tämän jälkeen sivun ulkonäköä ja sisältöä päivitetään käyttämällä hyödyksi JavaScriptiä ja AJAXia [39] [40]. Kyseinen toteutus astuu hiukan Djangon alueelle, koska samaiset sivulla nähtävät toiminnallisuudet pystyisi toteuttamaan pelkästään Djangon tarjoamia ominaisuuksia hyödyntäen. Tällöin koko sivu pitäisi kuitenkin luoda ja lähettää aina uudestaan, jotta muutokset tulisivat näkyviin. Tällaista uudelleen latailua ei haluttu kehitettävässä sovelluksessa hyödyntää pyrkimällä tällä tavoin tehostamaan koko viestiketjun toimintaa.

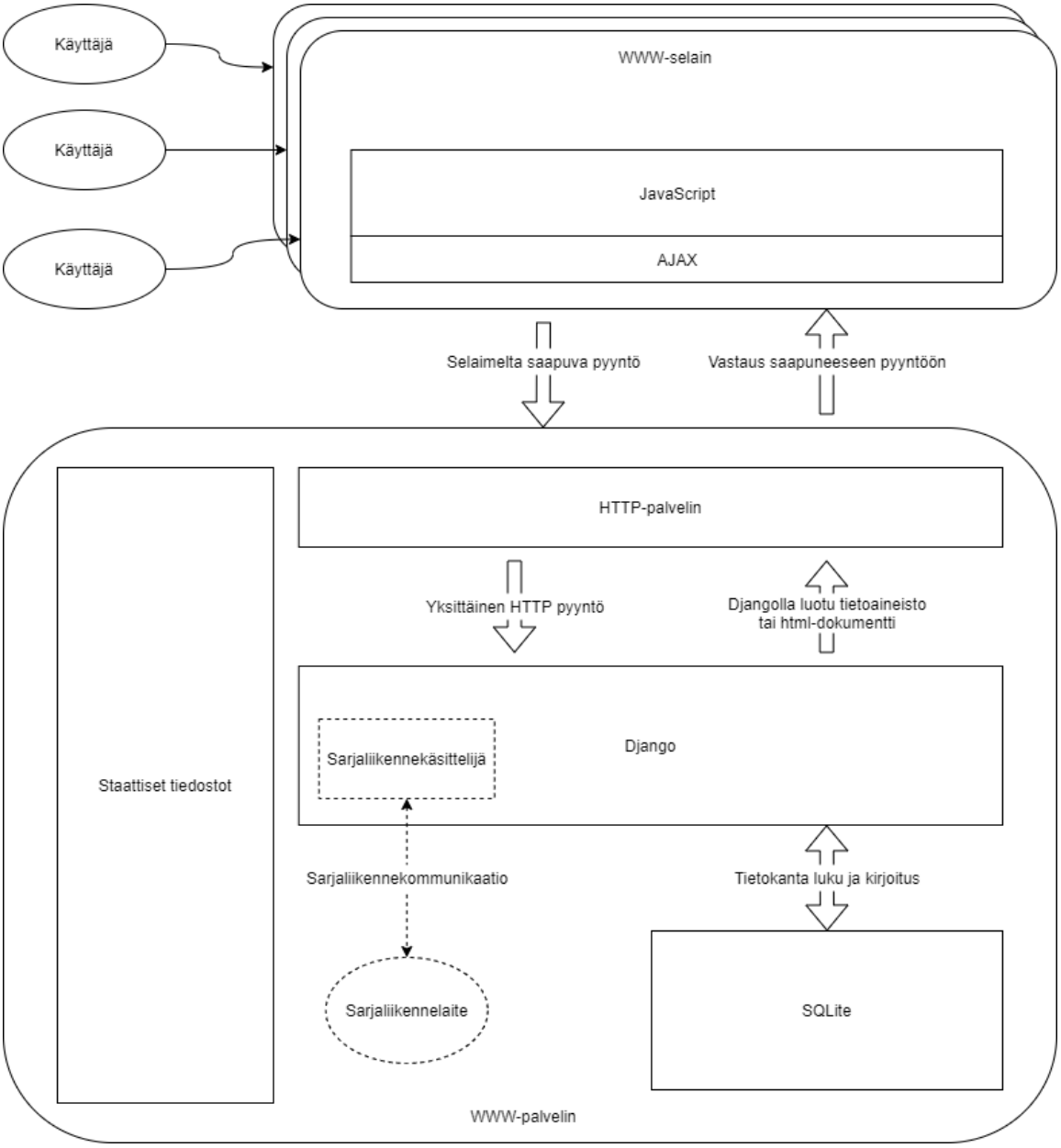
Django kuitenkin tarjoaa mahdollisuuden tällaiselle ”väljälle” kehitykselle, jossa samanlaisia operaatioita voidaan toteuttaa eri tavoilla. Lisäksi AJAX sopii paremmin käytettäväksi tilanteissa, joissa ei välttämättä haluta lähettää koko verkkonäkymän sisältöä uudestaan, vaan viimeisin saatavilla oleva tietoaineisto. Kuvassa 23 on esimerkki, jossa palvelimelta on haettu AJAX-pyyntöä hyödyntäen viimeisimmän 20 näytepisteen tietoaineisto ja esitetty se Charts.js-pohjaisessa kuvaajassa. Lisäksi itse näytepisteiden tietoaineiston mukana on toimitettu tiedonvälitykseen liittyneitä viivetietoja, jotka näkyvät alemmassa kuvaajassa.



Kuva 23. ”Chart View” -näkymästä otettu kuvakaappaus.

Koska useampi samanaikainen sarjaliikenneyhteys voi aiheuttaa käytössä ongelmia, suunniteltiin menetelmä, jolla sarjaliikennekäsittelijä käynnistetään sitä tarvitsevan sivun kohdalla ja sammutetaan kun sivulta poistutaan. Sarjaliikennekäsittelijän käynnistys toteutettiin tarjoamalla URL-osoitteen mukana sarjaliikennekäsittelijän tyypin ilmaiseva leima. Esimerkkinä siirryttäessä ”Controlpanel”-näkymään URL-osoitteella ”/th/controlpanel”, käynnistettäisiin näkymän käyttöön suoratoteutusta hyödyntävä käsittelijä. Jos taas sivulle siirryttäisiin osoitteella ”/mp/controlpanel”, käynnistettäisiin prosessitoteutusta hyödyntävä käsittelijä. Näkymästä poistuttaessa lähetetään HTTP GET -pyyntönä pysäytyskäsky, joka sammuttaa ja sulkee sillä hetkellä aktiivisena olevan käsittelijän. Kyseinen testaamista painottanut ratkaisu ei kuitenkaan ole toimiva usealla eri käyttäjällä ohjattavaksi. Usean aktiivisen käyttäjän aiheuttamaa rasitusta pystyi kuitenkin sovelluksella testaamaan, mutta kyseinen toteutus ei sellaisenaan toimisi lopullisessa sovellutuksessa.

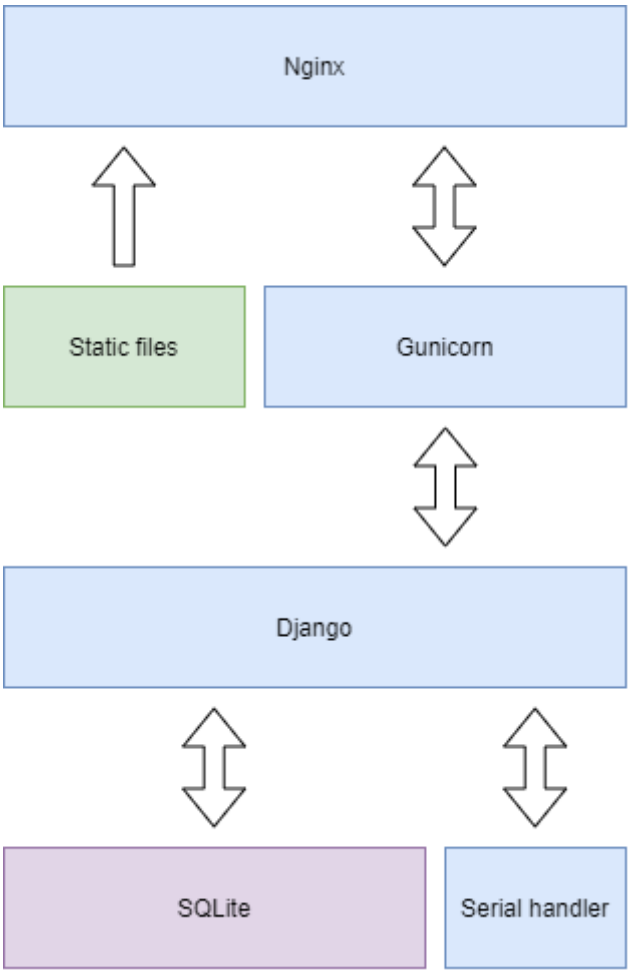
AJAX-peräiset pyynnöt käsitellään Djangon näkymäfunktioissa samalla tavalla kuin normaalit pyynnöt. Sarjaliikennekäsittelijälle ohjattavien pyyntöjen reititykseen käytettiin ”/api/transmit”- ja ”/api/post_transmit”-osoitteita. ”/api/transmit”-osoitteen kautta välitettiin GET-tyyppisiä HTTP-pyyntöjä ja ”/api/post_transmit”-osoitteen kautta POST-tyyppisiä pyyntöjä. GET- ja POST-pyyntöjen käsittelyyn käytetyt funktiot voi myös yhdistää yhdeksi käsittelijäksi, mutta yksinkertaistamisen takia päätettiin ne erotella omiksi funktioikseen. Loppuversiossa GET-tyyppisiä pyyntöjä hyödynnettiin vain sammuttamaan kullakin hetkellä aktiivisena oleva sarjaliikennekäsittelijä. Koko ohjelman toimintaketju on pyritty esittämään kuvassa 24.



Kuva 24. Kehitetyn sovelluksen toimintarakenne.

5.5 NGINX:n ja Gunicornin automaattinen käynnistys systemd:n avulla

Django ei ole alkujaankaan suunniteltu toimimaan verkkopalvelimena, vaikka kehityksessä tällaisen käsityksen voi saada. Django tarvitsee tehokkaasti toimiakseen oikeaa verkkopalvelinohjelmistoa, kuten NGINX tai Apache. Nämä eivät kuitenkaan suoraan kykene tarjoamaan Djangolla luotuja verkkosivuja, vaan väliin tarvitaan WSGI-liityntää tukeva HTTP-palvelin, jolla Django yhdistetään verkkopalvelinohjelmistoon. Tällaisia HTTP-palvelimia ovat Gunicorn, Tornado, mod_WSGI ja uWSGI. Tällaisessa kokoonpanossa Apache tai NGINX toimii niin sanottuna välityspalvelimena (engl. Proxy), joka kyselee tähän yhdistetyn WSGI HTTP -palvelinliittymän kautta Djangolla luotuja dynaamisia verkkosivuja tai tarjoilee suoraan tunnettuun sijaintiin tallennettuja staattisia tiedostoja kuten CSS-, JavaScript- tai kuvatiedostoja (kuva 25).



Kuva 25. NGINX-, Gunicorn-, Django- ja SQLite-ohjemapino.

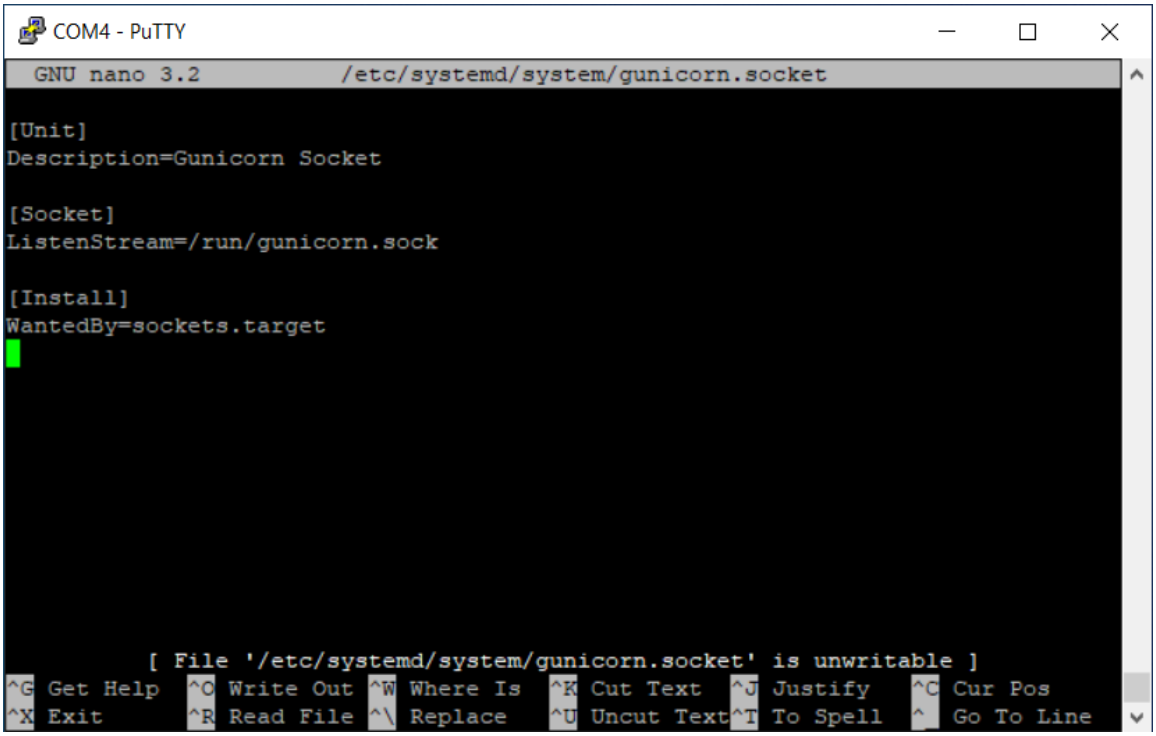
Jotta ohjelmistopino olisi käytännöllinen, täytyy NGINX ja Gunicorn käynnistää automaattisesti isäntäjärjestelmän eli Debianin mukana. Onneksi Debian tarjoaa tällaiseen tehtävään monipuolisen systemd-hallintaohjelmiston. Luomalla systemd-palvelu (engl. Service) voidaan systemd asettaa käynnistämään joku haluttu ohjelma tai skripti tietyssä vaiheessa käynnistysprosessia. Tämän lisäksi systemd:n tehtävänä on hallita käynnistämiään palveluita ja prosesseja. Tästä syystä onkin siis käytännöllistä hyödyntää systemd-ohjelmaa NGINX- ja Gunicorn-palvelimen käynnistämiseen.

5.5.1 Gunicorn-pistokkeen konfigurointi

NGINX- ja Gunicorn-sovellukset asennettiin aluksi osaksi järjestelmää. Tässä NGINX:n asentamiseen käytettiin apt-get-asennustyökalua ja Gunicornin asentamiseen Pipenv-virtuaaliympäristön install-komentoa. Tämän jälkeen luotiin Gunicornin käynnistävän palvelun konfiguroimiseksi tarvittavat gunicorn.service- ja gunicorn.socket-yksikkötiedostot. [41.]

Ensimmäisellä yksikkötiedostolla määritellään pistoke (engl. Socket), jonka kautta NGINX pystyy välittämään HTTP-pyyntöjä Gunicornille käsiteltäväksi. Saman pistokkeen kautta Gunicorn pystyy tarjoamaan Djangolla luodun HTML-dokumentin takaisin NGINX:lle, joka puolestaan välittää nämä eteenpäin niitä pyytäneelle käyttäjälle. Yksikkötiedosto luotiin sijaintiin ”/etc/systemd/system/” käyttämällä hyödyksi Debian-järjestelmään asennettua nano-tekstieditoria. Yksikkötiedostoon määriteltiin seuraavat valitsimet (engl. Options): [Unit], [Socket] ja [Install]. Kyseisistä valitsimista [Unit] ja [Install] ovat välttämättömiä kaikissa yksikkötiedostoissa. [Unit]-valitsinta käytetään määrittelemään luotavan yksikön perustietoja. Tässä tapauksessa ”Description=”-määrittelijällä asetetaan luotavan pistokeyksikön nimeksi ”Gunicorn Socket”. [Install]-valitsimella voidaan määritellä luotavan yksikön käynnistykseen liittyviä tietoja. Tässä tapauksessa ”WantedBy=”-määrittelijää käytetään käynnistämään luotava pistokeyksikkö samaan aikaan ”sockets.targets”-yksikön kanssa. [41.] [42.] [43.]

Lisäksi, kun luotavana on pistoke, täytyy myös lisätä valitsin [Socket], jolla määritellään minne ja minkä nimisenä pistoke luodaan. Tässä tapauksessa ”gunicorn.sock”-niminen pistoke luodaan ”/run/”-sijainnin alle. Kuvassa 26 on esitettynä kuvakaappaus pistokkeen yksikkötiedostosta. [44.]



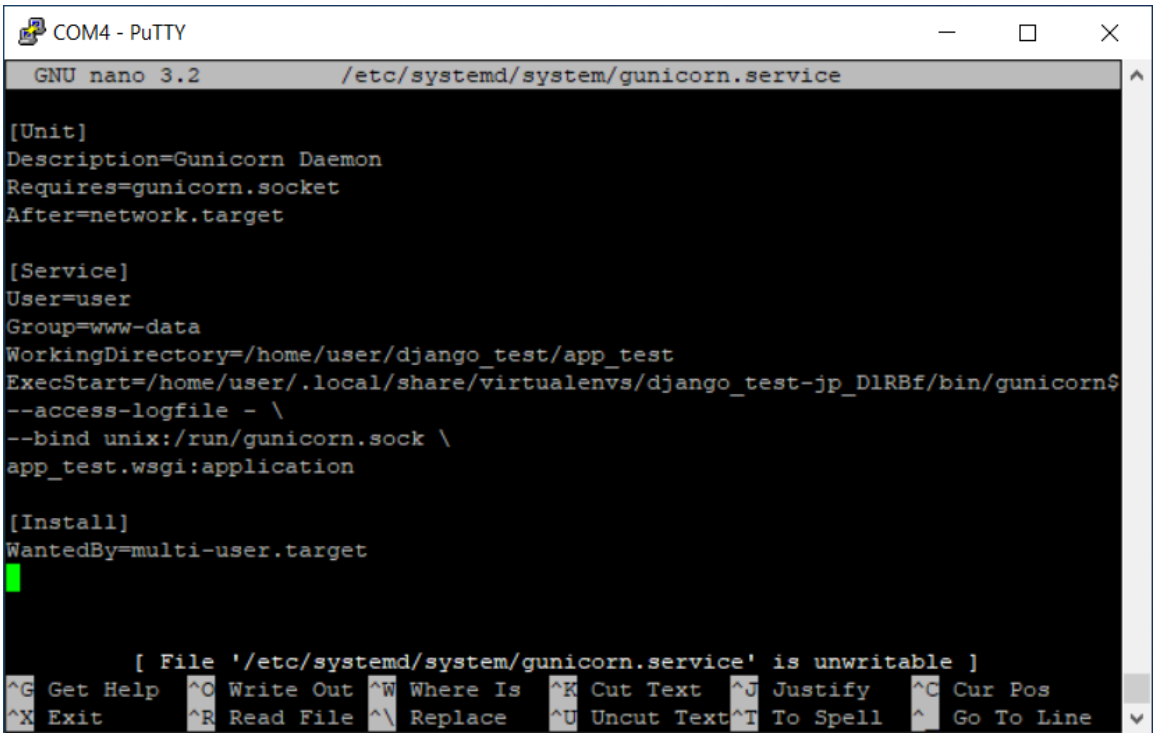
Kuva 26. Gunicornin systemd-pistokkeen määrittely.

5.5.2 Gunicorn-palvelun konfiguroiminen

”service”-tiedostopääteen omaavalla yksikkötiedostolla määritellään systemd-palvelu, jonka tehtävänä on käynnistää ja ylläpitää Gunicorn HTTP-palvelinta [41]. Gunicornin saadessa pistokkeen kautta HTTP-pyyntö, välitetään tämä Gunicornin työläisprosessille (engl. worker), joka ylläpitää instanssia Djangosta. Tämä instanssi luo URL-osoitetta vastaavan HTML-dokumentin, joka puolestaan välitetään samaa reittiä takaisin sitä pyytäneelle käyttäjälle.

Koska määriteltävä yksikkö on tyypiltään palvelu, täytyy yksikkömäärittelyyn lisätä [Service]-valitsin, jolla määritellään palvelulle ominaisia ominaisuuksia. Näistä tärkeimpiä ovat ”ExecStart=”-määrittelijä, jolla osoitetaan palvelulle käynnistyksessä ajettavan sovelluksen binääri tai skripti. Lisäksi samaan määrittelijään tulee sisällyttää sovelluksen tarvitsemat vivut (engl. Command Options). Tässä tapauksessa palvelulle osoitetaan Gunicornin käynnistämiseen tarvittava skripti, joka sijaitsee Pipenv-ympäristössä. Lisäksi määritellään sijainti, jossa kyseinen skripti ajetaan käyttämällä ”WorkingDirectory=”-määrittelijää. [41] [45.]

Näiden lisäksi käytettiin vielä ”User=”- ja ”Group=”-määrittelijöitä, joilla voidaan asettaa käynnistykseen liittyviä oikeuksia. ”User=”-määrittelijällä voidaan asettaa sovelluksen suorittajaksi tietty käyttäjä, tässä tapauksessa ”user”. ”Group=”-määrittelijällä taas asetetaan käynnistettävä palvelu kuuluvaksi tiettyyn Linux-ryhmään. Koska NGINX:n pitää pystyä kommunikoimaan luotavan Gunicorn-palvelun kanssa, täytyy käytettäväksi ryhmäksi asettaa ”www-data”. [45.] Kuvassa 27 palvelun määrittelyssä käytetty yksikkötiedosto.



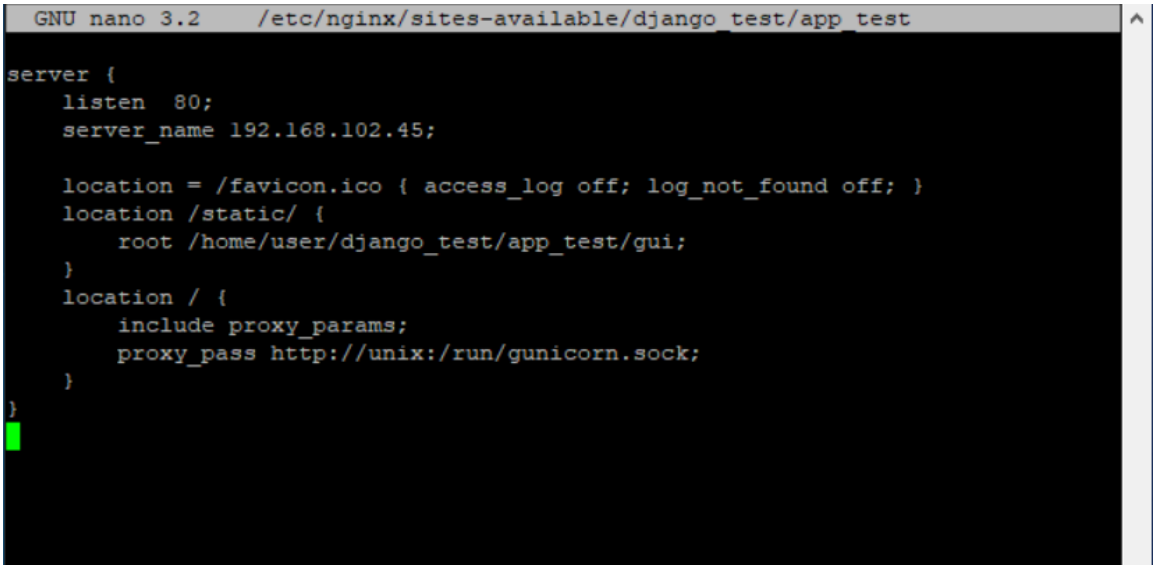
Kuva 27. Gunicornin systemd-palvelun määrittely.

5.5.3 NGINX:n konfigurointi ja käyttöönottaminen

NGINX omaa aivan omat konfigurointitiedostonsa. Kyseisillä tiedostoilla määritellään virtuaalisia palvelimia, joita NGINX:n tulee ylläpitää. Tällaiset virtuaalipalvelimen määrittelevät tiedostot luodaan sijaintiin ”/etc/nginx/sites-available/”. Myöhemmin, luomalla konfiguraatiotiedostosta symbolinen linkki sijaintiin ”/etc/nginx/sites-available/”, voidaan asettaa kyseinen virtuaalipalvelin aktiiviseksi. [41.]

”/etc/nginx/sites-available/”-sijaintiin luotiin uusi konfiguraatiotiedosto ”app_test”, joka nimettiin kehitetyn Django-projektin mukaan tunnistamisen helpottamiseksi. Tähän tiedostoon määriteltiin palvelimen käyttämä IP-osoite ja portti server_name- ja listen-direktiiveillä (engl. Directive). Tämän lisäksi määriteltiin eri HTTP-pyyntötilanteita vastaavat toiminnot location-direktiiveillä. Ensimmäisellä location-direktiivillä määriteltäisiin selaimen välilehdessä esitettävä ikoni ja sijainti, josta se löytyisi. Tässä tapauksessa kyseistä ikonia ei ole saatavilla, joten määrityksillä asetetaan NGINX hylkäämään kaikki ikonin lataamiseen viittaavat virheet. [41.]

Tärkeimpiä ovat jälkimmäiset ”location”-direktiivit, joista ensimmäisellä määritellään sijainti, josta NGINX voi tarjota staattisia tiedostoja ”/static/”-alkuisiin pyyntöihin. Jälkimmäisellä direktiivillä asetetaan NGINX välittämään kaikki muut määrittelemättömät pyynnöt luodun pistokkeen kautta Gunicornille. Kahdessa viimeisessä direktiivissä on tärkeää huomata, että yhtäsuuruusmerkki on jätetty tarkoituksella pois. Tällöin kaikki määritellyn osan sisältävät osoitteet, kuten ”/static/js/”, osataan reitittää oikein. Kuvassa 28 on NGINX:n virtuaalipalvelimen määrittelyyn käytetty konfigurointitiedosto. [41.] [46.] [47.]



Kuva 28. NGINX konfigurointitiedosto.

5.6 Suorituskyvyn testaus

Lopun testauksissa selvisi, että prosessitoteutuksen etuna oli vain sen kyky tarjota mahdollisuus tehokkaan kiertokyselyn toteuttamiseksi. Muuten se oli paljon suoratoteutusta monimutkaisempi ja tarjosi suoratoteutukseen verrattuna samanlaista suorituskykyä, jos prosessin pystyi käynnistämään etukäteen.

Testaamisen alkuperäisenä tarkoituksena oli selvittää eri toteutusversioiden välistä suorituskykyä toisiinsa verrattuna ja tämän kautta valita suorituskyvyltään tehokkain tapa toteuttaa Djangoa hyödyntävä verkko- ja sarjaliikennepalvelin. Testauksissa kuitenkin havaittiin, pienen sattuman kautta, että valitsemalla oikeantyyppisen laitemuistin ja pitämällä tietokantakirjoitukset mahdollisimman vähäisinä, pystyi suorituskykyä parantamaan huomattavasti enemmän, kuin pelkän sarjaliikennekommunikaation toteutuksella.

5.6.1 Tiedonsiirtoviiveen mittaus

Viiveellä tietotekniikassa tarkoitetaan tiedon lähettämisestä sen vastaanottamiseen ja prosessointiin kulunutta aikaa. Viive aiheutuu tiedolle suoritettavasta prosessoinnista ja siirtoteille ominaisista tiedonsiirtonopeuksista. Mitä useamman laitteen ja pidemmän matkan läpi tietoa pitää kuljettaa, sitä kauemmin tiedon kulkemisella tulee kestämään. [48.] [49.]

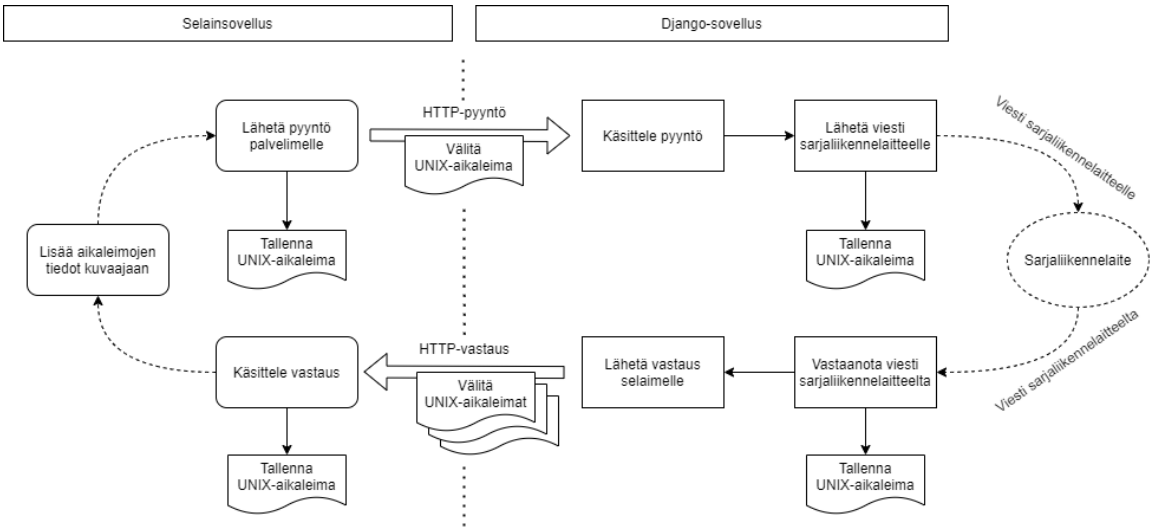
Käyttöjärjestelmällisissä toteutuksissa tiedonsiirtoon kuluvan ajan arvioiminen on hankalaa, koska tiedonsiirtoa ei ole taattu tapahtuvaksi välittömästi. Käyttöjärjestelmän sisältämä vaiheittainen ohjaus, jonka tehtävänä on päättää missä järjestyksessä ja milloin eri toimintoja tai prosesseja ajetaan, aiheuttaa epävarmuutta viiveen selvittämiseen.

Testauksen kannalta haluttiin selvittää, kuinka kauan komentojen välittämisessä sarjaliikennelaitteelle kestäisi ja kuinka suuria viiveitä viestien edestakaisessa kulussa pahimmillaan esiintyy. Verkkojärjestelmissä viestien edestakaiseen välitykseen selaimelta palvelimelle kuluneesta ajasta käytetään toisinaan termiä latenssi. Voidaan siis sanoa, että oltiin kiinnostuneita järjestelmässä esiintyvästä latenssista.

5.6.2 Testausjärjestelyt

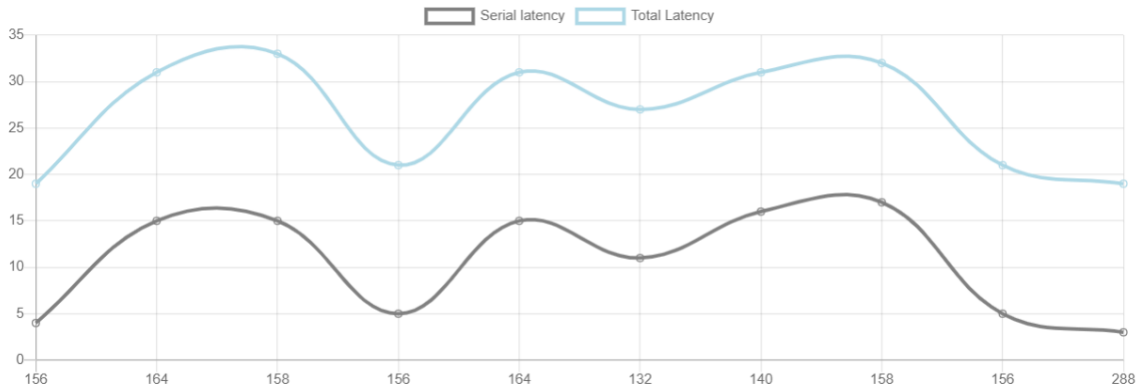
Testaamisessa hyödynnettiin itse toteutettua jäljittelijäskriptiä (engl. Mockup), jonka tarkoituksena oli simuloida sarjaliikennepalvelimeen yhdistetyn orjalaitteen toimintaa. Orjalaitteen jäljittelijä oli yksinkertainen Python-skripti, joka sarjaliikenneviestin saatuaan lähetti kuittauksen viestin lähettäjälle. Jäljittelijä sisälsi myös monimutkaisempia toimintoja, kuten 20-alkion satunnaismuotoisen tietoaineiston lähettämisen, jos tällaista pyydettiin selaimelta. Isojen tietomäärien siirrolla voitiin havainnoida välitettävän viestin suuruuden vaikutusta siirtoaikoihin.

Viiveaikojen mittauksessa käytettiin hyvin yksinkertaista ja hieman epätarkkaa menetelmää tiedonsiirrossa esiintyneiden viiveiden mittaamiseen. Tietyissä ohjelman suorituspisteissä tallennettiin UNIX-aikaleimoja, jotka välitettiin selaimelle ja lopuksi vähennettiin toisistaan. Näin voitiin selvittää mittauspisteiden välillä ilmenneitä viiveitä ja kokonaisviivettä. Toteutuksen toimintaa on pyritty esittämään kuvassa 29.



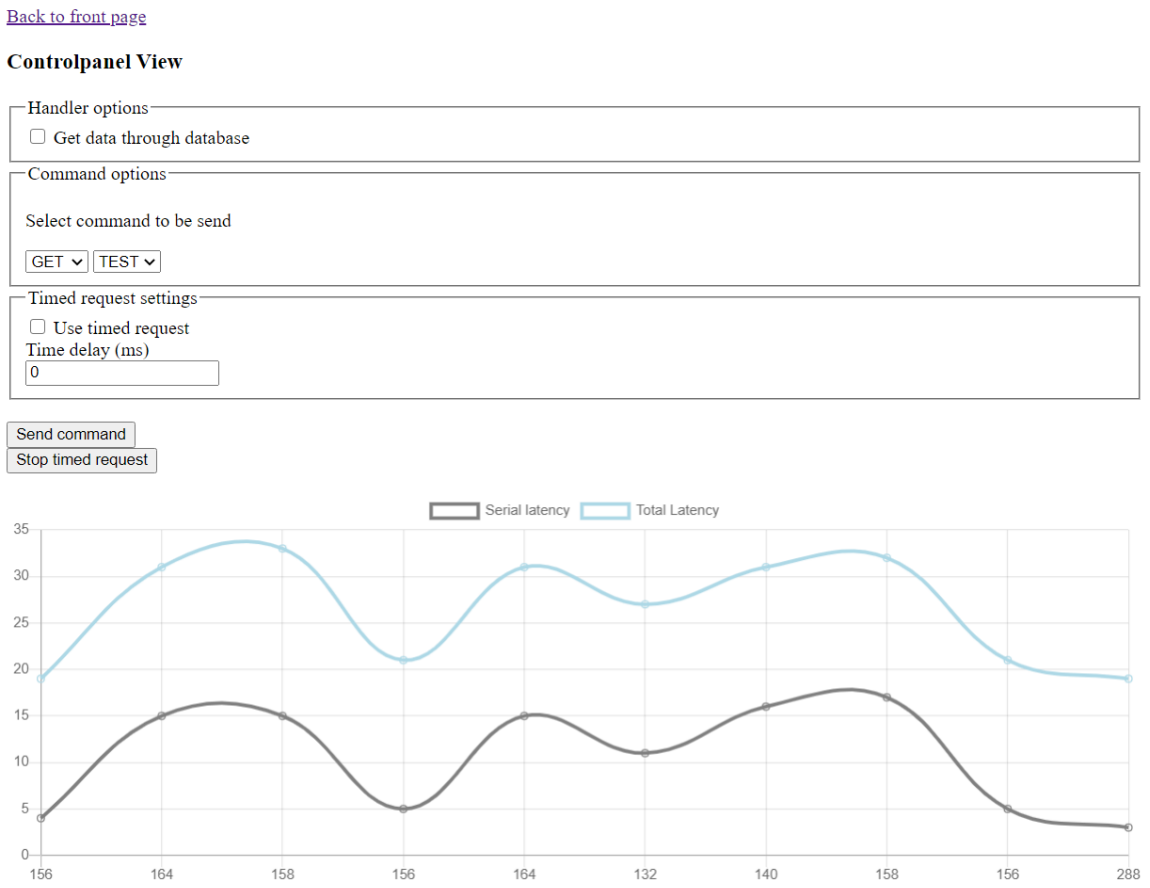
Kuva 29. Viiveaikojen mittaaminen sovelluksessa.

Viivearvojen esittämiseen oli kehitetty Charts.js-pohjainen kuvaaja osaksi käyttöliittymää (kuva 30). Kuvaajassa sinisen viivan pisteet esittävät yhden HTTP-kyselyn aikana esiintynyttä kokonaisviivettä, eli kuinka kauan aikaa kului kyselyn lähettämisestä vastauksen vastaanottamiseen. Alapuolella olevan harmaan viivan pisteet esittävät kuinka kauan HTTP-kyselyn aikana sarjaliikenneviestin lähettämisen ja vastauksen saamisen välillä kului aikaa. X-akselilla taas näkyy, kuinka kauan aikaa kahden pyynnön välillä on kulunut.



Kuva 30. Kokonais- ja sarjaliikenneviiveen kuvaajat.

Testauksessa käytettiin kolmea aikaisemmin kuvattua näkymää ”Controlpanel”, ”Chart View” ja ”Register Timed Task”, sekä näiden eri toteutustyyppejä suoratoteutus ja prosessitoteutus. Näkymistä tärkein oli ”Controlpanel”-näkymä (kuva 31), jonka kautta pystyi muokkaamaan lähetettävien viestien koostumusta ja tätä kautta pyytämään orjalaiteen jäljittelijältä erityyppisiä tietoja. Lisäksi näkymästä pystyi käynnistämään ja sammuttamaan Djangossa sekä selaimessa suoritettavia toimintoja, kuten tietokantaan kirjoittamisen.

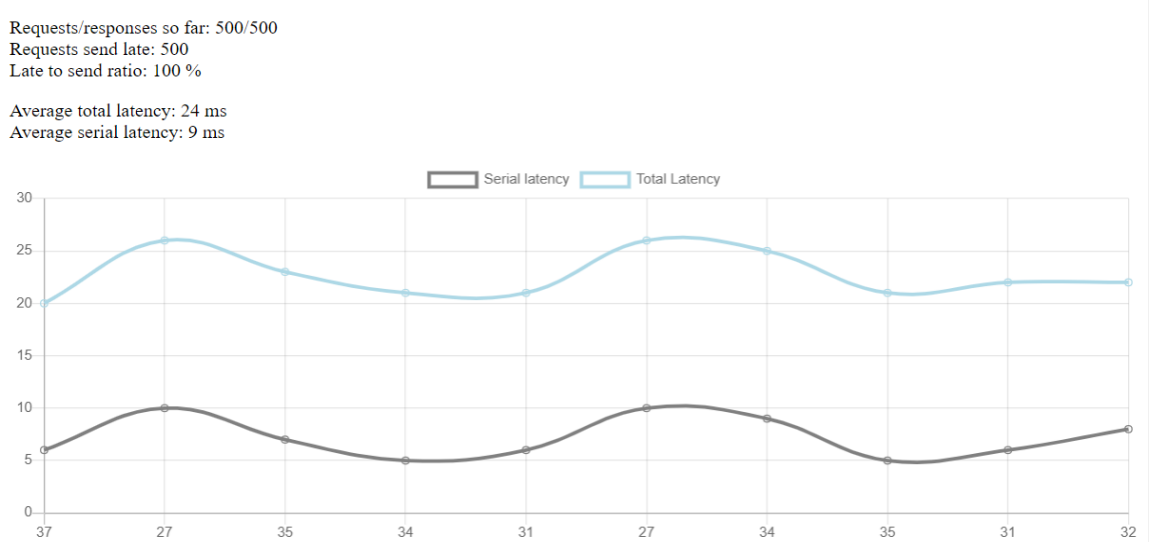


Kuva 31. ”Controlpanel”-näkymä.

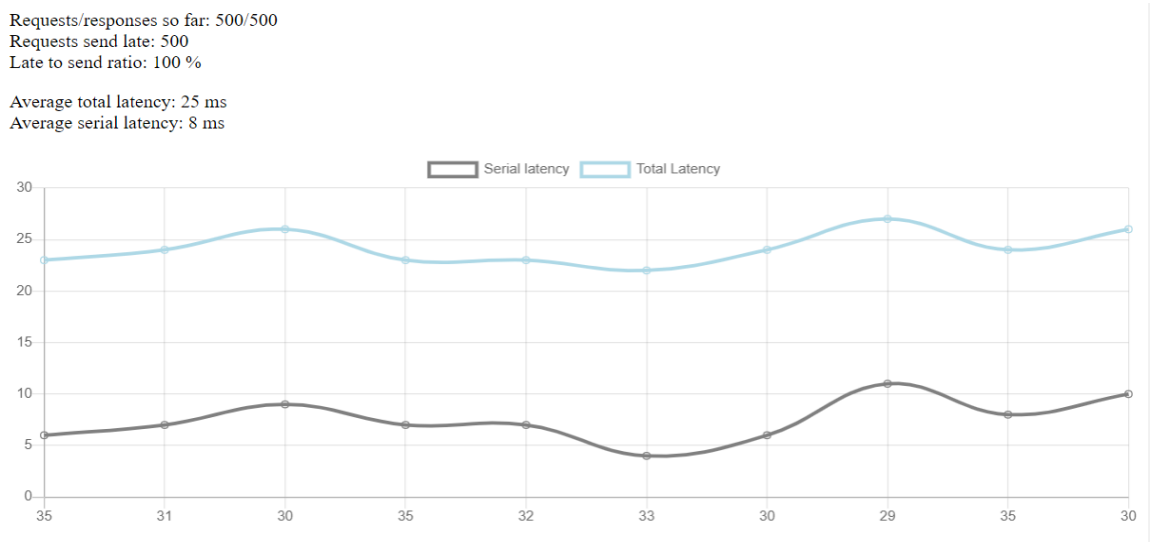
Testaukset suoritettiin ajamalla automatisoituja kyselyitä palvelimelle, jossa automatisoitu kysely lähetti heti seuraavan kyselyn saatuaan vastauksen palvelimelta. Vaihtoehtoisesti automatisoidut kyselyt pystyttiin asettamaan lähetettäväksi tietyin väliajoin, esimerkiksi 100 ms välein. Tällaisia tiedonvaihtoja suoritettiin 500 kappaletta per mittaus. Mittaustilanteita muutettiin muokkaamalla näkymän tarjoamia asetuksia.

5.6.3 Tulokset toteutusversioiden suorituskyvystä

Lopun sovellusversioiden välillä ei havaittu suorituskyvyllisesti suurta merkitystä, tehtiinkö testit suoratoteutuksella vai prosessitoteutuksella. Tämä selviää hyvin seuraavista mittaustuloksista (kuva 32) ja (kuva 33). Kuten kuvaajista ja tarjotusta lisätiedosta voi nähdä, molempien toteutuksien tapauksessa yksinkertaisen viestin välittämiseen kului aikaa noin 20–25 ms. Kuvaajista voi havaita, että kokonaisviiveen ilmoittava sininen viiva seuraa hyvin läheisesti sarjaliikenteen viiveen harmaata viivaa. Tästä voidaan päätellä, että selaimen ja Djangon välissä tapahtuvissa toiminnoissa kuluva aika on yleensä vakio. Välillä tässä kuitenkin ilmeni poikkeavuutta.

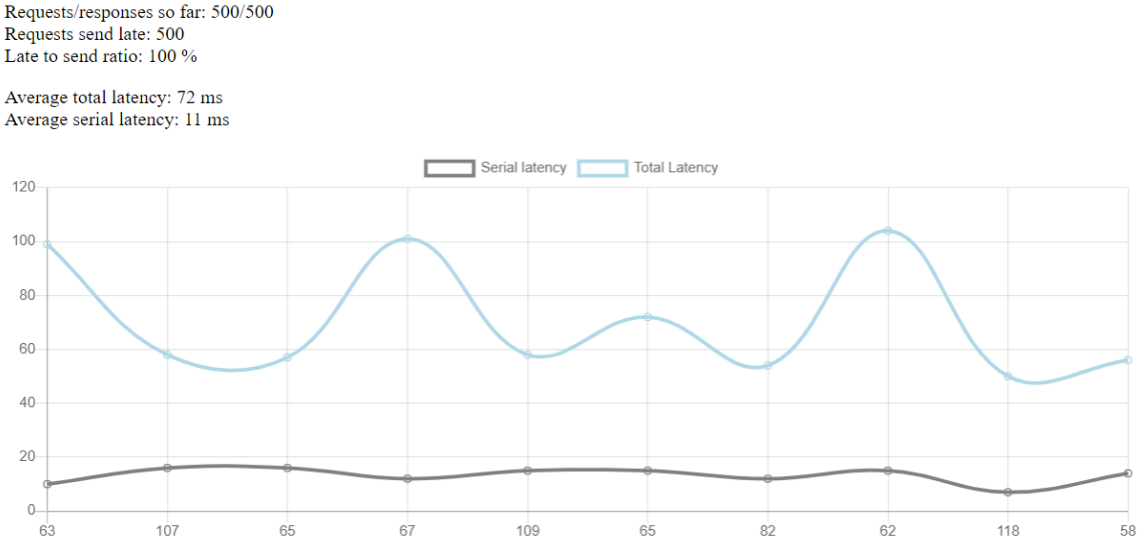


Kuva 32. Suoratoteutus ilman tietokantaan kirjoittamista.

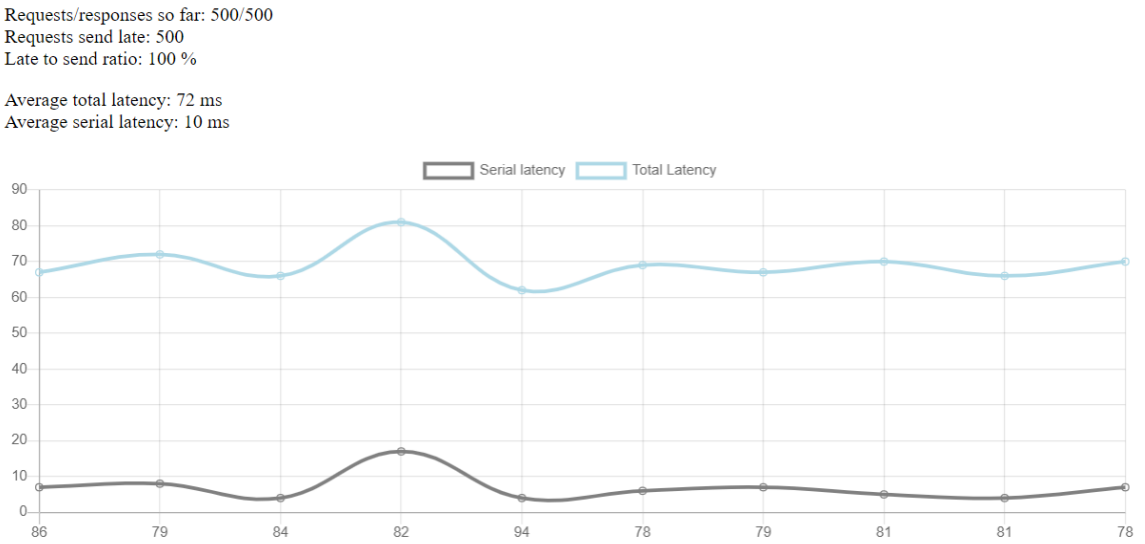


Kuva 33. Prosessitoteutus ilman tietokantaan kirjoittamista.

Lisäämällä tiedonvälitykseen mukaan tietokantaan kirjoittamisen, saatiin seuraavien kuvien 34 ja 35 esittämiä tuloksia. Välitettävän viestin rakenteen ollessa sama tietokantaan kirjoituksen myötä kokonaisviive kasvoi noin 72 ms. Lisäksi molemmissa tapauksissa havaittiin kuvassa 34 hyvin ilmenevää aaltoilua. Tuloksista voidaan päätellä, että tietokantaan kirjoittamisella on suuri vaikutus koko toimintaketjun suorituskykyyn.



Kuva 34. Suoratoteutus tietokantaan kirjoittamisella.

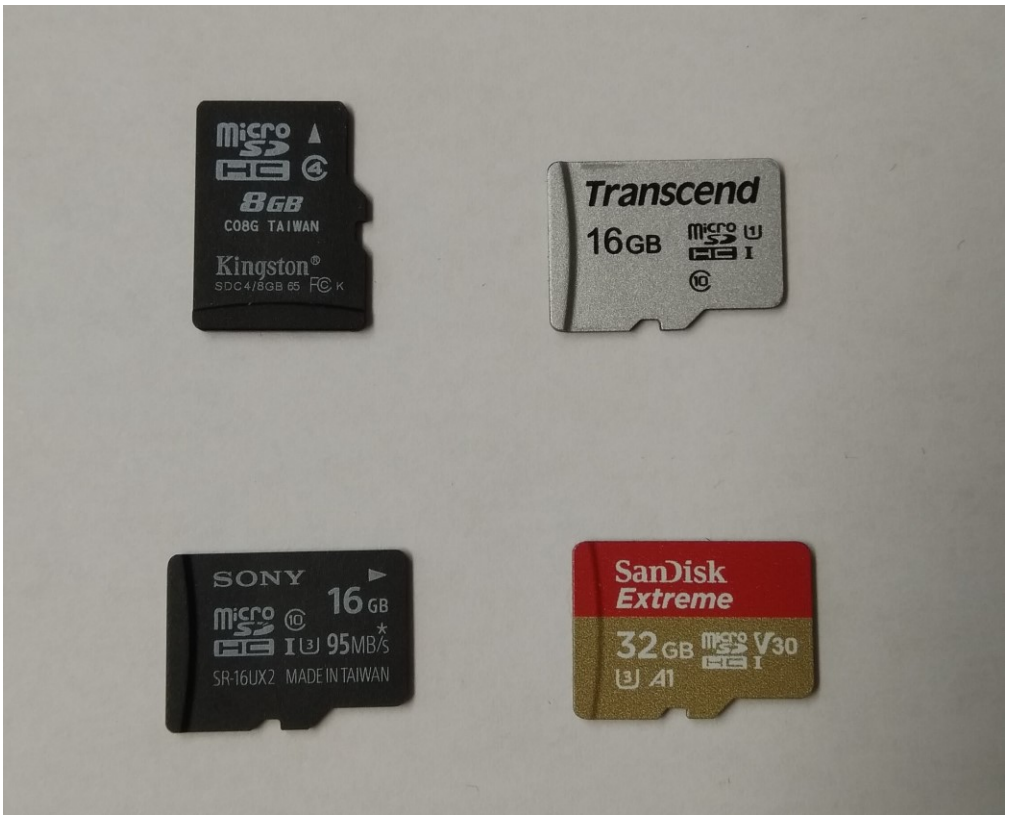


Kuva 35. Prosessitoteutus tietokantaan kirjoittamisella.

5.6.4 Käytettävän muistilaitteen merkitys suorituskykyyn

Merkittävimmät tulokset eivät liittyneet itsessään toteutettuun sovellukseen, vaan järjestelmään, jossa sitä ajettiin. Sattumalta kehityksessä oli pyöritelty kahdentyyppisiä muistikortteja, joista toiseen oli tallennettu puhdas Debian-asennus ja toiseen Yocton ja Debianin yhdistelmäasennus. Vaihdellessa eri järjestelmien välillä oli puhtaassa Debian-versiossa havaittavissa selvästi hitaammat käynnistys- ja sammutusajat kuin toisessa. Lisäksi SQLite-tietokannan toiminnassa havaittiin merkittäviä eroja riippuen siitä, kummalla muistikortilla tätä käytti. Pahimmissa tapauksissa HTTP-pyynnön käsittely saattoi olla tietokantakirjoituksessa jumittuneena useita sekunteja.

Asentamalla muistikorteille kopiot samasta Debian-versiosta ero käynnistys- ja sammutusajoissa säilyi edelleen näiden kahden muistikortin välillä. Syy ei siis ollut käytetyssä käyttöjärjestelmässä, vaan ilmeisimmin käytetyssä muistikortissa. Tämän testaamiseksi hankittiin vielä kaksi muistikorttia lisää, jotta nähtäisiin kuinka paljon muistikortin ominaisuudet voivat vaikuttaa suorituskykyyn. Testeissä käytetyt muistikortit on esitelty kuvassa 36.



Kuva 36. Testauksessa käytetyt muistikortit.

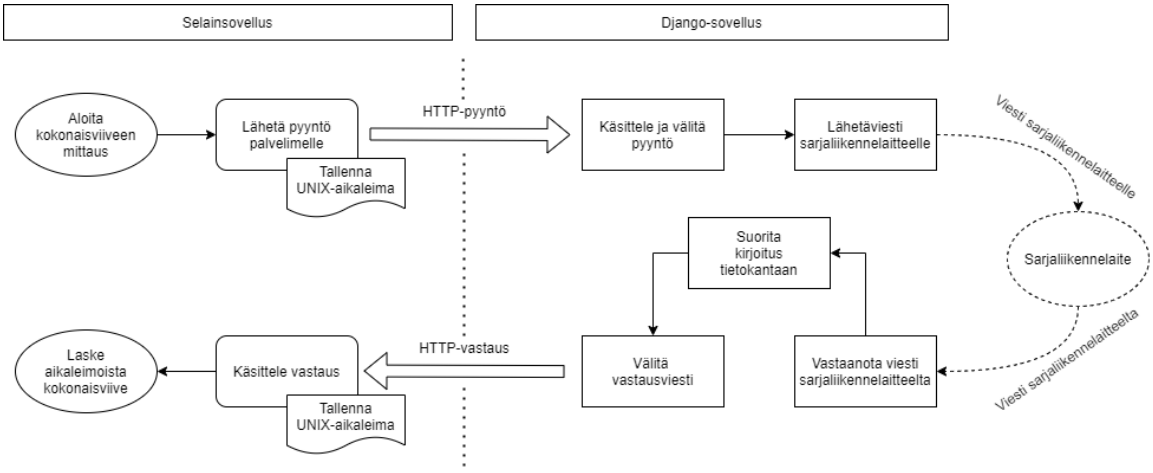
Muistikortit omasivat taulukossa 1 esitetyt ominaisuudet. Listatuista ominaisuuksista kiinnostavimpia olivat Speed Class, UHS Class ja Video Speed Class -luokat, jotka ilmoittavat minimisiirtonopeuden, johon muistikortin pitää pystyä [50]. Näiden ominaisuuksien lisäksi tärkeäksi nähtiin Bus Interface, joka määrää teoreettisen maksiminopeuden, jolla muistikortti voi tietoa siirtää [51]. Muistikortin teoreettinen tiedonsiirtonopeusalue on eritelty taulukon viimeiseen ”Nopeus”-nimiseen kenttään.

Taulukko 1. Testauksessa käytettyjen muistikorttien ominaisuudet.

Muistikortti	Kingston	Transcend	Sony	SanDisk
Muistinmäärä	8GB	16GB	16GB	32GB
Bus Interface	-	UHS-I	UHS-I	UHS-I
Speed Class	4	10	10	-
UHS Class	-	1	3	3
Video Speed Class	-	-	-	30
App Performance Class	-	-	-	1
Nopeus (MB/s)	4–25	10–104	10–104	30–104

Taulukossa esiteltiin vielä luokka App Performance Class, joka on tarkoitettu osoittamaan muistikortin soveltuvuutta sovellusten tallennusmuistiksi. Ilman tätä luokkaleimaa olevat SD-kortit on suunniteltu käytettäväksi normaalimpien tiedostojen, kuten kuvien, videoiden ja musiikin tallentamiseen. Tällainen muistikortti ei välttämättä takaa parhainta suorituskykyä sovelluksille. App Performance Class 1 -leimalla merkityn muistikortin tulee kyetä minimissään lukuoperaatioihin 1500 IOPS:n verran ja satunnaisiin kirjoitusoperaatioihin 500 IOPS:n verran [52]. Kyseisestä luokasta on myös taso 2, jonka tulee kyetä keskiarvoltaan 4000 IOPS:n lukuun ja 2000 IOPS:n kirjoitukseen [52].

Mittauksissa käytettiin kehitettävän Django-sovelluksen keskeneräistä versiota, jossa kirjoitettiin satunnaista tietoa SQLite-tietokantaan ennen vastauksen lähettämistä pyyntöön. Verkkopalvelimena testauksissa käytettiin Djangon tarjoamaa kehityspalvelinta. Mittausketjun toimintaa on pyritty selventämään kuvassa 37. Mittaustuloksissa esitettävä kokonaisviive on saatu tallennettujen aikaleimojen erotuksesta.



Kuva 37. Havainnollistava kuva mittauksen toimintaketjusta.

Sama mittaus suoritettiin kymmeneen otteeseen kaikilla muistikorteilla ja samalla kopiolla käyttöjärjestelmästä. Mittauksissa saatiin muistikorteille taulukon 2 mukaiset tulokset. Taulukosta nähdään selvästi, miten vanhimman tekniikan omaavalla muistikortilla (Kingston) tietokantaan kirjoittaminen aiheuttaa useita sekunteja kestävän kokonaisviiveen tiedonsiirtoon, kun taas uudemmilla muistikorteilla kokonaisviive on vain muutamia satoja millisekunteja. Uusimman tekniikan omaavien korttien välillä on myös nähtävissä suorituskykyeroa.

Taulukko 2. Eri muistikorttien vaikutus kokonaisviiveeseen.

Muistikortti	**Kingston**	**Transcend**	**Sony**	**SanDisk**
Aika 1 (m/s)	2658	190	155	152
Aika 2 (m/s)	2810	146	175	129
Aika 3 (m/s)	3515	137	180	127
Aika 4 (m/s)	3701	196	152	135
Aika 5 (m/s)	6347	134	151	133
Aika 6 (m/s)	2808	479	154	136
Aika 7 (m/s)	2813	141	178	126
Aika 8 (m/s)	2810	169	179	133
Aika 9 (m/s)	2803	184	146	124
Aika 10 (m/s)	2819	192	169	130
Keskiarvo (m/s)	3308.4	196.8	163.9	132.5

5.6.5 Käytetyn tietokannan vaikutus suorituskykyyn

Viimeiseksi päätettiin testata, voiko käytettävällä tietokantaversiolla parantaa suorituskykyä heikoimmin suoriutuneella muistikortilla. Django tukee suoraan SQLite, MySQL/MariaDB ja PostgresSQL -tietokantoja. Niinpä SQLiten lisäksi huonoiten suoriutuneelle muistikortilla asennettiin MySQL/MariaDB- ja PostgreSQL-tietokannat. Asennus suoritettiin apt-get-työkalun avulla.

Jokaisen testauksen välillä Django jouduttiin konfiguroimaan uudestaan, jotta testauksessa hyödynnettäisiin seuraavaa tietokantaversiota. Konfigurointi tapahtui muuttamalla Django-projektin settings.py-tiedoston sisältämiä asetuksia. Muuten testaukset suoritettiin samalla järjestelmällä kuin aikaisemmat muistikorttien suorituskyvyn mittaukset, jonka toiminta oli esitetty aikaisemmassa kuvassa 37.

Mittauksissa saatiin taulukon 3 mukaiset tulokset. Testeissä PostgreSQL suoriutui keskiarvoltaan parhaiten, kun taas MySQL/MariaDB tarjosi parhaimmat yksittäiset kirjoitusnopeudet. MySQL-tietokannan tapauksessa ilmeni ajoittain hetkiä, jolloin tiedonsiirron kokonaisviive kohosi useaan sekuntiin. Samanlaisia pitempiä aikoja nähtiin myös PostgreSQL-tietokannalla, mutta huomattavasti harvemmin kuin MySQL- tai SQLite-tietokantaa käytettäessä. SQLiten tapauksessa tiedonsiirron kokonaisviive ei ollut koskaan alle sekuntia.

Taulukko 3. Eri tietokantojen vaikutus kokonaisviiveeseen.

Tietokanta	SQLite	MySQL	PostgreSQL
Mittaus 1 (ms)	8240	842	54
Mittaus 2 (ms)	3418	1814	940
Mittaus 3 (ms)	3618	14	38
Mittaus 4 (ms)	3593	25	39
Mittaus 5 (ms)	3429	19	38
Mittaus 6 (ms)	3596	2306	38
Mittaus 7 (ms)	4734	2411	38
Mittaus 8 (ms)	3619	19	40
Mittaus 9 (ms)	3583	19	38
Mittaus 10 (ms)	3614	21	38
Keskiarvo (ms)	4144.4	749	130.1

5.6.6 Usean Gunicorn-työläisen käyttäminen

Testauksien aikana havaittiin myös, että nykyisellä Django-sovelluksen toteutuksella ei ole suositeltavaa käyttää Gunicornin kanssa kuin vain yhtä työläisprosessia. Gunicornin kohdalla Django-sovelluksesta luodaan jokaiselle työläiselle oma instanssinsa, jolloin Djangon käynnistyksessä luotavan yhden keskitetyn sarjaliikenneolion käyttäminen ei ole mahdollista luotujen työläisten kesken, vaan jokaiselle aktiiviselle työläiselle luodaan omansa. Tässä tapauksessa käytännöllisin menetelmä on luoda Serial-olio vain silloin, kun sarjaliikenneväylän yli tarvitsee lähettää viestejä ja sulkea tämä heti viestin lähetyksen jälkeen, jotta vältytään mahdollisilta päällekkäisyyksiltä. Tässä tapauksessa suoratoteutukseen pohjautuva sarjaliikennekäsittelijä on viestien välittämiseen tehokkain, koska säästytään uuden prosessin luomisen aiheuttamalta viiveeltä.

Usean Gunicorn-työläisen käyttämistä tulisi kuitenkin välttää tällaisessa toteutuksessa, jos näiden tarjoamaa lisäsuorituskykyä ei nähdä täysin välttämättömänä. Yhdelläkin työläisellä voidaan saavuttaa erittäin matalia viiveitä alle 5–10 samanaikaisen käyttäjän kohdalla, jos voidaan välttää tietokantaan kirjoittamista.

6 Sarjaliikenneväylässä ilmennyt ongelma ja sen ratkaisu

Rakennettavan prototyypin tärkeänä osana oli sarjaliikenteen yli tapahtuva kommunikaatio, joka toimi oletettavalla tavalla Yocto-pohjaisen jakelun puolella. Kuitenkin vaihdettaessa Yocto-jakelu Debian-jakeluksi lakkasi sarjaliikenne vastaanottamasta viestejä. Pääoire vaikutti olevan sarjaliikenteen reagoimattomuus ulkopuolelta tuleviin viesteihin. Käyttämällä komentoa ”**sudo cat /proc/tty/driver/IMX-uart**” voitiin varmistua, että kehitysalustan laitteisto kuitenkin vastaanotti ulkopuolelta lähetetyt viestit. UART-laitteiston vastaanottamaa viestiä ei kuitenkaan välitetty eteenpäin ylempien ohjelmien saatavaksi.

Pitkän selvittelyn ja yrittämisen jälkeen sarjaliikenne alkoi viimein vastaanottaa viestejä vaihtelevalla varmuudella, kun aikaisemmin kopioimatta jäänyt ”/lib/firmware”-sijainti tiedostoineen kopioitiin Debian-jakelun puolelle. Tämä ei kuitenkaan korjannut ongelmaa täysin, vaan sijainnin kopioimisen jälkeen sarjaliikenneviestien vastaanotto saattoi toimia ensimmäisellä kokeilulla, mutta ei taas enää seuraavalla.

Ratkaisun selvittämisessä eteenpäin pääseminen vaati etätapaamista NXP:n tuen kanssa. Kyseisen tapaamisen päätteeksi saatiin vahva epäilys siitä, että ongelma liittyy jotenkin SDMA-ohjaimen tarvitseman laiteohjelmiston lataamiseen. Tämän selvittämisessä auttoi lokikirjojen tallentaminen eri käynnistyskerroista ja näiden vertaaminen toisiinsa. Perehtyminen laiteajurin lataamisen toimintaan Linux-järjestelmässä johdatti systemd- ja udev-ohjelmien osallisuuteen. Tämä tieto auttoi rajaamaan tiedostojärjestelmästä sijainnit ”/etc”, ”/lib/systemd” ja ”/lib/udev”, joista ratkaisevat eroavaisuudet viimein löytyivät. Eroavaisuudet paikannettiin vertailemalla näiden kriittisten sijaintien sisältöä Yocto-pohjaisen jakelun ja Debianin välillä.

Ensimmäinen eroavaisuus oli, että Debian-jakelusta puuttui sdma-firmware-niminen systemdn yksikköpalvelu, jonka tehtävänä oli ajaa toinen puuttuva ”/etc/sdma”-niminen skripti käynnistyksessä. Tiedostot kopioitiin Yocto-pohjaisesta jakelusta Debianiin ja aktivoitiin systemdn ajettavaksi käynnistyksessä. Tämä ei kuitenkaan vielä ratkaissut ongelmaa, mutta tiedostojen lisääminen oli muuttanut käynnistyksen toimintaa. Tämä ilmeni vertailemalla käynnistyksen lokikirjoja journalctl- ja dmesg-komennoilla. Näistä lokikirjoista selvisi, että välillä sdma-skripti oli onnistunut lataamaan laiteohjelmiston ja välillä jokin oli aiheuttanut tämän epäonnistumisen.

Laiteohjelmiston lataamista osoittautui häirinneen ”/lib/udev/rules.d/50-firmware.rules”-niminen tiedosto, joka käski keskeyttää kaikkien laiteohjelmistojen lataamisen, jos udev saa laiteohjelmiston lataamista pyytävän tapahtuman. Ongelma saatiin ratkaistua poistamalla kokonaan tämä 50-firmware.rules-niminen tiedosto ”/lib/udev/rules.d/”-sijainnista.

Näillä toimenpiteillä SDMA-laiteohjelmiston lataaminen saatiin toimimaan Debianissa samalla tavalla kuin Yocto-pohjaisessa jakelussa. Sarjaliikenteen ongelmat olivat siis johtuneet SDMA-laiteohjelmiston lataamisen epäonnistumisesta, joka esti sarjaliikenneviestin välittämisen UART-laitteistolta käyttöjärjestelmälle. Se, mikä oli saanut viestin välillä välittymään käyttöjärjestelmälle, jäi arvoitukseksi, mutta esitellyn ratkaisun jälkeen sarjaliikenne toimi ongelmitta koko loppukehityksen ajan.

Laiteohjelmiston lataamisen pystyisi toteuttamaan nykyistä versiota paremmin. Yksi tällainen toteutustapa olisi käyttää udev-ohjelmaa laiteohjelmiston lataamiseen. Toinen tehokas tapa tarjota laiteohjelmisto on kääntää tämä osaksi Linux-ydintä, jolloin se on välittömästi ytimen tarjottavissa laiteohjelmistoa pyytävälle laiteajurille.

7 Loppupohdinta

Suoritettu työ tarjosi hyvin monipuolisen poikkileikkauksen sovellutuksen kehityksestä sulautettuun Linux-järjestelmään. Lisäksi tutustuminen Django-kehysympäristöön oli erittäin positiivinen kokemus. Django-pohjaisen verkkosovelluksen kehittäminen osoittautui erittäin nopeaksi sisäistää. Lisäksi työn suoritus lisäsi paljon tietämystä GNU/Linux-käyttöjärjestelmien toiminnasta ja rakenteesta. Työtä tehdessä myös tutustui verkkosovelluksien eri kehitys- ja toteutustyyleihin. Samalla tuli myös harjoittaneeksi omaa osaamistaan verkkosovelluksen kehityksessä.

Työn aloittamiseen ja suunnitteluun olisi pitänyt käyttää huomattavasti enemmän aikaa ja keskittymistä, kuin mitä siihen nyt käytettiin. Tällä tavalla olisi saatu parempi kokonaisymmärrys työn lopputavoitteesta jo heti alussa. Tästä huolimatta käytännön toteutus onnistui tehokkaasti, alun ongelmista ja väärinymmärryksistä huolimatta. Aktiivisen vuoropuhelun avulla työn etenemistä pystyttiin ohjaamaan työn tilanneen yrityksen toivomaan suuntaan, jolloin lopussa päädyttiin kaikkia osapuolia tyydyttävään lopputulokseen.

Arviolta puolet työn toteutusajasta kului sarjaliikenteen kanssa olleen ongelman selvittämiseen. Tämä onneksi saatiin ratkaistua laitteiston kehittäneeltä yhtiöltä saadun avun turvin. Kyseisen ongelman ratkaisua voi pitää jo suurena onnistumisena työn osalta, koska ongelmaan saatiin ratkaisu sen vaativuudesta ja monimutkaisuudesta huolimatta. Lisäksi juuri ongelman syiden selvittäminen ja ratkaisemisyritykset opettivat ehkä eniten Linuxin rakenteesta ja toiminnasta.

Työssä tehdyt mittaukset olisi pitänyt toteuttaa huomattavasti suunnitelmallisemmin, ja tätä kautta selkeämmin. Tällä hetkellä mittausten toistettavuus työn pohjalta ei ole kovin helppoa, kuitenkin käytetyillä menetelmillä saatiin suuntaa antavat tulokset, joiden avulla voitiin riittävissä määrin perustella prototyyppitoteutuksen tehokkuutta. Mielenkiintoisin mittauksissa tehty havainto oli lukumuistin ominaisuuksien merkitys koko järjestelmän suorituskykyyn.

8 Yhteenveto

Työn tavoitteena oli tutustua voiko i.MX 8M Nano -sovellussuorittimen sisältävään järjestelmään asentaa toimivan Debian GNU/Linux-jakelun ja kehittää tälle Django-pohjaisen verkko- ja sarjaliikennepalvelimen. Django-toteutuksen tulisi yksinkertaistaa sarjaliikennelaitteen ja LAN-verkosta tätä ohjaavan käyttäjän välistä vuorovaikutusta. Lisäksi työltä haluttiin selvittää, kuinka suorituskykyinen tällaisesta järjestelmästä saataisiin, ja mitkä olisivat parhaimmat toteutustavat suurimman suorituskyvyn aikaansaamiseksi.

Työn aikana opittiin valmistelemaan ja asentamaan i.MX 8M Nano -kehitysalustalle GNU/Linux-pohjainen Debian-käyttöjärjestelmä. Samalla myös varmistettiin Debian-jakelun käytettävyys i.MX 8M Nano -sovellussuorittimen sisältävällä kehitysalustalla Yocton sijasta. Lisäksi Debian-ympäristössä testattiin NGINX-, Gunicorn- ja Django-pohjainen verkkopalvelimen asennus ja käyttöönotto. Samalla tutustuttiin hieman Linux-pohjaisen järjestelmän sisältämään systemd-sovellusta, jolla voidaan ajoittaa Linux-järjestelmän sovellusten käynnistämistä. Systemd-sovellusta hyödynnettiin myös NGINX- ja Gunicorn-ohjelmien käynnistämiseen muun järjestelmän mukana.

Django-sovelluksen toteutuksessa selvisi, että sarjaliikenneviestien välitystavalla on suuri merkitys toteutettavan järjestelmän suorituskykyyn ja monimutkaisuuteen. Helpoimmaksi ja tehokkaimmaksi tavaksi välittää sarjaliikenneviestejä havaittiin Master-Slave-tyyppinen tiedonsiirto. Tässä toteutuksessa vältyttiin kiertokyselyperiaatteen käytöltä, joka vaatii toimiakseen paljon suoritusaikaa ja voi siksi merkittävästi haitata järjestelmän suorituskykyä.

Järjestelmän testauksessa havaittiin tietokannan käytön vaikuttavan välitysnopeuteen. Merkittävimpänä havaintona oli käytettävän lukumuistityypin vaikutus koko järjestelmään. Tämän havaittiin myös vaikuttavan merkittävästi ohjelman suoritukseen, jos sen aikana vaadittiin tiedon lataamista tai tallentamista tietokantaan. Käytetyn muistityypin ominaisuudet määräsivät hyvin selvästi, kestääkö tietokantaan kirjoittaminen millisekunteja vai sekunteja. Tällainen tilanne kohdattiin käytetyn tietokannan ja hitaan muistikortin kanssa, jossa tietokannan kirjoitukset muistikortille saattoi aiheuttaa useiden sekuntien viivästymisen verkkosovelluksen toiminnassa. Lisäksi nähtiin parhaimpana välttää tietokantaan kirjoittamista toteutuksissa, joissa halutaan mahdollisimman nopeaa suorituskykyä. Kuitenkin käyttämällä muuta tietokantaa, kuin SQLite, voitiin saavuttaa huomattavia suorituskykyhyötyjä.

Lähteet

1 NXP Semiconductors. (3.2020). i.MX 8M Nano Applications Processor Datasheet for Industrial Products. Haettu osoitteesta https://www.nxp.com/docs/en/data-sheet/IMX8MNIEC.pdf

2 NXP Semiconductors. Evaluation Kit for the i.MX 8M Nano Applications Processor. Haettu osoitteesta https://www.nxp.com/design/development-boards/i-mx-evaluation-and-development-boards/evaluation-kit-for-the-i-mx-8m-nano-applications-processor:8MNANOD4-EVK

3 Guru99. What is Operating System? Types of OS, Features and Examples. Haettu osoitteesta https://www.guru99.com/operating-system-tutorial.html

4 Monika Jha. (19.9.2019). Differences between Firmware and Operating System. Haettu osoitteesta https://www.includehelp.com/operating-systems/differences between-firmware-and-operating-system.aspx

5 Yocto Project. GETTING STARTED: THE YOCTO PROJECT OVERVIEW. Haettu osoitteesta https://www.yoctoproject.org/software-overview/

6 Zm Peterson. (8.3.2020). Yocto vs. Debian for Embedded Systems Design and IoT. Haettu osoitteesta https://www.nwengineeringllc.com/article/yocto-vs-debian-for-embedded-systems-design-and-iot.php

7. Mirza Krak. (5.3.2020). How to work with Python applications and modules in Yocto Project. Haettu osoitteesta https://hub.mender.io/t/how-to-work-with-python-applications-and-modules-in-yocto-project/1135

8 Tomi Niemi. (16.2.2015). Sovelluskehys – verkkoräätälin työkalupakki. Haettu osoitteesta https://www.sofokus.com/fi/blogi/2015/02/16/sovelluskehys-verkkoraatalin-tyokalupakki/

9 Harri Laine. (8.10.2012). Ohjelmistoarkkitehtuurit. Haettu osoitteesta https://www.cs.helsinki.fi/u/aptuovin/arkkit/s13/ohjelmistoarkkitehtuurit_11.pdf

10 (2008). Kehysarkkitehtuurit. Haettu osoitteesta https://www.cs.tut.fi/~ohar/luennot/luennot2008/Ohar11%20Kehykset1.pdf

11 Deploying Django. Haettu osoitteesta https://docs.djangoproject.com/en/3.1/howto/deployment/

12 What is a web server? Haettu osoitteesta https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_web_server

13 Writing your first Django app, part 1. Haettu osoitteesta https://docs.djangoproject.com/en/3.1/intro/tutorial01/

14 Django at a glance. Haettu osoitteesta https://docs.djangoproject.com/en/3.1/intro/overview/

15 Writing your first Django app, part 2. Haettu osoitteesta https://docs.djangoproject.com/en/3.1/intro/tutorial02/

16 Matt Makai. Full Stack Python – WSGI servers. Haettu osoitteesta https://www.fullstackpython.com/wsgi-servers.html

17 Raoof Naushad. (24.6.2020). Difference between WSGI and ASGI? Haettu osoitteesta https://medium.com/analytics-vidhya/difference-between-wsgi-and-asgi-807158ed1d4c

18 BSP, Drivers and Middleware. https://www.nxp.com/design/software/embedded-software/i-mx-software/embedded-linux-for-i-mx-applications-processors:IMXLINUX?tab=Design_Tools_Tab

19 Vagrant Cascadian. (2.7.2011). qemu-debootstrap(1). Haettu osoitteesta https://manpages.debian.org/stretch/qemu-user-static/qemu-debootstrap.1.en.html

20 Matt Kraai. (27.4.2001). debootstarp(8). Haettu osoitteesta https://manpages.debian.org/stretch/debootstrap/debootstrap.8.en.html

21 Guilem Jover. (8.2.2007). qemu-user-static. Haettu osoitteesta https://manpages.debian.org/stretch/qemu-user-static/qemu-user-static.1.en.html

22 Logan. (21.1.2017). Introduction to qemu-debootstrap. Haettu osoitteesta http://logan.tw/posts/2017/01/21/introduction-to-qemu-debootstrap/

23 Unzip(1) - Linux man page. Haettu osoiteesta https://linux.die.net/man/1/unzip

24 NXP Semiconductors. (2020). i.MX Linux User's Guide. Rev. L5.4.24_2.1.0.

25 Paul Rubin, David MacKenzie ja Stuart Kemp. dd(1) - Linux man page. Haettu osoitteesta https://linux.die.net/man/1/dd

26 Fdisk(8) - Linux man page. Haettu osoitteesta https://linux.die.net/man/8/fdisk

27 David Engel. Fred N. van Kempen ja Ron Sommeling. mkfs(8) - Linux man page. Haettu osoitteesta https://linux.die.net/man/8/mkfs

28 Dave McKay. (11.10.2019). How to Use the mkfs Command on Linux. Haettu osoitteesta https://www.howtogeek.com/443342/how-to-use-the-mkfs-command-on-linux/

29 David Both. (25.5.2017). An introduction to Linux's EXT4 filesystem. Haettu osoitteesta https://opensource.com/article/17/5/introduction-ext4-filesystem

30 mount(8) - Linux man page. Haettu osoitteesta https://linux.die.net/man/8/mount

31 Linuxize. (23.8.2019). How to Mount and Unmount File Systems in Linux. Haettu osoitteesta https://linuxize.com/post/how-to-mount-and-unmount-file-systems-in-linux/

32 Torbjorn Granlund, David MacKenzie ja Jim Meyering. cp(1) - Linux man page. Haettu osoitteesta https://linux.die.net/man/1/cp

33 i.MX 8M Nano EVK Board Getting Started Guide. Haettu osoitteesta https://www.nxp.com/document/guide/i-mx-8m-nano-evk-board-getting-started-guide:GS-8MNANOLPD4-EVK

34 Chris Liechti. pySerial API. Haettu osoitteesta https://pyserial.readthedocs.io/en/latest/pyserial_api.html

35 Multiprocessing — Process-based parallelism. Haettu osoitteesta https://docs.python.org/3.7/library/multiprocessing.html

36 Threading — Thread-based parallelism. Haettu osoitteesta https://docs.python.org/3.7/library/threading.html

37 (22.12.2020). Global Interpreter Lock. Haettu osoitteesta https://wiki.python.org/moin/GlobalInterpreterLock

38 (8.3.2021). The Python GIL. Haettu osoitteesta https://python.land/python-concurrency/the-python-gil

39 MDN Web Docs Mozilla. Ajax. Haettu osoitteesta https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX

40 Neoteric. (6.12.2016). Single-page application vs. multiple-page application. Haettu osoitteesta https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58

41 Justin Ellingwood ja Hanif Jetha. (25.7.2019). How To Set Up Django with Postgres, Nginx, and Gunicorn on Debian 10. Haettu osoitteesta https://www.digitalocean.com/community/tutorials/how-to-set-up-django-with-postgres-nginx-and-gunicorn-on-debian-10

42 systemd.unit - Unit configuration. Haettu osoitteesta https://www.freedesktop.org/software/systemd/man/systemd.unit.html

43 systemd.syntax - General syntax of systemd configuration files. Haettu osoitteesta https://www.freedesktop.org/software/systemd/man/systemd.syntax.html

44 systemd.socket - Socket unit configuration. Haettu osoitteesta https://www.freedesktop.org/software/systemd/man/systemd.socket.html

45 systemd.service - Service unit configuration. Haettu osoitteesta https://www.freedesktop.org/software/systemd/man/systemd.service.html

46 Configuring NGINX and NGINX Plus as a Web Server. Haettu osoitteesta https://docs.nginx.com/nginx/admin-guide/web-server/web-server/

47 Configuring Logging. Haettu osoitteesta https://docs.nginx.com/nginx/admin-guide/monitoring/logging/

48 Keycdn. (4.10.2018). What Is Latency and How to Reduce It. Haettu osoitteesta https://www.keycdn.com/support/what-is-latency

49 Tiedonsiirtoviiveet. Haettu osoitteesta http://www.oamk.fi/~janneku/T080104/SAU14SNS/images/tcp.pdf

50 SD Association. Speed Class. Haettu osoitteesta https://www.sdcard.org/developers/sd-standard-overview/speed-class/

51 SD Association. Bus Speed (Default Speed/High Speed/UHS/SD Express). Haettu osoitteesta https://www.sdcard.org/developers/sd-standard-overview/bus-speed-default-speed-high-speed-uhs-sd-express/

52 SD Association. Application Performance Class. Haettu osoitteesta https://www.sdcard.org/developers/sd-standard-overview/application-performance-class/

Liitteet

#On Linux PC
sudo apt-get install debian-archive-keyring
sudo apt-key add /usr/share/keyrings/debian-archive-keyring.gpg

sudo qemu-debootstrap --arch=arm64 --keyring /usr/share/keyrings/debian-archive-keyring.gpg --variant=buildd --exclude=debfoster buster debian-arm64 http://ftp.debian.org/debian

#above will create folder debian-arm64 with initial Debian rootfs
#download binary image for i.MX 8M Nano
https://www.nxp.com/webapp/Download?colCode=L5.4.24_2.1.0_MX8MN&appType=license

(
Same with i.MX8M Mini:
https://www.nxp.com/webapp/Download?colCode=L5.4.24_2.1.0_MX8MM&appType=license
)

unzip L5.4.24-2.1.0_images_MX8MNEVK.zip imx-image-full-imx8mnevk.wic
#flash the image to SD card connected to you Linux PC.
sudo dd if=imx-image-full-imx8mnevk.wic of=/dev/sdX bs=1M status=progress

#in case of DDR4, overwrite the u-boot
unzip L5.4.24-2.1.0_images_MX8MNEVK.zip imx-boot-imx8mnddr4evk-sd.bin-flash_ddr4_evk
sudo dd if=imx-boot-imx8mnddr4evk-sd.bin-flash_ddr4_evk of=/dev/sdX bs=1k seek=32 conv=fsync

#create new partition on /dev/sdX3. We'll store the Debian file system there.
sudo fdisk /dev/sdX
#format the file system and copy initial Debian rootfs.
sudo mkfs.ext3 /dev/sdX3
sudo mount /dev/sdX3 /mnt/tmp
cd debian-arm64
sudo cp -rup * /mnt/tmp/
sudo umount /mnt/tmp

#insert the SD card and boot to NXP Linux.
#execute following on i.MX 8MN EVK Linux!
mount /dev/sdX3 /mnt/
#copy lib/modules, lib/firmware from NXP Linux to Debian Linux
#so something like "cp -rup /lib/modules /mnt/lib/modules" etc.

#change rootfilesystem to Debian
chroot /mnt/
bash
echo nameserver 8.8.8.8 >> /etc/resolv.conf
apt-get install ifupdown net-tools network-manager
apt-get install udev sudo ssh
apt-get install vim-tiny
echo debian-imx8mn > /etc/hostname
echo "127.0.0.1   localhost.localdomain localhost" > /etc/hosts
echo "127.0.0.1   debian-imx8mn" >> /etc/hosts
echo "auto eth0" > /etc/network/interfaces.d/eth0
echo "iface eth0 inet dhcp" >> /etc/network/interfaces.d/eth0

#now you're ready. Reboot the system, change default partition in u-boot
#from mmcblk1p2 to mmcblk1p3 and boot the Debian.