Tampereen yliopisto

Ville-Veikko Julku

# AUTOMAATIOLIIKENNESIMULAATTORI

## Kandidaatintyö

Kandidaatintyö
Tekniikan ja luonnontieteiden tiedekunta
Tarkastaja: Jari Seppälä
Elokuu 2021

# TIIVISTELMÄ

Ville-Veikko Julku: Automaatioliikennesimulaattori

Kandidaatintyö

Tampereen yliopisto

Teknisten tieteiden kandidaatin tutkinto-ohjelma

Elokuu 2021

---

Ethernet:in käyttö on yleistynyt automaatioverkoissa viime vuosina. Automaatioverkoilla on erityisesti verkon alimmilla tasoilla vaatimuksia, joihin Ethernet ei ole suunniteltu, eikä Ethernet niitä natiivisti tue. Jotta Ethernet:iä pystyy käyttämään automaatioverkoissa erityisesti sen alimmalla, eli kenttäväylätasolla, tulee Ethernetin perusmekanismeja muokata. Yksi näistä mekanismeista on käyttää erikoistettua tyyppikenttää Ethernet-kehyksessä.

Kandidaatintyö toteutettiin kahdessa osassa. Ensimmäinen osa käsittelee Ethernetin historiaa ja toimintaperiaatetta ja automaatioverkkojen erityispiirteitä sekä automaation Ethernet-toteutusten luokittelua. Toisessa osassa toteutettiin automaatioliikennesimulaattoriasovellus, jolla on mahdollista lähettää automaatioverkoissa käytettäviä Ethernet-kehyksiä ilman tarkoitukseen erikoistettua laitteistoa.

Avainsanat: Ethernet, Automaatioverkot, RTE

# ABSTRACT

Ville-Veikko Julku Automaatioliikennesimulaattori

Bachelor’s theses

Tampere University

Automation Technology

August 2021

---

Use of Ethernet technology has grown its popularity rapidly during recent years. However, automation networks possess requirements which are not native to standard Ethernet especially in the lower levels. In order for automation networks to be able to efficiently utilize the use of Ethernet modifications to Ethernet basic working principles are required. One of these modifications is the use of a specialize type value in the standard Ethernet frame.

This theses constitutes of two parts. First part introduces the history and working principles of original Ethernet as well as requirements for automation networks and means to modify Ethernet in order to utilize the use of Ethernet in automation networks. In the second part automation communication simulator was implemented. Simulator enables transmitting Ethernet frames used in automation networks without having the necessary hardware used in automation networks

Keywords: Ethernet, automation networks, RTE

# SISÄLLYSLUETTELO

1. JOHDANTO ..... 1
2. ETHERNET ..... 2
   2.1 OSI-malli ..... 2
   2.2 Ethernetin historia ..... 3
   2.3 Toimintaperiaate ..... 4
   2.4 Ethernet kehys ..... 5
3. TEOLLINEN ETHERNET ..... 7
   3.1 Automaatioverkot ..... 7
   3.2 Automaatioverkkojen erityispiirteitä ..... 7
   3.3 Reaaliaika-Ethernetin toteutustapoja ..... 11
4. AUTOMAATIOLIIKENNESOVELLUS ..... 13
   4.1 Toimintaperiaate ja kehitysympäristö ..... 13
   4.2 Sovelluksen rakenne ..... 17
5. YHTEENVETO ..... 20
LÄHTEET ..... 21
LIITE A: SIMULATOR_CONFIGURATION.JSON ..... 22
LIITE B: AUTOMATION_ETHERNET_SIMULATOR.PY ..... 23
LIITE C: COMPONENT_BASE.PY ..... 25
LIITE D: CONFIGURATION_LOADER.PY ..... 26
LIITE E: CONSTANTS.PY ..... 27
LIITE F: DATA_CONVERSION_HELPER.PY ..... 28
LIITE G: ETHERNET_TYPE_DATA.PY ..... 29
LIITE H: RECEIVER.PY ..... 30
LIITE I: SENDER.PY ..... 33
LIITE J: USER_INPUT_DISPATCHER.PY ..... 34

# LYHENTEET JA MERKINNÄT

| | |
|---|---|
| OSI | Open Systems Interconnection |
| ISO | International  Organization for Standardization |
| DEC | Digital Equipment Corporation |
| IEEE | Institute of Electrical and Electronic Engineers |
| CSMA/CD | Carrier Sense Multiple Access/Collision Detection |
| MAC | Medium Access Control |
| SFD | Start Frame Delimiter |
| DA | Destination Address |
| SA | Source Address |
| FCS | Frame Check Sequence |
| PLC | Programmable Logic Controller |
| QoS | Quality of Service |
| TCP/IP | Transmission Control Protocol/Internet Protocol. |
| EPA | Ethernet for Plant Automation |
| RPC | Remote Procedure Call |
| DCOM | Distributed Computer Model |

# 1. JOHDANTO

Ethernetin jatkuva kehitys, kasvava tiedonsiirtonopeus ja laitteiston halpa hinta ovat tehneet siitä toimistoverkkojen suosituimman toteutusteknologian. Automaatioverkkojen kehitystyö, hajautus ja kasvavat tiedonsiirtotarpeet ovat puolestaan luoneet tarvetta tutkia mahdollisuuksia Ethernetin hyödyntämiseen automaatioverkoissa. Ethernetin pääperiaatteet eroavat suuresti automaatioverkkojen vaatimuksista, erityisesti OSI-mallin alimmilla kerroksilla. Kuitenkin toimistoverkot ja automaatioverkot ovat integroituneet toisiinsa enenevässä määrin ja Ethernet-pohjaisia automaatioprotokollia ja -verkkoja on jo laajalti saatavilla. Tämän työn tavoitteena oli toteuttaa automaatioliikennesimulaattorisovellus, jonka avulla voidaan lähettää automaatioverkoissa siirrettäviä erikoistettuja Ethernet-kehyksiä ilman siihen erikoistettua laitteistoa.

Tämä työ koostuu teoriaosiosta ja käytännön osiosta. Teoriaosio käsittää luvut 2 ja 3 ja käytännön osio luvun 4. Toisessa luvussa käydään läpi Ethernetin historia ja toimintaperiaate. Kolmannessa luvussa esitellään automaatioverkkojen erityispiirteitä ja vertaillaan niitä Ethernetin toimintaperiaatteeseen. Neljännessä luvussa esitellään työn osana toteutetun automaatioliikennesimulaattorisovelluksen arkkitehtuuri ja toimintaperiaate.

# 2. ETHERNET

## 2.1 OSI-malli

OSI-mallin (Open Systems Interconnection) kehitti International Organization for Standardization (ISO) 1970-luvun loppupuolella. OSI-mallin tarkoituksena oli tarjota viitekehys tiedonsiirrossa tarvittavien protokollien kehitystyölle 7-kerroksisen protokollapinon avulla. Nämä kerrokset ovat  sovelluskerros, esitystapakerros, yhteysjaksokerros, kuljetuskerros, verkkokerros, siirtoyhteyskerros ja fyysinen kerros. Jokaisella protokollakerroksella on OSI-mallissa oma vastuualueensa. Jokainen kerros myös tarjoaa palveluita ylemmälle kerrokselle, ja ylempi kerros voi käyttää alemman kerroksen käyttämiä palveluita. OSI-mallia on sittemmin käytetty laajalti alan kirjallisuudessa ja opetustyössä kuvaamaan tiedonsiirrossa tarvittavia protokollia ja standardeja. Yleisemmin käytössä on TCP/IP-viitemalli.[1] OSI- ja TCP/IP protokollapinojen eroavaisuuksia on esitelty kuvassa 1.

OSI
Sovellus
Esitystapa
Yhteysjakso
Kuljetus
Verkko
Siirtoyhteys
Fyysinen

TCP/IP
Sovellus
Kuljetus
Internet
Verkko
Fyysinen

***Kuva 1.*** *OSI- ja TCP/IP mallit. Mukailtu lähteestä [1].*

## 2.2 Ethernetin historia

Zeroxin Palo Alton tutkimuskeskuksessa tuli 1970-luvun puolivälissä tarve liittää kampusalueen tietokoneita toisiinsa. Sen toteuttaminen operaattorilta vuokrattavilla puhelinyhteyksillä olisi vaatinut suuren määrän kaapelointia näiden tietokoneiden välillä, mistä olisi koitunut suuria kustannuksia. Lisäksi puhelinlinjojen siirtonopeus tuohon aikaan oli keskimäärin 9600 bittiä sekunnissa ja korkeimmillaankin vain 56 kilobittiä sekunnissa. Xeroxilla ei kuitenkaan ollut kiinnostusta alkaa rakentamaan tarvittavaa laitteistoa tai osallistumaan tarvittavan ratkaisun kehitykseen. Samaan aikaan Bob Metcalf, Ethernetin keksijä, lähti Xeroxin palveluksesta. Metcalf tarvitsi kuitenkin sekä piirisarjojen valmistajan että tarvittavia komponentteja valmistavan yhtiön apua, jotta itse verkko voitiin rakentaa. Piirisarjojen valmistajaksi valikoitui Intel ja komponenttien valmistajaksi Digital Equipment Corporation (DEC). Ethernet v 1.0 julkaistiin vuonna 1980. Siinä kampuksen rakennukset kytkettiin yhteen paksuun kaapeliin, mikä kulki kampuksen rakennuksesta toiseen. Tietokoneet kytkettiin verkkoon käyttämällä erityisiä liittimiä, jotka läpäisivät koaksikaapelin vaipan ja muodostivat sähköisen yhteyden sisällä olevaan kuparikaapeliin. [2]

Paranneltu versio Ethernet v 2.0 julkaistiin vuonna 1982, ja se standardoitiin pienin muutoksin Institute of Electrical and Electronic Engineers:in (IEEE) toimesta vuonna 1984. Standardi sai nimekseen IEEE 802.3, ja se määrittelee Ethernetin tyypit 10Base2 ja 10Base5. Standardissa määritellään muun muassa OSI-mallissa mainitun linkkikerroksen datasiirron nopeus (10Mb/s), linkkikerroksen käyttämä protokolla (CSMA/CD, Carrier Sense Multiple Access/Collision Detection) sekä linkkikerroksen datasiirrossa käytettävä datagrammi, Ethernet-kehys. Ethernet-kehys käydään tarkemmin läpi luvussa 2.4 ja CSMA/CD luvussa 2.3. Standardissa määritellään myös OSI-mallin fyysisen kerroksen ominaisuuksia kuten: Ethernetissä voi käyttää kahta erilaista koaksiaalikaapelia, ja yhden verkkosegmentin maksimipituus on 500 metriä. Täten Ethernet sijoittuu OSI-mallissa kahteen alimpaan protokollakerrokseen. Ethernetin kehitys on jatkunut tähän päivään asti. Koaksiaalikaapeli korvattiin ensin kuparijohtimisella kierretyllä parikaapelilla, ja myöhemmin rinnalle tuli myös kuitukaapelin käyttö. [1]

## 2.3 Toimintaperiaate

Alkuperäisen Ethernetin toimintaperiaatteena oli käyttää jaettua siirtotietä kaikkien verkon solmujen kesken. Jaetun siirtotien käytössä ongelmana on, miten päättää lähetysvuorojen jakelusta (engl. Medium Access Control, MAC). Lähetysvuorojen jakelu tarkoittaa sitä, että jollain tavalla pitää päättää, kuka saa kulloinkin lähettää siirtotielle datapaketteja, jotta kaikki eivät lähettäisi samaan aikaan tehden kanavassa kulkevasta datasta tunnistamatonta. Ethernet oli ensimmäinen verkko, jossa lähetysvuorojen jakelu ratkaistiin käyttämällä CSMA/CD-protokollaa. [1]

CSMA/CD kuuluu satunnaisen pääsyn protokolliin (Random Access Protocol). Satunnaisen pääsyn protokollille ominaista on, että lähettävä solmu lähettää siirtotielle dataa maksiminopeudella ja kuuntelee siirtotietä samaan aikaan lähettämisen aikana. Jos lähettävä solmu havaitsee törmäyksen, se odottaa satunnaisen ajan ja yrittää lähettää saman datan uudestaan. Törmäyshetkellä lähettäviä solmuja voi olla useita, ja niistä jokainen valitsee uudelleen lähetykseen kuluvan satunnaisen ajan itsenäisesti. [1]. CSMA/CD:n toimintaperiaate on kuvattu alla:

1. Jos siirtotie on vapaa, lähetä. Muuten mene kohtaan 2.
2. Jos siirtotie on varattu, jatka kuuntelemista, kunnes siirtotie on vapaa. Kun siirtotie on vapaa, lähetä.
3. Jos tapahtuu törmäys, lähetä törmäyksen ilmaiseva datavirta (engl. jam signal) siirtotielle, jotta kaikki solmut saavat tiedon törmäyksestä.
4. Törmäyksen ilmaisevan datan lähetyksen jälkeen odota satunnainen aika, minkä jälkeen yritä uudelleen
5. Mikäli uudelleenlähetystä on odotettu 16 kertaa, raportoi virhe lähetyksessä. [4]

Ethernetissä käytetään satunnaisen ajan päättelyyn binary exponential backoff - algoritmia. Olkoon *n* peräkkäisten törmäysten lukumäärä ja *K* satunnainen luku joukosta:

$$\{0,1,2,\ldots 2^{n-1}\}, n \leq 10$$

Olkoon *t* aika, jonka kestää 512:sta bitin lähetys siirtotielle. Tällöin odottamiseen käytettävä satunnainen aika *r* valitaan:

$$r = t * K$$

[1]

Alkuperäisessä Ethernetissä jokainen verkon solmu oli kytketty samaan koaksiaalikaapeliin, joka kiersi ympäri kampusta. Myöhemmin yksittäinen kaapeli korvattiin kierretyllä parilla yksittäisellä kierretyllä kaapelilla per laite ja keskittimellä, johon kaikki laitteet kyt-

kettiin. Keskitin lähettää vastaanottamansa kehyksen jokaiselle siihen kytketylle laitteelle. Sekä alkuperäinen, yhdellä koaksiaalikaapelilla tehty verkko, että keskittimellä ja kierretyillä parikaapeleilla kytketty verkko toimivat half-duplex moodissa ja käyttivät CSMA/CD:ä lähetysvuoron valinnan ratkaisemiseen. Half-duplex tarkoittaa sitä, että verkon solmu ei voi lähettää ja vastaanottaa samaan aikaan. [1]

Ethernetissä jokainen solmu identifioidaan kuusi tavua pitkällä MAC-osoitteella. Lähetettävä datagrammi, Ethernet-kehys sisältää tiedon lähettäjän ja vastaanottajan MAC-osoitteesta. 2000-luvun alkupuolella toistimia alettiin korvata kytkimillä. Kytkin ylläpitää itse kytkentätaulua (engl. switch table), jossa määritellään, mikä kytkimen fyysinen portti liittyy tiettyyn MAC-osoitteeseen. Kytkin siis tietää, mihin fyysiseen porttiin kehys tarvitsee kussakin tilanteessa lähettää. Kytkin toimii full-duplex – moodissa, eli samasta portista voi samaan aikaan sekä lähettää, että vastaanottaa Ethernet-kehyksiä. Se myös poistaa kytkentätaulusta tietueita MAC-osoitteiden ja porttien välisistä sidoksista ajastetusti, tyypillisesti 10 sekunnin välein. Mikäli kehyksen vastaanottajan MAC-osoitetta ei löydy kytkentätaulusta, lähettää kytkin kehyksen kaikille porteille, eli tekee broadcast-lähetyksen. Kytkimessä on myös kehysten puskurointi jokaiselle portille. Kytkimen käytön yleistyttyä CSMA/CD-protokollan käytölle ei ollut enää tarvetta, koska törmäyksiä ei enää tapahdu. Nykyinen Ethernet tunnetaankin hyvin erilaisena, kuin IEEE 802.3 – standardin alun perin määrittelemä Ethernet. Ethernet-kehys on kuitenkin jäänyt pohjalle kaikkiin nykyisiin Ethernet-toteutuksiin. [3]

## 2.4 Ethernet kehys

IEEE 802.3 – standardissa on määritelty Ethernet-kehykselle minimimitta. Tämä johtuu siitä, että törmäysten havaitsemiseksi pitää kehyksen lähetyksen olla vielä käynnissä siinä vaiheessa, kun kehyksen ensimmäinen nouseva reuna saavuttaa verkon kauimmaisen vastaanottajan. Eli lähettäjän pitää tunnistaa mahdollinen törmäys ennen kehyksen lähetyksen loppumista. [3] Taulukko 1 havainnollistetaan IEEE 802.3 Standardin määrittelemä Ethernet-kehys:

***Taulukko 1**. IEEE 802.3 standardin mukainen Ethernet kehys. Muokattu lähteestä [3].*

| 7 | 1 | 6 | 6 | 2 | 46-1500 | 4 |
|---|---|---|---|---|---|---|
| **Preamble** | **S<br>F<br>D** | **DA** | **SA** | **Length/Type** | **Data** | **FCS** |

Ethernet-kehys alkaa purskeen alustusosalla, joka on 7 tavua pitkä ja koostuu vuorottaisista biteistä 1 ja 0. alustusosan tarkoitus on saavuttaa bittivirran synkronointi lähettäjän ja vastaanottajan välillä. Alustusosaa seuraa kehyksen aloituksen erotin (Start Frame Delimiter, SFD), joka on aina bitteinä 10101011. Erottimen avulla vastaanottajan on mahdollista tunnistaa, mistä itse kehys alkaa. Tämän jälkeen tulevat kehyksen vastaanottajan MAC-osoite (Destination Address, DA) ja lähettäjän MAC-osoite (Source Address, SA). MAC-osoite yksilöi jokaisen Ethernet-verkkoon kytketyn verkkokortin (Network Interface Card, NIC). Seuraavaksi on kehyksen pituutta ilmaiseva kenttä (Length), joka voi olla myös kehyksen tyyppiä ilmaiseva kenttä (Type). Alkuperäisen Ethernetin määrittelyssä kenttä ilmaisi kehyksen tyyppiä, mutta IEEE 802.3 – standardissa kenttä ilmaisi kehyksen pituutta. Data-kenttä sisältää varsinaisen kehyksessä lähetettävän datan. Data-kentän pituus voi olla välillä 46-1500 oktettia. Mikäli varsinainen data ei ole vähintään 46 oktettia pitkä, lisätään datan perään lisää oktetteja, kunnes minimimitta 46 oktettia täyttyy. Viimeisenä on kehyksen tarkistussekvenssi (engl. Frame Check Sequence, FCS). Mikäli FCS:n tarkistus ei mene läpi, kehys hylätään. [3]

# 3. TEOLLINEN ETHERNET

## 3.1 Automaatioverkot

Modernien automaatiojärjestelmien voidaan todeta saaneen alkunsa kenttäväylistä. Kenttäväylä on digitaalinen kaksisuuntainen kommunikaatiolinkki, joka toimii lähiverkkona edistyneelle prosessinohjaukselle kytkien toisiinsa varsinaisen ohjaimen sekä älykkäät anturit ja toimilaitteet. Ohjaimena voi toimia ohjelmoitava logiikka (PLC) tai sulautetuissa järjestelmissä tehtävään erikoistettu mikrokontrolleri tai tietokone. [5].

Kenttäväylässä kommunikaatio tapahtuu eritysten tarkoitukseen suunniteltujen protokollien avulla. Viimeisen kymmenen vuoden ajan on myös Ethernet-teknologiaa alettu enenevässä määrin hyödyntämään sekä kenttäväylätasolla, että automaatioverkkojen ylemmissä kerroksissa. Koska Ethernet on alun perin suunniteltu toimistokäyttöön, sen perusperiaatteet eroavat merkittävästi automaatioverkkojen vaatimuksista. [6]

## 3.2 Automaatioverkkojen erityispiirteitä

Automaatioverkot kytkeytyvät verkkoarkkitehtuurin alimmalla tasolla fyysisiin laitteisiin, joita käytetään valvomaan ja ohjaamaan reaalimaailman toimintoja, minkä vuoksi niillä on erilaiset vaatimukset, kuin toimistoverkoilla. Näistä keskeisimpiä ovat palvelun laatu (QoS), deterministisyys sekä reaaliaikainen tiedonsiirto. [6]. Automaatio- ja toimistoverkkojen välisiä eroavaisuuksia on esitelty Taulukossa 2.

***Taulukko 2.*** *Teollisten ja perinteisten verkkojen vertailu. Muokattu lähteestä [6].*

| | **Teollinen** | **Toimisto** |
|---|---|---|
| Ensisijainen toiminto | Fyysisten laitteiden ohjaus | Datan käsittely ja -siirto |
| Sovellusalue | Valmistus, prosessiteollisuus sekä infrastruktuurijärjestelmät, kuten sähköverkko | Yritykset ja kotitaloudet |
| Hierarkia | Syvä, toiminnallisesti eriytetyt hierarkiat. Käytössä monia protokollia sekä fyysisiä standardeja | Matala, integroidut hierarkiat, yhtenäinen protokolla sekä fyysinen standardi. |

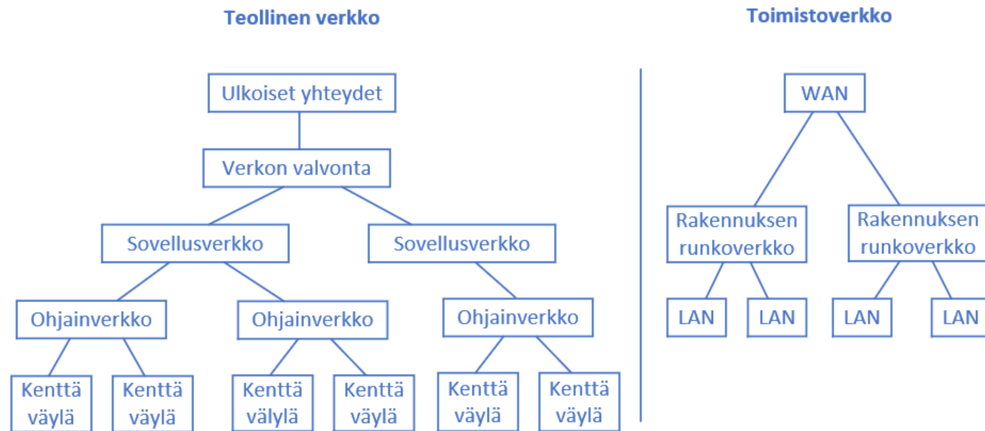
| Vikaantumisen vakavuus | Korkea | Matala |
|---|---|---|
| Luotettavuusvaatimus | Korkea | Matala |
| Vasteaika | 250 µs – 10 ms | 50+ ms |
| Deterministisyys | Korkea | Matala |
| Tiedon koostumus | Pienistä paketeista koostuva jaksottainen ja jaksoton liikenne | Suuret jaksottomat paketit |
| Ajallinen konsistenssi | Vaaditaan | Ei vaadita |
| Toimintaympäristö | Vaativat olosuhteet, joissa on usein suuret määrät pölyä, korkeat lämpötilat sekä tärinää. | Puhdas ympäristö, usein suunniteltu nimenomaan herkälle laitteistolle |

**Implementointi**

Teollisia verkkoja tarvitaan lähes jokaisessa tilanteessa, missä on tarve ohjata tai monitoroida koneita tai laitteita. Näitä ovat esim. kappaletavara- ja prosessiteollisuus, sähkön- tai vedenjakelu, elintarviketeollisuus sekä rakennusautomaatio. Eri teollisuudenaloilla on tyypillisesti hieman toisistaan eroavia vaatimuksia automaatioverkon implementoinnille. [6]

**Arkkitehtuuri**

Toimistoverkkojen arkkitehtuuri on tyypillisesti matalampi kuin teollisten verkkojen. Toimistoverkoissa saattaa olla vain lähiverkko, joka on kytketty runkoverkkoon, mutta teollisissa verkoissa on yleensä vähintään kolme tai neljä kerrosta syvä hierarkia. Esimerkiksi toimilaitteet on kytketty PLC:hen alimmalla, eli tyypillisesti kenttäväylätasolla. Kenttäväylät voivat olla joko keskenään tai käyttöliittymään kytkettynä seuraavalla tasolla, jonka päällä voi olla vielä datakeruutaso. [6]. Toimisto- ja automaatioverkon arkkitehtuurien eroja on havainnollistettu kuvassa 2.



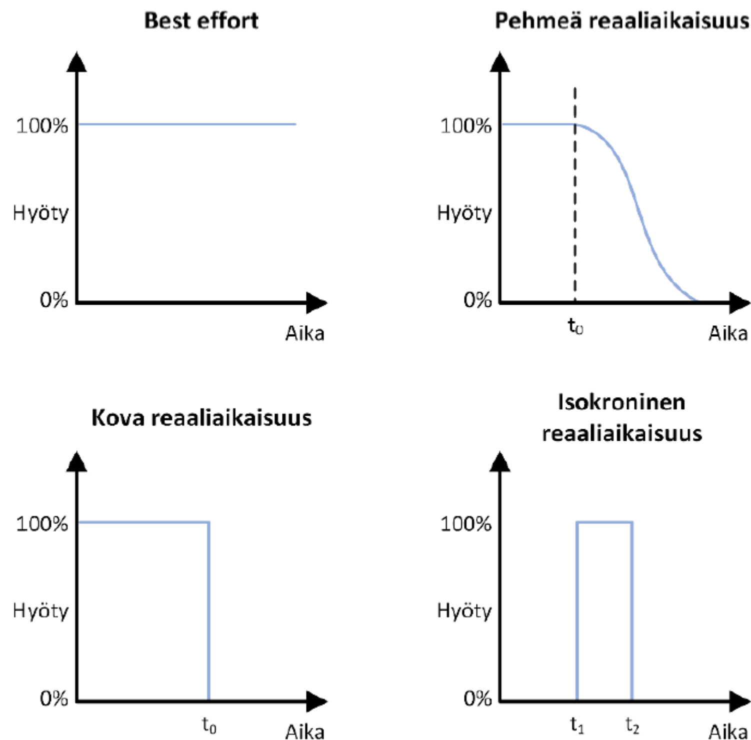

***Kuva 2.*** *Esimerkki teollisen ja toimistoverkon hierarkian tasoista. Muokattu lähteestä [6].*

**Vikaantumisen vakavuus**

Koska teolliset verkot ovat kytkettyinä fyysisen maailman laitteisiin, voi teollisen verkon vikaantumisella olla seurauksia, jotka voivat johtaa tuotannon menetykseen ja taloudellisiin tappioihin, laitteiden hajoamiseen, ympäristötuhoihin tai jopa henkilövahinkoihin [6].

**Reaaliaikavaatimukset**

Nopeus, jolla laitteet ja prosessit toimivat asettaa vaatimuksia myös datan siirron, prosessoinnin ja vastauksen lähettämisen nopeudelle. Esimerkiksi liikkeenohjauksen sovelluksissa tämän ajanjakson pituusvaatimus voi olla välillä 250 µs – 1 ms ja poikkeukset vasteajassa voivat johtaa merkittäviin toimintahäiriöihin ohjausjärjestelmissä. Toimistoverkoissa taas reaaliaikavaatimuksia ei tyypillisesti ole. [6] Reaaliaikavaatimuksia voidaan kategorisoida neljällä eri tavalla vaaditun vasteajan perusteella. Nämä kategoriat ovat best effort, pehmeä, kova sekä isokroninen reaaliaikaisuus. [7] Reaaliaikajärjestelmien kategorisointia on havainnollistettu kuvassa 3.




***Kuva 3.*** *Reaaliaikajärjestelmien kategorisointi [8].*

Best effort:ille ominaista on, että järjestelmän suorittama toiminto on hyödyllistä minä ajanhetkenä tahansa, eikä varsinaista reaaliaikavaatimusta ole. Pehmeässä reaaliaikavaatimuksessa suoritetun toiminnon hyödyllisyys heikkenee ajan funktiona siitä, mitä kauempana määritellystä aikarajasta toiminnon suoritus tehdään. Kovassa reaaliaikaisuudessa järjestelmän tulee suorittaa tietty toiminto tiettyyn aikarajaan mennessä. Aikarajan ylittäminen johtaa järjestelmän virheeseen tai siihen, että toiminto on hyödytön. Isokroninen reaaliaikaisuus taas tarkoittaa sitä, että toiminto tulee suorittaa tietyn aikarajan sisällä. [7]

**Deterministisyys**

Teollisuusverkkojen alimmalla tasolla on riittävän reaaliaikaisuusvaatimuksen lisäksi tarve sille, että data siirto tapahtuu ennustettavalla tai deterministisellä tavalla. Deterministisyys tarkoittaa sitä, että tulee olla mahdollista ennustaa, mikä on datan lähetyksen ja vastaanoton aikaväli. Tämä tarkoittaa sitä, että signaalin latenssin tulee olla rajattu ja varianssin pieni. Vasteajan varianssia kutsutaan jitteriksi. Säätöpiireissä vaaditaan usein myös pientä jitteriä. Toimistoverkot taas ovat tyypillisesti sietokykyisiä suuremmalle jitterille. [6]

**Datapakettien koko**

Teollisuusverkkojen alimmilla tasoilla datapaketit ovat tyypillisesti pieniä. Yhden paketin data voi sisältää esim. yhden mittarin anturin arvon tietyllä ajanhetkellä. Kuten luvussa 2.4 todettiin, Ethernet paketin pienin koko puolestaan on 48 oktettia, mikä on teollisuusverkon alimpien tasojen käyttötarkoitusta varten suuri. [6]

**Jaksollinen ja jaksoton liikenne**

Teollisissa verkoissa liikkuu sekä jaksollista, sykliajan rajoissa tapahtuvaa liikennettä, että jaksotonta liikennettä. Jaksollinen liikenne sisältää tyypillisesti antureiden tilaa ja toimilaitteiden ohjaussignaaleja, kun taas jaksoton liikenne on vaikkapa järjestelmän laitteessa tapahtuvan hälytyksen tilan viestittämistä järjestelmän muille laitteille tai järjestelmän tilassa tapahtuva tilanmuutos. Automaatioverkkojen jaksoton liikenne voi tapahtua milloin tahansa. [6]

**Ajallinen eheys ja tiedon järjestys**

Teollisissa verkoissa on usein tarve tietää datan lähetyksen ajanhetki, sekä varmistaa verkossa kulkevien datapakettien järjestys, erityisesti jaksottomassa liikenteessä. Tämä aikaansaadaan datapakettien aikaleimoilla ja verkossa olevien laitteiden kellojen synkronoinnilla. Ajallinen eheys ja tiedon järjestys eivät tyypillisesti ole osa toimistoverkkojen tiedonsiirrossa käytettyjä protokollia. [6]

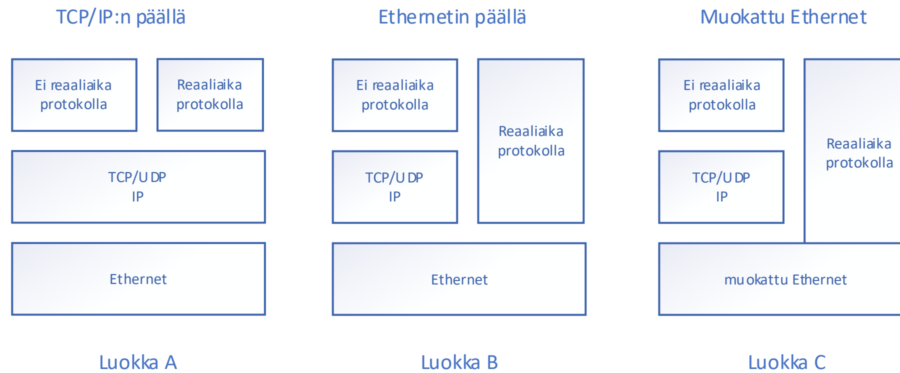
**Fyysiset olosuhteet**

Toimistoverkon laitteet sijaitsevat tyypillisesti puhtaissa tiloissa, joissa lämpötila on säädetty ilmastoinnilla. Teollisten verkkojen sovelluskohteet taasen sijaitsevat karuissa olosuhteissa, joissa kosteus, pöly, tärinä ja erinäisten kemikaalien läsnäolo voi olla tavallista. Tämä asettaa vaatimuksia sekä kaapeloinnin, että liittimien laadulle. [6]

## 3.3 Reaaliaika-Ethernetin toteutustapoja

Kuten luvussa 3.2 havainnollistettiin, eroavat toimistokäyttöön suunnitellun Ethernet-verkon, sekä automaatioverkon vaatimukset toisistaan. Markkinoilla kuitenkin on erilaisia Ethernet-pohjaisia reaaliaikasovelluksia, joissa käytettävää protokollapinoa muokataan toteutustekniikasta riippuen. Reaaliaika-Ethernetin toteutukset perustuvat TCP/IP-pinoon, jossa reaaliaikaprotokolla voi sijaita protokollapinon eri kerroksilla. Erilaisille toteutustavoille yhteistä on Ethernet-kaapelointi, eli fyysinen kerros, sekä TCP:n ja UDP:n

käyttö ei-reaaliaikaiselle liikenteelle. Toteutustapoja voidaan luokitella kolmella eri tavalla: TCP/IP-kerroksen päällä, Ethernet-kerroksen päällä ja muokattu Ethernet. [5] Reaaliaika-Ethernetin luokittelutavat on havainnollistettu kuvassa 4.




***Kuva 4.*** *Reaaliaika-Ethernetin luokittelu. Mukailtu lähteestä [5].*

Luokan A sovelluksissa TCP/IP-mallin sovelluskerros vastaa reaaliaikavaatimusten tapahtumisesta. Nämä sovellukset pystyvät takaamaan tietyn tason reaaliaikavaatimuksen muokkaamalla TCP/IP protokollapinon sovelluskerrosta. Luokan A sovellukset toteuttavat lähinnä best-effort reaaliaikavaatimuksen. Tämän luokan sovelluksia ovat esim. Modbus/TCP ja Ethernet/IP. [5]

Luokan B sovelluksissa on reaaliaikaominaisuudet rakennettu standardi Ethernetin päälle ja ne käyttävät omaa arvoa Ethernet-kehyksen tyyppikentässä, eikä kyseisen luokan sovelluksissa Ethernet ole täten enää standardi Ethernet. Luokan B sovellukset vaativat myös erikoistettua laitteistoa, jotta k.o. tyyppikenttää voidaan verkon eri laitteissa tulkita, ja reaaliaika Ethernetin kehyksiä priorisoida. Itse Ethernetin toimintaperiaate on kuitenkin sama, kuin normaalissa Ethernet-verkossa. Luokan B sovelluksissa ei käytetä standardia TCP/IP pinoa reaaliaikaisessa kommunikoinnissa. Näitä ovat esim. PROFINET, Ethernet Powerlink ja Ethernet for Plant Automation (EPA). [5]

Luokan C sovelluksissa taas on muokattu Ethernetin fundamentaalisia mekanismeja ja myös verkon topologia on tyypillisesti joko rengas- tai väylätopologia. Luokan C sovelluksissa ei siis ole standardi Ethernetin toimintaperiaatteista jäljellä muuta, kuin Ethernet-kehys. Tämän luokan sovelluksilla on mahdollista saavuttaa kova tai isokroninen reaaliaikavaatimus. Luokan C sovelluksia ovat esim. EtherCAT ja SERCOS III. [6]

# 4. AUTOMAATIOLIIKENNESOVELLUS

Tämän työn yhteydessä toteutettiin myös automaatioverkon Ethernet-liikennettä simuloiva sovellus. Sovelluksen avulla on mahdollista lähettää erinäisiä reaaliaika Ethernet-kehyksiä, missä Ethernetin tyyppikentän arvo on valittavissa käyttäjän syötteen mukaan. Sovellukseen valikoitui mukaan luvussa 3.3 mainittuja luokan B ja C reaaliaika Ethernetin kehystyyppejä. Taulukossa 3 on esitelty automaatioliikennesovelluksen tukemat Ethernet-kehykset.

***Taulukko 3.*** *Automaatioliikennesovelluksen Ethernet-kehyksen tyyppiarvot. Mukailtu lähteestä [5].*

| Reaaliaika Ethernet sovellus | Ethernet kehyksen tyyppikentän arvo |
|---|---|
| Ethernet Powerlink | 0x88AB |
| TCNet | 0x888B |
| EPA | 0x88CB |
| PROFINET | 0x8892 |
| EtherCAT | 0x88A4 |
| SERCOS III | 0x88CD |

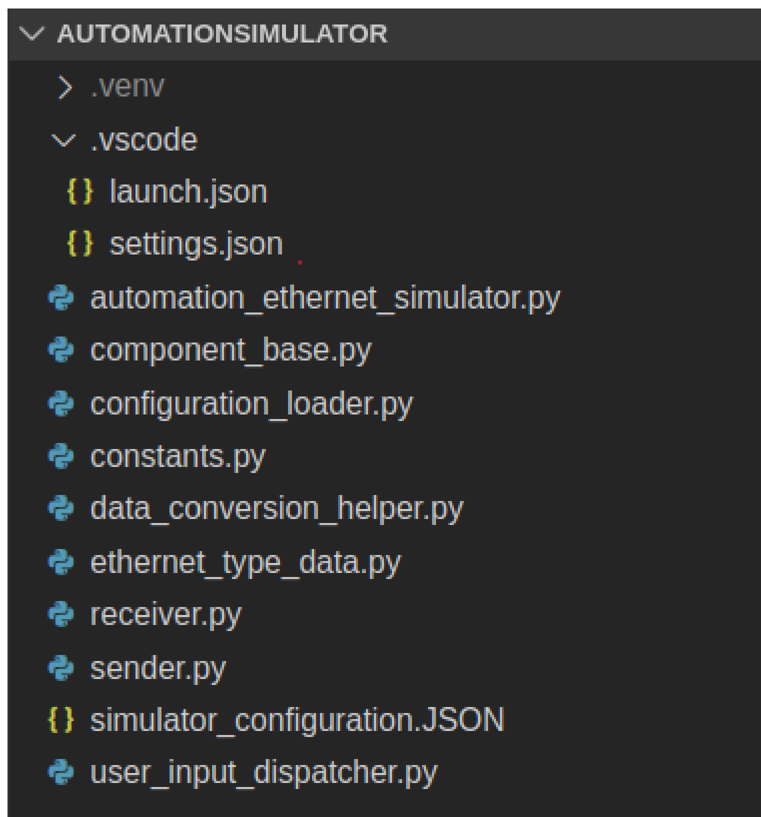
## 4.1 Toimintaperiaate ja kehitysympäristö

Automaatioliikennesovellus pohjautuu hajautettuun asiakas/palvelin malliin. Siinä järjestelmän toimintaa on mahdollista suorittaa tietokoneen eri prosesseissa tai kokonaan eri tietokoneissa tai palvelimissa. Palvelin tyypillisesti tarjoaa jotain palvelua, jota asiakasprosessit tai tietokoneet voivat käyttää. [9] Automaatioliikennesovelluksen jokainen instanssi voi toimia joko palvelimena tai asiakkaana ja palvelimena. Eli sama instanssi voi joko lähettää, tai lähettää ja vastaanottaa Ethernet kehyksiä.

Sovelluksen toteutuskielenä käytettiin python-ohjelmointikielen versiota 3.7.6. Python valikoitui käyttöön siksi, että se tarjoaa socket-kirjaston millä on mahdollista lähettää muokattuja Ethernet-kehyksiä Ethernet verkossa. Windows käyttöjärjestelmässä on muokattujen Ethernet-kehysten lähetystä rajoitettu, eikä Windows:in python-kääntäjä tue socket-kirjastoa, joten käyttöjärjestelmäksi valikoitui Kali Linux. Se on suunniteltu erityisesti tietoturvatestaukseen, joten Windowsille ominaisia rajoituksia lähetettävien Ethernet-kehyksien tyypistä ei ole [10]. Kali Linux asennettiin VirtualBox-virtualisointialustalle

ja VirtualBox:ista käytettiin versiota 6.1.12. Virtualisointi osoittautui hyväksi vaihtoehdoksi myös siitä syystä, että sen avulla on mahdollista ajaa samalla PC:llä kahta eri automaatioliikennesovelluksen instanssia ja konfiguroinnin avulla voidaan vaihtaa vastaanottajan MAC-osoite halutuksi VirtualBox-alustan toimiessa lähiverkon kytkimenä. Tällä tavoin saatiin simuloitua tilannetta, jossa kaksi sovelluksen instanssia ovat samassa lähiverkossa, mutta eri Ethernet verkkokorttiin kytkettynä.

Sovelluksen kehitystyö tehtiin Visual Studio Code-kehitystyökalua käyttäen. Sovellus on rakennettu siten, että liitteissä A-J kuvattujen python-tiedostojen tulee sijaita samassa kansiossa. Kuvassa 5 on esitelty automaatioliikennesovelluksen rakenne Visual Studio Code – kehitystyökalussa.

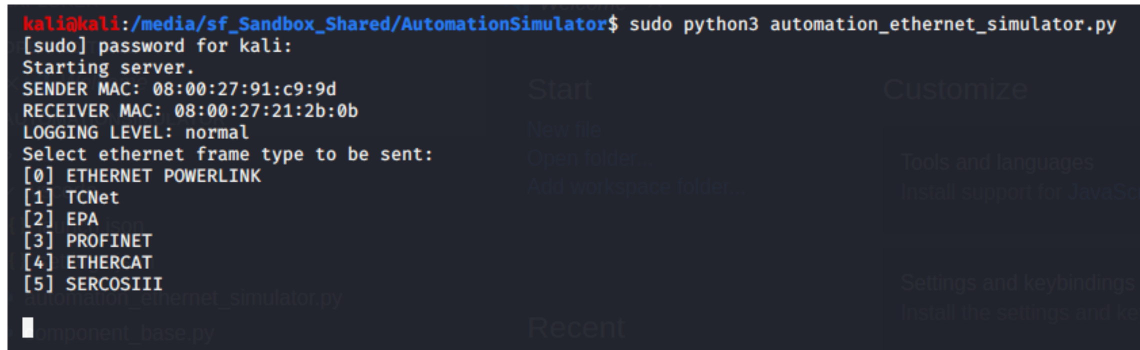



***Kuva 5.*** *Automaatioliikennesovelluksen rakenne.*

Sovellukselle tulee konfiguroida lähettäjän MAC-osoite, vastaanottajan MAC-osoite, kehyksien lähetyksen intervalli, sekä sovelluksen tuottaman lokituksen taso. Sovelluksen konfiguraatio on esitelty liitteessä A. Kun soveltuva konfiguraatio on asetettu, voidaan sovellus käynnistää navigoimalla Linux:n terminaaliapplikaatiolla kansioon, mistä liitteissä A-J mainitut tiedostot löytyvät ja kirjoittamalla terminaaliin:

*sudo python3 automation_ethernet_simulator.py*

Sovelluksen käynnistyessä käyttäjältä kysytään lähetettävän Ethernet-kehyksen tyyppi, minkä jälkeen sovellus lähettää konfiguroidun intervallin välein valitun tyyppisen Ethernet-kehyksen konfiguroituun kohde MAC-osoitteeseen. Käyttäjä valitsee numerolla 0-5, minkä tyyppisen Ethernet-kehyksen haluaa sovelluksen lähettävän. Kuvassa 6 on esitelty automaatioliikennesovelluksen käynnistysruutu.




***Kuva 6.*** *Automaatioliikennesovelluksen käynnistys.*

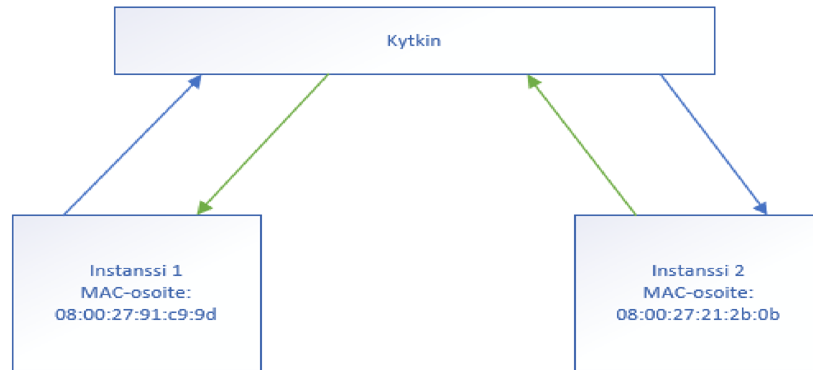
Käytännössä sovellusta ajettiin kahdella eri VirtualBox-virtuaalikoneella. Ensimmäiselle virtuaalikoneelle on konfiguroitu vastaanottajan MAC-osoitteeksi toisen virtuaalikoneen MAC-osoite ja päinvastoin. Kuvassa 7 on esitelty sovelluksen kahden instanssin konfigurointi.

```
{
  "SENDER_MAC_ADDRESS" : "08:00:27:21:2b:0b",
  "RECEIVER_MAC_ADDRESS" : "08:00:27:91:c9:9d",
  "LOGGING_MODE" : "normal",
  "SEND_INTERVAL_SECONDS" : "2"
}
```

```
{
  "SENDER_MAC_ADDRESS" : "08:00:27:91:c9:9d",
  "RECEIVER_MAC_ADDRESS" : "08:00:27:21:2b:0b",
  "LOGGING_MODE" : "normal",
  "SEND_INTERVAL_SECONDS" : "5"
}
```

***Kuva 7.*** *Sovelluksen instanssien konfiguroinnit.*

Sovelluksen instanssit siis tietävät toistensa MAC-osoitteet, jotta ne osaavat sijoittaa lähetettävään Ethernet-kehykseen oikean vastaanottajan MAC-osoitteet. Sovelluksesta on mahdollista olla rajaton määrä instansseja, jotka voivat lähettää keskenään Ethernet-kehyksiä, kunhan vastaanottajan MAC-osoite on sellainen, joka sijaitsee verkon samassa kytkimessä. Kahden instanssin liikennettä on havainnollistettu kuvassa 8.




***Kuva 8.*** *Sovelluksen toimintaperiaate.*

Sovelluksen toimiessa sekä asiakkaana että palvelimena se siis lähettää ja vastaanottaa kehyksiä samaan aikaan. Esimerkki-installaation instanssin 1 tulostusta on havainnollistettu ohjelmassa 1.

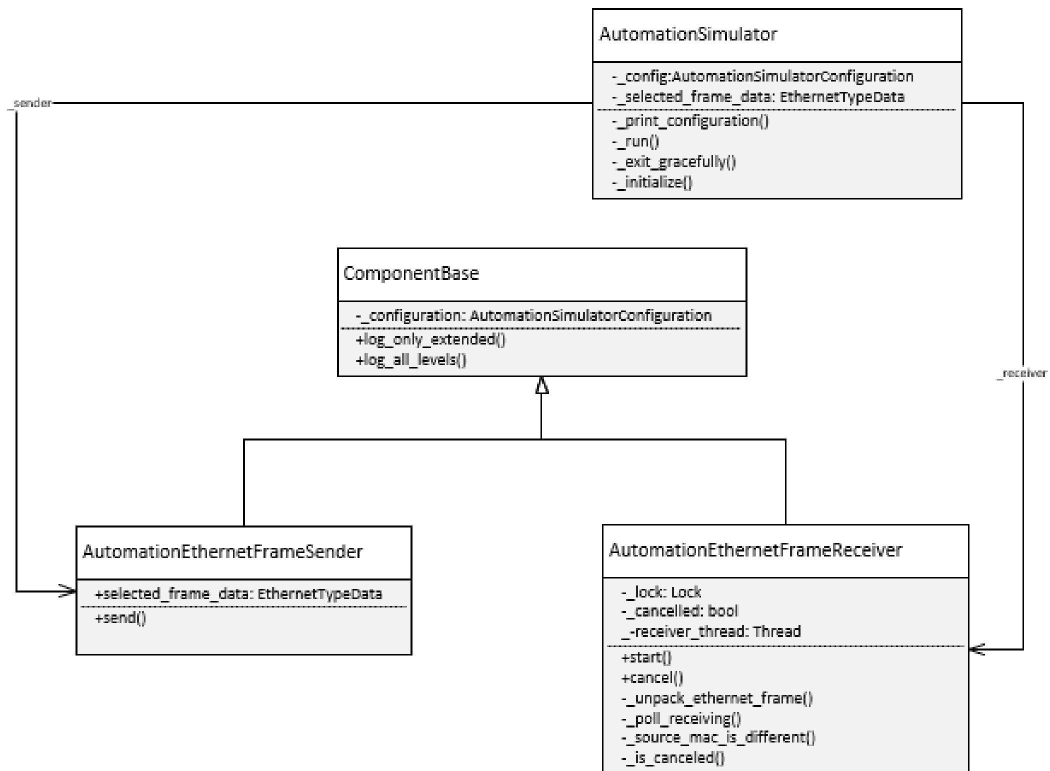
```
   //Ethernet Powerlink-tyyppisen kehyksen lähetys
   Frame of type: ETHERNET POWERLINK sent

   //PROFINET-tyyppisen kehyksen vastaanotto osoitteesta 08:00:27:21:2b:0b
   RECEIVED ACCEPTED FRAME TYPE: PROFINET:
   DATA RECEIVED: b'Ethernet sockets are fun with python, greetings from:
08:00:27:21:2b:0b'

   //PROFINET-tyyppisen kehyksen vastaanotto osoitteesta 08:00:27:21:2b:0b
   RECEIVED ACCEPTED FRAME TYPE: PROFINET:
   DATA RECEIVED: b'Ethernet sockets are fun with python, greetings from:
08:00:27:21:2b:0b'

   //PROFINET-tyyppisen kehyksen vastaanotto osoitteesta 08:00:27:21:2b:0b
   RECEIVED ACCEPTED FRAME TYPE: PROFINET:
   DATA RECEIVED: b'Ethernet sockets are fun with python, greetings from:
08:00:27:21:2b:0b'

   //Ethernet Powerlink-tyyppisen kehyksen lähetys
   Frame of type: ETHERNET POWERLINK sent
```

***Ohjelma 1.*** *Sovelluksen instanssi 1:n tulostus.*

Kuvasta on havaittavissa, että instanssi 1 lähettää kehystyyppiä Ethernet Powerlink ja instanssi 2 lähettää kehystyyppiä PROFINET. Instanssi 1:n lähetysfrekvenssi on myös pienempi, kuin toisen lähettävän instanssin koska instanssi 1 vastaanottaa kehyksiä useammin, kuin lähettää. Kehyksen datakentässä on viesti: Ethernet sockets are fun with python, greetings from: 08:00:27:21:2b:0b, missä 08:00:27:21:2b:0b on lähettäjän MAC-osoite.

## 4.2 Sovelluksen rakenne

Sovellus koostuu useasta eri python-tiedostosta, joihin on määritelty staattisia eli tilattomia tai tilallisia luokkia ja joilla on sovelluksen toiminnan kannalta omat määritellyt roolinsa. Sovelluksen pääluokat ovat tilallisia ja apuluokat tilattomia. Sovelluksen pääluokat on esitelty kuvassa 7.




***Kuva 9.*** *Automaatioliikennesovelluksen pääluokat.*

Pääluokkien lisäksi sovelluksessa on useita apuluokkia käyttäjän syötteen tulkitsemiseen, datan muokkaukseen Ethernet-kehykseen sopivaksi, sekä vastaanotetun Ethernet-kehyksen tulkitsemiseen. Lähetyksen ja vastaanoton luokat on periytetty kantaluokasta ComponentBase, jotta sovelluksen tulostus ja konfiguraation käyttö saadaan helposti yhdenmukaistettua.

#### AutomationSimulator

Sovellus käynnistyy luomalla AutomationSimulator-luokasta olion. AutomationSimulator lukee alustuksen yhteydessä sovelluksen konfiguraation, pyytää käyttäjältä halutun Ethernet-kehyksen tyypin, luo kehyksiä lähettävän ja vastaanottavan luokan ja käynnistää

kehyksien lähetyksen ja vastaanoton. Luokka myös huolehtii sovelluksen sammutuksesta. Ohjelmassa 2 on esitelty AutomationSimulator-luokan luonti, alustus ja käynnistys.

```
if __name__ == '__main__':
    try:
        #AutomationSimulator-olion luonti
        simulator = AutomationSimulator()
        #Alustus
        simulator._initialize()
        #Käynnistys
        simulator._run()

    #Sovelluksen sammutus CTRL+C näppäinyhdistelmällä
    except KeyboardInterrupt:
        pass

    finally:
        #Simulaattorin sammutus
        simulator._exit_gracefully()
```

***Ohjelma 2.*** *AutomationSimulator-luokan käynnistys ja alustus.*

Sovelluksen suoritus tehdään _run-metodissa. Metodi käynnistää kehysten vastaanoton, ajaa kehysten lähetystä while-loopissa ja asettaa jokaisen lähetyksen jälkeen sovelluksen suorittavan säikeen sleep-tilaan. Sovelluksen suoritusta ajavaa sovelluskoodia on havainnollistettu Ohjelmassa 3.

```
def _run(self):
        print("running..")
        #Käynnistä vastaanotto
        self._receiver.start()

        while True:
            #Lähetä kehys
            self._sender.send()

            #Aseta lähetystä suorittava säie sleep-tilaan
            time.sleep(self._config.send_interval_seconds)
```

***Ohjelma 3.*** *Sovelluksen suoritus.*

**AutomationEthernetFrameSender**

Tämä luokka vastaa kehysten lähetyksestä. Se myös valitsee käytettävän verkkokortin, muuttaa käyttäjän syötteen ja konfiguraatiossa olevan vastaanottajan MAC-osoitteen

socket-kirjaston vaatimaan muotoon. Tämä luokka myös tulostaa valitun tyyppisen kehyksen lähetyksen.

### AutomationEthernetFrameReceiver

Tämän luokan tehtävänä on huolehtia kehysten vastaanottamisesta. Vastaanottaja ajaa kehysten vastaanottoa rinnakkain omassa säikeessä, jotta kehyksen lähetyksestä ja vastaanotosta ei muodostuisi odotustilannetta, jossa lähettäjä odottaa vastaanottajaa ja päinvastoin. Sammutuksen yhteydessä tulee myös huolehtia säieturvallisesta suorituksesta, mikä tehdään sovelluksessa kooditason lukon, sekä sammutusbitin avulla. Vastaanoton alustusta ja rinnakkaistusta on havainnollistettu ohjelmassa 4.

```
   class AutomationEthernetFrameReceiver(ComponentBase):
       def __init__(self, configuration):
           super().__init__(configuration)

           #Kooditason lukko sovelluksen sammutusta varten.
           self._lock = Lock()

           #Sammutusbitti
           self._cancelled = False

           #Aseta _poll_receiving-metodi omaan säikeeseen.
           self._receiverThread=Thread(target=self._poll_receiving,
name= "receiver_thread")
```

***Ohjelma 4.*** *Vastaanoton alustus.*

Kehyksen vastaanottamisen yhteydessä myös tulostetaan terminaaliin vastaanotettu kehys, kehyksen tyyppikentän tulkittu arvo, kehyksessä olevat lähettäjän ja vastaanottajan MAC-osoitteet sekä kehyksen datakentän sisältö. Koska Ethernet-verkkokortti vastaanottaa myös muun tyyppisiä kehyksiä, kuin automaatiosimulaattorin tukemia, suodattaa vastaanottaja ennen tulostusta pois muut kuin sovelluksen tukemat kehystyypit. Mikäli suodatus otetaan pois, voidaan vastaanottajalla tulostaa kaikki verkkokortin vastaanottamat Ethernet-kehykset.

# 5. YHTEENVETO

Työn tarkoituksena oli toteuttaa automaatioverkoissa käytettäviä Ethernet-kehyksiä välittävä sovellus sekä tutkia Ethernetin toimintaperiaatetta ja vertailla automaatioverkkojen erityispiirteitä Ethernetin toimintaperiaatteeseen. Työssä myös kuvattiin tapoja luokitella automaatioverkkojen Ethernet-pohjaisia toteutustapoja sekä automaatioverkkojen reaaliaikavaatimuksia.

Työn käytännön osassa toteutettiin itse automaatioliikennesimulaattori, jonka avulla on mahdollista välittää automaatioverkossa käytettäviä Ethernet-kehyksiä kahden prosessin välillä samassa verkon segmentissä. Sovelluksella voidaan simuloida kahden automaatioverkon solmun välistä liikennettä ilman automaatioverkoille spesifistä laitteistoa. Sovellusta voidaan hyödyntää jatkossa esimerkiksi automaatioverkkojen tietoturvatestauksessa ja tutkimuksessa, ja sovellus on mahdollista käynnistää Ethernet-verkossa pelkästään konfiguraatiota muuttamalla.

Teoriaosassa haasteellista oli löytää sopiva näkökulma itse automaatioliikennesimulaattorin ympärille. Automaatioliikennesimulaattorin toteutuksessa haasteita aiheutti se, että toteutuksessa käytetyt teknologiat eivät olleet entisestään työn tekijälle kovin tuttuja. Käytännön osassa käytetyt python-ohjelmointikieli sekä VirtualBox-virtualisointialusta osoittautuivat erittäin käyttökelpoiseksi tähän työhön. Haasteellista oli myös saada sovellus toimimaan sekä vastaanottajana että lähettäjänä samaan aikaan, mikä ratkaistiin säikeistystä käyttäen.

Kokonaisuutena työ onnistui hyvin. Itse automaatioliikennesimulaattoria on mahdollista laajentaa pelkästään konfiguroimalla kattamaan myös automaatioverkkojen ulkopuolisten Ethernet-kehysten lähetys ja vastaanotto.

# LÄHTEET


[1] J. Kurose, K. Ross, C, Computer Networking – A Top-Down Approach, Sixth Edition, Pearson, 2006, 862p.

[2] W. Goralski, The Illustrated Network: How TCP/IP Works in a Modern Network, Morgan Kauffman, 2009, 899 p.

[3] W. Stallings, Data and Computer Communications, Eight Edition, Pearson Prentice Hall, 2007, 878 p.

[4] B. Wilanowski, J Irwin, Industrial Communication Systems, Second Edition, CRC Press, 2011, 963 p.

[5] R. Zurawski, Industrial Communication Technology Handbook, Second Edition, CRS Press, 2014, 1757 p.

[6] B. Galloway, G.P. Hancke, Introduction to Industrial Control Networks, IEEE Communications Surveys & Tutorials, Vol. 15, 2013, pp. 860–880.

[7] S. Siewert, Real-time resource management. Saatavissa (viitattu 03.05.2021): http://mercury.pr.erau.edu/~siewerts/extra/papers/IBM-out-of-print/soc-4.pdf

[8] A. Kaurto, Ethernet-pohjaisten automaatioverkkojen reaaliaikainen kunnonvalvonta, diplomityö, 2018. Saatavissa (viitattu 03.05.2021): https://trepo.tuni.fi/bitstream/handle/123456789/26818/Kaurto.pdf?sequence=4&isAllowed=y

[9] W. Jia, W. Zhou, Distributed Network Systems, from Concepts to Implementation, Springer, 2005, 542 p.

[10] What is Kali Linux. Saatavissa (viitattu 1.11.2020): https://www.kali.org/docs/introduction/what-is-kali-linux/

# LIITE A: SIMULATOR_CONFIGURATION.JSON


```
{
  "SENDER_MAC_ADDRESS" : "08:00:27:91:c9:9d",
  "RECEIVER_MAC_ADDRESS" : "08:00:27:21:2b:0b",
  "LOGGING_MODE" : "normal",
  "SEND_INTERVAL_SECONDS" : "5"
}
```

# LIITE B: AUTOMATION_ETHERNET_SIMULATOR.PY

```
import json
import constants
from configuration_loader import ConfigurationLoader
from user_input_dispatcher import UserInputDispatcher
import data_conversion_helper
import time
from sender import AutomationEthernetFrameSender
from receiver import AutomationEthernetFrameReceiver

class AutomationSimulator:
    '''
    Main class for application. Loads configuration, creates receiver
                and sender parties and is responsible for shutdown
    '''
    def __init__(self):
        self._sender = None
        self._receiver = None
        self._config = None
        self._selected_frame_type = None

    def _initialize(self):
        '''
        Simulator initialization required to run application
        '''
        print("Starting server.")

        #Load configuration
        self._config = ConfigurationLoader.load()

        #print loaded configuration
        self._print_configuration()

        #Request user input for wanted automation Ethernet type
        self._selected_frame_type = UserInputDispatcher.request_ether-
                net_frame_type_input()

        #Create object responsible for sending frames
        self._sender    =    AutomationEthernetFrameSender(self._config,
                self._selected_frame_type)

        #Create object responsible for receiving frames
        self._receiver = AutomationEthernetFrameReceiver(self._config)


    def _print_configuration(self):
        print("SENDER MAC: {}".format(self._config.sender_mac))
        print("RECEIVER MAC: {}".format(self._config.receiver_mac))
        print('LOGGING LEVEL: {}'.format(self._config.logging_mode))

    def _run(self):
        print("running..")
        self._receiver.start()
```

```
        while True:
            #Send selected frame
            self._sender.send()

            #Wait for configured interval
            time.sleep(self._config.send_interval_seconds)

    def _exit_gracefully(self):
        print("exiting gracefully...")

        #Stop receiver
        self._receiver.cancel()
        print('Application stopped')

if __name__ == '__main__':
    try:
        simulator = AutomationSimulator()
        simulator._initialize()
        simulator._run()

    #Enable stopping application with CTRL+C
    except KeyboardInterrupt:
        pass

    finally:
        simulator._exit_gracefully()
```

# LIITE C: COMPONENT_BASE.PY

```
import constants
from configuration_loader import AutomationSimulatorConfiguration

class ComponentBase:
    def __init__(self, configuration):
        self._configuration = configuration

    @property
    def configuration(self):
        return self._configuration

    def log_only_extended(self, message):
        if(self.configuration.logging_mode==constants.LOG-
                GING_LEVEL_EXTENDED):
            print(message)

    def log_all_levels(self, message):
```

# LIITE D: CONFIGURATION_LOADER.PY

```
import json
import constants

class AutomationSimulatorConfiguration:
    '''
    Data class for application configuration
    '''
    def __init__(self, sender_mac, receiver_mac, logging_mode, send_in-
terval_seconds):
        self.sender_mac = sender_mac
        self.receiver_mac = receiver_mac
        self.logging_mode = logging_mode
        self.send_interval_seconds = send_interval_seconds

class ConfigurationLoader:

    @staticmethod
    def load():
        '''
        Loads application configuration from simulator_configura-
tion.json in application root.
        '''
        data = []

        with open('simulator_configuration.json') as config_file:
            data = json.load(config_file)

        sender_mac = data[constants.SENDER_MAC_CONFIGURATION_KEY]
        receiver_mac = data[constants.RECEIVER_MAC_CONFIGURATION_KEY]
        logging_mode = data[constants.LOGGING_MODE_CONFIGURATION_KEY]
        send_interval_seconds = data[constants.SEND_INTERVAL_SEC-
ONDS_CONFIGURATION_KEY]

        return AutomationSimulatorConfiguration(sender_mac, re-
ceiver_mac, logging_mode, int(send_interval_seconds))
```

# LIITE E: CONSTANTS.PY

```
from ethernet_type_data import EthernetTypeData

SENDER_MAC_CONFIGURATION_KEY = "SENDER_MAC_ADDRESS"
RECEIVER_MAC_CONFIGURATION_KEY = "RECEIVER_MAC_ADDRESS"
LOGGING_MODE_CONFIGURATION_KEY = "LOGGING_MODE"
LOGGING_LEVEL_NORMAL = "normal"
LOGGING_LEVEL_EXTENDED = "extended"
SEND_INTERVAL_SECONDS_CONFIGURATION_KEY = "SEND_INTERVAL_SECONDS"

ETHERNET_POWERLINK_TYPE= '88ab'
TCNET_TYPE = '888b'
EPA_TYPE = '88cb'
PROFINET_TYPE = '8892'
ETHERCAT_TYPE = '88a4'
SERCOSIII_TYPE = '88cd'


POWERLINK_DATA = EthernetTypeData(0, "ETHERNET POWERLINK", ETHERNET_POW-
ERLINK_TYPE)
TCNET_DATA = EthernetTypeData(1, "TCNet", TCNET_TYPE)
EPA_DATA = EthernetTypeData(2, "EPA", EPA_TYPE)
PROFINET_DATA = EthernetTypeData(3, "PROFINET", PROFINET_TYPE)
ETHERCAT_DATA = EthernetTypeData(4, "ETHERCAT", ETHERCAT_TYPE)
SERCOSIII_DATA = EthernetTypeData(5, "SERCOSIII", SERCOSIII_TYPE)


accepted_protocols = [ETHERNET_POWERLINK_TYPE, TCNET_TYPE, EPA_TYPE,
PROFINET_TYPE, ETHERCAT_TYPE,
SERCOSIII_TYPE]
```

# LIITE F: DATA_CONVERSION_HELPER.PY

```
import binascii

class DataConversionHelper:
    '''
    Responsible for required conversions for data transmission types
    '''
    @staticmethod
    def convert_mac_to_hexa_bytes(macAdd):
        '''
        Returns the binary data represented by the hexadecimal string.
        E.g: '11:22:33:44:55:66' -> \\x11\\x22\\x33\\x44\\x55\\x66
        '''
        macbytes = binascii.unhexlify(macAdd.replace(':', ''))
        return macbytes

    @staticmethod
    def convert_ethernet_type(configuredType):
        '''
        Converts ethernet frame type char to bytes '88a4' -> b'88a4'
        '''
        return binascii.unhexlify(configuredType)

    @staticmethod
    def convert_mac_to_bytes(macAdd):
        '''
        Converts mac address from char to bytes: '11:22:33:44:55:66' ->
b'112233445566'
        '''
        test = bytes.fromhex(macAdd.replace(':', ''))
        return binascii.hexlify(test)
```

# LIITE G: ETHERNET_TYPE_DATA.PY

```
import constants

class EthernetTypeData:
    '''
    Data class containing necessary ethernet data.
    '''
    def __init__(self, numberInConsole, name, ethernetFrameType):
        self.numberInConsole = numberInConsole
        self.name = name
        self.ethernetFrameType = ethernetFrameType

class EthernetTypeDataProvider:
    '''
    Provides datas from where the user can select ethernet data frame
type to be sent
    '''
    @staticmethod
    def get_ethernet_type_data(inputType):
        '''
        Returns requested EthernetTypeData class based on user selection
        '''
        inputTypeInt = int(inputType)
        for   selectedType   in   EthernetTypeDataProvider.get_all_sup-
ported_type_datas():
            if selectedType.numberInConsole is inputTypeInt:
                return selectedType
        print("Could not find selected type: {0}".format(inputType))
        return None

    @staticmethod
    def get_all_supported_type_datas():
        '''
        Returns all supported ethernet type datas for application
        '''
        return [constants.POWERLINK_DATA,
        constants.TCNET_DATA,
        constants.EPA_DATA,
        constants.PROFINET_DATA,
        constants.ETHERCAT_DATA,
        constants.SERCOSIII_DATA]
```

# LIITE H: RECEIVER.PY

```
import socket
import struct
import binascii
import constants
from user_input_dispatcher import UserInputDispatcher
from data_conversion_helper import DataConversionHelper
from threading import Lock
from threading import Thread
from component_base import ComponentBase

class AutomationEthernetFrameReceiver(ComponentBase):
    '''
    Handles receiving ethernet frames with a polling method ran in a
separate thread
    '''
    def __init__(self, configuration):
        super().__init__(configuration)

        #Lock for cancellation bit. R
        self._lock = Lock()

        #Cancellation bit
        self._cancelled = False

        #Separate thread for receiving
        self._receiverThread   =   Thread(target=self._poll_receiving,
name="receiver_thread")

    # Unpack Ethernet Frame
    @staticmethod
    def _unpack_ethernet_frame(data):
        '''
        Parses required info from received data stream
        '''
        dest_mac, src_mac, proto = struct.unpack('!6s6s2s', data[:14])
        return  binascii.hexlify(dest_mac),  binascii.hexlify(src_mac),
binascii.hexlify(proto), data[14:]

    def _poll_receiving(self):
        '''
        Main method responsible for receiving frames
        '''
        #Poll until _cancel is set
        while not self._is_canceled():

            #Receive next raw ethernet frame from socket
            conn  =  socket.socket(socket.AF_PACKET,  socket.SOCK_RAW,
socket.ntohs(3))

            #Take only first part of tuple returned from method
            raw_data = conn.recvfrom(65536)[0]

            dest_mac,   src_mac,   eth_frame_type,   data   =   self._un-
pack_ethernet_frame(raw_data)

            frame_type = eth_frame_type.decode()
```

```
            #Receiver receives all ethernet frames behing the same 
switch, log only those of interest
            if(frame_type not in constants.accepted_protocols):
                super().log_only_extended('Frame type received: {} not 
in accepted protocols.'.format(frame_type))

            #Do not log frames targeted to other mac addresses that mine
            elif (not self._source_mac_is_different(src_mac)):
                super().log_only_extended('Source mac of frame is the 
same as receiver.')
            else:
                #Parse name from frame type
                name = UserInputDis-
patcher.get_name_for_frame_type(frame_type)

                #Log information we are interested in
                super().log_all_levels('RECEIVED ACCEPTED FRAME TYPE: 
{}:'.format(name))
                super().log_only_extended('FROM MAC ADD: {}'.for-
mat(src_mac))
                super().log_only_extended('TO MAC ADD: {}'.for-
mat(dest_mac))
                super().log_all_levels('DATA RECEIVED: {}\n'.for-
mat(data))


    def _source_mac_is_different(self, src_mac):
        '''
        Determines if my mac address is the same as source mac
        '''
        my_mac = super().configuration.sender_mac
        my_mac_converted = DataConversionHelper.con-
vert_mac_to_bytes(my_mac)
        if (src_mac == my_mac_converted):
            return False

        return True

    def cancel(self):
        '''
        Public method for cancellation of receiver thread
        '''
        #Set _cancelled behind lock since both receiver_thread and sim-
ulator main thread can handle it in the same time
        with self._lock:
            self._cancelled = True
            print('Cancelling receiving...')

        #Wait for receiver thread to handle last polling method execution
        self._receiverThread.join()

    def _is_canceled(self):
        '''
        Accessor for locked _cancelled bit
        '''
        with self._lock:
            return self._cancelled

    def start(self):
        '''
        Public method for starting poller
```

```
        '''
        self._receiverThread.start()
        print('Thread:      {}      started...'.format(self._receiver-
Thread.name))
```

# LIITE I: SENDER.PY

```
import socket
import time
from data_conversion_helper import DataConversionHelper
from user_input_dispatcher import UserInputDispatcher
import constants
from component_base import ComponentBase

class AutomationEthernetFrameSender(ComponentBase):
    def __init__(self, configuration, selected_frame_data):
        super().__init__(configuration)
        self.selected_frame_data = selected_frame_data

    def send(self):
        ETH_P_ALL = 3

        #Name of ethernet port to be used.
        interface = 'eth0'

        #Configured source and destination mac addresses
        src    =    DataConversionHelper.convert_mac_to_hexa_bytes(su-
per().configuration.sender_mac)
        dst    =    DataConversionHelper.convert_mac_to_hexa_bytes(su-
per().configuration.receiver_mac)

        #Get selected frame type ethernet frame type. E.g.: Ethercat ->
88a4
        frame_type        =        DataConversionHelper.convert_ether-
net_type(self.selected_frame_data.ethernetFrameType)
        payload = 'Ethernet sockets are fun with python, greetings from:
{}'.format(super().configuration.sender_mac).encode()

        #Use python socket to send ethernet frame
        s    =    socket.socket(socket.AF_PACKET,    socket.SOCK_RAW,
socket.htons(ETH_P_ALL))
        s.bind((interface, 0))
        s.sendall(dst + src + frame_type + payload)
        s.close()

        super().log_all_levels('Frame   of   type:   {}   sent\n'.for-
mat(self.selected_frame_data.name))
```

# LIITE J: USER_INPUT_DISPATCHER.PY

```
from ethernet_type_data import EthernetTypeData, EthernetTypeDataPro-
vider

class UserInputDispatcher:

    @staticmethod
    def get_applicable_types_string():
        '''
        Returns applicable ethernet frame data in string format for user
input
        '''
        all_supported_types = EthernetTypeDataProvider.get_all_sup-
ported_type_datas()
        all_supported_types.sort(key=lambda x: x.numberInConsole)

        message = "Select ethernet frame type to be sent:\n"
        for x in all_supported_types:
            message = message + "[{0}] {1}\n".format(x.numberInConsole,
x.name)

        return message

    @staticmethod
    def request_ethernet_frame_type_input():
        '''
        Prints applicable types for application and reads user input for
selected ethernet frame type
        '''
        print("Select ethernet frame to be used:")
        print(UserInputDispatcher.get_applicable_types_string())
        selected_frame_input = input()
        print("read input")
        selected_frame_data = EthernetTypeDataProvider.get_ether-
net_type_data(selected_frame_input)
        print("selected frame was:" + selected_frame_data.name)
        return selected_frame_data

    @staticmethod
    def get_name_for_frame_type(frameType):
        all_supported_types = EthernetTypeDataProvider.get_all_sup-
ported_type_datas()
        typeDatas = list(filter(lambda x: x.ethernetFrameType == frame-
Type, all_supported_types))
        typeData = typeDatas[0]
        if( typeData is None):
            print('Could not find frame type: {} from supported frame
types.'.format(frameType))
            return None
    return typeData.name
```