Sami Sarlin

ROS-KÄYTTÖJÄRJESTELMÄ ROBOTIIKAN POIMINTASOVELLUKSESSA

Opinnäytetyö
CENTRIA-AMMATTIKORKEAKOULU
Sähkö- ja automaatiotekniikan koulutus
Maaliskuu 2020

TIIVISTELMÄ OPINNÄYTETYÖSTÄ

Centria-ammattikorkeakoulu	**Aika** Maaliskuu 2020	**Tekijä/tekijät** Sami Sarlin
Koulutusohjelma Sähkö- ja automaatiotekniikka		
Työn nimi ROS-KÄYTTÖJÄRJESTELMÄ ROBOTIIKAN POIMINTASOVELLUKSESSA		
Työn ohjaaja Joni Jämsä		**Sivumäärä** 32 + 7
Työelämäohjaaja Tero Kaarlela		

Opinnäytetyössä tutkittiin ROS-käyttöjärjestelmän hyödyntämistä robotiikan poimintasovelluksessa. Tavoitteena oli kehittää työkappaleen tunnistamiseen ja poimintaan soveltuva robotiikan sovellus, jota on mahdollista hyödyntää Centria-ammattikorkeakoulun kehityshankkeissa. Työssä pyrittiin myös huomioimaan yhteistyörobotiikkaa koskevat turvallisuusvaatimukset.

Työn tilaajana oli Centria-ammattikorkeakoulu, joka toimii partnerina TRINITY-hankkeessa. Hankkeen tavoitteena on parantaa valmistavan teollisuuden tuotantoprosessien ketteryyttä ja tuottavuutta mm. digitaalisten teknologioiden ja robotiikan hyödyntämisellä teollisissa ympäristöissä.

Työ suoritettiin Centria-ammattikorkeakoulun tuotantotekniikan laboratoriossa. Sovelluksen testaamiseen käytettiin Universal Robots UR10 -yhteistyörobottia, jonka ohjelmointialustana käytettiin ROS-käyttöjärjestelmää ja Python-ohjelmointikieltä. ROS-käyttöjärjestelmän käsittelyssä käytetty materiaali koottiin käyttöjärjestelmän omilta kotisivuilta, ROS wiki -ohjesivustolta sekä alan kirjallisuudesta. Yhteistyörobotiikan käsittelyssä hyödynnettiin alan kirjallisuutta sekä internet-aineistoa.

Työn lopputuloksena saatiin kappaleen sijainnin tunnistamiseen ja poimintaan soveltuva robotiikan sovellus, jota on mahdollista käyttää jatkossa Centria-ammattikorkeakoulun tutkimus-, kehitys- ja innovaatiotoiminnassa. Lisäksi työssä saatiin yleiskuva ROS-käyttöjärjestelmän ja eri ohjelmointikielien yhteiskäytöstä robottien ohjelmoinnissa.

Asiasanat
Python, Robotiikka, ROS, TRINITY

Centria
UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Centria University of Applied Sciences	**Date** March 2020	**Author** Sami Sarlin

Degree programme
Electrical and automation engineering

Name of thesis
IMPLEMENTATION OF ROS IN A ROBOTIC PICK-AND-PLACE APPLICATION

Instructor Joni Jämsä	**Pages** 32 + 7

Supervisor
Tero Kaarlela

The purpose of this thesis was to study and research the implementation of ROS in a robotic pick-and-place application. The goal was to develop a robotic application suitable for recognizing and handling of workpieces, and which can later be integrated into other Centria R&D projects. In addition, the safety regulations of collaborative robots were also taken into consideration.

The work was commissioned by Centria, a cooperative partner in the TRINITY project. The goal of the project was to improve agility and product rates in manufacturing industries by implementing digital technologies and robotics into industrial environment.

The work was carried out in the industrial engineering laboratory of Centria University of Applied Sciences. The Universal Robots UR10 collaborative robot was used in the testing of the application with ROS operating system as the programming platform and Python as the main programming language. The material concerning ROS was collected from the operating system's own website, ROS wiki information bank and additional literature. With collaborative robotics, literature and internet sources regarding the field were utilized.

The end result was a robotic application suitable for handling and recognizing the position of certain objects. The application is also suitable for later implementation in Centria R&D department. A general overview of combining ROS with different programming languages in robot programming was also established.

Key words
Python, Robotics, ROS, TRINITY

KÄSITTEIDEN MÄÄRITTELY

Apache-lisenssi	Vapaan ohjelmiston lisenssi, joka vaatii tekijänoikeushuomautuksen ja erottamislausuman
BSD-lisenssi	Berkeley Software Distribution, vapaa ohjelmistolisenssi
IoT	Internet of things, esineiden internet
LTS	Long Time Support, pitkän aikavälin tuki
Master	Isäntä, ROS-käyttöjärjestelmän sisäinen nimipalvelin
Node	Noodi, ROS-käyttöjärjestelmässä laskentaa suorittava prosessi
RGBD	Väri- ja syvyyskuvan yhdistelmä
ROS	Robot Operating System, robottien ohjelmointiin suunniteltu avoimen lähdekoodin käyttöjärjestelmä
Topic	Aihe, ROS-käyttöjärjestelmässä kahden tai useamman noodin välinen viestintäväylä
URDF	Unified Robot Description Format, robotin ja käyttöympäristön kuvaukseen tarkoitettu spesifikaatio
XML	Extensible Markup Language, merkintäkielien standardi, jolla tiedon merkitys on kuvattavissa tiedon sekaan

TIIVISTELMÄ
ABSTRACT
KÄSITTEIDEN MÄÄRITTELY
SISÄLLYS

KUVIOT

KUVAT

1 JOHDANTO

Opinnäytetyön aiheena oli tutkia ROS-käyttöjärjestelmän käyttöä robotiikan poimintasovelluksessa. Työn tilaajana oli Centria-ammattikorkeakoulu, joka on partnerina TRINITY-hankkeessa. Tavoitteena oli tuottaa sovellus, jossa robotti kykenee poimimaan määritetyt kappaleet mistä tahansa työpöydän rajaamalta alueelta ja siirtämään ne toiseen, ennalta määritettyyn paikkaan. Työn aihe saatiin lehtori Joni Jämsältä.

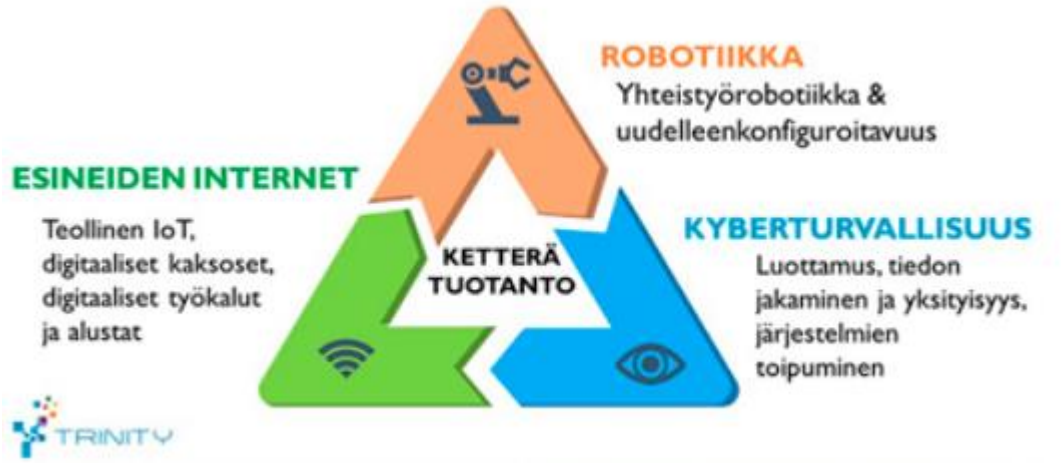
Työ aloitettiin perehtymällä ROS-käyttöjärjestelmään ja sen työkaluihin. Tietoa järjestelmän käytöstä sekä sen toiminnasta ja työkaluista on saatavilla käyttöjärjestelmän omasta ROS wiki -ohjepankista ja alan kirjallisuudesta. Lisäksi hankittiin tarvittavat laitteet, kuten kannettava tietokone ja Microsoftin valmistama Kinect v1 -liikeohjain, jota tässä työssä käytettiin RGBD-sensorina. Sovelluksen robottina käytettiin Universal Robots UR10 -yhteistyörobottia, joka saatiin käyttöön Centria-ammattikorkeakoulun Kokkolan yksiköltä. Tarttujia ja lisälaitteita hyödynnettiin tuotantotekniikan laboratorion resurssien mukaan.

Käytännön toteutuksen kannalta oleellista oli liittää robotti ROS-käyttöjärjestelmään ja luoda RGBD-sensoria käyttäen työalueesta syvyyskartta sekä luoda robotille etäkäyttöohjelma syvyyskartan tietojen perusteella. Etäkäyttöohjelman tuli löytää kappaleen sijainti määrätyssä koordinaatistossa ja syöttää robotille liikekäskyt näiden koordinaattien mukaan käyttöjärjestelmän työkaluja käyttäen. Toteutuksessa pyrittiin käyttämään valmiita ja vapaasti ladattavissa olevia käyttäjäyhteisön luomia ohjelmistopaketteja, joita muokattiin tarpeellisissa määrin toteutuksen kannalta sopiviksi. Lopputuloksena saatiin toiminnallinen sovellus, joka täyttää sille asetetut tavoitteet ja joka on laajennettavissa myös muihin sovelluskohteisiin.

Opinnäytetyö esittelee ROS-käyttöjärjestelmän käyttöä ja työkaluja sekä käsittelee ROS:in käytännön soveltamista robotiikan sovellukseen Python-ohjelmointikieltä ja käyttöjärjestelmän sisäisiä työkaluja ja ohjelmistopaketteja hyödyntäen. Myös yhteistyörobottien turvallisuusmääräykset pyrittiin ottamaan huomioon.

2 TRINITY-HANKE

TRINITY on Tampereen yliopiston koordinoima tutkimus- ja kehityshanke. Hankkeen tavoitteena on parantaa valmistavan teollisuuden tuotantoprosessien ketteryyttä ja tuottavuutta mm. digitaalisten teknologioiden ja robotiikan hyödyntämisellä teollisissa ympäristöissä. Hankkeen avainasemassa on kehittynyt robotiikka, jonka tukena käytetään digitaalisia teknologioita, IoT-ratkaisuja sekä kyberturvallisuuteen vaikuttavia tekijöitä (KUVIO 1). (TRINITY 2019.)



KUVIO 1. TRINITY-hankkeen fokusalueet (TRINITY 2019).

Hankkeen toimintamalliin kuuluvat digitaalisten innovaatiokeskittymien verkoston rakentaminen, sovellusesimerkkien tarjoaminen sekä digitaalisen yhteistyöalustan luominen. Verkoston tavoitteena on auttaa yrityksiä hyödyntämään digitaalisia teknologioita ja robotiikkaa sekä löytämään palveluntarjoajia ja yrityskumppaneita näiden ratkaisujen toteuttamiseen ja rahoitukseen. Verkosto koostuu tutkimuslaitoksista, yrityksistä, palveluntuottajista ja koulutusympäristöistä. Sovellusesimerkkejä tarjoamalla luodaan yhteys valmistavaan teollisuuteen ja osoitetaan robotiikan mahdollisuudet tuotantoprosessien ketteryyden kannalta. Digitaalisen kehitysalustan luominen tukee yhteistyötä, verkostoitumista ja tiedon jakamista alan eurooppalaisten tutkijoiden ja teollisuuden välillä. (TRINITY 2019.)

TRINITY-hanke on saanut rahoituksen EU:n Horizon 2020 -ohjelmasta. Ohjelma rahoittaa kansainvälisesti toteutettavia tutkimushankkeita yhteensä lähes 80 miljardilla eurolla vuosien 2014-2020 aikana. Ohjelman tavoitteena on kehittää Euroopan Unionin tutkimus- ja innovaatio-osaamista sekä parantaa

eurooppalaisen teollisuuden kestävyyttä ja kilpailukykyä. Tavoitteella pyritään vastaamaan tuleviin yhteiskunnallisiin haasteisiin, tukemaan uusien teknologisten ratkaisujen ja innovaatioiden käyttöönottoa sekä edistämään Euroopan Unionin talouskasvua. (Euroopan Komissio 2014, 5-7.)

3 YHTEISTYÖROBOTIIKKA

Yhteistyörobotti (cobot, collaborative robot, yhteistoimintarobotti) on robotti, joka on suunniteltu toimimaan vuorovaikutuksessa ihmisen kanssa samassa työtilassa. Yhteistoiminnasta puhutaan, kun robotti jakaa ihmisen kanssa saman työtilan ja ihminen ja robotti ovat yhteistoiminnassa keskenään. Yhteistyörobotiikan sovelluksen eroja perinteiseen robotiikan sovellukseen verrattuna ovat mm. korostuneet turvallisuustekijät sekä robotin helppo käytettävyys ja ohjelmoitavuus. Vuonna 2018 yhteistyörobottien osuus Suomessa hankituista uusista roboteista oli yli 10 %, mikä on selitettävissä yhteistyörobottien ohjelmointikynnyksen mataluudella. Erityisosaamista robotin ohjelmointiin ei tarvita, joten yhteistyörobotin hankinta nähdään siten matalan riskin investointikohteena. (Lempiäinen 2019, 9; Bélanger-Barrette 2015.)

3.1 Standardit, määritelmät ja turvallisuus

Teollisuusrobotteja ja niiden yhteistoimintaominaisuuksia koskevat turvallisuusvaatimukset määritellään standardeissa SFS-EN ISO 10218-1 ja SFS-EN ISO 10218-2. On huomioitava, että standardi SFS-EN ISO 10218-1 käsittelee yksinomaan robottia ja sen turvallisuustekijöitä, ja standardi SFS-EN ISO 10218-2 käsittelee robottijärjestelmiä sekä niiden yhdistelmiä ja turvallisuutta. Edellä mainittujen standardien lisäksi yhteistyörobottien turvallisuusmääräyksiä täydennetään teknisessä määrittelyssä ISO/TS 15066, joka tarjoaa ohjeistusta yhteistoimintasovelluksen riskien arviointiin ja niiden hallintaan. Erilaisia riskienhallintakeinoja yhteistoimintasovelluksissa ovat mm. turva-alueiden käyttö, toiminta-alueen esteettömyys ja turvallisuus sekä robotin voiman, massan ja nopeuden rajoittaminen. (Shea 2020.)

Standardeissa SFS-EN ISO 10218-1 ja -2 määritellään yhteistyörobotteihin liittyviä käsitteitä seuraavasti:

1) Yhteistoimintarobotti: Robotti, joka on suunniteltu suoraan vuorovaikutukseen ihmisen kanssa määritetyssä yhteistyötilassa.
2) Yhteistyötila: Turvalaitteen suojaamassa tilassa oleva työtila, jossa robotti ja ihminen voivat suorittaa tehtäviä samanaikaisesti tuotantotehtävien aikana.
3) Yhteistoiminta: Toimintatila, jossa tähän tarkoitukseen suunnitellut robotit työskentelevät ihmisten kanssa välittömässä yhteistyössä määritetyssä työtilassa. (SFS-EN ISO 10218-1, 12; SFS-EN ISO 10218-2, 8.)

Teollisuusrobotti määritellään standardissa SFS-EN ISO 10218-1 siten, että se on automaattisesti ohjattu, uudelleen ohjelmoitava, teollisuuden automaatiosovelluksissa käytettäväksi tarkoitettu monikäyttöinen käsittelylaite, jonka akseleista vähintään kolme on ohjelmoitavissa. Käsittelylaite voi olla joko kiinteästi asennettu tai liikkuva. Teollisuusrobotin määritelmään sisältyvät tämä käsittelylaite toimilaitteet mukaan luettuna sekä ohjauslaite, mukaan luettuna kannettava ohjelmointilaite ja tiedonsiirron rajapinnat, kuten ulkoiset tiedonsiirtolaitteistot ja niiden ohjelmistot. Teollisuusrobottijärjestelmä kattaa käsitteenä teollisuusrobotin lisäksi robotin työkalut sekä kaikki koneistot, laitteistot, laitteet, ulkopuoliset apuakselit ja anturit, jotka tukevat tehtävää suorittavaa robottia (KUVA 1). (SFS-EN ISO 10218-1, 12, 14.)

KUVA 1. Teollisuusrobotti osana teollisuusrobottijärjestelmää.

Standardin SFS-EN ISO 10218-1 mukaan yhteistoimintarobottia pidetään teollisuusrobottina. Yhteistoimintaroboteille on kuitenkin olemassa edellä mainitussa standardissa määritellyt erilliset vaatimukset. Robotissa on oltava näkyvä osoitus robotin olemisesta yhteistoiminnassa ja robotin on noudatettava vähintään yhtä seuraavista vaatimuksista, jotka tässä esitetään tiivistettyinä:

1) Turvaluokiteltu valvottu pysäytys: Robotin on pysähdyttävä, kun ihminen on yhteisessä työtilassa. Pysäytystoiminnon on pysäytettävä kaikki robotin liike, poistettava teho robotin toimilaitteilta ja aiheutettava järjestelmässä olevien mahdollisten muiden vaarojen lakkaaminen. Robotti voi jatkaa automaattista toimintaa, kun ihminen poistuu yhteisestä työtilasta.
2) Käsin ohjaaminen: Käsinohjauksen laitteisto on sijoitettava lähelle robotin työkalua ja siinä on oltava vaatimusten mukainen kolmiasentoinen sallintalaite ja hätäpysäytin. Robotin on käsinohjattuna toimittava nopeuden valvonta aktivoituna. Nopeusraja määritetään riskien arvioinnin avulla.

3) Nopeuden ja vähimmäisetäisyyden valvonta: Robotin on ylläpidettävä määritettyä nopeutta ja vähimmäisetäisyyttä käyttäjästä. Nämä toiminnot voidaan toteuttaa kiinteillä osilla tai robotin ohjausyksikön sisääntuloihin kytkettävillä laitteilla tai niiden yhdistelmällä.
4) Tehon ja voiman rajoittaminen luontaisesti turvallisella suunnittelulla tai ohjauksella: Robotin tehon tai voiman on oltava riskinarvioinnin mukaisesti rajoitettu. Robotin on pysähdyttävä, jos jokin raja-arvoista ylittyy. (SFS-EN ISO 10218-1, 28, 36.)

Lisäksi standardissa SFS-EN ISO 10218-2 määritellään yhteistoiminnan järjestelmäkohtaisia vaatimuksia, joiden tarkoituksena on varmistaa operaattorin ja robotin läheisyydessä työskentelevien henkilöiden turvallisuus. Robottijärjestelmälle on tehtävä integraattorin toimesta sovelluskohtainen riskin arviointi, jossa tarkastellaan yhteistyötilan turvallisuuteen vaikuttavia tekijöitä, kuten esteettömyyttä, ergonomiaa sekä turvatoimintoihin liittyviä suoritustasokriteereitä. Yhteistyötilaan liitettyjen robottien ja turvalaitteiden tulee olla vaatimusten mukaisia ja tilan on oltava merkitty selkeästi, esim. lattiamerkinnöillä ja kilvillä. (SFS-EN ISO 10218-2, 38-39.)

3.2 Hyödyt ja sovelluskohteet

Perinteiset teollisuusrobotit ovat tyypillisesti olleet raskaita, nopeita ja voimakkaita, mikä on tuonut turvallisuuden kannalta haasteita robotiikan integroinnissa valmistavaan teollisuuteen. Perinteisen teollisuusrobotiikan turvaratkaisuissa on pyritty minimoimaan tai eliminoimaan robotin ja ihmisen kontakti kokonaan esim. turva-aidoilla tai valoverhoilla. Yhteistyörobotiikkaa hyödyntämällä voidaan kuitenkin parantaa robotiikan käyttöastetta yrityksessä, sillä kehittyneiden turvaratkaisujen myötä robotille ei tarvita erillistä solua tai aidattua tilaa, mikä saattaa olla pk-yrityksissä ratkaiseva tekijä investointikustannusten ja tilankäytön kannalta. Oikein suoritetulla riskien arvioinnilla saadaan yhteistoimintaa muokattua edelleen joustavammaksi, tehokkaammaksi ja turvallisemmaksi. (Lumme 2019, 16-17; Malm & Salmi 2019, 13.)

Yhteistyörobottien mukautuvaisuus ja helppo ohjelmoitavuus ovat turvallisuuden ohella niiden suurimpia etuja. Ohjelmointi ja käyttöönotto on toteutettu yhteistyörobottien käyttöliittymissä mahdollisimman suoraviivaisesti, mikä on investointikynnyksen madaltamisen lisäksi eduksi myös tuotannon jatkuvuuden kannalta seisokkiaikojen jäädessä lyhyiksi. Yhteistyörobotit ovat kevytrakenteisia, mikä mahdollistaa niiden nopean uudelleensijoittamisen toiseen tuotantotehtävään parantaen investoinnin hyötysuhdetta. (Universal Robots 2020a.)

Yhteistyörobotiikkaa voidaan pitkälti käyttää samoissa sovelluskohteissa kuin perinteistä teollisuusrobotiikkaa. Tyypillisiä yhteistyörobotiikan sovelluskohteita ovat Universal Robots -robottivalmistajan mukaan mm. pakkaus- ja palletointisovellukset sekä kappaleenkäsittely ja työstökoneen palvelu, mutta yhteistyörobotiikkaa voidaan hyödyntää nimensä mukaisesti myös ihmisen kanssa yhteistoiminnassa. Malm & Salmi VTT:ltä esittelevät Automaatioväylä 06/2019 -lehden artikkelissaan kolme erilaista yhteistoimintatasoa: collaboration (läheinen yhteistoiminta), cooperation (vuorotteleva yhteistoiminta) ja coexistence (ei normaalisti yhteistoimintaa). Valtaosa Universal Robotsin internet-sivuilla esitellyistä sovelluskohteista kuuluisivat näin ollen cooperation- ja coexistence-kategorioihin, sillä sovellusten yhteistoiminnan määrä on vaihtelevaa tai sitä ei normaalitilanteessa ole. Tällaisia sovelluskohteita ovat esim. laboratorioanalyysit, kokoonpano- ja asennustehtävät sekä laadunvalvonta. (Malm & Salmi 2019, 13-14; Universal Robots 2020b.)

Yhdysvaltalainen ajoneuvovalmistaja Ford Motor Company hyödyntää yhteistyörobotiikkaa Craiovan kaupungissa sijaitsevalla tehtaallaan Romaniassa. Tehtaalla on käytössään neljä Universal Robots UR10 -yhteistyörobottia, joista kolme on sijoitettu tuotantolinjalle ja yksi testaus- ja koulutuskäyttöön. Tuotantolinjalla automatisoituja tehtäviä ovat venttiilinnostimien rasvaus, moottorin täyttäminen öljyllä sekä mahdollisten vuotokohtien tutkiminen ultraviolettilampulla ja robottiin kytketyllä kameralla. Tuotantolinjan automatisointi on parantanut työntekijöiden ergonomiaa ja vapauttanut työntekijöitä muihin tehtäviin tuotantolinjalla. (Universal Robots 2020c.)

4 ROS

Robot Operating System (ROS) on avoimen lähdekoodin käyttöjärjestelmä, joka on suunniteltu robottien ohjelmointiin. Käyttöjärjestelmän esiaste ja runko kehitettiin 2000-luvun alussa Stanfordin yliopistossa Yhdysvalloissa Stanford AI Robot (STAIR) -projektin pohjalta. Vuonna 2007 projektiin osallistunut Willow Garage tarjosi projektille resursseja jatkokehitykseen ja käytännön implementointiin. Näihin aikoihin ROS oli alkanut saada sille tunnusomaisia elementtejä, kuten nykyisten kaltaisten ohjelmistotyökalujen ja -pakettien hyödyntämisen sekä korkean laajennettavuusasteen. ROS perustuu vapaisiin BSD- ja Apache 2.0 -ohjelmistolisensseihin, mikä mahdollistaa ohjelmiston jälleenkäytön, jakelun ja muokkaamisen ja siten laajan hyödynnettävyyden, kehitettävyyden ja yhteisöllisyyden. ROS on nykyään laajassa käytössä mm. robotiikan tutkimuslaboratorioissa ja kehittämishankkeissa. (Cho, Jung, Lim & Pyo 2017, 15-16; ROS 2020.)

ROS-Industrial (ROS-I) on kokoelma ROS-yhteensopivia kirjastoja, työkaluja ja ajureita, jotka helpottavat teollisuusautomaation ja teollisuusrobotiikan käyttöä ROS-käyttöjärjestelmällä. ROS-I:n tavoitteena on kehittää luotettavia, teollisuuden käyttöön sopivia ohjelmistopaketteja, yhdistää ROS:n vahvuudet olemassa oleviin teollisuuden ratkaisuihin ja luoda teollisuusrobotiikan ammattiosaajien tukema käyttäjäyhteisö. ROS-I tarjoaa ajureita ja URDF-malleja mm. ABB-, Fanuc-, Motoman- ja Universal Robots -roboteille. (ROS Wiki 2019a.)

4.1 Rakenne ja käsitteet

ROS-käyttöjärjestelmää voidaan luonnehtia metakäyttöjärjestelmäksi, sillä se on riippuvainen isäntäkäyttöjärjestelmän arkkitehtuurista ja työkaluista, kuten prosessinhallinnasta ja tiedostojärjestelmästä. ROS vaatii siis toimiakseen isäntäjärjestelmäksi tavanomaisen käyttöjärjestelmän, jonka avulla suoritetaan ROS:in oman tiedostojärjestelmän hallinta sekä robotin hallinnan kannalta tärkeitä prosesseja, kuten tiedonsiirtoa, aikataulutusta sekä virheiden hallintaa. Isäntäjärjestelmä voi olla esim. Windows, Android tai Linux-pohjainen Ubuntu-käyttöjärjestelmä. (Cho ym. 2017, 10-11.)

ROS-käyttöjärjestelmän konsepti ja rakenne perustuvat kolmeen tasoon, joita ovat tiedostojärjestelmätaso (file system level), laskentataso (computation graph level) ja yhteisötaso (community level). Tiedostojärjestelmätaso käsittää käyttöalustan massamuistista löytyvät tiedostot ja niiden väliset suhteet,

laskentataso käsittää järjestelmien sisäisten prosessien välisen verkoston ja yhteisötaso käsittää ROS:in käyttäjä- ja kehittäjäyhteisöjen välisen tuki- ja ohjelmistoverkoston. (ROS Wiki 2014.)

4.1.1 Tiedostojärjestelmätaso

Tiedostojärjestelmätaso kattaa käsitteenä ohjelmointialustan massamuistista löytyvät ROS:in tarvikkeet ja tiedostot. Tällaisia ovat esim. ohjelmistopaketit (packages), viestityypit (msg), palvelutyypit (srv), sekä ohjelmistopaketteihin liittyvät xml-tiedostot eli manifestit, jotka tarjoavat ROS:in kääntäjälle tietoa mm. paketin ohjelmistoriippuvuuksista ja käytetystä ohjelmistolisenssistä. (ROS Wiki 2014.)

Viesti- ja palvelutyypit ovat ROS-käyttöjärjestelmän sisäisten viestintäväylien spesifikaatioita, jotka merkitään msg- ja srv-tiedostopäätteillä. Viestityypin tehtävä on kertoa viestintäväylän käyttäjälle, minkä tyyppistä dataa väylässä käsitellään, ja se sisältää tavallisesti viestin tietotyypin määritelmän (esim. merkkijono tai integeri) sekä tietotyypin määrittelemän muuttujan. Palvelutyyppi on rakenteeltaan samanlainen kuin viestityyppi, mutta siinä voi olla vaatimuksia käyttäjän syöttämille tiedoille, joita käytetään muissa ohjelmatiedostoissa esim. robotin digitaalilähtöjen ohjaamiseen. Kuvassa 2 on esimerkki robotin tilatiedon käyttämästä viestityyppirakenteesta (KUVA 2). (Cho ym. 2017, 62-63; ROS Wiki 2014.)

```
1   int8 NO_CONTROLLER=-1
2   int8 DISCONNECTED=0
3   int8 CONFIRM_SAFETY=1
4   int8 BOOTING=2
5   int8 POWER_OFF=3
6   int8 POWER_ON=4
7   int8 IDLE=5
8   int8 BACKDRIVE=6
9   int8 RUNNING=7
10  int8 UPDATING_FIRMWARE=8
```

KUVA 2. Esimerkki viestityyppirakenteesta.

ROS-käyttöjärjestelmässä käytettävät ohjelmistot jaetaan ohjelmistopaketteina, jotka voidaan ajatella sovellusta täydentävinä kokonaisuuksina. Yhdessä kehitetyssä sovelluksessa voi olla käytössä useita ohjelmistopaketteja ja yhtä ohjelmistopakettia voidaan hyödyntää useassa sovelluskohteessa, mikä mahdollistaa ohjelmistopakettien laajan käytettävyyden ja moduulimaisen hyödyntämisen. Tyypillinen ROS-ohjelmistopaketti sisältää ohjelmatiedostojen käyttämät viesti- ja palvelutyyppien määritelmät, muissa ohjelmistopaketeissa käytettävät lähdekooditiedostot, suoritettavat ohjelmatiedostot sekä ROS:in kääntäjän konfigurointitiedostot ja manifestit. (Cho ym. 2017, 42.)

4.1.2 Laskentataso

Laskentataso on järjestelmässä tietoa käsittelevien prosessien vertaisverkosto ja ROS-käyttöjärjestelmän ydin. Se koostuu laskentaa suorittavista prosesseista eli noodeista (node), noodien välisistä viestintäväylistä eli aiheista (topic), viesteistä (message), palveluista (service), ja noodien välisen kommunikaation mahdollistavasta nimipalvelimesta eli isännästä (master). Sovellusta kehitettäessä laskenta- ja hallintatehtävät jaetaan useamman noodin kesken, kuten esim. digitaalilähtöjen ohjaus, servomoottorien ohjaus, navigointi tai graafinen simulointi. Tämä parantaa sovelluksen stabiiliutta ja modulaarisuutta, jolloin yhden noodin vikaantuminen ei vaikuta koko järjestelmän toimintaan. (ROS Wiki 2014.)

Usein on tarpeen hyödyntää yhden noodin käsittelemää tietoa toisen noodin laskentatehtävässä. ROS:issa noodien välinen kommunikaatio hoidetaan julkaisija-tilaaja (publisher-subscriber) -periaatteella, jolloin julkaisijanoodi julkaisee käsittelemäänsä dataa toisen noodin käyttöön. Näiden kahden noodin välistä tiedonsiirtoväylää kutsutaan aiheeksi (topic). Ideaalitilanteessa julkaisijanoodi ja tilaajanoodi eivät ole toisiinsa suoraan yhteydessä, mikä mahdollistaa prosessien lopettamisen ja aloittamisen ilman vertaisriippuvuutta eikä siten aiheuta tahattomia virheilmoituksia. Aiheiden käyttö helpottaa myös sovelluskehitystä, kun tiedetään, mistä tietyn tyyppistä dataa voidaan ryhtyä etsimään. (ROS Wiki 2014.)

Palveluita (service) käytetään, kun on tarpeen hyödyntää kaksisuuntaista liikennettä kahden noodin välisessä tiedonsiirrossa tiedon tarpeen ollessa kertaluontoista tai synkronista tiedonsiirtoa tarvittaessa. Noodien välinen yhteys katkeaa siis välittömästi palveluprosessin jälkeen, mikä vähentää vertaisverkoston kuormitusta. Palveluita käytettäessä hyödynnetään ohjelmistopakettien sisältämiä palvelutyyppejä, joissa määritellään vaatimukset käyttäjän tai asiakasnoodin asettamille argumenteille ja palvelunoodin tarjoamalle vastaukselle. Kuvassa 3 on esimerkki robotin ulkoisten lähtöjen ohjaukseen tarkoitetusta

palvelutyypistä, jossa määritellään vaadittavat ulkoisia lähtöjä kuvaavat muuttujat sekä palvelunoodin antama palaute palveluprosessin onnistumisesta (KUVA 3). (Cho ym. 2017, 51.)

```
38  int8 fun
39  int8 pin
40  float32 state
41  ---
42  bool success
```

KUVA 3. Esimerkki palvelutyyppirakenteesta.

Laskentatason viestiliikennettä hallitaan ROS:n sisäisellä nimipalvelimella eli isännällä (master). Isännän tehtävänä on ylläpitää ja päivittää noodien rekisteröimiä nimi-, viesti- ja palvelutyyppitietoja, jotta noodien välinen viestiliikenne ohjautuu oikeaan paikkaan vertaisverkossa. Ilman isännän käynnistämistä noodien välinen kommunikaatio on siis mahdotonta. Käynnistyessään noodit rekisteröivät nimipalvelimelle tiedot järjestelmäteknisesti kevyttä ja useita ohjelmointikieliä tukevaa XMLRPC-protokollaa (XML-Remote Procedure Call) käyttäen, mikä edelleen parantaa ROS:in mukautuvaisuutta. XMLRPC-protokollaa käytettäessä viestiyhteyttä ei ylläpidetä noodin käynnistymisen jälkeen eikä noodien välistä yhteyttä tarkasteta, mikä vähentää vertaisverkon kuormitusta. (Cho ym. 2017, 41-42.)

4.1.3 Yhteisötaso

Yhteisötaso mahdollistaa ROS:in käyttäjä- ja kehittäjäyhteisöjen välisen ohjelmiston ja käytön tuen. Yhteisötaso muodostuu ROS-käyttöjärjestelmän jakeluversioista, yhteisöjen ylläpitämistä lähdekoodisäiliöistä (repository), ROS Wiki -ohjepankista, ROS Answers -kysymyspalstasta sekä vastaavista tukipalveluista, joihin yhteisön käyttäjät voivat olla yhteydessä. Tällaisiin tukipalveluihin lukeutuvat mm. ROS Blog -blogi sekä käyttöjärjestelmän sähköpostilista. (ROS Wiki 2014.)

ROS-käyttöjärjestelmän jakeluversioiden kehitys kulkee rinnakkain Linux-käyttöjärjestelmän jakeluversioiden kanssa, minkä tavoitteena on vakauttaa ROS:in ohjelmistopohjaa ja tarjota käyttäjille ohjelmistotukea koko jakeluversion elinkaaren ajan. Sopivan ROS-jakeluversion valinta perustuukin esim. Linux Ubuntu -käyttöjärjestelmää käytettäessä pitkälti käyttäjän suosimaan Ubuntu-jakeluversioon.

ROS-jakeluversioita on saatavilla kaksitoista kappaletta, joista kaksi ovat tällä hetkellä täyden ohjelmistotuen alaisia LTS (Long Time Support) -jakeluversioita: ROS Kinetic Kame ja ROS Melodic Morenia. ROS Kinetic Kamen ohjelmistotuki on määrä päättyä v. 2021 ja ROS Melodic Morenian v. 2023, koska niitä tukevien Linuxin jakeluversioiden ohjelmistotuet päättyvät. (ROS Wiki 2014.)

4.2 Käyttöönotto

ROS-käyttöjärjestelmän asennus ja käyttöönotto aloitetaan sopivan jakeluversion valinnalla. Tällä hetkellä ohjelmistotuen alaiset ROS-jakeluversiot jakautuvat siten, että ROS Kinetic Kame -jakeluversio on suunniteltu yhteensopivaksi Ubuntu 16.04 LTS -jakeluversion kanssa ja vastaavasti ROS Melodic Morenia Ubuntu 18.04 -version kanssa. Tässä työssä käytettiin Ubuntu 16.04 -jakeluversiota ja siten ROS -jakeluversioksi valikoitui ROS Kinetic Kame, jonka käyttöönottoa tässä alaluvussa käsitellään. Käyttöönottoprosessi käsittää käyttöjärjestelmän asentamisen ja työtilan konfiguroimisen.

Varsinaisen käyttöjärjestelmän asennus aloitetaan lisäämällä Linuxin paketinhallintatyökalulle määrittely ROS-pakettivarastosta, jotta käyttöjärjestelmän lataus ja asennus helpottuvat. Tämä on helpoin suorittaa komennolla *$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'*. Lisäksi määritellään paketinhallintatyökalulle pakettivaraston tunnistamiseen tarkoitettu numero- ja kirjainsarja eli avain, joka on saatavilla pakettivaraston haltijalta tai ohjesivustolta. Avaimen määritys tehdään komennolla *$ sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key avain*, jossa *avain* korvataan numero- tai kirjainsarjalla. (ROS Wiki 2019b.)

Kun paketinhallintatyökalun ja avaimen määrittely on tehty, voidaan käyttöjärjestelmä ladata ja asentaa ROS-pakettivarastosta komennolla *$ sudo apt-get install ros-kinetic-desktop-full*. Kyseinen komento asentaa ROS:in lisäksi myös hyödylliset rqt- ja rviz-visualisointiohjelmat sekä yleisiä robotiikan ja navigoinnin kirjastoja. Lisäksi käynnistetään ja päivitetään ROS:in ytimen ohjelmistoriippuvuuksia käsittelevä rosdep-toiminto komennoilla *$ sudo rosdep init* ja *$ rosdep update* sekä asennetaan ohjelmistopakettien käsittelyyn tarvittavat kirjastot ja työkalut komennolla *$ sudo apt install python-rosinstall python-rosinstall-generator python-wstool build-essential*. Lopuksi määritellään Linuxin komentotulkille ROS:in ympäristömuuttujien päivitys komennolla *$ echo ”source /opt/ros/kinetic/setup.bash” >*

~/.bashrc, jolloin ROS:in asetustiedostot ja parametrit ladataan aina uuden komentorivi-ikkunan avautuessa. Ympäristömuuttujia ovat esim. ROS:in juuri- ja pakettihakemistojen tiedot. (Cho ym. 2017, 25-27; ROS Wiki 2019b.)

Käyttöjärjestelmän asennuksen jälkeen tulee käyttäjän määritellä ja luoda järjestelmälle työtila (workspace), joka toimii ROS-ohjelmistopakettien muokkaus- ja säilytysalustana. Työtilaa varten luodaan kansio ja src-niminen alikansio komennolla *$ mkdir -p ~/catkin_ws/src*, minkä jälkeen siirrytään luotuun työtilan kansioon komennolla *$ cd catkin_ws* ja suoritetaan työtilan kääntö komennolla *$ catkin_make*. Kääntämisen tarkoituksena on luoda ohjelmistopaketeista lähdekoodin perusteella toiminnallisia ohjelmistoja. Lopuksi määritellään työtilan ympäristomuuttujien päivitys käyttöjärjestelmää vastaavalla tavalla, komennolla *$ echo "source /home/kayttajanimi/catkin_ws/devel/setup.bash" > ~/.bashrc*. ROS:in ohjelmistopolun voi tarkistaa komennolla *$ echo $ROS_PACKAGE_PATH*, joka palauttaa määritellyt ohjelmistopakettien juuripolut. Palautteen tulee olla muotoa */home/kayttajanimi/catkin_ws/src:/opt/ros/kinetic/share*. (ROS Wiki 2019b.)

Työtilan kääntämisen yhteydessä kääntäjä luo työtilakansioon kaksi kansiota, jotka ovat build ja devel. Build-kansiossa säilytetään kääntäjän välimuistitiedot ja muut välittäjätiedostot, ja devel-kansiossa säilytetään käännetyt ohjelmistot ennen niiden asentamista. Devel-kansio toimii hyvänä kehitysalustana, koska ohjelmistoja ei tarvitse kääntää jokaisen muokkauksen jälkeen, mikä nopeuttaa ohjelmistojen kehitystyötä ja lisää järjestelmän vikasietoisuutta. Asennusvaiheessa luotu src-kansio sisältää ohjelmistopaketteihin liittyvät lähdekooditiedostot ohjelmistopakettikohtaisissa kansioissaan. Tyypillisessä asennusprosessissa ohjelmistopaketin lähdekooditiedostot kopioidaan ulkoisesta pakettivarastosta työtilan src-kansioon, minkä jälkeen työtila käännetään komennolla *$ catkin_make*. (ROS Wiki 2017.)

4.3 Komennot ja työkalut

Tässä alaluvussa esitellään ROS-käyttöjärjestelmän käytön ja sovelluskehityksen kannalta muutamia hyödyllisiä komentoja, työkaluja ja kirjastoja. Esiteltävät työkalut ja kirjastot ovat käyttöjärjestelmän asennuspaketissa valmiiksi toimitettuja, joten ne ovat valmiita käytettäväksi ilman erillisiä asennustoimenpiteitä. Esiteltävät komennot helpottavat sovelluskehitystä ja ROS:in tiedostojärjestelmässä navigointia sekä laskentatason prosessien ymmärtämistä.

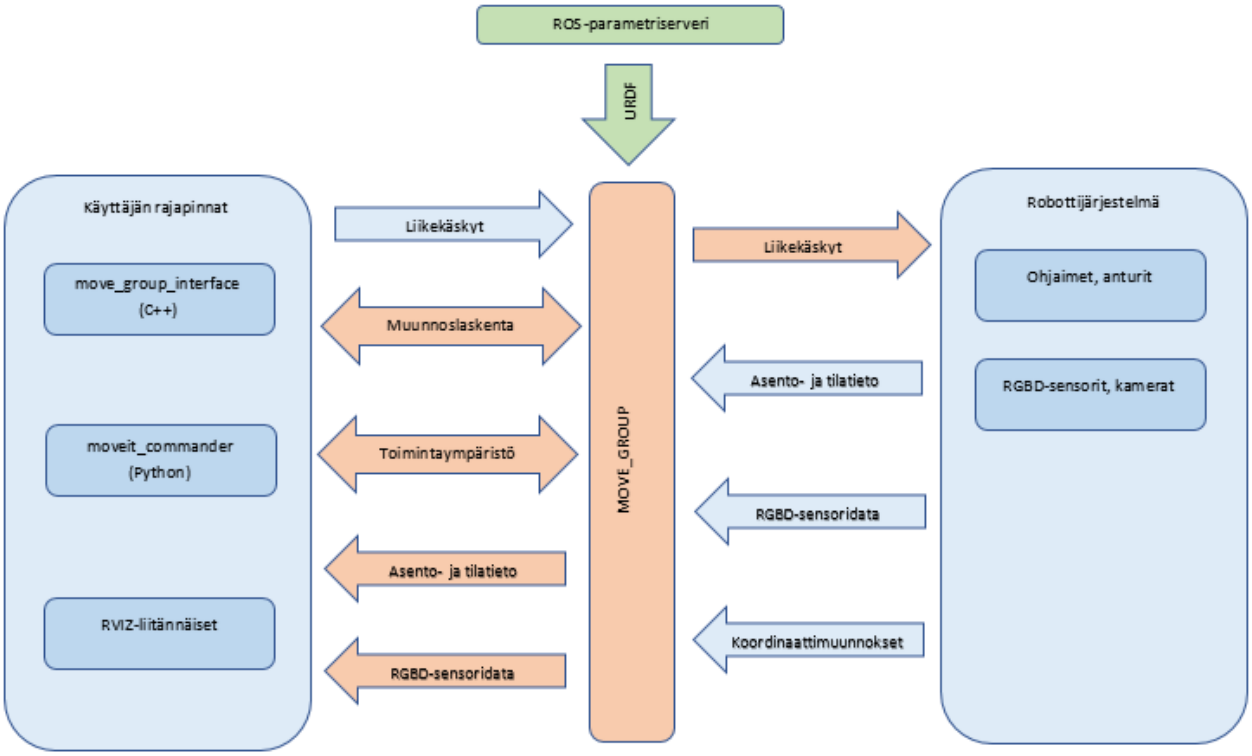
ROS:in keskeisiä suorittavia komentoja ovat *$ roscore, $ rosrun* ja *$ roslaunch*. Komento *$ roscore* käynnistää järjestelmän sisäisen nimipalvelimen eli isännän ja on siten suoritettava ensin ROS:ia käytettäessä. Isännän lisäksi komento käynnistää käyttöjärjestelmän lokikirjoittimen ja parametriserverin, jota käytetään noodien ja palveluiden ohjauksessa. Komennolla *$ rosrun* voidaan käynnistää komennon yhteydessä määritelty yksittäinen noodi, eli komento on muotoa *$ rosrun ohjelmistopaketti noodin_nimi*. Useita noodeja käynnistettäessä ja käynnistyksen yhteydessä noodeille tehtävää parametrointia varten käytetään komentoa *$ roslaunch*, joka käynnistää ohjelmistopaketin launch-kansiossa olevan määritellyn launch-tiedoston. Launch-tiedoston käynnistyskomennon syntaksi on samankaltainen *$ rosrun*-komennon kanssa: *$ roslaunch ohjelmistopaketti launch_tiedosto.launch*. (Cho ym. 2017, 95-98.)

Usein on tarpeen käsitellä laskentatason prosessien ja palveluiden keskinäisiä suhteita. Tätä varten ROS:issa on muutamia keskeisiä laskentatason tarkasteluun liittyviä komentoja, joista tässä tarkastellaan muutamia tärkeitä: *$ rostopic, $ rosservice* ja *$ rosnode*. Tarkastelukomennon nimestä voidaan päätellä komennon käyttökohde, esim. *$ rostopic* käsittelee aiheita, *$ rosservice* palveluita ja *$ rosnode* noodeja. Kunkin tässä käsiteltävän tarkastelukomennon syntaksi on likipitäen muotoa *$ komento lisämääre tarkastelukohde*, jossa *komento* on tarkastelukomento, *lisämääre* on komennolla suoritettava toiminto tai haettava informaatio ja *tarkastelukohde* on tarkasteltava kohde, esimerkiksi aiheen tai noodin nimi. Esimerkiksi komennolla *$ rostopic echo /esimerkkiaihe* saadaan tulostettua esimerkkiaihe-nimisessä aiheessa kulkeva tietovirta, komennolla *$ rosnode kill /esimerkkinoodi* pysäytetään esimerkkinoodi-niminen noodi ja komennolla *$ rosservice info /esimerkkipalvelu* saadaan tulostettua esimerkkipalvelu-nimisen palvelun tiedot, kuten palvelutyyppi. Lisätietoa komennoista ja lisämääreistä on saatavilla ROS Wiki -ohjepankista. (Cho ym. 2017, 99-107.)

ROS:in sovelluskehityksen kannalta tärkeitä työkaluja ja kirjastoja ovat mm. rospy, tf, rviz ja MoveIt!. Rospy on asiakaskirjasto, joka tarjoaa kehitysympäristön Python-ohjelmointikielen käyttöön noodien ohjelmoinnissa. Tf on koordinaatistojen hallintaan tarkoitettu kirjasto, jolla hoidetaan mm. robotin kinemaattisen ketjun määrittely ja paikoitetaan robottijärjestelmään kuuluvat laitteet (esim. kamerat) oikeaan sijaintiin ja orientaatioon maailmankoordinaatistossa. Rviz on ympäristön 3D-visualisointiin tarkoitettu työkalu, jota voidaan käyttää apuna esim. robotin mallintamisessa tai RGBD-sensorin tuottaman datan visualisoinnissa. (Cho ym. 2017, 47, 66, 129.)

MoveIt! on käyttöjärjestelmään integroitu kirjasto, joka sisältää työkaluja ja menetelmiä teollisuusrobottien ja robottikäsivarsien hallintaan. Kirjastoon tarjoamiin työkaluihin kuuluvat mm. suoran ja käänteisen kinematiikan laskenta, toimintaympäristön muokattavuus sekä liikekäskyjen laskenta, suunnittelu

ja toteutus. Kirjastoon sisältyvä move_group-noodi on yhteydessä robotin ohjaimiin, ROS:in parametriserveriin sekä käyttäjän valitsemaan rajapintaan ja se vastaanottaa tietoa mm. robotin tilasta, nivelten asentotiedosta, robotin URDF-mallista sekä käyttäjän määrittelemistä liikekäskyistä ja käyttöympäristöstä (KUVIO 2). Käyttäjä voi siis move_group-noodin välityksellä määritellä robotille liikekäskyjä sekä suunnitella ja toteuttaa ne noodin kommunikaatioketjun avulla. (Cho ym. 2017, 429-430.)



KUVIO 2. Move_group-noodin kommunikaatioketju (mukaillen Cho ym. 2017, 429).

5 SOVELLUS

Tässä luvussa esittelen työssä kehitetyn sovelluksen, sovelluksessa käytetyt laitteet ja ROS-ohjelmistopaketit sekä kuvaan sovelluksen ja kirjoitetun Python-ohjelman toimintaa. Sovellukseen kirjoitettu Python-ohjelmakoodi on kokonaisuudessaan nähtävillä liitteessä 1 (LIITE 1). Työssä pyrittiin käyttämään mahdollisimman paljon käyttäjäyhteisön rakentamia ja jakamia ohjelmistopaketteja, joita muokattiin sovelluksen tarpeisiin nähden sopiviksi. Työssä hyödynnetty laitteisto oli kokonaisuudessaan Centria-ammattikorkeakoulun omaisuutta eli ulkoisia laitehankintoja ei tarvinnut tehdä.

Työ aloitettiin laitteiston hankinnalla ja ROS-käyttöjärjestelmään ja sen työkaluihin perehtymisellä. Tämän jälkeen alettiin etsiä ja konfiguroida sovelluksen tarpeisiin sopivia ohjelmistopaketteja, joihin kuuluivat mm. RGBD-sensorin ajuri sekä robotin hallintaan tarvittavia kirjastoja. Lopuksi kirjoitettiin MoveIt!-kirjastoa käyttävä Python-ohjelma, johon määriteltiin robotin liikekäskyt ja tunnistetun kappaleen aseman hakeminen koordinaatistossa. Ohjelma testattiin lukuisia kertoja ohjelmointivaiheen aikana ja ohjelma todettiin toimivaksi. Lopputuloksena syntyi toiminnallinen sovellus, joka täyttää sille asetetut vaatimukset.

5.1 Yleiskuvaus

Työssä kehitettiin sovellus, joka kykenee määrittämään ohjelmistolle opetetun esineen aseman määrätyssä koordinaatistossa ja siirtämään esineen toiseen, ennalta määriteltyyn paikkaan teollisuusrobotilla. Robottina käytettiin Universal Robots UR10 -yhteistyörobottia, joka oli kiinnitetty pyörillä varustetun teollisuuspöydän kanteen pulteilla ja jonka ohjausyksikkö oli kiinnitetty teollisuuspöydän alaosaan. Tämä mahdollisti laitteiston helpon siirrettävyyden ja siten kehitystyön joustavuuden. Yhteistyörobottia käyttämällä varmistettiin myös kehitystyön turvallisuus mm. hyödyntämällä ohjelmallisesti luotuja turvarajoja sekä liikenopeuden valvontaa. Robotin ja tietokoneen välille muodostettiin tiedonsiirtoyhteys ethernet-kaapelin välityksellä ja yhteys katkaistiin testauksen ulkopuolisina aikoina, millä osaltaan estettiin liikeohjelman tahaton käynnistyminen.

Sovelluksen kehitysalustana käytettiin kannettavaa tietokonetta, johon asennettiin Linux Ubuntu 16.04- ja ROS Kinetic Kame -käyttöjärjestelmät. Sovelluksen kamerana ja RGBD-sensorina käytettiin Microsoft Kinect v1 -liikeohjainta, joka kiinnitettiin teipillä alumiiniprofiilipalkkiin, joka edelleen kiinnitettiin

ruuvipuristimilla robotin alustana toimineeseen teollisuuspöytään. Järjestelmäratkaisut pyrittiin pitämään väliaikaisina ja helposti purettavina koska oli tiedossa, että robotti on käytössä vain rajatun ajan testaustarkoituksessa. Tämä ei kuitenkaan oleellisesti vaikuttanut työn suoritukseen. Kuvassa 4 on nähtävissä yleiskuva järjestelmästä (KUVA 4).

KUVA 4. Yleiskuva järjestelmästä.

Tarttujaratkaisuksi valittiin alipainetarttuja, joka oli varustettu 3D-tulostetulla tartuntapinnalla kaksisormitarttujaa varten. Tällöin robotin mukana toimitettua kaksisormitarttujaa ja siihen liittyvää voima-anturia ei tarvinnut irroittaa testausvaiheen ajaksi. Paineilman ohjaus toteutettiin 24 V:n tasajännitteellä ohjattavilla magneettiventtiileillä. Alipainetarttujaan päädyttiin, sillä se soveltui parhaiten kappaleentunnistuksen kannalta helppojen työkappaleiden käsittelyyn: testausvaiheessa käytetyt työkappaleet ovat

tasaisia ja niiden kulloinkin näkyvä pinta on työalueen pinnan suuntainen, mikä parantaa kappaleen tunnistuksen mahdollisuutta myös eri kulmista katsottuna. Työssä käytetty tarttuja on esitetty tarkemmin kuvassa 5 (KUVA 5).

KUVA 5. Työssä käytetty tarttujaratkaisu.

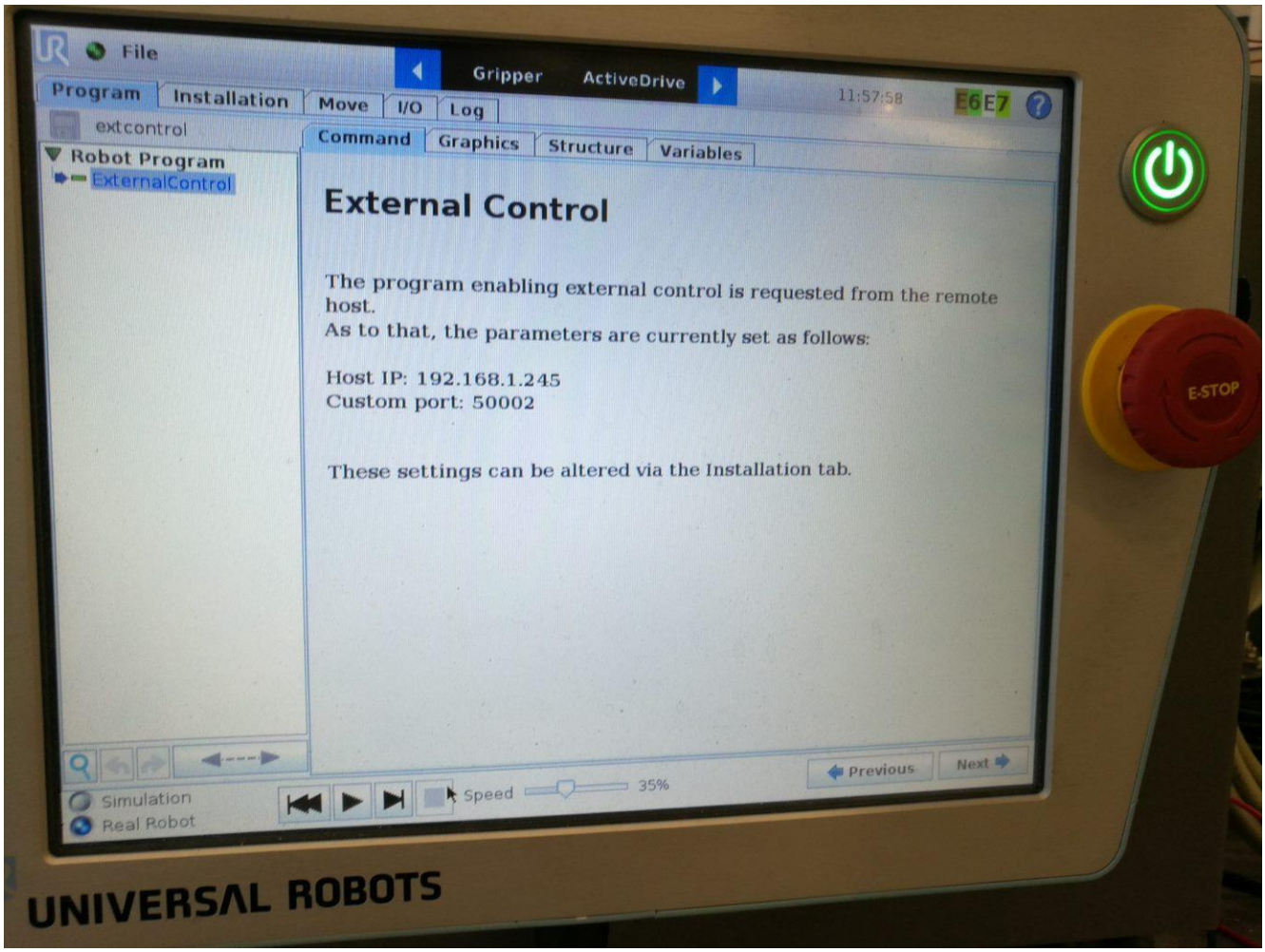
5.2 Hyödynnetyt paketit ja ohjelmistot

Käytännön toteutuksen kannalta keskeisiä ohjelmistopaketteja ovat laitteiston tukeen liittyvät ohjelmistot ja ajurit, joita tässä tapauksessa olivat kameran sekä robotin tukipaketit ja ajurit. Kameran ajurina käytettiin Microsoftin Kinect v1 -kameraa tukevaa Freenect-kirjastoa, joka ladattiin ja asennettiin Linuxin paketinhallintatyökalua hyödyntäen. Freenect-kirjaston mukana tuleva launch-tiedosto käynnistää

suuren määrän */camera*-etuliitteisiä noodeja, jotka alkavat julkaista kuvatietoa */camera*-etuliitteisiin aiheisiin. Esimerkkejä kameran noodien julkaisemista aiheista ovat mm. */camera/rgb/image_color*, */camera/depth_registered/image_raw* ja */camera/rgb/camera_info*. Näistä aiheista on saatavilla kameran väri- ja syvyyskuva sekä kameran metatiedot, joita käytetään kappaleen tunnistuksessa.

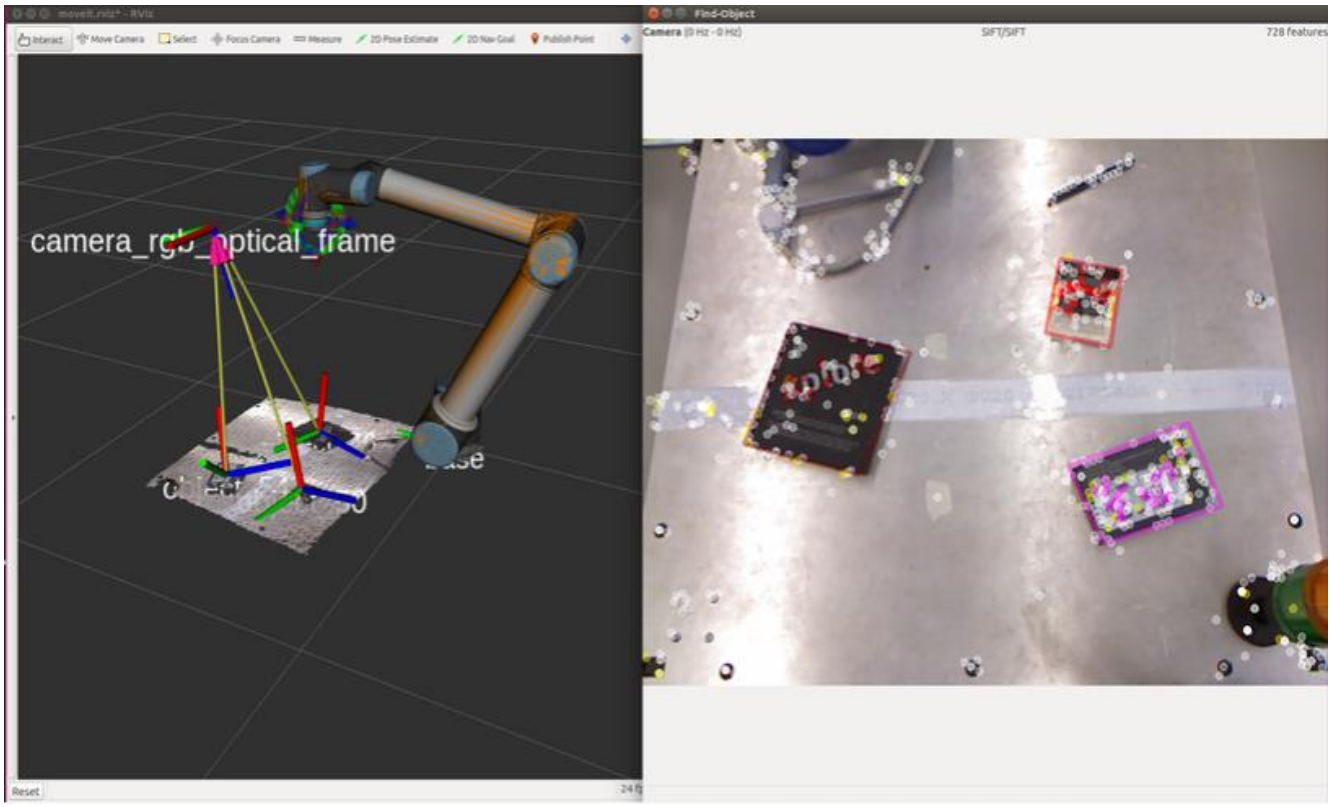
Universal Robots UR10 -tukipaketti ladattiin github.com-osoitteesta, ROS-Industrialin ylläpitämästä *universal_robot*-pakettivarastosta. Tukipaketti sisältää ajurin ja MoveIt!-kirjaston edellyttämät URDF-tiedostot kaikille Universal Robots -roboteille. Tukipaketin sisältämä ajuri oli jo kuitenkin vanhentunut eikä näin ollen tukenut robotissa käytettyä Polyscope 3.11 -ohjelmistoversiota, joten ajuri jouduttiin vaihtamaan. Universal Robots on kuitenkin julkaissut virallisen ROS-yhteensopivan ajurin, joka ladattiin yhtiön ylläpitämästä pakettivarastosta. Tämä paketti sisälsi robotin kalibrointiin käytettävät tiedostot sekä viesti- ja palvelutyypit ja ajurin, joka on yhteensopiva robotissa käytetyn ohjelmistoversion kanssa. Ohjelmistopaketin tärkeimpiä noodeja ovat robotin tilatietoa lukeva ja robotin komentokantaa päivittävä */ur_hardware_interface*-noodi sekä robotin kalibrointitiedot lukeva */ur_calibration*-noodi, joka on syytä ajaa ajuria käynnistäessä mahdollisten asento- ja tilatietopoikkeamien eliminoimiseksi.

Robotin etäohjauksen mahdollistamiseksi tuli robotin käyttöjärjestelmään asentaa *ExternalControl*-lisäpaketti (URCap), joka robotin pääohjelmassa ajettuna mahdollistaa liikekäskyjen ajamisen käytetyllä Universal Robots -ajurilla. Käytännössä *ExternalControl*-lisäpaketin toiminta perustuu Universal Robots -roboteissa käytettyyn RTDE-rajapintaan (Real-Time Data Exchange), joka mahdollistaa synkronoidun tiedonsiirron robotin sekä ulkoisen ohjainlaitteiston välillä. Kun lisäpaketti ajetaan robotin pääohjelmassa, se alkaa RTDE-rajapintaa hyödyntäen suorittaa URScript-muotoista ohjelmakoodia, jota päivitetään jatkuvasti ROS-ympäristössä. Tähän perustuu myös työssä käytetyn ajuripaketin toiminta. Lisäpakettiin määritellään käynnistyksen yhteydessä ulkoisen ohjainlaitteiston IP-osoite ja käytetty portti eli numeroitu palvelupiste (KUVA 6).



KUVA 6. ExternalControl-lisäpaketin näkymä robotin käyttöliittymässä.

Kappaleen tunnistamiseen ja sijainnin määrittämiseen käytettiin kanadalaisen Mathieu Labben ylläpitämää Find-object -ohjelmistopakettia. Find-object -ohjelmistopaketti perustuu avoimen lähdekoodin konenäköohjelmistoon nimeltä OpenCV, ja se tarjoaa graafisen käyttöliittymän yksinkertaisiin konenäköratkaisuihin OpenCV:n tunnistus- ja kuvausalgoritmeilla. Ohjelmistopakettiin sisältyvä */find_object_3d*-noodilla asetetaan tilaus sopiviin */camera*-etuliitteisiin aiheisiin, joista haetaan väri- ja syvyyskuvatiedot sekä kameran tiedot. Käyttäjä rajaa käyttöliittymän avulla kameran värikuvasta työkappaleen, jonka havaitessaan */find_object_3d*-noodi julkaisee kappaleen tunnus- ja sijaintitiedon */objects*-aiheeseen ja määrittää kappaleen sijainnin ja asennon edellä mainitulle tf-koordinaatistotyökalulle. Tästä aiheesta saatavia tietoja hyödynnetään työkappaleen poiminnan liikeratoja määritettäessä. Kuvassa 7 on nähtävissä Find-object -käyttöliittymä sekä esimerkki tf-koordinaattimuunnoksesta (KUVA 7).



KUVA 7. Find_object -käyttöliittymä ja visualisoitu esimerkki tf-koordinaattimuunnoksesta.

Työkappaleen koordinaattien noutamista sekä robotin liikekäskyjen laskentaa ja suoritusta varten hyödynnettiin rospy- ja moveit_commander -asiakaskirjastoja, joilla saatiin luotua ohjelmistotason rajapinnat ROS:in laskentatasoon ja MoveIt!-kirjastoon Python-ohjelmointikielellä. Rospy-kirjaston funktioilla voidaan esim. asettaa tilaus aiheeseen ja käyttää aiheessa julkaistua dataa ohjelman sisäisessä laskennassa, kuten tässä työssä tehtiin */objects*-aiheen osalta. *Moveit_commander*-kirjaston luokilla ja funktioilla voidaan esim. ohjelmallisesti muokata robotin käyttöympäristöä, määrittää robotille ohjelmapisteet ja liikeohjelma sekä lukea robotin tilatietoja, kuten aikaisemmin on havainnollistettu sivuilla 14-15.

5.3 Ohjelmointi

Robottijärjestelmän ohjausta varten luotiin Python-ohjelmointikielellä ohjelma, johon määritettiin robotin ohjelmapisteet, työkappaleen koordinaattien nouto sekä robotin liikekäskyistä koostuva pääohjelma. Tässä alaluvussa esittelen työhön kehitettyä Python-ohjelmaa ja ohjelmakoodiin liittyviä ratkaisuja, jolla edellä mainitut asiat saatiin toteutettua. Ohjelmakoodi kokonaisuudessaan on saatavilla liitteessä 1 (LIITE 1).

Python-ohjelmointikielessä koodiin sisältyvät kirjastot otetaan käyttöön koodin ensimmäisillä riveillä *import*-komennolla, jolloin saadaan käyttöön kaikki kirjaston sisältämät luokat ja funktiot. Samalla periaatteella voidaan ottaa käyttöön myös viesti- ja palvelutyyppejä, kuten esimerkiksi ROS-viestityypit *geometry_msgs* ja *moveit_msgs*. Tässä ohjelmassa otetaan yhteys ROS-laskentatasoon, joten *rospy*-kirjasto on otettava käyttöön. Lisäksi tarvitaan koordinaattimuunnoksia varten *tf*-kirjasto, liikekäskyjen laskentaa varten *moveit_commander*-kirjasto sekä *string*- ja *Float23MultiArray* -viestityyppejä, jotka saadaan otettua käyttöön *std_msgs*-kirjastosta. Kuvassa 8 on luettelo ohjelmassa käytetyistä kirjastoista (KUVA 8).

```
 3   import sys
 4   import copy
 5   import rospy
 6   import tf2_ros
 7   import tf
 8   import moveit_commander
 9   import moveit_msgs.msg
10   import geometry_msgs.msg
11   import math
12   from std_msgs.msg import String, Float32MultiArray
13   from moveit_commander.conversions import pose_to_list
```

KUVA 8. Ohjelmassa käytetyt kirjastot.

Python-ohjelmointikielessä on järkevää käyttää luokkia ohjelmakoodin yksinkertaistamiseksi ja laajennettavuuden kannalta. Luokan sisälle voidaan määrittää *def*-komennolla funktioita ja aliohjelmia, joita voidaan kutsua saman luokan muissa ohjelmissa, esim. pääohjelmassa. Luokan sisälle ensimmäisenä tuleva *__init__*-funktio alustaa luokan, ja sitä käytetään yleisesti ohjelmassa käytettävien muuttujien määrittelyyn ja perustietojen noutoon.

Tässä ohjelmakoodissa luotiin luokka nimeltä *UR10MoveItTest*, jonka sisään määriteltiin pääohjelmassa kutsuttavat funktiot ja aliohjelmat työkappaleen koordinaattien noutamista ja liikekäskyjen laskentaa varten. Luokan *__init__*-funktiossa käynnistetään *moveit_commander*-asiakaskirjasto sekä */moveittest*-noodi, jolla asetetaan myöhemmin tilaukset mm. edellä mainittuihin */tf*- ja */objects* -aiheisiin. Lisäksi funktiossa määritellään muutamia muuttujia, kuten koordinaattilaskennassa käytettävä *frame_id* ja luo-

daan luokkakutsu *group = moveit_commander.MoveGroupCommander(manipulator)* (LIITE 1). *MoveGroupCommander* on *moveit_commander*-kirjastoon sisältyvä liikekäskyjä käsittelevä luokka, joka toimii yhteytenä tietyn nivelsarjan suuntaan. *Manipulator* -argumentti kuvaa tässä yhteydessä UR10-robotin nivelsarjaa.

Robotin liikeohjelmaan sisällytettiin aloitusasema, johon se on turvallista paikoittaa työkappaleen koordinaattien käsittelyn ajaksi. Tämä saavutettiin määrittelemällä *go_to_joint_state()*-funktio, johon määriteltiin funktiokutsu *joint_goal = self.group.get_current_joint_values()*. Muuttujalla *joint_goal* kutsutaan *MoveGroupCommander*-luokasta *get_current_joint_values()*-funktiota, joka palauttaa tämänhetkiset robotin nivelten asentotiedot kuusialkioisena listana *joint_goal*-muuttujaan. Tämän listan alkiot voidaan määritellä uudelleen halutuilla asentotiedoilla, minkä jälkeen liikekäsky voidaan suorittaa komennolla *self.group.go(joint_goal, wait=True)*. *Wait*-parametri pysäyttää ohjelman suorituksen, kunnes haluttu asema on saavutettu. Lopuksi varmistetaan mahdollisen jäännösliikkeen lakkaaminen komennolla *self.group.stop()* (KUVA 9). Samalla periaatteella suoritetaan muiden liikekäskyjen laskenta, kun halutut asentotiedot ovat tiedossa.

```
67  def go_to_joint_state(self):
68
69      # In this function, we'll set joint goal values for init state and perform a joint move there:
70
71      joint_goal = self.group.get_current_joint_values()
72      joint_goal[0] = math.radians(273.90)
73      joint_goal[1] = math.radians(-116.06)
74      joint_goal[2] = math.radians(105.48)
75      joint_goal[3] = math.radians(-78.73)
76      joint_goal[4] = math.radians(-90.24)
77      joint_goal[5] = math.radians(184.37)
78
79      self.group.go(joint_goal, wait=True)
80
81      # Using stop() to ensure there's no residual movement
82      self.group.stop()
```

KUVA 9. Esimerkkifunktio alkuaseman määrittelyyn ja suorittamiseen.

Työkappaleen tunnuksen nouto suoritetaan ohjelmassa kahdella funktiolla: *pose_coord_node()* ja *frame_callback(data)*. Funktiolla *pose_coord_node()* asetetaan tilaus */find_object_3d*-noodin julkaisemaan */objects*-aiheeseen komennolla *rospy.Subscriber(”objects”, Float32MultiArray, self.frame_callback)*, jossa *Float32MultiArray* on aiheen viestityyppi ja *self.frame_callback* on viestin saapuessa kutsuttava funktio. *Frame_callback()*-funktiossa asetetaan aiemmin määritelty globaali *frame_id*-muuttuja */objects*-aiheesta saadun viestin mukaiseen arvoon ja palautetaan muun ohjelman käyttöön (KUVA 10).

Tällä menetelmällä saatuja työkappaleiden tunnuksia voidaan koordinaattilaskennassa käyttää juoksevasti, eikä niitä tarvitse erikseen merkitä ohjelmakoodiin.

```
84    def frame_callback(self, data):
85
86      global frame_id
87      frame_id = int(data.data[0])
88      return frame_id
89
90    def pose_coord_node(self):
91
92      rospy.Subscriber("objects", Float32MultiArray, self.frame_callback)
93      rospy.sleep(1)
```

KUVA 10. Työkappaleen tunnuksen määrittäminen.

Työkappaleen sijainnin selvittämiseksi määriteltiin *get_pose_coord*-funktio, jossa lasketaan koordinaattimuunnos kameran koordinaatistosta työkappaleen koordinaatistoon ja asetetaan x-y-z -koordinaatit *goal_coord*-muuttujaan, jota voidaan käyttää lähestymisaseman määrittelyssä. Aluksi määritellään luokkakutsu *listener = tf.TransformListener()* sekä kutsutaan *tf.TransformListener()*-luokasta *waitForTransform()*- ja *lookupTransform()*-funktioita. *WaitForTransform()*-funktiolla odotetaan muunnosta */object_%s*-koordinaatistosta */camera_rgb_optical_frame*-koordinaatistoon täsmällisellä ajan hetkellä (*rospy.Time()*) neljän sekunnin ajan (*rospy.Duration(4.0)*). *LookupTransform()*-funktiolla määritetään varsinainen koordinaattimuunnos kahden edellä mainitun koordinaatiston välillä ja asetetaan se muuttujiin *trans* ja *rot*, joista *trans* määrittää koordinaatiston sijainnin ja *rot* orientaation toisen koordinaatiston suhteen. Lopuksi tehdään *trans*-muuttujalle tarttujaan liittyviä muutoksia ja määritellään *goal_coord*-muuttuja (KUVA 11).

```
95    def get_pose_coord(self):
96
97      listener = tf.TransformListener()
98
99      listener.waitForTransform("/camera_rgb_optical_frame", "/object_%s" % frame_id, rospy.Time(), rospy.Duration(4.0))
100     (trans,rot) = listener.lookupTransform('camera_rgb_optical_frame', 'object_%s' % frame_id, rospy.Time())
101
102     goal_coord = [trans[0] - 0.020, trans[1] - 0.15, trans[2] - 0.34]
103
104     return goal_coord
```

KUVA 11. Työkappaleen sijainnin määrittäminen.

Kun työkappaleen lähestymispisteen sijainti koordinaatistossa on selvillä, voidaan tähän asemaan määrittää liikekäsky *pose_goal*-muuttujan avulla, kunhan referenssikoordinaatisto ja robotin työkalupiste ovat samassa *tf*-koordinaatistopuussa (LIITE 2). Tähän tarkoitukseen luotiin *go_to_pose_goal()*-funktio (KUVA 12). Aluksi määritellään referenssikoordinaatisto ja työkalupiste *MoveGroupCommander*-luokan *set_pose_reference_frame()*- ja *set_end_effector_link()*-funktioiden avulla sekä määritellään *goal_coord*-muuttuja aiemman *get_pose_coord()*-funktion palauttamaan arvoon. Lisäksi poistetaan varmuuden vuoksi mahdolliset muistiin jääneet aiemmat koordinaattipisteet, jotta ne eivät aiheuttaisi ongelmia liikekäskyn laskennassa.

```
106  def go_to_pose_goal(self):
107
108      self.group.set_pose_reference_frame('camera_rgb_optical_frame')
109      self.group.set_end_effector_link('tool0')
110      self.group.clear_pose_targets
111      goal_coord = self.get_pose_coord()
112
113      q = tf.transformations.quaternion_from_euler(-0.30, 0, 0)
114
115      pose_goal = geometry_msgs.msg.Pose()
116      pose_goal.orientation.x = q[0]
117      pose_goal.orientation.y = q[1]
118      pose_goal.orientation.z = q[2]
119      pose_goal.orientation.w = q[3]
120      pose_goal.position.x = goal_coord[0]
121      pose_goal.position.y = goal_coord[1]
122      pose_goal.position.z = goal_coord[2]
123      self.group.set_pose_target(pose_goal)
124
125      plan = self.group.go(wait=True)
126      self.group.stop()
127      self.group.clear_pose_targets()
```

KUVA 12. Esimerkkifunktio liikekäskyn suorittamiseen asematiedon perusteella.

MoveGroupCommander-luokan *set_pose_target()*-funktio vaatii ohjelmapisteen x-, y-, ja z -sijaintikoordinaatit sekä x-, y-, z-, ja w -orientaatiomuuttujat kvaterniomuodossa. Liikekäskylle on tehtävä orientaatiomuunnos, sillä käytetty koordinaatisto poikkeaa työalueen pinnan normaalista. Koska kvaternioiden käsin laskenta on huomattavan vaativaa, määritellään orientaatiomuuttujat *tf*-kirjaston *transformations*-luokan *quaternion_from_euler()*-funktion avulla, jolla suoritetaan muunnos radiaaneista kvaterniomuotoon. Alkiot tallennetaan listaan *q*, joista niitä voidaan käyttää orientaation määrittelyssä. Liikekäskyn suunnitteluun kutsutaan *geometry_msgs.msg.Pose*-viestityyppiä ja syötetään *q*- ja *goal_coord*-

alkiot listaan. Lopuksi liikekäsky tallennetaan ja suoritetaan sekä poistetaan muistista. Samalla periaatteella suoritetaan työkappaleen poiminta funktiolla *grasp_object()* (LIITE 1).

Lopuksi määritellään pääohjelma, jossa kutsutaan aiemmin luotuja funktioita oikeassa järjestyksessä ja alustetaan *UR10MoveItTest*-luokka muuttujaan *test*. Luokan *UR10MoveItTest* sisälle luotuja funktioita kutsutaan siten pääohjelmassa komennolla *test.funktion_nimi()*, esim. *test.go_to_joint_state()*. Funktiokutsut ovat kehystettynä *while*-looppiin siten, että ohjelma toistuu *rospy*-kirjaston ollessa käynnissä ja pysähtyy *rospy*-kirjaston tai */move_group*-noodin häiriön seurauksena, käyttäjän sulkiessa ohjelman näppäinkomennolla *ctrl+c* tai kun työalueella ei ole tunnistettavia työkappaleita.

5.4 Ohjelman suoritus ja toiminnan kuvaus

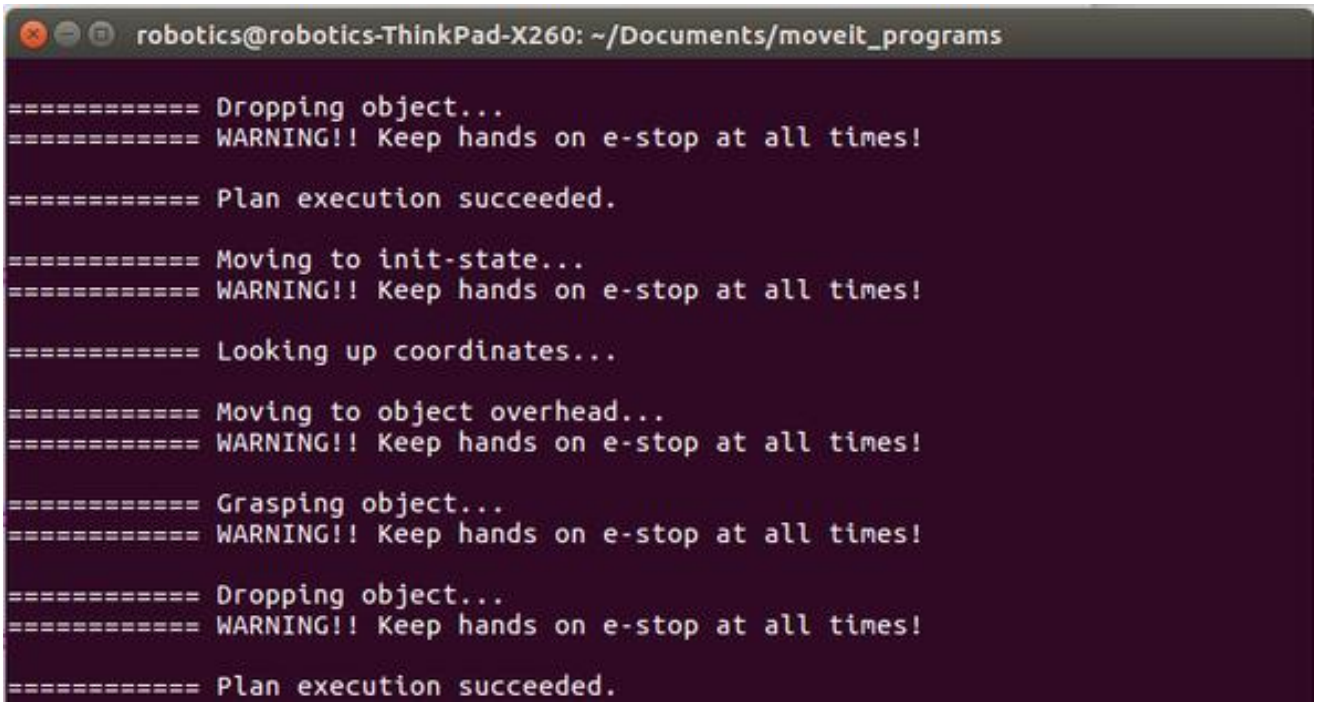
Ohjelman suoritusta varten on käynnistettävä noodit ja ohjelmistot, joista ohjelmakoodi on riippuvainen. Käynnistettävien noodien ja ohjelmistojen lista käsittää ROS-käyttöjärjestelmän nimipalvelimen, Kinect-kameran ajurit, */find_object_3d*-noodin sekä Universal Robots UR10 -robotin tukipaketteja, joihin kuuluivat robotin ajuri sekä MoveIt!-kirjaston edellyttämät konfiguraatiot. Noodien ja ohjelmistojen käynnistämiseen liittyvät komennot ja niiden selitteet ovat seuraavat:

1) *$ roscore* -komento käynnistää ROS-käyttöjärjestelmän nimipalvelimen eli isännän, joka mahdollistaa noodien välisen kommunikaation.
2) *$ roslaunch freenect_launch freenect.launch* -komento käynnistää Kinect-sensoria tukevat noodit, jotka julkaisevat kuvatietoa */camera*-aiheisiin.
3) *$ roslaunch ur_robot_driver ur10_bringup.launch robot_ip:=ip_osoite* -komento käynnistää UR10-robotin ajuripaketin ja mahdollistaa robotin ja ROS-ympäristön välisen kommunikaation. *Robot_ip* -parametrin käytöllä mahdollistetaan kehitysalustan käyttö useassa robotissa, mutta se on mahdollista asettaa myös tiettyyn oletusarvoon launch-tiedostossa.
4) *$ roslaunch ur10_moveit_config ur10_moveit_planning_execution.launch* -komento lataa UR10-robotin URDF-mallin MoveIt!-kirjastolle ja käynnistää */move_group*-noodin, jota käytetään ohjelmassa liikekäskyjen suunnitteluun.
5) *$ roslaunch ur10_moveit_config moveit_rviz.launch* -komento käynnistää *rviz*-ohjelman, johon visualisoidaan robotin malli ja toimintaympäristö.
6) *$ roslaunch find_object_2d find_object_3d.launch* -komento käynnistää */find_object_3d* -noodin ja mahdollistaa ohjelmiston käytön työkappaleen tunnistuksessa.

7) *$ python moveit_program.py* -komento käynnistää aiemmin määritetyn ohjelman, jossa suoritetaan esineiden tunnistus ja liikekäskyjen laskenta ROS-kirjastoja apuna käyttäen.

ROS-käyttöjärjestelmä sallii launch-tiedostorakenteiden yhdistämisen yhdeksi isoksi launch-tiedostoksi, mutta tässä yhteydessä pyrittiin noodien ja ohjelmistojen käynnistäminen pitämään hajautettuna. Tällä toimintamallilla voidaan tilapäisesti sulkea osa käytettävistä noodeista pitäen samalla järjestelmä toimintakuntoisena esim. pienten ohjelmakoodimuutosten aikana. Ohjelmaa suorittavien noodien ja aiheiden vertaisverkosto on esitetty liitteessä 3 (LIITE 3).

Ohjelma tulostaa käynnistyksen yhteydessä komentoriville tiedot referenssikoordinaatistosta, työkalupisteestä, liikeryhmästä sekä robotin tilasta. Tämän jälkeen alkaa varsinainen ohjelmakierto, jossa kutsutaan ohjelmaan määritettyjä funktioita oikeassa järjestyksessä ja tulostetaan komentoriville kulloinenkin työvaihe sekä tarvittaessa turvallisuuskehote (KUVA 13). Turvallisuuskehotteella pyrittiin minimoimaan esim. virheellisen liikekäskyn aiheuttaman vaaran esiintymisriski, vaikka robotin nopeutta ja voimaa rajoittavat turvajärjestelmät ovat edelleen käytössä ulkoisesta ohjauksesta huolimatta. Ohjelmakierto pysähtyy käyttäjän sulkiessa ohjelman tai kun työalueella ei ole tunnistettavia työkappaleita.



KUVA 13. Ohjelman suoritus komentorivillä.

Työssä kehitetty robottisovellus todettiin kehitysvaiheessa tehtyjen testiajojen perusteella toimivaksi. Robottisovelluksen runko ja ohjelmakoodi on helposti sovellettavissa myös muihin robotiikan sovelluskohteisiin ja tutkimus- ja kehityshankkeisiin pienillä parametrimuutoksilla. Sovellus on periaatteessa sellaisenaan siirrettävissä toisenlaisella robotilla suoritettavaksi, kunhan saatavilla on MoveIt!-kirjaston vaatimat konfiguraatiotiedostot, robotin ohjelmointikielelle sopiva kääntäjä eli ajuripaketti sekä robotin käyttöjärjestelmään asennettava, etähallinnan salliva ohjelmisto. Useimpien valmistajien robottimalleille on saatavilla etähallintaohjelmistoja, ja jotkut robottivalmistajat kehittävät omia ROS-tukipakettejaan, jotka sisältävät MoveIt!-konfiguraatiot sekä tarvittavat ajurit.

6 YHTEENVETO

Tässä opinnäytetyössä kehitettiin ROS-käyttöjärjestelmäympäristöllä ohjattu robottijärjestelmä ja sen osaksi robotisoitu poimintasovellus. Sovelluksessa suoritetaan työkappaleen tunnistus avoimen lähdekoodin konenäköohjelmistoa ja kuluttajatason RGBD-sensoria hyödyntäen sekä määritellään robotin liikekäskyt työkappaleen sijainnin perusteella. Robottisovelluksen tavoitteiksi asetettiin autonominen liikekäskyn suunnittelu ja toteutus sekä ohjausjärjestelmän helppo laajennettavuus tutkimus- ja kehityshankkeisiin sopivaksi. Työssä kehitetty robottijärjestelmä ja siihen kuuluva ohjausjärjestelmä täyttävät niille asetetut vaatimukset.

Työn alkuvaiheessa suoritettu ohjelmistopakettien valinta osoittautui arvioitua haasteellisemmaksi erityisesti työkappaleen tunnistamiseen liittyvän ohjelmiston osalta, ja ohjelmistopaketteja päädyttiin testaamaan huomattava määrä ennen sopivan ohjelmiston löytämistä, mikä vaikutti osaltaan työn kulkuun. Haasteita kohdattiin lisäksi Universal Robots -ajuripaketin toiminnassa digitaalilähtöjen ohjauksen osalta, sillä robotin ja tietokoneen välinen tiedonsiirtoyhteys katkesi aina ohjauskäskyä suoritettaessa. Tämä teki robotin digitaalilähtöjen ja siten myös alipainetarttujan ohjauksesta mahdotonta. Vika paikallistettiin ajurin kehittäjien avulla lopulta itse ajuripakettiin, ja paineilmaa ohjaavan magneettiventtiilin ohjaus päädyttiin tekemään tässä työssä kapasitiivisella rajakytkimellä. Tämä ei kuitenkaan oleellisesti vaikuttanut työn lopputulokseen, sillä työn kannalta ratkaisevassa asemassa oli liikekäskyn oikeaoppinen suoritus ja työkappaleen sijainnin määrittely.

Linux- ja ROS -käyttöjärjestelmien rakenteen ja käytön opettelu oli merkittävässä asemassa työn suorituksen kannalta. Olin aikaisemmin käyttänyt Linux-pohjaisia käyttöjärjestelmiä vain muutamia kertoja, ja ROS-käyttöjärjestelmä oli minulle rakenteeltaan täysin vieras. Käytinkin työn alkuvaiheessa edellä mainittujen käyttöjärjestelmien käytön opetteluun huomattavan määrän aikaa, mikä auttoi työssä tarvittavan tietopohjan saavuttamisessa ja helpotti työn etenemistä jatkossa. Sain työelämäohjaajalta Tero Kaarlelalta työn edetessä runsaasti vinkkejä ROS-käyttöjärjestelmän käyttöön sekä suosituksia ohjelmistopakettien konfiguroinnin osalta, mikä osaltaan edesauttoi työn etenemistä.

Työssä kehitetyssä Python-ohjelmakoodissa (LIITE 1) on muutama asia, jonka olisin voinut tehdä toisin. Esimerkiksi *frame_callback()*-funktion *return frame_id* -rivillä ei ole merkitystä, sillä *frame_id* on määritetty globaaliksi muuttujaksi eli se on joka tapauksessa käytössä koko luokan nimiavaruudessa. Muuttujaa ei siis näin ollen ole tarpeen erikseen palauttaa. Vastaavaksi olisin voinut määrittää muuttujan *q*

heti luokan *__init__*-alustusfunktiossa eli konstruktorissa, eikä sen määrittäminen joka funktiossa erikseen olisi ollut tarpeen. Käytännön toteutuksen kannalta tällaisilla seikoilla ei usein ole kovin suurta merkitystä, mutta ne vaikuttavat oleellisesti ohjelmakoodin luettavuuteen ja pidentävät ohjelmakoodin kirjoittamiseen käytettyä aikaa.

Kokonaisuudessaan robottijärjestelmän ja siihen liittyvän poimintasovelluksen kehitysprojekti sujui hyvin, ja työssä saatiin todella kattava yleiskuva ROS-käyttöjärjestelmän ja eri ohjelmointikielien yhteiskäytöstä robotiikan sovelluskehityksessä, mistä uskon hyötyväni ammatillisen osaamiseni ja tavoitteideni kannalta. Työn kuluessa saavutettiin erityisesti Python-ohjelmointikielestä hyvä tietopohja sekä kehitettiin ohjelmointityössä tarvittavia vianhaku- ja ongelmanratkaisutaitoja. Työssä kehitetystä robottijärjestelmästä ja -sovelluksesta on Centria-ammattikorkeakoulun tutkimus- ja kehityshankkeissa hyötyä, ja sovellusta on mahdollista jatkokehittää edelleen tutkimus- ja kehityshankkeisiin sopivaksi.

Yhteistyörobotiikan hyödyntäminen mahdollistaa entistä joustavammat tuotantoratkaisut teollisissa ympäristöissä, ja näiden ratkaisujen kehittämisessä ROS-käyttöjärjestelmä on avainasemassa sen modulaarisuuden ja laajan hyödynnettävyyden vuoksi. ROS-käyttöjärjestelmän kehitystyö ja tuki jatkuu edelleen, ja henkilökohtaisesti uskon sen edelleen vakiinnuttavan asemaansa robotiikan sovelluskehityksessä sekä tutkimustyössä.

LÄHTEET

Bélanger-Barrette, M. 2015. What Does Collaborative Robot Mean? Saatavissa: https://blog.robotiq.com/what-does-collaborative-robot-mean. Viitattu 28.1.2020.

Cho, H., Jung, R., Lim, T. & Pyo, Y. 2017. ROS Robot Programming. From the basic concept to practical programming and robot application. Seoul: ROBOTIS Co., Ltd.

Euroopan Komissio. 2014. Horizon 2020 lyhyesti. Luxemburg: Euroopan unionin julkaisutoimisto. Saatavissa: https://ec.europa.eu/programmes/horizon2020/sites/horizon2020/files/H2020_FI_KI0213413FIN.pdf. Viitattu 27.1.2020.

Lempiäinen, J. 2019. Robotti työllistyy – tilastot kertovat. Automaatioväylä 6, 8-11. Saatavissa: https://www.theseus.fi/bitstream/handle/10024/266543/Automaatiovayla_6_2019.pdf?sequence=1&isAllowed=y. Viitattu 29.1.2020.

Lumme, A. 2019. Cobotti jyrsii tuotantokuluja. Automaatioväylä 6, 16-18. Saatavissa: https://www.theseus.fi/bitstream/handle/10024/266543/Automaatiovayla_6_2019.pdf?sequence=1&isAllowed=y. Viitattu 29.1.2020.

Malm, T., Salmi, T. 2019. Yhteistyörobotit tulevat – oletko valmis? Automaatioväylä 6, 12-15. Saatavissa: https://www.theseus.fi/bitstream/handle/10024/266543/Automaatiovayla_6_2019.pdf?sequence=1&isAllowed=y. Viitattu 29.1.2020.

ROS Wiki. 2014. Concepts. Saatavissa: http://wiki.ros.org/ROS/Concepts. Viitattu 31.1.2020.

ROS Wiki. 2017. Catkin workspaces. Saatavissa: http://wiki.ros.org/catkin/workspaces. Viitattu 10.2.2020.

ROS Wiki. 2019a. Industrial. Saatavissa: http://wiki.ros.org/Industrial. Viitattu 31.1.2020.

ROS Wiki. 2019b. Ubuntu install of ROS Kinetic. Saatavissa: http://wiki.ros.org/kinetic/Installation/Ubuntu. Viitattu 8.2.2020.

ROS. 2020. History. Saatavissa: https://www.ros.org/history/. Viitattu 31.1.2020.

SFS-EN ISO 10218-1. Robotit ja robotiikkalaitteet. Turvallisuusvaatimukset. Osa 1: Teollisuusrobotit. 2011. Helsinki: Suomen standardoimisliitto SFS.

SFS-EN ISO 10218-2. Robotit ja robotiikkalaitteet. Turvallisuusvaatimukset. Osa 2: Robottijärjestelmät ja niiden yhdistelmät. 2011. Helsinki: Suomen standardoimisliitto SFS.

Shea, R. 2020. Collaborative Robot Technical Specification ISO/TS 15066 Update. Saatavissa: http://me.umn.edu/courses/me5286/robotlab/Resources/12-TR15066Overview-SafetyforCollaborativeApplications-RobertaNelsonShea.pdf. Viitattu 28.1.2020.

TRINITY. 2019. Trinity-esite Suomi. Saatavissa: https://projects.tuni.fi/uploads/2019/04/2b3127b7-trinity-esite-suomi.pdf. Viitattu 27.1.2020.

Universal Robots. 2020a. Why Cobots? Saatavissa: https://www.universal-robots.com/products/collaborative-robots-cobots-benefits/. Viitattu 29.1.2020.

Universal Robots. 2020b. Automate virtually anything. Saatavissa: https://www.universal-robots.com/case-stories/. Viitattu 30.1.2020.

Universal Robots. 2020c. Ford Motor Company. Saatavissa: https://www.universal-robots.com/case-stories/ford-motor-company/. Viitattu 30.1.2020.

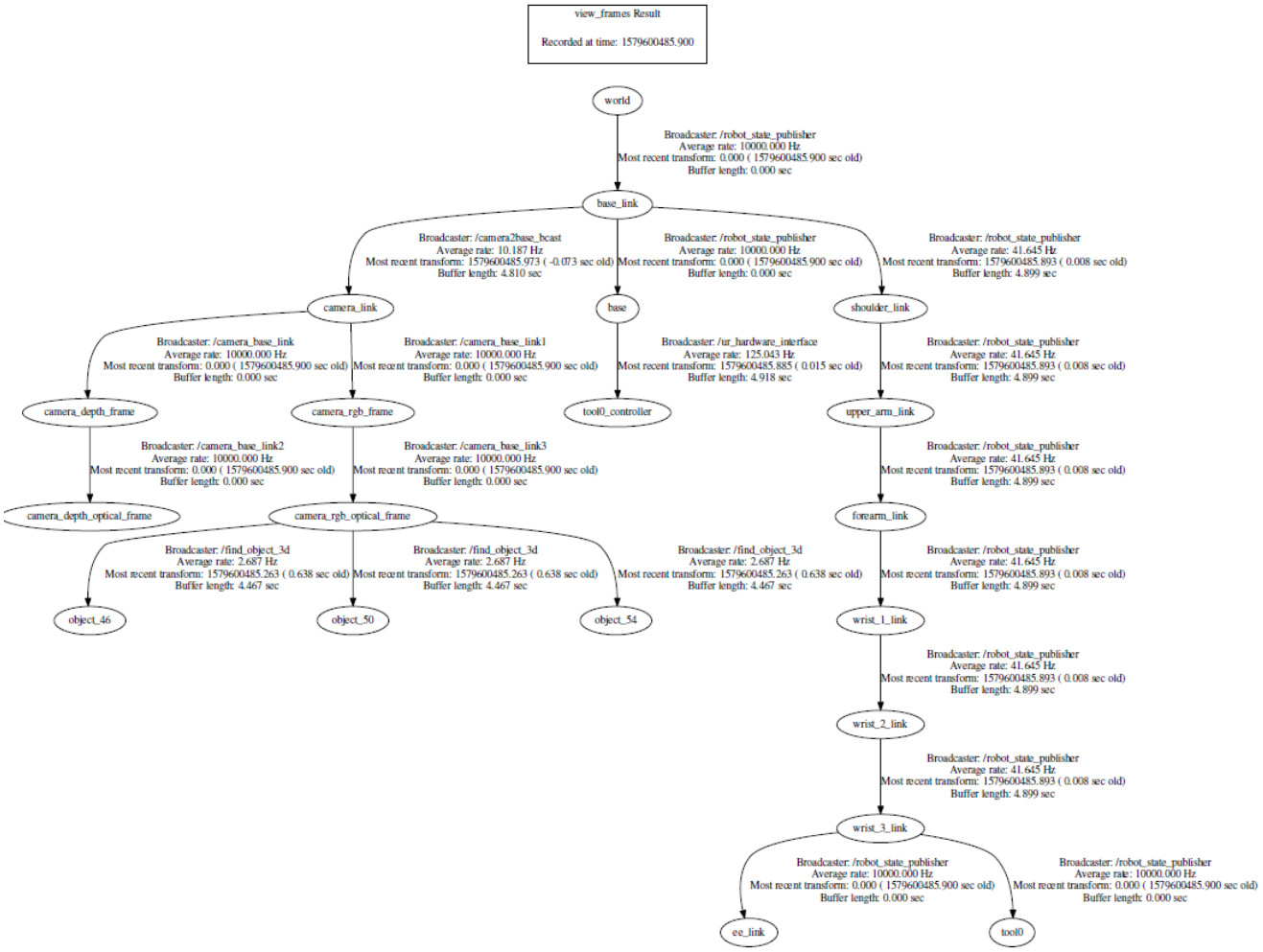
```
#!/usr/bin/env python

import sys
import copy
import rospy
import tf2_ros
import tf
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
import math
from std_msgs.msg import String, Float32MultiArray
from moveit_commander.conversions import pose_to_list

class UR10MoveItTest(object):
  """UR10MoveItTest"""
  def __init__(self):
    super(UR10MoveItTest, self).__init__()

    # Initialize moveit commander and rospy node
    moveit_commander.roscpp_initialize(sys.argv)
    rospy.init_node('moveittest',
                    anonymous=True)

    robot = moveit_commander.RobotCommander()
    scene = moveit_commander.PlanningSceneInterface()

    group_name = "manipulator"
    group = moveit_commander.MoveGroupCommander(group_name)

    display_trajectory_publisher = rospy.Publisher('/move_group/display_planned_path',
                                                   moveit_msgs.msg.DisplayTrajectory,
                                                   queue_size=20)

    ## Getting Basic Information for debugging purposes
    planning_frame = group.get_planning_frame()
    print "============ Reference frame: %s" % planning_frame

    eef_link = group.get_end_effector_link()
    print "============ End effector: %s" % eef_link

    group_names = robot.get_group_names()
    print "============ Robot Groups:", robot.get_group_names()

    print "============ Printing robot state"
    print robot.get_current_state()
    print ""

```

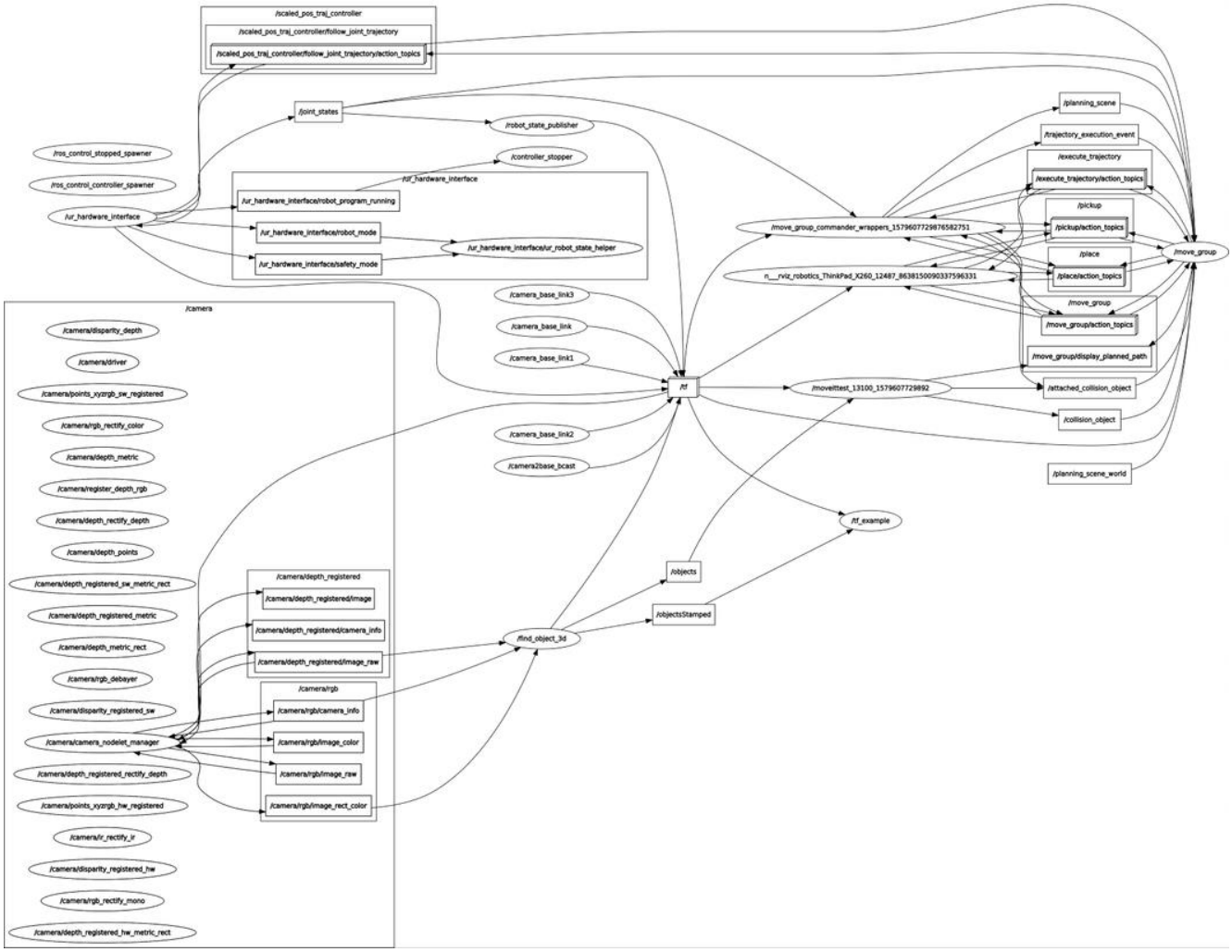
```
49      # Misc variables
50      self.box_name = ''
51      self.robot = robot
52      self.scene = scene
53      self.group = group
54      self.display_trajectory_publisher = display_trajectory_publisher
55      self.planning_frame = planning_frame
56      self.eef_link = eef_link
57      self.group_names = group_names
58      self.listener = tf.TransformListener()
59      frame_id = None
60
61  def go_to_joint_state(self):
62
63      # In this function, we'll set joint goal values for init state and perform a joint move there:
64
65      joint_goal = self.group.get_current_joint_values()
66      joint_goal[0] = math.radians(273.90)
67      joint_goal[1] = math.radians(-116.06)
68      joint_goal[2] = math.radians(105.48)
69      joint_goal[3] = math.radians(-78.73)
70      joint_goal[4] = math.radians(-90.24)
71      joint_goal[5] = math.radians(184.37)
72      self.group.go(joint_goal, wait=True)
73
74      # Using stop() to ensure there's no residual movement
75      self.group.stop()
76
77  def frame_callback(self, data):
78
79      global frame_id
80      frame_id = int(data.data[0])
81      return frame_id
82
83  def pose_coord_node(self):
84
85      rospy.Subscriber("objects", Float32MultiArray, self.frame_callback)
86      rospy.sleep(1)
87
88  def get_pose_coord(self):
89
90      listener = tf.TransformListener()
91      listener.waitForTransform("/camera_rgb_optical_frame", "/object_%s" % frame_id, rospy.Time(), rospy.Duration(4.0))
92      (trans,rot) = listener.lookupTransform('camera_rgb_optical_frame', 'object_%s' % frame_id, rospy.Time())
93
94      goal_coord = [trans[0] - 0.020, trans[1] - 0.15, trans[2] - 0.34]
95      return goal_coord
96
```

```
97  def go_to_pose_goal(self):
98
99      self.group.set_pose_reference_frame('camera_rgb_optical_frame')
100     self.group.set_end_effector_link('tool0')
101     self.group.clear_pose_targets
102     goal_coord = self.get_pose_coord()
103
104     q = tf.transformations.quaternion_from_euler(-0.30, 0, 0)
105
106     pose_goal = geometry_msgs.msg.Pose()
107     pose_goal.orientation.x = q[0]
108     pose_goal.orientation.y = q[1]
109     pose_goal.orientation.z = q[2]
110     pose_goal.orientation.w = q[3]
111     pose_goal.position.x = goal_coord[0]
112     pose_goal.position.y = goal_coord[1]
113     pose_goal.position.z = goal_coord[2]
114     self.group.set_pose_target(pose_goal)
115
116     plan = self.group.go(wait=True)
117     self.group.stop()
118     self.group.clear_pose_targets()
119
120 def grasp_object(self):
121
122     self.group.set_pose_reference_frame('camera_rgb_optical_frame')
123     self.group.set_end_effector_link('tool0')
124     self.group.clear_pose_targets
125     goal_coord = self.get_pose_coord()
126
127     q = tf.transformations.quaternion_from_euler(-0.30, 0, 0)
128
129     pose_goal = geometry_msgs.msg.Pose()
130     pose_goal.orientation.x = q[0]
131     pose_goal.orientation.y = q[1]
132     pose_goal.orientation.z = q[2]
133     pose_goal.orientation.w = q[3]
134     pose_goal.position.x = goal_coord[0]
135     pose_goal.position.y = goal_coord[1] + 0.027
136     pose_goal.position.z = goal_coord[2] + 0.057
137     self.group.set_pose_target(pose_goal)
138
139     plan = self.group.go(wait=True)
140     self.group.stop()
141     self.group.clear_pose_targets()
142
143     pose_goal = geometry_msgs.msg.Pose()
144     pose_goal.orientation.x = q[0]
145     pose_goal.orientation.y = q[1]
146     pose_goal.orientation.z = q[2]
147     pose_goal.orientation.w = q[3]
148     pose_goal.position.x = goal_coord[0]
149     pose_goal.position.y = goal_coord[1]
150     pose_goal.position.z = goal_coord[2]    # back to overhead pose
151     self.group.set_pose_target(pose_goal)
152
153     plan = self.group.go(wait=True)
154     self.group.stop()
155     self.group.clear_pose_targets()
156
```

```
156
157   def go_to_drop_point_1(self):
158
159     joint_goal = self.group.get_current_joint_values()
160     joint_goal[0] = math.radians(98.72)
161     joint_goal[1] = math.radians(-103.04)
162     joint_goal[2] = math.radians(128.64)
163     joint_goal[3] = math.radians(-114.92)
164     joint_goal[4] = math.radians(-90.24)
165     joint_goal[5] = math.radians(215.95)
166
167     self.group.go(joint_goal, wait=True)
168     self.group.stop()
169
170   def go_to_drop_point_2(self):
171
172     joint_goal = self.group.get_current_joint_values()
173     joint_goal[0] = math.radians(98.72)
174     joint_goal[1] = math.radians(-65.18)
175     joint_goal[2] = math.radians(88.39)
176     joint_goal[3] = math.radians(-107.48)
177     joint_goal[4] = math.radians(-90.86)
178     joint_goal[5] = math.radians(182.19)
179
180     self.group.go(joint_goal, wait=True)
181     self.group.stop()
182
183 # ------------------------------------------------------------------------
184
185 def main():
186   #try:
187   print "============ This is a simple moveit_commander test on the UR10."
188   print "============ Press `Enter` to begin the test, or ctrl+d to exit."
189   raw_input()
190   test = UR10MoveItTest()
191
192   while not rospy.is_shutdown():
193     print "============ Moving to init-state..."
194     print "============ WARNING!! Keep hands on e-stop at all times!"
195     test.go_to_joint_state()
196     print "============ Looking up coordinates..."
197     test.pose_coord_node()
198     test.get_pose_coord()
199
200     print "============ Moving to object overhead..."
201     print "============ WARNING!! Keep hands on e-stop at all times!"
202     test.go_to_pose_goal()
203
204     print "============ Grasping object..."
205     print "============ WARNING!! Keep hands on e-stop at all times!"
206     test.grasp_object()
207
208     print "============ Dropping object..."
209     print "============ WARNING!! Keep hands on e-stop at all times!"
210     test.go_to_drop_point_1()
211 #---------------------
212     print "============ Moving to init-state..."
213     print "============ WARNING!! Keep hands on e-stop at all times!"
214     test.go_to_joint_state()
215
216     print "============ Looking up coordinates..."
217     test.pose_coord_node()
218     test.get_pose_coord()
219
```

```
220     print "============ Moving to object overhead..."
221     print "============ WARNING!! Keep hands on e-stop at all times!"
222     test.go_to_pose_goal()
223
224     print "============ Grasping object..."
225     print "============ WARNING!! Keep hands on e-stop at all times!"
226     test.grasp_object()
227
228     print "============ Dropping object..."
229     print "============ WARNING!! Keep hands on e-stop at all times!"
230     test.go_to_drop_point_2()
231
232     print "============ Plan execution succeeded."
233
234
235 if __name__ == '__main__':
236   try:
237     main()
238
239   except rospy.ROSException, e:
240     rospy.logerr("============ Program shut down abruptly: %s"%e)
241   except KeyboardInterrupt:
242     print "============ Program terminated by user."
```


view_frames Result
Recorded at time: 1579600485.900
world
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1579600485.900 sec old)
Buffer length: 0.000 sec
base_link
Broadcaster: /camera2base_bcast
Average rate: 10.187 Hz
Most recent transform: 1579600485.973 (-0.073 sec old)
Buffer length: 4.810 sec
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1579600485.900 sec old)
Buffer length: 0.000 sec
Broadcaster: /robot_state_publisher
Average rate: 41.645 Hz
Most recent transform: 1579600485.893 (0.008 sec old)
Buffer length: 4.899 sec
camera_link
base
shoulder_link
Broadcaster: /camera_base_link
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1579600485.900 sec old)
Buffer length: 0.000 sec
Broadcaster: /camera_base_link1
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1579600485.900 sec old)
Buffer length: 0.000 sec
Broadcaster: /ur_hardware_interface
Average rate: 125.043 Hz
Most recent transform: 1579600485.885 (0.015 sec old)
Buffer length: 4.918 sec
Broadcaster: /robot_state_publisher
Average rate: 41.645 Hz
Most recent transform: 1579600485.893 (0.008 sec old)
Buffer length: 4.899 sec
camera_depth_frame
camera_rgb_frame
tool0_controller
upper_arm_link
Broadcaster: /camera_base_link2
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1579600485.900 sec old)
Buffer length: 0.000 sec
Broadcaster: /camera_base_link3
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1579600485.900 sec old)
Buffer length: 0.000 sec
Broadcaster: /robot_state_publisher
Average rate: 41.645 Hz
Most recent transform: 1579600485.893 (0.008 sec old)
Buffer length: 4.899 sec
camera_depth_optical_frame
camera_rgb_optical_frame
forearm_link
Broadcaster: /find_object_3d
Average rate: 2.687 Hz
Most recent transform: 1579600485.263 (0.638 sec old)
Buffer length: 4.467 sec
Broadcaster: /find_object_3d
Average rate: 2.687 Hz
Most recent transform: 1579600485.263 (0.638 sec old)
Buffer length: 4.467 sec
Broadcaster: /find_object_3d
Average rate: 2.687 Hz
Most recent transform: 1579600485.263 (0.638 sec old)
Buffer length: 4.467 sec
Broadcaster: /robot_state_publisher
Average rate: 41.645 Hz
Most recent transform: 1579600485.893 (0.008 sec old)
Buffer length: 4.899 sec
object_46
object_50
object_54
wrist_1_link
Broadcaster: /robot_state_publisher
Average rate: 41.645 Hz
Most recent transform: 1579600485.893 (0.008 sec old)
Buffer length: 4.899 sec
wrist_2_link
Broadcaster: /robot_state_publisher
Average rate: 41.645 Hz
Most recent transform: 1579600485.893 (0.008 sec old)
Buffer length: 4.899 sec
wrist_3_link
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1579600485.900 sec old)
Buffer length: 0.000 sec
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1579600485.900 sec old)
Buffer length: 0.000 sec
ee_link
tool0


/scaled_pos_traj_controller
/scaled_pos_traj_controller/follow_joint_trajectory
/scaled_pos_traj_controller/follow_joint_trajectory/action_topics
/joint_states
/robot_state_publisher
/ros_control_stopped_spawner
/ros_control_controller_spawner
/controller_stopper
/ur_hardware_interface
/ur_hardware_interface
/ur_hardware_interface/robot_program_running
/ur_hardware_interface/robot_mode
/ur_hardware_interface/safety_mode
/ur_hardware_interface/ur_robot_state_helper
/camera_base_link3
/camera_base_link
/camera_base_link1
/camera_base_link2
/camera2base_bcast
/tf
/camera
/camera/disparity_depth
/camera/driver
/camera/points_xyzrgb_sw_registered
/camera/rgb_rectify_color
/camera/depth_metric
/camera/register_depth_rgb
/camera/depth_rectify_depth
/camera/depth_points
/camera/depth_registered_sw_metric_rect
/camera/depth_registered_metric
/camera/depth_metric_rect
/camera/rgb_debayer
/camera/disparity_registered_sw
/camera/camera_nodelet_manager
/camera/depth_registered_rectify_depth
/camera/points_xyzrgb_hw_registered
/camera/ir_rectify_ir
/camera/disparity_registered_hw
/camera/rgb_rectify_mono
/camera/depth_registered_hw_metric_rect
/camera/depth_registered
/camera/depth_registered/image
/camera/depth_registered/camera_info
/camera/depth_registered/image_raw
/camera/rgb
/camera/rgb/camera_info
/camera/rgb/image_color
/camera/rgb/image_raw
/camera/rgb/image_rect_color
/find_object_3d
/objects
/objectsStamped
/tf_example
/planning_scene
/trajectory_execution_event
/execute_trajectory
/execute_trajectory/action_topics
/pickup
/pickup/action_topics
/place
/place/action_topics
/move_group
/move_group/action_topics
/move_group/display_planned_path
/attached_collision_object
/collision_object
/planning_scene_world
/move_group_commander_wrappers_1579607729876582751
n__rviz_robotics_ThinkPad_X260_12487_8638150090337596331
/moveittest_13100_1579607729892
/move_group