Robotski operacijski sustav ROS

Šarić, Martina

Master's thesis / Diplomski rad

2018

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Department of Mathematics / Sveučilište Josipa Jurja Strossmayera u Osijeku, Odjel za matematiku**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:126:293752

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-31**

mathos

Repository / Repozitorij:

Repository of School of Applied Mathematics and Informatics

zir.nsk.hr

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

dabar

DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni diplomski studij matematike i računarstva

Martina Šarić

Robotski operacijski sustav ROS

Diplomski rad

Osijek, 2018.

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni diplomski studij matematike i računarstva

Martina Šarić

Robotski operacijski sustav ROS

Diplomski rad

Mentor: izv. prof. dr. sc. Domagoj Matijević
Komentor: mag. math. Jurica Maltar

Osijek, 2018.

Zahvaljujem svojim roditeljima i baki na bezuvjetnoj podršci i ustrajnoj vjeri u moj uspjeh.
Njima posvećujem ovaj diplomski rad.
Hvala mojoj užoj i široj obitelji, prijateljima, kolegama iz klupe i Jurici.
Također, želim zahvaliti mentoru Domagoju Matijeviću i komentoru Jurici Maltaru bez kojih ovaj rad ne bi mogao nastati.

Sažetak. Ovaj diplomski rad istražuje operacijski sustav ROS (*engl. Robot Operating System*) čija je primarna svrha olakšati programiranje robota. To je softverska platforma otvorenog koda (*engl. open source*) koja omogućava razvojnim programerima da svoj kod i ideje dijele jednostavnije i brže. Tako izgrađena zajednica rezultira uštedom vremena pri izradi robota. ROS u svom okruženju pruža upotrebu svih potrebnih dijelova robotskog softverskog sustava koji bi inače zahtijevali odvojeno kodiranje i integriranje u cjelinu. On se sastoji od brojnih malih računalnih programa koji se međusobno povezuju i razmjenjuju poruke. Nakon što se opsežno upoznamo sa ROS-om i njegovim načinom rada, primjenit ćemo to znanje u praktične svrhe i izraditi robota koji se kreće u prostoru. Praktični dio rada sastoji se od dva dijela. Prvi dio uključuje izradu robota koji besciljno luta u prostoru i izbjegava prepreke. Drugi, kompleksniji dio, detaljno opisuje programiranje robota kojim se upravlja putem tipkovnice. Sve navedeno isprogramirano je u Python programskom jeziku koristeći ROS *rospy* knjižnicu.

Ključne riječi: ros, rospy, robot, robotski operacijski sustav, softver, navigacija, programiranje, Python, ROS

Abstract. This graduate thesis explores the Robot Operating System (ROS) whose primary purpose is to help to develop robots. ROS is an open source software platform which enables developers to share code and their ideas easier and faster. It provides all needed parts of the robotic software system in its environment that otherwise require separate coding and integration. It consists of a number of small computer programs that interconnect and exchange messages. After getting to know ROS and how to work with it, we will apply that knowledge for practical purposes and create a robot that moves in space. Practical part of this thesis consists of two sections. First section involves creating a robot that wanders aimlessly around the space and avoids obstacles. Second, more complex section, describes development of a robot that is driven via teleoperation in detail. All of the above mentioned will be developed using Python programming language and the ROS *rospy* library.

Key words: ros, rospy, robot, robot operating system, software, navigation, development, Python, ROS

Sadržaj

1 Uvod

Robotski operacijski sustav (ROS) je okvir (*engl. framework*) za izradu i razvoj robotskog softvera. To je zbirka alata, knjižnica i konvencija kojima je cilj pojednostaviti programiranje složenog i robusnog ponašanja robota. Isprogramirati robustan, općenamjenski softver za robota nije jednostavno. Primjerice, problem *dohvatiti predmet* (*engl. fetch an item problem*) se čini kao prilično trivijalan zadatak, ali poučiti robota kako preuzeti zadani predmet nije lako. Prvo, robot mora razumjeti zadatak koji treba izvršiti, bilo glasovnim putem ili putem nekog drugog korisničkog sučelja kao što je SMS, e-mail ili slično. Zatim, robot mora izraditi neku vrstu plana kako bi koordinirao traženje zadanog predmeta, što će vjerojatno zahtijevati prolazak kroz razne prostorije u zgradi, dizala i vrata. Robot treba pretražiti sobu kako bi prepoznao traženi predmet i dostavio ga na željeno mjesto. Svaki od tih pod-problema može imati proizvoljan broj složenih čimbenika koje treba uzeti u obzir prilikom programiranja robota. U praksi je najčešće programiranje robota namijenjenih za pomoć pri obavljanju nekog posla. Primjerice, u industriji, to su roboti na kotače kojima se može upravljati. Viličari kojima upravlja korisnik primjer su mobilnog robota koji je olakšao i ubrzao prijevoz tereta. Viličarom upravlja korisnik putem nekakvog sučelja. Najčešće su to tipke, dugmići i/ili štapić za upravljanje smjerom kretanja. Razvoj takvog stroja uvelike je uštedio vrijeme pri obavljanju posla, odnosno skratio je vrijeme izvršavanja posla i povećao je količinu izvršenog posla u jednoj radnoj smjeni. Također, ubrzao je utovar i istovar tereta, pa samim time i prijevoz robe od točke A do točke B, te pridonio razvoju industrije općenito. Danas, u želji za daljnim napredovanjem sve popularnije je programiranje autonomnog mobilnog robota. To je robot koji se samostalno kreće u prostoru, što znači da njime ne upravlja čovjek. Na konkretnom primjeru, u industriji, to bi bio robot na kotače koji bi preuzeo teret koji treba prevesti i sam ga dostavio na traženo mjesto. Svojim neovisnim kretanjem i izvršavanjem posla omogućila bi se dodatna radna snaga ukoliko za njom ima potrebe. Naime, roboti bi teoretski mogli raditi 24 sata na dan, odnosno sve dok imaju dovoljno struje za pogon. Roboti ne trebaju pauzu, ne jedu i fizički se ne umaraju. U praktičnom smislu idealni su i potrebni kao dodatna pomoć kod izvršavanja svih poslova koji se mogu automatizirati. Robotika danas, između ostalog, tu ideologiju koristi: programiranje robota u svrhu pomoći u radu kod automatiziranih poslova. Bilo da se radi o mobilnom robotu koji se kreće samostalno u prostoru ili se njime upravlja, potrebno je znati izraditi takvog robota. Ovaj rad namijenjen je svima onima koji žele znati i naučiti više o programiranju robota u robotskom operacijskom sustavu ROS. U prvom poglavlju ovog rada biti će opisana povijest robotskog operacijskog sustava ROS, njegove karakteristike i filozofski aspekti. Drugo poglavlje uključuje opis arhitekture i način rada sustava. Treće poglavlje posvećujemo praktičnom dijelu ovog diplomskog rada. Preciznije, programiranje Turtlebot robota kojim se upravlja putem tipkovnice na računalu u simuliranoj okolini uz pomoć simulacijskog ROS programa Gazebo. Turtlebot je mobilni robot na diferencijalni pogon. Izrađen je u Willow Garage-u od strane Melonee Wise i Tully Foote u studenom 2010. Odjel za matematiku u Osijeku posjeduje jednog Turtlebot 3 robota za edukativne svrhe kojeg je autorica ovog rada koristila kako bi proučila njegove mogućnosti i načine kojim se njime upravlja. Nakon proučavanja tehnologije za navigaciju, daljinsko upravljanje i simultano lokaliziranje i mapiranje koje su omogućene na Turtlebotu i savladanim tehnikama upravljanja robotom i navedenim tehnologijama, isti je predstavljala na Osijek Mini Maker Faire-u[1] 2018. godine. Nakon predstavljanja Tur-

[1]Originalni Maker Faire je održan u San Mateu, Kalifornija. Osijek Mini Maker Faire organizira Osijek Software City pod pokroviteljstvom Maker Media, izdavačem časopisa Make koji oko sebe okuplja globalni

tlebota na sajmu u Osijeku autorica se odlučuje na samostalnu implementaciju algoritama u ROS-u koji su prezentirani u praktičnom dijelu. Želja autorice ovog rada je približiti čitatelju robotski operacijski sustav i potaknuti ga na odluku da se sam upusti u izradu takvog robota.

makers pokret. To je kombinacija sajma, konferencije i festivala na kojem mogu sudjelovati hobisti, tehnološki entuzijasti, znastvenici te svi oni kojima je zajednička inventivnost i kreativnost. Autorica ovog rada je 05.svibnja.2018 sudjelovala na sajmu ispred Odjela za matematiku sa Turtlebot 3 robotom.

2 Robotski operacijski sustav

U službenoj distribuciji ROS-a postoje mnogobrojni softverski paketi koje su napisali i koje održava veliki broj razvojnih programera. ROS pruža mogućnost korištenja svih dijelova robotskog softverskog sustava kako ih ne bi morali zasebno pisati. Strukturiran je kao veliki broj malih računalnih programa koji brzo prenose poruke jedni drugima. To je struktura koja omogućuje stvaranje generičkih modula koji se primjenjuju na široke klase robotskih hardvera i softvera, te olakšava dijeljenje koda i ponovno korištenje u globalnoj robotičkoj zajednici. Takva je paradigma odabrana kako bi potaknula ponovno korištenje robotskog softvera izvan određenog robota i okruženja korištenog pri programiranju robota.

2.1 Povijest ROS-a

ROS je veliki projekt koji ima mnogo predaka i suradnika. Mnogi ljudi u istraživačkoj zajednici za robotiku osjetili su potrebu za otvorenom suradnjom. Razni projekti na Sveučilištu Stanford sredinom 2000.-e godine, kao što su STanford AI Robot (STAIR) i Personal Robots (PR) program, stvorili su vlastite prototipove fleksibilnih, dinamičkih softverskih sustava. Projekt ROS započeo je Morgan Quigley[2] 2007. godine na Sveučilištu Stanford. Iste te godine, Willow Garage, laboratorij za istraživanje robotike i inkubator tehnologije, uložio je značajne resurse kako bi proširio te koncepte mnogo dalje i isprogramirao dobro provjerene implementacije te je osmislio naziv ROS. Trudu su pridonijeli i nebrojeni istraživači koji su svoje vrijeme i stručnost usmjerili ka temeljnim ROS idejama i temeljnim programskim paketima. Od samog početka ROS se razvijao u više ustanova i za više robota. U početku se to činilo kao otegotna okolnost jer se smatralo da bi svim suradnicima bilo jednostavnije spremati programske kodove na isto mjesto. Tijekom godina, to se pokazalo kao jedna od najvećih snaga ekosustava ROS-a: svaka grupa može pokrenuti vlastiti ROS kod na vlastitim poslužiteljima i održati potpunu kontrolu i nadzor nad njim. ROS ekosustav sada se sastoji od desetaka tisuća korisnika širom svijeta, koji rade na domeni od malih hobi projekata do velikih industrijskih automatizacijskih sustava.

Vremenska crta

2009. objavljena je ROS 0.4 Mango Tango verzija, te je izrađen robot imena PR2.

2010. objavljena je ROS 1.0 verzija čije su mnoge značajke još uvijek u upotrebi, te je izrađen robot imena ROS C Turtle.

2011. objavljene su ROS Diamondback i ROS Electric Emys verzije.

2012. objavljene su Ros Fuerte, Ros Groovy Galapagos, te Open Source Robotics Foundation (OSRF) preuzima projekt ROS.

2013. objavljena je ROS Hydro Medusa.

[2]Morgan Quigley jedan je od osnivača OSRF-a (*Open Source Robotics Foundation*). Doktorirao je računalnu znanost u AI Labu na Sveučilištu Stanford 2012. godine.

2014. objavljena je ROS Indigo Igloo verzija. Ovo je bila prva izdana verzija za dugoročnu podršku (LTS), što znači da se ažuriranja i podrška pružaju dulje vrijeme (obično pet godina).

2015. objavljena je ROS Jade Turtle verzija.

2016. objavljena je ROS Kinetic Kame verzija. To je druga LTS verzija ROS-a.

2017. objavljena je ROS Lunar Loggerhead verzija.

2018. objavljena je dvanaesta inačica ROS-a, ROS Melodic Morenia.

Svaka verzija ROS-a naziva se ROS distribucija. Detaljnija povijest i vremenska crta projekta ROS dostupna je na `https://www.ros.org/history`.

2.2 Karakteristike ROS-a

ROS se sastoji od niza dijelova kao što su skup upravljačkih programa, zbirka osnovnih algoritama, skup alata za vizualizaciju i ROS ekosustav. Skup upravljačkih programa omogućava čitanje podataka sa senzora i slanje naredbi motorima i drugim aktuatorima u apstraktnom, dobro definiranom obliku. Podržan je široki raspon popularnih hardvera uključujući i sve veći broj komercijalno dostupnih robotskih sustava. Rastuća zbirka osnovnih robotskih algoritama omogućava gradnju mape svijeta, kretanje po njoj, prikazivanje i tumačenje podataka senzora, planiranje, manipuliranje predmetima i slično. ROS je postao vrlo popularan u istraživačkoj zajednici za robotiku i u njemu su dostupni najsuvremeniji algoritmi, računalna infrastruktura koja omogućuje premještanje podataka, povezivanje različitih komponenti složenog robota i ugradnja vlastitih algoritama. ROS je inherentno distribuiran i omogućuje jednostavnu podjelu radnog opterećenja na više računala. Veliki skup alata olakšava vizualizaciju stanja robota, ispravlja pogrešno ponašanje robota i pogrešno snimljene senzorske podatke. Konačno, ekosustav ROS-a obuhvaća opsežan skup resursa, kao što je *wiki*[3] koji dokumentira mnoge aspekte ROS okvira. ROS platformu karakteriziraju sljedeća svojstva:

Međuprocesna komunikacija Poruke prolaze kroz sučelje koje prenosi poruku za komunikaciju između dva programa ili procesa. Primjerice, kamera obrađuje sliku i pronalazi koordinate na slici, a zatim se te koordinate šalju procesu koji je odgovoran za daljnju obradu podataka. Ovo je jedna od značajki potrebnih za programiranje robota. Zove se međuprocesna komunikacija jer dva procesa komuniciraju međusobno.

Značajke poput operacijskog sustava Suprotno od onoga što ime kaže, ROS nije pravi operacijski sustav. To je meta operacijski sustav koji pruža neke funkcionalnosti operacijskog sustava. Te funkcionalnosti uključuju višedretvenost (*engl. multithreading*), kontrolu uređaja na niskoj razini, upravljanje paketima i apstrakciju hardvera.

[3]To je web mjesto na kojem je moguće postavljati pitanja i odgovore, podijeliti ono što je naučeno i pospješiti zajednicu korisnika i razvojnih programera: `https://wiki.ros.org/`

Upravljanje paketima Svaki paket ima izvorni kod, konfiguracijsku datoteku ili podatkovnu datoteku za određeni zadatak. Oni se mogu distribuirati i instalirati na druga računala.

Programska podrška i alati na visokoj razini ROS-a podržava popularne programske jezike koji se koriste u programiranju robota, uključujući C++, Python i Lisp. Postoji eksperimentalna podrška za jezike kao što su C#, Java, Node.js itd. Kompletan popis nalazi se na idućoj adresi: `https://wiki.ros.org/Client%20Libraries/`.

Klijentske knjižnice za navedene programske jezike Razvojni programer može dobiti ROS funkcionalnosti na navedenim jezicima. Na primjer, ako korisnik želi implementirati aplikaciju za Android koja koristi ROS funkcionalnost, može koristiti knjižnicu *rosjava*. ROS također pruža alate za izradu robotskih aplikacija. Pomoću njih možemo izgraditi mnoge pakete s jednom naredbom. Ova fleksibilnost pomaže programerima potrošiti manje vremena u stvaranju sustava za izgradnju svojih aplikacija.

Dostupnost biblioteka treće strane Okvir ROS-a je integriran s najpopularnijim knjižnicama treće strane. Na primjer, OpenCV[4] integriran je za robotski vid, a PC [5] je integriran za percepciju 3D robota. Ove knjižnice čine ROS snažnijim te omogućava programeru izgradnju moćnih aplikacija.

Gotovi algoritmi (***engl. off- the- shelf algorithms***) Ovo je korisna značajka obzirom da takvi algoritmi skraćuju vrijeme programiranja prototipa robota. ROS je implementirao popularne algoritme za robotiku kao što su: PID[6] algoritam , SLAM[7] algoritam za simultano lociranje i mapiranje i algoritmi za planiranje puta kao što su A*, Dijkstra[8] i AMCL[9].

Jednostavnost pri prototipiranju Jedna od prednosti ROS-a su navedeni gotovi algoritmi. Uz to, ROS koristi pakete koji se mogu lako upotrijebiti na bilo kojem robotu; lako je prototipirati osobi mobilni robot prilagodbom postojećeg paketa dostupnog u ROS repozitoriju. Konačno, takav način rada može smanjiti vrijeme razvoja robotskog softvera.

Ekosustav/podrška zajednici ROS programeri diljem svijeta aktivno razvijaju i održavaju ROS pakete. Velika podrška u zajednici uključuje programere koji postavljaju pitanja vezana za ROS na platformi za upite (`https://answers.ros.org/questions/`), raspravljaju o raznim temama i objavljuju vijesti na internet forumu ROS Discourse (`https://discourse.ros.org`) i slično.

[4]OpenCV Source Library je softverska knjižnica za razvoj strojnog učenja i računalnog vida. Detalji su dostupni na idućoj adresi: `https://opencv.org`

[5] Point Clouds je skup točaka, odnosno skup podataka u prostoru. Opširnije o PC na adresi `https://pointclouds.org`

[6] Proportional-Integral-Derivative upravljač, detalji dostupni na: `https://wiki.ros.org/pid`

[7]Simultaneous Localization and Mapping algoritam, detalji dostupni na: `https://wiki.ros.org/gmapping`

[8]A* i Dijkstra algoritmi su za planiranje puta, detalji dostupni na: `https://wiki.ros.org/globalplanner`

[9] Adaptive Monte Carlo Localization algoritam, detalji dostupni na: `https://wiki.ros.org/amcl`

Opsežni alati i simulatori ROS je izgrađen s mnogim naredbenim i GUI alatima za uklanjanje pogrešaka, vizualizaciju i simulaciju aplikacija u robotici. Ovi alati su vrlo korisni za rad s robotom. Na primjer, alat Rviz koristi se za vizualizaciju pomoću fotoaparata, laserskih skenera, inercijskih mjernih jedinica i tako dalje. Za rad s robotskim simulacijama, postoje simulatori poput Gazebo simulatora.

ROS jednadžba Projekt ROS može se definirati u jednoj jednadžbi:

$$\text{ROS} = \text{komunikacija} + \text{alati} + \text{mogućnosti} + \text{ekosustav}$$

Ukratko, ROS je kombinacija komunikacije, alata, sposobnosti i ekosustava/zajednice. ROS ima mnoge druge mogućnosti koje ćemo navesti u nastavku teksta.

2.3 Filozofski aspekti

Svi softverski okviri nastoje nametnuti svoju filozofiju razvoja svojim suradnicima, izravno ili neizravno. ROS slijedi UNIX filozofiju u nekoliko ključnih aspekata.

Aspekt partnerske mreže (*engl. peer-to-peer*) ROS sustavi sastoje se od brojnih malih računalnih programa koji se međusobno povezuju i kontinuirano mijenjaju poruke. Ove poruke putuju izravno iz jednog programa u drugi; ne postoji središnja usluga usmjeravanja.

Temeljenost na alatima Za razliku od mnogih drugih robotskih softverskih okvira, ROS nema kanonski integrirano razvojno i izvršno okruženje. Zadaci poput vizualizacije međusobnih veza sustava, grafičkog prikazivanja tokova podataka, stvaranja dokumentacije, zapisivanja podataka itd. obavljaju se zasebnim programima. To potiče stvaranje novih, poboljšanih implementacija koje se mogu razmjenjivati za potrebe implementacija prikladnijim nekoj specifičnoj domeni zadatka.

Višejezičnost Mnogi softverski zadaci postižu veću učinkovitost u skriptnim jezicima kao što su Python ili Ruby. Međutim, postoje slučajevi kada zahtjevi za izvedbom diktiraju korištenje bržih jezika, poput C++- a. Postoje i razni razlozi zbog kojih neki programeri preferiraju jezike poput LISP- a ili MATLAB- a. ROS softverski moduli mogu se pisati na bilo kojem jeziku pomoću klijentskih knjižnica.

Ponovna upotreba ROS konvencije potiču suradnike da izrađuju samostalne knjižnice (*engl. stand-alone libraries*), a zatim omotaju (*engl. wrap*) te knjižnice tako da šalju i primaju poruke sa drugim ROS modulima. Ovakav dodatni sloj namijenjen je za omogućavanje ponovne upotrebe softvera izvan ROS- a, za druge aplikacije, a uvelike pojednostavljuje izradu automatiziranih testova pomoću standardnih alata za kontinuiranu integraciju.

Besplatan i otvoren Jezgra ROS-a izdana je pod BSD licencom, koja omogućuje komercijalnu i nekomercijalnu upotrebu. ROS prenosi podatke između modula koristeći međuprocesnu komunikaciju (*engl. Inter-Process Communication* (*IPC*)), što znači da sustavi koji se grade pomoću ROS- a mogu imati različito licenciranje različitih komponenti. Primjerice, komercijalni sustavi većinom imaju nekoliko modula zatvorenog koda koji komuniciraju s velikim brojem modula otvorenog koda. Akademski i hobi projekti često su potpuno

otvoreni. Svi ovi slučajevi programiranja funkcioniraju bez poteškoća kao cjelina i valjani su pod licencom ROS- a.

3 Arhitektura ROS-a

Robotski operacijski sustav dijelimo na 3 razine (*engl. level*):

- Datotečni sustav (*engl. Filesystem Level*)
- Razina nadzora i upravljanja (*engl. Computation Graph Level*)
- Razina zajednice (*engl. Community Level*)

3.1 Datotečni sustav

Datotečni sustav služi za oblikovanje osnovne strukture i stvaranje minimalnog broja datoteka i mapa koje su potrebne za rad sustava.

Metapaketi (***engl. Metapackages***) Metapaketi su specijalizirani paketi koji služe samo za predstavljanje skupine povezanih paketa. Najčešće se koriste kao oznaka koja osigurava kompatibilnost među paketima koji se modificiraju tijekom programiranja i implementacije.

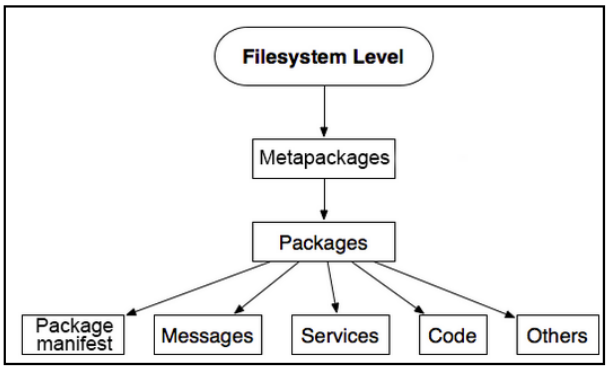
Paketi (***engl. Packages***) Paketi su glavna jedinica organizacije softvera u ROS-u. Paket može sadržavati ROS izvršne procese (čvorove), ROS knjižnicu o kojima ovisi, skupove podataka, konfiguracijske datoteke ili specifično odabrane podatke koje je korisno grupirati zajedno. Paketi su najviša stavka izrade i izdavanja u ROS-u.

Manifesti paketa (***engl. Package Manifests***) Manifesti paketa (*package.xml*) uključuju metapodatke o paketu: naziv, verzija, opis, informacije o licenci, ovisnosti i druge meta informacije kao što su izvezeni paketi.

Repozitoriji (***engl. Repositories***) Zbirka paketa koji dijele zajednički sustav upravljanja inačicama (*engl. Version Control System- VCS*). Paketi koji dijele VCS dijele istu verziju i mogu biti objavljeni zajedno. Repozitoriji također mogu sadržavati samo jedan paket.

Opis poruke (***engl. Message*** (***msg***) ***types***) Opisi poruka pohranjeni u `my_package/msg/MyMessageType.msg` definiraju strukture podataka za poruke poslane u ROS-u.

Opis usluge (***engl. Service*** (***srv***) ***types***) Opisi usluga pohranjeni u `my_package/srv/MyServiceType.srv` definiraju strukture podataka za komunikaciju za usluge u ROS-u.



Slika 1: Shematski prikaz razine datotečnog sustava (*engl. Filesystem level*)

3.2 Razina nadzora i upravljanja

Sustav za nadzor i upravljanje predstavlja partnersku mrežu (*engl. peer-to-peer network*) ROS procesa koji obrađuju podatke - ROS graf. Razina nadzora i upravljanja zadužena je za komunikaciju između procesa i sustava. Osnovni koncept takvog sustava su čvorovi, glavni čvor ili glavni poslužitelj (*engl. Master*), poslužitelj parametara, poruke, usluge, teme i spremnici. Svi oni zajedno šalju podatke sustavu za nadzor i upravljanje na različite i unaprijed definirane načine. Takav koncept sustava implementiran je u *ros_comm* repozitoriju.

Čvorovi (***engl. nodes***) Čvorovi u ROS grafu predstavljaju procese. Naprimjer, sustav upravljanja robotom obično sadrži razne čvorove: čvor za kontrolu senzora, čvor za upravljanje motorom kotača, čvor za lokalizaciju, čvor odgovoran za planiranje puta, čvor za grafički prikaz sustava i tako dalje. ROS čvor piše se pomoću ROS klijentske knjižnice, kao što je *roscpp* ili *rospy*.

Glavni poslužitelj ili Master ROS Master omogućuje registraciju imena (*engl. names*) i usluga u sustav i pretraživanje po njemu. Također, postavlja veze među čvorovima. Ako ROS Master ne postoji u sustavu nadzora i upravljanja, tada nije moguće komunicirati s čvorovima, uslugama, porukama i ostalim članovima sustava. U distribuiranom sustavu glavni poslužitelj nalazi se na jednom računalu, a rad u mreži i komunikaciju među članovima mreže možemo izvršiti na njemu ili nekom drugom računalu.

Poslužitelj parametara (***engl. Parameter Server***) Poslužitelj parametara omogućava pohranjivanje podataka na središnjem mjestu (*engl. central location*). Pomoću ovog poslužitelja moguće je konfigurirati čvorove tijekom rada ili mijenjati radne parametre čvora. Poslužitelj parametara dio je glavnog poslužitelja.

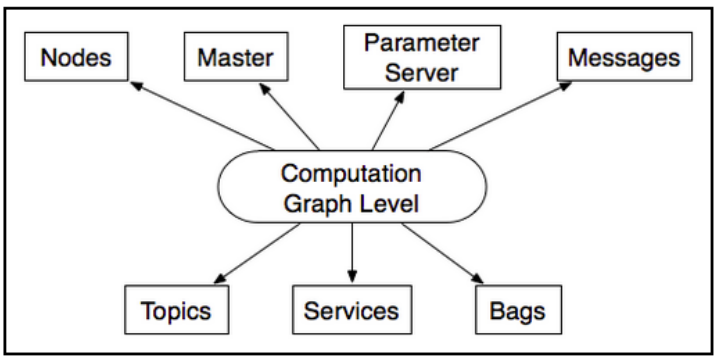
Poruke (***engl. Messages***) Čvorovi međusobno komuniciraju prosljeđivanjem poruka. Poruka je jednostavna podatkovna struktura koja sadrži nekakav podatak. Podržane su standardne primitivne vrste (integer, floating point, boolean itd.), kao i polja primitivnih tipova.

Teme (***engl. Topics***) Poruke se izmjenjuju na principu objavljivanje- pretplata. Čvor objavljivač (*engl. publisher*) šalje poruku objavljivanjem određene teme. Tema je naziv koji se koristi za prepoznavanje sadržaja poruke. Onaj čvor koji je zainteresiran za određenu vrstu poruka na odgovarajućoj temi pretplatit će se na nju kao čvor pretplatnik (*engl. subscriber*). Može postojati više istovremenih objavljivača (izdavača) i pretplatnika za jednu temu, a jedan čvor može objavljivati i/ili pretplatiti se na više tema. Općenito, izdavači i pretplatnici nisu svjesni postojanja drugih. Važno je da su nazivi tema jedinstveni kako bi se izbjegao sukob tema s istim imenom.

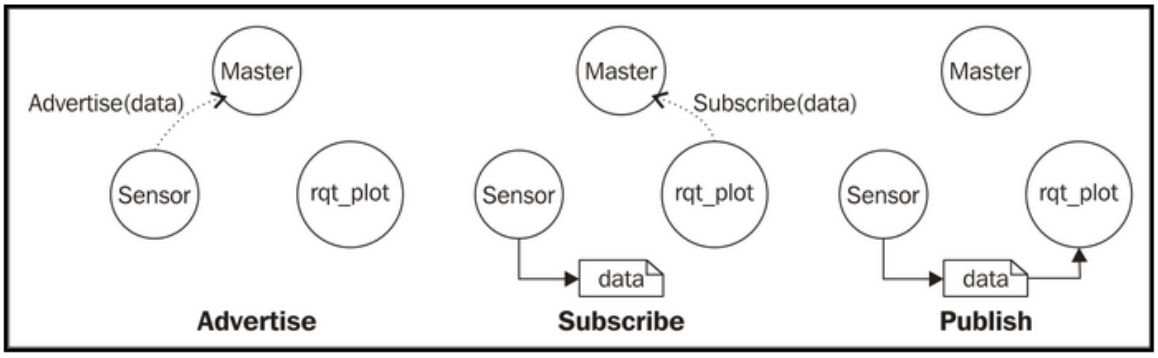
Usluge (***engl. Services***) Model objavljivanja/pretplate je vrlo fleksibilna komunikacijska paradigma, ali njegov višestruki i/ili jednosmjerni prijenos informacija nije prikladan za interakciju na relaciji zahtjev/odgovor (*engl. request and response*), koja je često potrebna u distribuiranom sustavu. Zahtjev/odgovor obavlja se putem usluga koje definiraju nekoliko struktura poruka: jedna za zahtjev i jedna za odgovor. Poslužiteljski čvor nudi uslugu pod imenom, a klijent koristi uslugu slanjem poruke zahtjeva i čeka odgovor. Kada objavljujemo teme, šaljemo podatke na mnogo načina, ali kada šaljemo zahtjev čvoru i želimo odgovor iz čvora, ne možemo to učiniti sa temama. Usluge nam pružaju mogućnost interakcije s čvorovima. Također, usluge moraju imati jedinstveno ime. Kada čvor ima uslugu, svi čvorovi mogu komunicirati s njom, zahvaljujući ROS klijentskim knjižnicama.

Spremnici (***engl. Bags***) Spremnici su format za spremanje i reprodukciju podataka ROS poruka. Oni su važan mehanizam za pohranu podataka kao što su podaci sa senzora i nužni su za programiranje robota i testiranje algoritama.

ROS Master djeluje kao servis imena u ROS grafu. On pohranjuje informacije o registriranim tema i uslugama u sustavu ROS čvorova. Čvorovi komuniciraju s Master- om kako bi prijavili svoje podatke u sustav i uspješno se registrirali u mrežu. Budući da čvorovi komuniciraju s Master-om oni mogu primati informacije o drugim registriranim čvorovima i povezati se s njima po potrebi. Čvorovi se izravno povezuju s drugim čvorovima, a Master čvor samo pruža informacije o pretraživanju, slično DNS poslužitelju. Čvorovi koji se pretplate na temu zatražit će veze s onima koji objavljuju tu temu, te će uspostaviti vezu preko dogovorenog protokola veze. Najčešći protokol koji se koristi u ROS-u naziva se TCPROS i on koristi standardne TCP/IP pristupne točke. Ovakva arhitektura omogućava modularnost; odvojeno izvršavanje operacija, gdje su imena primarna sredstva pomoću kojih se mogu graditi veći i složeniji sustavi. Imena imaju vrlo važnu ulogu u ROS-u: čvorovi, teme, usluge i parametri imaju jedinstvena imena. Svaka ROS klijentska knjižnica podržava remapiranje imena putem naredbenog retka, što znači da se program tijekom izvođenja može prilagoditi za rad na drugom dijelu ROS grafa.



Slika 2: Shematski prikaz razine nadzora i upravljanja (*engl. Computation Graph level*)



Slika 3: Shematski prikaz komunikacije među čvorovima

3.3 Razina zajednice

Razina zajednice, omogućava razmjenu znanja, algoritama i kodova među razvojnim programerima. Ova razina je iznimno važna kao i kod većine softverskih projekata otvorenog koda. Snažna zajednica olakšava shvaćanje softvera onima koji se sa njime tek upoznaju, pomaže u rješavanju problema s kojima se članovi zajednice susreću i potiče rast zajednice. Koncept ROS zajednice su ROS resursi koji omogućuju razmjenu softvera i znanja u zajednici. Ti resursi uključuju:

Distribucije ROS distribucije su zbirke verzioniranih paketa koje možemo instalirati. Distribuiranje u ROS-u igra sličnu ulogu kao u Linux distribucijama: one olakšavaju instaliranje zbirki softvera, te sadrže kontrolirane inačice objavljenog sustava i održavaju se kroz vrijeme.

Repozitoriji ROS se oslanja na zajedničku mrežu spremišta za kodove gdje različite institucije mogu isprogramirati i objaviti vlastite komponente softvera robota.

ROS wiki ROS wiki je glavni forum za dokumentiranje informacija o ROS- u. Svatko se može prijaviti i pridonijeti vlastitom dokumentacijom, dati ispravke ili ažurirati informacije, pisati vodiče i još mnogo toga.

Bug Ticket System Sustav za prijavu pronađenih pogrešaka i/ili problema ili za predlaganje novih značajki.

Mailing Lists Popis korisnika ROS- a primarni je komunikacijski kanal o novim ažuriranjima za ROS, kao i forum za postavljanje pitanja o ROS softveru.

ROS Answers Korisnici mogu postavljati pitanja na forumima koristeći ovaj resurs.

Blog Redovita ažuriranja, fotografije i vijesti dostupne su na adresi `https://www.ros.org/news`.

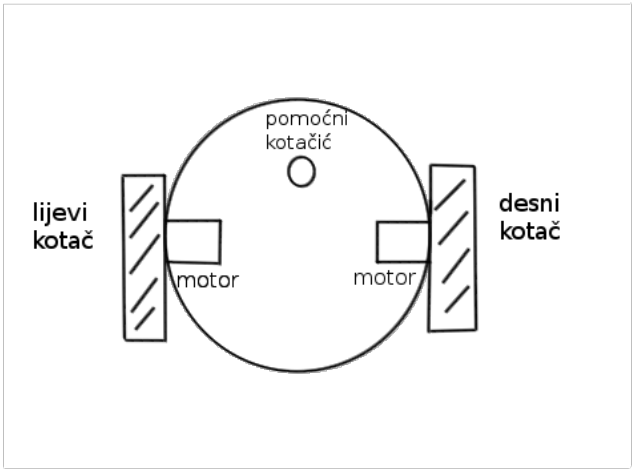
4 Praktični rad

4.1 Mobilni robot

Mogućnost kretanja temeljna je sposobnost mnogih robota. Međutim, široko govoreći, mobilni robot čini čvrsta baza (tijelo) robota zajedno sa kotačima, odnosno aktuatorima koji omogućavaju upravljanje robotom tako da ga pokreću. Mobilni roboti postoje u raznim oblicima i veličinama. Iako je popularno i intrigantno vidjeti robota koji se kreće kao čovjek (pomoću nogu; *engl. legged robots*), roboti se najčešće kreću na kotačima. Platforme na kotačima su često jednostavnije za dizajn i izradu. Dodatno, kotači su energetski učinkovit način kretanja na vrlo glatkim površinama koje su uobičajene u umjetnim sredinama poput unutarnjih podova, vanjskih podova ili pločnika.

4.1.1 Robot na diferencijalni pogon

Najjednostavnija konfiguracija mobilnog robota je robot na diferencijalni pogon. On se sastoji od dva kotača kojima se neovisno upravlja. Stoga, robot se pomiče unaprijed ako se oba kotača okreću istom brzinom prema naprijed i okreće se u mjestu ako se istom brzinom jedan kotač okreće prema naprijed, a drugi unatrag. Roboti na diferencijalni pogon često imaju jedan ili više kotačića kojima se ne upravlja, odnosno koji su bez pogona. Oni se slobodno kreću kako bi podržali stabilnost robota, baš kao i kotačići na uredskoj stolici. Vrlo stabilni roboti su jednostavni za modeliranje i upravljanje, a ako robota isključimo u bilo koje vrijeme kretanja, možemo biti sigurni da robot neće pasti.



Slika 4: Shematski prikaz mobilnog robota na diferencijalni pogon s dva kotača

Shema mobilnog robota na diferencijalni pogon može se proširiti i na više od dva kotača. Često je slučaj upravljanje sa četiri kotača i šest kotača kod kojih svi kotači na lijevoj strani robota djeluju zajedno, i svi kotači s desne strane djeluju zajedno. Kako se broj kotača povećava, primjerice upravljanje sa šest kotača, tada su obično kotači sa svake strane zajedno povezani u "jedan" kotač, kao kod bagera tzv. bagera gusjeničara.

Slika 5: Bager gusjeničar

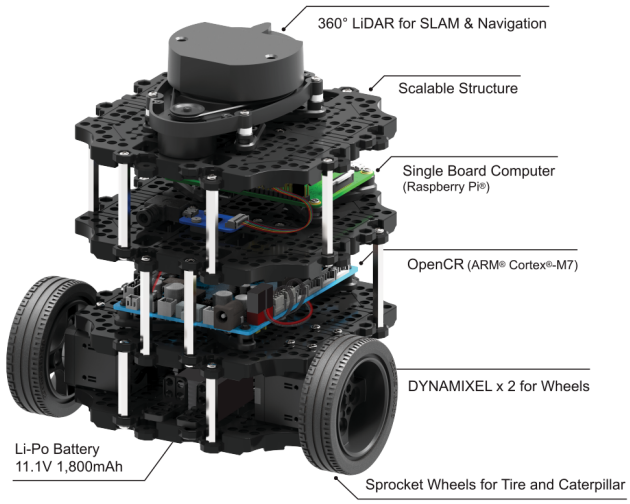
Sve dosad opisane platforme mogu se sažeti kao ne-holonomske platforme, što znači da se ne mogu kretati u svim smjerovima u bilo kojem trenutku. Na primjer, niti platforme diferencijalnih pogona niti Ackerman platforme (npr. model platformi za automobile) ne mogu se kretati bočno. Za postizanje holonomskog kretanja mobilnog robota svaki kotač kojim se upravlja treba dva motora: jedan motor okreće kotač naprijed i unatrag, a drugi motor upravlja kotačem oko njegove okomite osi. Takav način omogućava platformi kretanje u bilo kojem smjeru, odnosno svesmjerno kretanje. Iako je složenija za izgradnju i

održavanje, takva platforma pojednostavljuje planiranje kretanja. Zamislimo jednostavnost bočnog parkiranja ako bismo mogli voziti paralelno u parkirališno mjesto. Kao poseban slučaj, kada se robot treba kretati samo na vrlo glatkim površinama, holonomska platforma može se graditi pomoću švedskih kotača. To su kotači u kojima svaki kotač ima niz valjaka na svom rubu, zakrivljenih 45 stupnjeva prema ravnini kotača. Koristeći ovu shemu, kretanje u bilo kojem smjeru je moguće u svakom trenutku, koristeći samo četiri aktuatora. No međutim, zbog malog promjera valjaka, prikladan je samo za vrlo glatke površine poput tvrdih podova ili iznimno kratkih sagova.

Slika 6: Mecanum kotač (Švedski kotač) za svesmjerno kretanje robota

4.2 Mobilni robot na diferencijalni pogon

Turtlebot 3 je mobilni robot na diferencijalni pogon dizajniran tako da lako podučava one koji se tek upoznaju sa robotskim operacijskim sustavom. TurtleBot je standardna platforma ROS-a, a najpopularnija je platforma među programerima i studentima. TurtleBot 3 se može prilagoditi na različite načine ovisno o tome kako rekonstruiramo mehaničke dijelove i koristimo dodatne dijelove kao što su računalo i senzori. Sastoji se od mobilne baze na dva kotača uz dodatni pomoćni kotačić. Komponente Turtlebot 3 robota čine: laserski skener- LiDAR, mini računalo Raspberry Pi, Open CR (AM Cortex M7), dva Dinamyxel kotača sa motorima i napajanje, odnosno litijska baterija od 11.1V jakosti 1800mAh za bežično napajanje.



Slika 7: Turtlebot 3 mobilni robot na diferencijalni pogon

Budući da je jedan od dizajnerskih ciljeva ROS-a omogućiti ponovnu upotrebu softvera, ROS softver koji komunicira s mobilnim platformama gotovo uvijek koristi Twist poruke za izražavanje linearne i kutne brzine. Iako se čini lakšim kretanje mobilnog robota izraziti korištenjem brzina lijevog i desnog kotača, prirodnije je razumjeti kretanje robota općenito. U tom slučaju važan nam je položaj robota. Na taj način razmišljamo o brzini robota, a ne o brzini njegovog lijevog i desnog kotača. Pitamo se koliko brzo se robot kreće; linearna brzina i koliko brzo se okreće; kutna brzina. Tako linearna brzina i kutna brzina predstavljaju brzinu robota i otkrivaju nam u kojem smjeru se robot kreće. Kako bi se robot nesmetano kretao u željenom smjeru i željenom brzinom važno je biti u mogućnosti pristupiti podacima koji nam govore što robot "vidi".

4.2.1 Senzorski podaci

Robot može koristiti razne senzore kao što su: ultrasonični senzori, infracrveni senzori, laserski skener, kameru i slično kako bi detektirao nalazi li se ispred njega slobodan put ili prepreka. Čitanjem podataka sa senzora dobivamo korisne informacije o svijetu kojim se robot kreće. U ROS-u je prilično jednostavno doći do tih podataka. Naime, pristupom određenim temama koje sadrže poruke sa podacima možemo lako doći do traženih informacija. Tako, ako želimo pristupiti podacima sa senzora potrebno je pretplatiti se na temu koja odašilje pripadajuće podatke, odnosno povezati se sa čvorom koji izdaje odgovarajuću temu. Ponovimo, svaki ROS čvor može predstavljati izdavača i/ili pretplatnika. U ovom slučaju, čvor koji je zaslužan za odašiljanje podataka sa senzora izdaje te podatke kroz određenu temu i šalje ih putem određenog tipa poruka. U slučaju Turtlebot 3 robota koji ima laserski skener, podacima koje skener očitava možemo pristupiti putem ROS paketa sensormsgs koji je unaprijed definiran za poruke laserskog senzora i nazivaju se LaserScan poruke.

sensor_msgs/LaserScan Message

File: sensor_msgs/LaserScan.msg

Raw Message Definition

```
# Single scan from a planar laser range-finder
#
# If you have another ranging device with different behavior (e.g. a sonar
# array), please find or create a different message, since applications
# will make fairly laser-specific assumptions about this data

Header header            # timestamp in the header is the acquisition time of
                         # the first ray in the scan.
                         #
                         # in frame frame_id, angles are measured around
                         # the positive Z axis (counterclockwise, if Z is up)
                         # with zero angle being forward along the x axis

float32 angle_min        # start angle of the scan [rad]
float32 angle_max        # end angle of the scan [rad]
float32 angle_increment  # angular distance between measurements [rad]

float32 time_increment   # time between measurements [seconds] - if your scanner
                         # is moving, this will be used in interpolating position
                         # of 3d points
float32 scan_time        # time between scans [seconds]

float32 range_min        # minimum range value [m]
float32 range_max        # maximum range value [m]

float32[] ranges         # range data [m] (Note: values < range_min or > range_max should be discarded)
float32[] intensities    # intensity data [device-specific units].  If your
                         # device does not provide intensities, please leave
                         # the array empty.
```

Slika 8: Predefinirana struktura LaserScan poruke iz sensor_msgs paketa

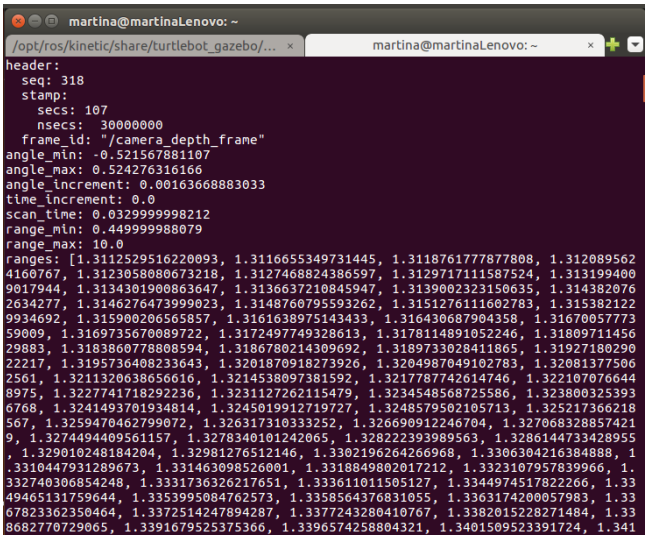
4.2.2 Primjena senzorskih podataka u programu

Kako bismo započeli s upotrebom podataka sa senzora možemo jednostavno pozvati ispis LaserScan poruke na konzolu kako bi bolje razumjeli informacije koje primamo. Prvo ćemo pokrenuti simulaciju Turtlebota u Gazebo simulatoru kako bi simulirali kretanje robota. To ćemo učiniti upisivanjem sljedeće naredbe u terminal:

```
user@hostname$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

Sada, kako bi na konzolu ispisali informacije upotrijebiti ćemo sljedeću naredbu u novom terminalu:

```
user@hostname$ rostopic echo scan
```



Slika 9: Ispis senzorskih podataka na terminalu

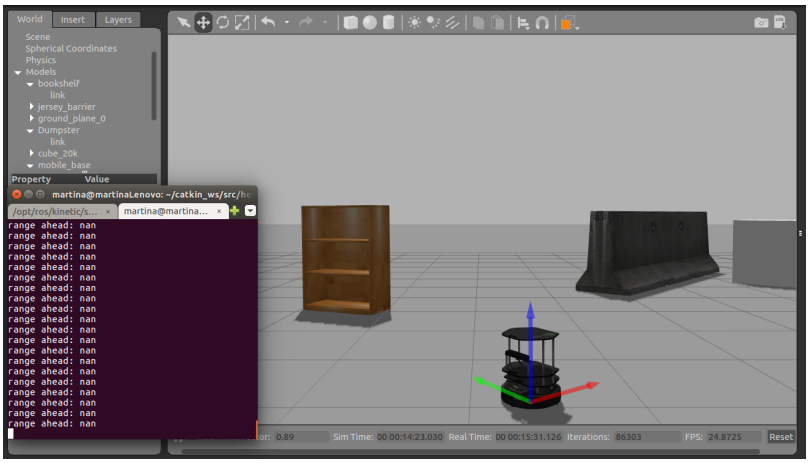
Dobivamo kontinuirani tok teksta koji predstavlja LaserScan poruku koju odašilje sensormsgs čvor. Iz poruke između ostalog možemo iščitati vrijeme u kojem je snimljen po-

datak, minimalnu udaljenost koju skener očitava, maksimalnu udaljenost, te polje *ranges* u kojem su upisane udaljenosti svih prepreka koje se nalaze oko robota. Ako želimo izračunati udaljenost do najbliže prepreke izravno ispred robota koju otkriva skener možemo odabrati srednji element niza. Sljedeći mali program kreira ROS čvor koji ispisuje udaljenost do prepreke koja se nalazi ispred robota i pokazuje nam kako je lako povezati se sa tokovima podataka u ROS-u i procesuirati ih u Pythonu:

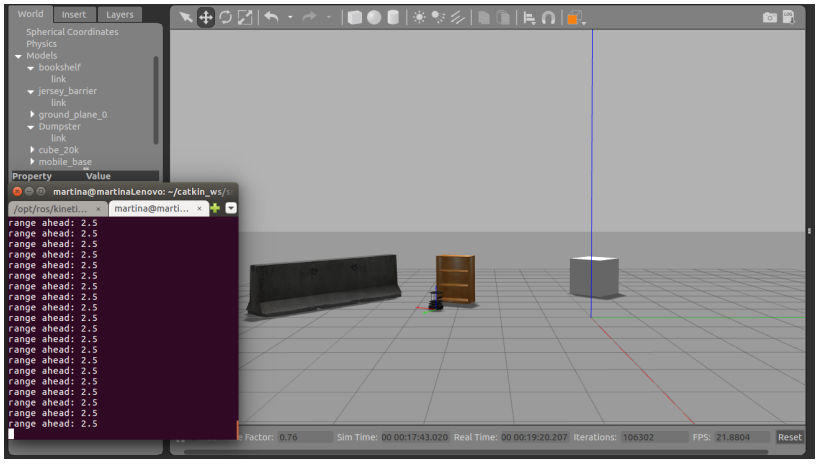
Listing 1: range_ahead.py

```
#!/usr/bin/env python
import rospy
from sensor_msgs.msg import LaserScan

# Funkcija se poziva svaki puta kada postoji ocitanje na lidaru
# Po jednom za svaku rotaciju skenera

def scan_callback(msg):
  range_ahead = msg.ranges[len(msg.ranges)/2]
  print "range ahead: %0.1f" % range_ahead

# Kreiranje cvora range_ahead
rospy.init_node('range_ahead')

# Deklariranje pretplatnika na temu imena scan i poruke LaserScan
# Za svako ocitanje pozovi funkciju scan_callback
scan_sub = rospy.Subscriber('scan', LaserScan, scan_callback)

# Ne zatvaraj petlju dok ne ocitas ctrl+C na tipkovnici
rospy.spin()
```

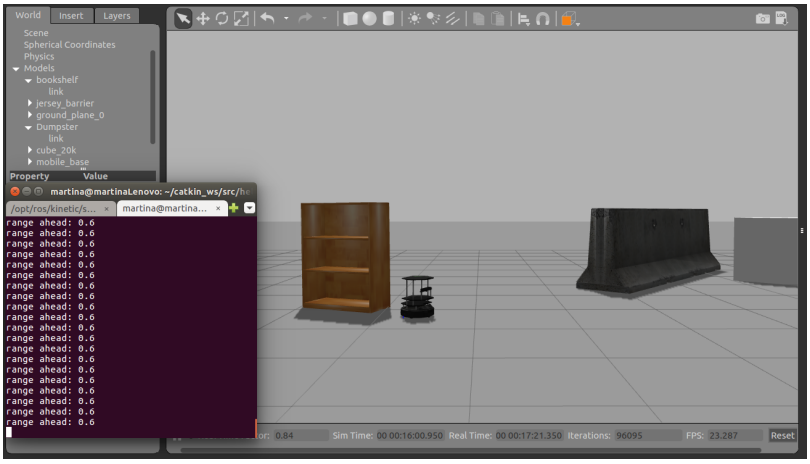
Funkcija *scan_callback*() poziva se svaki puta kada nova poruka pristigne, a zatim ispisuje udaljenost do objekta (u metrima) koji se nalazi izravno ispred robota tako što bira srednji element niza. Konačno, kada znamo pristupiti podacima sa senzora i koristiti ih na robotu, izradit ćemo robota koji besciljno luta i izbjegava prepreke. Robot će se kretati prema naprijed, a kada naiđe na prepreku koja se nalazi na njegovoj putanji, tada će stati, zarotirati se i ukoliko ima slobodan prostor za kretanje nastavit će se kretati prema naprijed. Također, postoji mogućnost da su podaci sa senzora nevaljani ako senzor neispravno očitava objekte ili se senzor mehanički isključio i ne šalje podatke. U tom slučaju ćemo mjerenjem vremena u kojem je robot u pokretu (kreće se prema naprijed), odnosno robot se ne kreće prema naprijed, zarotirati robota pod pretpostavkom da neki objekt sprječava njegovo kretanje i pokrenuti prema naprijed. Ukoliko je put slobodan, robot će se nastaviti kretati, a ako nije, biti će mu onemogućeno kretanje pa ćemo ponoviti navedeni korak.



(a) Slobodan put

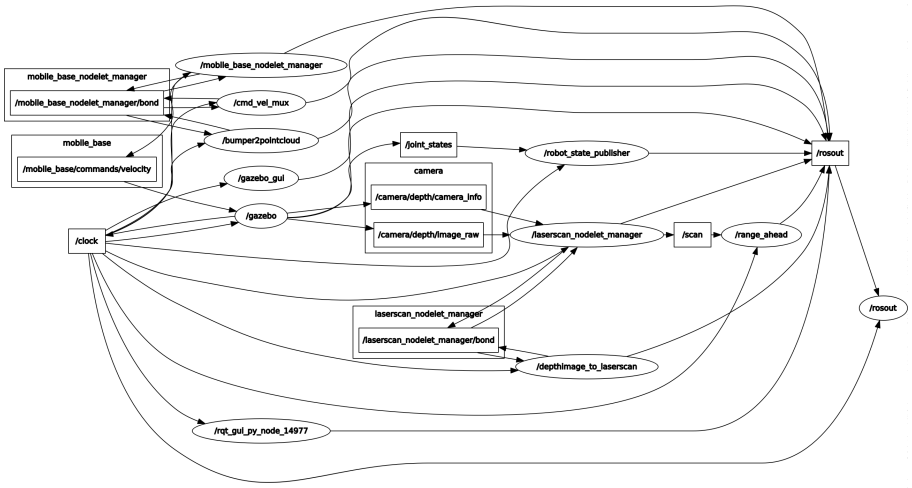


(b) Prepreka na udaljenosti 2.5 m



(c) Prepreka na udaljenosti 0.6 m

Slika 10: Detekcija prepreke na različitim udaljenostima u simulacijskom programu Gazebo



Slika 11: Dijagram razine nadzora i upravljanja tijekom izvršavanja programa *range_ahead.py*

4.2.3 Robot koji besciljno luta i izbjegava prepreke

Program započinjemo sa uvozom *rospy* knjžnice i ROS poruka koje će nam trebati. ROS koristi pojednostavljeni jezik za opisivanje poruka kako bi opisao vrijednosti podataka koje ROS čvorovi objavljuju. Uz ovaj opis, ROS može generirati pravi izvorni kod za poruke na nekoliko programskih jezika. U ovom slučaju, Twist poruke za upravljanje robotom i njegovim brzinama te LaserScan poruke za čitanje podataka sa senzora i korištenje istih. Koristimo globalnu varijablu *g_range_ahead* u koju ćemo spremiti minimalan raspon udaljenosti do objekta (u metrima) koji naš skener očitava. Stoga je procedura koju izvršava *scan_callback*() funkcija vrlo jednostavna. Ona samo kopira sva očitana mjerenja i sprema ih u *g_range_ahead* globalnu varijablu. Iako se ovakav način kodiranja u velikim i zahtjevnim sustavima ne preporuča, u ovom slučaju će raditi sasvim korektno.

Listing 2: wander.py

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

import rospy
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan

# Funkcija se poziva svaki puta kada postoji ocitanje
# Spremamo najmanju udaljenost do ocitanog objekta - najbliza prepreka
# Bez obzira u kojem smjeru se prepreka nalazi
# Potrebna nam je globalna varijabla g_range_ahead kako bi joj mogli pristupiti
    izvan funkcije

def scan_callback(msg):
   global g_range_ahead
   g_range_ahead=min(msg.ranges)

g_range_ahead=1 # za pocetak bilo koja vrijednost
# Deklariranje pretplatnika na temu imena 'scan' klase LaserScan
```

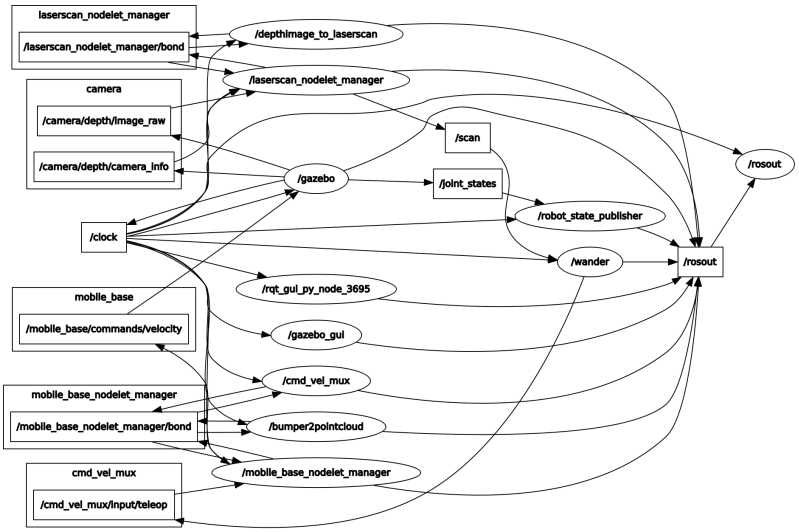
```
scan_sub=rospy.Subscriber('scan',LaserScan,scan_callback)
# Deklariranje izdavaca na temu cmd_vel_mux/input/teleop klase Twist
cmd_vel_pub=rospy.Publisher('cmd_vel_mux/input/teleop',Twist,queue_size=1)
# Deklariranje cvora wander
rospy.init_node('wander')

state_change_time=rospy.Time.now()

# kretanje prema naprijed(driving_forward): forward(true) vs. vrti se u mjestu
    (false)
#   TRUE: dok x sekundi ne prode ili se priblizis prepreci, kreci se prema naprijed
#   FALSE: dok y sekundi ne prode, vrti se u mjestu
driving_forward=True

rate=rospy.Rate(10)

while not rospy.is_shutdown():
   if driving_forward:
      if(g_range_ahead<0.9 or rospy.Time.now()>state_change_time):
         driving_forward=False
         state_change_time=rospy.Time.now()+rospy.Duration(5)
  # provjeri nalazi li se sto na udaljenosti manjoj od x ili ako je vrijeme koje
      provodis u mjestu isteklo, onda se pocni vrtiti u mjestu
   else: # ne krecemo se prema naprijed
      if rospy.Time.now()>state_change_time:
         driving_forward=True # zavrsi s vrtnjom u mjestu, kreci se prema naprijed
         state_change_time=rospy.Time.now()+rospy.Duration(50)
# Kreiraj praznu/nul Twist() poruku. Napomena: u svakom prolasku petlje kreira se
    nova poruka
   twist=Twist()
  # Ako se krecemo prema naprijed postavi linearnu i kutnu brzinu
   if driving_forward:
      twist.linear.x=0.5
   else:
      twist.angular.z=0.5
 # Objavi cmd_vel sa zeljenim kretanjem
   cmd_vel_pub.publish(twist)
# Zaustavi petlju na 1/rate sekudi
   rate.sleep()
```

Nakon toga, nastavljamo sa pretplatom na *scan* temu (pristupamo podacima sa laserskog skenera) i izdajemo poruke na *cmd_vel* temu (šaljemo brzine robotu). Za upravljanje našim robotom potrebne su nam dvije varijable: *state_change_time* i *driving_forward*. U uvjetnoj *if* petlji zapravo upravljamo dvama stanjima u kojima se robot nalazi: *driving_forward* ili *not driving_forward*. Ako se robot kreće prema naprijed (*driving_forward*), tada se nastavlja kretati tako sve dok ne naiđe na prepreku koja se nalazi na udaljenosti manjoj od 0.8 metara ili sve dok se robot iz nekog razloga nalazi u nepokretnom stanju 30 sekundi. Ako je robot nepokretan 30 sekundi, tada algoritam postavlja robota u stanje u kojem se ne kreće prema naprijed (*not driving_forward*) te po uvjetu *not driving_forward* rotira robota na 5 sekundi i vraća ga nazad u stanje kretanja prema naprijed. Treća varijabla koja se naziva *rate* nam omogućava da se naša glavna petlja odvija fiksnom frenkvencijom

postavljenoj na 10Hz.

```
rate = rospy.Rate (10)
```

Ova linija koda kreira objekt *rate* tipa *Rate*. Uz pomoć metode *sleep*() omogućeno je kontinuirano odvijanje petlje u zadanoj frekvenciji od 10Hz. Dakle, očekujemo da zadanom petljom u našem algoritmu prođemo 10 puta u sekundi (sve dok naše vrijeme procesuiranja ne premaši 1/10 sekunde).



Slika 12: Dijagram razine nadzora i upravljanja tijekom izvršavanja programa *wander.py*

Primjenom znanja o ROS arhitekturi i konceptima koje koristi, naučili smo kreirati čvorove, razmjenjivati poruke među njima, pristupiti podacima koje oni odašilju i izdavati poruke. U idućem poglavlju opisano je programiranje robota kojim se upravlja putem tipkovnice na računalu.

4.3 Robot upravljan putem tipkovnice

Za izradu ovog robota korištena je Linux distribucija operacijskog sustava; Ubuntu 16.04.LTS i ROS kinetic inačica robotskog operacijskog sustava. Važno je imati na umu da ne podržavaju sve inačice ROS sustava iste operacijske sustave na računalu. Konkretno, ROS kinetic inačica podržava Ubuntu 15.10, Ubuntu 16.04 i Debian 8 operacijski sustav. Upute kako instalirati ROS kinetic na računalo dostupne su na adresi: `https://wiki.ros.org/kinetic/Installation/Ubuntu`. Instalacijom ROS-a na računalo uključena je instalacija *catkin* sustava. Sustav *catkin*[10] je službeni sustav za izgradnju u ROS- i nasljednik je originalnog ROS sustava za izgradnju, *rosbuild*. Sustav za izgradnju je odgovoran za generiranje "ciljeva" iz sirovog izvornog koda koji može koristiti krajnji korisnik. Ti ciljevi mogu biti u obliku biblioteka, izvršnih programa, generiranih skripti ili bilo čega što nije statički kod. U ROS terminologiji, izvorni kod je organiziran u pakete, gdje se svaki paket obično sastoji od jednog ili više ciljeva kada je izgrađen. Sustav *catkin* kombinira makronaredbe CMake i Python skripte kako bi pružio neke funkcije CMakeovog normalnog tijeka rada. Dizajniran je kako bi omogućio bolju distribuciju paketa, bolju podršku i prenosivost.

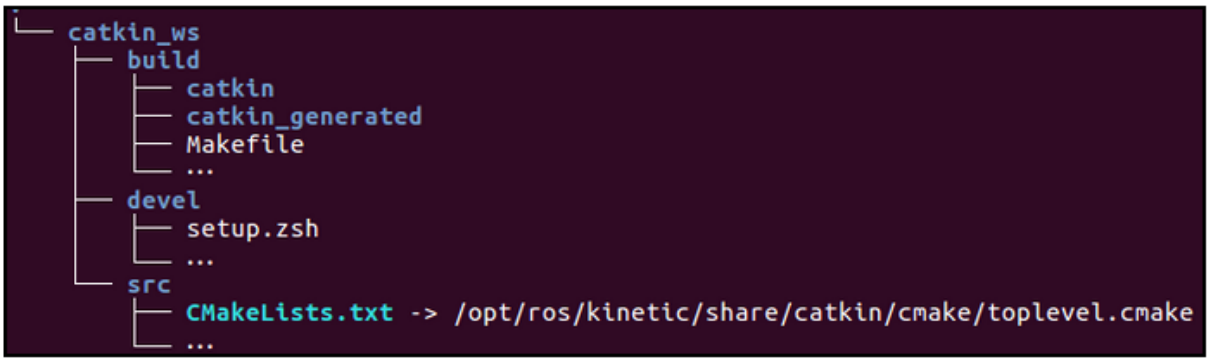
[10]Ime catkin na engleskom jeziku označava cvjetnu skupinu koja se nalazi na drvetu vrbe (*engl. willow*) - referenca na Willow Garage gdje je sustav izrađen.

4.3.1 Kreiranje radnog prostora

Prije nego započnemo sa programiranjem robota potrebno je izraditi radne mape, odnosno kreirati radni prostor u kojem ćemo programirati robota. Općenito, radni prostor je mapa koja sadrži pakete koji sadrže izvorne datoteke, a okolina ili radni prostor omogućuju način prikupljanja tih paketa. Korisno je kada istodobno želimo sastaviti različite pakete i to je dobar način centralizacije svih naših zbivanja. Radni prostor u kojem ćemo raditi smjestit ćemo u mapu *~/catkin_ws* upisivanjem sljedećih naredbi u terminal:

```
user@hostname$ mkdir -p ~/catkin_ws/src
user@hostname$ cd ~/catkin_ws/src
user@hostname$ catkin_init_workspace
```

Uobičajeni radni prostor prikazuje se na sljedećoj slici. Svaka mapa je drugačiji prostor s drugačijom ulogom:



Slika 13: Minimalan kostur mape radnog prostora

Izvorni prostor (*engl. source space*): U izvornom prostoru (*src* mapa) smještamo pakete, projekte, klonske pakete i tako dalje. Jedna od najvažnijih datoteka u ovom prostoru je *CMakeLists.txt*. Mapa *src* ima tu datoteku jer se poziva po cmake naredbi kada konfiguriramo pakete u radnom prostoru. Ova datoteka je stvorena pomoću naredbe *catkin_init_workspace*. Općenito, *CMakeLists.txt* datoteka je odgovorna za pripremu i izvršavanje procesa izrade.

Prostor za izgradnju (*engl. build space*): Mapa *catkin* u prostoru za izgradnju čuva informacije o cacheu, konfiguraciji i ostale posredne datoteke za naše pakete i projekte.

Razvojni prostor (*engl. development* (*devel*) *space*): Devel mapa koristi se za zadržavanje kompiliranih programa. Devel mapa se koristi za testiranje programa bez instalacijskog koraka. Nakon testiranja programa možemo instalirati ili izvesti paket da bismo ga podijelili s drugim programerima.

4.3.2 Kreiranje paketa

Dvije su mogućnosti s obzirom na izgradnju paketa s *catkin* sustavom. Prva je mogućnost upotreba standardnog CMake tijeka rada. Njome je omogućeno kompiliranje jednog

paketa po pozivu, kao što je prikazano sljedećim naredbama:

```
user@hostname$ cmakepackageToBuild/
user@hostname$ make
```

Ako želimo kompilirati više paketa u jednom pozivu, možemo koristiti *catkin* naredbu *catkin_make*:

```
user@hostname$ cd ~/catkin_ws
user@hostname$ catkin_make
```

Obje naredbe grade izvršnu datoteku u mapi za izgradnju prostora konfiguriranoj u ROS-u.

```
user@hostname$ cd ~/catkin_ws/src
user@hostname$ catkin_create_pkg superbot rospy geometry_msgs sensor_msgs
```

Naredbu *catkin_create_pkg* koristimo za kreiranje paketa u mapi izvornog koda (*src*) postavljenog radnog prostora. Prvi argument superbot označava ime paketa, a sljedeća 3 argumenta označavaju da taj paket koristi Python *rospy* knjižnicu i klase poruka: *geometry_msgs* i *sensor_msgs*.

4.3.3 Izgradnja radnog prostora i postavljanje konfiguracije

Nakon kreiranja radnog prostora i paketa potrebno je izgraditi (*engl. build*) radni prostor i pakete koji se u njemu nalaze, u ovom slučaju riječ je o jednom paketu- superbot. Preostalo nam je dodati izgrađeni radni prostor u ROS okolinu. To ćemo učiniti tako da postavimo konfiguracijsku datoteku izvornog koda u razvojni prostor (*engl. devel space*) na sljedeći način:

```
user@hostname$ cd ~/catkin_ws
user@hostname$ source devel/setup.bash
```

Sada imamo postavljenu bazu za programiranje robota. U sljedećem poglavlju kreirat ćemo upravljački čvor za tipkovnicu. Konkretno, kreirat ćemo čvor *key_pub* koji će odašiljati poruke tipa String na tu temu naziva *keys*. Poruke koje odašilje čvor *key_pub* predstavljati će znakove koje korisnik programa pritišće na tipkovnici svog računala.

4.3.4 Upravljački program za tipkovnicu

Listing 3: key_publisher.py

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

# termios knjiznica nam omogucava pristup ulaznim podacima sa konzole/terminala

import sys, select, tty, termios
import rospy
from std_msgs.msg import String
```

```
if __name__ == '__main__':
#Deklariranje cvora izdavaca na temu naziva keys i poruke tipa String
    key_pub = rospy.Publisher('keys', String, queue_size=1)
# Kreiranje cvora pod nazivom keyboard_driver
    rospy.init_node("keyboard_driver")
# Postavljanje fiksne frekvencije od 100 Hz
    rate = rospy.Rate(100)

    # Spremanje atributa sa tipkovnice
    old_attr = termios.tcgetattr(sys.stdin)
    tty.setcbreak(sys.stdin.fileno())
    print "Izdajem unos znakova sa tipkovnice. \n \nZa upravljanje robotom
        koristite tipkovnicu na sljedeci nacin: \n \n znak 'w': vozi prema
        naprijed \n znak 'a': okreci se u lijevu stranu \n znak 'd': okreci se u
        desnu stranu \n znak 'x': vozi unatrag \n znak 's': zaustavi se \n
        ----------------------------------- \n Zelim vam ugodnu voznju! :) \n \n
        \n \n Unesite Ctrl + C za prekid... "

    while not rospy.is_shutdown():
        if select.select([sys.stdin], [], [], 0)[0] == [sys.stdin]:
            key_pub.publish(sys.stdin.read(1))
        rate.sleep()

    # Vracanje atributa sa tipkovnice
    termios.tcsetattr(sys.stdin, termios.TCSADRAIN, old_attr)
```

Ova Python skripta koristi *termios* knjižnicu kako bi dohvatila podatke o pritisnutim tipkama sa tipkovnice na računalu. Obzirom da Unix konzole funkcioniraju tako da spremaju čitavu liniju teksta prije nego korisnik pritisne tipku Enter, pa tek onda šalju poruku programu sve što je korisnik natipkao, potrebno je modificirati takav način ponašanja sljedećim linijama koda:

```
  old_attr = termios.tcgetattr (sys.stdin)
  tty.setcbreak (sys.stdin.fileno ())
```

Na ovaj način možemo neprestano dobivati *stdin* ulaz sa tipkovnice i provjeriti ako su neki znakovi unešeni putem nje. Kako bi osigurali brzo primanje unešenih znakova, odnosno dobivanje jedan po jedan znak sa tipkovnice, možemo koristiti *select*() metodu sa mjeračem vremena postavljenim na vrijednost 0. To znači da ćemo dobiveni znak sa tipkovnice dohvatiti odmah, a našu petlju ćemo završiti sa *rate.sleep*() metodom.

```
if select.select ([sys.stdin], [], [], 0)[0] == [sys.stdin]: #ako imas podatak i
    on je jednak ulasku sa tipkovnice objavi ga na temu keys putem cvora key\_pub
  key_pub.publish (sys.stdin.read (1))
rate.sleep ()
```

Konačno, trebamo vratiti konzolu u stadardni način rada prije nego program završi:

```
termios.tcsetattr (sys.stdin, termios.TCSADRAIN, oldattr)
```

Sada ćemo testirati radi li naš upravljački program za tipkovnicu tako što ćemo otvoriti tri

terminala. U prvi terminal unijet ćemo naredbu za pokretanje ROS Master čvora:

```
user@hostname roscore
```

U idućem terminalu pokrenut ćemo python skriptu *key_publisher.py*:

```
user@hostname ./key_publisher.py
```

Na kraju, u trećem terminalu ćemo pozvati ispis svih *std_msgs/String* poruka koje odašilje tema naziva keys na sljedeći način:

```
user@hostname rostopic echo keys
```

Primjetimo, ako kliknemo na drugi prozor u kojem smo pozvali izvršavanje python skripte *key_publisher.py* i pritisnemo neku od tipki na tipkovnici, u trećem terminalu ćemo vidjeti ispis onih tipki koje smo pritisnuli. Sada možemo isprogramirati ostale potrebne dijelove kako bi isprogramirali robota kojim se upravlja putem tipkovnice u Gazebo ROS simulatoru. U nastavku teksta ćemo navesti potrebne dijelove koje ćemo na kraju sastaviti u jednu Python skriptu pod nazivom *superbot.py*.

4.3.5 Mapiranje pritisnutih tipki na tipkovnici u željenu kretnju robota

Upotrijebit ćemo mapiranje tipki w, x, a, d, s sa tipkovnice kako bi upravljali robotom i izrazili željeno kretanje pritiskom navedenih tipki. Konkretno i redom, tipke w, x, a, d, s mapirat ćemo u iduće kretnje: prema naprijed, unatrag, u lijevo, u desno i zaustavljanje robota. Kreirati ćemo ROS čvor koji izdaje Twist poruku svaki puta kada primi *std_msgs/String* poruku. Za mapiranje znakova w, x, a, d, s sa tipkovnice i ciljanih brzina na robotu koji će omogućiti željeno kretanje koristit ćemo Python rječnik.

4.3.6 Specificiranje porasta linearne i kutne brzine pritiskom na pripadajuću tipku

ROS koristi SI jedinice 0, -1 ili +1, što znači da tražimo od našeg robota da se kreće prema naprijed i natrag brzinom od jedan metar u sekundi i okreće se brzinom jedan radijan u sekundi. Roboti koriste različite brzine u različitim primjenama: za robotski automobil, jedan metar u sekundi je vrlo sporo kretanje; međutim, za mali mobilni robot koji se kreće hodnikom, jedan metar u sekundi je zapravo vrlo brzo kretanje. Pomoću ROS parametara možemo slati parametre programu za određivanje linearnih i kutnih brzina. ROS Master, često zvan *roscore*, uključuje poslužitelja parametara i omogućuje svim ROS čvorovima izdavanje parametara. Poslužitelj parametara je generički i oblika je ključ/vrijednost. U ROS-u termin privatni parametar označava parametar kojem sami zadajemo ime. Naime, on je i dalje javno dostupan. Pojam privatan jednostavno znači da se njegovo puno ime formira dodavanjem izabranog naziva u naziv čvora. To osigurava da ne može doći do sukoba naziva, jer su nazivi čvorova uvijek jedinstveni. Naprimjer, ako se čvor naziva *keys_to_twist* možemo imenovati privatne parametre *keys_to_twist linear_scale* i *keys_to_twist angular_scale*. Za postavljanje privatnih parametara u naredbenom retku u vrijeme pokretanja čvora, pišemo naziv parametra podcrtom i postavljamo njegovu vrijednost pomoću := sintakse. Primjer: *_linear_scale:= 0.5*. Uočimo, kao i svi objekti s masom, roboti ne mogu započeti kretanje i zaustaviti se istog trenutka, odnosno naglo. Fizika diktira da roboti postupno ubrzavaju tijekom vremena. Kada motori na kotačima robota pokušavaju naglo dostići velike brzine ili se naglo zaustaviti dok imaju veliku brzinu obično se dogadi nešto loše; kao što su klizanje, proklizavanje, podrhtavanje ili čak i mehanički prekid pogonjenja robota. Da bismo izbjegli ove probleme, trebamo omogućiti postepeno

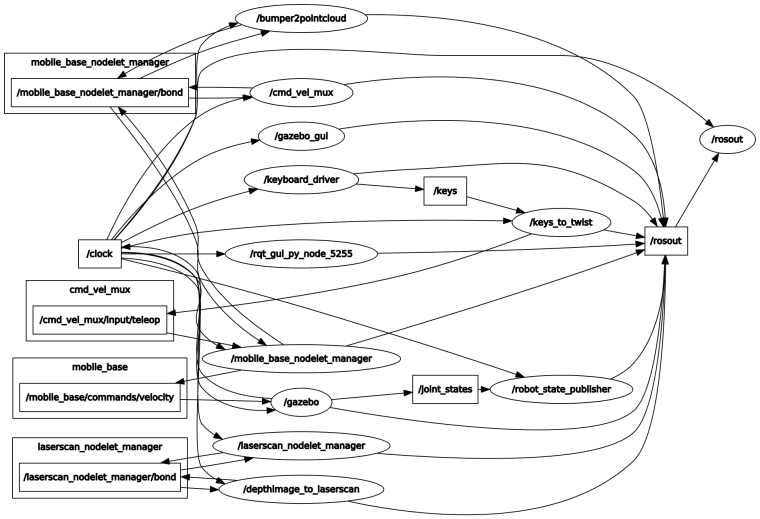
ubrzavanje i postepeno smanjenje brzine na robotu. Tako smo sigurni da naredbe koje šaljemo robotu je fizički moguće postići. Konačno, naš program kojim ćemo upravljati našim robotom izgleda ovako:

Listing 4: superbot.py

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

import rospy
import math
from std_msgs.msg import String
from geometry_msgs.msg import Twist

key_mapping = { 'w': [ 0, 1], 'x': [ 0, -1],
                'a': [ 1, 0], 'd': [-1, 0],
                's':[0,0]}

g_twist_pub = None
g_target_twist = None
g_last_twist = None
g_last_send_time = None
g_vel_scales = [0.1, 0.1] # pocetno na vrlo sporo
g_vel_ramps = [1, 1] # jedinica: metar po sekundi

# Ovo je kljucna funkcija. Obzirom na trenutno i prethodno vrijeme kada je
    pozvana funkcija, te obzirom na trenutnu i ciljanu vrijednost, izracunaj i
    azuriraj vrijednost (ili postigni cilj)
# Ova funkcija se koristi za obje brzine: linearnu i kutnu brzinu
def ramped_vel(v_prev, v_target, t_prev, t_now, ramp_rate): # izracunaj
    maksimalni korak brzine
    # korak = koliki je korak potrebno poduzeti, obzirom koliko je vremena proslo
        od zadnjeg poziva funkcije
    step = ramp_rate * (t_now - t_prev).to_sec()
    sign = 1.0 if (v_target > v_prev) else -1.0
    error = math.fabs(v_target - v_prev) # math.fabs() vraca apsolutnu vrijednost
        argumenta

    # do cilja mozemo stici unutar ovog vremenskog koraka, zavrsili smo
    if error < step:
        return v_target
    else:
        return v_prev + sign * step # poduzmi korak prema cilju

# Vrati Twist objekt ispunjen novim vrijednostima temeljenim na proteklo vrijeme
    i zeljenu stopu promjene
def ramped_twist(prev, target, t_prev, t_now, ramps):
    tw = Twist()
    tw.angular.z = ramped_vel(prev.angular.z, target.angular.z, t_prev, t_now,
        ramps[0])
    tw.linear.x = ramped_vel(prev.linear.x, target.linear.x, t_prev, t_now,
        ramps[1])
    return tw
```

```
# Posalji Twist za ovaj prolazak petljom
def send_twist():
    global g_last_twist_send_time, g_target_twist, g_last_twist, g_vel_scales,
        g_vel_ramps, g_twist_pub

    # Obzirom na dano novo vrijeme, kreiraj Twist za slanje uzimajuci u obzir
        stopu promjene
    t_now = rospy.Time.now()
    g_last_twist = ramped_twist(g_last_twist, g_target_twist,
        g_last_twist_send_time, t_now, g_vel_ramps)

    g_last_twist_send_time = t_now
    g_twist_pub.publish(g_last_twist)

def keys_cb(msg):
    global g_target_twist, g_last_twist, g_vel_scales
    if len(msg.data) == 0 or not key_mapping.has_key(msg.data[0]):
        return # nepoznat kljuc
    vels = key_mapping[msg.data[0]]
    # Ciljane vrijednosti su azurirane u globalnoj varijabli. One se koriste za
        racunanje aktualne Twist poruke koja se salje u svakoj petlji
    g_target_twist.angular.z = vels[0] * g_vel_scales[0]
    g_target_twist.linear.x = vels[1] * g_vel_scales[1]

# Koristimo ROS posluzitelj parametara kako bi dohvatili parametre. Radi se
    provjera i na komandnoj liniji i na posluzitelju parametara. Postoje 4
    parametra. Napomena: jedan od argumenata funkcije "default" zadaje se ako neki
    parametar nije isporucen
def fetch_param(name, default):
    if rospy.has_param(name):
        return rospy.get_param(name)
    else:
        print "parametar [%s] nije definiran. Primjenjujem default vrijednost na
            %.3f" % (name, default)
    return default

if __name__ == '__main__':
    rospy.init_node('keys_to_twist')
    g_last_twist_send_time = rospy.Time.now()
    g_twist_pub = rospy.Publisher('cmd_vel_mux/input/teleop', Twist, queue_size=1)

    rospy.Subscriber('keys', String, keys_cb)

    # Prikupi parametre
    g_target_twist = Twist() # inicijaliziraj na 0
    g_last_twist = Twist()
    g_vel_scales[0] = fetch_param('~angular_scale', 0.1)
    g_vel_scales[1] = fetch_param('~linear_scale', 0.1)
    g_vel_ramps[0] = fetch_param('~angular_accel', 1.0)
    g_vel_ramps[1] = fetch_param('~linear_accel', 1.0)
```

```
# Petlja se vrti 20 puta u sekundi. Napomena: pozivamo funkciju send_twist() jer
    se globalne vrijednosti (koje koristimo za izracunavanje potrebne stvarne
    linearne i kutne brzine) azuriraju kada "dohvate" kljucnu temu
    rate = rospy.Rate(20)
    while not rospy.is_shutdown():
        send_twist()
        rate.sleep()
```



Slika 14: Dijagram razine nadzora i upravljanja tijekom izvršavanja programa *superbot.py*

4.3.7 Pokretanje programa u simulacijskom programu Gazebo

Da bismo testirali naš program u Gazebo simulatoru potrebno je instalirati *gazebo_ros_pkgs* paket. Instalaciju izvršavamo upisom sljedeće naredbe u terminal:

```
  user@hostname sudo apt-get install ros-kinetic-gazebo-ros-pkgs
     ros-kinetic-gazebo-ros-control
```

U slučaju neuspješne instalacije paketa ovom naredbom, alternativne metode mogu se pronaći na sljedećoj adresi: `https://gazebosim.org/tutorials?tut=ros_installing`. Dodatno, potrebno je instalirati simulacijskog robota Turtlebota u Gazebo simulator:

```
  user@hostname$ sudo apt-get install ros-kinetic-turtlebot-gazebo
```

Pokrenimo Gazebo simulacijski svijet i Turtlebot robota u njemu upisom sljedeće naredbe u novi terminal:

```
user@hostname$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

Konačno, pokrenimo u novom terminalu čvor *key_publisher* koji smo izradili kako bi osluškivao znakove sa tipkovnice i slao ih modificirane kao brzine motorima kotača.

```
  user@hostname$ ./key_publisher.py
```

Proslijedimo parametre za izrađeni program *superbot.py* poslužitelju parametara u novom terminalu:

```
user@hostname$ ./superbot.py _linear_scale:=0.5 _angular_scale:=0.5
    _linear_accel:=1.0 _angular_accel:=1.0
```

Na kraju, pokrenimo naš program u novom terminalu:

```
user@hostname$ ./superbot.py
```

Za upravljanje simulacijskim Turtebot robotom u Gazebo simulatoru potrebno je otvoriti terminal u kojem je pokrenut naš čvor za osluškivanje znakova sa tipkovnice i koristiti tipkovnicu na definirani način.

5 Zaključak

Iako ROS ima mnoge korisne značajke, još uvijek postoje područja u kojima se ROS ne može koristiti ili se ne preporuča koristiti. Primjerice, u slučaju programiranja samohodnog automobila, možemo koristiti ROS za izradu prototipa, ali programeri ne preporučuju ROS za programiranje stvarnog proizvoda. Naime, zbog različitih problema, poput sigurnosti, obrade u stvarnom vremenu i sl., ROS nije dobar izbor. U korporativnoj robotici istraživačkih centara i na sveučilištima, ROS je idealan izbor za prototipove. Projekt pod nazivom ROS 2.0 razvija puno bolju verziju postojećeg ROS-a u pogledu sigurnosti i obrade u stvarnom vremenu te može postati dobar izbor za robotske proizvode u budućnosti. Programiranje robota prije ROS-a odvijalo se tako da svaki programer izradi softver za vlastiti robot, koji se u većini slučajeva nije mogao upotrijebiti za bilo kojeg drugog robota. Također, većina koda nije aktivno održavana pa nije bilo podrške za softver. Programeri su trebali implementirati standardne algoritme na svoje, pa je bilo potrebno više vremena za programirnje prototipa robota. Nakon projekta ROS, stvari su se promijenile. Sada postoji zajednička platforma za razvoj aplikacija u robotici. Besplatan i otvoren izvor za komercijalne i istraživačke svrhe. Ukratko, projekt ROS promijenio je način i pristup robotskom programiranju.

Životopis

Martina Šarić rođena je 29.08.1991 godine u Vinkovcima. Pohađala je osnovnu školu Zrinskih u Nuštru, te opću gimnaziju Matije Antuna Reljkovića u Vinkovcima. Nakon završene srednje škole upisuje sveučilišni preddiplomski studij matematike na Odjelu za matematiku sveučilišta J. J. Strossmayer. Završava predidplomski studij matematike u Osijeku te stječe zvanje prvostupnika matematike. Stečenom diplomom upisuje sveučilišni diplomski studij matematike i računarstva na istoimenom fakultetu u Osijeku. Na prvoj godini diplomskog studija izvršava praksu u tvrtki Gideon Brothers u trajanju od 8 tjedana na mjestu front end developera. U svibnju 2018.- e godine sudjeluje na Osijek Mini Maker Fair sajmu i predstavlja ROS Turtlebot 3 robota ispred Odjela za matematiku.

Literatura

[1] Lentin Joseph - Mastering ROS for Robotics Programming (2015, Packt Publishing)

[2] Lentin Joseph - Learning Robotics Using Python Design, simulate, program, and prototype an interactive autonomous mobile robot from scratch with the help of Python, ROS, and Open-CV! (2015, Packt Publishing)

[3] Morgan Quigley, Brian Gerkey, William D. Smart - Programming Robots with ROS (2015, O'Reilly Media)

[4] https://hub.packtpub.com/ros-architecture-and-concepts/, studeni. 2018