Ida Palotie

ROS-käyttöjärjestelmä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Automaatiotekniikka

Insinöörityö

22.3.2017

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Ida Palotie ROS käyttöjärjestelmä 35 sivua 22.3.2017
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Automaatiotekniikka
Suuntautumisvaihtoehto	
Ohjaaja(t)	Lehtori Jari Savolainen Lehtori Timo Tuominen

Insinöörityössä tutkittiin ROS-käyttöjärjestelmää. Tavoitteena oli laatia kattava kokonaisuus siitä, mikä ROS on, miten sitä käytetään ja millainen tulevaisuus käyttöjärjestelmällä on. Insinöörityö tehtiin Metropolia Ammattikorkeakoululle.

Työssä käytettävä materiaali koottiin ROS-käyttöjärjestelmän wiki-sivuilta, sekä sen omilta kotisivuilta. Informaatiota saatiin myös järjestelmän ylläpitäjältä Open Source Robotics Foundation -yhtiöltä sekä haastattelun muodossa että internetsivujen avulla. Tietojen pohjalta laadittiin kattava kokonaisuus, jolla pääsisi jyvälle käyttöjärjestelmään. ROS-käyttöjärjestelmän ohjelmointi ja käyttö suoritettiin Linux-pohjaisella tietokoneella.

Työ sisältää tarkkoja ohjeita pakettien luontiin ja järjestelmän käyttöön, sekä listauksia erilaisista työkaluista ja ominaisuuksista, joita käyttöjärjestelmä pitää sisällään.

ROS-käyttöjärjestelmää käytetään enimmäkseen tuotekehityksessä. Tämän takia kehittelijät ovat suunnitelmissaan miettineet saavansa käyttöjärjestelmän käyttöön myös lopputuotteissa, jotka päätyvät markkinoille. Tavoitteena on laajentaa käyttöä useamman robotin kohteisiin kuten teollisuuden tehtaisiin ja järjestelmiin. Pidemmän ajan tavoite on laajentua itseajavien autojen pariin.

Avainsanat	ROS, robotiikka, käyttöjärjestelmä

Author(s) Title Number of Pages Date	Ida Palotie ROS operating system 35 pages 22 May 2017
Degree	Bachelor of Engineering
Degree Programme	Automation Technology
Specialisation option	
Instructor(s)	Jari Savolainen, Senior Lecturer Timo Tuominen, Senior Lecturer

This study concerns ROS operating system. The aim was to draw up a comprehensive package of ROS, what is it, how to use it and what is the future of ROS.

The material used for this work were collected from ROS operating system wiki pages, as well as its own home page. Information was also obtained from the system administrator the Open Source Robotics Foundation company, both in the form of an interview and as information on their internet pages.

The work includes detailed instructions as well as listings of the various tools and features, which are included in ROS operating system.

ROS operating system is mainly used for product development. Developers are planning to get the operating system also for use in end products that are released to the market. The aim is to expand the use of the system to more than one robot schemes such as industrial plants and systems.

Keywords	ROS, robotics, operating system

Sisällys

Lyhenteet

ROS	*Robot Operating System.* Open source pohjainen käyttöjärjestelmä.
STAIR	*STanford AI Robot*. Stanfordin yliopiston kehittelemä robotti.
BSD	*Berkley Software Distributionin.* vapaa ohjelmistolisenssi.
RPC	*Remote Procedure Call*. Prosessien välinen tekniikka, joka mahdollistaa tilaajan ja serverin ohjelmiston kommunikoinnin.
URI	*Uniform Resource Identifier.* Merkkijono, jolla ilmaistaan tiedoston sijainti.
MSG	*Message*. Tekstitiedosto.
SRV	*Service*. Palvelu.

1 Johdanto

Insinöörityössä käyttöönotetaan ROS-käyttöjärjestelmä ja tutkitaan sen tulevaisuuden näkymiä. Työssä kerrotaan yleisesti käyttöjärjestelmästä ja sen historiasta. Tavoitteena on luoda kattava kokonaisuus käyttöjärjestelmästä ja sen käytöstä. Työ tehdään Metropolia Ammattikorkeakoululle.

2 ROS

ROS (Robot Operating System) on joustava käyttöjärjestelmä, jonka avulla voidaan luoda erilaisten robottien ohjelmistoja. Se sisältää erilaisia työkaluja, kirjastoja sekä merkintätapoja, jotka helpottavat monimutkaisten ja vakaiden ohjelmien tekemisen erilaisille robottialustoille [1.]

ROS sisältää erilaisia ohjaimia, algoritmeja sekä voimakkaita kehittäjän työkaluja. Sen käyttö on suunniteltu niin harrastajalle kuin ammattilaiselle. Ohjelman tarkoitus on auttaa käyttäjää luomaan robottiohjelma omaan käyttöön. Se sisältää kaiken tarvittavan kaikkiin robottiprojekteihin. Lisäksi ROS toimii open source –pohjalla [1.]

ROS on myös käytössä opetuksessa. Sitä käyttävät kaiken ikäiset oppilaat niin pienistä lapsista, jotka ovat vuorovaikutuksessa museoiden robotteihin, että korkeakoulu opiskelijoihin, jotka tutkivat uusimpia ratkaisuja robotiikan ongelmiin [1.]

ROS:a käytetään laajasti erilaisissa roboteissa, mukaan lukien halvan alustan omaavia robotteja, kuten Turtlebot ja LEGO Mindstroms. ROS soveltuu erinomaisesti luokkahuone käyttöön [1.]

Historiaa

ROS:a alettiin kehittämään osana Stanfordin yliopiston projektia, jossa tavoitteena oli toteuttaa tukeva järjestelmä, sekä interaktiivinen robottiohjelma Stanfordin omalle STAIR-robotille (STanford AI Robot), 2000-luvun puolessa välissä [2.]

Vuonna 2007 projektiin osallistui kalifornialainen yritys nimeltä Willow Garage. Yritys tarjosi merkittäviä resursseja ja oli näin osallisena viemässä kehitystä kohti joustavampaa ja dynaamisempaa ohjelmistoa. Tämä myös mahdollisti projektiin lisää resursseja sekä asiantuntemusta lukuisista tutkimuksista [2.]

ROS oli suunniteltu BSD-lisenssin alaisuuteen, ja vähitellen asiantuntijat alkoivat käyttää järjestelmää apuvälineenään tutkimuksissaan sekä tuotekehityksissään. Ajan kuluessa siitä on tullut laajasti käytetty alusta robottien tutkimuksessa ja kehityksessä [2.]

Vuonna 2013 ROS:n ylläpito ja kehitys siirrettiin Open Source Robotics Foundationi:lle. Tänä päivänä ROS:a käytetään maailmanlaajuisesti sekä harrastekäytössä että ison skaalan teollisuuden automaatiojärjestelmissä [2.]

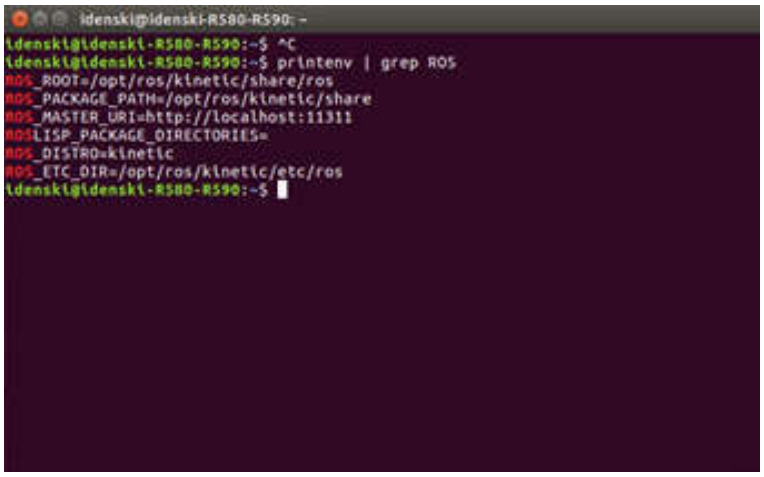
3 Käyttö

Työssä lähdettiin tutkimaan ROS-käyttöjärjestelmää internetistä löytyvien käyttöoppaiden kautta. Näissä käydään läpi ROS:n konfigurointi sekä perusteet järjestelmään.

3.1 Asennus ja konfigurointi

ROS-käyttöjärjestelmän pystyy asentamaan Linux-pohjaiselle tietokoneelle taikka virtuaalikoneelle. Asennus riippuu käyttäjän Linux-versiosta. Ohjeet löytyvät erikseen jokaiseen ROS-versioon.

Konfiguroinnissa tärkeintä on tarkistaa, missä erilaiset muuttujat, kuten ROS_ROOT ja ROS_PACKAGE_PATH sijaitsevat. Näiden sijainteja tarvitaan myöhemmässä vaiheessa. Tarkistus suoritetaan komennolla *$ printenv* | *grep ROS*. Kuvassa 1 nähdään esimerkki muuttujien sijainneista [3.]



Kuva 1. Kuvakaappaus muuttujien sijainnista.

Tarkistuksen jälkeen haetaan tiedostot komennolla *$ source /opt/ros/kinetic/setup.bash* [3.]

Tästä eteenpäin ohjeita löytyy kahta erilaista, rosbuild ja catkin. Työssä käytettiin catkin-menetelmää. ROS-käyttöjärjestelmässä ohjelmointimenetelmänä käytettävä Catkin yhdistää makroja sekä Python-koodia. Catkin on perinteisempi tapa toimia ROS:ssa. Se helpottaa pakettien jakelua, antaa tuen useamman lähteen kokoamiseen samaan tiedostoon sekä helpottaa siirrettävyyttä. Rosbuild on alkuperäinen menetelmä ohjelmoida ja toimia ROS-käyttöjärjestelmässä. Se on osa ytimenrakennustyökalu Cmake:n ohjelmointimenetelmää. Rosbuildin toiminta on parhaimmillaan vanhemmissa ROS-versioissa [4.]

3.2 Tietojärjestelmän käyttö

Työtila luodaan komennoilla *$ mkdir -p ~/catkin_ws/src, $ cd ~/catkin_ws/src* ja *$ catkin_init_workspace*. Työtilaan tallennetaan kaikki tiedostot ja kansiot, joita projektissa tarvitaan, ilman työtilaa ei ole projektia [5.]

Jotta työtilaa voitaisiin käyttää, täytyy se vielä rakentaa komennoilla *$ cd ~/catkin_ws/* ja *$ catkin_make*. Tämän jälkeen työtilan kansiosta tulisi löytyä build- ja devel-kansiot. Devel-kansion sisältä löytyy asetustiedostoja. Jos jokin näistä tiedostoista päivitetään, työtila muuttuu päällimmäiseksi työympäristössä [5.]

Tietojärjestelmään kuuluvat paketit, jotka ovat ohjelmiston yksikköjä ROS-koodissa. Ne voivat sisältää kirjastoja, koodeja, ohjelmatiedostoja ja muita tiedostoja. Manifestit ovat pakettien kuvauksia [6.]

Käyttöjärjestelmässä koodi on hajautettuna eri paketteihin. Navigointi suoritetaan päätteen komentorivin työkaluilla, kuten ls ja cd, mikä voi kuitenkin olla turhauttavaa isoissa projekteissa. Tämän takia ROS:iin on kehitetty työkaluja, jotka helpottavat tiedostojen ja kansioiden löytämistä [6.]

Rospack-työkalu antaa sinulle tietoja paketista. Find-komennolla saadaan vastaukseksi paketin sijainti. Esimerkiksi *$ rospack find roscpp* -komennolla saadaan päätteelle tulostettua kyseisen paketin sijainti: *[sinun asennuspolku] /share/roscpp* [6.]

Roscd vaihtaa sijaintiasi suoraan pakettiin, pinoon taikka suoraan, jonkin paketin alahakemistoon. Esimerkiksi *$ roscd roscpp/cmake* [6.]

Komennolla *$ roscd log* voidaan siirtyä suoraan kansioon, johon ROS kerää kaikki lokitiedot. Jos ei ole ennen tätä komentoa suorittanut yhtään prosessia järjestelmässä, tulee tästä error-ilmoitus [6.]

Rosls-komennolla saadaan näkyviin lista paketin sisällöstä, sen absoluuttisen sijainnin sijaan. Tarvittava käsky on *$ rosls [paketin nimi[/alahakemisto]]* [6.]

Jotkin ROS-työkalut tukevat TAB-täydennystä. Tämä toiminto toimii tietokoneen TAB-näppäimestä. Painalluksen jälkeen järjestelmä täydentää loput nimestä, mikäli se löytää vain yhdenvaihtoehdon. Jos järjestelmä ei ole varma mitä käyttäjä tarkoittaa, toiminto täyttää niin pitkälle kuin osaa, minkä jälkeen uudelleen painamalla TAB-näppäintä saadaan lista vaihtoehdoista [6.]

3.3 ROS-paketin luonti

Jotta pakettia voidaan pitää catkin-pakettina, sillä on muutamia vaatimuksia:

- Paketin pitää sisältää catkinin kanssa yhteen sopiva Package.xml-tiedosto.

- Paketin pitää sisältää CMakelists.txt, joka käyttää catkinia.

- Kussakin kansiossa voi olla enintään yksi paketti. Se tarkoittaa, ettei voi olla sisäkkäisiä paketteja eikä useita paketteja jakamassa samaa hakemistoa.

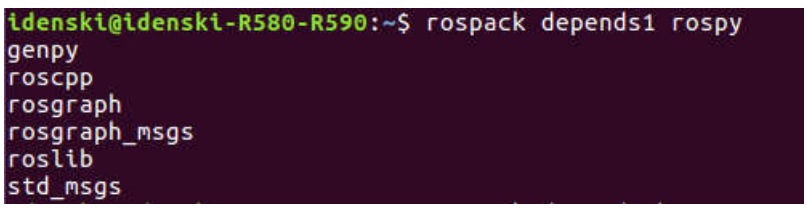
On suositeltavaa työskennellä catkin-pakettien kanssa catkin-työtilassa, mutta tarvittaessa paketit kykenevät olemaan itsenäisiä [6.]

Catkin-paketti luodaan komennolla *$catkin_create_pkg,* haluttuun kansioon. Esimerkiksi *$ catkin_create_pkg beginner_tutorials std_msgs rospy roscpp*. Tässä luotiin catkin-paketti nimeltä beginner_tutorials, joka riippuu riippuvuuksista std_msgs:sta, roscpp:sta ja rospystä. Paketin luonnin yhteydessä luotiin kansio, joka sisältää Package.xml- ja CMakeLists.txt-tiedostot. Catkin-paketin luontikomento vaatii aina paketin nimen, lisäksi voi ottaa vapaavalintaisia riippuvuuksia [6.]

3.3.1 Paketin riippuvuudet

Paketin luonnin yhteydessä tulee riippuvuuksia. Nämä ovat niin sanottuja ensimmäisen tason riippuvuuksia, joita voidaan luonnin jälkeen tarkastella rospack-komennolla. Tässä listassa tulisi olla riippuvuudet, jotka määriteltiin paketin luonnin yhteydessä. Riippuvuudet sijaitsevat Package.xml-tiedostossa [6.]

Komennolla *$ rospack depends1 beginner_tutorials* saadaan listattua beginner_tutorials-paketin riippuvuudet. Joissain tapauksissa riippuvuuksilla on myös omia riippuvuuksia. Esimerkiksi rospyllä. Nämä saadaan listattua komennolla *$ rospack depends1 rospy*. Tulostuksessa tulisi olla listattuna kuvassa 2 näkyvät riippuvuudet [6.]



Kuva 2. Rospyn riippuvuudet.

Mikäli tahdotaan paketin kaikista riippuvuuksista vain yksi lista, voidaan käyttää seuraavaa komentoa *$ rospack depends [paketinnimi]*. Tulostuksessa tulisi olla kuvan 3 mukaiset riippuvuudet [6.]

```
idenski@idenski-R580-R590:~$ rospack depends beginner_tutorials
cpp_common
rostime
roscpp_traits
roscpp_serialization
catkin
genmsg
genpy
message_runtime
gencpp
geneus
gennodejs
genlisp
message_generation
rosbuild
rosconsole
std_msgs
rosgraph_msgs
xmlrpcpp
roscpp
rosgraph
rospack
roslib
rospy
```

Kuva 3. Koko paketin riippuvuudet.

3.3.2 Paketin kustomointi

Paketin kustomoinnissa täytyy olla erittäin tarkka. Ohjeissa opastetaan muutama kohta, joissa muutetaan kuvausta, ylläpitäjän sähköpostia ja nimeä, lisenssiä ja riippuvuuksien tunnisteita. Tämä kaikki tehdään teksti editorilla. Ubuntu-käyttäjät voivat myös käyttää komentoa *$ sudo gedit [tiedoston nimi mitä muutetaan]* [6.]

ROS-käyttöjärjestelmässä on myös toinen tapa muokata tiedostoja. Komennolla *$ rosed* voidaan muokata tiedostoa suoraan päätteeltä [6.]

Rosed on osa rosbash-kokonaisuutta. Se mahdollistaa tiedoston muokkauksen nimen perusteella. Muokkaus saadaan toimintaan komennolla *$ rosed [paketin nimi] [tiedoston nimi]* [6.]

Mikäli ei tiedetä haetun tiedoston nimeä, voidaan paketin nimen jälkeen kaksi kertaa painaa TAB-näppäintä, joka tulostaa päätteeseen kyseisen paketin sisällä olevat tiedostot [7.]

3.3.3 ROS-paketin rakennus

ROS-paketti pitää vielä rakentaa, se suoritetaan komennolla *$ catkin_make*. Tämä on komentorivityökalu, joka lisää kätevyyttä normaaliin catkin-työnkulkuun. Työkalu yhdistää cmake- ja make-toiminnot [8.]

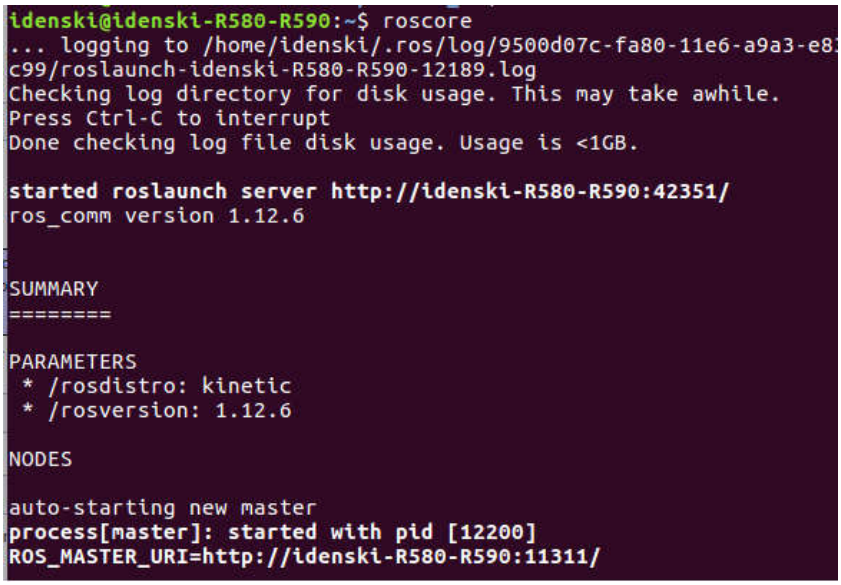
Komennossa tulee ottaa huomioon, että komento rakentaa kaikki paketit kyseisestä kansiosta, esimerkiksi *$ cd ~/catkin_ws/* ja *$ ls src.* Tämän jälkeen suoritetaan rakennus komento *$ catkin_make.* Ruudulle tulee paljon tulostusta, jossa ensimmäisenä on cmake-työkalun paketin rakennusosia ja tämän jälkeen make-rakennusosia. Kun toiminto on valmis, tulisi kohde työtilasta löytyä build-, devel- ja src-kansiot. Build-kansio on oletussijainti, jossa sijaitsee rakennustila, missä cmake ja catkinmake konfiguroivat ja rakentavat paketteja. Devel-kansio on oletussijainti, missä on kehitystila mihin rakennettavat tiedostot ja kirjastot menevät ennen paketin rakennusta [8.]

3.4 ROS-prosessit

Prosessit ovat yhteydessä toisiinsa luodakseen grafiikkaa. Ne keskustelevat keskenään käyttäen aiheita, RPC-palveluja sekä parametriservereitä [9]. Prosessit voivat myös tarjota ja käyttää palveluita. ROS-asiakaskirjastot mahdollistavat prosessin kommunikoinnin kirjoituksen eri ohjelmointikielillä. Ne ovat:rospy; Python-kirjasto ja roscpp; C++:n kirjasto [10.]

3.4.1 Roscore

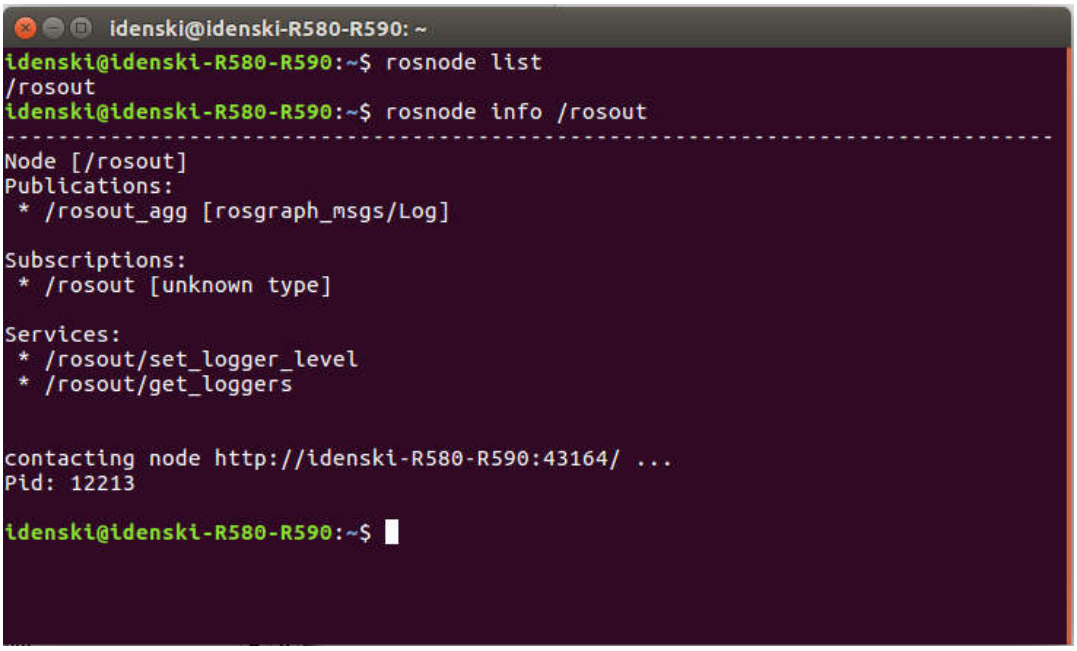
Roscore on kokoelma erilaisia prosesseja ja ohjelmia, jotka ovat edellytyksiä ROS-pohjaiselle järjestelmälle. Roscoren täytyy olla käynnissä, jotta ROS-prosessit pystyvät keskustelemaan keskenään. Roscore käynnistetään komennolla *$ roscore* [11.] Kuvassa 4 nähdään roscore-komennon tulostus.



Kuva 4. Roscore tulostuksen tulisi näyttää tältä.

3.4.2 Rosnode

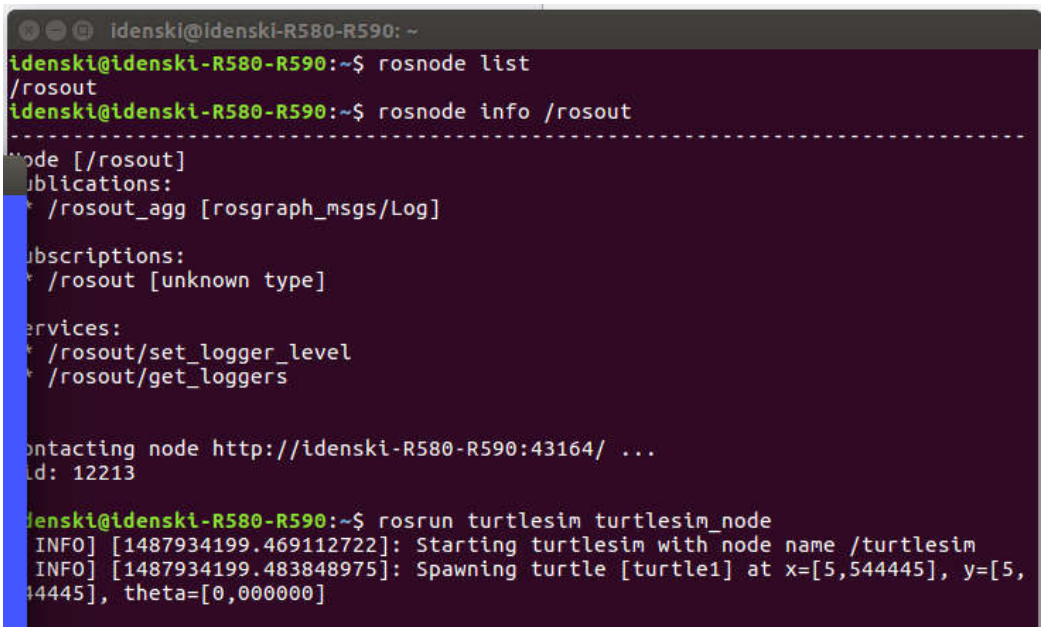
Rosnode-komento näyttää tietoja, mitä prosessissa tapahtuu. *$ rosnode list* listaa aktiivisena olevat prosessit. *$ rosnode info* –komento kertoo enemmän tietoja kyseisestä prosessista [10.] Kuvasta 5 nähdään list- ja info-komentojen tulostus.



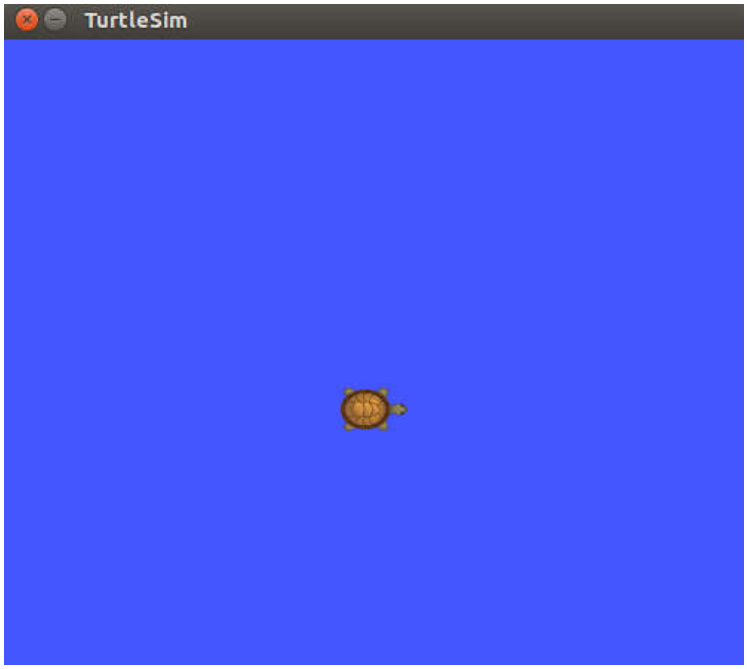
Kuva 5. Komentojen list ja info käyttö.

3.4.3 Rosrun

Rosrun mahdollistaa prosessin käyttöä suoraan paketista ilman, että tietää paketin sijaintia. Kuvassa 6 suoritetaan rosrun-komento, joka avaa kuvan 7 mukaisen turtlesim-prosessin [11.]

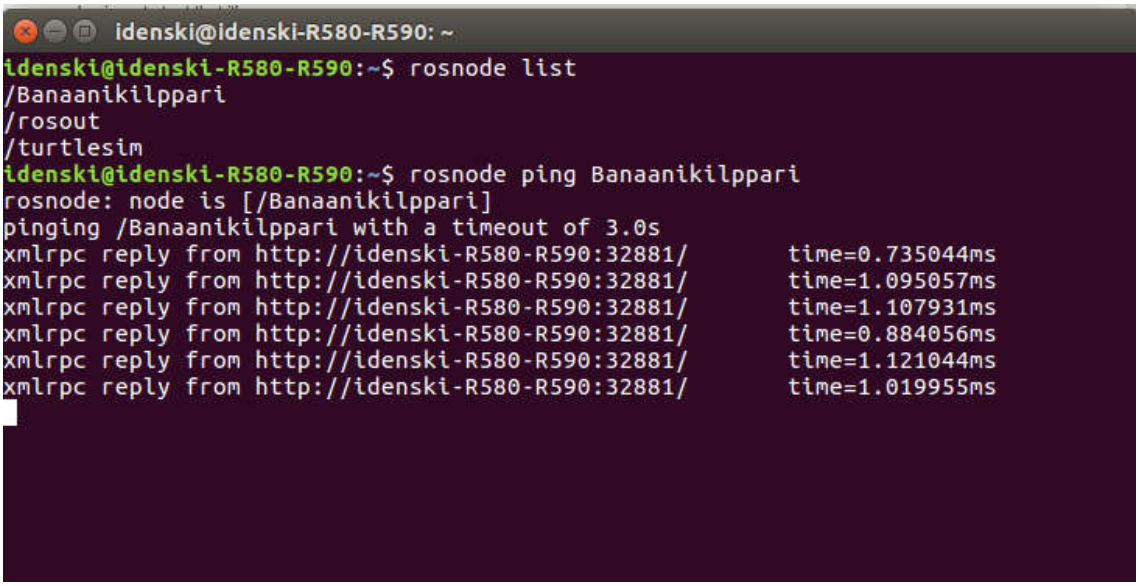


Kuva 6. Rosrun-komennon käyttö.



Kuva 7. Rosrun-komennon kilpikonnat voivat olla erinäköisiä.

Rosnode-komennolla pystymme testaamaan turtlesim-prosessia. Käytetään komentoa *$ rosnode ping [kilpikonnannimi]* [11.] Kuvasta 8 nähdään ping-toiminnon tulostus.



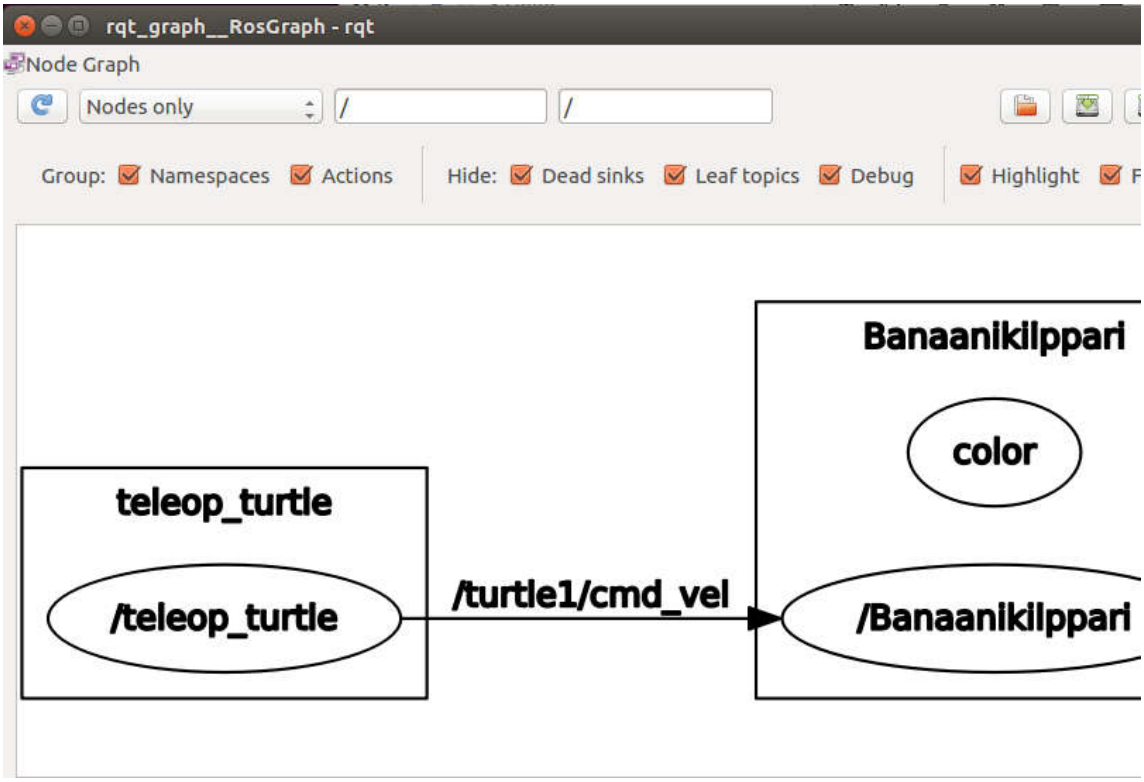
Kuva 8. Ping-toiminnon tulostus.

Kaikki ylläolevat komennot lukittavat päätteen käytön, joten jokaista uutta komentoa varten täytyy avata uusi pääte. Lukituksen voi avata painamalla ctrl+C.

3.5 Turtlesim

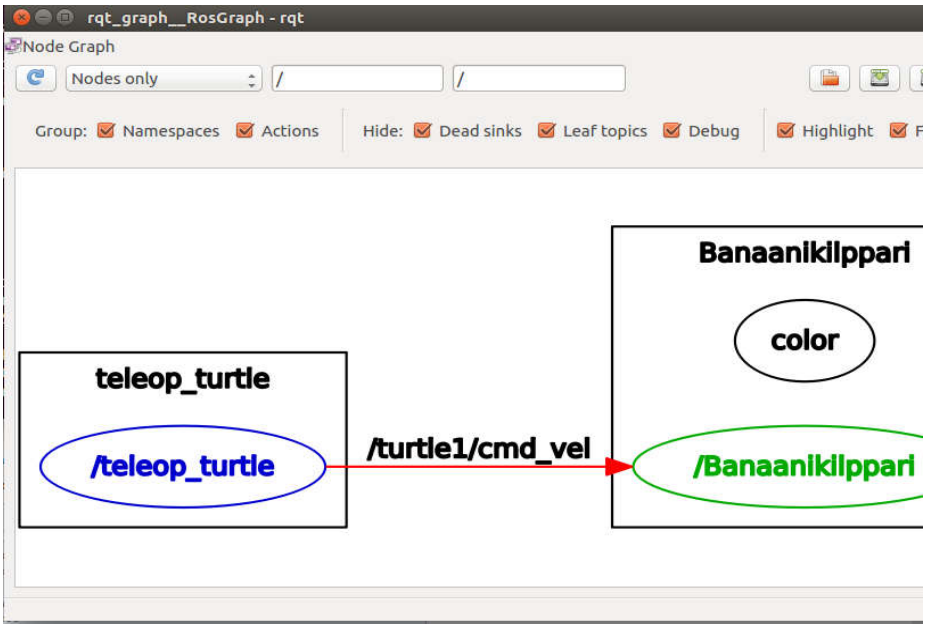
ROS-käyttöjärjestelmä sisältää turtlesim-prosessin. Tämän avulla opetetaan ROS-käyttöjärjestelmää, sekä pakettien käyttöä. Pystymme tämän prosessin avulla tutustumaan parametrien muutoksiin, aiheiden käyttöön ja tutkimiseen, sekä muihin työkaluihin, joita ROS-käyttöjärjestelmässä voidaan käyttää [12.]

Turtlesim saadaan käyttöön komennolla *$ rosrun turtlesim turltesim_node*. Tämä komento avaa turtlesim-ikkunan, jossa on sininen pohja ja kilpikonna. *$rosrun turtlesim turtle_teleop_key* saadaan konfiguroitua nuolinäppäimet kilpikonnan liikutteluun. Sen lineaarinen ja kulmikas nopeus saadaan *$ turtlex/ cmd_vel* komennolla. *$ rosrun rqt_graph rqt_graph* -käskyllä saadaan aikaiseksi dynaaminen kuvaaja, joka kertoo mitä järjestelmässä tapahtuu. Esimerkkinä kuvan 9 mukainen kuvaaja [13.]



Kuva 9. Esimerkki miltä kuvaaja alkuvaiheessa näyttäisi.

Jos hiiri sijoitetaan cmd_*vel* kohdalle, järjestelmä korostaa eri väreillä sen osat. Vihreä ja sininen kuvastavat prosesseja, punainen aiheita. Kuvassa 10 on korostettu kuvaaja malliksi [13.]



Kuva 10. Esimerkkikuva korostetuista järjestelmän osista.

Komennolla *$ rostopic hz /turtle1/pose* saadaan tieto millä nopeudella tietoa julkaistaan järjestelmässä [13.]

Turtlesim-prosessista löytyy useita palveluita, jotka ovat seuraavat:

Clear	tällä komennolla saadaan tyhjennettyä kilpikonnan tausta ja samalla se päivittää taustavärin sille määrättyyn parametriin.
Reset	toiminto palauttaa turtlesimin aloitus konfigurointiin ja asettaa taustavärin vakioksi.
Kill	komento hävittää kilpikonnan. Esimerkiksi jos simulaattorissa pyörii useampi kilikonna, voidaan jokin niistä hävittää ilman, että kaikki katoavat.
Spawn	luo kilpikonnan määriteltyyn pisteeseen ja palauttaa kilpikonnan nimen.
TurtleX/set_pen	asettaa kynän värin, leveyden ja laittaa kynän päälle tai pois.
TurtleX/teleport_absolute	siirtää kilpikonnan haluttuun pisteeseen.
TurtleX/teleport_relative	siirtää kilpikonnan lineaarisen tai kulmikkaan etäisyyden kilpikonnan nykyisestä sijainnista [12.]

Turtlesim sisältää myös taustakuvan parametreja. Näitä pystytään muuttamaan komennolla *$ rosparam set [parametrin nimi].* Esimerkiksi *$ rosparam set /background_r 150.* Kyseisen komennon jälkeen tulee tausta tyhjentää, jotta järjestelmä päivittää parametrit. Tämä suoritetaan toiminolla *$ rosservice call /clear* [13.]

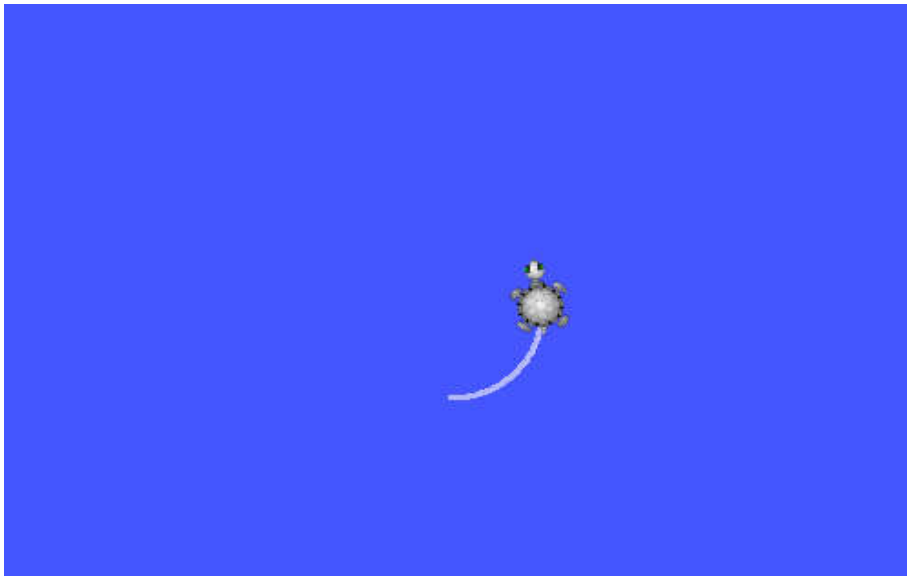
Taustan värin parametrit voidaan muuttaa seuraavilla parametrinimillä.

~background_b	hallinnoi sinistä väriä taustavärissä ja sen vakioarvo on 255.

~background_g hallinnoi vihreää väriä taustassa ja sen vakioarvo on 86.

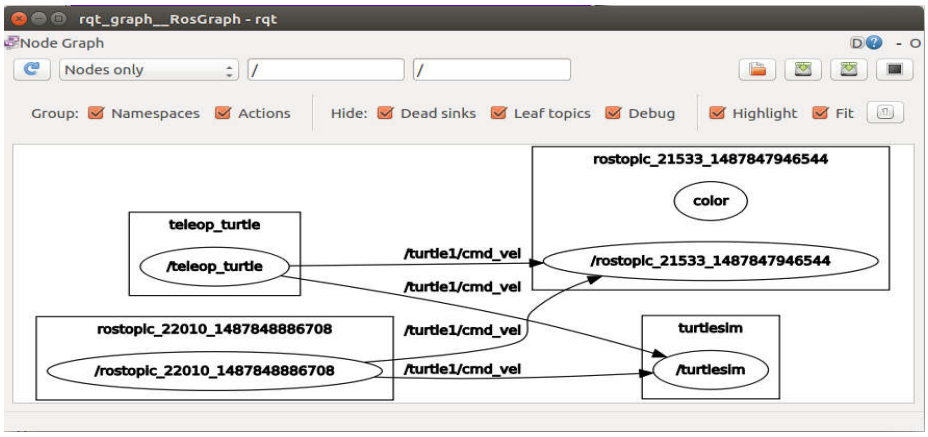
~background_r hallinnoi punaista väriä taustassa ja sen vakioarvo on 69 [12.]

Komennolla *$ rostopic pub [aihe] [viestityyppi] [argumentit]* saadaan julkaistua tietoa käytössä olevaan aiheeseen. Esimerkiksi komennolla *$ rostopic pub -1 /turtle1/command_velocity turtlesim/Velocity -- 2.0 1.8* saadaan kilpikonna liikkumaan neljännes ympyrä. Kuvassa 11 nähdään kilpikonnan liikerata rostopic-komennolla [13.]



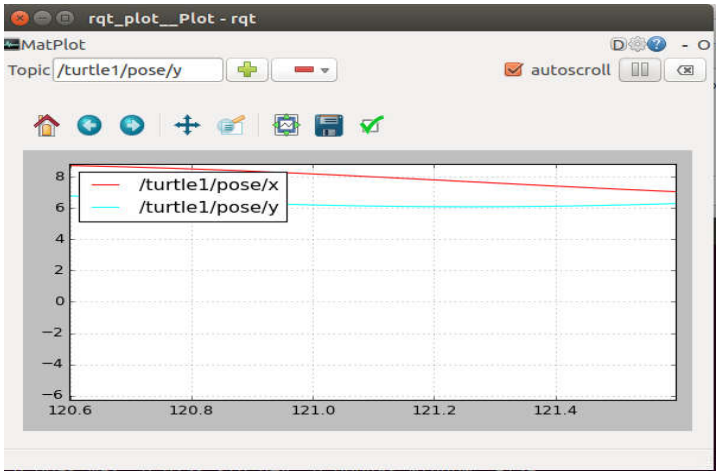
Kuva 11. Kilpikonnan neljännesympyrä.

Tällä komennolla kilpikonna ei kuitenkaan liiku jatkuvasti. Mikäli kohde tahdotaan jatkuvaan liikkeeseen voidaan se suorittaa komennolla *$ rostopic pub /turtle1/command_velocity turtlesim/Velocity -r 1 -- 2.0 -1.8.* Tällä komennolla saadaan haluttu kohde pyörimään ympyrää 1 Hz taajuudella. Jos tutkitaan uudelleen kuvaajaa, huomataan sen muuttuneen huomattavasti. Kuvassa 12 on nähtävissä ympyrää liikkuvan kilpikonnan kuvaaja [13.]



Kuva 12. Kuvaaja jatkuvan liikkeen aikana.

Jos halutaan turtlesimin liikkeestä käyrät, saadaan se näkyviin komennolla *$ rosrun rqt_plot rqt_plot.* Tämä kuitenkin avaa tyhjän kuvaajan. Kuvassa 13 nähdään vasemmassa yläkulmassa tekstilaatikko, joka määrittelee mitä aihetta kyseinen käyrä seuraa. Käyrään voidaan lisätä useampi aihe [13.]



Kuva 13. Turtlesim-käyrä.

3.6 ROS-aiheet

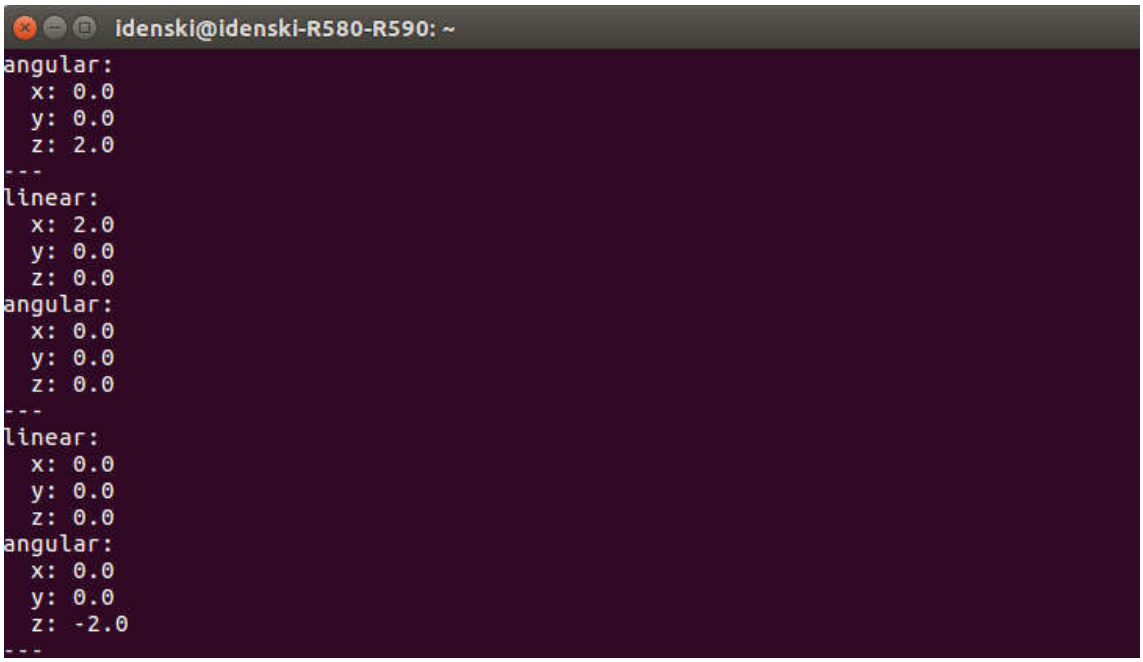
Aiheet toimivat prosessien kommunikointiväylinä. *$ rostopic –h* -komennolla saadaan lista alakomennoista.

Bw näyttää aiheen käyttämän kaistanleveyden.

Echo tulostaa viestin päätteelle

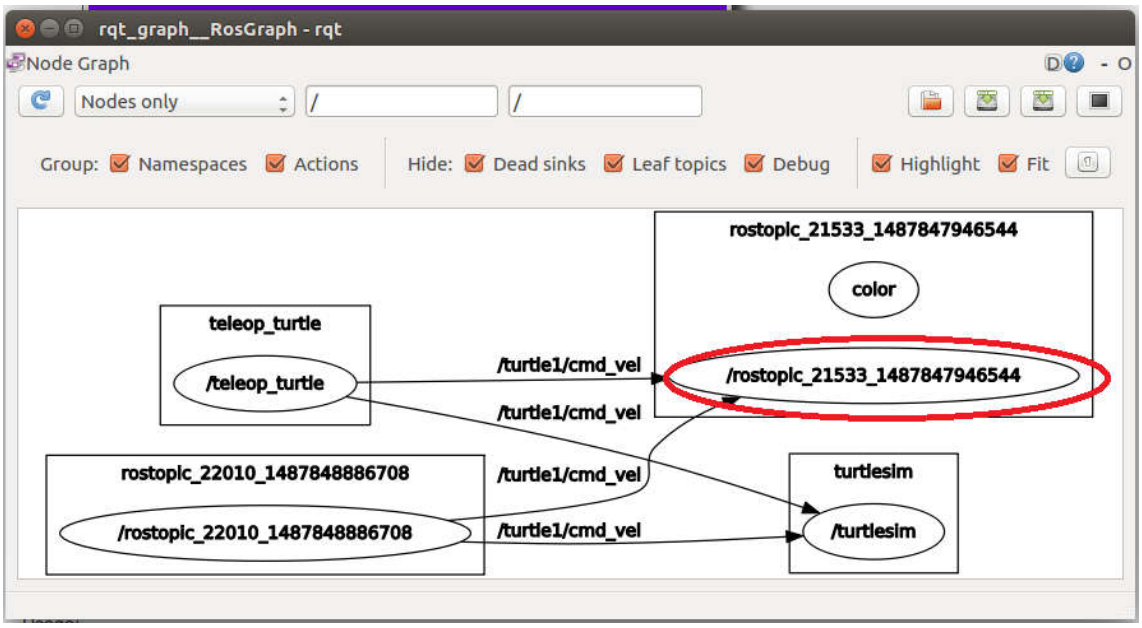
Hz	näyttää aiheen julkaisu tahdin.
List	tulostaa tiedon aktiivisista aiheista.
Pub	julkaisee tietoa aiheeseen
Type	tulostaa aiheen tyypin.

Esimerkiksi komennolla *$ rostopic echo /turtle1/cmd_vel* saadaan tulostettua prosessissa liikkuvat viestit päätteelle. Kuvassa 14 on esimerkki tulostus rostopic echo-komennosta [13.]



Kuva 14. Esimerkkikuva rostopic echo-toiminnosta.

Kuvassa 15 on punaisella ympyröity rostopic-komennon vaikutus grafiikkaan.



Kuva 15. Kuvassa punaisella ympyröity rostopic echo.

$ rostopic list –v -komennolla saadaan listattua monisanainen lista aiheista, joihin voidaan julkaista. Kuvasta 16 nähdään aihe listaus [13.]

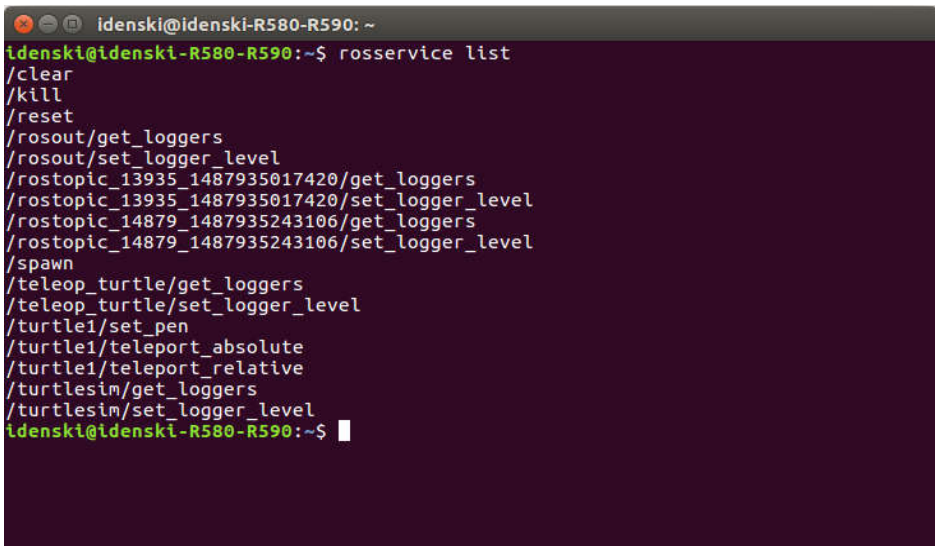
```
idenski@idenski-R580-R590: ~
idenski@idenski-R580-R590:~$ rostopic list -v

Published topics:
 * /turtle1/color_sensor [turtlesim/Color] 1 publisher
 * /turtle1/cmd_vel [geometry_msgs/Twist] 1 publisher
 * /rosout [rosgraph_msgs/Log] 4 publishers
 * /rosout_agg [rosgraph_msgs/Log] 1 publisher
 * /turtle1/pose [turtlesim/Pose] 1 publisher

Subscribed topics:
 * /turtle1/cmd_vel [geometry_msgs/Twist] 2 subscribers
 * /rosout [rosgraph_msgs/Log] 1 subscriber
 * /statistics [rosgraph_msgs/TopicStatistics] 1 subscriber

idenski@idenski-R580-R590:~$
```

Kuva 16. Aihelistaus.

3.7 ROS -palvelut

Palvelut ovat toinen tapa, millä prosessit voivat kommunikoida toisilleen. Palvelut mahdollistavat prosessien lähettää pyyntöjä ja vastaan ottaa vastauksia. Rosservice-komennon alla on useita käskyjä, joita voidaan käyttää aiheissa [14.]

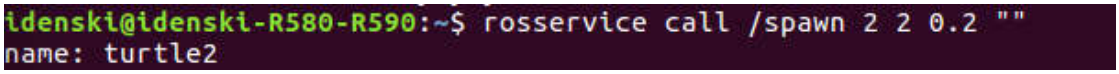
List tulostaa listan kaikista tarjolla olevista palveluista. Kuvassa 17 on esimerkki list-komennon tulostuksesta [15.]



Kuva 17. Rosservice list -tulostus

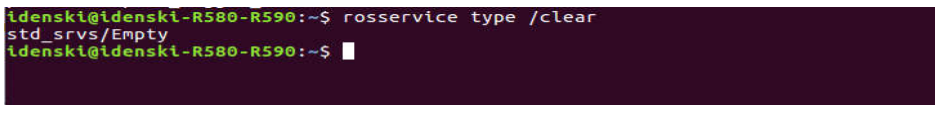
Call soittaa palvelulle, joka tarjoaa tarvitut riippuvuudet. Kuvista 18 ja 19 nähdään call-komennon tulostuksista [15.]

Kuva 18. Call-toiminto. Tämä ei tulosta ruudulle mitään, mutta se tyhjentää turtlesim -ruudun.



Kuva 19. Call-toiminto. Tämä komento luo toisen kilpikonnan ruudulle, haluttuun pisteeseen.

Type tulostaa palvelun tyypin. Kuvassa 20 nähdään rosservice type-komennon tulostus [15.]



Kuva 20. Type servisen käyttö.

Find etsii palveluja palvelutyypin mukaan.
Uri tulostaa palvelun osoitteen eli ROSRPC URI:n [15.]

3.8 ROSparam

Rosparam antaa sinulle mahdollisuuden tallentaa ja manipuloida tietoa ROS-parametripalvelussa. Se voi tallettaa kokonaislukuja, liukulukuja, totuusarvoja, sanasto- ja ja listoja. Rosparam sisältää useita komentoja [14.]

Set asettaa parametrin.

Get hakee parametrin.

Load lataa parametrin tiedostosta.

Dump siirtää parametrit tiedostoon.

Delete poistaa parametrin.

List listaa parametrien nimet.

Esimerkiksi turtlesim -prosessista löytyy kolme parametria taustan väriä varten. Nämä ovat background_b, background_g ja background_r. Parametreja voidaan muuttaa set-komennolla ja clear-komennolla saadaan muutokset päivitettyä kuvaan. Get-komennolla saadaan tiedot haetuista parametreista [14.]

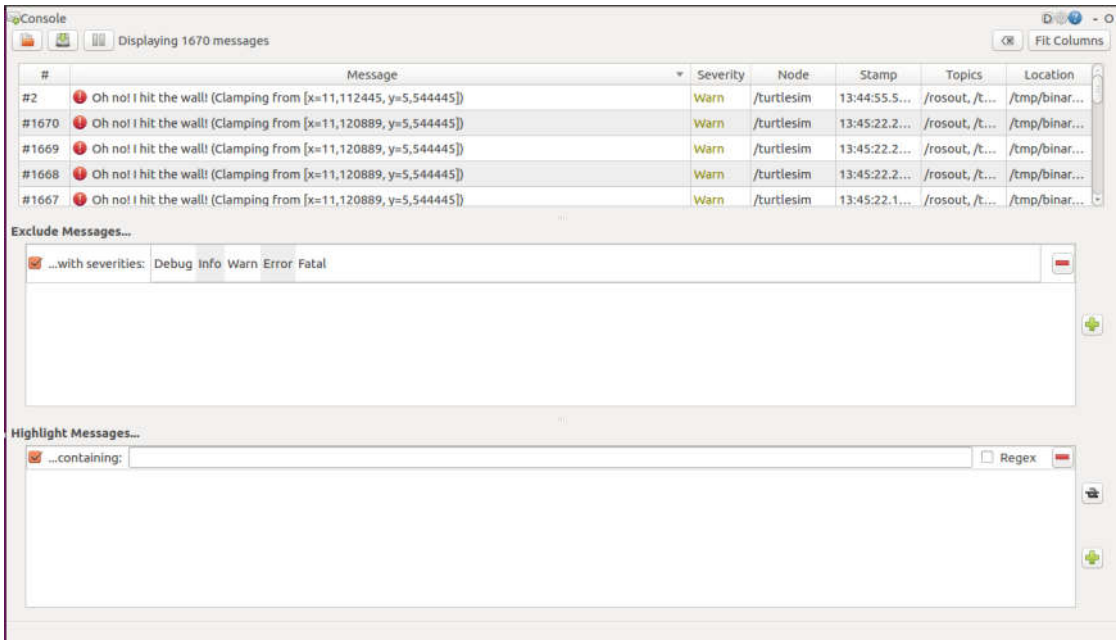
```
idenski@idenski-R580-R590:~$ rosparam set /background_r 150
idenski@idenski-R580-R590:~$ rosservice call /clear

idenski@idenski-R580-R590:~$ rosparam get /background_g
86
idenski@idenski-R580-R590:~$ rosparam get /
background_b: 255
background_g: 86
background_r: 150
rosdistro: 'kinetic

  '
roslaunch:
  uris: {host_idenski_r580_r590__44778: 'http://idenski-R580-R590:44778/'}
rosversion: '1.12.6

  '
run_id: 5d1cb170-fa81-11e6-a9a3-e839df2a2c99
```

Kuva 21. Set-, clear- ja get-komennot.

Kuvan 21 mukaisesti tehdyn komennon jälkeen, tulisi turtlesim-prosessin taustavärin muistuttaa violettia väriä [14.]

3.9 Graafinen käyttöliittymä ja virheenkorjaus

ROS-käyttöjärjestelmässä voidaan käyttää graafista käyttöliittymää sekä virheen korjausta, komennoille *$ rosrun rqt_console rqt_console* ja *$ rosrun rqt_logger_level rqt_logger_level*. Nämä tulee syöttää omiin päätteisiinsä. Console-komento tuo viestejä prosesseista. Loggerlevel-ikkunasta voidaan vaihtaa kyseisen prosessin viestien tärkeyttä, esimerkiksi infosta varoitukseen, taikka error-viestiin [16.]



Kuva 22. rqt_console.

3.10 Launch-tiedosto

Launch-komento käynnistää prosessit, jotka on määritelty launch-tiedostossa. Kansiota ei ole valmiiksi ROS-käyttöjärjestelmässä, joten se pitää luoda. Kansio luodaan komennoilla *$ mkdir launch* ja *$ cd launch* [16.]

Kansioon voidaan luoda nyt tiedostoja, jota launch-komento käynnistää. Esimerkiksi turtlemimic.launch-tiedosto, jonka tarkoituksena on toiseen turtlesim-prosessiin matkia ensimmäisen turtlesim-prosessin liikkeet. Tiedosto luodaan yksinkertaisesti notepad-

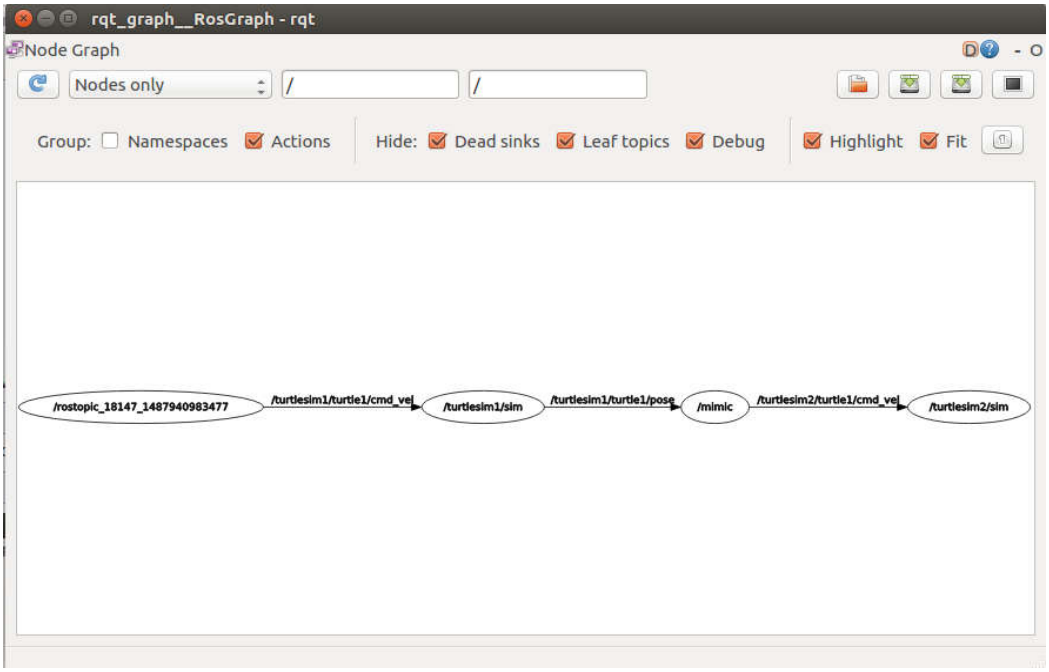
pohjalle, joka nimetään turtlemimic.launch -nimiseksi. Tiedoston sisään kirjoitetaan haluttu koodi [16.]

```
<launch>

  <group ns="turtlesim1">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>

  <group ns="turtlesim2">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>

  <node pkg="turtlesim" name="mimic" type="mimic">
    <remap from="input" to="turtlesim1/turtle1"/>
    <remap from="output" to="turtlesim2/turtle1"/>
  </node>

</launch>
```

Esimerkkikoodi 1. Turtlemimic-koodi

Tässä <launch> -tunnisteilla määritellään tiedoston tyyppi. Seuraavaksi luodaan kaksi ryhmää, joiden nimi tunnisteella mahdollistetaan kahden simulaattorin toiminta ilman nimiristiriitaa. Viimeiseksi luodaan matkinta prosessi, jossa uudelleen nimetään sisään-tulo- ja ulostulo-linjat, jotta saadaan aikaiseksi toisen kilpikonnan matkiminen [16.]

Tämän jälkeen tallennetaan tiedosto ja siirrytään päätteelle. Kirjoitetaan päätteelle komento *$ roslaunch beginner_tutorials turtlemimic.launch*. Tämän pitäisi avata kaksi turtlesim-prosessia. Uuteen päätteeseen, kun kirjoittaa toiselle kilpikonnalle komennon, seuraa myös toinen kilpikonna, vaikka vain toista käskytettiin. Kun käynnistää kuvaajan nähdään kyseisen tiedoston toiminta [16.] Kuvassa 23 nähdään graafinen kuvaaja turtlemimic-tiedoston toiminnasta.



Kuva 23. Turtlemimi.launch -toiminta.

3.11 Viestit ja palvelukuvaus

Viestit ovat teksti tiedostoja, jotka kuvaavat ROS-viestikenttiä. Näitä käytetään tuottamaan lähdekoodin viestit eri kielillä. Srv:t ovat tiedostoja, jotka kuvaavat palveluja. Ne sisältävät kaksi osaa, pyyntöjä ja vastauksia [17.]

Viestit luodaan komennoilla *$ roscd beginner_tutorials, $ mkdir msg* ja *$ echo "int64 num" > msg/Num.msg.* Edeltävä komento luo vain yksirivisen viestin. Niitä voidaan luoda myös monitasoisempia viestejä, joissa on useampi rivi [17.]

Viestin luonnin jälkeen tulee tarkistaa, että package.xml-tiedostossa on esimerkkikoodi 2:ssa ilmentyvät rivit. Ne eivät myöskään saa olla kommentteina tiedostossa [17.]

```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

Esimerkkikoodi 2. Package.xml-tiedostoon tarvittavat rivit.

Muokataan vielä CMakeLists.txt-tiedostoa. Lisätään find_package-komponenttilistaan messaga_generation. Find_package-listasta löytyy kaikki paketin riippuvuudet, jotka

määriteltiin aikaisemmassa vaiheissa. Tulee myös varmistaa, että catkin_package:sta löytyy catkin_depends -kohdasta message_runtime, jos ei niin se lisätään tiedostoon. Etsitään vielä tiedostosta esimerkkikoodi 3:ssa ilmentyvä kohta. Poistetaan siitä kommenttiviittaukset poistamalla # -merkki [17.]

```
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )
```

Esimerkkikoodi 3. CMakeLists.txt teksti pätkä.

Samainen toimenpide tehdään myös esimerkkikoodi 4:n kohdalle [17.]

```
# generate_messages(
#   DEPENDENCIES
#   std_msgs
# )
```

Esimerkkikoodi 4. CMakeLists.txt tekstipätkä.

Myös srv -tiedoston kohdalla täytyy muokata CMakeLists.txt-tiedostoa. Etsitään tiedostosta esimerkkikoodi 5:ssa ilmentyvä kohta ja poistetaan siitä kommenttiviittaukset poistamalla # -merkki [17.]

```
# add_service_files(
#   FILES
#   Service1.srv
#   Service2.srv
# )
```

Esimerkkikoodi 5. CMakeLists.txt teksti.

Muutetaan vielä haluttu paikanvaraaja esimerkkikoodi 6 kohdalle [17.]

```
add_service_files(
  FILES
  AddTwoInts.srv
)
```

Esimerkkikoodi 6. CMakeLists.txt teksti.

Nämä toimenpiteet tehdään vain kerran, kun viestejä aletaan luomaan kyseisessä työtilassa. Varmistetaan, että järjestelmä osaa lukea tiedostot. Käytetään *komentoa* $ rossrv show <palvelun tyyppi> [17.]

Kun tiedostot on luotu, tulee paketti luoda uudelleen. Tämä suoritetaan komennoilla *$ roscd beginner_tutorials, $ cd../.. , $ catkin_make install* ja *$ cd* – [17.]

Mikäli ei kuitenkaan muista mitä kaikkia viestejä on, voidaan ne aputyökalujen avulla saada listattua. Komennolla *$ rosmsg –h* saadaan lista kaikista mahdollisista komennoista viestien käsittelyyn. Alakomentoja ovat seuraavat:

Show näyttää viestin kuvauksen.

List listaa kaikki viestit.

Package listaa viestit paketissa.

Packages listaa kaikki paketit, jotka sisältävät viestejä.

Viestejä voidaan lukea komennolla *$ rosmsg show [viestin tyyppi].* Tämän pitäisi tulostaa päätteelle viestin sisältö [17.]

3.12 Yksinkertaisen julkaisijan ja tilaajan ohjelmointi

Julkaisija ja tilaaja voidaan ohjelmoida kahdella eri kielellä: Python ja C++. Tässä työssä käytettiin C++-ohjelmointikieltä. Julkaisijalla tarkoitetaan prosessia, joka jatkuvalla syötöllä julkaisee viestejä [18.]

Jotta julkaisija tulee oikeaan pakettiin, siirrytään komennolla $ *roscd [paketin nimi]*, haluttuun pakettiin. Tämän jälkeen luodaan src-kansio komennolla *$ mkdir src*. Luodaan kansioon tiedosto talker.cpp ja kirjoitetaan esimerkkikoodi 7:n mukainen koodi [18.]

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub =
n.advertise<std_msgs::String>("chatter", 1000);

  ros::Rate loop_rate(10);
  int count = 0;
  while (ros::ok())
  {
    std_msgs::String msg;
    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();
    ROS_INFO("%s", msg.data.c_str());
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
    ++count;
  }
  return 0;
}
```

Esimerkkikoodi 7. Talker -palvelun koodi.

Koodissa määritetään talker-prosessi ja sen toiminta. Selitetään hieman koodia ros::init()-function täytyy nähdä argc ja argv, jotta se voi toimia. Argc ja argv ovat argumentteja, jotka ovat käyttöjärjestelmässä. Nodehandle on objekti, joka edustaa ROS-prosesseja [19]. Se on myös pääkommunikointipiste ROS-järjestelmässä. Ensimmäinen nodehandle alustaa prosessin ja viimeinen sulkee sen. Advertise() funktio kertoo

järjestelmälle, mitä käyttäjä tahtoo julkaista kyseiseen aiheeseen. Funktio palauttaa julkaisu kohteen, mikä mahdollistaa viestien julkaisun. Toinen advetisen-parametri on viestin koko, jota käytetään viestin julkaisussa. Tämä kertoo järjestelmälle, miten monta viestiä puskuriin menee, ennen niiden poistoa. Int count = 0 kertoo, kuinka monta viestiä on lähetetty. Seuraavassa kohdassa on viesti objekti. Tähän tulee kaikki tieto, mitä halutaan julkaista aiheeseen. Publish-funktio julkaisee viesti kohteen sisällä olevan tiedon [18.]

Seuraavaksi luodaan kansioon toinen tiedosto, jonka nimeksi tulee listener.cpp. Kirjoitetaan sinne esimerkkikoodi 8:n mukainen koodi [18.]

```
#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}
int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("chatter", 1000, chatter-
Callback);
  ros::spin();
  return 0;
}
```

Esimerkkikoodi 8. Listener-koodi.

Tässä koodissa määritetään listener -prosessi ja sen toiminta. Koodissa on samanlainen alku kuin on talker -prosessissa. Subscribe()-funktio kertoo järjestelmälle, että halutaan vastaanottaa viestejä tietystä aiheesta, kyseinen toiminto toimii vastaavanlaisesti kuin advertise -toiminto. Viestit toimitetaan chatterCallback:n kautta. Ros::spin() tekee silmukan, joka tehostaa takaisin soittoja [18.]

Jotta julkaisija ja tilaaja toimisivat oikein, tulee muokata CMakeLists.txt-tiedostoa. Lisätään tiedoston loppuun esimerkkikoodin 9 mukainen teksti [18.]

```
include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
add_dependencies(talker begin-
ner_tutorials_generate_messages_cpp)

add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
add_dependencies(listener begin-
ner_tutorials_generate_messages_cpp)
```

Esimerkkikoodi 9. CMakeLists.txt teksti.

Tämän jälkeen paketti täytyy taas luoda uudelleen, joten käytetään komentoa *$ catkin_make* [18.]

Julkaisijan ja tilaajan suorittamien

Kun julkaisija ja tilaaja on luotu, voidaan niitä tarkastella ja suorittaa. Jotta ne kuitenkin toimisivat oikein, tulee avata ensimmäiseksi roscore. Tämä tehtiin komennolla *$ roscore,* tämän jälkeen uuteen päätteeseen voidaan käyttää komentoa *$ rosrun beginner_tutorials talker.* Tämän pitäisi tulostaa päätteelle tekstiä, joka riippuu, siitä mitä julkaisijan koodiin on kirjoitettu. Kuvassa 24 on esimerkki julkaisijan tulostuksesta [20.]

```
[INFO] [WallTime: 1314931831.774057] hello world 1314931831.77
[INFO] [WallTime: 1314931832.775497] hello world 1314931832.77
[INFO] [WallTime: 1314931833.778937] hello world 1314931833.78
[INFO] [WallTime: 1314931834.782059] hello world 1314931834.78
[INFO] [WallTime: 1314931835.784853] hello world 1314931835.78
[INFO] [WallTime: 1314931836.788106] hello world 1314931836.79
```

Kuva 24. Talker-tuloste [20.]

Seuraavaksi voidaan avata uusi pääte, johon voidaan avata tilaaja. Se saadaan toimintaan komennolla *$ rosrun beginner_tutorials listener.* Päätteelle pitäisi nyt tulostua tietoa, mitä tilaaja vastaanottaa julkaisijalta [20.]

3.13 Yksinkertaisen palvelun ja asiakasohjelman ohjelmointi

Palvelu ja asiakasohjelma voidaan ohjelmoida kahdella eri kiellä: Python ja C++. Tässä työssä käytettiin C++-ohjelmointikieltä. Palveluprosessi luodaan notepad-tiedostona halutun paketin sisällä. Tässä tapauksessa se on beginner_tutorials-paketti. Tiedostoon kirjoitetaan haluttu koodi. Esimerkiksi esimerkkikoodi 10:n mukainen [21.]

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"

bool add(beginner_tutorials::AddTwoInts::Request  &req,
         beginner_tutorials::AddTwoInts::Response &res)
{
  res.sum = req.a + req.b;
  ROS_INFO("request:  x=%ld,  y=%ld",  (long  int)req.a,  (long
int)req.b);
  ROS_INFO("sending back response: [%ld]", (long int)res.sum);
  return true;
}
int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_server");
  ros::NodeHandle n;
  ros::ServiceServer service =
n.advertiseService("add_two_ints", add);
  ROS_INFO("Ready to add two ints.");
  ros::spin();
  return 0;
}
```

Esimerkkikoodi 10. Palvelun esimerkkikoodi [21.]

Tässä luodaan palvelu, joka yhdistää kaksi kokonaislukua ja antaa vastaukseksi totuusarvomuuttujan [21.]

Seuraavaksi luodaan asiakasohjelma prosessi. Tehdään paketin sisälle uusi tiedosto, johon kirjoitetaan haluttu koodi. Esimerkiksi esimerkkikoodin 11: sta mukainen [21.]

```
#include "ros/ros.h"

#include "beginner_tutorials/AddTwoInts.h"
#include <cstdlib>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_client");
  if (argc != 3)
  {
    ROS_INFO("usage: add_two_ints_client X Y");
    return 1;
  }
  ros::NodeHandle n;
  ros::ServiceClient client =
n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_ints");
  beginner_tutorials::AddTwoInts srv;
  srv.request.a = atoll(argv[1]);
  srv.request.b = atoll(argv[2]);
  if (client.call(srv))
  {
    ROS_INFO("Sum: %ld", (long int)srv.response.sum);
  }
  else
  {
    ROS_ERROR("Failed to call service add_two_ints");
    return 1;
  }
  return 0;
}
```

Esimerkkikoodi 11. Asiakasohjelman esimerkkikoodi [21.]

Tässä koodissa luodaan asiakasohjelma äsken luotuun prosessiin. Tämä toimittaa palvelulle tarvittavat kokonaisluvut, jonka jälkeen se tulostaa kyseisen vastauksen [21.]

Palveluiden rakennuksen jälkeen tulee muokata CMakeLists.txt-tiedostoa. Lisätään tiedoston loppuun esimerkkikoodin 12:sta mukainen teksti [21.]

```
add_executable(add_two_ints_server src/add_two_ints_server.cpp)
target_link_libraries(add_two_ints_server ${catkin_LIBRARIES})
```

```
add_dependencies(add_two_ints_server beginner_tutorials_gencpp)

add_executable(add_two_ints_client src/add_two_ints_client.cpp)
target_link_libraries(add_two_ints_client ${catkin_LIBRARIES})
add_dependencies(add_two_ints_client beginner_tutorials_gencpp)
```

Esimerkkikoodi 12. CMakeLists.txt teksti [21.]

Tämän jälkeen paketti täytyy taas rakentaa uudelleen komennoilla *$ cd ~/catkin_ws* ja *$ catkin_make* [21.]

Palvelu testataan komennolla *$ rosrun beginner_tutorials add_two_ints_server*. Tulostukseen pitäisi tulla ilmoitus, että palvelu on valmis lisäämään kaksi kokonaislukua. Seuraavaksi annetaan komento *$ rosrun beginner_tutorials add_two_ints_client 1 3*. Tässä annetaan asiakasohjelmalle kaksi lukua, jotka se ilmoittaa palveluun. Tulostukseksi pitäisi tulla laskutoimitus sekä sen vastaus [22.]

3.14 Tietojen tallennus ja uudelleen toisto

Tietojen tallennus voidaan suorittaa vain käynnissä olevasta prosessista. Tallennettava tieto kerätään niin sanottuun bag-tiedostoon. Tiedostojen tallennus aloitetaan komennoilla *$ mkdir ~/bagfiles, $ cd ~/bagfiles* ja *$ rosbag record –a*. Tämä luo väliaikaisen kansion, mihin järjestelmä tallentaa tietoja. *Rosbag record -a* indikoi, että kaikki aiheet tallennetaan kyseiseen kansioon. Tämän jälkeen voidaan kohdetta, esimerkiksi kilpikonnaa, liikuttaa halutulla tavalla ja järjestelmä tallentaa kaiken ylös. Tallennus lopetetaan painamalla ctrl+c [23.]

Tallennuksia voidaan tutkia kansiosta, mihin tallennukset on viety. Kansiosta tulisi löytyä tiedosto, missä on otsikkona vuosi, tieto ja aika. Esimerkiksi 2017-03-10-15-08-34.bag. Sen sisällä on kaikki aiheen tallennetut julkaisut tallennuksen ajalta [23.]

Käyttäessä *$ rosbag info [tallennus tiedostonnimi]* -komentoa saadaan päätteelle kaikki tiedot kyseisestä tallennuksesta. Sieltä löytyy niin tiedoston polku, versio, kesto, alku, loppu, koko, viestit, pakkaus, tyyppi sekä aiheet [23.]

Tallennuksen toistaminen suoritetaan komennolla *$ rosbag play<tiedoston nimi>.* Päätteelle pitäisi ilmestyä tieto, että pakettia avataan ja järjestelmä odottaa muutaman sekunnin ennen, kuin se alkaa julkaisemaan viestejä. Odottamisella estetään viestien hukkaan menemistä. Järjestelmän pitäisi kuitenkin alkaa hetken päästä toistamaan tallennettua tietoa ja simulaation seuraamaan julkaistuja viestejä. Tallennuksen toiston nopeuteen voidaan myös vaikuttaa. Komennolla *$ rosbag play -r 2 <tiedoston nimi>* saadaan tiedoston toisto tupla nopeudelle [23.]

Järjestelmässä voidaan myös tallentaa vain muutamia aiheita, mikäli kaikkia ei haluta. Tämä kuitenkin vaikuttaa paljon liikeratoihin, joita kohde liikkuu. Komennolla *$ rosbag record -o subset <aihe mitä tallennetaan> <toinen aihe mitä tallennetaan>* saadaan vain tallennutettua valittuja aiheita. Tässä myös määritellään, että tiedot tallennetaan tiedostoon nimeltä subset.bag. Esimerkiksi turtlesim -prosessissa, jos kuvataan vain cmd_vel(nopeus) ja pose(asento) -aiheita. Kilpikonna ei liiku täydellisesti haluttua polkua, koska kilpikonnan liikkeeseen liittyy paljon muitakin aiheita kuin nämä kaksi [23.]

3.15 Vianetsintä

ROS-käyttöjärjestelmästä löytyy roswtf-työkalu vianetsintään. Se tutkii käyttäjän ROS-asetukset, kuten ympäristömuuttujat, ja etsii konfigurointiongelmia. Jos ROS-järjestelmä on online-tilassa, se tutkii siitä mahdolliset ongelmat. Vianetsinässä on muutakin kuin yleiset tarkistukset, siinä on myös kaksi muuta käyttötapaa. Roswtf:llä voidaan tarkistaa paketteja ja pinoja, sekä launch-tiedostoja. Roswtf saadaan käyttöön komennolla *$ roswtf* [24.]

4 ROS käyttöjärjestelmän komennot

Alle on listattu kaikki työkalut ROS-käyttöjärjestelmästä.

Rosinstall	helpottaa ROS-pakettien lähteiden asennuksen.
Roswtf	vianetsintätyökalu.
Rosdash	ei ole työkalu vaan kokoelma erityökaluista.

Rosbag	tallennuksen työkalu, joka sisältää toiston, tallennuksen ja vahvistuksen.
Roscd	osa rosdash-kokoelmaa. Ohjaa sinut haluttuun tiedostoon, pinoon taikka pakettiin.
Rosclean	tiedostojen siivoustyökalu.
Roscore	on kokoelma prosesseja, joka mahdollistaa prosessien kommunikoinnin.
Rosdep	asentaa järjestelmän riippuvuudet.
Rosed	osa rosdash-kokoelmaa. Mahdollistaa pakettien tiedostojen muokkauksen vain tiedostonnimen avulla.
Roscreate-pkg	luo uuden ROS-paketin ja siihen liittyvät tiedosto.
Roscreate-stack	luo uuden ROS-pinon ja siihen liittyvät tiedostot.
Rosrun	mahdollistaa paketin prosessien käynnistyksen pelkästään prosessin nimellä.
Roslaunch	käynnistää kaikki prosessit kansiosta, missä työkalua käytetään.
Roslocate	osa rosinstall -työkalua. Hakee ROS-paketin sijainnin.
Rosmake	rakennustyökalu ROS-käyttöjärjestelmässä.
Rosmsg	näyttää valitun viestin järjestelmästä.
Rosnode	näyttää käynnissä olevan prosessin tietoja. Voidaan myös kokeilla yhteyttä prosesseihin.
Rospack	hakee tietoja ROS-paketeista.

Rosparam	työkalulla muutetaan ja haetaan parametreja.
Rossrv	näyttää palveluiden tietojen määritelmät.
Rosservice	näyttää aktiivisten palveluiden tietoja. Mahdollistaa myös viestien vastaanottamisen ja lähettämisen.
Rosstack	hakee tietoja ROS-pinoista, mitä järjestelmässä on.
Rostopic	näyttää aktiivisten aiheiden tietoja ja tulostaa tulevat viestit aiheeseen.
Rosversion	raportoi ROS-pinon version.
Rqt_bag	graafinen työkalu tallennus tiedostojen katseluun.
Rqt_deps	luo PDF-tiedoston ROS:n riippuvuuksista.
Rqt_graph	luo interaktiivisen kaavion prosesseista ja aiheista.
Rqt_plot	tekee diagrammin valituista aiheista [25;26;27;28;29.]

5 Mahdollisuudet

ROS-käyttöjärjestelmä on jo useamman vuoden ollut ahkerasti käytössä testauksessa. Nyt kuitenkin muutaman vuoden sisään asiakkaat ovat enemmän tahtoneet käyttää järjestelmää lopputuotteissaan ja tuotannossaan. Ongelmana on ollut se, että ROS on alun perin suunniteltu vain yhdelle robotille. Siitä puuttuu monen robotin yhdenaikainen ohjattavuus, ja se vaatii tietokoneellisesti paljon järjestelmältä sekä tilalta. Ongelmaksi on myös koettu sen puutteet reaaliaikaisissa ohjelmissa, sekä heikkous toimia heikoissa tietoverkoissa.

ROS-käyttöjärjestelmän tulevaisuus on teollisuuden automaatiojärjestelmissä, missä käyttöjärjestelmän tulisi toimia useamman robotin kanssa. Ideana olisi muokata järjestelmän tietokoneellisia vaatimuksia pienemmäksi, esimerkiksi jokaiseen robottiin tulisi

ison tietokoneen tilalle vain Raspberry Pi. Myös ohjelmallisia muutoksia tehtäisiin muun muassa luotettavuuden suhteen reaaliaikaisissa toiminnoissa. Tämän avulla päästäisiin muun muassa itseajaviin autoihin laajennettua käyttöjärjestelmän käyttöä [30.]

6 Yhteenveto

Insinöörityön tarkoituksena oli tutustua ROS-käyttöjärjestelmään ja miettiä sen tulevaisuuden mahdollisuuksia. Pohjalla työlle oli Metropolia Ammattikorkeakoulun uudet robotit, jotka käyttävät ROS-käyttöjärjestelmää, sekä vähäinen informaatio kyseisestä järjestelmästä. Työ on hyvä pohja käyttöjärjestelmän käytön aloitukseen ja sen ymmärtämiseen. Koska työstä ei haluttu liian pitkää jouduttiin komentoja karsimaan. Kaikki komennot ovat kuitenkin listattuna ja hieman selitettynä yksityiskohtaisten ohjeiden jälkeen.

Työ luotiin yhteenvetona ja tiivistelmänä ROS-käyttöjärjestelmän käyttöohjeiden pohjalta. Lisäksi työhön haastattelin järjestelmän ylläpitäjä open Source Robotics Foundation -yrityksen Deanna Hoodia, joka toimii yrityksessä ROS ohjelmistoinsinöörinä. Häneltä sain hyvin tietoa heidän suunnitelmistaan ROS-käyttöjärjestelmälle.

Lähteet

1 About ROS. Verkkodokumentti. <http://www.ros.org/about-ros/>. Luettu 20.2.2017.

2 Fiki, Jusuf. 2016. Auto-navigation for robots. Implementation of ROS. Insinöörityö. Metropolia Ammattikorkeakoulu.

3 Installing and configuring your ROS environment. 2016. <http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment>. Luettu 20.1.2017.

4 Catkin conceptual overview. 2014. Verkodokumentti. <http://wiki.ros.org/catkin/conceptual_overview>. Luettu 20.2.2017.

5 Navigating the ROS filesystem. 2015. Verkkodokumentti. <http://wiki.ros.org/ROS/Tutorials/NavigatingTheFilesystem>. Luettu 20.2.2017.

6 Creating a ROS package. 2013. Verkkodokumentti. <http://wiki.ros.org/ROS/Tutorials/CreatingPackage>. Luettu 20.2.2017.

7 Using rosed to edit files in ROS. 2016. Verkkodokumentti. <http://wiki.ros.org/ROS/Tutorials/UsingRosEd>. Luettu 20.3.2017.

8 Building a ROS package. 2012. Verkkodokumentti. <http://wiki.ros.org/ROS/Tutorials/BuildingPackages>. Luettu 20.2.2017.

9 Catkin Workspaces. 2014. Verkkodokumentti. <http://wiki.ros.org/catkin/workspaces#Build_Space>. Luettu 1.3.2017.

10 Nodes. 2012. Verkkodokumentti. <http://wiki.ros.org/Nodes>. Luettu 3.3.2017.

11 Understanding ROS nodes. 2016. Verkkodokumentti. <http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes>. Luettu 3.3.2017.

12 Roscore. 2016. Verkkodokumentti. <http://wiki.ros.org/roscore>. Luettu 3.3.2017.

13 Package summary.2016. Verkkodokumentti. <http://wiki.ros.org/turtlesim>. Luettu 18.3.2017.

14 Understanding ROS topics. 2017. Verkkodokumentti. <http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics>. Luettu 15.3.2017.

15 Understanding ROS services and parameters. 2016. Verkkodokumentti. <http://wiki.ros.org/ROS/Tutorials/UnderstandingServicesParams>. Luettu 15.3.2017.

16 Package summary. 2011. Verkkodokumentti. <http://wiki.ros.org/rosservice>. Luettu 20.3.2017.

17 Using rqt_console and roslaunch. 2016. Verkkodokumentti. <http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch>. Luettu 18.3.2017.

18 Creating a ROS msg and srv. 2017. Verkkodokumentti. <http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv>. Luettu 28.2.2017.

19 Writing a simple publisher and subscriber (C++). 2016. Verkkodokumentti. <http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29>. Luettu 20.2.2017.

20 What is NodeHandle?. 2013. Verkkodokumentti. <http://answers.ros.org/question/68182/what-is-nodehandle/>. Luettu 20.3.2017.

21 Examining the simple publisher and subscriber. 2016. Verkkodokumentti. <http://wiki.ros.org/ROS/Tutorials/ExaminingPublisherSubscriber>. Luettu 20.2.2017.

22 Writing a simple service and client (c++). 2015. Verkkodokumentti. <http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28c%2B%2B%29>. Luettu 20.2.2017.

23 Examining the simple serice and client. 2016. Verkkodokumentti. <http://wiki.ros.org/ROS/Tutorials/ExaminingServiceClient>. Luettu 18.3.2017.

24 Recording and playing back data. 2015. Verkkodokumentti. <http://wiki.ros.org/ROS/Tutorials/Recording%20and%20playing%20back%20data>. Luettu 18.3.2017.

25 Package Summary. 2012. Verkkodokumentti. <http://wiki.ros.org/roswtf>. Luettu 18.3.2017.

26 Ubuntu install of ROS Kinetic. 2016. Verkkodokumentti. <http://wiki.ros.org/kinetic/Installation/Ubuntu>. Luettu 21.1.2017.

27 ROS Command-line tools. 2015. Verkkodokumentti. <http://wiki.ros.org/ROS/CommandLineTools>. Luettu 21.3.2017.

28 Rosmake. 2012. Verkkodokumentti. <http://wiki.ros.org/rosmake>. Luettu 21.3.2017.

29 Rospack. 2015. Verkkodokumentti. <http://wiki.ros.org/rospack>. Luettu 21.3.2017.

30 Rosstack. 2012. verkkodokumentti. <http://wiki.ros.org/rosstack>. Luettu 21.3.2017.

31 Hood Deanna. 2017. Ohjelmistoinsinööri, Open Source Robotics Foundation, San Francisco. Keskustelu 2.3.2017.