TOMMI TUUSA

ROS-alusta ja potentiaalinen käyttö Cimcorpilla

SÄHKÖ- JA AUTOMAATIOTEKNIIKAN KOULUTUSOH-
JELMA
2021

Tekijä(t) Tuusa, Tommi	Julkaisun laji Opinnäytetyö, AMK	Päivämäärä Lokakuu 2021
	Sivumäärä 35	Julkaisun kieli Suomi

Julkaisun nimi
ROS-alusta ja potentiaalinen käyttö Cimcorpilla

Tutkinto-ohjelma
Sähkö- ja automaatiotekniikka

Tämän opinnäytetyön toimeksiantajana toimi Cimcorp Oy, joka on Ulvilassa pääkonttoriaan pitävä automaatioalan yritys. Työn tavoitteena oli tutustua ROS-alustaan, kirjoittaa kuvaus sen toiminnasta ja rakentaa demoluonteinen sovellus sen käytöstä Cimcorpin liikkeenohjauksen kontekstissa.

Opinnäytetyön teko alkoi ROS-alustaan tutustumisella ja kuvauksen kirjoittamisella. Tutustuminen alustaan tapahtui pääasiassa SAMK:n järjestämän täydennyskoulutuskurssituksen, sekä ROS-projektin kotisivujen tarjoaman materiaalin kautta.

Sovelluksen aihe rajattiin ROS-alustan käyttöön yhdessä Bosch Rexrothin uuden ctrlX-automaatioalustan ohjelmistoytimen ctrlx COREn kanssa. CtrlX COREen tutustuttiin pääasiassa Rexrothin tarjoaman virtuaalisen ctrlX COREn, sekä ctrlX community -sivuston kautta löytyneiden materiaalien avulla. Olennaisia tutustumisen kohteita sovellusta rakennettaessa olivat Snapcraft, joka on ctrlX-alustalle asennettavien sovellusten paketointiin käytetty työkalu, sekä Gazebo, joka on ROS-alustan yhteydessä yleisesti käytetty 3D-fysiikkasimulointityökalu. Näihin molempiin tutustuttiin lähinnä ohjelmistojen omien kotisivujen kautta.

Työssä saatiin aikaan yleinen selvitys ROS-alustasta, jossa käydään läpi ROS1-version rakennetta ja toimintaa, sekä lyhyemmin ROS2 ja esimerkkisovelluksen rakentamisen yhteydessä käytettyjä ohjelmistoja ja alustoja, joihin kuuluivat ctrlX, snapcraft ja Gazebo.

Esimerkkisovelluksena rakennettiin ctrlX CORElle asennettavissa oleva ROS1-snap-sovellus, jolla voidaan julkaista ctrlX COREn datakerroksen tietoja myös COREn ulkopuolelta saatavilla olevaan ROS topic:iin, sekä erillinen Gazebo-simulaatio, joka käyttää näitä ctrlX CORElta julkaistuja tietoja simuloidun gantry-robotin kahden akselin liikuttamiseen.

Asiasanat
ohjelmointi, simulointi, automaatio, väliohjelmistot

Author(s) Tuusa, Tommi	Type of Publication Bachelor's thesis	Date October 2021
	Number of pages 35	Language of publication: Finnish

Title of publication

The ROS platform and its potential use at Cimcorp

Degree program
Electric- and Automation Engineering

This thesis was comissioned by Cimcorp Oy, which is an automation company headquartered at Ulvila. The aim of this thesis was to get to know the ROS platform, write a description about its functionality and to build a demo application about using the platform in the context of Cimcorp motion controls.

The work began by getting to know the ROS platform and writing the description. Studying the platform was mainly done through SAMKs continuing professional development courses and by materials available from the homepage of the ROS project.

The subject of the application was specified to using the ROS platform in conjunction with ctrlX CORE, which is the software core of Bosch Rexroth's new ctrlX automation platform. Getting to know ctrlX CORE was done with a virtual ctrlX CORE offered by Bosch Rexroth and through materials found from the ctrlX community site. Other relevant subjects of study related to building the application were Snapcraft, which is a software tool used in packaging applications for the ctrlX CORE, and Gazebo, which is a 3D physics simulation tool commonly used with ROS. Both Snapcraft and Gazebo were mainly researched through the materials offered by their respective home sites.

As a result, a general report about the ROS platform was produced. The report focused mainly on the structure and functionality of the ROS1 version and introduced, in lesser detail, the ROS2 version and the software tools and platforms used in building the application. These included ctrlX, snapcraft and Gazebo.

As the demonstrational application, a ROS1 application was built that could be installed on the ctrlX CORE and which would then publish data from the ctrlX CORE datalayer to a ROS topic also available from outside of the CORE. An external Gazebo-simulation was also built to move two axes of a simulated gantry robot based on the published data.

Key words
programming, simulation, automation, middleware

SISÄLLYS

1 JOHDANTO 7
2 MIKÄ ON ROS? 8
2.1 ROS:n lyhyt historia 8
3 ROS1 9
3.1 Käyttöönotto 9
3.1.1 ROS-ympäristömuuttujat 10
3.2 Catkin ja ROS1-projektin tiedostorakenne 10
3.2.1 Catkin-työtila 10
3.2.2 Paketit (*ROS packages*) 11
3.3 Ohjelmistorakenne 14
3.3.1 Roscore ja ROS master 14
3.3.2 Noodit (*ROS node*) 14
3.3.3 Launch-tiedostot 15
3.3.4 ROS-viestit (*Messages*) 15
3.4 Viestiarkkitehtuuri 16
3.4.1 Aiheet (*ROS topic*) 16
3.4.2 Palvelut (*service)* 17
3.4.3 Actions 18
3.4.4 ROS Bags 18
3.4.5 Parametrit 19
4 ROS 2 20
4.1 Viestiarkkitehtuuri 20
4.2 RCL Ros Client Library 21
4.3 Parametrit 21
5 GAZEBO 22
5.1 Määrittelytiedostot 22
6 CTRLX 23
6.1 ctrlX CORE 23
7 SNAPCRAFT 24
7.1 Käyttö lyhyesti 24
8 KÄYTÄNNÖN OSUUS 26
8.1 Työn aloitus 26
8.2 ROS-versio 27
8.3 ROS1-sovellus 27
8.4 Sovelluksen paketointi 28

8.5 ROS-Gazebo -simulaatiosovellus 29
8.5.1 Simulaation määrittely 30
8.5.2 Toiminta 30
8.5.3 Ulkoasu 31
9 TULOKSET JA PÄÄTELMÄT 33
9.1 Ongelmia 33
9.2 Jatkokehitys 34
LÄHTEET
LIITTEET

SYMBOLI- JA LYHENNELUETTELO

ROS	*Robot Operating System*. Robotiikkasovellusten pohjaksi tarkoitettu metakäyttöjärjestelmä.
TCP	*Transmission Control Protocol*. Luotettava, yhteydellinen tietoliikenneprotokolla tietokoneiden väliseen tiedonsiirtoon. Sisältää pakettien kuittauksen ja menetettyjen pakettien uudelleenlähetyksen
UDP	*User Datagram Protocol*. TCP-protokollaa kevyempi ja epävarmempi, yhteydetön tietoliikenneprotokolla. Ei sisällä pakettien kuittauksia tai uudelleenlähetystä.
DDS	*Data Distribution Service*. Object Management Groupin kehittämä standardi tietokoneiden väliseen reaaliaikaiseen tiedonsiirtoon, joka perustuu publisher-subscriber -viestintämalliin.
ctrlX	Bosch Rexrothin kehittämä automaatioalusta.
ctrlX CORE	ctrlx-alustan Ubuntu core18 -pohjainen ohjelmistoydin.
Gazebo	3D-fysiikkasimulaatio -ohjelmisto
SDK	*Software Development Kit.* Ohjelmistokehityspaketti. Kokoelma ohjelmistokehitystyökaluja yhdessä asennettavassa paketissa.
Catkin	CMakeen perustuva ROS1-projektien rakennustyökalu.
CMake	Avoimen lähdekoodin alustariippumaton työkalukokoelma ohjelmistojen rakentamiseen, testaamiseen ja paketointiin (https://cmake.org/).
EOL	*End Of Life.* Ajankohta, johon ohjelmiston tai ohjelmistoversion virallinen kehitys ja mahdolliset tukipalvelut loppuvat.
LTS	*Long Term Support.* Ohjelmiston LTS-versio on sen tavallista pidempään ylläpidetty ja tuettu vakaa versio.
Snap	Usealle eri linux-alustalle asennettavissa oleva sovelluspakettityyppi.

1 JOHDANTO

Automaatiojärjestelmien eri osien ja korkeamman tason hallintaohjelmistojen saattaminen yhteen toimivaksi, keskenään kommunikoivaksi kokonaisuudeksi on aina enemmän tai vähemmän haastavaa. Erityyppisillä laitteilla, ja toisaalta samankin tyyppisillä, mutta eri laitevalmistajien tuottamilla laitteilla, on usein toisistaan poikkeavia rajapintoja ja kommunikointitapoja, joiden sovittaminen vaihtamaan tietoja keskenään on edellä mainitun haasteellisuuden suuri osatekijä. Erityisesti robotiikkasovellusten kontekstissa erilaisten alijärjestelmien ja rajapintojen kirjo voi kasvaa suureksi, mikä on johtanut tarpeeseen luoda yleiskäyttöisiä sovellusalustoja, jotka tarjoavat valmiita ratkaisuja erilaisten laitteiden liittämiseen yhdeksi kokonaisuudeksi. Yksi tällainen sovellusalusta on ROS (*Robot Operating System*).

Opinnäytetyön tilaaja on Cimcorp Oy, joka on automatisoidun materiaalinhallinnan ratkaisuihin keskittyvä satakuntalainen yritys ja ratkoo omissa sovelluksissaan osittain juuri niitä haasteita, joihin vastaamiseen ROS on suunniteltu. Cimcorp oli kiinnostunut ROS-alustasta, sekä sen potentiaalisista käyttömahdollisuuksista omissa liikkeenohjaussovelluksissaan ja tarjosi aihetta opinnäytetyöksi.

Tämän opinnäytetyön tarkoituksena oli kirjoittaa kuvaus ROS-alustasta, sekä tuottaa esimerkinomainen sovellus jostakin sen potentiaalisesta käyttökohteesta Cimcorpilla. Sovelluksen lopullinen aihe tarkentui ROS1:n käytön testaamiseen Rexrothin uuden ctrlX-automaatioalustan yhteydessä. Sovelluksen tarkoituksena oli toimia tiedonvälittäjänä ctrlX COREnn datakerroksen ja erillisellä laitteella olevan Gazebo-simulaation välillä. Tähän tarkoitukseen piti rakentaa käyttöön soveltuva ROS-sovellus, paketoida se snap-muotoon ja asentaa ctrlX CORElle, sekä rakentaa erillinen ROS-paketti, jossa määritellään kyseinen Gazebo-simulaatio, sekä ROSn ja simulaation väliseen tiedonvaihtoon tarvittava Gazebo-ROS -rajapinta. Ensimmäisenä oli tarkoitus saada aikaan ei-reaaliaikainen tiedonsiirto ja tutkia sen jälkeen reaaliaikaisen tiedonsiirron mahdollisuuksia ja/tai vastaavan systeemin toteutusta ROS2-versiolla.

2 MIKÄ ON ROS?

ROS on robotiikkasovellusten pohjaksi tarkoitettu avoimen lähdekoodin (BSD-lisenssi) ohjelmisto/metakäyttöjärjestelmä, joka koostuu TCP/IP-protokollaan perustuvasta viestinvälitysarkkitehtuurista ja kokoelmasta ohjelmistokirjastoja. ROS:n perimmäinen tarkoitus on toimia alustana, jonka kautta on mahdollista liittää suhteellisen helposti yhteen useita erilaisia rajapintoja omaavia ja eri valmistajien tuottamia robotiikkakomponentteja kuten sensoreita ja toimilaitteita, sekä tarjota valmiina robotiikkasovellusten yleisesti tarvitsemia ohjelmistokomponentteja kuten sensori- ja toimilaiterajapintoja, sekä yleisesti käytettyjä algoritmeja. ROS:n kantavana ajatuksena on valmiin viestiarkkitehtuurin tarjoamisen ohella toimia tutkimuslaitosten, sekä muiden ROS-käyttäjien yhteisenä ohjelmistovarantona, joka ehkäisee kehittäjien tarvetta ”keksiä pyörää uudelleen”, kun kerran kehitetty algoritmi tai rajapinta on saatavilla myös muille ROS-käyttäjille. Ohessa Puluroboticsin jokseenkin osuva luonnehdinta aiheesta:

> ROS is there to integrate the multitude of existing sensor, actuator, etc. hardware, often hard to use and incompatible as is, by converting their datastreams into a message bus, with compatible datatypes between the hardware drivers and calculation units. It's also a set of conversion interfaces to run several external (i.e., not developed for ROS) open source computation algorithms. (Oja, 2017.)

2.1 ROS:n lyhyt historia

ROS-alustan kehityksen voidaan katsoa alkaneen Stanfordin yliopistossa Eric Bergerin and Keenan Wyrobekin opiskelijaprojektina nimeltä ”Stanford Personal Robotics Program”. Projektin tavoitteena oli rakentaa yleiskäyttöinen robottisovellusalusta, jonka avulla voitaisiin välttää samojen perusasioiden toistuvaa rakentamista robotiikkaprojektista toiseen. Varsinainen ROS syntyi 2007 kun Bergerin ja Wyrobekin siirtyivät töihin Willow Garageen, joka alkoi rahoittaa projektia. Alustan kehitystä jatkettiin siellä vuoteen 2013 saakka, jolloin yritys lopetti toimintansa ja siitä eteenpäin ROS:n kehitys on edennyt Open Source Robotics Foundationin alaisuudessa, joka on itsekin Willow Garagesta irtautunut voittoa tavoittelematon yhdistys. (Tellez, 2019; Ackerman, 2012).

3 ROS1

ROS-alustasta on tämän työn kirjoitushetkellä olemassa kaksi toisistaan poikkeavaa versiota; ROS1 ja ROS2 (Open Source Robotics Foundation, 2021c). Näistä ROS1 on virallisesti rakennettu käytettäväksi Linux Ubuntu -käyttöjärjestelmän päällä, mutta tukee jakeluversiosta riippuen myös muita alustoja vaihtelevissa määrin (Foote & Conley, 2010). Muiden alustojen tuki on ROS-yhteisön tuottamaa lisäarvoa eikä siis virallisesti ylläpidettyä tai testattua.

ROS1:stä on tämän kirjoitushetkellä olemassa kaksi tuettua LTS-versiota, koodinimiltään ’Melodic Morenia’ ja ’Noetic Ninjemy’ (Open Source Robotics Foundation, 2021e). Näistä uudempi on ’ROS Noetic Ninjemy’ ja se tulee olemaan myös ROS1:n viimeinen virallinen LTS-julkaisu. Noetic Ninjemyn virallinen EOL on asetettu toukokuuhun 2025, minkä jälkeen ROS-alustan virallinen kehitystyö jatkuu puhtaasti ROS2-version parissa (Open Source Robotics Foundation, 2021d).

Yhteensopivuuksien johdosta tämän työn käytännön osiossa käytetty ROS1-versio on toiseksi uusin LTS-versio ’Melodic Morenia’ ja myös selvitysosa viittaa tähän versioon. Käytöltään ’Melodic Morenia’ ei olennaisesti poikkea uusimmasta ’Noetic Ninjemy’ -versiosta ja suurin osa tästä työstä, snapcraft-osuutta lukuun ottamatta, pitäisi olla suoraan sovellettavissa myös uusimpaan versioon.

3.1 Käyttöönotto

Sen lisäksi, että ROS1-alustan ainoa virallisesti tukema käyttöjärjestelmä on Ubuntu Linux, on ROS1:n jokainen versio sidottu tiettyyn Ubuntu-versioon. Tämän kirjoitushetkellä ROS1:n suositellut versiot ovat ’Noetic Ninjemy’ Ubuntu 20.04 -alustalle, sekä ’Melodic Morenia’ Ubuntu 18.04 -alustalle. Kaikki mainitut ROS-versiot asennusohjeineen ovat saatavilla ROS-projektin kotisivuilta (https://www.ros.org/) ja tässä työssä on esimerkin vuoksi liitteenä yksinkertaistetut ROS ”Melodic Morenian” asennusohjeet Ubuntu 18.04:lle (Liite 1).

3.1.1 ROS-ympäristömuuttujat

ROS vaatii toimiakseen tiettyjen ympäristömuuttujien asettamisen. Nämä muuttujat tulee muistaa ladata erikseen jokaiseen uuteen bash-istuntoon, sillä ne eivät tule asetetuiksi ROS-asennuksen yhteydessä. Tarvittavien muuttujien asettaminen tapahtuu (asennuksen jälkeen) helpoiten komennolla:

```
$ source /opt/ros/melodic/setup.bash
```

Jatkuvassa käytössä edellä mainittu komento on käytännöllistä lisätä suoraan ’.bashrc’ -tiedostoon, jonka kautta se tulee ajetuksi automaattisesti aina uuden bash-komentoikkunan käynnistyksen yhteydessä.

3.2 Catkin ja ROS1-projektin tiedostorakenne

Koska ROS-projektit koostuvat yleensä useista erillisistä ja mahdollisesti myös eri kielillä kirjoitetuista ohjelmistopaketeista, voivat ROS:n projektirakenne ja pakettien keskinäiset riippuvuudet muodostaa lopulta hyvinkin monimutkaisen kokonaisuuden. Tästä syystä ROS-alustalle on rakennettu erillinen rakennustyökalu, ’catkin’, helpottamaan projektien kääntämistä.

Catkin on uudemmissa ROS1-versioissa, kuten tässä työssä käytetyssä Melodic Moreniassa, sisällytetty suoraan asennuspakettiin, eikä sen asennus näin ollen yleensä vaadi käyttäjältä mitään erillisiä toimenpiteitä. Catkin itsessään on yhdistelmä CMake-makroja, sekä python-scriptejä ja se on korvannut aiemman ’rosbuild’-työkalun ROS1-alustan virallisena kääntötyökaluna. (Open Source Robotics Foundation, 2020a.)

3.2.1 Catkin-työtila

Koska ROS1 käyttää projektien kääntämiseen catkin-komentorivityökaluja, on jokaisen ROS1-projektin tiedostorakenteen ylimmällä tasolla käytännössä aina catkin-työtila, joka sisältää kaikki yksittäisen ROS-projektin tiedostot. Myös uusien pakettien

luonti ja projektin muu kehitys tapahtuu tämän työtilan sisässä. (Open Source Robotics Foundation, 2014a.)

Uutta työtilaa varten pitää ensin luoda sen tarvitsema perustiedostorakenne, joka koostuu työtilan juurihakemistosta, sekä sen sisältämästä src-hakemistosta. Kun nämä perushakemistot on luotu, voidaan työtila alustaa käyttöön juurihakemistossa annettavalla ’catkin_make’ -komennolla. Perushakemistojen luonti ja työtilan alustus onnistuvat esimerkiksi seuraavalla komentosarjalla:

```
$ mkdir -p catkin_workspace/src
$ cd catkin_workspace
$ catkin_make
```

Catkin_make generoi alustuksen yhteydessä juurihakemistoon uudet build- ja devel-hakemistot. Build-hakemistoa käyttävät ROS-kääntötyökalut ROS-pakettien rakentamisen yhteydessä ja devel on käännettyjen ROS-pakettien välisijainti, jonka kautta paketteja voidaan ajaa ja testata ennen asennusta. (Open Source Robotics Foundation, 2017.)

3.2.2 Paketit (*ROS packages*)

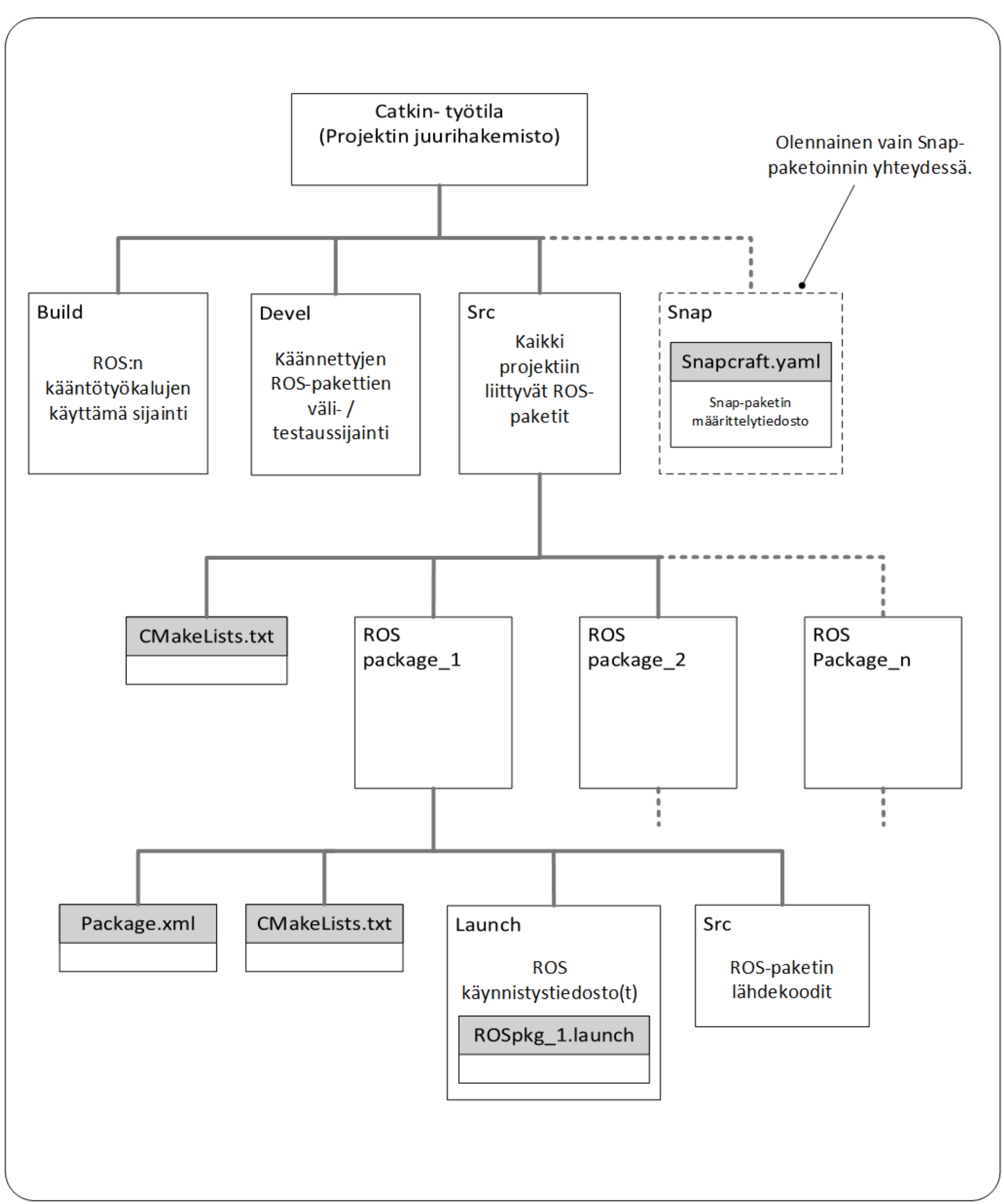
ROS-alustan ohjelmistokokonaisuudet jaetaan paketteihin. Paketit on tarkoitus pyrkiä pitämään loogisina kokonaisuuksina ja yksittäisen paketin sisältönä voi olla esimerkiksi kokoelma ROS-noodeja, jotka toteuttavat jonkin tietyn toiminnallisuuden, itsenäinen ROS-kirjasto tai vaikka data-aineisto. Kantavana ajatuksena on ohjelmistojen jakaminen modulaarisiin, helposti jaeltaviin ja uudelleenkäytettäviin kokonaisuuksiin. Paketit ovat pienimpiä ROS-projektien jaeltavia osasia.

Projektin ROS-paketit lisätään catkin-työtilan src-hakemistoon ja uusien pakettien luonti tapahtuu catkin_create_pkg -komentorivityökalulla seuraavasti:

```
$ catkin_create_pkg <paketin_nimi> <riippuvuus_1> … <riippuvuus_n>
```

Komento ottaa parametrina luotavalle paketille halutun nimen, sekä sen tarvitsemat riippuvuudet. Catkin_create_pkg luo uudelle paketille automaattisesti tyhjän src-hakemiston sekä CMakeLists.txt- ja package.xml -tiedostot, jotka pitävät sisällään ohjeet paketin tarvitsemista riippuvuuksista sekä rakennetun paketin toimintaan tarvittavien suoritettavien tiedostojen asennusohjeet. Komennon parametreissa annetut niin sanotut ensimmäisen tason riippuvuudet tulevat myös automaattisesti lisätyiksi näihin luotuihin ohjetiedostoihin.

Nimenomaan ROS1:n yhteydessä pitää omia paketteja rakennettaessa kiinnittää huomiota kunkin paketin juuresta löytyvien ’CMakeLists.txt’- sekä ’package.xml’-tiedostojen pitämiseen ajan tasalla, koska mahdolliset uudet, paketin luomisen jälkeen lisätyt, riippuvuudet eivät päivity tiedostoihin automaattisesti projektin kääntämisen yhteydessä. Puutteelliset asennusohjeet johtavat helposti siihen, että omaa pakettia voi (ROS1:ssä) kyllä ajaa aktiivisessa catkin-työtilassa, koska sen vaatimat riippuvuudet löytyvät paikallisesta ympäristöstä, mutta samaa pakettia ei pysty suoraan käyttämään esimerkiksi toisessa ympäristössä olevassa ROS-projektissa tai snap-muotoon paketoituna, koska tiedot tarvittavista riippuvuuksista ja/tai suoritettavien tiedostojen asennusohjeet puuttuvat itse paketista. (Open Source Robotics Foundation, 2019a.) Uudet paketit pitää aina muistaa myös kääntää catkin-työtilan juuressa annettavalla ”catkin_make” -komennolla ennen kuin niitä voidaan käyttää edes omassa kehitysympäristössä.



Kuva 1. ROS- projektin tyypillinen hakemistorakenne

3.3 Ohjelmistorakenne

3.3.1 Roscore ja ROS master

Roscore on kokoelma noodeja ja ohjelmia, jotka ovat esivaatimus ROS1:n noodien keskinäiselle kommunikaatiolle ja koko alustan käytölle. Koska ROS1-sovellukset käynnistetään yleensä launch-tiedostojen kautta ja roscore käynnistyy tässä yhteydessä automaattisesti, ei käyttäjän yleensä tarvitse valmiin sovelluksen yhteydessä huolehtia roscoren toiminnasta tai käynnistyksestä, mutta esimerkiksi yksittäisen noodin toimintaa testattaessa pitää roscore käynnistää erikseen omassa konsoli-ikkunassaan ”roscore” -komennolla ennen kuin testattavaa noodia on mahdollista käynnistää. (Open Source Robotics Foundation, 2019b.)

ROS master (*master node*) on ROS1-järjestelmän ydinelementti, joka pitää kirjaa järjestelmän muista noodeista ja niiden tiedoista, sekä pitää sisällään ROS1:n parametriserverin. Kaikki noodit rekisteröityvät käynnistyessään ROS masteriin, jonka kautta ne myös löytävät toisensa, mikä puolestaan mahdollistaa noodien keskinäisen viestinnän. Käytännössä muiden noodien tietojen hakeminen ja noodien keskinäisen viestinnän alustus tapahtuvat automaattisesti systeemin toimesta ja käyttäjän tulee koodin tasolla huolehtia vain esimerkiksi noodin lisäämisestä jokin tietyn viestiaiheen (*ROS topic*) kuuntelijaksi halutessaan käsitellä viestiaiheeseen julkaistua tietoa noodissaan. ROS master luodaan ja käynnistyy automaattisesti roscoren mukana, eikä käyttäjän tarvitse erikseen huolehtia sen käynnistämisestä tai toiminnasta.

3.3.2 Noodit (*ROS node*)

Aiemmin mainitut ROS-paketit sisältävät yleensä useita noodeja, jotka yhdessä toteuttavat jonkin tietyn toiminnallisuuden. Noodit itsessään ovat yksittäisiä erikseen suoritettavia pikkuohjelmia, joilla on yleensä jokin selkeä ja tarkkaan rajattu tehtävä ROS-sovelluksessa kuten esimerkiksi tietyn sensorin tuottaman datan julkaisu tai jonkin toisen noodin julkaiseman sensoridatan jatkokäsittely. Jokainen noodi on yksilöllisesti nimetty ja noodien välinen kommunikointi tapahtuu näiden yksilötunnisteina toimi-

vien nimien perusteella. Yksittäisen noodin voi esimerkiksi testitarkoituksessa käynnistää rosrun-työkalulla antamalla parametreina noodin sisältävän paketin nimi, sekä itse noodin suoritettava tiedosto:

```
$ rosrun <paketin_nimi> <suoritettavan_nooditiedoston_nimi>
```

Esivaatimuksena yksittäisen noodin käynnistykselle on, että roscore on käynnissä (erillisessä konsoli-ikkunassa), käytettävä paketti on käännetty ja sen ympäristö on aktiivisena ($ source devel/setup.bash).

3.3.3 Launch-tiedostot

Koska laajempien ROS-ohjelmistojen käynnistys noodi kerrallaan ei ole mielekästä tai järkevää, tarjoaa ROS käyttöön roslaunch-työkalun, jolla on mahdollista käynnistää kokonainen ROS-sovellus yhdellä komennolla. Tätä varten työkalun ajettavaksi pitää kirjoittaa niin sanottu launch-tiedosto. Launch-tiedoston ajaminen roslaunch-työkalulla käynnistää automaattisesti roscoren, sekä sen mukana ROS Master -noodin, eli niiden käynnistystä ei kuulu erikseen määritellä tässä yhteydessä. Launch-tiedostossa määritellään kuitenkin kaikki muut noodit, joiden halutaan käynnistyvän heti sovelluksen käynnistyksen yhteydessä, sekä niiden mahdolliset alkuparametrit ja muut ROS-sovelluksen käynnistyksen yhteydessä ladattavaksi halutut elementit. Launch-tiedosto itsessään on xml-tiedosto, johon jokainen käynnistettävä noodi tai muu käynnistyksen yhteydessä ladattava elementti (esimerkiksi tarvittava ympäristömuuttuja, Gazebon simulaatioympäristö tai muu vastaava) määritellään omassa tagissaan. Noodia määritellessä tagin sisään tulee antaa noodin sisältävä paketti, noodin suoritettava tiedosto ja noodin nimi, sekä sen tarvitsemat parametrit. Tästä liitteenä esimerkkinä työn simulaatio-osassa käytetty launch-tiedosto (LIITE 6).

3.3.4 ROS-viestit (*Messages*)

Kaikki ROS-sovelluksen sisäinen kommunikointi tapahtuu eri tyyppisillä ROS-viesteillä, jotka ovat tyypitetyistä kentistä muodostuvia datarakenteita. Kentät voivat olla primitiivisiä datatyyppejä (integer, float, boolean yms.) tai näistä muodostettuja tietueita. Viestit saavat myös sisältää sisäkkäisiä rakenteita.

ROS tarjoaa käyttöön useita valmiiksi määritettyjä viestirakenteita, mutta niitä voi tarvittaessa luoda myös itse. (Open Source Robotics Foundation, 2014a.)

3.4 Viestiarkkitehtuuri

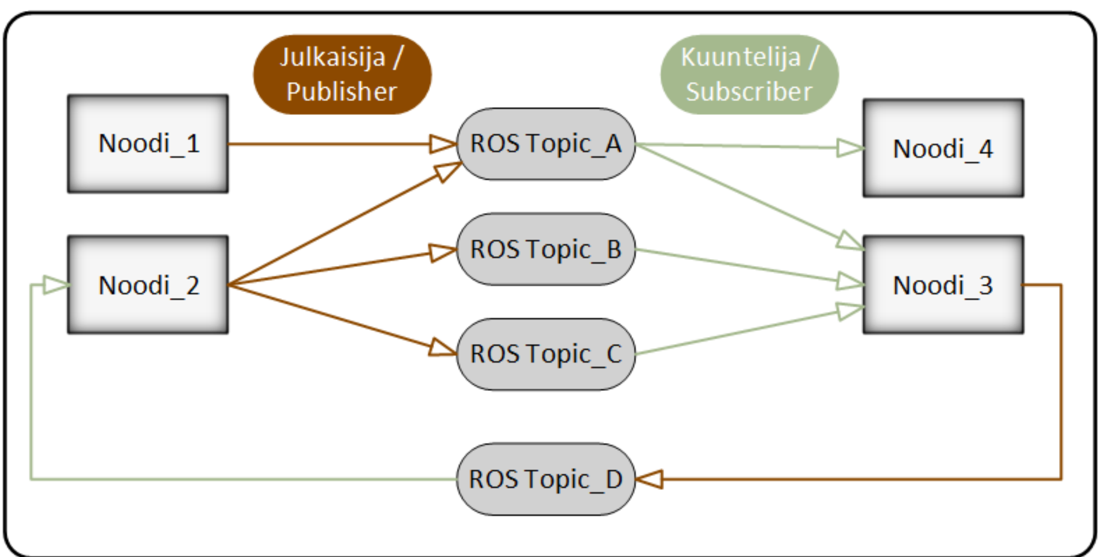
ROS on suunniteltu helposti hajautettavaksi systeemiksi ja kaikki ROS1-ohjelmistokomponenttien viestintä tapahtuu TCP/IP-protokollan kautta. Noodit voivat keskustella keskenään ja ROS Master -noodin kanssa riippumatta siitä ovatko ne käynnissä samalla fyysisellä laitteella niin kauan kuin ROS-sovelluksen eri osia sisältävät laitteet kuuluvat samaan verkkoon, kykenevät kommunikoimaan keskenään ja jokaisella erillisellä osalla on tieto Master-noodin sijainnista. Toisaalta kommunikaatio tapahtuu saman protokollan kautta myös koko systeemi ollessa rakennettuna keskitetysti samalle laitteelle.

3.4.1 Aiheet (*ROS topic*)

Noodien keskinäinen viestintä tapahtuu yleisimmin viestiaiheiden (ROS topic) kautta. Esimerkkinä voisi olla vaikkapa aihe, johon julkaistaan yksittäiseltä sensorilta saatavat tiedot. Tällaiseen aiheeseen liittyisi todennäköisesti vain yksi julkaisijanoodi (*publisher*), joka jatkuvasti julkaisee aiheessa sensorilta tulevaa dataa. Näin edellä mainitun sensorin data on saatavilla kaikille noodeille, jotka on lisätty kyseisen aiheen kuuntelijoiksi (*subscription*).

Aiheet toimivat julkaisija-tilaaja (*publisher-subscriber*) -periaatteella eli yksittäinen noodi voidaan liittää tiettyyn aiheeseen joko kuuntelijan tai julkaisijan ominaisuudessa (tai molemmissa). Jokaisella aiheella on yksilöllinen nimi, minkä perusteella noodien liittäminen tiettyyn aiheeseen tapahtuu. Yhteen aiheeseen voi liittyä useita julkaisija- ja/tai kuuntelija-noodeja (esim. Kuva 2, Topic_A) ja yksittäinen noodi voi sekä julkaista viestejä useampaan eri aiheeseen, että kuunnella useaa erillistä aihetta (esim. Kuva 2, Noodit 2 & 3).

ROS:n aiheiden kautta tapahtuvassa viestinnässä noodit eivät ole varsinaisesti tietoisia toistensa toiminnasta ja kommunikointiin ei liity sanomien vastaanottamisen kuittaamista tai muutakaan viestinvälityksen varmistamista eli esimerkiksi tiettyyn aiheeseen viestejä julkaisevan noodin (*publisher*) toimintaan ei mitenkään vaikuta aiheen kuuntelijoiden määrä tai niiden puute. Tällä saadaan eriytettyä noodit toisistaan, jolloin esimerkiksi tietyn noodin kaatuminen ei suoraan kaada muita noodeja, mutta myöskään viestien menettämisestä tai vääristymisestä matkalla ei saada tietoa.



Kuva 2: Mahdollisia aiheiden (topic) ja noodien välisiä suhteita

3.4.2 Palvelut (*service)*

ROS-palvelu (*service*) on request-reply -tyyppinen kuitattu viestinvälitystapa. ROS-noodien kommunikointi tapahtuu yleisimmin viestiaiheiden kautta, mutta jos yksittäisten viestien menettäminen tai vääristyminen matkalla on sovelluksen toiminnan kannalta erityisen haitallista, tai viestit tulee ehdottomasti saada kuitattua, voidaan käyttää service-viestintätapaa. ROS-palveluja (ROS service) ja -aiheita (ROS topic) voi ideaaltaan suunnilleen verrata TCP- ja UDP-protokolliin.

Serviceen liittyy aina kaksi viestirakennetta, joista toista käytetään palvelun kutsuun ja toista vastaukseen. Palvelut ovat aina nimettyjä ja palveluja kutsuvat noodit käyttä-

vät niitä lähettämällä kutsuviestin nimetylle palvelulle, minkä jälkeen ne jäävät odottamaan vastausta palvelun tarjoavalta noodilta. Koodin tasolla palvelukutsut on ROS-toteutuksissa yleensä toteutettu näyttämään etäproseduurikutsuilta.

3.4.3 Actions

Jos noodin pyytämän palvelun (*ROS service*) suorittaminen on erityisen raskasta ja kestää pitkän aikaa, tällä saattaa olla haittavaikutuksia myös muiden noodien toimintaan. Tätä varten on kehitetty actionit, joilla voidaan toteuttaa pidempiä suoritusaikoja vaativia toimintoja asynkronisesti hidastamatta muuta suoritusta. ROS actions ei ole osa ROS1:n perusasennuspakettia eli se pitää haluttaessa lisätä erillisen actionlib-paketin kautta.

Kommunikaatiopaketin toiminta perustuu client-server -tyyppiseen toteutukseen, missä actions käyttää ROS viestiaiheita lähettääkseen tavoitteita (*goals)* action-clientilla action-serverille. Tavoite voi olla esimerkiksi robotin siirtäminen uuteen sijaintiin tai muu suhteellisen pitkäkestoinen suoritus, jota serveri ryhtyy suorittamaan tavoitteen saatuaan. Olennainen ajatus on, että clientilla voi myös kesken tavoitteen suorituksen kysellä serveriltä tietoja tavoitteen edistymisestä tai peruuttaa tarvittaessa suorituksen kokonaan. (Open Source Robotics Foundation, 2018a.)

3.4.4 ROS Bags

ROS bag on ROS:n viestidatan tallentamiseen ja toistamiseen tarkoitettu tiedostoformaatti, joka saa nimensä tiedostojen ”.bag”-loppuliitteestä. Bageihin voi tallentaa yhteen tai useampaan ROS-aiheeseen (topic) julkaistavaa viestiliikennettä, kuten sensoridataa tai muuta noodien välistä viestintää. Bageihin tallennettua viestiliikennettä on mahdollista myös julkaista uudelleen samoihin (myös muihin) viestiaiheisiin, joista ne on kerätty, jolloin esimerkiksi fyysisellä laitteella ajettu testi, jonka viestiliikenne on tallennettu bagiin, voidaan toistaa virtuaaliympäristössä tai toisinpäin, mikä voi olla olennainen apu sovelluksen testauksessa ja kehityksessä. (Open Source Robotics Foundation, 2020c.)

3.4.5 Parametrit

ROS1-parametrit ovat yksinkertaistettuna kokoelma globaalisti saatavilla olevia muuttujia, joita säilytetään samassa paikassa. Parametriominaisuus on toteutettu keskitettynä parametriserverinä, eikä sitä ole rakennettu suorituskykyä ajatellen, eli parametriserverillä on tarkoitus säilyttää lähinnä dataa, jota ei ole tarkoitus jatkuvasti päivittää, kuten esimerkiksi noodien käynnistyskonfiguraatioita. Parametriserveri toimii ROS Masterin sisällä, mikä tarkoittaa sitä, että noodien kaikki parametrikyselyt ovat käytännössä tietopyyntöjä Master-noodille ja parametrit ovat vapaasti saatavilla kaikista noodeista, sekä kaikilla ROS Masteriin kytköksissä olevilla laitteilla. (Open Source Robotics Foundation, 2018b.)

4 ROS 2

Vaikka tämä työ liittyy lähinnä ROS1-versioon, on ROS2 kuitenkin siinä määrin olennainen, että se käydään myös pintapuolisesti läpi. ROS2 on ROS-systeemin täydellinen uudelleenrakennus, joka tuo mukanaan useita parannuksia ja korjaa ROS1-versiossa havaittuja puutteita. Myös ROS-systeemin virallinen kehitys tulee jatkumaan pelkästään ROS2-version parissa.

ROS2:n tämänhetkinen, ja samalla myös ensimmäinen, LTS-versio, Foxy Fitzroy, julkaistiin 5.6.2020 ja sen tuetuiksi alustoiksi on ilmoitettu Ubuntu 20.04, Windows 10, sekä macOS. Vastaavasti ROS1:n viimeisin julkaistu LTS-versio, Noetic Ninjemy, tulee myös olemaan sen viimeinen virallinen LTS-julkaisu. (Open Source Robotics Foundation, 2021e.)

ROS1 ja ROS2 poikkeavat toisistaan olennaisimmin viestiarkkitehtuurinsa suhteen, mutta myös ydinohjelmisto, tai ainakin sen arkkitehtuuri, on rakennettu käytännössä kokonaan uudelleen ROS2-versiota varten, mikä mahdollistaa tulevaisuudessa uusien ohjelmistokirjastojen helpomman ja ennen kaikkea yhdenmukaisemman lisäämisen ROS-alustan ydinkirjastoihin.
Käytöltään ROS2 ei lopulta poikkea kovinkaan paljon aiemmasta toteutuksesta, mutta esimerkiksi käsitteet kuten roscore ja master node ovat olemassa vain ROS1:n kontekstissa, kun taas ROS2-toteutuksessa jokainen noodi on kykeneväinen itsenäisesti löytämään viestiaiheet, joihin se liitetään kuuntelijaksi eikä master noden kaltaista keskeistä hallintaelementtiä tarvita. ROS2 myös ohjaa koodin tasolla käyttäjää selvästi enemmän kohti OOP (*Object Oriented Programming*) -tyylistä ohjelmointia, koska jokainen käyttäjän luoma noodi tulee periyttää noodien kantaluokasta eli toteuttaa luokkana toisin kuin ROS1:ssä, jossa noodina voi käytännössä toimia myös yksittäinen funktio.

4.1 Viestiarkkitehtuuri

ROS2:n viestiarkkitehtuuri on rakennettu DDS-standardin päälle eli viestiarkkitehtuuri ei enää ole ROS:n oma toteutus, vaan ROS2 vaatii toimiakseen erillisen DDS-

toteutuksen. DDS-standardin omaksuminen mahdollistaa QoSn (*Quality of Service*) säätämisen sovelluskohtaisen tarpeen mukaisesti TCP-protokollan kaltaisesta kuitatusta viestittelystä aina reaaliaikaiseen tiedonsiirtoon saakka, mikä ei ainakaan luontaisesti ollut mahdollista ROS1:n yhteydessä. (Open Source Robotics Foundation, 2021a.) ROS2 Foxy Fizroyn asennuspaketti sisältää kaksi DDS-toteutusta; eProsiman Fast RTPS ja Cyclone DDS, joista perusasetuksena on käytössä Fats RTPS (Open Source Robotics Foundation, 2021f).

4.2 RCL Ros Client Library

RCL on ohjelmistokirjasto, jonka kautta ROS2:n perusominaisuuksia (noodit, parametrit jne.) käytetään. Tämä peruskirjasto on toteutettu mahdollisimman keskitetysti, mikä tarkoittaa sitä, että eri ohjelmointikielillä kirjoitetut ROS2-noodit käyttävät ROS:n perusominaisuuksia aina saman ydinkirjaston kautta. Tämä yhteinen pohja tekee eri kielten kirjastoista ROS2-versiossa toiminnaltaan ja nimeämiskäytännöiltään paljon yhteneväisempiä kuin ROS1-versiossa, jossa jokaista ohjelmointikieltä varten oli käytännössä oma erillinen ROS-toteutuksensa (esim. rospy/ roscpp) ja uudet ominaisuudet piti aina luoda erikseen joka toteutukseen. Erilliset kirjastot johtivat myös siihen, että ROS1-versiossa monia ominaisuuksia löytyy vain yhden tietyn kielen toteutuksesta ja puuttuu muista. Keskitetty ydinkirjasto ei tarkoita sitä, etteikö eri kielten kirjastoihin olisi mahdollista luoda omia erillisiä laajennuksia, mutta mahdolliset ydintoiminnallisuuteen haluttavat lisäykset ja päivitykset ovat ROS2:n keskitetyssä mallissa huomattavasti helpompia toteuttaa ja ylläpitää.

4.3 Parametrit

Koska ROS2:n arkkitehtuuri ei sisällä samanlaisia keskuselementtejä kuin ROS1, myös ROS2:n parametrit on toteutettu hajautetusti. Sen sijaan, että parametrit olisivat saatavilla yhdestä keskitetystä sijainnista, ne ovat aina osa jotakin tiettyä noodia. (Open Source Robotics Foundation, 2021g.)

5 GAZEBO

Gazebo on yleisesti ROS-projektien kanssa käytetty avoimen lähdekoodin 3D- (robotiikka-) simulaatio-ohjelmisto. Se on saanut alkunsa Willow Garagessa samoihin aikoihin ROS-alustan kanssa ja on nykyään ROS:n ohella toinen Open Source Robotics Foundationin alaisuudessa kehitetty ja ylläpidetty avoin ohjelmistotuote.

Vaikka ROS:lle löytyy valmiita rajapintoja myös muihin simulointityökaluihin, on Gazebo näistä käytetyin ja sen yhteensopiva versio on sisällytetty esimerkiksi tässä työssä käytetyn ROS Melodic Morenian täysversioon (*ros-melodic-desktop-full*), mikä tekee käytön aloittamisesta suhteellisen kivutonta. Jos toisaalta haluaa käyttää erillistä 'standalone' Gazebo asennusta, myös tähän löytyy valmiit integraatiopaketit ROS-projektin kotisivuilta (https://wiki.ros.org/gazebo_ros_pkgs), mutta tässä tapauksessa tulee itse huolehtia Gazebo- ja ROS-versioiden yhteensopivuudesta.

5.1 Määrittelytiedostot

ROS1:n yhteydessä gazebossa simuloitavien kappaleiden (robottien) määrittelyyn käytetään URDF (*Unified Robot Description Format*) -formaattia, mikä tarkoittaa käytännössä xml-tiedostoa, johon määritellään erikseen jokainen robottiin kuuluva osa ja sen ominaisuudet (massa, muoto, sijainti yms.), näiden osien keskinäiset suhteet, kuten suhteelliset sijainnit ja mahdollinen liikkuminen toisiinsa nähden sekä mainitun liikkeen tyyppi ja rajoitukset.

Robotin fysikaaliset ja visuaaliset ominaisuudet määritellään URDF-tiedostossa erikseen, mikä mahdollistaa fysiikkavuorovaikutusten laskennan vaativuuden säilyttämisen järkevällä tasolla, vaikka simulaatioon lisättäisiin hyvinkin (visuaalisesti) monimutkaisia rakenteita. Tämä kuitenkin tarkoittaa sitä, että URDF tiedostossa on käytännössä aina määriteltävä kaksi kappaletta yhtä robotin osaa varten; fysiikkasimuloitava kappale, sekä näytettävä visuaalinen kappale. Samassa URDF-tiedostossa on kappaleiden ominaisuuksien lisäksi mahdollista myös määritellä kappaleille niiden liikkeitä ajavia gazebo- plugineja (Open Source Robotics Foundation, 2014b).

6 CTRLX

ctrlX on Bosch Rexrothin kehittämä automaatioalusta, joka perustuu uudenlaiseen avoimeen ohjelmistoarkkitehtuuriin. Ydinajatuksena ovat alustalle asennettavat, älypuhelinmaailmasta innoituksensa saaneet, sovellukset (*apps*), joiden kautta Rexroth pyrkii tuomaan avoimen ohjelmistoekosysteemin ajatusta mukaan teollisuusautomaatioon. Alustan käyttäjät voivat hieman mobiilisovellusten hengessä vapaasti kehittää/asentaa alustalle uusia sovelluksia omien automaatiosovellustensa tarpeisiin sekä ladata/jaella niitä ctrlX App Store -sivuston tai muun kanavan kautta (esim. github). Asennettavat sovellukset itsessään ovat ctrlX:n yhteydessä niin sanottuja snappeja, eli snapcraft-paketointityökalulla luotuja sovelluspaketteja, jotka ovat käytössä myös esimerkiksi Ubuntu Linux -alustalla. (Bosch Rexroth, 2020.)

6.1 ctrlX CORE

CtrlX-alustan keskeinen hallintaelementti on sen Ubuntu Linuxiin (Ubuntu core 18.04) perustuva ohjelmistoydin ’ctrlX CORE’, jolle aiemmin mainitut sovelluspaketit asennetaan ja jolla niitä ajetaan. CORE on käytännössä CPU-yksikkö, jolle on asennettuna Rexrothin ctrlX WORKS -ohjelmisto, jonka kautta COREn sovelluksia puolestaan asennetaan ja hallinnoidaan. CtrlX COREsta on saatavilla myös täysin virtuaalinen versio, joka toimi myös tämän työn esimerkkisovelluksen asennus- ja testialustana.

7 SNAPCRAFT

CtrlX-alustalle asennettavien sovellusten tulee olla pakattuna snap-muotoon, mistä johtuen snapcraft-paketointityökaluun tutustuminen ja sen käytön opettelu olivat olennainen osa työn käytännön osuutta. Snapcraft-työkalulla luotuja sovelluspaketteja kutsutaan snapeiksi, ja ne ovat sovelluspaketteja, jotka tuovat kaikki tarvitsemansa riippuvuudet mukanaan samassa paketissa. Tämä mahdollistaa saman paketin käyttämisen usealla eri Linux-alustalla ja tekee niistä tavallisia sovelluksia turvallisempia, sekä eristettyjä siitä ympäristöstä, johon ne on asennettu. Snappeja ei myöskään ole asennuksen jälkeen mahdollista (päivityksiä lukuun ottamatta) muokata ulkoapäin. (Canonical Ltd, 2021b.)

Snapcaft-paketointityökalu on itsekin helpoiten asennettavissa snap-muodossa, koska asennukseen vaadittava snap-työkalu tulee ainakin Ubuntu Linuxin tapauksessa nykyään esiasennettuna käyttöjärjestelmän mukana. Snapcraftin asennus onnistuu snap-työkalun kautta seuraavalla komennolla:

```
$ sudo snap install snapcraft --classic
```

7.1 Käyttö lyhyesti

ROS sovellusten yhteydessä snapcraft on jälleen uusi rakennustyökalukerros jo olemassa olevien CMake- ja Catkin-työkalujen päälle ja se vaatii myös omat erilliset paketointiohjeensa kyetäkseen tuottamaan toimivan snap paketin. Snapcaft rakennustyökalulle tulee ROS-paketoinnin yhteydessä luoda tyhjä hakemisto catkin-työtilan juureen (Kuva 1) ja ajaa siellä komento:

```
$ snapcraft init
```

Komento generoi hakemistoon ’snapcraft.yaml’-tiedoston. Tämä on Snapcraft-työkalun vaatima ohjetiedosto, jossa määritellään paketoitavan ROS-sovelluksen halutut ominaisuudet kuten käytettävät pluginit, käyttäjälle saatavilla olevat komennot, paketin vaatimat oikeudet, sekä muut luotavan snapin ominaisuudet. (Canonical Ltd, 2021c.)

Kun tarvittavat rakennustiedot on määritelty ’snapcraft.yaml’-tiedostoon, tarvitsee snap-paketin luomiseksi enää ajaa catkin-työtilan juuressa komento:

```
$ snapcraft
```

Komennolla työkalu rakentaa halutun ROS-snap -paketin annettujen rakennustietojen pohjalta. Perusasetuksena snapcraft pyrkii luomaan paketin multipass-virtualisointi-työkalun avulla ja pyytää sen asennusta, ellei sitä ympäristöstä vielä löydy.

8 KÄYTÄNNÖN OSUUS

Työn käytännön osuudessa oli tavoitteena tuottaa demoluontoinen sovellus ROS-alustan potentiaalisesta käyttökohteesta Cimcorpin liikkeenohjauksessa. Käytännön osuuden sovelluskohde rajattiin työssä aluksi Cimcorpin robottisoluun ja tarkemmin gantry-robotin ohjaukseen, mistä se edelleen tarkentui ROS:n käytön testaamiseen Rexrothin ctrlX-ohjaimen yhteydessä.

Käytetty kehitysympäristö rakennettiin Cimcorpin tarjoamalle Windows10-koneelle, jolle asennettiin virtuaalinen ctrlX CORE, sekä VMWare Workstation Player, jolla puolestaan luotiin ROS-/Gazebo-kehitystä, sekä sovelluksen testausta varten Ubuntu 18.04 -virtuaalikone. Myös VirtualBox-, sekä Windows Subsystem for Linux -työkaluja kokeiltiin, mutta näillä huomattiin olennaisia ongelmia itse snapcraftin ja/tai snapcraftin käyttämän multipass-virtualisointityökalun johdosta vaaditun sisäkkäisen virtualisoinnin kanssa.

8.1 Työn aloitus

Sovelluksen toiminnallisuuden suhteen ensimmäinen tavoite oli rakentaa python-toteutuksena ROS1-sovellus, joka toimii ctrlX:n REST-rajapinnan kautta ei-reaaliaikaisena linkkinä ctrlX-ohjaimen datakerroksen ja ulkoisen Gazebo-simulaation välillä. Sovellus oli tavoitteena saada paketoitua snapcraft-työkalulla snap-muotoon ja asentaa ctrlX Corelle itsenäisenä ’appina’. Tämän toteutuksen jälkeen seuraavina potentiaalisina askeleina mietittiin reaaliaikaisen tiedonvälityksen mahdollisuuksien tutkimista (C++ toteutus ctrlX:n SDK:ta käyttäen) ja/tai jo aikaansaadun toiminnallisuuden toteuttamista ROS2:lla. Varsinaisen fyysisen ctrlX-alustan sijasta työssä käytettiin paikallisesti asennettua virtuaaliversiota ctrlX:stä, joka on kuitenkin käytännössä toiminnan kannalta sama asia.

Koska ctrlX:n datakerroksesta haluttiin nimenomaan välittää jotakin merkityksellistä tietoa simulaatioon, käytettiin ctrlX COREN päässä simulaation ”datalähteenä” Rex-

rothin valmiina tarjoamaa ctrlX COREIle asennettavaa ’Motion’-appia, jolla on mahdollista luoda ja manipuloida virtuaalisia liikeakseleita. Näihin virtuaalisiin akseleihin liittyviä sijaintitietoja sitten välitettiin ctrlX:n datakerroksesta simulaation käyttöön.

8.2 ROS-versio

Rexrothilta saadut materiaalit suosittelivat CtrlX-alustan snap-pakettien pohjaksi (snap base) core18 pohjaa, joka vastaa Ubuntu 18.04 käyttöjärjestelmää. Ubuntu 18.04 käyttöjärjestelmälle ei kuitenkaan ole enää virallisesti tuettua ROS2LTS-julkaisua. Käytettäväksi ROS-versioksi valittiin viimeisin Ubuntu 18.04 -pohjalle suunniteltu tuettu ROS-LTS -versio, joka on ROS1 Melodic Morenia. Myös uudempaa, Ubuntu 20.04 -käyttöjärjestelmälle suunniteltua ROS Noetic Ninjemy -versioita, sekä sen pohjaksi sopivaa Ubuntu core20:tä vastaavaa core20 snap basea testattiin, mutta huonolla menestyksellä. Mainitun testauksen yhteydessä selvisi myös, että Snapcraft-paketointityökalun kehitystyö liittyen core20-basea käyttävien ROS-snappien paketointiin on edelleen kehitysvaiheessa.

8.3 ROS1-sovellus

Varsinainen ctrlX CORE:lle asennettava ROS-sovellus oli itsessään lopulta suhteellisen yksinkertainen toteutus koostuen käytännössä ROS-Masterista sekä yhdestä ROS-noodista (LIITE 3), joka julkaisee ennalta määrättyjen datakerroksen muuttujien tietoja kahteen ROS-topic:iin.

Sovellusta varten ctrlX:lle luotiin uusi käyttäjä, jonka oikeuksilla ROS-sovellus saa ctrlXn REST-rajapinnan kautta yhteyden ctrlXn datakerrokseen. Sovellus pyytää aluksi käynnistyessään ctrlX COREn REST-rajapinnalta käyttöoikeustietueen (*access token*) sovellukseen kovakoodatuilla käyttäjätiedoilla, aloittaa saaduilla oikeuksilla uuden yhteyden ja ryhtyy sen jälkeen tasaisesti kyselemään rajapinnalta kahden ’Motion’-apilla luodun virtuaaliakselin sijaintitietoja noin 50 ms jaksonajalla ja julkaisee saamansa tiedot ROS-topiceissa ’axis_x’ ja ’axis_y’.

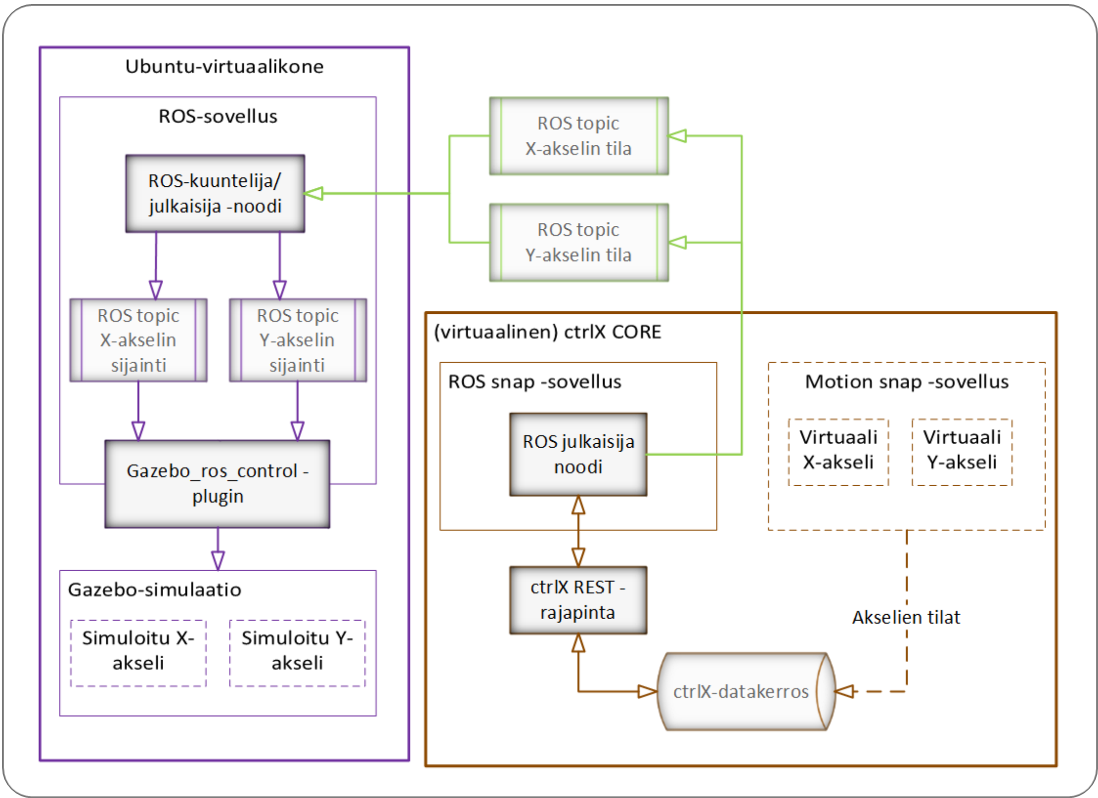
Olennaisin työ itse koodin suhteen liittyi lopulta python-kieleen perehtymiseen, koska REST-clienttien tai minkään muunkaan kirjoittaminen kyseisellä kielellä oli tekijälle entuudestaan suhteellisen tuntematonta. Käytettäväksi työkaluksi valikoitui pythonin 'Requests' http-kirjasto lähinnä helppokäyttöisyytensä takia, mutta myös siksi, että se on saatavilla myös pythonin 2.7 -versiolle, joka on työssä käytetyn ROS1 Melodic Morenian vakiona tukema python-versio ja tästä syystä myös tässä sovelluksessa käytettäväksi valikoitunut versio.

8.4 Sovelluksen paketointi

Sovelluksen toiminnallisuutta enemmän päänvaivaa tuotti sen onnistunut paketointi snap-muotoon. Testausvaiheessa kävi ilmi, että sovellukselle on tärkeää asettaa ympäristömuuttuja ROS_MASTER_URI (IP-osoite, jossa ROS master nodea ajetaan, eli tässä yhteydessä ctrlX:n oma IP-osoite) paketointivaiheessa nimenomaan catkin-pluginin 'catkin-ros-master-uri' parametrina. CtrlXn ulkopuoliset noodit eivät kyenneet kommunikoimaan ctrlX COREIla käynnissä olevan ROS Masterin kanssa, vaikka ne nimenomaan osoitettiin etsimään ROS Master -noodia COREn IP-osoitteesta, jos kyseistä muuttujaa ei ollut erikseen paketoitaessa catkin-pluginin parametreissa asetettu. Luodulle sovellukselle ei rakennettu erillistä käyttöliittymää ja näin ollen sovelluksesta luotu snap määriteltiin käynnistymään palveluna ctrlX COREn käynnistymisen yhteydessä ja yrittämään automaattista uudelleenkäynnistystä aina kaatumisen tai muun vastaavan asiattoman sammumisen jälkeen. Snapille on määritelty ainoastaan yksi käyttökomento, jonka se suorittaa aina käynnistyessään. Tämä käyttökomento ajaa ROS-paketille luodun launch-tiedoston (LIITE 6) roslaunch-työkalulla, mikä puolestaan käynnistää ROS master noden, sekä varsinaisen datakerroskyselyistä ja tietojen julkaisemisesta vastuussa olevan noodin (LIITE 3). Tässä kappaleessa edellä mainitut, sekä muut snap-paketin ominaisuudet on määritelty sovelluksen snapcraft.yaml -määrittelytiedostossa (LIITE 5).

8.5 ROS-Gazebo -simulaatiosovellus

Gazebo-simulaatiota varten toteutettiin erillinen ROS-sovellus, jota ajettiin omassa virtuaalisessa Ubuntu-ympäristössään. Sovellus huolehtii erikseen määritellyn Gazebo-simulaation käynnistämisestä ja sisältää yhden itse toteutetun noodin, joka kuuntelee topicia, johon ctrlX COREllе asennettu ROS-noodi julkaisee ctrlXn datakerroksen tietoja. Kuunteleva noodi tulkkaa ja uudelleen julkaisee datakerroksesta saadut akselien tiedot vastaaviin simulaatioakseleihin liitettyihin gazebo pluginin command-topiceihin (Kuva 3). Koska ROS master on tässä tapauksessa käynnissä erillisellä laitteella (virtuaalinen ctrlX CORE), tulee myös simulaatioon liittyville noodeille kertoa niiden käynnistyksen yhteydessä ROS Masterin osoite, jotta ne kykenevät rekisteröimään itsensä Master noodiin ja aloittamaan tiedonvaihdon. Tämä tapahtuu asettamalla simulaatiota pyörittävällä laitteella ”ROS_MASTER_URI” -ympäristömuuttujan arvoksi ctrlXn osoite.



Kuva 3. Tiedonkulku ctrX COREn datakerrokselta Gazebo-simulaatioon

8.5.1 Simulaation määrittely

Simuloitavan robotin määrittelevä URDF-tiedosto generoitiin tässä tapauksessa niin sanotun xacro-tiedoston kautta. Työn xacro (LIITE 4) on kirjoitettu URDF-tiedostoa vastaavaan muotoon, mutta xacro-tiedostoja käyttämällä on mahdollista kirjoittaa myös tiiviimpiä määrittelytiedostoja käyttämällä esimerkiksi niiden macro-ominaisuutta, jonka kautta voi automaattisesti generoida URDF-tiedoston toisteisia osuuksia. Tämä puoltaa niiden käyttöä etenkin monimutkaisemmissa projekteissa (Open Source Robotics Foundation, 2021h). Varsinaisen URDF-tiedoston generointi xacron perusteella tapahtuu launch-tiedostossa (LIITE 6) sovelluksen käynnistyksen yhteydessä.

Itse robotin lisäksi simulaatioon pitää määritellä myös ympäristö, johon robotti luodaan/ jossa sitä ajetaan. Simulaatioympäristönä toimii tämän työn tapauksessa lähes tyhjä maailma sisältäen pelkän maa-tason. Ympäristö on luotu ennalta käsin Gazebo-sovelluksessa ja tallennettu sieltä .world-tiedostoksi, joka puolestaan ladataan käyttöön launch-tiedostossa simulaatiota käynnistettäessä.

8.5.2 Toiminta

ROSn ja Gazebo-simulaation väliseen tiedonvaihtoon käytettiin gazebo_ros_control-pluginia. Pluginin asetukset määriteltiin pääosin xacro-/URDF-tiedoston sisällä, mikä tapahtui lisäämällä ensin käyttöön itse plugin ja liittämällä tämän jälkeen robotin joint-elementteihin pluginin liikkeenhallintaan vaatimat transmission-elementit määrittelyineen. Näiden xacro-/URDF-määrittelyiden lisäksi pluginin käyttämät joint controllerit pitää erikseen konfiguroida ja luoda/käynnistää roslaunch-tiedoston kautta simulaatiota käynnistettäessä.

Robotin liike tuotetaan yksinkertaistettuna siten, että gazebo-plugin ottaa jatkuvasti vastaan ctrlX:ltä julkaistusta datasta tulkattuja akselien sijaintitietoja ja pyrkii siirtämään simuloidut akselit vastaavaan sijaintiin. Koska plugin toimii PID-ohjaimen tavoin, simulaatio käytännössä laahaa ctrlX:ltä saatujen tietojen perässä, eikä tämä ratkaisu välttämättä ole paras mahdollinen hyvän digitaalisen kaksosen luontiin, mutta toimii kuitenkin demona ctrlX-Gazebo -tiedonvaihdosta.

Parempi lähestymistapa voisi olla hallinnoida pluginin kautta akselien sijaintitietojen sijasta niiden kiihtyvyyksiä, koska näin simulaatioakseleille voisi mahdollisesti syöttää saadut arvot lähes suoraan rikkomatta simulaation fysiikkaa. Tämä on nähtävästi myös mahdollista käytetyn pluginin konfiguraatiota muokkaamalla, mutta jää tulevaisuuden tutkimuskohteeksi. PID-ohjauksen arvot myös asetetaan pluginin akseleita ajaville controllereille erillisestä tiedostosta simulaation käynnistyksen yhteydessä ja näitä arvoja säätämällä/optimoimalla on varmasti mahdollista päästä parempiin tuloksiin myös nykyisellä suoraa sijaintia ohjaavalla ratkaisulla.

8.5.3 Ulkoasu

Ei aivan keskeinen, mutta mainitsemisen arvoinen asia, oli sinänsä yksinkertaiselta kuulostava Cimcorpin värien tuominen mukaan simulaatioon. Myös simulaatiokappaleiden materiaalit ja tekstuurit on mahdollista määritellä URDF-tiedostossa <material>-tagien sisällä jokaiselle simulaatioon määritellylle elementille erikseen. Tämä kuitenkin vaati toimiakseen muutamia askeleita: URDF-tiedostoa käyttävän ROS-paketin package.xml -tiedostoon pitää olla erikseen määriteltynä ’gazebo_media_path’-sijainti:

```
<export>
    <gazebo_ros gazebo_media_path="${prefix}"/>
</export>
```

Yllä ’gazebo_media_path’ on ”${prefix}”-merkinnällä määritelty samaan sijaintiin kuin, jossa itse package.xml-tiedosto sijaitsee. Materiaalimäärittelyjen ja tekstuurien tulee lisäksi löytyä tästä sijainnista ennalta määrättyjen tiedostopolkujen takaa:
media/materials/scripts/gazebo.material tulee sisältää kaikkien käytettyjen materiaalien määrittelyt gazebon ymmärtämässä muodossa mukaan lukien niihin liitettävien tekstuuritiedostojen nimet.
media/materials/textures tulee sisältää ’gazebo.material’-tiedostossa määritellyn nimiset ja tyyppiset tekstuuritiedostot.

Koska simulaatioon haluttiin juurikin Cimcorpin gantry, piti Cimcorpin tarjoamat 3D-meshit UV-kartoittaa ja teksturoida, sekä tallentaa .dae-tiedostomuotoon, jotta niiden UV-kartat ja sitä kautta tekstuurit saatiin tuotua simulaatioon oikein.

Kuva 4. Simuloitu gantry. Kuvankaappaus gazebo-simulaatiosta

9 TULOKSET JA PÄÄTELMÄT

Työssä saatiin tuotettua yleiskuvaus ROS-systeemistä, sekä snap-muotoon paketoitu ROS1-sovellus, joka on asennettavissa ctrlX CORE:lle ja kykenee kommunikoimaan ctrlX COREn datakerroksen, sekä ulkoisten ROS-noodien kanssa välittäen datakerroksen muuttujien tietoja ulkoiselle simulaatiolle. Näiltä osin alustavat tavoitteet saavutettiin, mutta erinäisiä mietittyjä asioita/tavoitteita jäi myös työn ulkopuolelle. Näistä merkittävimpinä ROS2-version testaaminen ja reaaliaikainen tiedonvaihto, sekä ctrlXn oman SDK:n käyttö osana ROS-sovellusta. Kaiken kaikkiaan olen kuitenkin kohtuullisen tyytyväinen tuloksiin, koska toimiva, joskin suhteellisen yksinkertainen, sovellus saatiin kuitenkin aikaan.

9.1 Ongelmia

Työn olennaisimpia ongelmia oli snapcraftin käytön kanssa toimivan simulaatioympäristön pystytys. Työtä tehdessä käytettävissä olevat ympäristöt olivat Cimcorpin tarjoama Windows10 ja oma Ubuntu 20.04, mutta koska käytettävä ROS-versio oli Melodic, tarvittiin kehitykseen juuri tätä versiota vastaava käyttöjärjestelmä Ubuntu 18.04. Tähän tarkoitukseen virtualisointi oli selkeä valinta ja testasin aluksi WSL2- (*Windows Subsystem for Linux*) sekä VirtualBox-ympäristöjä, mutta molemmilla oli omat ongelmansa joko itse snapcraftin tai viimeistään snapcraftin snappien rakentamiseen käyttämän multipass-virtualisointityökalun kanssa ja lopulta päädyin käyttämään VMware Workstation Playeria (VWP), jolla snapcraftin rakennustyökalun vaatima sisäkkäinen virtualisointi onnistui suhteellisen kivuttomasti pelkästään VWP:n omia asetuksia säätämällä.

Snapcraftilla on mahdollista rakentaa snappeja myös paikallisesti ilman multipass-virtuaalikonetta (potentiaalina paikallisen systeemin korruptointi), tai ulkoisella serverillä käyttäen Launchpadia (vaatii launchpad tilin ja rakennettavan snapin pitää olla avoimen lähdekoodin ohjelmisto, koska koodi menee julkisesti saataville), mutta kumpikaan näistä vaihtoehdoista ei vaikuttanut tämän työn yhteydessä houkuttelevalta. (Canonical Ltd., 2021d.) Tästä ongelmasta pitää kuitenkin myös todeta, että vastaavan

virtuaalisen työympäristön käyttö mahdollisessa tulevassa kehitystyössä on epätodennäköistä eli nämä ongelmat lienevät olleet olennaisia lähinnä vain tämän opinnäytetyön kontekstissa.

ROS- ja oikeastaan erityisesti Gazebo-simulaation rakentamisen suhteen ongelmia tuotti saatavilla olevan tiedon oikeellisuuden ja tiettyyn ohjelmiston versioon sovellettavuuden selvittäminen. Koska sekä ROS, että Gazebo ovat kehittyneet nykyiseen tilaansa useamman iteraation kautta, niin vaikka materiaalia on saatavilla paljon, (tai ehkä myös siitä johtuen) on myös vanhentuneen, virheellisen tai tietyn version kontekstissa väärän tiedon määrä suuri ja ainoa tapa selvittää asioita on lopulta usein ”yritys ja erehdys”.

9.2 Jatkokehitys

Koska ROS1:n virallinen tuki on päättymässä, olennaisin jatkokehitysaihe lienee vastaavan systeemin tuottaminen ROS2:n pohjalta. Tätä kannattaisi todennäköisesti lähteä aluksi tutkimaan Ubuntu 18.04:lle suunnitellulla ROS2-versiolla kuten ’eloquent’ tai ’dashing’ ainakin niin kauan kuin Ubuntu core20 -pohjan vaativien ROS-snappien rakentaminen on snapcraft-työkalun puolesta kehitystyön alla ja ctrlX:n suositeltu snap base edelleen Ubuntu core 18.

Toinen olennainen kehityksen kohde on ctrlX:lle asennettavan ROS noden käyttö ctrlX SDK:n kanssa, jolloin datakerroksen tietojen kysely tapahtuisi COREn sisäisen rajapinnan kautta, eikä lähinnä ulkoiseen käyttöön tarkoitetun REST-rajapinnan kautta. Tällöin kehityskieleksi tosin pitää SDK:sta johtuen vaihtaa C++, ellei ctrlX:n SDK:sta sitten ole julkaistu versiota uusilla kielillä. Mielenkiintoista olisi myös pyrkiä lisäämään ROS-snapille käyttöliittymä ctrlX:n puolelle, jolla vaihtaa tai suoraan valita ne datakerroksen tiedot, joita halutaan julkaista, sekä luoda/muokata niihin liittyviä topiceja tarpeen mukaan.

Koska tässä työssä luotu simulaatio oli luonteeltaan enemmänkin ROS-paketin asennuksen ja tiedonvaihdon onnistumista painottava ”proof of concept” -tyyppinen rakennelma kuin varsinainen digitaalinen kaksonen, olisi fysikaalisesti ja ohjauksellisesti

tarkan digitaalisen kaksosen luominen esimerkiksi testauskäyttöön myös mielenkiin-toinen jatkokehityksen aihe. Tähän liittyviä aiheita olisivat ainakin alkuun vähintään-kin PID-ohjaimen asetusten optimointi, parhaiten soveltuvan ohjaintyypin selvittämi-nen, kappaleiden fysikaalisten ominaisuuksien määrittelyiden säätäminen vastaamaan tarkemmin esikuviaan, sekä robotin jointeille asetettujen transmission-elementtien asetusten säätäminen.

LÄHTEET

Ackerman, E. (2012). Open Source Robotics Foundation Officially Announced. IEEE Spectrum. https://spectrum.ieee.org/automaton/robotics/robotics-software/open-source-robotics-foundation-officially-announced

Alhonen, A. (2017). Why don't we use ROS? Pulurobotics Oy Ltd. https://www.pulurobotics.fi/blog/pulurobotics-blog-1/post/why-don-t-we-use-ros-7

Bosch Rexroth. (2020). The new freedom in engineering: The ctrlX automation platform. https://dc-ca.resource.bosch.com/media/general_use/products/electric_drives_and_controls_2/images_13/ctrlx_automation_the_new_freedom_in_engineering_en_202006.pdf

Canonical Ltd. (2021a). Create your first snap. Haettu 11.6.2021 osoitteesta https://ubuntu.com/tutorials/create-your-first-snap#1-overview

Canonical Ltd. (2021b). ROS applications. Haettu 11.6.2021 osoitteesta https://snapcraft.io/docs/ros-applications

Canonical Ltd. (2021c). Creating snapcraft.yaml. Haettu 11.6.2021 osoitteesta https://snapcraft.io/docs/creating-snapcraft-yaml

Canonical Ltd. (2021d). Build options. Haettu 11.6.2021 osoitteesta https://snapcraft.io/docs/build-options)

Foote, T. & Conley, K (2010). Target Platforms. Open Source Robotics Foundation, Inc. Haettu 31.3.2021 osoitteesta https://www.ros.org/reps/rep-0003.html

Open Source Robotics Foundation, Inc. (2014a). ROS/ concepts. Haettu 12.9.2021 osoitteesta http://wiki.ros.org/ROS/Concepts

Open Source Robotics Foundation, Inc. (2014b). Tutorial: ROS Control. Haettu 11.8.2021 osoitteesta http://gazebosim.org/tutorials/?tut=ros_control

Open Source Robotics Foundation, Inc. (2017). Catkin Workspaces. Haettu 31.3.2021 osoitteesta http://wiki.ros.org/catkin/workspaces

Open Source Robotics Foundation, Inc. (2018a). actionlib. Haettu 18.2.2021 osoitteesta http://wiki.ros.org/actionlib

Open Source Robotics Foundation, Inc. (2018b). Parameter Server. Haettu 13.1.2021 osoitteesta http://wiki.ros.org/Parameter%20Server

Open Source Robotics Foundation, Inc. (2019a). catkin/ CMakeLists.txt. Haettu 31.3.2021 osoitteesta http://wiki.ros.org/catkin/CMakeLists.txt

Open Source Robotics Foundation, Inc. (2019b) roscore. Haettu 13.1.2021 osoitteesta http://wiki.ros.org/roscore

Open Source Robotics Foundation, Inc. (2020a). catkin/ conceptual_overview. Haettu 31.3.2021 osoitteesta https://wiki.ros.org/catkin/conceptual_overview

Open Source Robotics Foundation, Inc. (2020b). Packaging your ROS project as a snap. Haettu 12.9.2021 osoitteesta http://wiki.ros.org/ROS/Tutorials/Packaging%20your%20ROS%20project%20as%20a%20snap

Open Source Robotics Foundation, Inc. (2020c). Bags. Haettu 11.8.2021 osoitteesta http://wiki.ros.org/Bags

Open Source Robotics Foundation, Inc. (2021a). About Quality of Service settings. Haettu 13.1.2021 osoitteesta https://index.ros.org/doc/ros2/Concepts/About-Quality-of-Service-Settings/

Open Source Robotics Foundation, Inc. (2021b). Haettu 18.2.2021 osoitteesta https://www.openrobotics.org/

Open Source Robotics Foundation, Inc. (2021c). ROS install page. Haettu 10.08.2021 osoitteesta https://www.ros.org/install/

Open Source Robotics Foundation, Inc. (2021d). Noetic Ninjemys: The Last Official ROS 1 Release. https://www.openrobotics.org/blog/2020/5/23/noetic-ninjemys-the-last-official-ros-1-release

Open Source Robotics Foundation, Inc. (2021e). Distributions. Haettu 11.8.2021 osoitteesta http://wiki.ros.org/Distributions#List_of_Distributions

Open Source Robotics Foundation, Inc. (2021f). Installing DDS implementations. Haettu 18.2.2021 osoitteesta https://docs.ros.org/en/foxy/Installation/DDS-Implementations.html

Open Source Robotics Foundation, Inc. (2021g). About parameters in ROS 2. Haettu 28.1.2021 osoitteesta https://index.ros.org/doc/ros2/Concepts/About-ROS-2-Parameters/

Open Source Robotics Foundation, Inc. (2021h). xacro. Haettu 11.8.2021 osoitteesta http://wiki.ros.org/xacro

Tellez, R. (2019). A History of ROS (Robot Operating System). The Construct. https://www.theconstructsim.com/history-ros/

[ROS1 Melodic Morenia karsittu asennusoje]

Ennen asennusta paikallinen ohjelmistokirjasto pitää asettaa hakemaan paketteja myös Ubuntun "restricted" "universe," ja "multiverse" ohjelmistokirjastoista.

Tämä onnistuu komentosarjalla:

```
$ sudo add-apt-repository restricted
$ sudo add-apt-repository universe
$ sudo add-apt-repository multiverse
$ sudo apt update
```

Käyttöjärjestelmä pitää myös asettaa hyväksymään ohjelmistopaketteja kohteesta packages.ros.org:

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_re-
lease -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

Tunnisteavainten asetus:

```
$ sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --
recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

Yllä olevien askelten jälkeen paikallinen järjestelmä pystyy löytämään ja asentamaan tarvittavat ROS-paketit. Lopuksi lisättyjen sijaintien tiedot pitää päivittää paikalliseen paketti-indeksiin:

```
$ sudo apt update
```

Nyt on mahdollista asentaa haluttu ROS-kokoonpano:

Täysi työpöytäasennus (Suositeltu)**:** ROS, rqt, rviz, robottigeneeriset kirjastot, 2D-/3D-simulaattorit ja 2D-/3D-havainnointi

```
$ sudo apt install ros-melodic-desktop-full
```

Työpöytä asennus: ROS, rqt, rviz, ja robottigeneeriset kirjastot

```
$ sudo apt install ros-melodic-desktop
```

ROS-Base: (Bare Bones) ROS, build, and communication libraries. No GUI tools.

```
$ sudo apt install ros-melodic-ros-base
```

Yksittäinen paketti: Myös yksittäisten ROS-pakettien asentaminen erikseen on mahdollista komennolla:

```
$ sudo apt install ros-melodic-<paketin_nimi>
(esim: $ sudo apt install ros-melodic-slam-gmapping)
```

Saatavilla olevien pakettien löytämiseksi voi käyttää komentoa:

```
$ apt search ros-melodic
```

(Esimerkkiohje on yksinkertaistettu ja lyhennetty käännös englanninkielisestä lähteestä: http://wiki.ros.org/melodic/Installation/Ubuntu. Em. lähteessä on myös selitetty erilaisten erikseen asennettavien ROS-työkalujen ja muiden lisäosien asennusta.)

LIITE 2

[Hyvin yksinkertaisen ROS-noodin toteutus koodin tasolla]

```
#! /usr/bin/python2.7

import rospy //ros python-pääkirjasto
from std_msgs.msg import String //String-tyyppinen ROS message

def timed_talker():
    #luodaan julkaisija 'example_topic'- topickiin
    pub = rospy.Publisher('example_topic', String, queue_size=10)
    #noodin alustus
    rospy.init_node('example_node', anonymous=True, log_level=rospy.INFO)
    rate = rospy.Rate(1)

    #julkaistaan viestiä loopissa niin kauan kuin ros on käynnissä
    while not rospy.is_shutdown():
        #viestin julkaisu topic:iin
        pub.publish("testing…")
        rate.sleep()

if __name__ == '__main__':
    try:
        timed_talker()

    except rospy.ROSInterruptException:
        pass
```

[ctrlX CORElle asennetun julkaisijanoodin toteutus]

```
#! /usr/bin/python2.7

import rospy
from std_msgs.msg import String

import sys
import requests
import json

def poller():
    # initialize the node and create publisher for topics 'axis_x and axis_y'
    rospy.init_node('poller', anonymous=True, log_level=rospy.INFO)
    pub_x = rospy.Publisher('axis_x', String, queue_size=10)
    pub_y = rospy.Publisher('axis_y', String, queue_size=10)
    rate = rospy.Rate(20) # trying to get a 20Hz polling rate

    # start a requests session to continuously poll for datalayer info
    # with a single existing connection. With 'with', the session should also
    # get closed when the ROS-node is closed or an unhandled exception occurs.
    with requests.Session() as session:

        # get an access token for the ctrlX and add it as an
        # Authorization header for the session requests 127.0.0.1
        user = '{"name":"boschrexroth","password":"Bossi"}'
        # the content-type must also be specified in the header
        session.headers.update({"Content-Type":"application/json"})
        auth_response = session.post("https://127.0.0.1/identity-man-
ager/api/v1/auth/token", data=user, verify=False)
        response_dict = json.loads(auth_response.text) # modify re-
sponse text to python dictionary
        token = str(response_dict["access_token"]) # get the access to-
ken from the dict
        session.headers.update({'Authorization': 'Bearer ' + token}) # update re-
quest headers for the session with the token

        # keep polling for datalayer data as long as ROS is running
        while not rospy.is_shutdown():

            try:
                # request timeout in 5s, don't use ssl verfica-
tion since it would fail anyways for ctrlX
                response_x = session.get("https://127.0.0.1/automation/api/v1.0/mo-
tion/axs/Axis_x/state/values/actual", timeout=5, verify=False)
                response_x.raise_for_status() # raise an exception for inva-
lid http statuscodes
```

```
                response_y = session.get("https://127.0.0.1/automation/api/v1.0/mo-
tion/axs/Axis_y/state/values/actual", timeout=5, verify=False)
                response_y.raise_for_status() # raise an exception for inva-
lid http statuscodes

                pub_x.publish(response_x.text)
                pub_y.publish(response_y.text)
                rate.sleep() # wait and poll again

            except requests.exceptions.HTTPError as error:
                print(error)
            except requests.exceptions.ConnectionError as error:
                print(error)
            except requests.exceptions.Timeout as error:
                print(error)
            except requests.exceptions.RequestException as error:
                print(error)

if __name__ == '__main__':
    try:
        poller()

    except rospy.ROSInterruptException:
        pass
```

[Gantry-simulaation URDF-määrittelytiedosto (xacro)]

```
<?xml version="1.0" ?>
<robot name="fake_gantry" xmlns:xacro="http://www.ros.org/wiki/xacro">

  <link name="world"/>
  <joint name="fixed" type="fixed">
    <parent link="world"/>
    <child link="fixed_base"/>
  </joint>
<!--oooooooooooooooooooooooooooooooooooooooooooooooooooooo-->
  <link name="fixed_base">
    <inertial>
      <origin xyz="0.0 0.0 0.25" rpy="0.0 0.0 0.0" />
      <mass value="500"/>
      <iner-
tia ixx="100.0" ixy="0.0" ixz="0.0" iyy="100.0" iyz="0.0" izz="100.0"/>
    </inertial>

    <visual>
      <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
      <geometry>
        <mesh filename="$(find simu)/meshes/cimcorp_gan-
try_rails2.dae"/><!--scale="1.0 1.0 1.0"-->
      </geometry>
    </visual>

    <collision>
      <origin xyz="0.0 0.0 0.25" rpy="0.0 0.0 0.0"/>
        <geometry>
          <box size="5.0 1.0 0.5"/>
        </geometry>
    </collision>
  </link>
<!--oooooooooooooooooooooooooooooooooooooooooooooooooooooo-->
  <joint name="joint_1" type="prismatic">
    <origin xyz="0.0 0.0 0.0" rpy="0 0 0" />
    <parent link="fixed_base"/>
    <child link="moving_part_1"/>
    <axis xyz="1.0 0.0 0.0"/>
    <limit lower="-18.3" upper="19.3" effort="100.0" velocity="10.0"/>
  </joint>
<!--oooooooooooooooooooooooooooooooooooooooooooooooooooooo-->
  <link name="moving_part_1">
    <inertial>
      <origin xyz="0.0 0.0 0.75" rpy="0.0 0.0 0.0"/>
      <mass value="10.0"/>
```

```
      <iner-
tia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
    </inertial>

    <visual>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
      <geometry>
        <mesh filename="$(find simu)/meshes/cimcorp_gantry_bridge.dae"/>
      </geometry>
    </visual>

    <collision>
      <origin xyz="0.0 0.0 0.75" rpy="0.0 0.0 0.0"/>
      <geometry>
        <box size="0.5 3.0 0.5"/>
      </geometry>
    </collision>
  </link>
<!--oooooooooooooooooooooooooooooooooooooooooooooooooooooo-->
  <joint name="joint_2" type="prismatic">
    <origin xyz="0.0 0.0 0.0" rpy="0 0 0" />
    <parent link="moving_part_1"/>
    <child link="moving_part_2"/>
    <axis xyz="0.0 1.0 0.0"/>
    <limit lower="-6.0" upper="6.0" effort="100.0" velocity="10.0"/>
  </joint>
<!--oooooooooooooooooooooooooooooooooooooooooooooooooooooo-->
  <link name="moving_part_2">
    <inertial>
      <origin xyz="0.375 0.0 0.75" rpy="0.0 0.0 0.0"/>
      <mass value="10.0"/>
      <iner-
tia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
    </inertial>

    <visual>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
      <geometry>
        <mesh filename="$(find simu)/meshes/cimcorp_gantry_y.dae"/>
      </geometry>
    </visual>

    <collision>
      <origin xyz="0.375 0.0 0.75" rpy="0.0 0.0 0.0"/>
      <geometry>
        <box size="0.5 3.0 0.5"/>
      </geometry>
    </collision>
  </link>
```

```
<!--oooooooooooooooooooooooooooooooooooooooooooooooooooooooo-->
<gazebo reference="fixed_base">
  <material>
    RailsMaterial
  </material>
</gazebo>
<!--oooooooooooooooooooooooooooooooooooooooooooooooooooooooo-->
<gazebo reference="moving_part_1">
  <material>
    BridgeMaterial
  </material>
</gazebo>
<!--oooooooooooooooooooooooooooooooooooooooooooooooooooooooo-->
<gazebo reference="moving_part_2">
  <material>
    YMaterial
  </material>
</gazebo>
<!--oooooooooooooooooooooooooooooooooooooooooooooooooooooooo-->
  <gazebo>
    <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
      <robotNamespace>/fake_gantry</robotNamespace>
    </plugin>
  </gazebo>
<!--oooooooooooooooooooooooooooooooooooooooooooooooooooooooo-->
  <transmission name="trans_1">
    <type>transmission_interface/SimpleTransmission</type>

    <joint name="joint_1">
      <hardwareInterface>hardware_interface/EffortJointInterface</hard-
wareInterface>
    </joint>

    <actuator name="motor_1">
      <mechanicalReduction>1</mechanicalReduction>
      <hardwareInterface>EffortJointInterface</hardwareInterface>
    </actuator>
  </transmission>
  <!--oooooooooooooooooooooooooooooooooooooooooooooooooooooooo-->
  <transmission name="trans_2">
    <type>transmission_interface/SimpleTransmission</type>

    <joint name="joint_2">
      <hardwareInterface>hardware_interface/EffortJointInterface</hard-
wareInterface>
    </joint>

    <actuator name="motor_2">
      <mechanicalReduction>1</mechanicalReduction>
```

```
      <hardwareInterface>EffortJointInterface</hardwareInterface>
    </actuator>
  </transmission>

</robot>
```

LIITE 5

[ROS-snapin paketointimäärittelyt (snapcraft.yaml)]

```
name: test
base: core18 # the base snap is the execution environment for this snap
version: '0.1' # just for humans
summary: A ROS-REST poller.
description: |
  A ROS master & a node that polls the datalayer via ctrlX:s REST inter-
face
  and publishes info about the state of 2 specific axes using
  rostopics 'axis_x' and 'axis_y'
grade: devel # must be 'stable' to release into candidate/stable channels
confinement: strict #has to be strict to work with ctrlX resrictions

parts:
  missing-lib:
    plugin: nil
    override-build: | #trying to add the persistently non-present li-
brary...
      snapcraftctl build
      ln -sf /usr/lib/x86_64-linux-gnu/libpsm_infini-
path.so.1 $SNAPCRAFT_PART_INSTALL/

  workspace:
    plugin: catkin
    include-roscore: true
    catkin-ros-master-uri: 'http://192.168.1.1:11311' #Needs to be explic-
itly set here
    source: .
    catkin-packages: [test]
    after: [missing-lib] # run after adding the missing-lib

  # dump the contents of ./configs to $SNAP folder
  # trying to add the sidepanel.
  configs:
    plugin: dump
    source: ./configs

apps:
  run:
    command: opt/ros/melodic/bin/roslaunch test test.launch
    plugs: [network, network-bind, network-control, home]
    environment:
      # ignore python warnings
      # used here due to the excessive ssl-warning-generation when poll-
ing ctrlX
      PYTHONWARNINGS: ignore
```

```
      #ROS_LOG_DIR: opt/ros/melodic/
    daemon: simple # ROS app will now run on startup
    restart-condition: always # always try to restart the app af-
ter crash etc.

# trying to add the sidepanel
slots:
  package-assets:
    interface: content
    content: package-assets
    source:
      read:
        - $SNAP/package-assets/${SNAPCRAFT_PROJECT_NAME}

plugs: #something to do with allocating space for the snap to use...
  active-solution:
    interface: content
    content: solutions
    target: $SNAP_COMMON/solutions
```

LIITE 6

[Gazebo-simulaation käynnistävän ROS-sovelluksen .launch-tiedosto]

```
<launch>
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find simulation)/world/simulation.world"/>
    <arg name="gui" value="true"/>
    <arg name="paused" value="true"/>
  </include>

  <!--generate an URDF-description from the xacro-file and store to ROS parameter server-->
  <param name="robot_description" command="$(find xacro)/xacro '$(find simulation)/urdf/block.xacro'"/>

<!--spawn a robot based on the generated URDF-->
  <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" output="screen" args="-param robot_description -urdf -model block"/>

  <!--load the .yaml configurations of the joint controller-->
  <rosparam file="$(find simulation)/config/config.yaml" command="load"/>

  <!—spawning joint-controllers according to the .yaml specs-->
  <node name="controller_spawner" pkg="controller_manager" type="spawner" ns="/palikka" args="joint_1_position_controller joint_2_position_controller joint_state_controller"/>

  <!--Has to do with data exchange between gazebo and ros-->
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" respawn="false" output="screen">
    <remap from="/joint_states" to="/palikka/joint_states"/>
  </node>
```

```
    <!--add the node that publishes joint position information based on info from ctrlX-
ROS publisher-->
    <node name="provider" pkg="simulation" type="controller.py">
    </node>

</launch>
```