РОБОТОТЕХНІЧНІ ОПЕРАЦІЙНІ СИСТЕМИ

Лахно В.А., Блозва А.І.
Касаткін Д.Ю., Матус Ю.В.

РОБОТОТЕХНІЧНІ ОПЕРАЦІЙНІ СИСТЕМИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Київ
2021

ІКТ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Кафедра комп'ютерних систем, мереж і кібербезпеки

Лахно В.А., Касаткін Д.Ю., Блозва А.І.

РОБОТОТЕХНІЧНІ ОПЕРАЦІЙНІ СИСТЕМИ

(навчальний посібник для самостійної роботи студентів ОС Магістр з дисципліни «Робототехнічні операійні системи»)

КОМПРІНТ

КИЇВ - 2021

УДК 004.3(072)
ББК 39.97
К 63

Копіювання, сканування, запис на електронні носії і тому подібне, книжки в цілому, або будь-якої її частини заборонено

Рекомендовано до друку Вченою радою Національного університету біоресурсів і природокористування України
(протокол №3 від 27.10.2021р.)

Рецензенти:

Криворучко О.В. – доктор технічних наук, професор, завідувач кафедри інженерії програмного забезпечення та кібербезпеки Київського національного торговельно-економічного університету;

Казмірчук С.В. – доктор технічних наук, професор, завідувач кафедри комп'ютеризованих систем захисту інформації Національного авіаційного університету;

Болбот І.М. – доктор технічних наук, доцент, доцент **кафедри автоматики та робототехнічних ім. академіка І.І. Мартиненка** Національного університету біоресурсів і природокористування України.

Лахно В.А., Блозва А.І. Касаткін Д.Ю., Матус Ю.В.
К 63 Комп'ютерна схемотехніка та логіка [навчальний посібник] / В.А. Лахно, А.І. Блозва Д.Ю. Касаткін, Ю.В. Матус // - К.: НУБіП України, 2021.- 672с.

Навчальний посібник «Робототехнічні операційні системи» призначений для здійснення самостійної теоретичної підготовки студентів освітнього ступеня «Магістр» закладів вищої освіти за спеціальностями 123 «Комп'ютерна інженерія» і 125 «Кібербезпека» в галузі розробки, структурної і алгоритмічної організації сучасних арифметичних пристроїв комп'ютерних систем. У навчальному посібнику розглянуто методи розробки сучасних операційних систем для роботів (ROS). У посібнику розкрито підходи у роботі ROS, роботи фреймоворків, їх роботи та написання: Розглянуто програмні бібліотеки фреймворків ROS; описано підхід до самостійного написання програм на фреймворку ROS; Розписано підхід до знаходження і виправлення помилок коду ROS; у посібнику описано принципи до моделювання роботи у фреймворку ROS.

© Лахно В.А., Блозва А.І., Касаткін Д.Ю., Матус Ю.В. 2021

© НУБіП України, 2021

ПЕРЕДМОВА

Robot Operating System (ROS)

Robot Operating System (ROS) - це гнучка платформа (фреймворк) для розробки програмного забезпечення роботів. Це набір різноманітних інструментів, бібліотек і певних правил, метою яких є спрощення завдань розробки ПО роботів.

Створення дійсно надійного, універсального програмного забезпечення для роботів надзвичайно складне завдання. З точки зору робота, проблеми, які здаються тривіальними для людей, часто вимагають дуже складних технічних рішень. Часто розробка такого рішення не під силу одній людині.

ROS була створена, щоб стимулювати спільну розробку програмного забезпечення робототехніки. Кожна окрема команда може працювати над однією конкретною задачею, але використання єдиної платформи, дозволяє всьому співтовариству отримати і використовувати результат роботи цієї команди для своїх проектів.

Платформи (фреймворки) в робототехніці

Останнім часом, в області робототехніки особлива увага приділяється платформ. Поняття платформа зазвичай резделяют на програмну платформу і апаратну платформу. Програмна платформа для роботів включає в себе набір інструментів, які використовуються для розробки ПО роботів. Можна виділити типові, завдання програмної платформи: робота з низькорівневими пристроями, апаратна абстракція і комунікація, навігація, розпізнавання образів,

управління і установка пакетів і залежностей, підключення бібліотек, інструменти для налагодження і розробки.

Апаратні платформи, включають в себе готові дослідні та освітні пристрої (TurtleBot, TurtleBro). А також готові промислові системи.

Важливо відзначити, що апаратні платформи сумісні з програмними платформами, що дозволяє розробляти прикладні програми не маючи досвіду роботи з обладнанням і не витрачаючи час на його розробку. Сумісність інтерфейсів і методів взаємодії з обладнанням, дозволило величезній кількості розробників ПЗ внести свій вклад в розвиток робототехніки.

Уніфіковані інтерфейси і методи роботи з пристроями дозволяють накопичувати і обмінюватися готовими рішеннями всьому співтовариству зацікавлених людей в робототехніці.

Найбільш активні платформи

- MSRDS10 Microsoft Robotics Developer Studio, Microsoft - U.S.
- ERSP11 Evolution Robotics Software Platform, Evolution Robotics - Europe
- ROS Robot Operating System, Open Robotics12 - U.S.
- OpenRTM National Institute of Adv. Industrial Science and Technology (AIST) - Japan
- OROCOS Europe

- OPRoS ETRI, KIST, KITECH, Kangwon National University - South Korea

Чому варто почати з ROS

В даний момент для цілей вивчення і занурення в робототехніку можна однозначно рекомендувати ROS. Важливими критеріями на етапі занурення в робототехніку, є: активність спільноти, наявність різних бібліотек, розширюваність і простота використання. За цими критеріями в даний момент рівних ROS немає.

Слід особливо відзначити, що підприємницькі кола ROS надзвичайно активно. Коли ви стикаєтеся з проблемою, знайти рішення і отримати допомогу ставати простіше, не тільки від розробника ROS (компанії Open Robotics), але і від інших ентузіастів і професіоналів.

Що дає готова платформа

Повторне використання програмних модулів. Розроблений програмний модуль, легко запускається і переіспользуется в будь-якому іншому додатку. Питання установки залежностей та інших бібліотек добре опрацьований і автоматизований.

Готовий протокол комунікації Основна проблема комплексних робототехнічних систем, це рішення задач комунікації в рамках однієї програми. Для вирішення цих завдань ROS містить всі необхідні утиліти. Будь-програмних модуль може бути представлений як окремий процес, який взаємодіє з іншими процесами з мережевого

протоколу. Такий підхід дозволяє створювати незалежні і прості в повторному використанні програмні модулі, які можливо запустити / зупинити / модифікувати на будь-якому пристрої.

Розвиненість засобів розробки і налагодження ROS надає готові інструменти для налагодження, інструмент 2D-візуалізації (rqt), і інструмент 3D-візуалізації (RViz), інстумент 3D симуляції (Gazebo).

Активне і відкрите співтовариство Товариства розробників робототехніки з академічного світу і промисловості, були відносно закритими до останнього часу. Але зараз ми бачимо активне, і головне відкрите співробітництво всіх учасників. У центрі цієї зміни - програмна платформа з відкритим вихідним кодом. У разі ROS існує більше 5000 пакетів, які були розроблені і викладені в загальний доступ. Опис цих пакетів, інструкцій та іншої корисної інформації - перевищує 18 000 Wiki сторінок.

Власна екосистема Навколо ROS сформована власна екосистема (по аналогії з платформами Android і Apple). У ній існують розробники апаратних платформ, розробники програмних модулів, ентузіасти і компанії виробники промислового устаткування, єдине місце поширення та зберігання готових модулів, тисячі станиць документації. Всі учасники взаємодіють і робота

Розділ 1. Робот, програми платформи

1.1. Компоненти платформи

Рис. 1 ПК і смартфон

"Що спільного у цих двох товарних груп?"

ПК (персональний комп'ютер) та ПТ (персональний телефон) можна класифікувати як ІТ-продукти. Як випливає з їх назв, це особисті товари, якими володіє майже кожен. Як показано на малюнку 1-2, якщо розібрати загальні риси цих продуктів, ми можемо побачити, що вони складаються з апаратного модуля, що дозволяє інтегрувати з різним обладнанням та операційною системою, яка управляє цим обладнанням. У середовищі розробки програмного забезпечення на основі абстракції, що надається операційною системою, є додатки, що надають різні послуги, і численні користувачі, які використовують ці групи продуктів.

В рамках ІТ-галузі апаратне забезпечення, операційна система, додаток та користувач називаються чотирма основними

екосистемними компонентами платформи, як показано на малюнку 1-2. Коли всі ці компоненти існують і коли існує невидимий розподіл та

Рис. 2 Чотири основні компоненти екосистем та повторення історії, які можна побачити для ПК, PP та PR

співпраця роботи між цими компонентами, кажуть, що платформа може успішно стати популярною та персоналізованою.

Згадані раніше ПК та PP не мали всіх чотирьох цих компонентів з самого початку. На початку у них було лише вбудоване програмне забезпечення для управління певним апаратним пристроєм, використовуючи спеціальне програмне забезпечення, розроблене однією компанією, і могли використовувати лише послуги, що надаються виробником. Якщо це поняття важко зрозуміти, давайте використаємо функціональні телефони як приклад. Особливі телефони випускалися незліченними виробниками до появи iPhone від

Apple. Можна сказати, що загальним фактором, що дозволив успіх цих ПК або PP, є поява операційних систем (Windows, Linux, Android, iOS тощо). Поява операційних систем уніфікувала апаратне та програмне забезпечення, що призвело до модульності обладнання. Масове виробництво зменшило вартість, спеціалізована розробка принесла високі показники,

Крім того, інженери здатні розробляти прикладні програми в середовищі розробки, яке надає операційна система, навіть не досконало розуміючи апаратне забезпечення, а в області смартфонів була представлена нова група робочих місць під назвою Розробники програм, яка не існувала навіть 10 років тому. Таким чином, модульність апаратного забезпечення прогресує навколо операційних систем, а прикладні програми, засновані на апаратній абстракції, надані операційною системою, відокремлюються. Тому послуги, які бажали отримати користувачі, були створені та стали популярним продуктом або платформою. Що стосується PR (Personal Robot), який привертає увагу разом із ПК та PP, наскільки далеко просунулась репрезентативна платформа робочого робота? Як кажуть, історія повторюється, чи PR увійде в наше життя, як це відбувалося раніше з ПК та PP? Ми розглянемо це в наступному розділі.

1.2. Програмна платформа робота

Останнім часом у галузі робототехніки платформи привертають увагу. Платформа поділяється на програмну платформу та апаратну платформу. Програмна платформа робота включає в себе інструменти, які використовуються для розробки програм-програм-роботів, таких як апаратна абстракція, низькорівневе управління пристроями, зондування, розпізнавання, SLAM (одночасна локалізація та відображення), навігація, маніпуляції та управління пакетами, бібліотеки, засоби налагодження та розробки . Апаратні платформи роботів не лише дослідницькі платформи, такі як мобільні роботи, безпілотники та гуманоїди, але й комерційні продукти, такі як Pepper SoftBank, Jibo MIT Media Lab.

Примітно те, що ця апаратна абстракція відбувається спільно з вищезазначеними програмними платформами, що дає змогу розробляти прикладні програми з використанням програмної платформи, навіть не маючи досвіду роботи з обладнанням. Це те саме, що ми можемо розробляти мобільні програми, не знаючи апаратного складу або технічних характеристик останнього смартфона. Крім того, на відміну від попереднього робочого процесу, як розробники роботів виконували все, починаючи від проектування апаратного забезпечення і закінчуючи розробкою програмного забезпечення, тепер більше розробників програмного забезпечення, що не є роботами, можуть брати участь у розробці програм для

роботів. Іншими словами, програмні платформи дозволили багатьом людям зробити свій внесок у розробку роботів, а робоче обладнання розробляється відповідно до інтерфейсу, наданого програмними платформами.

Серед цих програмних платформ основними є Robot Operating System (ROS)**1**, Японське відкрите робототехнічне проміжне програмне забезпечення (OpenRTM)**2**, Європейський контроль реального часу OROCOS**3**, Корейські ОПРОС **4**і т. д. Хоча їх імена різні, основна причина появи робочих програмних платформ полягає в тому, що існує занадто багато різних видів програмного забезпечення для роботів, і їх ускладнення викликають багато проблем. Тому дослідники роботів з усього світу співпрацюють, щоб знайти рішення. Найпопулярнішою програмною платформою для роботів є ROS, операційна система роботів, яка буде висвітлена в цій книзі.

Наприклад, при реалізації функції, яка допомагає роботові розпізнати навколишню ситуацію, різноманітність обладнання та той факт, що воно безпосередньо застосовується в реальному житті, може бути тягарем. Деякі завдання можна вважати простими для людей, але дослідникам лабораторії або компанії коледжу занадто складно мати справу з роботами, щоб виконувати багато функцій, таких як зондування, розпізнавання, картографування та планування руху. Однак інша історія була б, якби професіонали з усього світу ділились

своїм спеціалізованим програмним забезпеченням для використання іншими. Наприклад, компанія-робототехніка Robotbase **5**, який привернув увагу до соціального фінансування KickStarter та CES2015, нещодавно розробив Robotbase Personal Robot та успішно запустив його через соціальне фінансування. У випадку з Robotbase вони зосередилися на своїй базовій технології - розпізнаванні обличчя та розпізнаванні об'єктів, а для свого мобільного робота вони використали базу мобільних роботів від Yujin Robot **6** який підтримує АФК, для виконавчого механізму вони використовували ROBOTIS Dynamixel**7**, а для розпізнавання перешкод, навігації, приводу автомобіля тощо вони використовували загальнодоступний пакет ROS. Інший приклад можна знайти в Промисловому консорціумі ROS(ROS-I) **8**. Багато компаній, що ведуть галузь промислових роботів, беруть участь у цьому консорціумі і вирішують деякі нові та складні проблеми з галузі промислових роботів по одному, такі як автоматизація, зондування та спільний робот. Доведено, що використання загальної платформи, особливо програмної, сприяє співпраці для вирішення проблем, які раніше було важко вирішити, та підвищення ефективності.

1.3. Потреба в програмній платформі робота

"Чому ми повинні використовувати робочу програмну платформу?"

Чому ми повинні вивчати ROS, що є новою концепцією робочої програмної платформи? Це поширене запитання на офлайн-семінарах ROS. Коротка відповідь полягає в тому, що це може скоротити час розробки. Часто люди кажуть, що не хочуть витрачати свій час на вивчення нової концепції, а скоріше дотримуватимуться своїх сучасних методів, щоб уникнути зміни вже побудованої системи чи існуючих програм. Однак ROS не вимагає розробки існуючої системи та програм заново, але досить легко може перетворити не-ROS-систему на ROS-систему, просто вставивши кілька стандартизованих кодів. Крім того, ROS надає різноманітні інструменти та програмне забезпечення, які широко використовуються, і це дозволяє користувачам зосередитись на функціях, які їх цікавлять або хотіли б внести свій внесок, що в кінцевому рахунку скорочує час розробки та обслуговування.

По-перше, це повторне використання програми. Користувач може зосередитися на функції, яку користувач хотів би розробити, і завантажити відповідний пакет для решти функцій. У той же час вони можуть ділитися програмою, яку вони розробили, щоб інші могли використовувати її повторно. Як приклад, сказано, що для НАСА слід керувати своїм роботом Robonaut2**9**використовувані на Міжнародній космічній станції, вони не тільки використовували програми, розроблені власноруч, але також використовували ROS, що

забезпечує різні драйвери для мультиплатформ, та OROCOS, який підтримує управління в режимі реального часу, відновлення зв'язку та надійність зв'язку, для досягнення їхня місія в космічному просторі. Наведена вище Robotbase - ще один приклад ретельно реалізованих багаторазових програм.

По-друге, ROS - це програма, заснована на спілкуванні. Часто для того, щоб надати послугу, такі програми, як апаратні драйвери для датчиків і приводів, та такі функції, як зондування, розпізнавання та експлуатація, розробляються в одному кадрі. Однак, щоб досягти можливості багаторазового використання програмного забезпечення-робота, кожна програма та функція розділена на менші частини залежно від її функцій. Це називається компонентизацією або модуляризацією відповідно до платформи. Дані повинні обмінюватися між вузлами (процес, який виконує обчислення в ROS), які розділені на одиниці мінімальних функцій, а платформи мають всю необхідну інформацію для обміну даними між вузлами. Мережеве програмування, що є дуже корисним у віддаленому керуванні, стає можливим, коли зв'язок між вузлом базується на мережі, так що вузли не обмежуються апаратним забезпеченням. Концепція мережевих мінімальних функцій застосовується також до Інтернету речей (IOT), тому ROS може замінити платформи IoT. Це надзвичайно корисно для пошуку помилок, оскільки програми, розділені на мінімальні функції, можна налагоджувати окремо.

Третє - підтримка інструментів розробки. ROS надає інструменти налагодження, 2D інструмент візуалізації (rqt, rqt - це програмний фреймворк ROS, який реалізує різні інструменти графічного інтерфейсу у формі додатків) та інструмент 3D-візуалізації (RViz), який можна використовувати без розробки необхідних інструментів для розробки роботів. Наприклад, є багато випадків, коли модель робота повинна бути візуалізована під час розробки робота. Просто узгодження заздалегідь визначеного формату повідомлення дозволяє користувачам не тільки безпосередньо перевірити модель робота, але й виконати моделювання за допомогою наданого 3D-симулятора (Gazebo). Інструмент також може отримувати 3D-інформацію про відстань від нещодавно виділених Intel RealSense або Microsoft Kinect, легко перетворювати їх у форму хмари точок та відображати на інструменті візуалізації. Крім цього, він також може записувати дані, отримані під час експериментів, і відтворювати їх, коли це необхідно для відтворення точного середовища експерименту. Як показано вище,

Четверте - активна спільнота. Академічний світ і промисловість роботів, які до цього часу були відносно закритими, змінюються у напрямку наголошення на співпраці в результаті цих раніше згаданих функцій. Незалежно від різниці в окремих цілях, фактично відбувається співпраця через ці програмні платформи. В центрі цих змін є спільнота програмної платформи з відкритим кодом.

У випадку ROS існує понад 5000 пакетів, які були добровільно розроблені та розподілені станом на 2017 рік, а сторінки Wiki, які пояснюють їх пакети, перевищують 18 000 сторінок завдяки внеску окремих користувачів. Більше того, ще одна важлива частина спільноти, яка є питаннями та відповідями, перевищила 36 000 постів, створюючи спільноту, яка зростає. Спільнота виходить за рамки обговорення інструкцій, і знайти необхідні компоненти програмного забезпечення для робототехніки та створити правила щодо них. Крім того, це прогресує до стану, коли користувачі об'єднуються і думають, що програмне забезпечення для роботів повинно передбачати для просування робототехніки та співпрацювати, щоб заповнити відсутні частини в головоломці.

По-п'яте, це формування екосистеми. Зазначено, що раніше згадана революція платформи для смартфонів сталася тому, що існувала екосистема, яку створили такі програмні платформи, як Android або iOS. Цей тип прогресування так само триває для роботизованого поля. Спочатку всі види апаратних технологій були переповнені, але не було операційної системи, яка б їх інтегрувала. Розроблено різні програмні платформи, і найповажніша платформа серед них, ROS, зараз формує свою екосистему. Це створює екосистему, якою можуть бути задоволені всі - розробники обладнання з роботизованої галузі, такі як роботи з роботами та датчиками, операційна команда розробників ROS, розробники

прикладного програмного забезпечення та користувачі. Хоча початок ще може бути маргінальним,

1.4. Майбутнє, яке принесе робоча програмна платформа

Поле робота рухається по тій же доріжці, що і попередній показаний приклад поля смартфона. Хоча в порівнянні з операційною системою для смартфонів ще багато чого слід розробити, програмна платформа-робот перебуває в жвавій стадії, коли кожен може стати лідером галузі. Наступні платформи, перелічені нижче, є найбільш активними платформами роботів.

- **MSRDS 10** Студія розробників Microsoft Robotics, Microsoft - США
- **ERSP 11** Програмна платформа Evolution Robotics, Evolution Robotics - Європа
- **АФК** Операційна система робота, Open Robotics**12** - НАС
- **OpenRTM** Національний інститут адв. Промислова наука та технології (AIST) - Японія
- **OROCOS** Європа
- **ОПРОСИ** ETRI, KIST, KITECH, Кангвонський національний університет - Південна Корея
- **NAOqi OS13** SoftBank та Aldebaran - Японія та Франція

Окрім них, є також Player, YARP, MARIE, URBI, CARMEN, Orca та MOOS.

Рис. 3 Різні робочі програмні платформи

Як ви бачите вище, з’являються різні робочі програмні платформи, але важко зробити висновок, яка з них краща. Причина полягає в тому, що кожна з них забезпечує унікальні та зручні функції, такі як розширення компонентів, функція зв'язку, візуалізація, симулятор, реальний час та багато іншого. Однак, подібно до сучасних операційних систем персональних комп’ютерів, робочі програмні платформи, які обирають користувачі, стануть більш популярними, тоді як інші зменшуються. Оскільки ми не розробляємо фактичну програмну платформу, ми зосередимося на наших навичках розробки прикладних програм, які можуть працювати на робочих програмних платформах загального призначення.

Багато разів мене запитують: “Яка найкраща платформа роботів?” і моя відповідь на це: “Давайте зараз же припинимо робити новий дитячий майданчик! Давайте мріяти бути чудовим гравцем на цьому майданчику, рухаючись вперед ".

Ми можемо порівняти це з випадком Android. Подібно до того, як ми не розробляли, ми не маємо повноважень контролювати екосистему Android, але вона стала домінуючою на ринках апаратного та програмного забезпечення та значним чином сприяє зростанню економіки. Ми, швидше за все, можемо стати чудовим гравцем на ринку програмного забезпечення роботів.

Тоді з якою з існуючих на сьогодні робочих програмних платформ нам було б добре ознайомитись? Моєю найкращою відповіддю буде ROS, який розробляється та підтримується Open Robotics. Не тільки завдяки високоактивному співтовариству, а й враховуючи різні бібліотеки, розширюваність та зручність розробки, іншої платформи, подібної ROS, не існує. Для вашої інформації Open Source Robotics Foundation (OSRF) змінив свою назву на Open Robotics у травні 2017 року.

Слід також зазначити, що глобальне співтовариство ROS є більш активним, ніж будь-яке інше співтовариство робототехнічної платформи. Легше знайти інформацію, коли ви стикаєтесь із проблемою під час її використання, оскільки ROS розробляється не лише Open Robotics, але академічними дослідниками, розробниками на місцях і навіть любителями, і всі ці люди активно використовують спільноту, коли стикаються з питаннями, тому , роблячи його доступним для інших користувачів для пошуку цінної інформації. На додаток до цього, є не тільки спеціалісти-роботи, а й велика кількість

мережевих спеціалістів та людей у галузі інформатики та комп'ютерного зору, які розробляють ROS ще більш перспективну програмну платформу для роботів.

Використовуючи програмну платформу робота, навіть якщо робот складається з різного обладнання, доки готові основні функції, ви можете створити прикладну програму, не досконало розуміючи обладнання. Це майже так само, як ми можемо розробляти мобільні додатки, не знаючи апаратного складу або деталей останнього смартфона.

На відміну від минулих робочих процесів, коли розробникам роботів доводилося робити все, починаючи від проектування апаратного забезпечення і закінчуючи розробкою програмного забезпечення, тепер більше інженерів програмного забезпечення можуть брати участь у процесі розробки власне роботоданих. Іншими словами, програмна платформа дозволила багатьом інженерам ефективно сприяти розробці роботів, а технічні спеціалісти, наприклад, можуть зосередитися на розробці обладнання, яке підтримує інтерфейс, необхідний програмній платформі. Ця зміна дає можливість галузям роботів швидко прогресувати.

Розділ 2. Робот працює, система ROS

2.1. Вступ до ROS

ROS Wiki визначає ROS як вище. Іншими словами, ROS включає апаратний рівень абстракції, подібний до операційних систем. Однак, на відміну від звичайних операційних систем, він може бути використаний для численних комбінацій апаратної реалізації. Крім того, це робоча програмна платформа, яка забезпечує різні середовища розробки, спеціалізовані для розробки програм-програм для роботів.

2.2. Мета-операційна система

Операційні системи (ОС) для комп'ютерів загального призначення включають Windows (XP, 7, 8, 10), Linux (Linux Mint, Ubuntu, Fedora, Gentoo) та Mac (OS X Mavericks, Yosemite, El Capitan). Для смартфонів існують Android, iOS, Symbian, RiMO, Tizen тощо.

ROS - це нова операційна система для роботів?

ROS - це абревіатура для Robot Operating System, тому можна було б з упевненістю сказати, що це операційна система. Зокрема, ті, хто новачок у ROS, можуть подумати, що це подібна операційна система, як і вищезазначені операційні системи. Коли я вперше зіткнувся з ROS, я також подумав, що це нова операційна система для роботів.

Однак більш точним описом буде те, що ROS - це мета-операційна система[1]. Хоча мета-операційна система не є визначеним терміном у словнику, вона описує систему, яка виконує такі процеси, як планування, завантаження, моніторинг та обробка помилок, використовуючи рівень віртуалізації між програмами та розподіленими обчислювальними ресурсами.

Тому ROS - це не звичайна операційна система, така як Windows, Linux та Android, а мета-операційна система, яка працює на існуючій операційній системі. Для роботи ROS спочатку потрібно встановити операційну систему, таку як Ubuntu, яка є одним із дистрибутивів Linux. Після завершення інсталяції ROS на вершині Linux можна використовувати такі функції, що надаються звичайною операційною системою, такі як система управління процесами, файлова система, користувальницький інтерфейс та утиліта програми (компілятор, модель потоку). На додаток до основних функцій, що надаються Linux, ROS надає основні функції, необхідні для програм-роботів-прикладних програм, таких як бібліотеки, такі як передача / прийом даних серед різнорідного обладнання, планування та обробка помилок. Цей тип програмного забезпечення також називають проміжним програмним забезпеченням або програмним забезпеченням.

Як мета-операційна система, ROS розробляє, управляє та надає пакети програм для різних цілей, і вона сформувала екосистему, яка

розповсюджує пакети, розроблені користувачами. Як описано на малюнку 2-1, ROS є допоміжною системою управління роботом і датчиком з апаратною абстракцією та розробкою програми-робота на основі існуючих звичайних операційних систем.

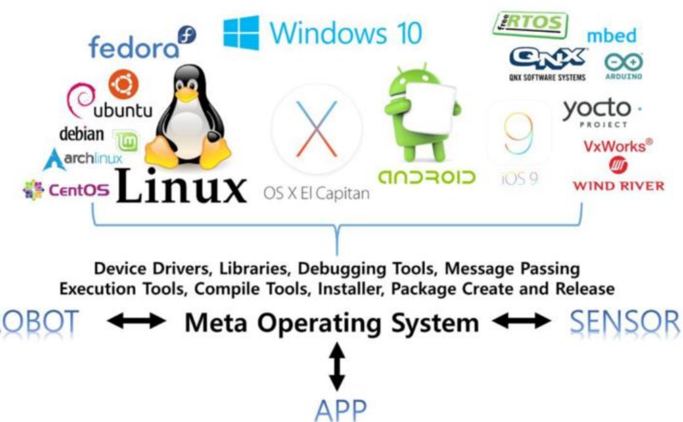


Рис. 3 ROS як мета-операційна система

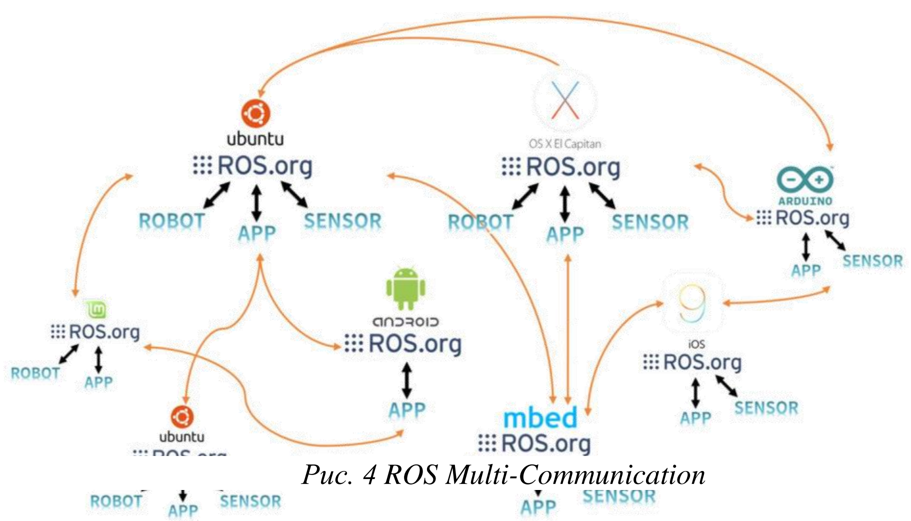


Рис. 4 ROS Multi-Communication

Як показано на Рис. 3, передача даних ROS підтримується не тільки однією операційною системою, але також безліччю операційних систем, обладнання та програм, що робить її надзвичайно придатною для розробки роботів, де поєднується різне обладнання. Це буде детально обговорено в наступному розділі.

2.3. Цілі АФК

Одне з найпоширеніших питань, яке я отримав протягом багатьох років на семінарах, пов'язаних з ROS, - це порівняння ROS з іншими програмними платформами роботів (OpenRTM, OPRoS, Player, YARP, Orocos, CARMEN, Orca, MOOS, Microsoft Robotics Studio). Просте порівняння з цими платформами може бути можливим, але порівняння не має сенсу, оскільки кожна з них має різні цілі. Як користувач ROS, я відчував, що метою ROS є "побудувати середовище розробки, яке дозволяє розробці робототехнічного програмного забезпечення співпрацювати на глобальному рівні!" Тобто ROS орієнтована на максимізацію повторного використання коду в робототехнічних дослідженнях і розробках, а не на орієнтацію на так звану робочу програмну платформу, проміжне програмне забезпечення та фреймворк. Для підтвердження цього ROS має такі характеристики.

- Розподілений процес: Він запрограмований у вигляді мінімальних одиниць виконуваного файлу процеси (вузли), і кожен процес працює самостійно і систематично обмінюється даними.
- Управління пакетами: Багато процесів, що мають ту саму мету, керуються як пакет, таким чином, щоб він був простим у використанні та розробці, а також зручним для спільного використання, модифікації та розповсюдження.
- Громадський сховище: Кожен пакет оприлюднюється улюбленим загальнодоступним сховищем розробника (наприклад, GitHub) і вказує їх ліцензію.
- API: При розробці програми, яка використовує ROS, ROS призначена просто викликати API і легко вставити його в використовуваний код. У вихідному коді, представленому в кожному розділі, ви побачите, що програмування ROS мало чим відрізняється від C ++ та Python.

Підтримка різних мов програмування: Програма ROS надає клієнтську бібліотеку**2** для підтримки різних мов програмування. Бібліотеку можна імпортувати мовами програмування, які популярні в галузі робототехніки, такими як Python, C ++ та Lisp, а також такими мовами, як JAVA, C #, Lua та Ruby. Іншими словами, ви можете розробити програму ROS, використовуючи бажану мову програмування.

Ці характеристики ROS дозволили користувачам створити середовище, де можна співпрацювати над розробкою програмного забезпечення для робототехніки на глобальному рівні. Повторне використання коду в робототехнічних дослідженнях і розробках стає все більш поширеним явищем, що є кінцевою метою ROS.

2.4. Компоненти АФК

Як показано на рис. 5, ROS складається з клієнтської бібліотеки для підтримки різних мов програмування, апаратного інтерфейсу для апаратного управління, зв'язку для передачі та прийому даних, Robotics Application Framework, що допомагає створювати різні Robotics Applications, Robotics Application, який сервісний додаток, заснований на Robotics Application Framework, інструментах моделювання, які можуть керувати роботом у віртуальному просторі, та засобах розробки програмного забезпечення.

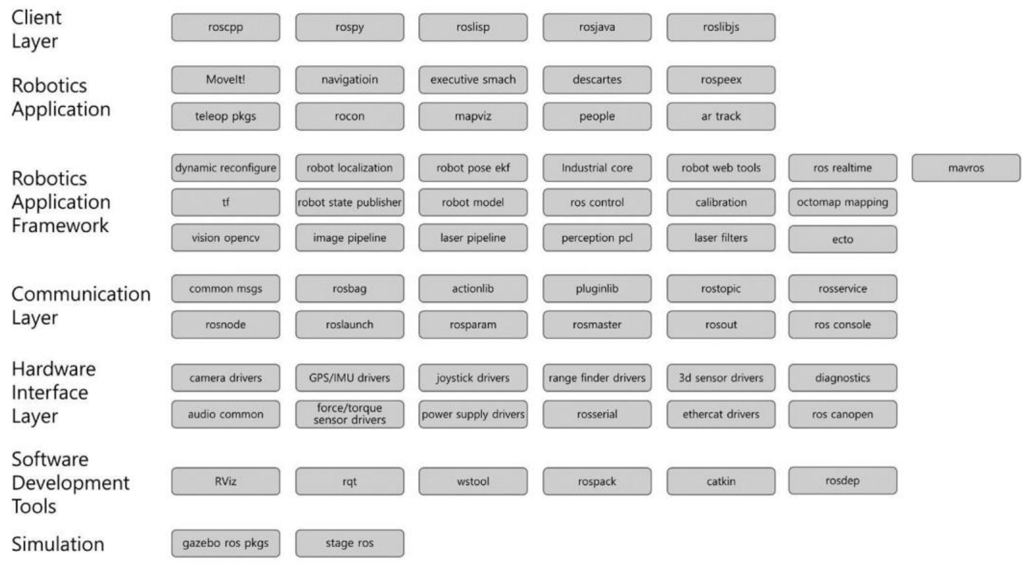


Рис. 5 Компоненти АФКЗ

2.5. Екосистема ROS

Термін `` екосистема " часто згадується на ринку смартфонів після появи різних операційних систем, таких як Android, iOS, Symbian, RiMO та Bada. Екосистема відноситься до структури, яка пов’язує виробників обладнання, компанії, що розробляють операційні системи, розробників додатків та кінцевих користувачів.

Наприклад, виробники смартфонів випускатимуть пристрої, що підтримують апаратні інтерфейси операційної системи, а компанії, що працюють над операційними системами, створюють загальну бібліотеку для управління пристроями різних виробників. Тому розробники програмного забезпечення можуть використовувати

численні пристрої, не розуміючи обладнання, для розробки додатків. Екосистема включає розподіл програми серед кінцевих користувачів.

Екосистема не була новою концепцією на ринку. На ринку персональних комп’ютерів також існували різні виробники обладнання, і це обладнання зв’язувалось головним чином операційною системою Microsoft Windows і Linux з відкритим кодом. Формування технологічної екосистеми видається таким же природним, як і природна екосистема.

Робототехніка також формує свою екосистему. Спочатку різні апаратні технології були переповнені, але не було операційної системи для їх інтеграції. З'явилося кілька програмних платформ, і ROS привернув достатньо уваги для побудови екосистеми. Незважаючи на те, що його ефект ще може бути незначним, розглядаючи зростаючу кількість користувачів, компаній, пов'язаних з роботами, та відповідних інструментів та бібліотек, ми можемо передбачити повністю функціональну екосистему найближчим часом.

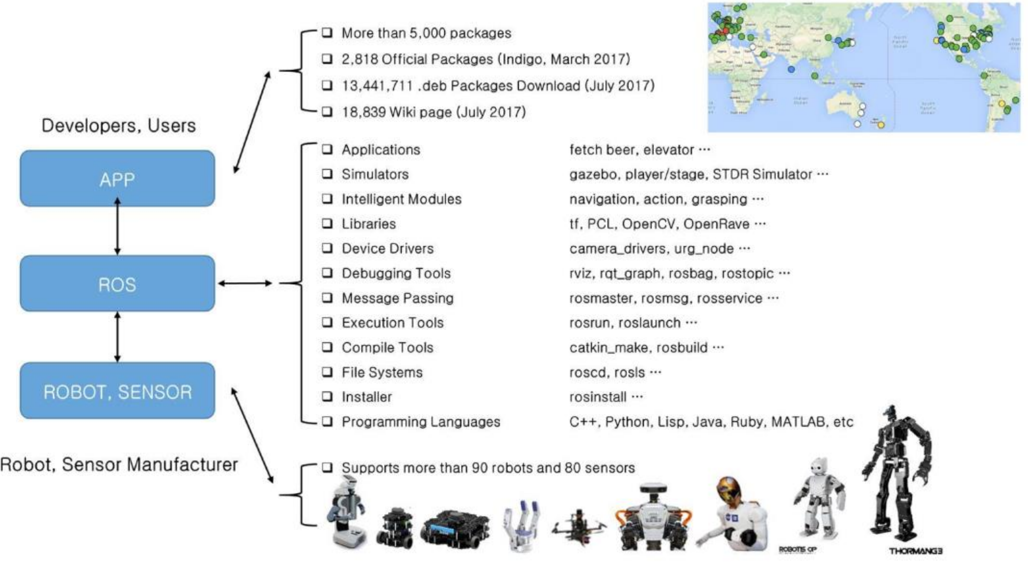


Рис. 6 Екосистема ROS

На рис. 6 показано стан ROS під час ROSCon у 2017 році, на основі офіційної статистики АФК та дані ROS Wiki 2017 року. Деякі можуть подумати, що це все ще на граничному рівні, але немає жодної іншої програмної платформи для роботів, яка б широко використовувалася в галузі робототехніки, наприклад ROS. Я дуже хочу побачити, як він буде зростати в майбутньому.

2.6. Історія ROS

Давайте глибше розглянемо ROS. У травні 2007 р. ROS було започатковано шляхом запозичення ранніх програм роботизованого програмного забезпечення з відкритим кодом, включаючи електростанцію, який розроблений доктором Морганом Квіглі

Стенфордською лабораторією штучного інтелекту на підтримку роботи Стенфордського A.I. Проект STAIR (STanford AI Robot).

У листопаді 2007 року американська компанія-робот Willow Garage стала наступником розробки ROS. Willow Garage - відома компанія в галузі персональних роботів та робочих роботів. Він також відомий розробкою та підтримкою бібліотеки точок хмари (PCL), яка широко використовується для 3D-пристроїв, таких як Kinect та бібліотека з відкритим кодом для обробки зображень OpenCV.

Willow Garage розпочав розробку ROS у листопаді 2007 року, а 22 січня 2010 року ROS 1.0 вийшов у світ. Офіційна версія, відома нам, вийшла 2 березня 2010 року під назвою ROS Box Turtle. Потім були випущені C Turtle, Diamondback та багато версій в алфавітному порядку, таких як Ubuntu та Android.

ROS базується на ліцензії BSD із 3 положеннями та ліцензія Apache 2.0, що дозволяє будь-кому змінювати, повторно використовувати та розповсюджувати. ROS також надав велику кількість найновішого програмного забезпечення та брав активну участь в освіті та науковій роботі, ставши відомим спочатку завдяки академічному товариству робототехніки. Зараз існують ROSDay та ROSCon конференції для розробників та користувачів, а також різні збори спільнот під назвою ROS Meetup. Крім того, пришвидшується також розвиток роботизованих платформ, які можуть застосовувати ROS. Деякі приклади є PR2, що означає Personal Robot та TurtleBot, і

було багато додатків впроваджено через ці платформи, роблячи ROS як домінуючу робочу програмну платформу.

Рис. 7 Логотип OSRF
(http://osrfoundation.org/)

Рис. 8 Відкрити логотип Robotics
(https://www.openrobotics.org/)

2.7. Версії ROS

У 2007 році Willow Garage вдався до дослідження робочого програмного забезпечення для роботів, яке розпочалося з Комутаційного заводу в лабораторії штучного інтелекту Стенфордського університету і продовжило розробку під назвою Robot Operating System (ROS). З шостою офіційною версією версії ROS Groovy Galapagos, Willow Garage намагався проникнути на ринок комерційних роботів у 2013 році, але в підсумку розділився на кілька стартапів і, зрештою, був переданий Фонду робототехніки з відкритим кодом (OSRF). З тих пір було випущено ще чотири версії, і з травня

2017 року OSRF змінив свою назву на Open Robotics і розробляє, працює та управляє ROS. Зовсім недавно, 23 травня 2017 року, була випущена 11-та версія ROS, ROS Lunar Loggerhead.

ROS-релізи та конференції

08 грудня 2017 р	Випуск ROS 2.0
21 вересня 2017 р	ROSCon2017 (Канада)
23 травня 2017 р	Випуск місячного лісоруба
16 травня 2017 р	Змінено назву з OSRF на Open Robotics
8 жовтня 2016 р	ROSCon2016 (Південна Корея)
23 травня 2016 р	Випуск Kinetic Kame
3 жовтня 2015 р	ROSCon2015 (Німеччина)
23 травня 2015 р	Випуск Джейд Черепахи
12 вересня 2014 р	Конференція ROSCon2014 (США)
22 липня 2014 р	Випуск Indigo Igloo
6 червня 2014 р	Конференція ROS Kong 2014 (Гонконг)
4 вересня 2013 р	Випуск Hydro Medusa
11 травня 2013 р	Конференція ROSCon2013 (Німеччина)

11 лютого 2013 р	Фонд робототехніки з відкритим кодом бере на себе розробку / управління
31 грудня 2012 р	Випуск Groovy Galapagos
19 травня 2012 р	Конференція ROSCon2012 (США)
23 квітня 2012 р	Випуск Fuerte
30 серпня 2011 р	Випуск Electric Emys
2 березня 2011 р	Випуск Diamondback
2 серпня 2010 р	Випуск C Turtle
2 березня 2010 р	Випуск Box Черепахи
22 січня 2010 р	Випуск ROS 1.0
1 листопада 2007 р	Willow Garage розпочинає розробку під назвою 'ROS'
1 травня 2007 р	проект електростанції, Morgan Quigley, Stanford AI LAB, Stanford University
2000 рік	Проект гравця / сцени, Брайан Геркі, Університет Південної Каліфорнії (USC)

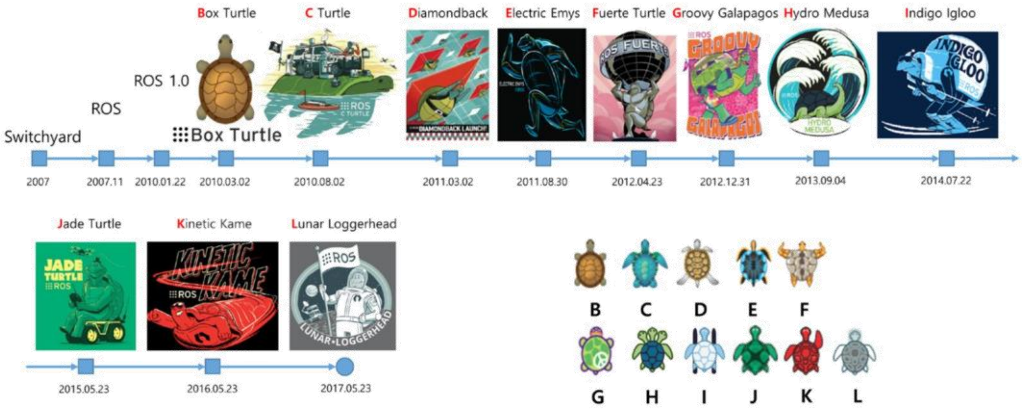


Рис. 8 Версії ROS (http://wiki.ros.org/)

На сьогоднішній день ROS випустив версії ROS 1.0, Box Turtle, C Turtle, Diamondback, Electric Emys, Fuerte Turtle, Groovy Galapagos, Hydro Medusa, Indigo Igloo, Jade Turtle, Kinetic Kame та Lunar Loggerhead.

Окрім версії 1.0, ROS називає їх версії для запуску в алфавітному порядку, як і Ubuntu та Android. Наприклад, версія Kinetic Kame є 11-ю версією, таким чином, починаючи з алфавіту K, і 10-ю офіційною версією випуску.

Крім того, є ще одне правило. Кожна версія має ілюстрацію у вигляді плаката та піктограми черепахи, як показано на малюнку 2-8. Ці піктограми черепах також використовуються в офіційному навчальному посібнику з моделювання, який називається 'turtlesim'. Символ черепахи для АФК походить від логотипу навчальної мови програмування з лабораторії штучного інтелекту MITу 1960-х. Понад 50 років тому в 1969 році був розроблений робот-черепаха з

використанням логотипу, який міг фактично пересуватися по підлозі та малювати зображення відповідно до інструкцій, наданих комп'ютером. На основі цього робота була розроблена програма 'turtlesim', а фактично робот згодом також називався TurtleBot.

Рис. 9 Піктограми черепахи для кожної версії ROS

Версія ROS оновлюється двічі на рік (квітень та жовтень), так само, як і період випуску ROS, що підтримує операційну систему Ubuntu. Однак у 2013 році користувачі почали пропонувати новий цикл версій через часті оновлення, і відгуки були враховані. Таким чином, починаючи з версії Hydro Medusa у 2013 році, нова версія випускалася раз на рік у травні, через місяць після випуску версії Ubuntu xx.04. Довідково: 23 травня - Всесвітній день черепахи, і ROS випускає свою нову версію цього символічного дня.

Період підтримки ROS для кожної версії різний, але, як правило, два роки підтримки доступні після її випуску. Кожні два роки ROS та Linux випускають довгострокову підтримку версія та ROS підтримується протягом наступних п’яти років. Наприклад, версія Kinetic Kame, яка підтримує Ubuntu 16.04 LTS, буде підтримуватися до квітня 2021 року. Інші версії, крім версій LTS, зазвичай призначені для незначних модернізацій та обслуговування, оскільки вони

підтримують останнє ядро Linux. Тому багато користувачів ROS використовувати версії LTS, які виходять раз на два роки. Вийшла остання версія ROS з 2014 року наведені на малюнку 10.

Distro	Release Date	Poster	Symbol	EOL Date
Lunar Loggerhead	2017.05.23			2019.05
Kinetic Kame **(Recommended)**	2016.05.23			2021.04 (Xenial EOL)
Jade Turtle	2015.05.23			2017.05
Indigo Igloo	2014.07.22			2019.04 (Trusty EOL)

Рис. 10 Останні версії ROS та дата закінчення підтримки

Оскільки ROS є мета-операційною системою, вам потрібно буде вибрати ОС для використання. ROS підтримує Debian, Ubuntu, Linux Mint, OS X, Fedora, Gentoo, openSUSE, Arch Linux та Windows, але найпопулярнішими операційними системами є Debian, Ubuntu та Linux Mint. З цієї причини я хотів би рекомендувати Debian, Ubuntu LTS або Linux Mint, які є найбільш часто використовуваною версією ROS.

Інформацію про перенесений пакет Ubuntu для кінетичної версії ROS можна знайти на відповідній інформаційній сторінці**22**. На цій сторінці ви можете побачити, чи завершена кінетична версія, чи перебуває в процесі міграції пакетів (джерело) для кожної версії Linux.

Назва випуску Linux може бути вам невідома. Як ви можете бачити з наступного списку версій Ubuntu, 14.04 Trusty - це версія T Ubuntu, яка випускається в алфавітному порядку. Ви зможете знайти стабільну версію зі списку. Остання версія ROS показує, що багато пакетів все ще тривають, але якщо це не критично для вашої програми, ви можете скористатися останньою версією або доступною на даний момент версією. Якщо пакет, який ви використовували, не перетворено на останню версію ROS, можливо, доведеться трохи почекати.

Я рекомендую наступну комбінацію до 2019 року, коли нові версії Linux та ROS LTS стануть стабільними.

Операційна система: Ubuntu 16.04 Xenial Xerus (LTS) або Linux Mint 18.x або Debian Jessie

АФК: ROS Kinetic Kame

Розділ 3. Конфігурація, розробка ROS, середовище

Хоча ROS підтримує різноманітні операційні системи, оскільки Ubuntu є найбільш широко використовуваним серед користувачів ROS, ця книга стосується лише Ubuntu та Linux Mint, сумісних з Ubuntu.

У цій книзі використано середовище розробки додатків ROS.

Апаратне забезпечення: Робочий стіл або ноутбук з використанням процесорів Intel або AMD

Операційна система: Ubuntu 16.04.x Xenial Xerus або Linux Mint 18.x

АФК: Кінетична Каме

Якщо на вашому комп'ютері встановлена інша версія Ubuntu, перевірте офіційний сайт, і якщо у вас операційна система OS X**1** або Windows**2** Ви можете перевірити відповідний Wiki**3**сторінку для методів встановлення. Якщо ви використовуєте одноплатний комп'ютер (SBC), який використовує ARM-процесор замість процесорів Intel або AMD, ми окремо не надаємо інструкцій щодо встановлення ROS, але якщо ви використовуєте Ubuntu або Linux Mint, ви можете слідувати нижче інструкції.

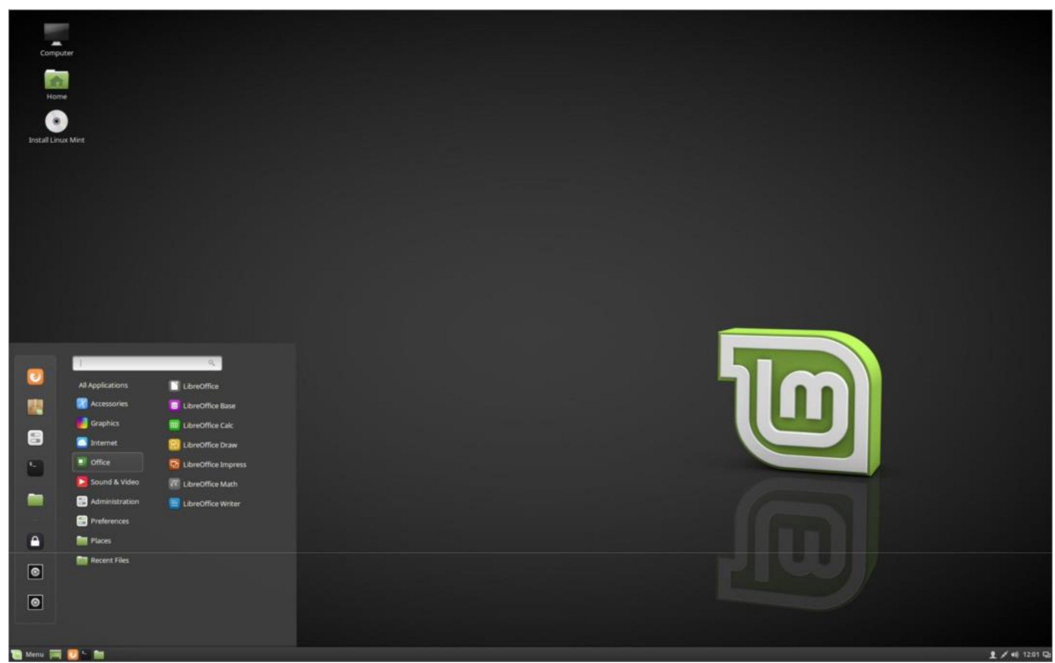


Рис. 11 Екран робочого столу Linux Mint

3.1. Встановлення ROS

Давайте встановимо ROS Kinetic. Ви зможете встановити ROS Kinetic без особливих труднощів, дотримуючись інструкцій нижче. Ви також можете скористатися сценарієм швидкого встановлення, наведеним у розділі 3.1.2, для ще більш простого встановлення.

Конфігурація NTP (Network Time Protocol)

Хоча він не входить до офіційного інсталяційного пакета ROS, ми можемо налаштувати NTP4для того, щоб зменшити різницю в часі ROS у зв'язку між кількома ПК. Спосіб встановлення полягає в тому,

щоб спочатку встановити 'chrony', а потім призначити NTP-сервер за допомогою команди 'ntpdate'. Це покаже різницю в часі між сервером та клієнтським комп’ютером і встановить час, який відповідає призначеному серверу. Це метод мінімізації різниці в часі між різними ПК шляхом призначення одного і того ж сервера NTP.

$ sudo apt-get install -y chrony ntpdate

$ sudo ntpdate -q ntp.ubuntu.com

Додавання списку джерел

Ми додамо адресу сховища ROS до ros-latest.list. Відкрийте нове вікно терміналу та введіть наступну команду.

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

Якщо ви використовуєте Linux Mint версії 18.x, використовуйте таку команду. Згаданий вище код $ (lsb_release -sc) отримує кодове ім'я версії дистрибутива Linux. Linux Mint 18.x використовує кодове ім'я Xenial, тому можна додати той же список джерел, що і Ubuntu.

$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu xenial main"> / etc / apt / sources.list.d / ros-latest.list'

Налаштування ключа

Ми додамо відкритий ключ, щоб завантажити пакет із сховища ROS за допомогою наступної команди. Зверніть увагу, що наступний

ключ може змінюватися залежно від ситуації роботи сервера, тому зверніться до офіційної сторінки Wiki 5.

$ sudo apt-key adv --keyserver hkp: //ha.pool.sks-keyservers.net: 80 --recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116

Оновлення індексу пакетів

Тепер, коли ми помістили адресу сховища ROS у вихідний список, ми повинні виконати індексацію списку пакетів знову. Хоча це не є обов’язковим, ми рекомендуємо оновити всі встановлені на даний момент пакети Ubuntu перед установкою ROS.

$ sudo apt-get update && sudo apt-get upgrade -y

Встановлення ROS Kinetic Kame

Ми встановимо пакети ROS для робочого столу, використовуючи наступну команду. Це буде включати ROS, rqt, RViz, пов'язані з роботами бібліотеки, моделювання, навігацію тощо.

$ sudo apt-get install ros-kinetic-desktop-full

Команда вище встановить базовий пакет rqt, але ми можемо додатково встановити всі пакунки, пов’язані з rqt. Встановлення всіх пакунків, пов'язаних з rqt, за допомогою наступної команди полегшить багато аспектів, а також дозволить використовувати різні плагіни rqt.

$ sudo apt-get install ros-kinetic-rqt *

Пакет ROS двійковий

Щоб встановити пакети ROS, ви можете використовувати наступну команду apt-cache для пошуку всіх пакетів, які починаються

на 'ros-kinetic'. Запустивши цю команду, ви зможете побачити приблизно 1600 пакетів.

$ apt-cache пошук ros-kinetic

Якщо ви хочете встановити пакет окремо, ви можете скористатися наступною командою.

$ sudo apt-get install ros-kinetic- [NAME_OF_PACKAGE]

Іншим способом буде використання інструменту графічного інтерфейсу Synaptic Package Manager.

APT (Додатковий інструмент упаковки) 6

Apt (Advanced Packaging Tool) з apt-get, apt-key та apt-cache - це команда управління пакетами, яка широко використовується в Linux серії Debian, таких як Ubuntu та Linux Mint.

http://en.wikipedia.org/wiki/Advanced_Packaging_Tool

Видалення попередньої версії ROS та почергове використання різних версій ROS

'sudo apt-get purge ros-indigo- *' Ця команда дозволяє конфігурацію та видалення файлів. При використанні разом із попередньою версією з команди, яка завантажує конфігураційний файл ROS, доданий до '~ / .bashrc'.

$ source /opt/ros/kinetic/setup.bash

Цю частину можна змінити на кінетичну або індиго.

Ініціалізація rosdep

Не забудьте ініціалізувати rosdep перед використанням ROS. Rosdep - це функція, яка покращує зручність користувача завдяки легкому встановленню залежних пакетів при використанні або компіляції основних компонентів ROS.

$ sudo rosdep init

$ rosdep оновлення

Встановлення rosinstall

Ця програма встановлює різні пакети ROS. Обов'язково встановіть цей корисний інструмент, який часто використовується.

$ sudo apt-get install python-rosinstall

Завантажте файл середовища

Ця команда імпортує файл налаштування середовища. Визначаються змінні середовища, такі як ROS_ROOT, ROS_PACKAGE_PATH тощо.

$ source /opt/ros/kinetic/setup.bash

Створення та ініціалізація папки робочої області

ROS використовує спеціальну систему збірки ROS, що називається catkin. Для цього потрібно створити та ініціалізувати папку робочої області catkin наступним чином. Цю конфігурацію потрібно виконати лише один раз, якщо ви не створили нову папку робочого простору.

$ mkdir -p ~ / catkin_ws / src

$ cd ~ / catkin_ws / src

$ catkin_init_workspace

Тепер, коли ми створили папку робочої області catkin, давайте її створимо. На даний час робоча область catkin містить всередині лише папку 'src' та файл 'CMakeLists.txt', але в якості тесту використовуйте команду 'catkin_make' для побудови.

$ cd ~ / catkin_ws /

$ catkin_make

Після закінчення побудови без помилок запустіть команду 'ls'. На додаток до папки 'src', створеної користувачем, були створені папки 'build' та 'devel'. Файли, пов’язані зі збіркою для системи збірки catkin, зберігаються в папці 'build', а файли, пов'язані з виконанням, зберігаються в папці 'devel'.

$ ls

побудувати

розвивати

src

Нарешті, ми імпортуємо файл налаштувань, пов’язаний із системою побудови catkin.

$ source ~ / catkin_ws / devel / setup.bash

Тестування

Встановлення ROS завершено. Наступна команда перевірить, чи вдала інсталяція. Закрийте всі вікна терміналів і відкрийте нове вікно терміналу. Тепер введіть таку команду, щоб запустити roscore.

$ roscore

Якщо він працює, як зазначено нижче, без помилок, установка завершена. Завершіть процес за допомогою [Ctrl + c] реєстрація на /home/pyo/.ros/log/9e24585a-60c8-11e7-b113-08d40c80c500/roslaunch-pyo-5207.log Перевірка каталогу журналу на використання диска. Це може зайняти деякий час. Натисніть Ctrl-C, щоб перервати

Виконано перевірку використання диска файлу журналу. Використання <1 Гб.

запущений сервер запуску http: // localhost: 38345 /

ros_comm версія 1.12.7

РЕЗЮМЕ

ПАРАМЕТРИ

/ rosdistro: кінетичний

/ роверсія: 1.12.7

ВУЗЛИ

автозапуск нового майстра

процес [майстер]: розпочато з pid [5218]

ROS_MASTER_URI = http: // localhost: 11311 /

setting / run_id до 9e24585a-60c8-11e7-b113-08d40c80c500 процес [rosout-1]: розпочато з pid [5231] запущено основну службу [/ rosout]

Якщо ви використовуєте 16.04.x або Linux Mint 18.x, наступний сценарій дозволить вам спростити згадану процедуру встановлення ROS.

$ wget https://raw.githubusercontent.com/ROBOTIS-GIT/robotis_tools/master/install_ros_kinetic.sh

$ chmod 755 ./install_ros_kinetic.sh

$ bash ./install_ros_kinetic.sh

Сценарій оболонки install_ros_kinetic.sh, завантажений за допомогою швидкої інсталяції командою 'wget', містить загальну процедуру встановлення, описану в розділі 3.1.1, та налаштування середовища ROS, які будуть розглянуті в наступному розділі 3.2.1.

3.2. Середовище розробки ROS

3.2.1. Налаштування ROS

Наступну команду потрібно виконувати кожного разу, коли ми відкриваємо нове вікно терміналу, щоб застосувати налаштування до поточного вікна терміналу.

$ source /opt/ros/kinetic/setup.bash

$ source ~ / catkin_ws / devel / setup.bash

Щоб уникнути цього повторюваного завдання, ми можемо встановити його для імпорту файлу налаштувань, коли ми кожного разу відкриваємо нове вікно терміналу. Крім того, ми можемо налаштувати мережу ROS і створити швидкі команди для часто використовуваних команд.

По-перше, ми використаємо програму текстового редактора 'gedit', щоб відкрити файл '.bashrc'. Ця книга використовує gedit як текстовий редактор за замовчуванням, але ви також можете використовувати atom, піднесений текст, vim, emacs, nano, код візуальної студії тощо.

$ gedit ~ / .bashrc

Коли ви імпортуєте файл .bashrc, існує безліч налаштувань, які вже налаштовано. Не змінюючи жодного з цих налаштувань, прокрутіть униз до файлу 'bashrc' та додайте наступні рядки (замініть xxx.xxx.xxx.xxx на вашу IP-адресу. Будь ласка, зверніться до розділу 'ifconfig' у виносці 7 для налаштування IP-адреси). Після того, як ви все встановили, збережіть зміни та закрийте gedit.

Встановити ROS Kinetic

джерело /opt/ros/kinetic/setup.bash

джерело ~ / catkin_ws / devel / setup.bash

Встановити мережу ROS, експортувати ROS_HOSTNAME = xxx.xxx.xxx.xxx

експортувати ROS_MASTER_URI = http: // $ {ROS_HOSTNAME}: 11311

Встановіть команду псевдоніму ROS

псевдонім cw = 'cd ~ / catkin_ws' псевдонім cs = 'cd ~ / catkin_ws / src'

псевдонім cm = 'cd ~ / catkin_ws && catkin_make'

Тепер ми введемо таку команду, щоб застосувати нові налаштування файлу .bashrc. Ви також можете закрити та відкрити вікно терміналу, щоб застосувати конфігурацію файлу .bashrc. Відтепер кожного разу, коли ви відкриваєте вікно терміналу, до вікна терміналу застосовуватимуться налаштування '.bashrc'.

$ джерело ~ / .bashrc

А тепер давайте детальніше розглянемо те, що ми створили раніше.

Імпортування файлу налаштувань ROS

"#" - це символ, що вказує на початок коментаря, а вміст, що йде далі, - це коментар. 'source /opt/ros/kinetic/setup.bash' у другому рядку та 'source ~ / catkin_ws / devel / setup.bash' у третьому рядку - це файли налаштування ROS, які потрібно налаштувати.

Встановити ROS Kinetic

джерело /opt/ros/kinetic/setup.bash

джерело ~ / catkin_ws / devel / setup.bash

Налаштування мережі ROS

Це конфігурація для ROS_MASTER _URI та ROS_HOSTNAME. ROS використовує мережу для обміну повідомленнями між вузлами, тому конфігурація мережі дуже важлива. По-перше, ми будемо використовувати власну IP-адресу для обох полів. Пізніше, якщо є головний ПК, а робот використовує головний ПК, мережа повинна бути налаштована на обох ПК, щоб

дозволити кільком комп’ютерам взаємодіяти між собою. Наразі ми введемо власну мережеву IP-адресу в обидва поля, як використовуватиме лише один ПК. Наступний приклад показує випадок, як налаштувати, якщо ваша IP-адреса "192.168.1.100". Ви можете перевірити свою IP-інформацію у вікні терміналу за допомогою команди 'ifconfig'.

Встановити мережу ROS

експортувати ROS_HOSTNAME = 192.168.1.100

експортувати ROS_MASTER_URI = http: // $ {ROS_HOSTNAME}: 11311

Якщо ви запускаєте всі пакети на одному ПК, немає необхідності призначати певний IP, а замість цього призначте 'localhost' для обох полів.

Встановити мережу ROS

експортувати ROS_HOSTNAME = localhost

експортувати ROS_MASTER_URI = http: // localhost: 11311

Для перевірки IP-адреси в Linux використовуйте команду ifconfig. Запустивши команду ifconfig у вікні терміналу, як показано в наступному прикладі, IP-адреса відображатиметься поруч із «inet addr» у «enp3s0» для дротової локальної мережі та в «wlp2s0» для бездротової локальної мережі. У наступному прикладі показано як бездротову, так і дротову локальну мережу. IP для дротового

підключення до LAN - 192.168.1.100, тоді як бездротова мережа - 192.168.11.19.

$ ifconfig

MTU: 65536 Метричний: 1

```
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:3520 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3520 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:560728 (560.7 KB)  TX bytes:560728 (560.7 KB)

wlp2s0    Link encap:Ethernet  HWaddr 08:d4:0c:80:c5:00
          inet addr:192.168.11.19  Bcast:192.168.11.255  Mask:255.255.255.0
          inet6 addr: fe80::a60b:e157:4157:d9dc/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:675821 errors:0 dropped:0 overruns:0 frame:0
          TX packets:219992 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:919561165 (919.5 MB)  TX bytes:46928931 (46.9 MB)
```

Швидка команда

Давайте встановимо швидкі команди, які часто використовуються при розробці ROS. Наступні cw, cs та cm - це власні команди, які я визначив як команди псевдонімів.

- **cw**: Перейти до попередньо визначеного каталогу робочої області catkin '~ / catkin_ws'
- **cs**: Перейдіть до каталогу '~ / catkin_ws / src' у каталозі робочої області catkin, який містить вихідні файли

- **cм**: Перейдіть до каталогу робочої області catkin '~ / catkin_ws' та створіть ROS-пакети за допомогою команда 'catkin_make'

Встановіть команду псевдоніму ROS

псевдонім cw = 'cd ~ / catkin_ws' псевдонім cs = 'cd ~ / catkin_ws / src'

псевдонім cm = 'cd ~ / catkin_ws && catkin_make'

Інтегроване середовище розробки (IDE) забезпечує середовище розробки, щоб користувач міг виконувати завдання, пов'язані з розробкою програми, такі як кодування, налагодження, компіляція, розподіл в одній програмі. Більшість розробників мають принаймні пару своїх улюблених середовищ розробки.

ROS підтримує багато середовищ розробки. Найчастіше використовуються IDE - Eclipse, CodeBlocks, Emacs, Vim, NetBeans, QtCreator. У моєму випадку я раніше працював з Eclipse, але він став досить важким в останніх версіях, і мені було незручно використовувати з системою побудови котів ROS. Тому, розглянувши інші IDE, здається, що найбільш підходящим інструментом для простих завдань буде Visual Studio Code, а для розробки графічного інтерфейсу - QtCreator. Особливо, rqt та RViz для розробки ROS, налагодження, візуалізації розробляються за допомогою Qt, і той факт, що користувачі можуть розробляти плагіни для ROS-інструментів за допомогою плагінів Qt, робить QtCreator дуже корисним.

Крім того, навіть якщо ви не використовуєте Qt, цілком адекватно використовувати його як загальний редактор, і він може імпортувати проект безпосередньо через 'CMakeLists.txt', що робить його дуже зручним при використанні 'catkin_make'.

Наступний розділ містить інформацію про налаштування середовища розробки ROS за допомогою QtCreator. Однак я хотів би чітко пояснити, що навіть якщо ви використовуєте інші середовища розробки, не буде проблем із розумінням змісту цієї книги.

Встановлення QtCreator

$ sudo apt-get install qtcreator

Запуск QtCreator

QtCreator також можна запустити з піктограмою. Однак, якщо ми хочемо застосувати до QtCreator такі параметри, як шлях ROS, який ми налаштували в '~ / .bashrc', тоді ми повинні відкрити нове вікно терміналу та ввести наступну команду для запуску QtCreator. У цьому питанні всі налаштування, налаштовані в '~ / .bashrc', будуть застосовані під час запуску QtCreator.

$ qtcreator

QtCreator було відкрито командою вище, як показано на малюнку 12.

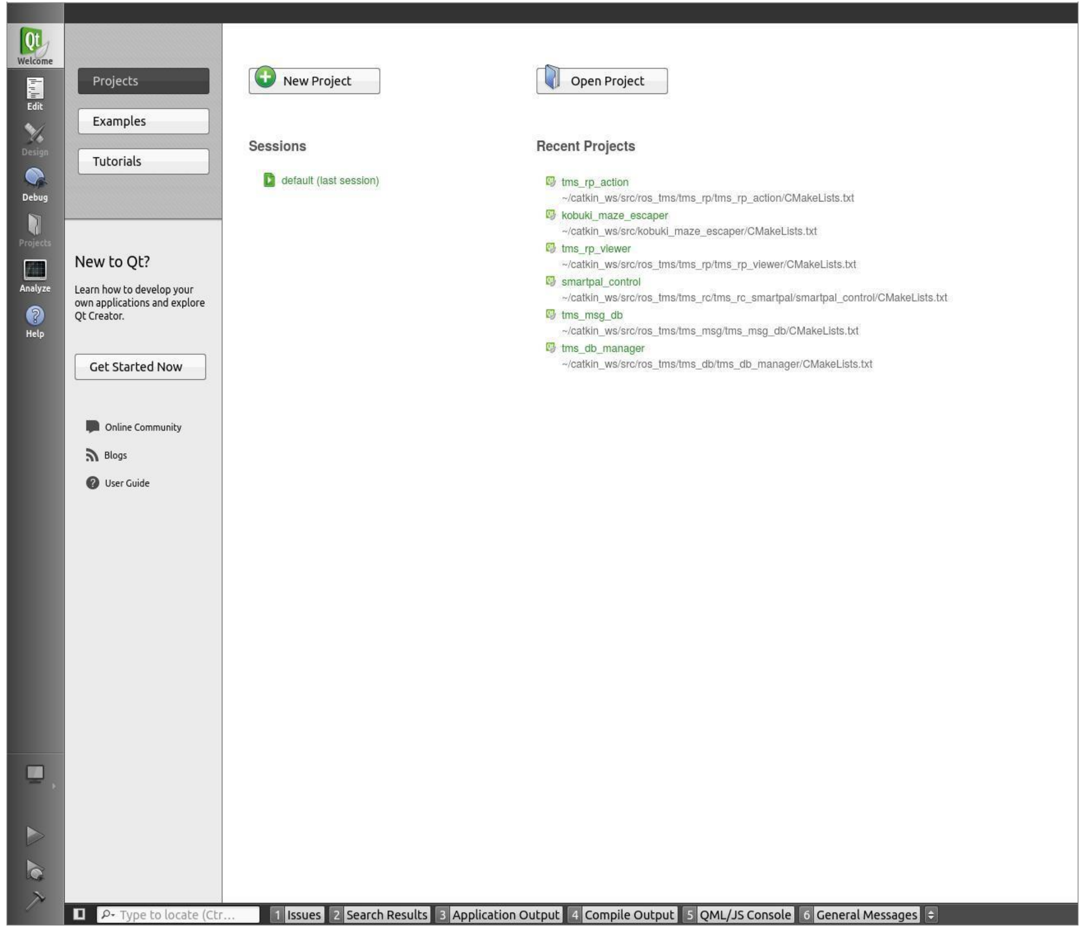


Рис. 12 IDE QtCreator

Імпорт ROS-пакету як проекту

Як вже згадувалося раніше, QtCreator використовує 'CMakeLists.txt', а пакет ROS також базується на 'CMakeLists.txt', тому, натиснувши кнопку 'OpenProject' і вибравши 'CMakeLists.txt' певного пакета ROS, ви можете легко імпортуйте пакет як проект, як показано на рис. 13.

Ярлик для побудови - [Ctrl + b], і 'catkin_make' буде виконано під час компіляції вихідного коду. Вбудовані файли будуть збережені в папці "build" у тому ж каталозі пакета. Наприклад, коли ви компілюєте пакет "tms_rp_action", вбудовані файли поміщаються в "build-tms_rp_ action-Desktop-Default folder". Файли, які спочатку зберігалися в '~ / catkin_ws / build' та '~ / catkin_ws / devel', компілюються окремо та розміщуються в новому розташуванні, тому

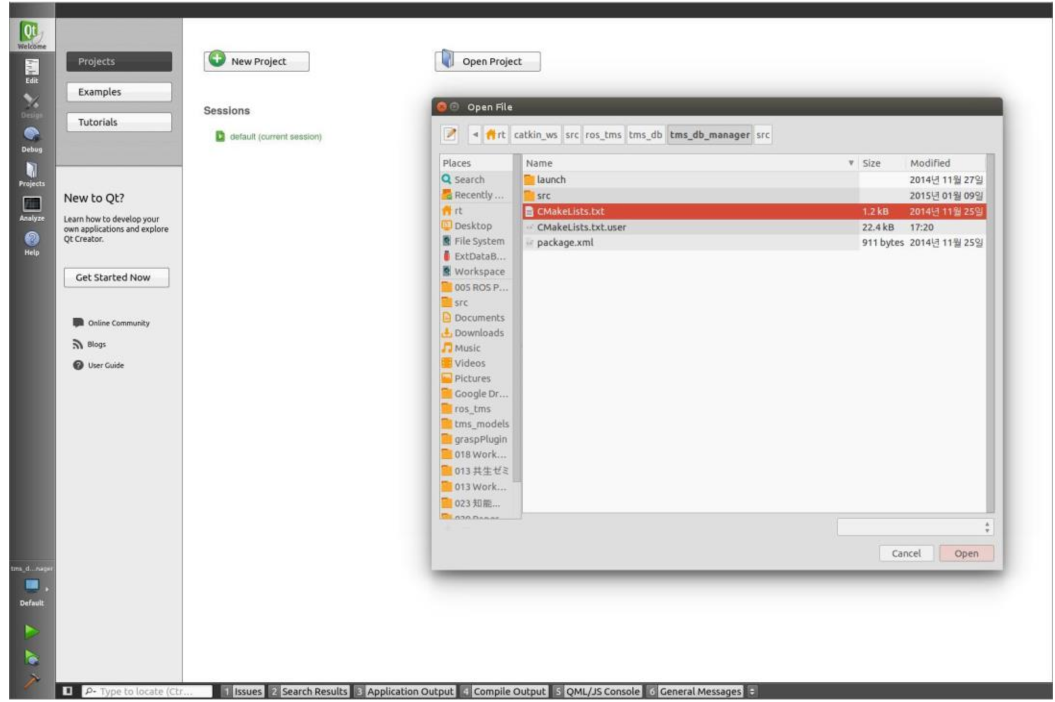


Рис. 13 Відкриття проекту з QtCreator

для його запуску вам пізніше потрібно буде знову запустити 'catkin_make' у вікно терміналу. Не потрібно повторювати цей процес щоразу, тому під час розробки ми можемо розробляти та налагоджувати в QtCreator, а після завершення розробки ми можемо

використовувати 'catkin_make'. Для вашої інформації існує плагін Qt Creator для ROS (https: // github.com)

На додаток до QtCreator, ми також рекомендуємо Visual Studio Code та Eclipse. Visual Studio Code - це дуже легкий редактор, такий як

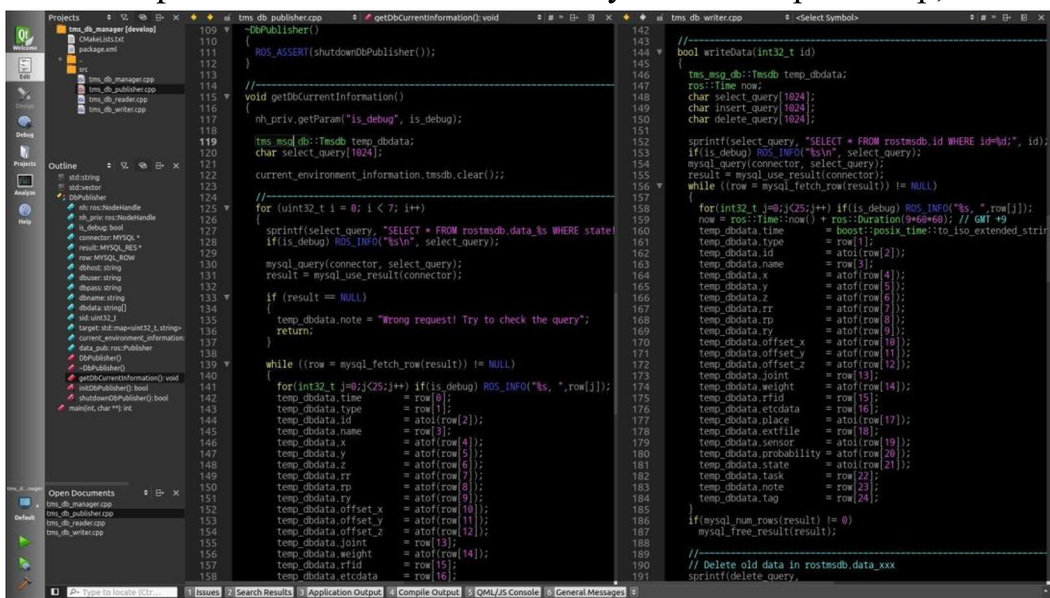


Рис. 14 Робоча область проекту QtCreator

'Atom', 'Sublime Text', 'Clion', тому він дуже швидкий, а розширення ROS полегшує використання ROS. Eclipse - це IDE загального призначення, що використовується дуже великою кількістю людей, а також використовується багатьма користувачами ROS. Для отримання додаткової інформації див. наступну вікі.

3.3. Тест експлуатації ROS

Тепер, коли ми встановили ROS, давайте перевіримо, чи працює він правильно. Наступний приклад - пакет turtlesim (набір

вузлів), наданий ROS для відображення черепахи на екрані та управління черепахою за допомогою вузла клавіатури (програми).

Починаючи з цього розділу, з'являться багато специфічних для ROS термінів, таких як node, package та roscore, що буде детально пояснено в розділі 4. Термінологія ROS. У цьому розділі ми перевіримо, що ROS було встановлено без проблем.

Запуск roscore

```
File  Edit  View  Search  Terminal  Help
pyo@pyo ~ $ roscore
... logging to /home/pyo/.ros/log/d257f510-60cc-11e7-b113-08d40c80c500/roslaunch
-pyo-7562.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://localhost:38881/
ros_comm version 1.12.7


SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.7

NODES

auto-starting new master
process[master]: started with pid [7573]
ROS_MASTER_URI=http://localhost:11311/

setting /run_id to d257f510-60cc-11e7-b113-08d40c80c500
process[rosout-1]: started with pid [7586]
started core service [/rosout]
```

Рис. 15 Екран, що показує працюючий

Відкрийте нове вікно терміналу (Ctrl + Alt + t) і запустіть наступну команду. Це буде запускати roscore, який буде контролювати всю систему ROS.

Запуск turtlesim_node у пакеті turtlesim

Відкрийте нове вікно терміналу та введіть наступну команду. Тоді ви побачите додані повідомлення, і turtlesim_node в пакеті turtlesim буде виконано. У середині блакитного вікна ви побачите

черепаху (форма черепахи може змінюватися випадковим чином, коли її виконують, так що вона може виглядати інакше, як на малюнку 3-6).

$ rosrun turtlesim turtlesim_node

[ІНФОРМАЦІЯ] [1499182058.960816044]: Запуск turtlesim з іменем вузла / turtlesim

[ІНФОРМАЦІЯ] [1499182058.966717811]: нерестова черепаха [черепаха1] при x = [5.544445], y = [5.544445], theta = [0.000000]

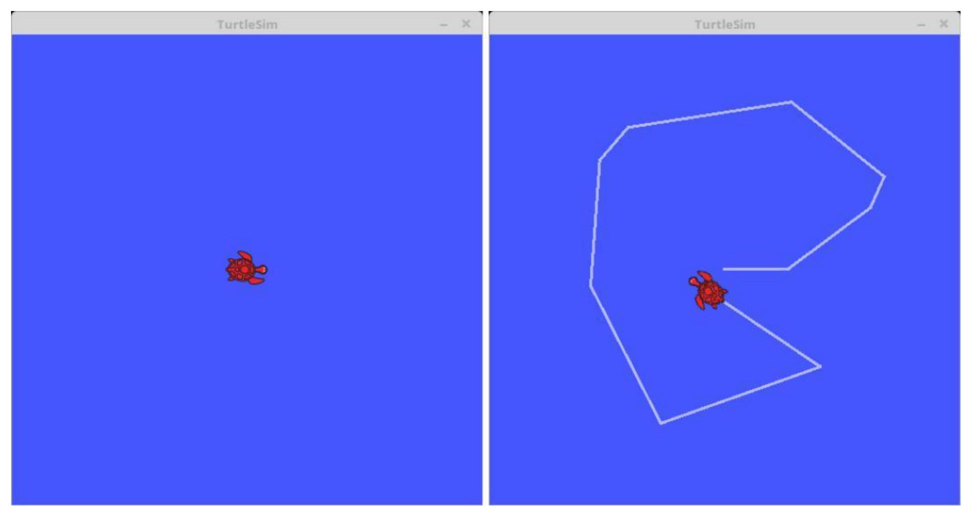


Рис. 16 Екран, що відображає черепаху, яку рухають

Запуск turtle_teleop_key у пакеті turtlesim

Відкрийте нове вікно терміналу та введіть наступну команду. Тоді ви побачите додані повідомлення та інструкції, і turtle_teleop_key пакета turtlesim буде виконано. Якщо в цьому вікні терміналу натиснути будь-яку з клавіш зі стрілками на клавіатурі (←, →, ↑, ↓), черепаха рухатиметься відповідно до клавіші зі стрілкою, як показано на малюнку справа на малюнку 3-6. Ви повинні ввести ключ у

відповідному вікні терміналу. Хоча це лише просте моделювання, ми зможемо керувати реальним роботом тим же методом.

$ rosrun turtlesim turtle_teleop_key

Читання з клавіатури

Використовуйте клавіші зі стрілками для переміщення черепахи.

Using the [Tab] key in the terminal window

In Linux we often enter commands in the terminal window. There are many users who are unfamiliar with the command line interface, but as one becomes proficient with it, it becomes a very fast and convenient method. However, even experienced users do not memorize all the commands and frequently use the [Tab] key instead. In the Linux terminal window, the [Tab] key supports auto-completion. This feature eliminates the need to memorize all the commands, and lets you enter commands quickly and accurately without a typo. As an example, take a look at the rosrun command used earlier. As shown below, after typing turtlesim we can use the [Tab] key to find the various nodes that we can use in the turtlesim package.

```
$ rosrun turtlesim [Tab]
```

Additionally, if we type turtle_teleop and press the [Tab] key, then it will auto-complete the rest of command that we can use. This applies not only for ROS but for all Linux commands, so we suggest you to make use of this feature.

```
$ rosrun turtlesim turtle_teleop[Tab]
$ rosrun turtlesim turtle_teleop_key
```

Запуск rqt_graph у пакеті rqt_graph

Введення команди rqt_graph у новому вікні терміналу запустить вузол rqt_graph у пакеті rqt_graph. Результат показаний у вигляді схеми інформації поточно працюючих вузлів (програм), як показано на рис. 17.

$ rqt_graph

Вузол rqt_graph відображає інформацію про поточно працюючі вузли у формі графічного інтерфейсу. Коло представляє вузол, а квадрат - тему. Якщо ми подивимося на рис. 17, від вузла '/ teleop_turtle', який з'єднується з '/ turtlesim', проводиться стрілка. Це демонструє, що два вузли працюють, і між цими двома вузлами відбувається обмін повідомленнями.

Квадратне поле '/ turtle1 / cmd_vel', яке є підтемою теми turtle1, розташоване між двома стрілками, є назвою теми для двох вузлів і відображає, що команда швидкості, введена за допомогою клавіатури у вузлі teleop_turtle надсилається на вузол turtlesim як повідомлення в темі.

Тобто, використовуючи два вищезгадані вузли, команди клавіатури передавались в моделювання робота. Більш детальна інформація буде пояснена в наступних розділах, і якщо ви дотепер добре дотримувались результатів, ви пройшли перевірку роботи ROS.

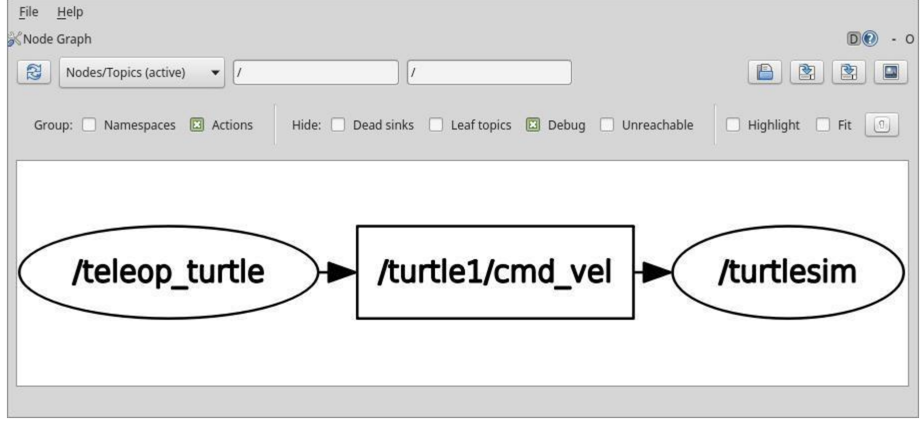


Рис. 17 Вузол rqt_graph

Закриття вузла

Перелік та кожен вузол можна завершити, натиснувши [Ctrl + c] у відповідних вікнах терміналу. Для вашої інформації [Ctrl + c] використовується для примусового закриття програми на Linux / Unix.

Розділ 4

Важливі поняття

АФК

Для того, щоб розробити робота, пов'язаного з ROS**1**, необхідно зрозуміти основні компоненти та концепції АФК. У цьому розділі буде представлено термінологію, що використовується в ROS, та важливі поняття ROS, такі як комунікація повідомлень, файл повідомлення, ім'я, перетворення координат (TF), клієнтська бібліотека, зв'язок між різнорідними пристроями, файлова система та система побудови.

4.1. Термінологія ROS

У цьому розділі пояснюються найбільш часто використовувані терміни ROS. Використовуйте цей розділ як глосарій ROS. Багато термінів можуть бути новими для читача, і навіть якщо є незнайомі терміни, перегляньте визначення та рухайтесь далі. Ви станете більш знайомими з поняттями, коли будете брати участь у прикладах та вправах у кожному з наступних розділів.

АФК

ROS надає стандартні послуги операційної системи, такі як апаратне абстрагування, драйвери пристроїв, реалізація загальновживаних функцій, включаючи зондування, розпізнавання, зіставлення, планування руху, передачу повідомлень між процесами, управління пакетами, візуалізатори та бібліотеки для розробки, а також засоби налагодження.

Майстер

Майстер**2**діє як сервер імен для з'єднань між вузлами та зв'язку. Команда roscore використовується для запуску ведучого, і якщо ви запустите ведучий, ви можете зареєструвати ім'я кожного вузла та отримати інформацію за потреби. Зв’язок між вузлами та комунікацією повідомлень, таких як теми та послуги, неможливий без майстра.

Ведучий спілкується з підлеглими за допомогою XMLRPC (XML-віддалений виклик процедур)**3**, який є протоколом на основі HTTP, який не підтримує зв’язок. Іншими словами, підлеглі вузли можуть отримати доступ лише тоді, коли їм потрібно зареєструвати власну інформацію або запитувати інформацію інших вузлів. Стан з'єднання один одного не перевіряється регулярно. Завдяки цій функції ROS можна використовувати у дуже великих та складних середовищах. XMLRPC дуже легкий і підтримує різноманітні мови програмування, що робить його придатним для ROS, який підтримує різноманітні апаратні засоби та мови програмування.

Коли ви виконуєте ROS, ведучий буде налаштований за допомогою адреси URI та порту, налаштованих у ROS_MASTER_URI. За замовчуванням URI-адреса використовує IP-адресу локального ПК та номер порту 11311, якщо інше не змінено.

Вузол

Вузол[4]відноситься до найменшої одиниці процесора, що працює в ROS. Подумайте про це як про одну виконувану програму. ROS рекомендує створити один єдиний вузол для кожної мети, і рекомендується розробляти для зручності повторного використання. Наприклад, у випадку мобільних роботів програма для роботи з роботом розбита на спеціалізовані функції. Спеціалізований вузол використовується для кожної функції, наприклад приводу датчика, перетворення даних датчика, розпізнавання перешкод, приводу двигуна, введення кодера та навігації.

Після запуску вузол реєструє таку інформацію, як ім'я, тип повідомлення, URI-адреса та номер порту вузла. Зареєстрований вузол може діяти як видавець, абонент, сервер обслуговування чи клієнт служби на основі зареєстрованої інформації, а вузли можуть обмінюватися повідомленнями за допомогою тем та служб.

Вузол використовує XMLRPC для зв'язку з ведучим і використовує XMLRPC або TCPROS[5]протоколів TCP / IP при спілкуванні між вузлами. Запит на підключення та відповідь між вузлами використовують XMLRPC, а обмін повідомленнями використовує TCPROS, оскільки це прямий зв’язок між вузлами, незалежними від ведучого. Що стосується адреси URI та номера порту, то в якості адреси URI використовується змінна ROS_HOSTNAME, яка зберігається на комп'ютері, де працює вузол, а для порту встановлено довільне унікальне значення.

Пакет

Пакет[6]є основною одиницею АФК. Додаток ROS розробляється на основі пакетів, і пакет містить або файл конфігурації для запуску інших пакетів або вузлів. Пакет також містить усі файли, необхідні для запуску пакету, включаючи бібліотеки залежностей ROS для запуску різних процесів, наборів даних та файлу конфігурації. Кількість офіційних пакетів становить близько 2500 для ROS Indigo станом на липень 2017 року (http://repositories.ros.org/status_page/ ros_indigo_ default.html) та близько 1600 пакетів для ROS Kinetic (http://repositories.ros.org / status_page / ros_kinetic_default.html). Крім того, хоча можуть бути деякі надмірності, існує близько 4600 пакетів, розроблених і випущених користувачами (http://rosindex.github.io/stats/).

Метапакет

Метапакет[7]- це набір пакетів, що мають спільне призначення. Наприклад, навігаційний метапакет складається з 10 пакетів, включаючи AMCL, DWA, EKF та map_server.

Повідомлення

Вузол 8 надсилає або отримує дані між вузлами за допомогою повідомлення. Повідомлення - це такі змінні, як ціле число, плаваюча крапка та булева. В повідомленні може бути використана вкладена структура повідомлень, що містить інші повідомлення або масив повідомлень.

Для доставки повідомлень використовується протокол зв'язку TCPROS та UDPROS. Тема використовується при односпрямованій доставці повідомлень, тоді як служба використовується при двонаправленій доставці повідомлень, що включає запит та відповідь.

Тема

Тема 9 буквально як тема в розмові. Вузол видавця спочатку реєструє свою тему в майстрі, а потім починає публікувати повідомлення за темою. Абонентські вузли, які хочуть отримувати інформацію про запит теми вузла видавця, що відповідає імені теми, зареєстрованій у майстрі. На основі цієї інформації абонентський вузол безпосередньо підключається до вузла видавця для обміну повідомленнями як теми.

Публікація та видавництво

Термін "опублікувати" означає дію передачі відносних повідомлень, що відповідають темі. Вузол видавця реєструє власну інформацію та тему з ведучим та надсилає повідомлення підключеним абонентським вузлам, які зацікавлені в тій самій темі. Видавець оголошується у вузлі і може бути оголошений кілька разів в одному вузлі.

Підписка та передплатник

Термін "підписатись" означає дію отримання відповідних повідомлень, що відповідають темі. Абонентський вузол реєструє власну інформацію та тему з ведучим і отримує інформацію про

видавця, яка публікує відносну тему від головного. На основі отриманої інформації про видавця абонентський вузол безпосередньо запитує підключення до вузла видавця та отримує повідомлення від підключеного вузла видавця. Абонент оголошується у вузлі і може бути оголошений кілька разів в одному вузлі.

Тематичне спілкування - це асинхронне спілкування, яке базується на видавництві та передплатнику, і корисно передавати певні дані. Оскільки тема постійно передає та отримує потік повідомлень після підключення, вона часто використовується для датчиків, які повинні періодично передавати дані. З іншого боку, існує потреба у синхронному спілкуванні, з яким використовуються запит та відповідь. Таким чином, ROS забезпечує метод синхронізації повідомлень, який називається «послуга». Послуга складається із серверного сервера, який відповідає на запити, та клієнтського сервісу, який вимагає відповісти. На відміну від теми, послуга є разовим повідомленням комунікації. Коли запит і відповідь служби завершені, зв’язок між двома вузлами розривається.

Обслуговування

Сервіс - це синхронний двоспрямований зв’язок між клієнтом служби, який запитує послугу щодо певного завдання, і сервером служби, який відповідає за відповідь на запити.

Сервер обслуговування

«Сервер-сервер» - це сервер у повідомленні службового повідомлення, який отримує запит як вхід і передає відповідь як вихід. І запит, і відповідь мають форму повідомлень. На запит на послугу сервер виконує призначену послугу та доставляє результат клієнту служби у відповідь. Сервер служби реалізований у вузлі, який приймає та виконує заданий запит.

Клієнт послуги

«Клієнт послуги» - це клієнт, який надсилає повідомлення на сервіс і отримує відповідь як вхід. І запит, і відповідь подаються у формі повідомлення. Клієнт відправляє запит на сервер обслуговування і отримує відповідь. Клієнт служби реалізований у вузлі, який запитує вказану команду і отримує результати.

Дія

Дія є іншим способом передачі повідомлень, що використовується для асинхронного двонаправленого зв'язку. Дія застосовується там, де після отримання запиту потрібно більше часу, щоб відповісти, а проміжні відповіді потрібні до повернення результату. Структура файлу дій також подібна до структури служби. Однак розділ даних зворотного зв'язку для проміжної відповіді додається разом із розділом даних про цілі та результати, які представлені як запит та відповідь в службі відповідно. Є клієнт дії, який встановлює мету сервера дій та дії, який виконує дію, визначену ціллю, і повертає відгук та результат клієнту дії.

Сервер дій

"Сервер дій" відповідає за отримання цілі від клієнта та відповідь відгуком та результатом. Як тільки сервер отримує ціль від клієнта, він виконує заздалегідь визначений процес.

Клієнт дії

"Клієнт дії" відповідає за передачу цілі на сервер і отримує дані результатів або зворотного зв'язку як вхідні дані від сервера дій. Клієнт доставляє ціль на сервер дій, потім отримує відповідний результат або зворотний зв'язок і передає подальші інструкції або скасовує інструкції.

Параметр

Параметр в ROS відноситься до параметрів, що використовуються у вузлі. Подумайте про це як про файли конфігурації * .ini у програмі Windows. Значення за замовчуванням встановлюються в параметрі і можуть бути прочитані або записані при необхідності. Зокрема, це дуже корисно, коли налаштовані значення можна змінювати в режимі реального часу. Наприклад, ви можете вказати такі налаштування, як номер порту USB, параметри калібрування камери, максимальні та мінімальні значення швидкості двигуна.

Сервер параметрів

Коли параметри викликаються в пакеті, вони реєструються на сервері параметрів який завантажується в майстер.

Кішка

Кішка 14 відноситься до системи збірки ROS. Система збірки в основному використовує CMake (Cross Platform Make), а середовище збірки описано у файлі 'CMakeLists.txt' у папці пакета. CMake було модифіковано в ROS для створення ROS-системи побудови. Catkin розпочав альфа-тест з ROS Fuerte, і основні пакети почали переходити на Catkin у версії ROS Groovy. Catkin застосовується до більшості пакунків у версії ROS Hydro. Система збірки Catkin спрощує використання збірок ROS, управління пакетами та залежностей між пакетами. Якщо ви збираєтеся використовувати ROS в цей момент, вам слід використовувати Catkin замість ROS build (rosbuild).

Побудова ROS

Збірка ROS (rosbuild) - це система складання, яка використовувалася до системи збірки Catkin. Хоча є деякі користувачі, які все ще використовують його, це зарезервовано для сумісності ROS, тому офіційно використовувати його не рекомендується. Якщо потрібно використовувати старий пакет, який підтримує лише rosbuild, ми рекомендуємо використовувати його після перетворення rosbuild на catkin.

Роскор

Роскор - це команда, яка запускає майстер ROS. Якщо кілька комп'ютерів знаходяться в одній мережі, її можна запустити з іншого комп'ютера в мережі. Однак, за винятком особливих випадків, що

підтримують множинні результати, у мережі повинен працювати лише один список. Коли запущено майстер ROS, використовуються адреса URI та номер порту, призначені змінним середовища ROS_MASTER_URI. Якщо користувач не встановив змінну середовища, в якості адреси URI використовується поточна локальна IP-адреса, а використовується номер порту 11311, який є номером порту за замовчуванням для ведучого.

Rosrun

Rosrun є основною командою виконання ROS. Він використовується для запуску одного вузла в пакеті. Вузол використовує змінну середовища ROS_HOSTNAME, що зберігається на комп'ютері, на якому працює вузол, як адресу URI, а для порту встановлено довільне унікальне значення.

Roslaunch

Хоча rosrun - це команда для виконання одного вузла, roslaunch 18 на противагу виконує кілька вузлів. Це команда ROS, що спеціалізується на виконанні вузлів із додатковими функціями, такими як зміна параметрів пакета або імен вузлів, налаштування простору імен вузлів, встановлення ROS_ ROOT та ROS_PACKAGE_PATH та зміна змінних середовища.19 при виконанні вузлів.

Roslaunch використовує файл '* .launch', щоб визначити, які вузли виконувати. Файл заснований на XML (розширювана мова розмітки) і пропонує безліч варіантів у вигляді тегів XML.

Bag

Дані з ROS-повідомлень можна записувати. Формат файлу називається bag20, а '* .bag' використовується як розширення файлу. У ROS пакет можна використовувати для запису повідомлень та відтворення їх, коли це необхідно для відтворення середовища під час запису повідомлень. Наприклад, під час проведення експерименту з роботом за допомогою датчика значення датчика зберігаються у формі повідомлення за допомогою пакета. Це записане повідомлення можна повторно завантажувати, не виконуючи те саме тестування, відтворюючи збережений файл сумки. Функції запису та відтворення rosbag особливо корисні при розробці алгоритму з частими модифікаціями програм.

Ros Wiki

ROS Wiki - це основний опис ROS на основі Wiki (http://wiki.ros.org/), який пояснює кожен пакет та функції, що надаються ROS. Ця сторінка Wiki описує основне використання ROS, короткий опис кожного пакету, використовувані параметри, автора, ліцензію, домашню сторінку, сховище та навчальний посібник. На даний момент ROS Wiki має понад 18 800 сторінок вмісту.

Сховище

Відкритий пакет визначає сховище на сторінці Wiki. Сховище - це URL-адреса в Інтернеті, де зберігається пакет. Сховище керує проблемами, розробкою, завантаженнями та іншими функціями за

допомогою систем контролю версій, таких як svn, hg та git. Багато ROS-пакетів, які зараз доступні, використовують GitHub 21як сховища для вихідного коду. Для того, щоб переглянути вміст вихідного коду для кожного пакета, перевірте відповідне сховище.

Графік

Взаємозв'язок між вузлами, темами, видавцями та передплатниками, представлені вище, можна зобразити у вигляді графіку. Графічне представлення повідомлень не включає послугу, оскільки це відбувається лише один раз. Графік можна відобразити, запустивши вузол 'rqt_graph' у пакеті 'rqt _graph'. Є дві команди виконання, 'rqt_graph' і 'rosrun rqt_graph rqt_graph'.

Ім'я

Вузли, параметри, теми та служби мають усі імена 22. Ці імена реєструються на головній панелі та шукаються за іменем для передачі повідомлень при використанні параметрів, тем та служб кожного вузла. Імена гнучкі, оскільки їх можна змінювати під час виконання, а різні імена можна призначати при багаторазовому виконанні однакових вузлів, параметрів, тем та служб. Використання імен робить ROS придатними для масштабних проектів та складних систем.

Клієнтська бібліотека

ROS забезпечує середовища розробки для різних мов за допомогою клієнтської бібліотеки23з метою зменшення залежності від використовуваної мови. Основними клієнтськими бібліотеками є

C ++, Python, Lisp та інші мови, такі як Java, Lua, .NET, EusLisp та R. Для цього розроблено клієнтські бібліотеки, такі як roscpp, rospy, roslisp, rosjava, roslua, roscs, roseus, PhaROS та rosR.

URI

URI (уніфікований ідентифікатор ресурсу) - це унікальна адреса, яка представляє ресурс в Інтернеті. URI є одним з основних компонентів, що забезпечує взаємодію з Інтернетом і використовується як ідентифікатор в Інтернет-протоколі.

MD5

MD5 (Алгоритм повідомлення-дайджест 5) 24 є 128-розрядною криптографічною хеш-функцією. Він використовується в основному для перевірки цілісності даних, наприклад, для перевірки того, чи програми чи файли перебувають у незміненому вихідному вигляді. Цілісність передачі / прийому повідомлення в АФК перевіряється за допомогою MD5.

RPC

RPC (віддалений виклик процедур) 25 означає функцію, яка викликає додаткову процедуру на віддаленому комп'ютері з іншого комп'ютера в мережі. RPC використовує такі протоколи, як TCP / IP та IPX, і дозволяє виконувати функції або процедури, не маючи розробника писати програму для віддаленого управління.

XML

XML (Extensible Markup Language) - це широка та універсальна мова розмітки, яку W3C рекомендує для створення інших мов розмітки спеціального призначення. XML використовує теги для опису структури даних. У ROS він використовується в різних компонентах, таких як * .launch, * .urdf та package.xml.

XMLRPC

XMLRPC (XML-віддалений виклик процедур) - це тип протоколу RPC, який використовує XML як формат кодування та використовує метод запиту та відповіді протоколу HTTP, який не підтримує та не перевіряє з'єднання. XMLRPC - це дуже простий протокол, який використовується лише для визначення малих типів даних або команд. Як результат, XMLRPC дуже легкий і підтримує різноманітні мови програмування, що робить його придатним для ROS, який підтримує різноманітне обладнання та мови.

TCP / IP

TCP розшифровується як протокол управління передачею. Його часто називають TCP / IP. Рівень протоколу Інтернету гарантує передачу даних за допомогою протоколу TCP, який базується на рівні IP (Internet Protocol) у рівнях протоколу Інтернету. Це гарантує послідовну передачу та прийом даних.

TCPROS - це формат повідомлень на основі TCP / IP, а UDPROS - формат повідомлень, заснований на UDP.

TCPROS частіше використовується в ROS.

CMakeLists.txt

Catkin, система збірки ROS, за замовчуванням використовує CMake. Середовище збірки вказано в 'CMakeLists.txt' 26 файл у кожній папці пакунка.

package.xml

XML-файл 27 містить інформацію про пакет, яка описує назву пакета, автора, ліцензію та залежні пакети.

4.2. Повідомлення спілкування

Поки що ми лише дали вступ до ROS, але не детально пояснили, як працює ROS. У цьому розділі ми розглянемо основні функції та концепції ROS. Детальний опис кожного терміна, що використовується в описі концепції ROS, наведено в глосарії термінів, обговорених раніше. Цей розділ стосуватиметься лише понять ROS. Фактичний метод програмування висвітлений у главі 7, Основне програмування ROS.

Як описано в главі 2, ROS розроблено в одиниці вузлів, що є мінімальною одиницею виконуваної програми, яка розбита для максимального повторного використання. Вузол обмінюється даними з іншими вузлами через повідомлення, утворюючи велику програму в цілому. Ключовою концепцією тут є методи обміну повідомленнями

між вузлами. Існує три різні методи обміну повідомленнями: тема, яка забезпечує односпрямовану передачу / прийом повідомлення, послуга, яка забезпечує двосторонній запит / відповідь на
повідомлення та дія, яка забезпечує двонаправлене повідомлення цілі / результату / зворотного зв'язку. Крім того, параметри, що використовуються у вузлі, можуть бути змінені із зовнішньої сторони вузла. Це також можна розглядати як тип комунікації повідомлень у ширшому контексті. Зв'язок з повідомленнями проілюстрований на малюнку 4-1, а відмінності узагальнені на рис. 18. Важливо використовувати кожну тему, послугу, дію та параметр відповідно до її правильного призначення під час програмування на ROS.

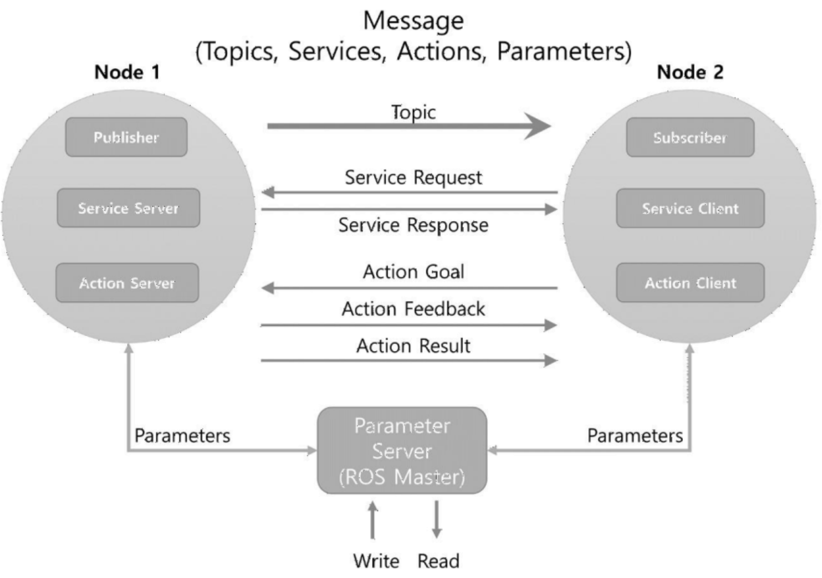


Схема 1. Обмін повідомленнями між вузлами

Type	Features		Description
Topic	Asynchronous	Unidirectional	Used when exchanging data continuously
Service	Synchronous	Bi-directional	Used when request processing requests and responds current states
Action	Asynchronous	Bi-directional	Used when it is difficult to use the service due to long response times after the request or when an intermediate feedback value is needed

Спілкування за темою використовує однаковий тип повідомлень як для видавця, так і для передплатника, як показано на рис. 19. Абонентський вузол отримує інформацію про вузол видавця, що відповідає ідентичному імені теми, зареєстрованому в головному. На основі цієї інформації абонентський вузол безпосередньо підключається до вузла видавця для отримання повідомлень. Наприклад, якщо поточне положення робота генерується у вигляді одометрії28 інформацію шляхом обчислення значень кодера обох

коліс мобільного робота, інформація про асинхронну одометрію може безперервно передаватися в односпрямованому потоці, використовуючи тематичне повідомлення (x, y,і). Оскільки теми є односпрямованими і залишаються пов'язаними для постійного надсилання або отримання повідомлень, він підходить для даних датчиків, які вимагають періодичної публікації повідомлень. Крім того, кілька абонентів можуть отримувати повідомлення від видавця і навпаки. Також доступні кілька підключень видавців та передплатників.

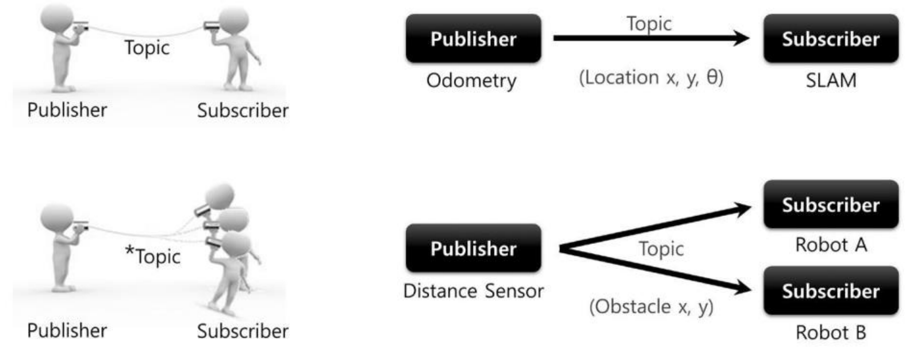


Рис. 19 Повідомлення теми

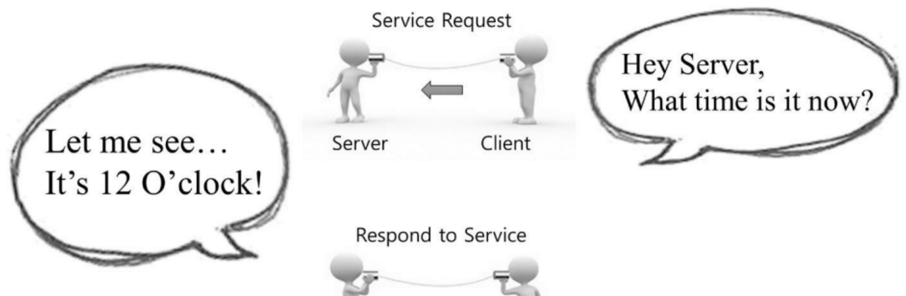


Рис. 20 Повідомлення службових повідомлень

Зв'язок про послугу - це двонаправлений синхронний зв'язок між клієнтом послуги, що запитує послугу, і сервером служби, що відповідає на запит, як показано на рис. 2. Вищезазначені "публікація" та "передплата" теми є асинхронним методом, який є вигідним при періодичній передачі даних. З іншого боку, існує потреба у синхронному спілкуванні, яке використовує запит та відповідь. Відповідно, ROS забезпечує синхронізований спосіб передачі повідомлень, який називається «послуга».

Послуга складається із серверного сервера, який відповідає лише тоді, коли є запит, і клієнта служби, який може надсилати запити, а також отримувати відповіді. На відміну від теми, послуга є одноразовим повідомленням. Отже, коли запит і відповідь служби завершені, зв'язок між двома вузлами буде розірвано. Послуга часто використовується для командування роботом виконати певну дію або вузлів для виконання певних подій з певною умовою. Сервіс не підтримує з'єднання, тому корисно зменшити навантаження на мережу, замінивши тему. Наприклад, якщо клієнт запитує сервер на

поточний час, як показано на рис. 20, сервер перевірить час і відповість клієнту, і з'єднання розірветься.

Спілкування про дії**29**використовується, коли запитувана мета займає тривалий час, тому необхідний зворотний зв’язок щодо прогресу. Це дуже схоже на послугу, де "цілі" та "результати" відповідають відповідно "запитам" та "відповідям". Крім того, додається "зворотний зв'язок", щоб періодично повідомляти клієнту про зворотний зв'язок, коли потрібні проміжні значення. Спосіб передачі повідомлення такий самий, як і асинхронна тема. Зворотній зв'язок передає асинхронне двонаправлене повідомлення між клієнтом дії, який встановлює мету дії, і сервером дії, який виконує дію, і надсилає відгук клієнту дії. Наприклад, як показано на малюнку 4-4, якщо клієнт ставить завдання прибирання будинку як мету для сервера, сервер інформує користувача про хід миття посуду, прання, прибирання тощо у формі зворотного зв'язку, і, нарешті, надсилає остаточне повідомлення клієнту як результат. На відміну від служби, ця дія часто використовується для командування складних завдань робота, таких як скасування переданої мети, поки триває операція.

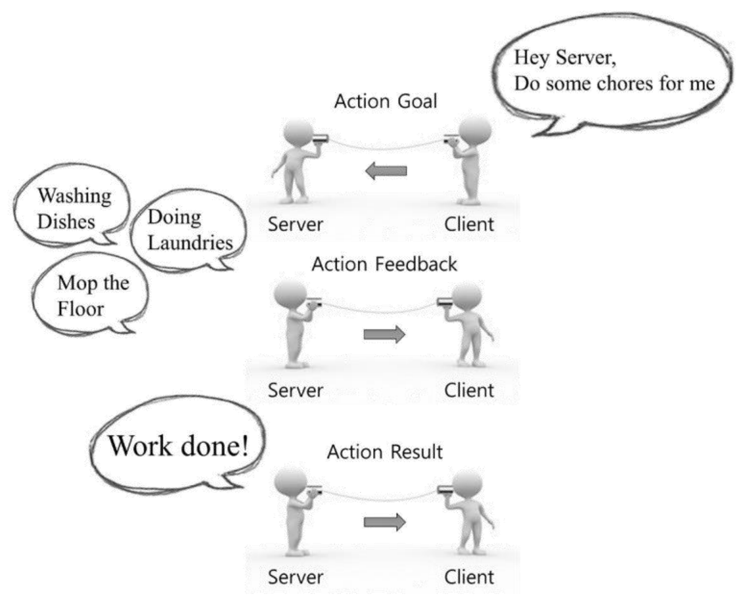


Рис. 21 Дія обміну повідомленнями

Видавець, передплатник, сервер обслуговування, клієнт служби, сервер дій та клієнт дій можуть бути реалізовані в окремих вузлах. Для обміну повідомленнями між цими вузлами спочатку слід встановити з'єднання за допомогою ведучого. Майстер діє як сервер імен, оскільки зберігає імена вузлів, тем, служб та дій, а також адресу URI, номер порту та параметри. Іншими словами, вузли реєструють власну інформацію у ведучому під час запуску та отримують відносну інформацію про інші вузли у ведучого. Потім кожен вузол безпосередньо підключається один до одного для здійснення обміну повідомленнями. Це показано на малюнку 21.

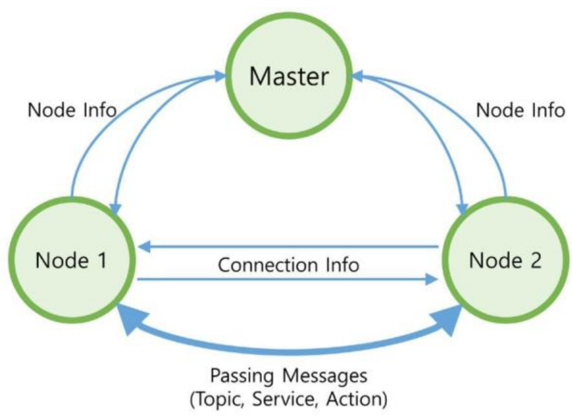


Рис. 22 Процес спілкування

Спілкування повідомленнями в основному поділяється на теми, послуги та дії. Параметри - це глобальні змінні, що використовуються у вузлах, і у більшому контексті їх також можна розглядати як комунікацію повідомлень. У програмах Windows * .ini-файл використовується для збереження конфігурацій як параметрів у ROS. Конфігурація встановлюється зі значеннями за замовчуванням і за потреби може бути прочитана або записана зовні. Зокрема, налаштовані значення можуть бути змінені в реальному часі ззовні за допомогою функції запису. Дуже корисно гнучко справлятися зі зміною середовища.

Хоча параметри не є строго методом комунікації повідомлень, я думаю, що вони належать до сфери комунікації повідомлень, оскільки використовують повідомлення. Наприклад, ви можете змінити параметри, щоб встановити USB-порт для підключення,

отримати значення корекції кольору камери та налаштувати максимальне та мінімальне значення швидкості та команд.

Майстер управляє інформацією про вузли, і кожен вузол підключається та взаємодіє з іншими вузлами за необхідності. Давайте дізнаємося про найважливішу послідовність комунікацій майстра, вузлів, тем, служб та повідомлень про дії.

Запуск Майстра

Майстер, який управляє інформацією про з'єднання в повідомленні між вузлами, є важливим елементом, який повинен бути запущений спочатку, щоб використовувати ROS. Майстер ROS запускається за допомогою команди 'roscore' і запускає сервер з XMLRPC. Майстер реєструє ім'я вузлів, тем, служб, дій, типів повідомлень, адрес URI та портів для з'єднань між вузлами та передає інформацію іншим вузлам за запитом.

```
$ roscore
```


Рис. 23 Запуск Майстра

Запуск абонентського вузла

Абонентські вузли запускаються за допомогою команд 'rosrun' або 'roslaunch'. Абонентський вузол реєструє своє ім’я вузла, ім’я теми, тип повідомлення, адресу URI та порт із ведучим пристроєм під час його роботи. Ведучий і вузол спілкуються за допомогою XMLRPC.

$ rosrun PACKAGE_NAME NODE_NAME

$ roslaunch PACKAGE_NAME LAUNCH_NAME

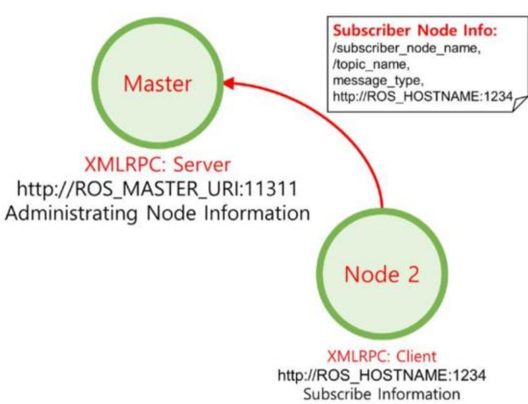


Рис. 24 Запуск абонентського вузла

Запуск вузла видавця

Вузли видавця, як і абонентські вузли, виконуються командами 'rosrun' або 'roslaunch'. Вузол видавця реєструє у вузлі назву вузла, назву теми, тип повідомлення, адресу URI та порт. Ведучий і вузол спілкуються за допомогою XMLRPC.

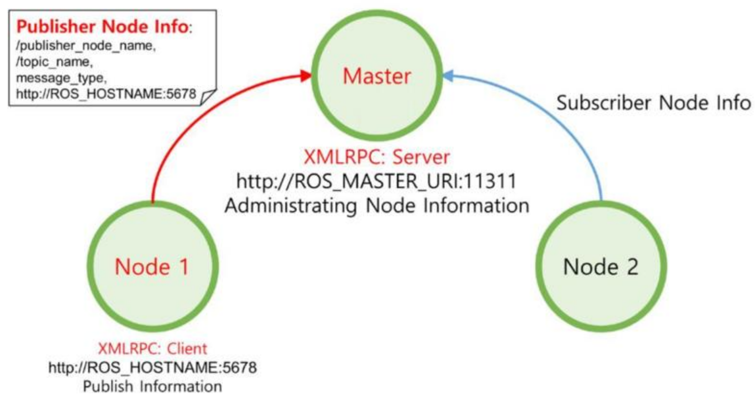


Рис. 25 Запуск вузла видавця

Надання інформації про видавця

Майстер розподіляє таку інформацію, як ім'я видавця, назва теми, тип повідомлення, адреса URI та номер порту видавця, передплатникам, які хочуть підключитися до вузла видавця. Ведучий і вузол взаємодіють за допомогою XMLRPC.

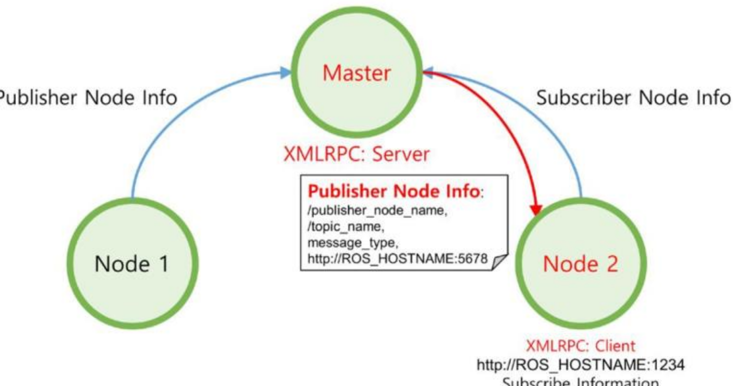


Рис. 26 Надання інформації про вузол для абонентського вузла

Запит на підключення від абонентського вузла

Абонентський вузол вимагає прямого підключення до вузла видавця на основі інформації про видавця, отриманої від ведучого. Під час процедури запиту вузол абонента передає на вузол видавця таку інформацію, як ім'я абонента, ім'я теми та тип повідомлення. Вузол видавця та вузол абонента спілкуються за допомогою

XMLRPC.

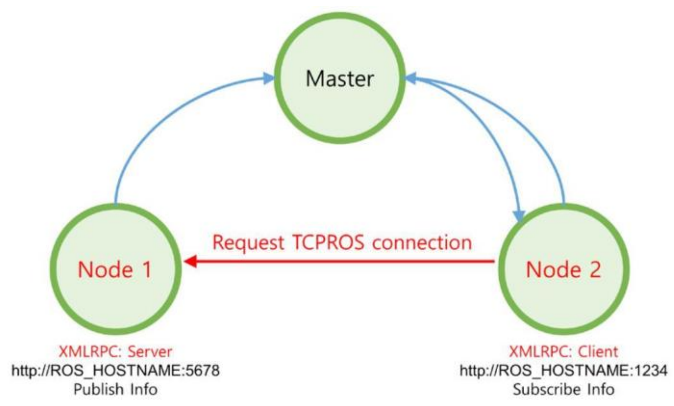


Рис. 27 Запит абонента на підключення до вузла

Відповідь на підключення від вузла видавця

Вузол видавця надсилає адресу URI та номер порту свого TCP-сервера у відповідь на запит на підключення від абонентського вузла.

Вузол видавця та абонентський вузол взаємодіють за допомогою XMLRPC.

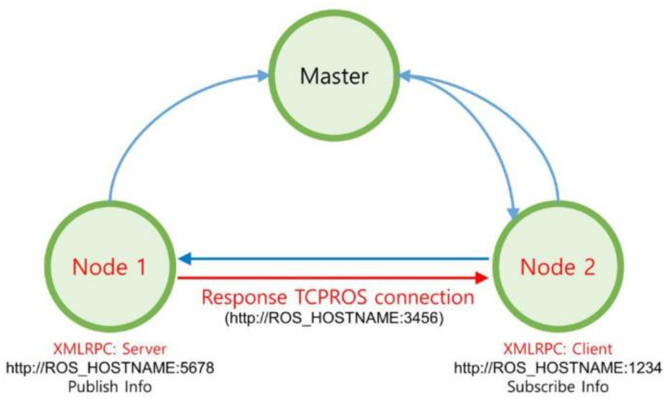


Рис. 28 Відповідь на підключення від вузла видавця

Підключення TCPROS

Абонентський вузол створює клієнт для вузла видавця за допомогою TCPROS і підключається до вузла видавця. На даний момент для зв'язку між вузлами використовується протокол TCP / IP, який називається TCPROS.

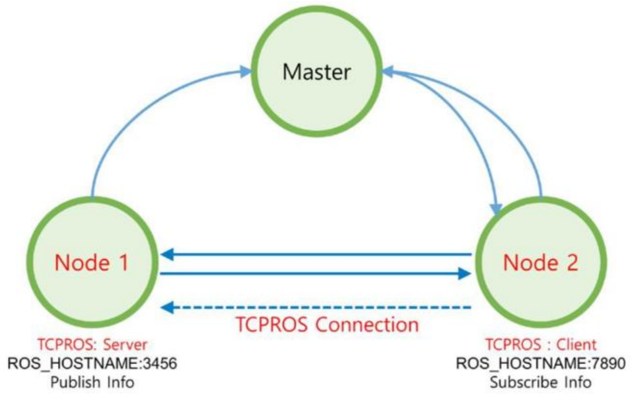


Рис. 29 TCP-з’єднання

Передача повідомлень

Вузол видавця передає попередньо визначене повідомлення абонентському вузлу. Зв'язок між вузлами використовує TCPROS.

Рис. 30 Передача повідомлення теми

Запит на обслуговування та відповідь

Розглянуті вище процедури відповідають повідомленням на тему. Тематичне спілкування публікує та передплачує повідомлення безперервно, якщо видавець або передплатник не припинено. Існує два типи послуг.

- Клієнт послуги: запитуйте послугу та отримайте відповідь
- Сервер послуг: Отримайте послугу, виконайте вказане завдання та поверніть відповідь

Зв'язок між сервером служби та клієнтом такий самий, як описане вище з'єднання TCPROS для видавця та абонента. На відміну від теми, служба розриває підключення після успішного запиту та відповіді. Якщо необхідний додатковий запит, процедуру підключення потрібно провести ще раз.

Рис. 31 Запит на обслуговування та відповідь

Ціль дії, результат, відгук

Дія може виглядати подібно до запиту та відповіді служби з додатковим повідомленням зворотного зв'язку, щоб забезпечити проміжний результат між запитом (метою) та відповіддю (результатом), але на практиці це скоріше схоже на тему. Насправді, якщо ви використовуєте команду 'rostopic' для переліку тем, у дії використовується п'ять тем, таких як мета, статус, скасування, результат та відгук. Зв'язок між сервером дій та клієнтом схожий на TCPROS-з'єднання видавця та передплатника, але використання дещо відрізняється. Наприклад, коли клієнт дії надсилає команду скасування або сервер надсилає значення результату, з'єднання буде розірвано.

Рис. 32 Дія повідомлення

Раніше ми тестували АФК з використанням "turtlesim". У цьому тесті було використано головний та два вузли, а тема '/ turtle1 / cmd_vel' була використана між двома вузлами для передачі поступальних та обертальних повідомлень віртуальному TurtleBot. Помічаючи це в перспективі з описаною вище концепцією АФК, її можна представити, як на рис. 33.

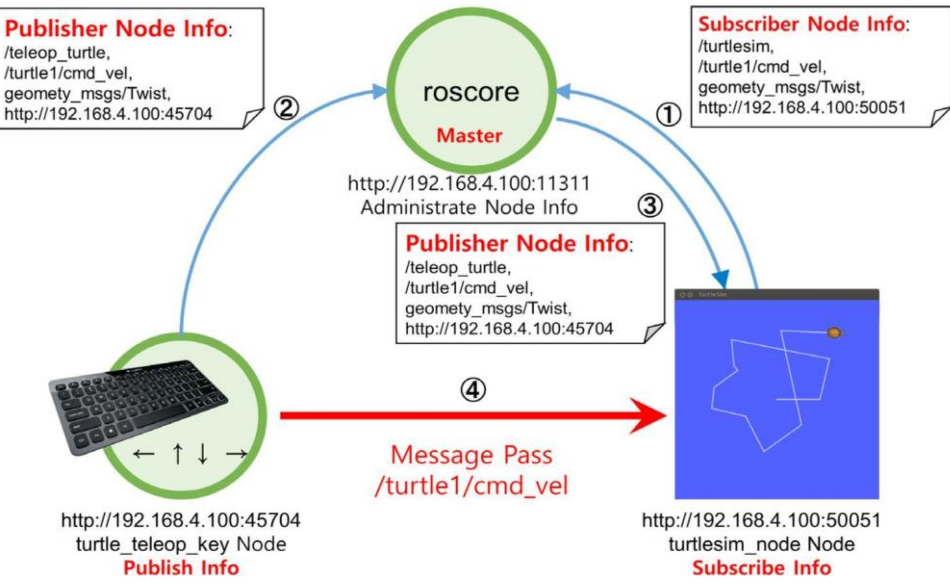


Рис. 33 Приклад обміну повідомленнями

4.3. Повідомлення

Повідомлення - це пакет даних, що використовується для обміну даними між вузлами. Теми, послуги та дії використовують повідомлення для спілкування. Повідомлення може включати основні типи даних, такі як ціле число, плаваюча крапка, булева, а також масиви повідомлень, такі як 'float32 [] діапазони', 'Point32 [10] точки'. Більше того, повідомлення може містити інші повідомлення, такі як 'geometry_msgs / PoseStamped'. Також до повідомлення може бути включений заголовок 'std_msgs / Header', який зазвичай використовується в ROS. Ці повідомлення можна описати як типи полів та назви полів, як показано нижче.

fieldtype1 fieldname1

fieldtype2 fieldname2

fieldtype3 fieldname3

Тип даних ROS, показаний у таблиці 4-2, може використовуватися в польовому типі. Наступний приклад є найпростішою формою повідомлення, і ви можете використовувати масив для типу поля, як показано в таблиці 4-3. Також часто використовуються вбудовані повідомлення у повідомлення.

Таблиця ? Основні типи даних для повідомлень у ROS, методи серіалізації, відповідні C ++ та Python типи даних

Тип даних ROS	Серіалізація	Тип даних C ++	Тип даних Python
bool	непідписаний 8-бітний int	uint8_t	bool

int8	підписаний 8-бітний int	int8_t	інт
uint8	непідписаний 8-бітний int	uint8_t	інт
int16	підписаний 16-розрядний int	int16_t	інт
uint16	непідписаний 16-бітний int	uint16_t	інт
int32	підписаний 32-розрядний int	int32_t	інт
uint32	непідписаний 32-розрядний int	uint32_t	інт
int64	підписаний 64-розрядний int	int64_t	довго
uint64	непідписаний 64-розрядний int	uint64_t	довго
float32	32-розрядна плаваюча система IEEE	плавати	плавати
float64	64-розрядна плаваюча система IEEE	подвійний	плавати
рядок	рядок ascii	std :: рядок	вул
час	secs / nsecs без підпису 32-бітові ints	ros :: Час	час

Таблиця ? Як використовувати типи даних ROS-повідомлень як масив, що відповідає типам даних C ++ та Python

Тип даних ROS	Серіалізація	Тип даних C ++	Тип даних Python
фіксованої довжини	відсутність зайвої серіалізації	boost :: array, std :: vector	кортеж
змінної довжини	префікс довжини uint32	std :: вектор	кортеж

uint8 []	префікс довжини uint32	std :: вектор	байт
bool []	префікс довжини uint32	std :: vector <uint8_t>	список bool

Заголовок (std_msgs / Header), який зазвичай використовується в ROS, також може використовуватися як повідомлення. Файл Header.msg у файлі std_msgs**31** містить ідентифікатор послідовності, позначку часу та ідентифікатор кадру та використовує їх для перевірки повідомлення або вимірювання часу.

std_msgs/Header.msg

```
# Sequence ID: Messages are sequentially incremented by 1.
uint32 seq
# Timestamp: Has two child attributes, the stamp.sec for second and the stamp.nsec for
nanosecond.
time stamp
# Stores the Frame ID
string frame_id
```

Далі показано, як насправді використовувати повідомлення в програмі ROS. Наприклад, у випадку вузла 'teleop_turtle_key' пакету turtlesim, який ми перевірили в розділі 3, швидкість поступальної передачі (метр / сек) та швидкість обертання (радіан / сек) надсилаються як повідомлення вузолу черепахи відповідно до до клавіш управління (←, →, ↑, ↓), введених з клавіатури. TurtleBot рухається по екрану, використовуючи отримані значення швидкості. Повідомлення, яке використовується в цей час, - це "поворот"

повідомлення у 'geometry_msgs'.

```
Vector3  linear
Vector3  angular
```

У наведеній вище структурі повідомлень значення «лінійне» та «кутове» оголошено як тип Vector3. Це подібна форма до вкладеного повідомлення, оскільки Vector3 є типом повідомлення в "geometry_ msgs". Вектор3 містить наступні дані.

float64 x

float64 p

float64 z

Іншими словами, шість тем, опублікованих із вузла 'teleop_turtle_key', - linear.x, linear.y, linear.z, angular.x, angular.y та angular.z. Все це тип float64, який є одним з основних типів даних, описаних у ROS. За допомогою цих даних клавіші зі стрілками на клавіатурі можуть бути перетворені на повідомлення про швидкість поступального перетворення (метр / с) та швидкість обертання (радіан / с), щоб можна було керувати TurtleBot.

Тема, послуга та дія, описані в попередньому розділі, використовують повідомлення. Хоча вони подібні за формою та концепцією, вони поділяються на три типи відповідно до їх використання. Це буде обговорено більш докладно в наступному розділі.

Файл 'msg' - це файл повідомлення, що використовується темами, із розширенням файлу '* .msg'. "Поворот" повідомлення в "geometry_msgs", описаному вище, є прикладом повідомлення. Такий файл повідомлень складається з типів полів та назв полів.

```
geometry_msgs/Twist.msg
Vector3  linear
Vector3  angular
```

Файл 'srv' - це файл повідомлення, що використовується службами, із розширенням файлу '*.srv '. Наприклад, SetCameraInfo36повідомлення в описаному вище 'sensor_msgs' є типовим файлом srv. Головна відмінність від файлу msg полягає в тому, що серія з трьох дефісів (---) служить роздільником; верхнє повідомлення - повідомлення про запит на обслуговування, а нижнє - повідомлення про відповідь на послугу.

```
sensor_msgs/SetCameraInfo.srv
sensor_msgs/CameraInfo camera_info
---
bool success
string status_message
```

Файл повідомлення про дію - це файл повідомлення, який використовується діями, із розширенням файлу '* .action'. На відміну від msg та srv, це відносно незвичайний файл повідомлення, тому

немає типового прикладу файлу повідомлення, але його можна використовувати, як показано в наступному прикладі. Основна відмінність від файлів msg та srv полягає в тому, що серія з трьох дефісів (---) використовується в двох місцях як роздільники, причому перше - целеве повідомлення, друге - повідомлення результату, а третє - повідомлення зворотного зв'язку . Найбільша відмінність файлу дій - це функція повідомлення про зворотній зв'язок.

Повідомлення цілі та повідомлення результату файлу дій можна порівняти із запитом та повідомленням відповіді згаданого вище файлу srv, але додаткове повідомлення відгуку файлу дії використовується для надсилання зворотного зв'язку під час виконання призначеного процесу. Як описано в наступному прикладі, коли початкова позиція 'start_pose' і цільова позиція 'goal_pose' робота передаються як значення запиту, робот переміщається в отриману цільову позицію і повертає 'result_pose'. Поки робот рухається до цільової позиції, повідомлення 'процент_завершеного' періодично передає значення зворотного зв'язку, що відображають прогрес у вигляді відсотка досягнутої точки.

geometry_msgs / PoseStamped start_pose

geometry_msgs / PoseStamped goal_pose

geometry_msgs / PoseStamped result_pose

float32 процента_завершено

4.4. Ім'я

Основною концепцією ROS є абстрактний тип даних, який називається "граф". Цей графік показує взаємозв'язок між кожним вузлом та зв'язок повідомлень (даних), надісланих та отриманих зі стрілками. Для цього повідомлення та параметри, що використовуються у вузлах, темах та службах у ROS, мають унікальні імена. Давайте детальніше розглянемо назви тем. Назва теми поділяється на відносний метод, глобальний метод і приватний метод, як показано в таблиці 4-4.

Тема зазвичай оголошується, як показано в наступному коді. Це буде детальніше висвітлено у розділі 7. Тут давайте змінимо назву теми, щоб зрозуміти, як використовувати імена.

```
int main(int argc, char **argv)             // Node Main Function
{
  ros::init(argc, argv, "node1");           // Node Name Initialization
  ros::NodeHandle nh;                       // Node Handle Declaration
  // Publisher Declaration, Topic Name = bar
  ros::Publisher node1_pub = nh.advertise<std_msg::Int32>("bar", 10);
```

У наведеному вище прикладі ім'я вузла - '/ node1'. Якщо видавець оголошено як "бар" без будь-яких символів, тема матиме відносну назву "/ bar". Навіть якщо символ косої риси (/) використовується для глобального оголошення, назва теми все одно буде '/ bar'.

ros :: Видавець node1_pub = nh.advertise <std_msg :: Int32> ("/ bar", 10);

Однак якщо ви оголосите ім'я приватним, використовуючи символ тильди (~), ім'я теми стане '/ node1 / bar'.

ros :: Видавець node1_pub = nh.advertise <std_msg :: Int32> ("~ бар", 10);

Оголошення імені може змінюватися, як показано в таблиці 4-4. '/ Wg' означає зміну простору імен. Про це докладніше йдеться в наступному розділі.

Таблиця ? Naming Rule

Node	Relative (Default)	Global	Private
/node1	bar → /bar	/bar → /bar	~bar → /node1/bar
/wg/node2	bar → /wg/bar	/bar → /bar	~bar → /wg/node2/bar
/wg/node3	foo/bar → /wg/foo/bar	/foo/bar → /foo/bar	~foo/bar → /wg/node3/foo/bar

Як можна запустити дві камери? Просте виконання відповідного вузла двічі призведе до завершення раніше виконаного вузла через те, що в ROS має бути унікальне ім'я. Однак досягнення двох камер не вимагає запуску окремої програми або зміни вихідного коду. Просто змініть ім'я вузла під час його запуску, використовуючи простори імен або перепризначення.

Щоб допомогти вам зрозуміти, припустимо, у вас є віртуальний 'camera_package'. Припустимо, що вузол камери виконується, коли виконується 'camera_node' з 'camera_package', спосіб його запуску такий.

$ rosrun camera_package camera_node

Якщо 'camera_node' передає дані зображення камери через тему зображення, цю тему зображення можна отримати з 'rqt_image_view' наступним чином

$ rosrun rqt_image_view rqt_image_view

Тепер давайте змінимо значення теми цих вузлів шляхом перепризначення. Наступна команда змінить назву теми на '/ front / image'. У наведеній нижче команді "зображення" - це назва теми "camera_node", а в наведеному нижче прикладі показано, як змінити назву теми, встановивши параметри в командах виконання.

$ rosrun camera_package camera_node image: = спереду / зображення

$ rosrun rqt_image_view rqt_image_view зображення: = спереду / зображення

Наприклад, якщо є три камери, такі як фронтальна, ліва та права, коли кілька вузлів виконуються під тим самим іменем, будуть конфліктні імена, і тому раніше виконаний вузол припиняється. Отже, вузли з однаковим іменем можуть бути виконані наступним чином. Нижче за варіантом імені слідують послідовні підкреслення (__). Такі параметри, як '__ns', '__name', '__log', '__ip', '__hostname' та '__master' - це спеціальні параметри, що використовуються під час запуску вузла. Крім того, перед назвою теми ставиться одинарне підкреслення (_), якщо воно використовується як приватне.

$ rosrun camera_package camera_node __name: = front _device: = / dev / video0 $ rosrun camera_package camera_node __name: = left _device: = / dev / video1 $ rosrun camera_package camera_node __name: = right _device: = / dev / video2 $ rosrun_view_qtview

Наступний приклад зв'яже вузли та теми в єдиний простір імен. Це гарантує, що всі вузли та теми згруповані в єдиний простір імен, і всі імена відповідно змінені.

$ rosrun camera_package camera_node __ns: = назад

$ rosrun rqt_imgae_view rqt_imgae_view __ns: = назад

Ми бачили різні способи використання імен. Імена підтримують можливість безперебійного підключення системи ROS в цілому. У цьому розділі ми дізналися, як змінити значення імені за допомогою команди вузла 'rosrun'. Подібним чином, використовуючи 'roslaunch', можна виконати ці параметри відразу. Це буде обговорено більш докладно в розділі 7 з прикладами.

4.5. Перетворення координат (TF)

Описуючи позу руки робота, як показано на малюнку 4-17, це можна описати як відносне перетворення координат кожного суглоба. Наприклад, рука робота-гуманоїда підключена до зап'ястя, зап'ястя - до ліктя, а лікоть - до плеча. Крім того, лікоть може відображатися по відношенню до поз суглобів ніг, коли робот йде і рухається. Нарешті, координата ліктя корелює з центром ніг робота. І навпаки, коли робот йде, координати рук робота рухаються відповідно до відносного перетворення координат відповідних корельованих суглобів. Крім того, якщо робот намагається зловити об'єкт, початок роботи буде відносно розміщений на конкретній карті, а об'єкт також буде розміщений на карті. Робот може обчислити положення об'єкта щодо

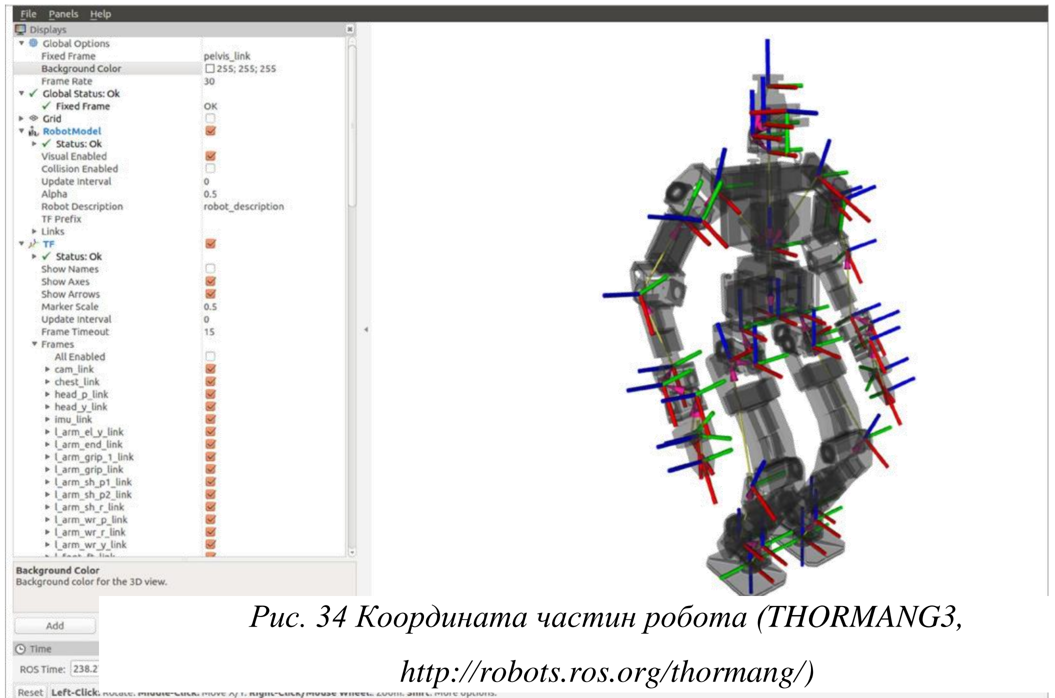


Рис. 34 Координата частин робота (THORMANG3, http://robots.ros.org/thormang/)

його положення на карті, щоб зловити об’єкт. У програмуванні робототехніки.

У ROS трансформація координат TF є однією з найбільш корисних концепцій при описі частин робота, а також перешкод та об'єктів. Позу можна описати як поєднання позицій та орієнтацій. Тут положення виражається трьома векторами x, y та z, а орієнтація - чотирма векторами x, y, z та w, які називаються кватерніонами. Кватерніон не є інтуїтивно зрозумілим, оскільки вони не описують обертання трьох осей (x, y, z), таких як кути крену, кроку та повороту, які часто використовуються. Однак форма кватерніону вільна від блокування карданного з’єднання або проблем зі швидкістю, які існують у методі Ейлера з нахилом, нахилом та похитуванням. Тому в робототехніці кращий тип кватерніона, і ROS також використовує кватерніон з цієї причини. Звичайно, для зручності передбачені функції для перетворення значень Ейлера в кватерніони.

TF використовує структуру повідомлень**43**показано нижче. Заголовок використовується для запису перетвореного часу, а повідомлення з назвою 'child_frame_id' використовується для вказівки дочірніх координат. Відносне розташування та орієнтація описані у такій формі даних: transform.translation.x / transform.translation.y / transform.translation.z / transform.rotation.x / transform.rotation.y / transform.rotation.z / transform .rotation.w

```
geometry_msgs/TransformStamped.msg

Header header
string child_frame_id
Transform transform
```

Дано короткий опис ФТ. Більш детальні приклади TF будуть пояснені в моделюючій частині мобільних роботів (розділи 10, 11) та маніпуляторів (глава 13).

4.6. Клієнтська бібліотека

Мови програмування дуже різноманітні. C ++ часто використовується для контролю продуктивності та апаратного забезпечення, а Python або Ruby - для підвищення продуктивності. LISP широко використовується в галузі штучного інтелекту, MATLAB для наукового програмного забезпечення, такого як чисельний аналіз, Java для Android, а також багатьох інших, таких як C #, Go, Haskell, Node.js, Lua, R, EusLisp, Julia тощо Не можна сказати, яка з цих мов важливіша за інші. Розробник підбере найбільш підходящу мову залежно від характеру роботи. Тому ROS, який підтримує роботів різного призначення, дозволяє розробнику вибрати найбільш відповідну мову відповідно до цілі. Вузли можна писати різними мовами, а інформація обмінюється за допомогою комунікації повідомлень між вузлами. Програмним модулем, що дозволяє писати вузли різними мовами, є клієнтська бібліотека. Деякі з найбільш часто використовуваних бібліотек - "roscpp" для мови C ++, "rospy" для мови Python, "roslisp" для LISP та "rosjava" для Java. Крім того, існують

"roscs, roseus, rosgo, roshask, rosnodejs, RobotOS.jl, roslua, PhaROS, rosR, rosruby" та "Unreal-Ros-Plugin". Кожна бібліотека клієнта все ще знаходиться в стадії розробки для різноманітності мов ROS.

Ця книга буде зосереджена на "roscpp" для мови C ++. Навіть якщо використовуються різні мови, поняття однакові з різним синтаксисом програмування. Тому розробники можуть використовувати мову, найбільш підходящу для цієї мети, посилаючись на вікі кожної бібліотеки клієнта.

4.7. Зв'язок між гетерогенними пристроями

Як описано в частині мета-операційної системи глави 2, ROS підтримує зв'язок між різними пристроями за допомогою зв'язку, повідомлення, імені, функції перетворення та клієнтської бібліотеки (див. Рисунок 4-18). ROS можна запускати незалежно від типу операційної системи, на якій він встановлений, і незалежно від використовуваної мови програмування. Поки ROS встановлений і кожен вузол правильно розроблений, зв’язок між вузлами дуже простий. Наприклад, статус робота може контролюватися на MacOS, навіть якщо на роботі встановлено Ubuntu, дистрибутив Linux. У той же час користувач може керувати роботом із програми на базі Android. Приклад цього викладено в описі USB-камери в главі 8, на прикладі передачі відеопотоку між двома різними ПК.

Навіть для вбудованої системи мікроконтролера, яка не може встановити ROS, зв’язок повідомлень можливий, якщо вона

розроблена для можливості надсилання та отримання ROS-повідомлень. Це детальніше обговорюється в розділі Вбудовані системи глави 9.

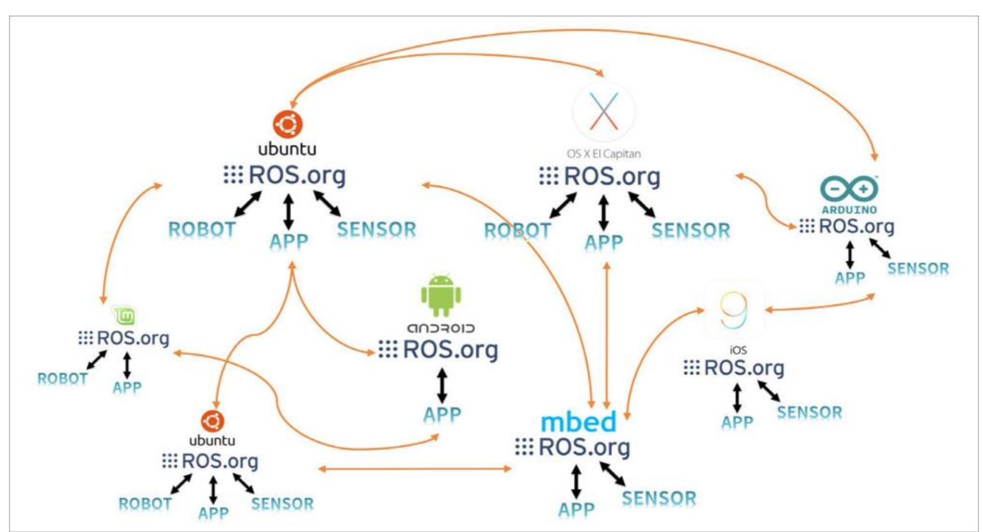


Рис. 35 Зв'язок між гетерогенними приладами

4.8. Файлова система

Давайте дізнаємося про конфігурацію файлів ROS. У ROS пакет є основною одиницею для конфігурації програмного забезпечення, а додатки ROS розробляються як пакет. Пакет містить один або кілька вузлів, які є найменшими процесорами виконання в ROS, або включає файли конфігурації для запуску інших вузлів. Станом на липень 2017 року ROS Indigo має близько 2500 пакетів, а ROS Kinetic - близько 1600 офіційних пакетів. Користувачами також розроблено та випущено близько 5000 пакетів, хоча можуть бути деякі надмірності. Цими пакетами також управляють як набір пакетів, що

являє собою набір пакетів із загальним призначенням, який називається метапакетом. Наприклад, навігаційний метапакет складається з 10 пакетів, включаючи AMCL, DWA, EKF та map_server та багато іншого. Кожен пакет містить 'package.xml', який є XML-файлом, що містить інформацію про пакет, включаючи його назву, автора, ліцензію та залежні пакети. Крім того, Catkin, яка є системою збірки ROS, використовує CMake, а 'CMakeLists.txt' у папці пакета описує середовище збірки. Крім того, пакет складається з вихідного коду для вузлів та файлів повідомлень для обміну повідомленнями між вузлами.

Файлова система ROS ділиться на папки встановлення та папки робочої області. Якщо встановлена настільна версія ROS, папка інсталяції створюється в папці / opt, а основні утиліти, включаючи roscore, rqt, RViz, бібліотеку, пов’язану з роботом, моделювання та навігацію встановлюються всередині папки. Користувач рідко потребує модифікації файлів у цій області. Однак для того, щоб змінити пакет, який офіційно розповсюджується у вигляді двійкового файлу, перевірте сховище, що містить вихідне джерело, та скопіюйте джерело у робочу область, використовуючи 'git clone [REPOSITORY_ ADDRESS]' у '~ / catkin_ws / src' замість того, щоб використовувати команду встановлення пакунка 'sudo apt-get install ros-kinetic-xxx' ,.

Робочу область користувача можна створювати де завгодно користувач, але давайте створимо її в папці користувача користувача

Linux ~ ~ / catkin_ws / ('~ /' - це папка '/ home / user /' в Linux). Далі, давайте дізнаємося про папку встановлення ROS та робочу область.

ROS встановлюється в папці '/ opt / ros / [VERSION_NAME]'. Наприклад, якщо ви встановили

Версія ROS Kinetic Kame, шлях встановлення ROS:

- Шлях до папки встановлення ROS '/ opt / ros / kinetic'

Конфігурація файлу

Коли інстальовано ROS, папка '/ opt / ros / kinetic' складається з кошика тощо, включаючи lib, спільну папку та деякі файли конфігурації, як показано на малюнку 4-19.

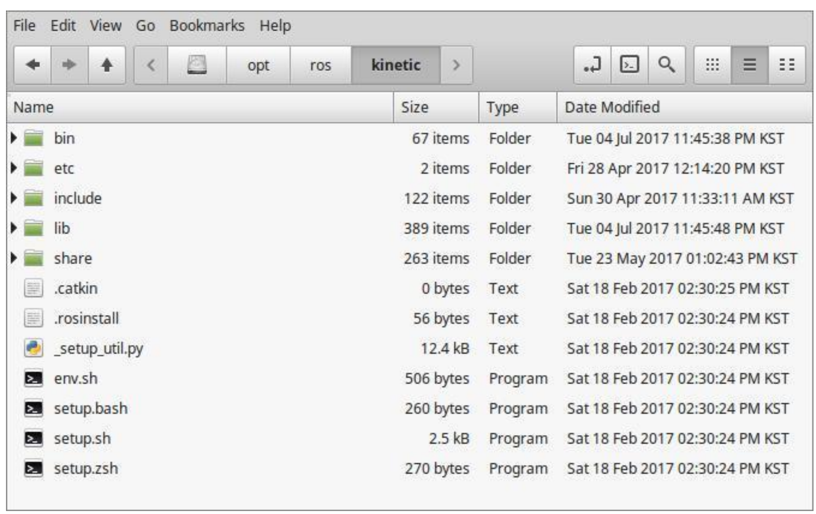


Рис. 36 Конфігурація файлу ROS

Описи файлів та папок

Папка ROS містить пакети та програми ROS, обрані під час встановлення ROS.

Деталі такі.

- / смітник Виконувані двійкові файли
- / тощо Файли конфігурації, пов'язані з ROS та Catkin
- /включати Файли заголовка
- / lib Бібліотечні файли
- / поділитися Пакети ROS
- env. * Файли конфігурації середовища
- налаштування. * Файли конфігурації середовища

Ви можете створити робочу область, де завгодно, але в цій книзі для зручності використовуйте '~ / catkin_ws /', яка знаходиться в папці користувача Linux. Повний шлях до вибраної папки робочої області будебути '/ home / username / catkin_ws'. Наприклад, якщо ім'я користувача "oroca", а ім'я папки catkin - "catkin_ws", шлях:

- *Шлях робочої області: / home / oroca / catkin_ws /*

Конфігурація файлу

Як показано на рис. 37, у папці '/ home / username /' є папка, яка називається 'catkin_ws', і вона складається з папок build, devel та src. Зверніть увагу, що папки build і devel створюються після catkin_make.

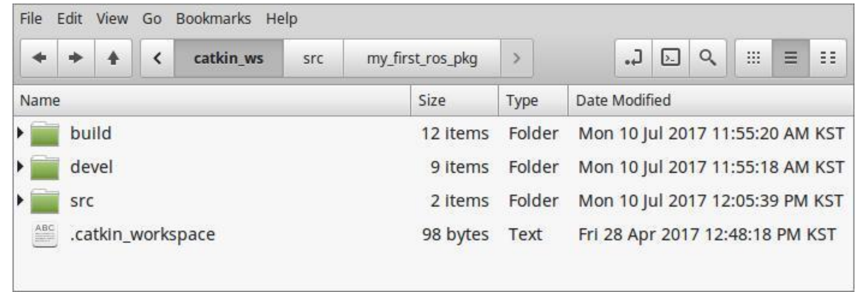


Рис. 37 Конфігурація файлу робочої області catkin

Детальна конфігурація файлу

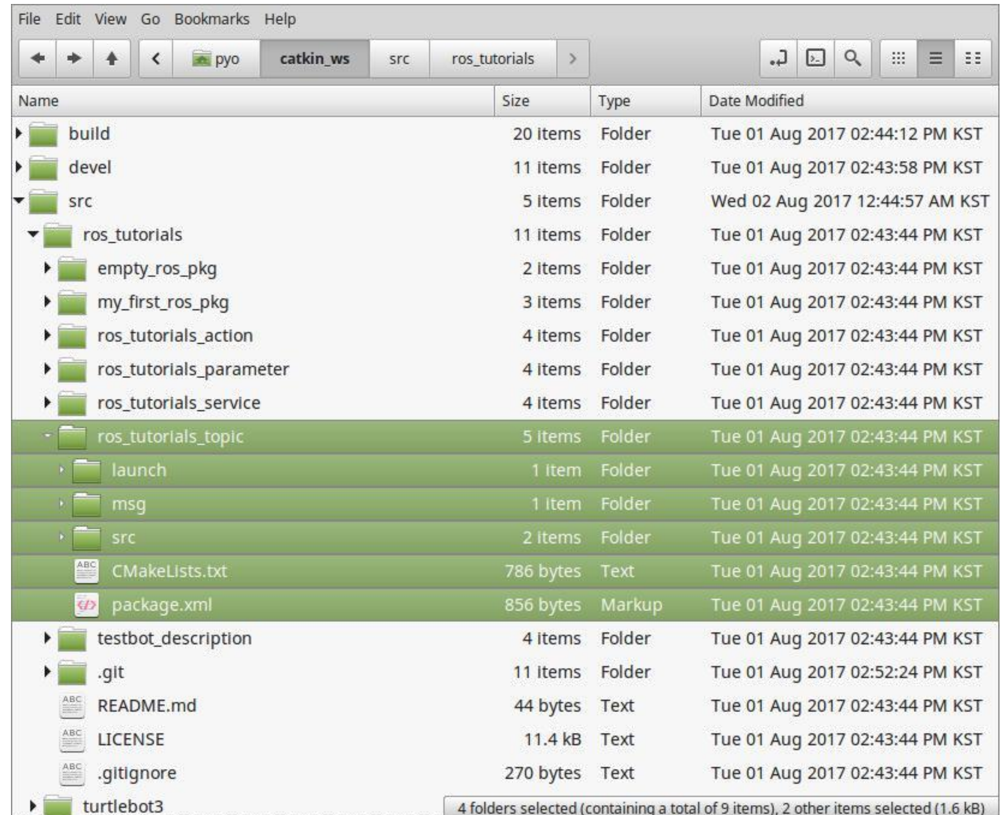
Робоча область - це простір, який зберігає та створює створені користувачем пакети та пакети, опубліковані іншими розробниками. У цій папці користувачі виконують більшість операцій, пов’язаних із ROS. Деталі такі.

Таблиця ?

- / побудувати	Створення пов'язаних файлів msg, srv
- / розробляти	Файли заголовків та Бібліотека користувацьких пакетів, Файли виконання файлів
- / src	Користувацькі пакети

Пакет користувача

Папка '~ / catkin_ws / src' - це місце для вихідного коду користувача. У цій папці ви можете зберігати та створювати власні пакети ROS або пакети, розроблені іншими розробниками. Система



побудови ROS буде детально описана в наступному розділі. На рис. 38 нижче показано стан після заповнення пакета 'ros_tutorials_topic'. Він описує папки та файли, які зазвичай використовуються, хоча конфігурація може змінюватися залежно від цілі пакету.

Рис. 38 Конфігурація файлу користувацького пакета

- **/включати** Файли заголовка
- **/ запуск** Файли запуску, що використовуються разом із запуском
- **/ вузол** Сценарій для rospy
- **/ повідомлення** Файли повідомлень
- **/ src** Файли вихідного коду
- **/ srv** Сервісні файли
- **CMakeLists.txt** Побудуйте файл конфігурації
- **package.xml** Файл конфігурації пакета

4.9. Система побудови

Система збірки ROS за замовчуванням використовує CMake (Cross Platform Make), а середовище збірки описується у файлі

CMakeLists.txt у папці пакета. ROS забезпечує специфічну для ROS систему побудови котячих з модифікованим CMake.

Причиною використання CMake на ROS є надання дозволу на створення пакета ROS на декількох платформах. На відміну від Make, який покладається лише на системи на базі Unix, CMake підтримує Windows, а також Unix на базі Linux, BSD та OS X. Він також підтримує Microsoft Visual Studio і може бути легко застосований до розробки Qt. Крім того, система збірки Catkin спрощує використання збірок ROS, управління пакетами та залежностей між пакетами.

Команда для створення пакета ROS така.

$ catkin_create_pkg [PACKAGE_NAME] [DEPENDENT_PACKAGE_1] [DEPENDENT_PACKAGE_N]

Команда 'catkin_create_pkg' створює папку пакета, яка містить файли 'CMakeLists.txt' та 'package.xml', необхідні для системи складання тортів. Давайте створимо простий пакет, який допоможе вам зрозуміти. Спочатку відкрийте нове вікно терміналу (Ctrl + Alt + t) і запустіть наступну команду, щоб перейти до папки робочої області.

$ cd ~ / catkin_ws / src

Ім'я пакета, яке потрібно створити, - 'my_first_ros_pkg'. Назви пакетів у ROS мають бути малими і не повинні містити пробілів. Настанова щодо іменування також використовує підкреслення (_) між кожним словом замість тире (-) або пробілу. Див. Відповідні сторінки для керівництва стилем кодування44 45 і конвенції про іменування в

ROS. Тепер давайте створимо пакет із назвою 'my_first_ros_pkg' за допомогою наступну команду:

$ catkin_create_pkg my_first_ros_pkg std_msgs roscpp

'std_msgs' та 'roscpp' були додані як додаткові залежні пакети в попередній команді. Це означає, що 'std_msgs', що є стандартним пакетом повідомлень ROS, і 'roscpp', що є клієнтською бібліотекою, необхідною для використання C / C ++ в ROS, повинні бути встановлені до створення пакету. Ці залежні налаштування пакета можна вказати під час створення пакету, але їх можна створити безпосередньо в 'package.xml'.

Після створення пакета в папці ~ ~ / catkin_ws / src буде створена папка пакета «my_first_ros_pkg», а також внутрішня папка за замовчуванням, яку повинен мати пакет ROS, а також «CMakeLists.txt» та «package.xml файли. Вміст можна перевірити за допомогою команди 'ls', як показано нижче, а внутрішню частину пакета можна перевірити за допомогою графічного інтерфейсу інструменту Nautilus, який діє як Провідник вікон.

$ cd my_first_ros_pkg

$ ls

включати → Включити папку

src → Папка вихідного коду

CMakeLists.txt → Створіть файл конфігурації

package.xml → Файл конфігурації пакета

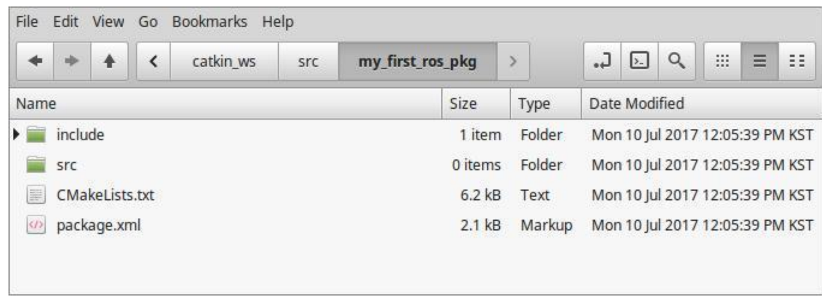


Рис. 39 Автоматично створювані файли та папки під час створення нового пакету

'Package.xml', який є одним із важливих файлів конфігурації ROS, - це XML-файл, що містить інформацію про пакет, включаючи ім'я пакета, автора, ліцензію та залежні пакети. Оригінальний файл без будь-яких змін показано нижче.

package.xml

```
<?xml version="1.0"?>
<package>
  <name>my_first_ros_pkg</name>
  <version>0.0.0</version>
  <description>The my_first_ros_pkg package</description>
```

<? xml>	Цей тег вказує на те, що вміст документа дотримується
	Версія XML 1.0.
<пакет>	Цей тег поєднується з тегом </package> для позначення конфігурації
	частина конфігурації пакета ROS.
<ім'я>	Цей тег вказує назву пакета. Введена назва пакета
	при створенні використовується пакет. Назва пакету може бути
	змінено розробником.
<версія>	Цей тег вказує версію пакета. Розробник може призначити
	версія пакету.
<опис>	Короткий опис упаковки. Зазвичай 2-3 речення.
<maintainer>	Ім'я та електронна адреса адміністратора пакета.
<ліцензія>	Цей тег позначає ліцензію, таку як BSD, MIT, Apache, GPLv3,
	LGPLv3.
<url>	Цей тег вказує адресу веб-сторінки, що описує пакет, або
	управління помилками, сховище тощо. Залежно від типу, ви можете
	призначити його веб-сайтом, програмою пошуку помилок або сховищем.
<author>	Ім'я та електронна адреса розробника, який брав участь у
	розробка пакету. Якщо брали участь кілька розробників, додайте
	кілька тегів <author> до наступних рядків.
<buildtool_depend>	Описує залежності системи побудови. Оскільки ми використовуємо

<build_depend>	
<run_depend>	
<test_depend>	
<export>	
<metapackage>	

Я змінив файл конфігурації пакета (package.xml) наступним чином. Давайте модифікуємо це у вашому власному середовищі. Якщо ви незнайомі з ним, ви можете використовувати файл нижче, як є:

package.xml

<? xml version = "1.0"?>

<пакет>

<name> my_first_ros_pkg </name>

<version> 0.0.1 </version>

<description> Пакет my_first_ros_pkg </description> <license> Apache License 2.0 </license>

<author email = " pyo@robotis.com "> Yoonseok Pyo </author> < maintenanceer email = " pyo@robotis.com "> Yoonseok Pyo </maintainer>

<url type = "bugtracker"> https://github.com/ROBOTIS-GIT/ros_tutorials/issues </url>

<url type = "repository"> https://github.com/ROBOTIS-GIT/ros_tutorials.git </url>

<url type = "website"> http://www.robotis.com </url>

<buildtool_depend> catkin </buildtool_depend>

<build_depend> std_msgs </build_depend>

<build_depend> roscpp </build_depend>

<run_depend> std_msgs </run_depend>

<run_depend> roscpp </run_depend>

<export> </export>

</package>

Catkin, система збірки для ROS, використовує CMake та описує середовище збірки у 'CMakeLists.txt' у папці пакунків. Він налаштовує створення виконуваного файлу, побудову пріоритету пакету залежностей, створення посилань тощо. Оригінальний файл без будь-яких змін показано нижче.

cmake_minimum_required (ВЕРСІЯ 2.8.3)

проект (my_first_ros_pkg)

Знайдіть макроси і бібліотеки

якщо КОМПОНЕНТИ перелічують find_package (catkin ПОТРІБНІ КОМПОНЕНТИ xyz) також використовуйте інші пакунки для котячих злаків

find_package (catkin ПОТРІБНІ КОМПОНЕНТИ

roscpp

std_msgs

Системні залежності знаходяться за умовами CMake

find_package (система Boost ПОТРІБНИХ КОМПОНЕНТІВ)

Розкоментуйте це, якщо в пакунку є файл setup.py. Цей макрос забезпечує встановлення модулів та глобальних сценаріїв, декларованих в них.

Див.
Http://ros.org/doc/api/catkin/html/user_guide/setup_dot_py.html

 # catkin_python_setup ()

Декларувати ROS-повідомлення, послуги та дії ##

Декларувати та створювати повідомлення, послуги чи дії зсередини пакет, виконайте такі дії:

Нехай MSG_DEP_SET є набором пакетів, типи повідомлень яких ви використовуєте ваші повідомлення / послуги / дії (наприклад, std_msgs, actionlib_msgs, ...).

* У файлі package.xml:

додати тег build_depend для "message_generation"

* додати тег build_depend та run_depend для кожного пакета в MSG_DEP_SET

* Якщо MSG_DEP_SET не порожній, введено таку залежність але тим не менше може бути оголошено:

* додати тег run_depend для "time_runtime"

* У цьому файлі (CMakeLists.txt):

* додати "message_generation" та кожен пакет у MSG_DEP_SET до find_package (catkin ПОТРІБНІ КОМПОНЕНТИ ...)

* додати "message_runtime" та кожен пакет у MSG_DEP_SET до

catkin_package (CATKIN_DEPENDS ...)

* за потреби розкоментуйте розділи додавання _ * _ файлів нижче

і перелічіть усі файли .msg / .srv / .action, які потрібно обробити

* розкоментуйте наведений нижче запис create_messages

* додати кожен пакет у MSG_DEP_SET, щоб згенерувати повідомлення (ЗАЛЕЖНОСТІ ...)

Створюйте повідомлення в папці 'msg'

add_message_files (

ФАЙЛИ

Message1.msg

Message2.msg

Створіть служби в папці 'srv'

add_service_files (

ФАЙЛИ

Service1.srv

Сервіс2.srv

Згенеруйте дії у папці 'action'

add_action_files (

ФАЙЛИ

Action1.action

Action2.action

Створюйте додані повідомлення та служби з будь-якими переліченими тут залежностями

create_messages (

ЗАЛЕЖНОСТІ

std_msgs

Оголосити параметри динамічного переналаштування ROS ##

Щоб оголосити та побудувати параметри динамічного перенастроювання всередині цього пакет, виконайте такі дії:

* У файлі package.xml:

* додати тег build_depend та run_depend для "dynamic_reconfigure"

* У цьому файлі (CMakeLists.txt):

* додати "dynamic_reconfigure" в

find_package (catkin ПОТРІБНІ КОМПОНЕНТИ ...)

* розкоментуйте розділ "generated_dynamic_reconfigure_options" нижче та перелічіть кожен файл .cfg, який буде оброблено

Створіть параметри динамічного переналаштування в папці 'cfg'

generated_dynamic_reconfigure_options (

cfg / DynReconf1.cfg

cfg / DynReconf2.cfg

специфічна конфігурація catkin ##

Макрос catkin_package генерує конфігураційні файли cmake для вашого пакету Заявіть, що речі слід передавати залежним проектам

INCLUDE_DIRS: розкоментуйте це, якщо пакет містить заголовкові файли

БІБЛІОТЕКИ: бібліотеки, які ви створюєте в цьому проекті, які також потрібні залежним проектам

CATKIN_DEPENDS: потрібні також проекти, залежні від catkin_packages

ЗАЛЕЖИТЬ: системні залежності цього проекту, що залежним проектам також потрібен catkin_package (

INCLUDE_DIRS включає

БІБЛІОТЕКИ my_first_ros_pkg

CATKIN_DEPENDS roscpp std_msgs

ЗАЛЕЖИТЬ system_lib

Збірка

Вкажіть додаткові розташування файлів заголовків

Місцеположення вашої упаковки мають бути вказані перед іншими місцями

включати_каталоги (включати) включати_каталоги (

$ {catkin_INCLUDE_DIRS}

Оголосіть бібліотеку C ++

add_library (my_first_ros_pkg

src / $ {PROJECT_NAME} /my_first_ros_pkg.cpp

Додайте цільові залежності бібліотеки cmake як приклад, можливо, доведеться згенерувати код перед бібліотеками або від генерації повідомлень, або від динамічної реконфігурації

add_dependencies (my_first_ros_pkg $ {$ {PROJECT_NAME} _EXPORTED_TARGETS} $ {catkin_EXPORTED_TARGETS})

Оголосіть виконуваний файл C ++

add_executable (my_first_ros_pkg_node src / my_first_ros_pkg_node.cpp)

Додайте цільові залежності виконуваного файлу cmake так само, як і для бібліотеки вище

add_dependencies (my_first_ros_pkg_node $ {$ {PROJECT_NAME} _EXPORTED_TARGETS} $ {catkin_EXPORTED_TARGETS})

Вкажіть бібліотеки, з якими потрібно пов’язати бібліотеку або виконуваний об’єкт

target_link_libraries (my_first_ros_pkg_node

$ {catkin_LIBRARIES}

Встановити усі цілі встановлення повинні використовувати змінні CATIN DESTINATION

Див.

Http://ros.org/doc/api/catkin/html/adv_user_guide/variables.html

Позначте виконувані сценарії (Python тощо) для встановлення

на відміну від setup.py, ви можете вибрати пункт призначення

встановити (ПРОГРАМИ

scripts / my_python_script

DESTINATION $ {CATKIN_PACKAGE_BIN_DESTINATION}

Позначте виконувані файли та / або бібліотеки для встановлення

встановити (ЦІЛІ my_first_ros_pkg my_first_ros_pkg_node

АРХІВ ДЕСТИНАЦІЯ $ {CATKIN_PACKAGE_LIB_DESTINATION}

НАЗНАЧЕННЯ БІБЛІОТЕКИ $ {CATKIN_PACKAGE_LIB_DESTINATION}

RUNTIME DESTINATION $ {CATKIN_PACKAGE_BIN_DESTINATION}

Позначте файли заголовка cpp для встановлення

встановити (КАТАЛОГ включає / $ {PROJECT_NAME} /

ДЕСТИНАЦІЯ $ {CATKIN_PACKAGE_INCLUDE_DESTINATION}

FILES_MATCHING Шаблон "* .h"

Шаблон ".svn" ВИКЛЮЧИТИ

Позначте інші файли для встановлення (наприклад, файли запуску та пакету тощо)

встановити (ФАЙЛИ

myfile1

myfile2

DESTINATION $ {CATKIN_PACKAGE_SHARE_DESTINATION}

Тестування

Додайте бібліотеки цільових тестів cpp та бібліотеки посилань

catkin_add_gtest ($ {PROJECT_NAME} -тестовий тест / test_my_first_ros_pkg.cpp)

якщо (TARGET $ {PROJECT_NAME} -тест)

target_link_libraries ($ {PROJECT_NAME} -тест $ {PROJECT_NAME})

endif ()

Додайте папки для запуску python nosetests

catkin_add_nosetests (тест)

Параметри у файлі конфігурації збірки (CMakeLists.txt) такі. Нижче описано мінімально необхідну версію програми "cmake", встановлену в операційній системі. Оскільки в даний час вона вказана як версія 2.8.3, якщо ви використовуєте нижчу версію Cmake, ніж ця, вам потрібно оновити 'cmake', щоб відповідати мінімальним вимогам.

cmake_minimum_required (ВЕРСІЯ 2.8.3)

Проект описує назву пакета. Використовуйте ім'я пакета, введене в 'package. xml '. Зверніть увагу, що якщо ім'я пакета відрізняється від імені пакета, описаного в тезі <name> у 'package.xml', під час створення пакету буде виникати помилка.

проект (my_first_ros_pkg)

Запис 'find_package' - це пакет компонентів, необхідний для побудови збірки на Catkin. У цьому прикладі 'roscpp' та 'std_msgs' встановлюються як залежні пакети. Якщо введений тут пакет не знайдено в системі, під час побудови пакету виникне помилка. Іншими словами, це можливість вимагати встановлення залежних пакетів для спеціального пакету.

find_package (catkin ПОТРІБНІ КОМПОНЕНТИ

roscpp

std_msgs

Нижче наведено метод, який використовується при використанні пакетів, відмінних від ROS. Наприклад, при використанні Boost заздалегідь потрібно встановити пакет "system". Ця функція є опцією, яка дозволяє встановлювати залежні пакети.

find_package (система Boost ПОТРІБНИХ КОМПОНЕНТІВ)

'Catkin_python_setup ()' є опцією при використанні Python з 'rospy'. Він викликає процес встановлення Python 'setup.py'.

catkin_python_setup ()

'add_message_files' - це можливість додати файл повідомлення. Параметр "ФАЙЛИ" автоматично генерує файл заголовка (* .h), посилаючись на файли ".msg" у папці "msg" поточного пакета. У цьому прикладі використовуються файли повідомлень Message1.msg та Message2.msg.

add_message_files (

ФАЙЛИ

Message1.msg

Message2.msg

'add_service_files' - це можливість додати службовий файл для використання. Параметр "ФАЙЛИ" буде посилатися на файли ".srv" у папці "srv" у пакунку. У цьому прикладі у вас є можливість використовувати службові файли Service1.srv та Service2.srv.

add_service_files (

ФАЙЛИ

Сервіс1.srv

Сервіс2.srv

'generated_messages' - це можливість встановити залежні повідомлення. Цей приклад встановлює опцію ЗАВИСИМОСТІ для використання пакету повідомлень 'std_msgs'.

generated_messages (

ЗАЛЕЖНОСТІ

std_msgs

'generated_dynamic_reconfigure_options' завантажує файли конфігурації, на які посилаються при використанні 'dynamic_reconfigure'.

generated_dynamic_reconfigure_options (

cfg / DynReconf1.cfg

cfg / DynReconf2.cfg

Нижче наведено варіанти під час виконання збірки на Catkin. 'INCLUDE_DIRS' - це параметр, який визначає використання заголовного файлу в папці 'include', яка є внутрішньою папкою пакету. 'БІБЛІОТЕКИ' - це параметр, який використовується для вказівки бібліотеки пакетів у наступній конфігурації. 'CATKIN_DEPENDS' визначає залежні пакети, і в цьому прикладі для залежних пакетів встановлюються значення 'roscpp' та 'std_msgs'. 'ЗАЛЕЖИТЬ' - це параметр, який описує системно-залежні пакети.

catkin_package (

INCLUDE_DIRS включає

БІБЛІОТЕКИ my_first_ros_pkg

CATKIN_DEPENDS roscpp std_msgs

ЗАЛЕЖИТЬ system_lib

'include_directories' - це можливість вказати папки для включення. У прикладі налаштовано '$ {catkin_ INCLUDE_DIRS}', який посилається на заголовочний файл, що містить папку 'include'.

Щоб вказати додаткову папку включення, додайте її до наступного рядка "$ {catkin_INCLUDE_DIRS}".

включити_каталоги (

$ {catkin_INCLUDE_DIRS}

'add_library' оголошує бібліотеку, яка буде створена після збірки. Наступна опція створить бібліотеку "my_first_ros_pkg" з файлу "my_first_ros_pkg.cpp" у папці "src".

add_library (my_first_ros_pkg

src / $ {PROJECT_NAME} /my_first_ros_pkg.cpp

'add_dependencies' - це команда для виконання певних завдань до процесу побудови, таких як створення залежних повідомлень або динамічна реконфігурація. Наступні параметри описують створення залежних повідомлень та динамічну реконфігурацію, які є залежностями бібліотеки 'my_first_ros_pkg'.

add_dependencies (my_first_ros_pkg $ {$ {PROJECT_NAME} _EXPORTED_TARGETS} $ {catkin_EXPORTED_TARGETS})

'add_executable' визначає виконуваний файл, який буде створений після збірки. Опція визначає систему, яка посилається на файл 'src / my_first_ros_pkg_node.cpp' для створення виконуваного файлу 'my_first _ros_pkg_ node'. Якщо на посилання є кілька файлів "* .cpp", додайте їх після "my_ first_ros_pkg_node.cpp". Якщо потрібно створити два або більше виконуваних файлів, додайте додатковий запис "add_executable".

add_executable (my_first_ros_pkg_node src / my_first_ros_pkg_node.cpp)

Опція "add_dependencies" подібна до описаної раніше "add_dependencies", яка необхідна для виконання певних завдань, таких як створення залежних повідомлень або динамічна реконфігурація перед створенням бібліотек або виконуваних файлів. Далі описується залежність виконуваного файлу з назвою 'my_first_ros_pkg_node', а не бібліотеки, згаданої вище. Найчастіше використовується під час створення файлів повідомлень до створення виконуваних файлів.

add_dependencies (my_first_ros_pkg_node $ {$ {PROJECT_NAME} _EXPORTED_TARGETS} $ {catkin_EXPORTED_TARGETS})

'target_link_libraries' - це опція, яка пов'язує бібліотеки та виконувані файли, які потрібно пов'язати перед створенням виконуваного файлу.

target_link_libraries (my_first_ros_pkg_node

$ {catkin_LIBRARIES}

Крім того, передбачена опція Встановити, яка використовується під час створення офіційного розподільчого пакета ROS, і опція тестування, яка використовується для тестування пакету.

Далі наведено змінений файл конфігурації збірки (CMakeLists.txt). Змініть файл для вашого пакету. Для отримання

додаткової інформації про те, як використовувати файл конфігурації, зверніться до пакетів TurtleBot3 та ROBOTIS OP3, опублікованих за посиланням 'https://github.com/ROBOTIS-GIT'.

cmake_minimum_required (ВЕРСІЯ 2.8.3)

проект (my_first_ros_pkg)

find_package (catkin ПОТРІБНІ КОМПОНЕНТИ roscpp std_msgs)

catkin_package (CATKIN_DEPENDS roscpp std_msgs)

включити_каталоги ($ {catkin_INCLUDE_DIRS})

add_executable (hello_world_node src / hello_world_node.cpp)

target_link_libraries (hello_world_node $ {catkin_LIBRARIES})

Наступний параметр налаштовано у розділі створення виконуваного файлу (add_executable) файлу 'CMakeLists.txt', про який згадувалося вище.

add_executable (hello_world_node src / hello_world_node.cpp)

Це налаштування для створення виконуваного файлу 'hello_world_node', посилаючись на вихідний код 'hello_ world_node' у папці 'src' пакета. Оскільки вихідний код 'hello_world_node.cpp' повинен бути створений та написаний розробником вручну, напишемо простий приклад.

Спочатку перейдіть до папки з вихідним кодом (src) у вашій папці пакунків за допомогою команди 'cd' і створіть файл 'hello_world_node.cpp', як показано нижче. У цьому прикладі

використовується редактор gedit, але ви можете використовувати бажаний редактор, такий як vi, gedit, qtcreator, vim або emacs.

```
$ cd ~ / catkin_ws / src / my_first_ros_pkg / src /
$ gedit hello_world_node.cpp
```

Потім у створений файл напишіть наступний вихідний код.

hello_world_node.cpp

```
#include <ros / ros.h>
#include <std_msgs / String.h>
#include <sstream>
int main (int argc, char β argv)
ros :: init (argc, argv, "hello_world_node");
ros :: NodeHandle nh;
ros :: Видавець chatter_pub = nh.advertise <std_msgs :: String> ("say_hello_world", 1000);
ros :: Оцінювати швидкість циклу (10);
int count = 0;
while (ros :: ok ())
std_msgs :: String msg;
std :: stringstream ss;
<< "привіт світ!" << рахувати; msg.data = ss.str (); ROS_INFO ("% s", msg.data.c_str ()); chatter_pub.publish (повідомлення); ros :: spinOnce (); loop_rate.sleep ();
++ рахунок;
```

повернути 0;

Після збереження вищезазначеного коду у файлі завершено всю необхідну роботу зі створення пакета. Перш ніж створювати пакет, оновіть профіль пакета ROS за допомогою наведеної нижче команди. Це команда застосувати раніше створений пакет до списку пакетів ROS. Хоча це не є обов’язковим, його зручно оновлювати після створення нового пакету, оскільки це дозволяє знайти пакет за допомогою функції автоматичного заповнення клавішею Tab.

Профіль $ rospack

Далі йде збірка Catkin. Перейдіть до робочої області Catkin і створіть пакет.

$ cd ~ / catkin_ws && catkin_make

Команда "Псевдонім"

Як згадувалося в Розділі 3.2 Налаштування середовища розробки ROS, якщо ви встановите псевдонім cm = 'cd ~ / catkin_ws && catkin_make' у файлі '.bashrc', ви можете замінити наведену вище команду командою 'cm' у вікні терміналу. Оскільки це дуже корисно, обов’язково встановіть його, звернувшись до розділу налаштування середовища розробки ROS.

Якщо збірка була завершена без помилок, файл 'hello_world_node' повинен був бути створений в '~ / catkin_ws / devel / lib / my_first_ros_pkg'.

Наступним кроком є запуск вузла. Відкрийте вікно терміналу (Ctrl + Alt + t) і запустіть roscore перед запуском вузла. Зверніть увагу, що roscore повинен працювати, щоб виконувати ROS-вузли, і roscore потрібно запускати лише один раз, якщо він не зупиниться.

$ roscore

Нарешті, відкрийте нове вікно терміналу (Ctrl + Alt + t) і запустіть вузол за допомогою команди нижче. Це команда для запуску вузла під назвою 'hello_world_node' у пакеті з назвою 'my_first_ros_pkg'.

$ rosrun my_first_ros_pkg hello_world_node

[ІНФО] [1499662568.416826810]: привіт світ! 0

[ІНФО] [1499662568.516845339]: привіт світ! 1

[ІНФОРМАЦІЯ] [1499662568.616839553]: привіт світ! 2

[ІНФОРМАЦІЯ] [1499662568.716806374]: привіт світ! 3

[ІНФО] [1499662568.816807707]: привіт світ! 4

[ІНФО] [1499662568.916833281]: привіт світ! 5

[ІНФО] [1499662569.016831357]: привіт світ! 6

[ІНФО] [1499662569.116832712]: привіт світ! 7

[ІНФО] [1499662569.216827362]: привіт світ! 8

[ІНФОРМАЦІЯ] [1499662569.316806268]: привіт світ! 9

[ІНФОРМАЦІЯ] [1499662569.416805945]: привіт світ! 10

Коли вузол запущений, повідомлення, такі як «привіт світ! 0,1,2,3 ...», можна переглядати у вікні терміналу в рядках. Це не є

фактичною передачею повідомлень у ROS, але це можна розглядати як результат прикладу системи побудови, обговореного в цьому розділі. Оскільки цей розділ призначений для опису системи збірки ROS, вихідний код для повідомлень і вузлів буде розглянуто більш докладно в наступних главах.

Розділ 5 РОС Команди

5.1. Список команд ROS

ROS Wiki

Команди ROS детально пояснюються на сторінці Wiki „http://wiki.ros.org/ROS/Command LineTools“. Крім того, репозиторій GitHub 'https://github.com/ros/cheatsheet/releases' узагальнює важливі команди, описані в цій главі. Це буде корисним посиланням для використання разом з описами в цій главі.

Використовуючи ROS, ми вводимо команди в середовищі Shell для виконання таких завдань, як використання файлових систем, редагування, побудова, налагодження вихідних кодів, управління пакетами тощо. Для правильного використання ROS нам потрібно буде ознайомитись не тільки з основні команди Linux, але також із командами, що стосуються ROS.

Для того, щоб освоїти різні команди, що використовуються для ROS, ми дамо короткий опис функцій кожної команди та представимо їх по одному з прикладами. При введенні команди кожна з них оцінюється трьома зірками залежно від частоти використання та

важливості. Можливо, вам знадобиться деякий час, щоб звикнути до команд, але чим більше ви їх використовуєте, незабаром ви швидко і легко використовуєте всі види функцій ROS, використовуючи ці команди.

Команди оболонки ROS

Команда	Важливість	Пояснення команди	Опис
roscd	★★★	ros + cd (змінює каталог)	Перемістіться до каталогу призначеного
			ROS-пакет
rosls	★ ☆☆	ros + ls (список файлів)	Перевірте список файлів пакета ROS
троянда	★ ☆☆	ros + ed (редактор)	Редагувати файл пакета ROS
roscp	★ ☆☆	ros + cp (копіює файли)	Копіювати файл пакета ROS
rospd	☆☆☆	ros + pushd	Додайте каталог до індексу каталогів ROS
росд	☆☆☆	каталог ros +	Перевірте індекс каталогу ROS

Команди виконання ROS

Команда	Важливість	Пояснення команди	Опис
роскор	★★★	рос + ядро	master (служба імен ROS) +
			rosout (журнал записів) + параметр
			сервер (керувати параметром)

Команда	Важливість	Пояснення команди	Опис
росрун	★★★	рос + біг	Запустити вузол
розпродаж	★★★	ros + запуск	Запустіть кілька вузлів і налаштуйте
			варіанти
росклін	★★ ☆	рос + чистий	Перевірте або видаліть файл журналу ROS

Інформаційні команди ROS

Команда	Важливість	Пояснення команди	Опис
ростопічний	★★★	рос + тема	Перевірте інформацію про тему ROS
россервіс	★★★	рос + послуга	Перевірте інформацію про службу ROS
рознос	★★★	ros + вузол	Перевірте інформацію про вузол ROS
роспарам	★★★	ros + param (параметр)	Перевірте та відредагуйте параметр ROS
			інформація
росбаг	★★★	рос + мішок	Запис і відтворення ROS-повідомлення
rosmsg	★★ ☆	ros + повідомлення	Перевірте інформацію про повідомлення ROS

rossrv	★★ ☆	рос + срв	Перевірте інформацію про службу ROS
роверсія	★ ☆☆	ros + версія	Перевірте пакет ROS та випуску
			інформація
roswtf	☆☆☆	ros + wtf	Вивчіть систему АФК

ROS Catkin Commands

Команда	Важливість	Опис
catkin_create_pkg	★★★	Автоматичне створення пакета
catkin_make	★★★	Збірка на основі системи збірки catkin
catkin_eclipse	★★ ☆	Змінити пакет, створений системою збірки catkin, так
		що його можна використовувати в Eclipse
catkin_prepare_release	★★ ☆	Очищення журналу та версії тегу під час випуску
catkin_generate_changelog	★★ ☆	Створіть або оновіть файл "CHANGELOG.rst" під час випуску
catkin_init_workspace	★★ ☆	Ініціалізуйте робочий простір системи побудови catkin
catkin_find	★ ☆☆	Пошуковий котик

Команди пакету ROS

Команда	Важливість	Пояснення команди	Опис
ропак	★★★	рос + упаковка (упаковка)	Перегляньте інформацію щодо конкретного
			ROS-пакет
каніфоль	★★ ☆	ros + встановити	Встановіть додаткові пакети ROS
росдеп	★★ ☆	ros + dep (залежності)	Встановіть пакет залежностей ROS
			відповідний пакет
рослокація	☆☆☆	ros + знайти	Показати інформацію про пакет ROS
roscreate-pkg	☆☆☆	ros + create-pkg (пакет)	Автоматичне створення пакета ROS
			(використовується в попередній системі rosbuild)
росмейк	☆☆☆	рос + зробити	Складіть пакет ROS
			(використовується в попередній системі rosbuild)

5.2. Команди оболонки ROS

Команди оболонки ROS також називаються rosbash1. Ці команди дозволяють нам використовувати команди оболонки bash, які зазвичай використовуються в Linux, і для середовища розробки ROS.

Зазвичай префікси "ros" використовуються разом із суфіксами, такими як "cd, pd, d, ls, ed, cp, run". Відповідні команди такі.

Команда	Важливість	Пояснення команди	Опис
roscd	★★★	ros + cd (змінює каталог)	Перемістіться до каталогу
			призначений пакет ROS
rosls	★☆☆	ros + ls (список файлів)	Перевірте список файлів пакета ROS
троянда	★☆☆	ros + ed (редактор)	Редагувати файл пакета ROS
roscp	★☆☆	ros + cp (копіює файли)	Копіювати файл пакета ROS
rospd	☆☆☆	ros + pushd	Додати каталог до каталогу ROS
			індекс
росд	☆☆☆	каталог ros +	Перевірте індекс каталогу ROS

З них ми розглянемо команди 'roscd', 'rosls' та 'rosed', які часто використовуються.

Середовище для використання команд оболонки ROS

Для того, щоб використовувати команди оболонки ROS, 'rosbash' повинен бути встановлений за допомогою наведеної нижче команди і може використовуватися лише у вікні терміналу, в якому налаштовано 'source / opt / ros / <ros distribution> / setup.bash'. Це не потрібно встановлювати окремо, і якщо ви закінчили створювати середовище розробки ROS з розділу 3, тоді ви зможете ним користуватися.

$ sudo apt-get install ros- <ros distribution> -rosbash

roscd [PACKAGE_NAME]

Це команда для переміщення до каталогу, де зберігається пакет. Основна інструкція - ввести команду 'roscd', за якою в якості параметра вводиться назва пакета. У наступному прикладі пакет turtlesim знаходиться в папці, де встановлено ROS, тому ми отримаємо такий результат, але якщо ви введете в якості параметра створений вами пакет (наприклад, my_first_ros_pkg, створений у розділі 4), то він переміщується до папки пакунку, яку ви визначили. Це команда, яка часто використовується в ROS на основі команд.

$ roscd черепаха

/ opt / ros / kinetic / share / turtlesim $

$ roscd my_first_ros_pkg

~ / catkin_ws / src / my_first_ros_pkg $

Спочатку потрібно встановити пакет ros-kinetic-turtlesim, щоб отримати однаковий результат, як показано вище. Якщо він ще не встановлений, ви можете встановити його за допомогою наступної команди.

$ sudo apt-get install ros-kinetic-turtlesim

Якщо пакет вже встановлений, ви побачите повідомлення про те, що пакет уже встановлений, як показано нижче.

$ sudo apt-get install ros-kinetic-turtlesim

[sudo] пароль для КОРИСТУВАЧА:

Читання списків пакетів ... Готово

Побудова дерева залежностей

Читання інформації про стан ... Готово

ros-kinetic-turtlesim - це вже найновіша версія (0.7.1-0xenial-20170613-170649-0800).

0 оновлено, 0 нещодавно встановлено, 0 видалено та 29 не оновлено.

Середовище для використання команд оболонки ROS

Щоб використовувати команди оболонки ROS, 'rosbash' повинен бути встановлений за допомогою наступної команди і може використовуватися тільки у вікні терміналу, налаштованому 'source /opt / ros / <ROS distribution>/ setup.bash’. Це не обов'язково встановлювати окремо, і якщо ви закінчили створення середовища розробки ROSE з глави 3, то зможете її використовувати.

$ sudo apt-get install ros - <ROS distribution> - rosbash

roscd [PACKAGE_NAME]

Це команда для переміщення в каталог, в якому збережений пакет. Основна інструкція полягає в тому, щоб ввести команду "roscd", за якою слідує ім'я пакета в якості параметра. В наступному пакет turtlesim знаходиться в папці, де встановлено ROS, тому ми отримуємо наступний результат, але якщо ви помістіть ім'я створеного вами пакета (наприклад, my_first_ros_pkg, створеного в розділі 4) в якості

параметра, то він переміститься в папку зазначеного вами пакета. Це команда, яка часто використовується в ROS на основі команд.

$ roscd turtlesim

/opt/ros/kinetic/share/turtlesim $

$ roscd my_first_ros_pkg

~/catkin_ws/src/my_first_ros_pkg $

Пакет ros-kinetic-turtlesim повинен бути встановлений першим, щоб отримати ідентичний результат, як показано вище. Якщо він ще не встановлений, ви можете встановити його за допомогою наступної команди.

$ sudo apt-get install ros-kinetic-turtlesim

Якщо пакет вже встановлений, ви побачите повідомлення про те, що пакет вже встановлено, як показано нижче.

$ sudo apt-get install ros-kinetic-turtlesim

[sudo] password for USER:

Reading package lists... Done

Building dependency tree Reading state information... Done

ros-kinetic-turtlesim is already the newest version (0.7.1-0xenial-20170613-170649-0800).

0 upgraded, 0 newly installed, 0 to remove and 29 not upgraded.

rosls [PACKAGE_NAME]

Це команда для перевірки списку файлів конкретного пакета ROS. Ми можемо використовувати команду " roscd" для переміщення

в відповідну папку пакета, а потім використовувати команду " ls " для виконання тієї ж функції, але ця команда використовується іноді, коли нам потрібно перевірити, не переміщаючись в каталог пакета

$ rosls turtlesim

cmake images msg srv package.xml

Це команда, яка використовується для редагування певного файлу в пакеті. Якщо ви запустите цю команду, вона відкриє відповідний файл за допомогою редактора, налаштованого Користувачем. Це часто використовується, коли ви хочете швидко зробити просту модифікацію. Користувач може призначити, який редактор буде використовуватися, відредагувавши команду export EDITOR = 'emacs-nw ' у вашому файлі'~/.bashrc’.

Як вже згадувалося раніше, ця команда використовується для простих завдань, які необхідно змінити безпосередньо в командному вікні, і не рекомендується для складних завдань. Це не та команда, яка використовується часто.

$ rosed turtlesim package.xml

5.3. Команди виконання ROS

Команди виконання ROS керують виконанням вузлів ROS. Перш за все, найбільш важливим є roscoe, який використовується в якості сервера імен для вузлів. А для виконання команд є 'rosrun' і 'roslaunch'. Ми використовуємо "rosrun' для запуску одного вузла і 'roslaunch" для запуску декількох вузлів або для додаткової настройки

різних опцій. А "rosclean" - це команда, яка видаляє журнали, записані під час роботи вузла.

Command	Importance	Command Explanation	Description
roscore	★★★	ros+core	master(ROS name service) + rosout(record log) + parameter server(manage parameter)
rosrun	★★★	ros+run	Run node
Command	**Importance**	**Command Explanation**	**Description**
roslaunch	★★★	ros+launch	Launch multiple nodes and configure options
rosclean	★★☆	ros+clean	Examine or delete ROS log file

Таблиця 1

5.3.1. roscore: запустити roscore

roscore [OPTION]

Roscoe-це майстер, який керує інформацією про з'єднання для зв'язку між вузлами і є важливим елементом, який повинен бути першим запущений для використання ROS. Майстер ROSE запускається командою "roscore' і працює як сервер XMLRPC. Майстер реєструє інформацію вузла, таку як імена, теми та імена служб, типи повідомлень, адреси URI і порти, і при надходженні запиту ця інформація передається іншим вузлам.

При запуску "roscore" також виконується "rosout", який використовується для запису стандартного виводу ROS журнали, такі як DEBUG, INFO, WARN, ERROR, FATAL і т. д. також виконується сервер параметрів, який управляє параметрами.

Коли roscore працює, URI, налаштований в ROS_MASTER_URI, встановлюється в якості головного Урі, щоб запустити майстра. ROS_MASTER_URI може бути налаштований користувачем у файлі'~/.bashrc', як вказано в розділі конфігурації ROS в главі 3.

$ roscore

... logging to /home/pyo/.ros/log/c2d0b528-6536-11e7-935b-08d40c80c500/roslaunch-pyo-20002.log Checking log directory for disk usage. This may take awhile.

Press Ctrl-C to interrupt

Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://localhost:43517/

ros_comm version 1.12.7

SUMMARY

========

*PARAMETERS * /rosdistro: kinetic*

/rosversion: 1.12.7

NODES

auto-starting new master

process[master]: started with pid [20013]

ROS_MASTER_URI=http://localhost:11311/

setting /run_id to c2d0b528-6536-11e7-935b-08d40c80c500

process[rosout-1]: started with pid [20027]

started core service [/rosout]

На екрані терміналу ми бачимо, що журнали зберігаються в каталозі ' / home / xxx/.ros/журнал/'. Відображуване повідомлення також повідомляє про те, що ми можемо закрити roscoe за допомогою [Ctrl+c], інформації сервера roslaunch і ROS_MASTER_URI, а також про те, що сервер параметрів '/ rosdistro' і '/rosversion' і вузол / rosout всі були запущені.

Шлях збереження журналу

У наведеному вище прикладі показано, що шлях, за яким зберігаються журнали, називається " / home / xxx/.ros/ log/", але насправді він зберігається в тому місці, де налаштована змінна оточення ROS_HOME. Якщо змінна оточення ROS_HOME не налаштована, то значення за замовчуванням дорівнює'~/.ros/log/'.

rosrun [PACKAGE_NAME] [NODE_NAME]

Rosrun-це команда, яка запускає лише один вузол у вказаному пакеті. Наступний приклад-це команда, яка запускає вузол 'turtlesim_node' в пакеті turtlesim. До Вашого відома, значок черепахи, який з'являється на екрані, вибирається випадковим чином

$ rosrun turtlesim turtlesim_node

[INFO] [1512634911.275228307]: Starting turtlesim with node name /turtlesim

[INFO] [1512634911.281614642]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445],

theta=[0.000000]

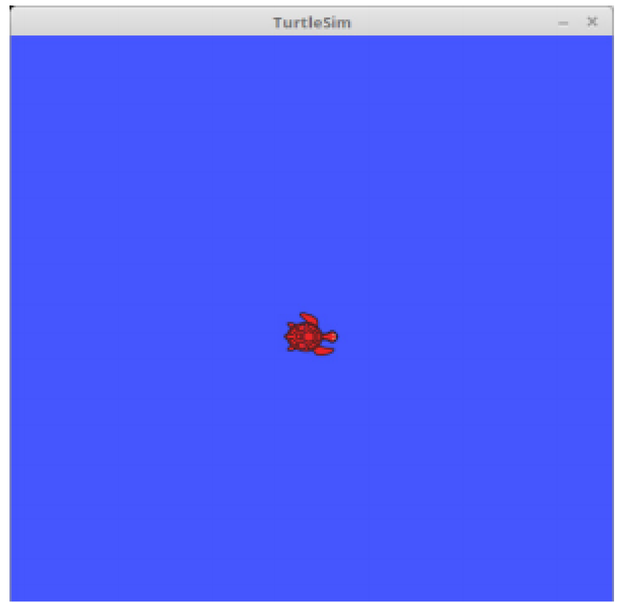


Рис. 40 Екран після виконання вузла turtlesim_node

roslaunch [PACKAGE_NAME] [launch_FILE_NAME]

Roslaunch-це команда, яка виконує більше одного вузла в зазначеному пакеті або задає параметри виконання. Як показано в наступному прикладі, простий запуск пакета openni_launch призведе до запуску більше 20 вузлів і більше 10 серверів параметрів, таких як

camera_nodelet_manager, depth_metric, depth_metric_rect, depth_points і т. Д.

Як ми бачимо, використання файлу запуску дуже корисно для одночасного запуску декількох вузлів і є часто використовуваним методом виконання в ROS. Додаткові відомості про створення файлу'*. launch', який використовується в цей приклад буде наведено в розділі 7.6 з використанням roslaunch.

$ roslaunch openni_launch openni.launch

~ omitted ~

Зверніть увагу, що для того, щоб запустити цей приклад і отримати той же результат, необхідно встановити відповідний пакет 'roskinetic-openni-launch'. Якщо він ще не встановлений, ви можете встановити його з допомогою наступної команди.

$ sudo apt-get install ros-kinetic-openni-launch

rosclean [OPTION]

Це команда для перевірки або видалення файлу журналу ROS. При запуску" roscore " історія всіх вузлів записується в файл журналу, і дані накопичуються з плином часу, тому їх необхідно періодично видаляти за допомогою команди "rosclean".

Нижче наведено приклад вивчення використання журналу.

$ rosclean check

320K ROS node logs → Це означає, що загальне використання вузла ROS становить 320 КБ

При запуску "roscore", якщо з'являється наступне попереджувальне повідомлення, це означає, що файл журналу перевищує 1 ГБ. Якщо в системі не вистачає місця для журналу, очистіть його командою"rosclean".

WARNING: disk usage in log directory [/xxx/.ros/log] is over 1GB.

Нижче наведено приклад видалення журналів в репозиторії журналів ROS (в даному прикладі це ' / home/rt/.ros/log'). Якщо ви хочете видалити файл, натисніть клавішу "y", щоб продовжити.

$ rosclean purge

Purging ROS node logs.

PLEASE BE CAREFUL TO VERIFY THE COMMAND BELOW!

Okay to perform:

rm -rf /home/pyo/.ros/log

(y/n)?

5.4. Інформаційні команди ROS

Інформаційні команди ROS використовуються для перевірки інформації про теми, служби, вузли, зокрема, часто використовуються "ростопік", "россервіс", "роснод" і " роспарам", а "росбаг" може записувати і відтворювати дані, що є однією з основних особливостей ROS, тому обов'язково повністю розберіться в ній.

Command	Importance	Command Explanation	Description
rostopic	★★★	ros+topic	Check ROS topic information
rosservice	★★★	ros+service	Check ROS service information
rosnode	★★★	ros+node	Check ROS node information
rosparam	★★★	ros+param(parameter)	Check and edit ROS parameter information
rosbag	★★★	ros+bag	Record and play ROS message
rosmsg	★★☆	ros+msg	Check ROS message information
rossrv	★★☆	ros+srv	Check ROS service information
rosversion	★☆☆	ros+version	Check ROS package and release version information
roswtf	☆☆☆	ros+wtf	Examine ROS system

Таблиця 2

Ми будемо використовувати наступні команди, щоб практикувати turtlesim, що надається ROS, щоб дізнатися про вузли, теми та послуги. Перед тестуванням за допомогою інформаційних команд ROS нам потрібно буде зробити наступні приготування.

Запуск roscore

Закрийте всі термінали, щоб уникнути конфліктів з іншими процесами. Потім відкрийте новий термінал і виконайте наступну команду.

$ roscore

Щоб запустити 'turtlesim_node' в пакеті 'turtlesim', відкрийте новий термінал і виконайте наступну команду. Це призведе до запуску 'turtlesim_node' в пакеті 'turtlesim'. З'явиться синій екран з випадковим зображенням черепахи

$ rosrun turtlesim turtlesim_node

Запустіть вузол turtle_teleop_key в пакеті turtlesim

Відкрийте новий термінал і виконайте наступну команду. Це призведе до запуску вузла 'turtle_teleop_key' в пакеті 'turtlesim'. Як тільки він буде запущений, ми зможемо використовувати клавіші зі стрілками в цьому вікні терміналу для управління черепахою. Натискання клавіш зі стрілками буде переміщати черепаху по екрану. Хоча це проста симуляція, вона все ще посилає повідомлення зі швидкістю переміщення (м/с) і швидкістю обертання (рад / с), необхідними для переміщення черепахи, і ми зможемо керувати реальним роботом пізніше з тим же повідомленням віддалено. Для отримання більш детальної інформації та інструкцій щодо повідомлень зверніться до розділу 4.2 Передача повідомлень і главі 7 Основи програмування ROS.

$ rosrun turtlesim turtle_teleop_key

Розуміння вузлів необхідно, тому, будь ласка, зверніться до розділу 4.1 термінологія ROS.

Command	Description
rosnode list	Check the list of active nodes
rosnode ping [NODE_NAME]	Test connection with a specific node
rosnode info [NODE_NAME]	Check information of a specific node
rosnode machine [PC_NAME OR IP]	Check the list of nodes running on the corresponding PC
rosnode kill [NODE_NAME]	Stop running a specific node
rosnode cleanup	Delete the registered information of the ghost nodes for which the connection information cannot be checked

Таблиця 3

Список rosnode: Перевірте список запущених вузлів

Це команда для перерахування всіх вузлів, підключених до"роскору". Якщо у вас є тільки запустити ' roscore і у попередньому прикладі ('turtlesim_node', 'turtle_teleop_key'), то ви побачите, що 'rosout, які виконується разом з 'roscore для журналів, записи, і teleop_turtle" і "turtlesim' вузли були виконані в попередньому прикладі.

```
$ rosnode list
/rosout
/teleop_turtle
/turtlesim
```

Запуск вузла і фактичне ім'я вузла

У попередньому прикладі, коли' turtlesim_node 'і' turtle_teleop_key ' виконуються, а

'teleop_turtle' і' turtlesim 'з'являються в списку 'rosnode', це відбувається тому, що ім'я запущеноговузла відрізняється від фактичного імені вузла. Наприклад, вузол' turtle_teleop_key ' налаштований як " ros:: init (argc, argv, "teleop_turtle"); " у вихідному файлі. Ми рекомендуємо створити одне і те ж ім'я запущеного вузла і фактичне ім'я вузла.

rosnode ping [NODE_NAME]: тестове з'єднання з певним вузлом

Нижче наведено тест, щоб перевірити, чи дійсно вузол turtlesim підключений до комп'ютера, який використовується в даний момент.

Якщо він підключений, то отримає відповідь XMLRPC від відповідного вузла наступним чином.

```
$ rosnode ping /turtlesim
rosnode: node is [/turtlesim]
pinging /turtlesim with a timeout of 3.0s
xmlrpc reply from http://192.168.1.100:45470/       time=0.377178ms
```

Якщо виникла проблема з запуском відповідного вузла або зв'язок була перервана, з'явиться наступне повідомлення про помилку.

```
ERROR: connection refused to [http://192.168.1.100:55996/]
```

rosnode info [NODE_NAME]: Перевірка інформації про конкретний вузол

Використовуючи команду "rosnode info", ми можемо перевірити інформацію про конкретний вузол. Як правило,ви можете перевірити публікації, підписки, служби, а також інформацію про URI запущеноговузла і введенні / виведення теми. Більша частина інформації, яка відображається тут, була опущена, тому рекомендується виконати цю практику.

```
$ rosnode info /turtlesim
--------------------------------------------
Node [/turtlesim] Publications:
 * /turtle1/color_sensor [turtlesim/Color]
~ omitted ~
```

ROS node machine [PC_NAME Або IP]: Перевірте список вузлів, що працюють на відповідному ПК

```
$ rosnode machine 192.168.1.100
/rosout
/teleop_turtle
/turtlesim
```

rosnode kill [NODE_NAME]: зупинити запуск певного вузла

Це команда для закриття запущеного вузла. Ми можемо закрити вузол безпосередньо за допомогою [Ctrl + c] у вікнітерміналу, де був запущений вузол, але ми також можемо закрити його, вказавши вузол для знищення наступним чином.

```
$ rosnode kill /turtlesim
killing /turtlesim
killed
```

Якщо ми закриємо вузол за допомогою цієї команди, у вікні терміналу, де працює відповідний вузол, з'явиться попереджувальне повідомлення, як показано нижче, і вузол буде закритий.

```
[WARN] [1512635717.915684117]: Shutdown request received.
[WARN] [1512635717.915711940]: Reason given for shutdown: [user request]
```

Очищення вузла ros: видалення зареєстрованої інформації примарних вузлів з неперевіреною інформацією про з'єднання.

Ця команда видаляє неперевірену інформацію про з'єднання примарних вузлів. Коли вузол аварійно завершує роботу через несподівану помилку, Ця команда видаляє відповідний вузол зі списку, в якому була вирізана інформація про з'єднання. Хоча ця

команда використовується не часто, вона корисна, тому що вам не потрібно завершувати роботу і знову запускати " roscore", щоб видалити примарний вузол.

```
$ rosnode cleanup
```

Розуміння тем необхідне, тому, будь ласка, зверніться до розділу 4.1 термінологія ROS.

Command	Description
rostopic list	Show the list of active topics
rostopic echo [TOPIC_NAME]	Show the content of a message in real-time for a specific topic
rostopic find [TYPE_NAME]	Show the topics that use specific message type
rostopic type [TOPIC_NAME]	Show the message type of a specific topic
rostopic bw [TOPIC_NAME]	Show the message data bandwidth of a specific topic
rostopic hz [TOPIC_NAME]	Show the message data publishing period of a specific topic
rostopic info [TOPIC_NAME]	Show the information of a specific topic
rostopic pub [TOPIC_NAME] [MESSAGE_TYPE] [PARAMETER]	Publish a message with the specific topic name

I

Таблиця 4

Закрийте всі вузли перед запуском прикладу, що відноситься до теми ROS. Потім запустіть "roscore', "turtlesim_node" і "turtle_teleop_key" в трьох різних вікнах терміналу, виконавши наступні команди.

```
$ roscore
$ rosrun turtlesim turtlesim_node
$ rosrun turtlesim turtle_teleop_key
```

rostopic list: Показати список активних тем

Команда 'rostopic' list перераховує теми, які в даний момент відправляються і приймаються.

```
$ rostopic list
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

Якщо ви додасте опцію '- v' в команду 'rostopic list', вона розділить опубліковані теми і підписані теми, а також покаже тип повідомлення для кожної теми.

```
$ rostopic list -v
Published topics:
*     /turtle1/color_sensor [turtlesim/Color] 1 publisher
*     /turtle1/cmd_vel [geometry_msgs/Twist] 1 publisher
*     /rosout [rosgraph_msgs/Log] 2 publishers
*     /rosout_agg [rosgraph_msgs/Log] 1 publisher
*     /turtle1/pose [turtlesim/Pose] 1 publisher
Subscribed topics:
*     /turtle1/cmd_vel [geometry_msgs/Twist] 1 subscriber
*     /rosout [rosgraph_msgs/Log] 1 subscriber
```

rostopic echo [TOPIC_NAME]: відображення вмісту повідомлення в режимі реального часу для певної теми

У наступному прикладі показані дані для 'x', 'y', 'theta, linear_velocity' і 'angular_velocity', які складають тему '/ turtle1 / pose' в режимі реального часу.

```
$ rostopic echo /turtle1/pose
x: 5.35244464874
y: 5.544444561
theta: 0.0
linear_velocity: 0.0
angular_velocity: 0.0
~ omitted ~
```

rostopic find [TYPE_NAME]: показати теми, що використовують певний тип повідомлення

```
$ rostopic find turtlesim/Pose
/turtle1/pose
```

rostopic type [TOPIC_NAME]: показати тип повідомлення певної теми

```
$ rostopic type /turtle1/pose
turtlesim/Pose
```

rostopic BW [TOPIC_NAME]: показати пропускну здатність даних повідомлення певної теми

наступному прикладі ми бачимо, що пропускна здатність даних, що використовується в розділі "/ turtle1 / pose", дорівнює в середньому 1,27 КБ в секунду.

```
$ rostopic bw /turtle1/pose
subscribed to [/turtle1/pose]
average: 1.27KB/s
mean: 0.02KB min: 0.02KB max: 0.02KB window: 62 ...
~ omitted ~
```

rostopic Hz [TOPIC_NAME]: показати період публікації даних повідомлення зазначеної теми

У наступному прикладі ми можемо перевірити період публікації даних '/ turtle1 / pose'. В результаті ми бачимо, що повідомлення публікується з періодом близько 62,5 Гц (0,016 сек =16 мсек).

```
$ rostopic hz /turtle1/pose
subscribed to [/turtle1/pose]
average rate: 62.502
min: 0.016s max: 0.016s std dev: 0.00005s window: 62
```

rostopic info [TOPIC_NAME]: Показати інформацію з певної теми

У наступному прикладі ми бачимо, що тема "/ turtle1 / pose "використовує тип повідомлення" turtlesim / Pose " і публікується з вузла "/turtlesim", а підписаних тем немає.

```
$ rostopic info /turtle1/pose
Type: turtlesim/Pose
Publishers:
 * /turtlesim (http://192.168.1.100:42443/)
Subscribers: None
```

rostopic pub [TOPIC_NAME] [MESSAGE_TYPE] [PARAMETER]: Публікація повідомлення з вказаним ім'ям теми

Нижче наведено приклад публікації повідомлення з назвою теми "/ turtle1 / cmd_vel " з

типом повідомлення "geometry_msgs / Twist".

```
$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'
publishing and latching message for 3.0 seconds
```

Нижче наводиться опис кожного з варіантів.

■ -l опублікуйте повідомлення лише один раз (він запускається лише один раз, але працює протягом 3 секунд, як показано високий).

■ / turtle1 / cmd_vel конкретне ім'я теми

■ geometry_msgs / Twist ім'я опублікованого типу повідомлення

■ -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]' переміщення в координаті осі x зі швидкістю 2,0 м в секунду і з обертанням 1,8 рад в секунду навколо осі z

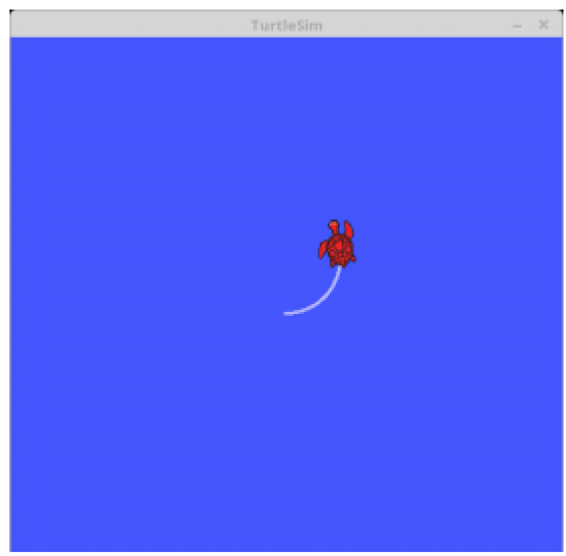


Рис. 41 Екран, що показує застосоване опубліковане повідомлення

5.4.4. rosservice: ROS Service

Розуміння послуг необхідно, тому, будь ласка, зверніться до розділу 4.1 термінології ROS.

Command	Description
rosservice list	Display information of active services
rosservice info [SERVICE_NAME]	Display information of a specific service
rosservice type [SERVICE_NAME]	Display service type
rosservice find [SERVICE_TYPE]	Search services with a specific service type
rosservice uri [SERVICE_NAME]	Display the ROSRPC URI service
rosservice args [SERVICE_NAME]	Display the service parameters
rosservice call [SERVICE_NAME] [PARAMETER]	Request service with the input parameter

Таблиця 5

Закрийте всі вузли перед запуском прикладу, що стосується служби ROS. Потім запустіть 'roscore', 'turtlesim_node' і

'turtle_teleop_key' в різних вікнах терміналу, виконавши наступні команди.

```
$ roscore
$ rosrun turtlesim turtlesim_node
$ rosrun turtlesim turtle_teleop_key
```

rosservice list: відображення інформації про активні сервіси

Ця команда відображає інформацію про активні служби. Будуть показані всі служби, використовувані в одній мережі.

```
$ rosservice list
/clear
/kill
/reset
/rosout
/get_loggers
/rosout
/set_logger_level
/spawn
/teleop_turtle/get_loggers
/teleop_turtle/set_logger_level
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level
```

rosservice info [SERVICE_NAME]:Відображення інформації про конкретну послугу

Нижче наведено приклад перевірки імені вузла, URI, типу та параметра '/turtle1 / set_pen ' сервіс з використанням опції info 'rosservice'.

```
$ rosservice info /turtle1/set_pen
Node: /turtlesim
URI: rosrpc://192.168.1.100:34715
Type: turtlesim/SetPen
Args: r g b width off
```

rosservice type [SERVICE_NAME]: Тип обслуговування дисплея

У наступному прикладі ми бачимо, що сервіс '/ turtle1 / set_pen 'є типом' turtlesim/SetPen''.

```
$ rosservice type /turtle1/set_pen
turtlesim/SetPen
```

rosservice find [SERVICE_TYPE]:Пошукові сервіси з певним типом сервісу

Наступний приклад-це команда для пошуку сервісів з типом ''turtlesim / SetPen'. Ми бачимо, що в результаті виходить '/ turtle1 / set_pen'

```
$ rosservice find turtlesim/SetPen
/turtle1/set_pen
```

rosservice uri [SERVICE_NAME]: Відображення URI-сервісу РОСРПК

Використовуючи опцію uri 'rosservice', ми можемо перевірити URI ROSRPC сервісу '/ turtle1 / set_pen', як показано нижче.

```
$ rosservice uri /turtle1/set_pen
rosrpc://192.168.1.100:50624
```

rosservice args [SERVICE_NAME]: Відображення параметрів обслуговування

Давайте перевіримо кожен параметр сервісу '/ turtle1 / set_pen', як показано в наступному прикладі. За допомогою цієї команди ми можемо перевірити, що параметри, використовувані в сервісі '/ turtle1 / set_pen', є 'r',' g',' b',' width 'i'off'.

```
$ rosservice args /turtle1/set_pen
r g b width off
```

rosservice call [SERViCE_NAME] [PARAMETER]:Запит сервісу з вхідним параметром

Наступний приклад-це команда, яка запитує службу '/ turtle1 / set_pen'. Значення '255 0 0 5 0' відповідають параметрам (r, g, b, width, off), використовуваним для сервісу '/turtle1/set_pen'.Значення ''r', Що представляє червоний колір, має максимальне значення 255, в той час як " g " і "b" рівні " 0", тому колір пера буде червоним. 'Ширина' встановлена на товщину 5, а "викл' - на 0 (брехня), тому лінія буде відображатися. 'rosservice call' - це надзвичайно корисна команда, яка використовується для тестування при використанні сервісу і часто використовується.

```
$ rosservice call /turtle1/set_pen 255 0 0 5 0
```

Використовуючи наведену вище команду, ми запросили послугу, яка змінює властивості пера, використовуваного в turtlesim, і, замовивши команду для переміщення в "turtle_teleop_key", ми бачимо, що колір пера, який був білим, тепер відображається червоним, як показано нижче.

Рис. 42 Приклад rosservice call

розуміння параметрів необхідно, тому, будь ласка, зверніться до розділу 4.1 термінологія ROS

Command	Description
rosparam list	View parameter list
rosparam get [PARAMETER_NAME]	Get parameter value
rosparam set [PARAMETER_NAME]	Set parameter value
rosparam dump [FILE_NAME]	Save parameter to a specific file
rosparam load [FILE_NAME]	Load parameter that is saved in a specific file
rosparam delete [PARAMETER_NAME]	Delete parameter

Таблиця 6

Давайте закриємо всі вузли перед запуском прикладу з параметром ROS. Потім запустіть 'roscore', 'turtlesim_node' і 'turtle_teleop_key' в різних вікнах терміналу, виконавши наступні команди.

```
$ roscore
$ rosrun turtlesim turtlesim_node
$ rosrun turtlesim turtle_teleop_key
```

rosparam list: Перегляд списку параметрів

Буде показано список параметрів, що використовуються в одній і тій же мережі.

```
$ rosparam list
/background_b
/background_g
/background_r
/rosdistro
/roslaunch/uris/host_192_168_1_100__39536
/rosversion
/run_id
```

rosparam get [PARAMETER_NAME]:Отримати значення параметра

Якщо ви хочете перевірити значення певного параметра, ви можете ввести його ім'я в якості опції після команди "rosparam get".

```
$ rosparam get /background_b
255
```

Якщо ви хочете перевірити значення всіх інших параметрів, крім певного параметра, ви можете використовувати ' / ' як опцію, яка покаже значення всіх параметрів, як показано нижче.

```
$ rosparam get /
background_b: 255
background_g: 86
background_r: 69
rosdistro: 'kinetic'
roslaunch:
  uris: {host_192_168_1_100__43517: 'http:// 192.168.1.100:43517/'}
rosversion: '1.12.7'
run_id: c2d0b528-6536-11e7-935b-08d40c80c500
```

rosparam dump [FILE_NAME]:Зберегти параметр у певний файл

Наступний приклад-це команда, яка зберігає поточне значення параметра у файлі parameters.yaml. Це дуже корисно, тому що він може зберегти значення параметра, яке було використано для застосування в наступному запуску ('~/' представляє домашній каталог користувача).

```
$ rosparam dump ~/parameters.yaml
```

rosparam set [PARAMETER_NAME]:Встановіть значення параметра

У наступному прикладі параметр' background_b ' вузла turtlesim, який є параметром, що відноситься до кольору фону, встановлюється в значення '0'.

```
$ rosparam set background_b 0
$ rosservice call clear
```

RGB змінюється з '255, 86, 69' на '0, 86, 69', тому колір стає темно-зеленим, як показано на малюнку праворуч на малюнку 5-4. Однак вузол "turtlesim" не читає і не застосовує параметри відразу, тому нам потрібно спочатку змінити параметри командою " rosparam set background_b 0", а потім оновити екран командою"rosservice call clear".

Застосування параметра змінюється в залежності від вузла.

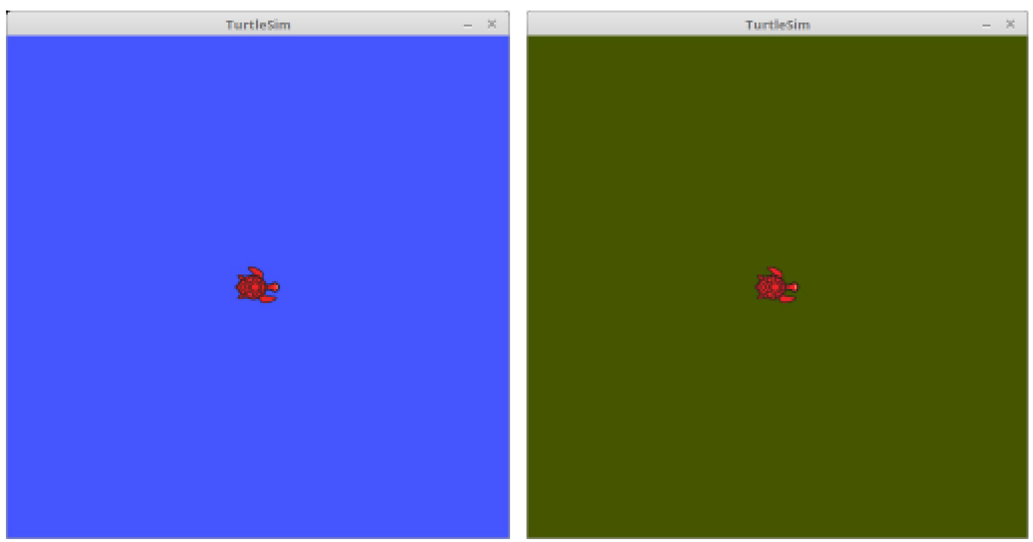


Рис. 43 Приклад‘rosparam set’

rosparam load [FILE_NAME]:Параметр завантаження, збережений у вказаному файлі

На відміну від дампа rosparam, він завантажує файл parameters.yaml і використовує його в якості поточного значення параметра. Як показано в наступному прикладі, якщо ми запустимо команду " rosservice call clear", вона буде замінена значенням параметра завантаженого файлу, а колір фону, який був змінений на зелений на малюнку 5-4, знову зміниться на синій, як це було при запуску команди дампа. Rosparam load-корисна команда, яка використовується досить часто, тому давайте познайомимося з нею.

```
$ rosparam load ~/parameters.yaml
$ rosservice call clear
```

rosparam delete [PARAMETER_NAME]:Видалити параметр

Це команда для видалення певного параметра.

```
$ rosparam delete /background_b
```

Розуміти повідомлення необхідно, тому, будь ласка, зверніться до розділу 4.1 термінології ROS.

Command	Description
rosmsg list	Show a list of all messages
rosmsg show [MESSAGE_NAME]	Show information of a specified message
rosmsg md5 [MESSAGE_NAME]	Show the md5sum
rosmsg package [PACKAGE_NAME]	Show a list of messages used in a specified package
rosmsg packages	Show a list of all packages that use messages

Таблиця 7

Давайте закриємо всі вузли перед запуском прикладу, що стосується інформації про повідомлення ROS. Потім запустіть

'roscore', 'turtlesim_node' і 'turtle_teleop_key' в різних вікнах терміналу, виконавши наступні команди.

```
$ roscore
$ rosrun turtlesim turtlesim_node
$ rosrun turtlesim turtle_teleop_key
```

rosmsg list:Показати список всіх повідомлень

Ця команда перераховує всі повідомлення в встановлених в даний момент пакетах. Результуюче значення може варіюватися в залежності від пакетів, включених в ROS.

```
$ rosmsg list
actionlib/TestAction
actionlib/TestActionFeedback
actionlib/TestActionGoal
actionlib/TestActionResult
```

```
actionlib/TestFeedback
actionlib/TestGoal
sensor_msgs/Joy
sensor_msgs/JoyFeedback
sensor_msgs/JoyFeedbackArray
sensor_msgs/LaserEcho
zeroconf_msgs/DiscoveredService
~ omitted ~
```

rosmsg show [MESSAGE_NAME]:Показати інформацію про конкретне повідомлення

Це показує інформацію про конкретне повідомлення. Нижче наведено приклад відображення інформації про повідомлення ''turtlesim/Pose'. Ми бачимо, що повідомлення містить п'ять частин інформації змінних типу float32 'x', 'y', 'theta', 'linear_velocity' і 'angular_velocity'.

```
$ rosmsg show turtlesim/Pose
float32 x
float32 y
float32 theta
float32 linear_velocity
float32 angular_velocity
```

rosmsg md5 [MESSAGE_NAME]:Показати md5sum

Нижче наведено приклад перевірки md5-інформації повідомлення "turtlesim/Pose'.

Коли під час передачі повідомлень виникає проблема MD5, вам потрібно буде перевірити md5sum за допомогою цієї команди. Як правило, він зазвичай не використовується. Для отримання додаткової інформації про md5sum зверніться до розділу 4.1 термінологія ROS.

```
$ rosmsg md5 turtlesim/Pose
863b248d5016ca62ea2e895ae5265cf9
```

rosmsg package [PACKAGE_NAME]:Показати список повідомлень, що використовуються в певному пакет

За допомогою цієї команди ми можемо бачити повідомлення, які використовуються в певному пакеті.

```
$ rosmsg package turtlesim
turtlesim/Color
turtlesim/Pose
```

rosmsg packages:Показати список всіх пакетів, що використовують повідомлення

```
$ rosmsg packages
actionlib
actionlib_msgs
actionlib_tutorials
base_local_planner
bond
control_msgs
costmap_2d
~omitted~
```

Розуміння послуг дуже важливо, тому, будь ласка, зверніться до розділу 4.1 термінологія ROS.

Command	Description
rossrv list	Show a list of all services
rossrv show [SERVICE_NAME]	Show information of a specific service
rossrv md5 [SERVICE_NAME]	Show the md5sum
rossrv package [PACKAGE_NAME]	Show a list of services used in a specific package
rossrv packages	Show a list of all packages that use services

Таблиця 8

Давайте закриємо всі вузли перед запуском прикладу, що стосується Службової Інформації ROS. Потім запустіть 'roscore', 'turtlesim_node' і 'turtle_teleop_key' в різних вікнах терміналу, виконавши наступні команди.

```
$ roscore
$ rosrun turtlesim turtlesim_node
$ rosrun turtlesim turtle_teleop_key
```

rossrv list:Показати список всіх послуг

Це команда для перерахування всіх служб в пакетах, встановлених в даний час на ROS. Залежно від пакетів, включених в даний час в ROS, результуюче значення може варіюватися.

```
$ rossrv list
control_msgs/QueryCalibrationState
control_msgs/QueryTrajectoryState
diagnostic_msgs/SelfTest
dynamic_reconfigure/Reconfigure
gazebo_msgs/ApplyBodyWrench
gazebo_msgs/ApplyJointEffort
gazebo_msgs/BodyRequest
gazebo_msgs/DeleteModel
~ omitted ~
```

rossrv show [SERVICE_NAME]:Показати інформацію про конкретну послугу

Нижче наведено приклад відображення Службової Інформації "turtlesim / SetPen'. Ми бачимо, що це сервіс, що містить п'ять частин інформації змінних типу uint8 'r', 'g', 'b', 'width' і 'off'. До Вашого відома, "- - - " використовується в якості рядка, що розділяє запит і відповідь в сервісному файлі, тому в разі "turtlesim / SetPen" ми бачимо, що існує тільки запит і немає вмісту, відповідного відповіді. Додаткові відомості про службові файли див. у розділі 4.3 і практичні приклади наведені в розділі 7.3.

```
$ rossrv show turtlesim/SetPen
uint8 r
uint8 g
uint8 b
uint8 width
uint8 off
---
```

rossrv md5 [SERVICE_NAME]:Показати md5sum

Нижче наведено приклад перевірки інформації md5 сервісу "turtlesim / SetPen'.

Коли під час запиту/відповіді служби виникає проблема MD5, вам потрібно буде перевірити md5sum, і це та команда, яку Ви можете використовувати. Як правило, він зазвичай не використовується.

```
$ rossrv md5 turtlesim/SetPen
9f452acce566bf0c0954594f69a8e41b
```

rossrv package [PACKAGE_NAME]:Показати список послуг, що використовуються в конкретному пакет

Ця команда перераховує служби, що використовуються в певному пакеті.

```
$ rossrv package turtlesim
turtlesim/Kill
turtlesim/SetPen
turtlesim/Spawn
turtlesim/TeleportAbsolute
turtlesim/TeleportRelative
```

rossrv packages:Показати список всіх пакетів, що використовують служби

```
$ rossrv packages
control_msgs
diagnostic_msgs
dynamic_reconfigure
gazebo_msgs
map_msgs
nav_msgs
navfn nodelet
oroca_ros_tutorials
roscpp
sensor_msgs
std_srvs
tf
tf2_msgs
turtlesim
~omitted~
```

У розділі 4.1 термінології ROS було пояснено, що в ROS ми можемо зберігати різні повідомлення в форматі bag і відтворювати їх при необхідності, щоб відтворити ту ж середу при запису даних. Rosbag-це програма, яка створює, відтворює і стискає пакети і має наступні різні функції.

Command	Description
rosbag record [OPTION] [TOPIC_NAME]	Record the message of a specific topic on the bsg file
rosbag info [FILE_NAME]	Check information of a bag file
rosbag play [FILE_NAME]	Play a specific bag file
rosbag compress [FILE_NAME]	Compress a specific bag file
rosbag decompress [FILE_NAME]	Decompresses a specific bag file
rosbag filter [INPUT_FILE] [OUTPUT_FILE] [OPTION]	Create a new bag file with the specific content removed
rosbag reindex bag [FILE_NAME]	Reindex
rosbag check bag [FILE_NAME]	Check if the specific bag file can be played in the current system
rosbag fix [INPUT_FILE] [OUTPUT_FILE] [OPTION]	Fix the bag file version that was saved as an incompatible version

Таблиця 9

Давайте закриємо всі вузли перед запуском прикладу, що стосується інформації журналу ROS. Потім запустіть 'roscore', 'turtlesim_node' і 'turtle_teleop_key' в різних вікнах терміналу, виконавши наступні команди.

```
$ roscore
$ rosrun turtlesim turtlesim_node
$ rosrun turtlesim turtle_teleop_key
```

rosbag record [OPTION][TOPIC_NAME]:Запишіть повідомлення певної теми

По-перше, ми будемо використовувати команду rostopic list для перевірки списку тем, що використовуються в даний час час в мережі ROS.

```
$ rostopic list
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

Як показано в наступному прикладі, з використовуваних тем ми введемо тему, яку хочемо записати, в якості опції при запуску запису файлу мішка. Після того як ми почнемо запис, у вікні терміналу, де працює вузол "turtle_teleop_key", якщо ми керуємо черепахою за допомогою клавіш зі стрілками на клавіатурі, буде записана тема "/ turtle1 / cmd_vel", призначена в якості опції. Потім,

якщо ми натиснемо [Ctrl + c], щоб зупинити запис, буде створено файл bag '2017-07-10-14-16-28.bag’, як показано нижче.

```
$ rosbag record /turtle1/cmd_vel
[INFO] [1499663788.499650818]: Subscribing to /turtle1/cmd_vel
[INFO] [1499663788.502937962]: Recording to 2017-07-10-14-16-28.bag.
```

Якщо ви хочете записати всі теми одночасно, додайте опцію "-a'.

```
$ rosbag record -a
[WARN] [1499664121.243116836]: --max-splits is ignored without --split
[INFO] [1499664121.248582681]: Recording to 2017-07-10-14-22-01.bag.
[INFO] [1499664121.248879947]: Subscribing to /turtle1/color_sensor
[INFO] [1499664121.252689657]: Subscribing to /rosout
[INFO] [1499664121.257219911]: Subscribing to /rosout_agg
[INFO] [1499664121.260671283]: Subscribing to /turtle1/pose
```

rosbag info [bag FILE_NAME]:Перевірте інформацію про файл сумки

За допомогою цієї команди ми можемо перевірити інформацію про файл мішка. Наступний приклад-це записана тема '/ turtle1 / cmd_vel', і було записано 373 повідомлення. Використовуваний тип повідомлення - "geometry_msgs / Twist", і ми також можемо перевірити таку інформацію, як шлях, версія мішка, час і т. д.

```
$ rosbag info 2017-07-10-14-16-28.bag
path:          2017-07-10-14-16-28.bag
version:       2.0
duration:      17.4s
start:         Jul 10 2017 14:16:30.36 (1499663790.36)
end:           Jul 10 2017 14:16:47.78 (1499663807.78)
size:          44.5 KB
messages:      373
compression:   none [1/1 chunks]
types:         geometry_msgs/Twist [9f195f881246fdfa2798d1d3eebca84a]
topics:        /turtle1/cmd_vel   373 msgs    : geometry_msgs/Twist
```

rosbag play [bag FILE_NAME]:Відтворення певного файлу сумки

Наступний приклад-це команда для відтворення записаного файлу '2017-07-10-14-16-28.bag’. Повідомлення "/ turtle1 / cmd_vel " з моменту запису передається точно, і ми бачимо, як черепаха рухається на екрані. Однак той же результат, як показано на рис. 5-5, може бути отримано тільки при перезапуску "turtlesim_node' і ініціалізації траєкторії робота і його положення.

```
$ rosbag play 2017-07-10-14-16-28.bag
[INFO] [1499664453.406867251]: Opening 2017-07-10-14-16-28.bag
Waiting 0.2 seconds after advertising topics... done.
Hit space to toggle paused, or 's' to step.
[RUNNING]  Bag Time: 1499663790.357031   Duration: 0.000000 / 17.419737
[RUNNING]  Bag Time: 1499663790.357031   Duration: 0.000000 / 17.419737
[RUNNING]  Bag Time: 1499663790.357163   Duration: 0.000132 / 17.419737
~ omitted ~
```

Як і на наступному малюнку, ми бачимо, що вихідні дані і дані під час відтворення збігатися.

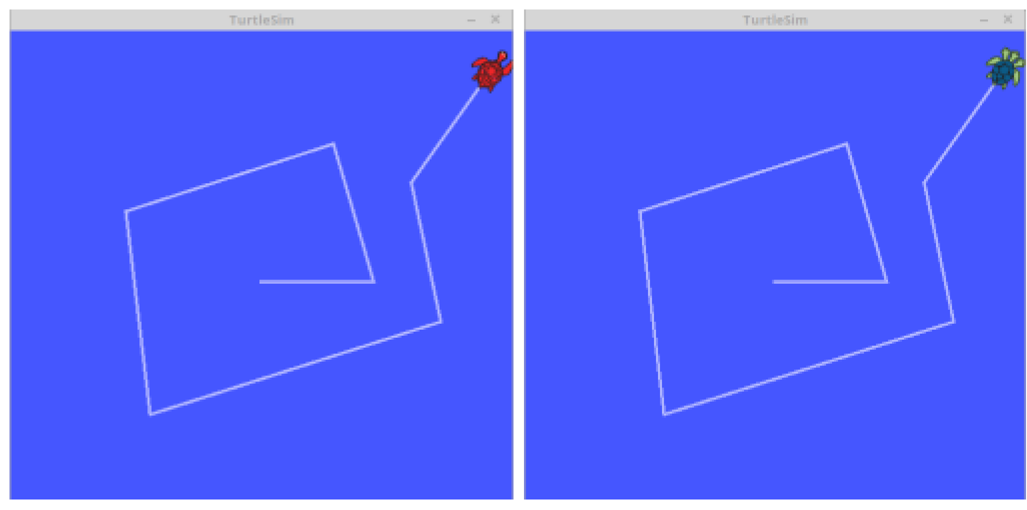


Рис. 44 Приклад гри rosbag

rosbag compress [bag FILE_NAME]:Стиснення певного файлу пакета

Файл мішка, записаний протягом короткого періоду часу, створює файл відносно невеликого розміру, що не є проблемою, але якщо файл мішка записує дані протягом тривалого періоду часу, то він займає багато місця для зберігання. У цьому випадку використовується наступна команда стиснення, і при стисненні вона займе дуже мало місця для зберігання.

```
$ rosbag compress 2017-07-10-14-16-28.bag
2017-07-10-14-16-28.bag    0%     0.0 KB 00:00
2017-07-10-14-16-28.bag    100%   35.0 KB 00:00
```

Файл мішка з наведеного вище прикладу скорочується до чверті, як показано нижче. А вихідний файл перед стисненням зберігається окремо з тегом "orig', доданим до його імені.

```
2017-07-10-14-16-28.bag 12.7kB
2017-07-10-14-16-28.orig.bag 45.5kB
```

rosbag decompress [bag FILE_NAME]:Розпаковує певний файл пакета

Щоб розпакувати стиснутий файл пакета, використовуйте наступну команду. Це відновить файл мішка у початковий стан перед стисненням.

```
$ rosbag decompress 2017-07-10-14-16-28.bag
2017-07-10-14-16-28.bag    0%     0.0 KB 00:00
2017-07-10-14-16-28.bag    100%   35.0 KB 00:00
```

5.5. ROS Catkin Commands

Команди Ros Catkin використовуються при побудові пакета за допомогою системи збірки catkin.

Command	Importance	Description
catkin_create_pkg	★★★	Automatic creation of package
catkin_make	★★★	Build based on catkin build system
catkin_eclipse	★★☆	Modify package created by catkin build system so that it can be used in Eclipse
catkin_prepare_release	★★☆	Cleanup log and tag version during release
catkin_generate_changelog	★★☆	Create or update 'CHANGELOG.rst' file during release
catkin_init_workspace	★★☆	Initialize workspace of the catkin build system
catkin_find	★☆☆	Search catkin

Таблиця 10

catkin_create_pkg:Автоматичне створення пакета

```
catkin_create_pkg [PACKAGE_NAME] [DEPENDENCY_PACKAGE1] [DEPENDENCY_PACKAGE 2] ...
```

catkin_create_pkg-це команда, яка створює порожній пакет, що містить CMakeLists.txt' і ' package.XML ' файли. Для отримання докладних інструкцій, будь ласка, зверніться до розділу 4.9, де пояснюється система збірки ROS. Нижче наведено приклад використання команди 'catkin_create_pkg' для створення пакета 'my_package', який залежить від 'roscpp' і 'std_msgs'.

```
$ catkin_create_pkg my_package roscpp std_msgs
```

catkin_make:Збірка на основі системи збірки catkin

```
catkin_make [OPTION]
```

catkin_make-це команда для складання пакета, створеного користувачем або завантаженого пакет. Нижче наведено приклад побудови всіх пакетів у папці '~ / catkin_ws / src'.

```
$ cd ~/catkin_ws
$ catkin_make
```

Щоб побудувати лише деякі пакунки, а не всі пакунки, виконайте команду ' --pkg [PACKAGE_NAME] ' опція, як показано нижче.

```
$ catkin_make --pkg user_ros_tutorials
```

catkin_eclipse:Змініть пакет, створений системою збірки catkin, щоб його можна було використовувати в Eclipse

catkin_eclipse-це команда для налаштування середовища пакета для керування та програмування за допомогою Eclipse, однієї з інтегрованих середовищ розробки (IDE). Виконання ця команда

створить файли проекту для Eclipse, такі як " ~/catkin_ws/build/.cproject", "~/catkin_ws/build/.project " і т. д. в меню Eclipse, вибравши [Makefile Project with Existing Code] і вибравши ' ~ / catkin_ws / build/', ми можемо керувати всіма пакетами в '~ / catkin_ws/ src ' з Eclipse.

```
$ cd ~/catkin_ws
$ catkin_eclipse
```

catkin_generate_changelog:Створити файл CHANGELOG.rst

catkin_generate_changelog-це команда, яка створює файл CHANGELOG.rst, який реєструє зміни при оновленні версії пакета.

catkin_prepare_release:Управління записами змін і тегами версій при підготовці до випуску

'catkin_prepare_release' - це команда, яка використовується для оновлення ' CHANGELOG.RST', створеного командою 'catkin_generate_changelog'. Команди 'catkin_generate_changelog' і 'catkin_prepare_release ' використовуються при реєстрації створеного пакета в офіційному репозиторії ROS або при оновленні версії зареєстрованого пакета.

catkin_init_workspace:Ініціалізувати робочу папку системи збірки catkin

catkin_init_workspace-це команда для ініціалізації робочої папки користувача (~/catkin_ws / src). Як уже згадувалося в розділі 3.1, за винятком особливих випадків, ця команда виконується тільки один раз під час встановлення ROS.

```
$ cd ~/catkin_ws/src
$ catkin_init_workspace
```

catkin_find:Пошук Catkin, знайти і показати робочий простір

catkin_find-це команда, яка показує робочі папки для кожного проекту.

```
catkin_find [PACKAGE_NAME]
```

Використовуючи команду 'catkin_find', ми можемо дізнатися всі робочі папки, які ми використовуємо. Крім того, якщо ми запустимо 'catkin_find PACKAGE_NAME', він покаже робочі папки, що відносяться до пакету, вказаного в опції, як показано нижче.

```
$ catkin_find
/home/pyo/catkin_ws/devel/include
/home/pyo/catkin_ws/devel/lib
/home/pyo/catkin_ws/devel/share
/opt/ros/kinetic/bin
/opt/ros/kinetic/etc
/opt/ros/kinetic/include
/opt/ros/kinetic/lib
/opt/ros/kinetic/share
```

```
$ catkin_find turtlesim
/opt/ros/kinetic/include/turtlesim
/opt/ros/kinetic/lib/turtlesim
/opt/ros/kinetic/share/turtlesim
```

5.6. ROS Package Commands

Команди пакета ROS використовуються для управління пакетами ROS, наприклад для відображення інформації про пакетах і установки пов'язаних пакетів.

Command	Importance	Command Explanation	Description
ropack	★★★	ros+pack(package)	View information regarding a specific ROS package
rosinstall	★★☆	ros+install	Install additional ROS packages
rosdep	★★☆	ros+dep(dependencies)	Install dependency package of the ROS corresponding package
roslocate	☆☆☆	ros+locate	Show information of ROS package
roscreate-pkg	☆☆☆	ros+create-pkg(package)	Automatic creation of ROS package (used in previous rosbuild system)
rosmake	☆☆☆	ros+make	Build ROS package (used in previous rosbuild system)

Таблиця 11

rospack:Перегляд інформації про конкретний пакет ROS

```
rospack [OPTION] [PACKAGE_NAME]
```

"Rospack' - це команда для відображення такої інформації, як місце збереження, залежність, весь список пакетів відносно конкретного пакета ROS, і ми можемо використовувати такі опції, як " знайти", "список", "залежить від", "залежить", 'профіль' і т. д. як показано в наведеному нижче прикладі, якщо ми задамо ім'я пакета після команди "rospack find", то буде показано збережене місце розташування пакета.

```
$ rospack find turtlesim
/opt/ros/kinetic/share/turtlesim
```

Команда 'rospack list' показує всі пакети на ПК. Об'єднавши команду 'rospack list' з командою пошуку Linux 'grep', ми можемо легко знайти пакет. Наприклад, запуск 'rospack list / grep turtle' буде відображати тільки пакети, пов'язані з turtle, як показано в наведеному нижче прикладі.

```
$ rospack list
actionlib /opt/ros/kinetic/share/actionlib
actionlib_msgs /opt/ros/kinetic/share/actionlib_msgs
actionlib_tutorials /opt/ros/kinetic/share/actionlib_tutorials
amcl /opt/ros/kinetic/share/amcl
angles /opt/ros/kinetic/share/angles
base_local_planner /opt/ros/kinetic/share/base_local_planner
bfl /opt/ros/kinetic/share/bfl
```

```
$ rospack list | grep turtle
turtle_actionlib /opt/ros/kinetic/share/turtle_actionlib
turtle_tf /opt/ros/kinetic/share/turtle_tf
turtle_tf2 /opt/ros/kinetic/share/turtle_tf2
turtlesim /opt/ros/kinetic/share/turtlesim
```

Якщо ми задамо ім'я пакета після команди "rospack depends-on" , вона буде показувати тільки ті пакети, які використовують конкретний пакет, як показано в наступному прикладі.

```
$ rospack depends-on turtlesim
turtle_tf2
turtle_tf
turtle_actionlib
```

Якщо ми задамо ім'я пакета після команди "rospack depends" , вона покаже пакети залежностей, необхідні для запуску конкретного пакета, як показано в наступному прикладі.

```
$ rospack depends turtlesim
cpp_common
rostime
roscpp_traits
roscpp_serialization
genmsg
genpy
message_runtime
std_msgs
geometry_msgs
catkin
gencpp
genlisp
message_generation
```

```
rosbuild
rosconsole
rosgraph_msgs
xmlrpcpp
roscpp
rospack
roslib
std_srvs
```

Команда "профіль rospack" повторно індексує пакет, перевіряючи інформацію про пакет і робочі папки, такі як "/ opt / ros/kinetic / share " або "~ / catkin_ws / src" , в яких зберігаються пакети. Ви можете використовувати цю команду, якщо щойно доданий пакет не вказаний командою " roscd"

```
$ rospack profile
Full tree crawl took 0.021790 seconds.
Directories marked with (*) contain no manifest.  You may
want to delete these directories.
To get just of list of directories without manifests,
re-run the profile with --zombie-only
-------------------------------------------------------------
0.020444   /opt/ros/kinetic/share
0.000676   /home/pyo/catkin_ws/src
0.000606   /home/pyo/catkin_ws/src/ros_tutorials
0.000240 * /opt/ros/kinetic/share/OpenCV-3.2.0-dev
0.000054 * /opt/ros/kinetic/share/OpenCV-3.2.0-dev/haarcascades
0.000035 * /opt/ros/kinetic/share/doc
0.000020 * /opt/ros/kinetic/share/OpenCV-3.2.0-dev/lbpcascades
0.000008 * /opt/ros/kinetic/share/doc/liborocos-kdl
```

rosinstall:Встановлення додаткових пакетів ROS

TH 'rosinstall' - це команда, яка автоматично встановлює або оновлює пакети ROS, керовані системи управління вихідним кодом (SCMS), такі як SVN, Mercurial, Git, Bazaar. Як видно з розділу 3.1, як тільки ми запустимо програму, необхідні пакети будуть автоматично встановлені або оновлені при кожному оновленні пакета.

rosdep:Встановіть пакет залежностей відповідного пакета ROS

```
rosdep [OPTION]
```

"rosdep" - це команда, яка встановлює файл залежностей конкретного пакета. Є такі опції, як "перевірити", "встановити", 'ініціалізувати' і "оновити". Як показано в наступному прикладі, запуск 'rosdep check PACKAGE_NAME' перевірить залежність конкретного пакета. Запуск 'rosdep install PACKAGE_NAME'

встановить пакет залежностей конкретного пакета. Крім того, існують також "rosdep init" або "rosdep update", але докладні інструкції див. у розділі 3.1

```
$ rosdep check turtlesim
All system dependencies have been satisified
$ rosdep install turtlesim
All required rosdeps installed successfully
```

roslocate:Показати інформацію про пакет ROS

```
roslocate [OPTION] [PACKAGE_NAME]
```

"roslocate" - це команда, яка показує таку інформацію, як версія ROS, яка використовується для пакета, тип SCM, розташування репозиторію і т. д. доступні опції: "інформація', 'vcs', 'тип', 'uri', 'repo' і т. д. тут ми розглянемо "інформацію", яка показує всю цю інформацію відразу.

```
$ roslocate info turtlesim
Using ROS_DISTRO: kinetic
- git:
local-name: turtlesim
uri: https://github.com/ros/ros_tutorials.git
version: kinetic-devel
```

roscreate-pkg:Автоматичне створення пакета ROS (використовується в попередній системі rosbuild)

"rosreate-pkg" - це команда, яка автоматично створює пакет, подібний до команди " catkin_create_pkg'. Це команда, яка використовувалася в попередній системі rosbuild, досистеми збірки

catkin. Він був залишений для сумісності версій і не використовується в останніх версія.

rosmake:Build ROS package (використовується в попередній системі rosbuild)

"rosmake" - це команда, яка створює пакет, подібний до команди "catkin_make'. Це команда, яка використовувалася в попередній системі rosbuild, до системи збірки catkin. Він був залишений для сумісності версій і не використовується в останніх версіях.

Розділ 6. Інструменти ROS

Крім команд, представлених в главі 5, існують різні інструменти, які можуть допомогти нам при використанні ROS. Слід зазначити, що ці графічні інструменти доповнюють інструменти командного рядка. Існує досить багато інструментів ROS, включаючи інструменти, які користувачі ROS особисто випустивши. Серед цих інструментів ті, які ми обговоримо в цьому розділі, безпосередньо не обробляють функцію в ROS, але вони є дуже корисними додатковими інструментами для програмування з ROS.

У цьому розділі ми розглянемо наступні інструменти

- **RViz** — 3D visualization tool
- **rqt** — Qt-based ROS GUI development tool
- **rqt_image_view** — Image display tool (a type of rqt)
- **rqt_graph** — A tool that visualizes the correlation between nodes and messages as a graph (a type of rqt)
- **rqt_plot** — 2D data plot tool (a type of rqt)
- **rqt_bag** — GUI-based bag data analysis tool (a type of rqt)

6.1. Інструмент 3D -візуалізації (RViz)

RViz 1 - це інструмент 3D-візуалізації ROS. Основна мета-показати повідомлення ROS в 3D, що дозволить нам візуально перевірити дані. Наприклад, він може візуалізувати відстань від датчика лазерного датчика відстані (LDS) до перешкоди, дані хмари

точок (PCD) 3D - датчика відстані, такого як RealSense, Kinect або Xtion, значення зображення, отримане з камери, і багато іншого без необхідності окремої розробки програмного забезпечення.

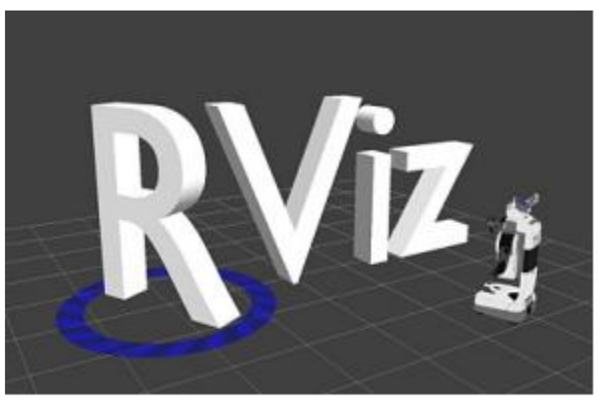


Рис. 45 Завантажувальний екран RViz, інструменту 3D-візуалізації ROS

Він також підтримує різні візуалізації з використанням заданих користувачем полігонів, а інтерактивні маркери 2 дозволяють користувачам виконувати інтерактивні рухи за допомогою команд і даних, отриманих від користувача вузол. Крім того, ROS описує роботів в уніфікованому форматі опису роботів (URDF)3, який виражається у вигляді 3D-моделі, для якої кожна модель може переміщатися або управлятися відповідно до відповідним ступенем свободи, тому вони можуть використовуватися для моделювання або управління. Модель мобільного робота може бути відображена і отримана даними про відстань від лазерного датчика відстані (LDS) може використовуватися для навігації, як показано на рис. 46. RViz також може відображати зображення з камери, встановленої на роботі,

як показано в лівому нижньому кутку рис. 46. В додаток до цього він може отримувати дані від різних датчиків, таких як Kinect, ITS, Real Sense, і візуалізувати їх в 3D, як показано на зображеннях 47, 48 і 49.

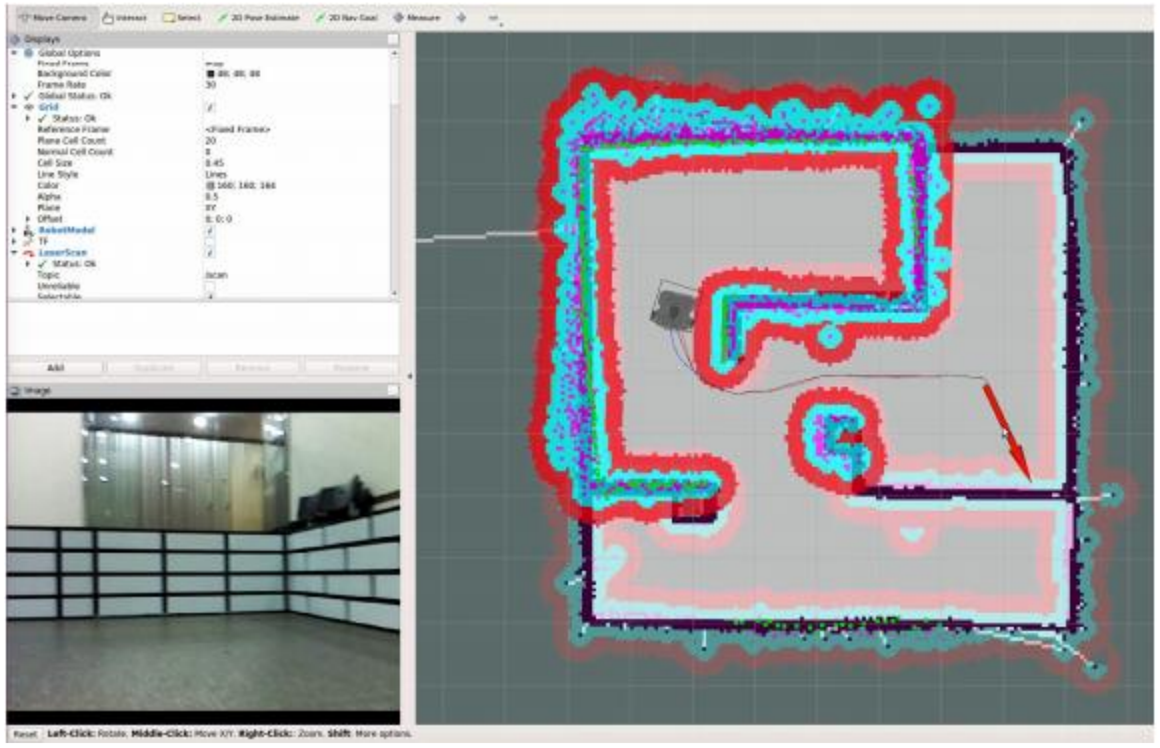


Рис. 46 Приклад 1: навігація за допомогою TurtleBot 3 і датчика IDS

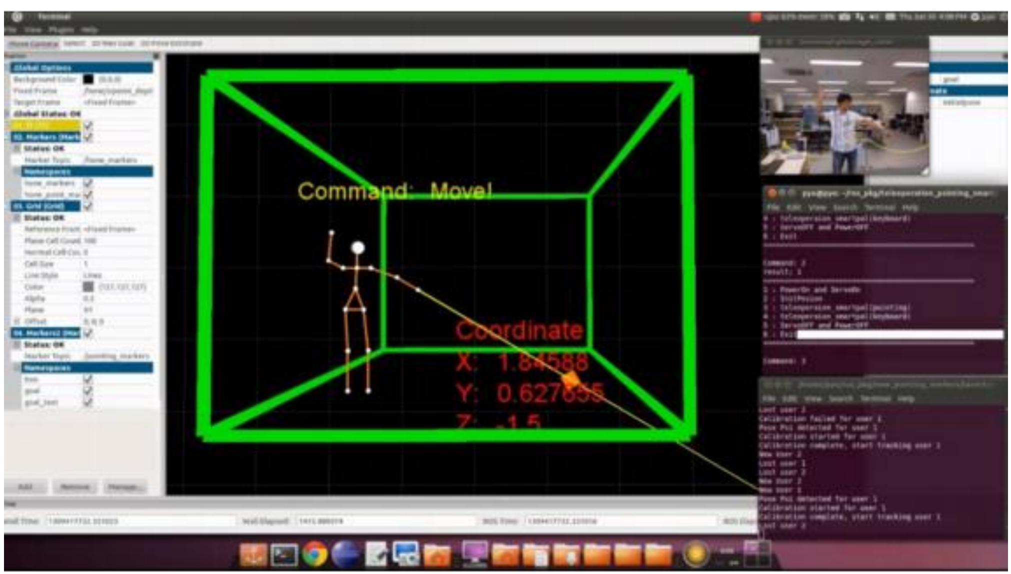


Рис. 47 Rviz Приклад 2: отримання скелета людини за допомогою Kinect і команди з рухом

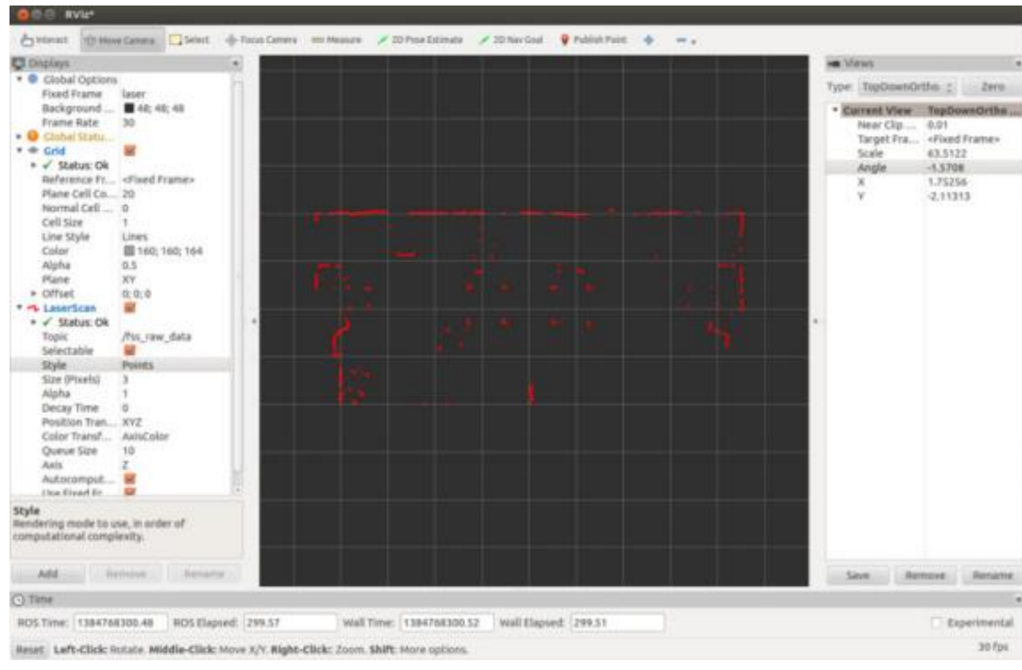


Рис. 48 Rviz приклад 3: Вимірювання відстані за допомогою LDS

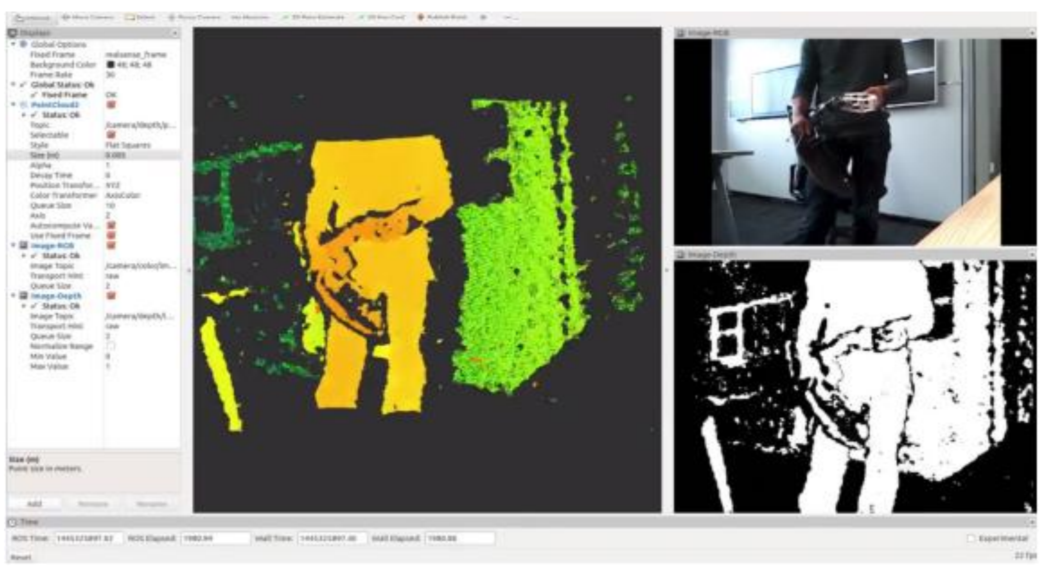


Рис. 49 Rviz приклад 4: значення відстані, інфрачервоного, кольорового зображення, отримане з Intel RealSense

Якщо ви встановили ROS за допомогою команди 'ros - [ROS_DISTRO] - desktop-full', то RViz повинен бути встановлено за замовчуванням. Якщо ви не встановили версію ROS 'desktop-full' або з якоїсь причини RViz Не встановлено, використовуйте наступну команду для установки RViz.

```
$ sudo apt-get install ros-kinetic-rviz
```

Команда виконання RViz виглядає наступним чином. Однак, як і для будь-якого іншого інструменту ROS, роско повинен бути запущений. Для довідки, Ви також можете запустити його з допомогою команди запуску вузла"rosrun rviz rviz'.

```
$ rviz
```

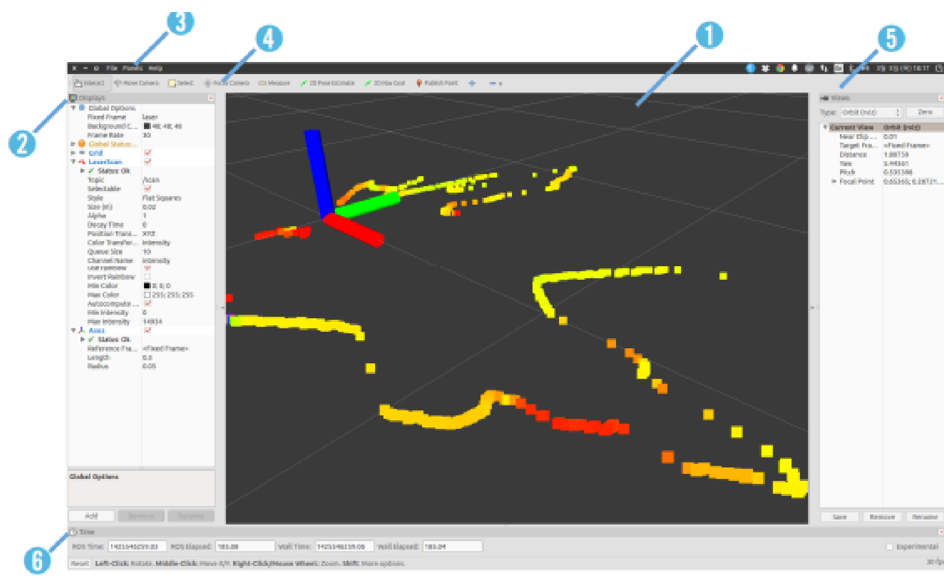


Рис. 50 Склад екрану Rviz

❶ 3D View: ця чорна область розташована в середині екрану. Це головний екран, який дозволяє нам бачити різні дані в 3D. такі параметри, як колір фону 3D-виду, фіксований кадр і сітка, можна налаштувати в глобальних параметрах і настройках сітки в лівій колонці екран.

❷ Displays: панель дисплеїв в лівому стовпці призначена для вибору даних, які ми хочемо відобразити з різних тем. Якщо ми натиснемо кнопку [Додати] в лівому нижньому кутку панелі, з'явиться екран вибору display4, як показано на рис. 6-7. В даний час їх налічується близько 30

різні типи дисплеїв, які ми можемо вибрати, ми розглянемо докладніше в наступному розділ.

❸ Menu: рядок меню розташований у верхній частині екрана. Ми можемо вибирати команди для збереження або завантаження поточних налаштувань дисплея, а також вибирати різні панелі.

❹ Tools: інструменти розташовані під рядком меню, де ми можемо вибрати кнопки для різних функцій, таких як взаємодія, рух камери, вибір, зміна фокусу камери, вимірювання відстані, 2D-оцінка положення, 2D-навігаційна цільова точка, точка публікації.

❺View: Панель "види" налаштовує точку огляду 3D-виду.

■ Orbit: зазначена точка називається фокусом, і орбіта обертається навколо цієї точки. Це значення за замовчуванням і найбільш часто використовуване представлення.

■ FPS(first-person): відображається в режимі перегляду від першої особи.

■ ThirdPersonFollower: це відображається в точці зору третьої особи, яка слідує за певною мета.

■ TopDownOrtho: він використовує вісь Z як основу і відображає ортогональну проекцію об'єктів на площину XY.

■ Xorbit: це схоже на Налаштування за замовчуванням, яке є орбітою, але фокус фіксується на площина XY, де значення координати осі Z фіксовано до нуля.

❻ Time:Час показує поточний час (час стіни), час підйому і минулий час для кожного з них. Це в основному використовується в симуляторах, і якщо є необхідність перезапустити його, просто натисніть кнопку Кнопка [Reset] в самому низу.

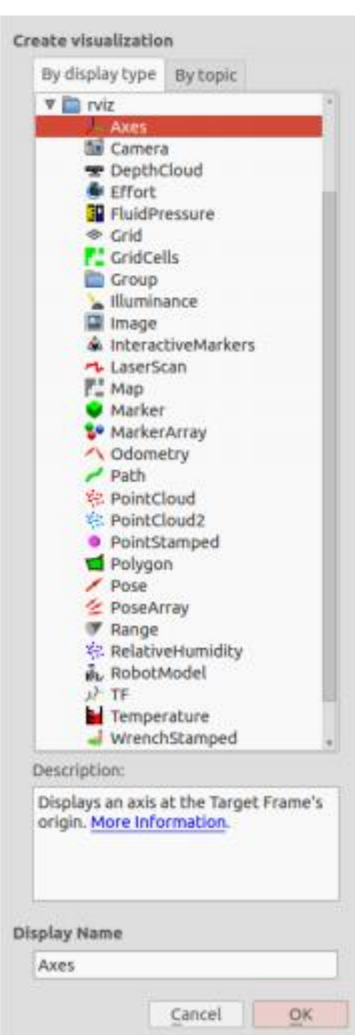


Рис. 51 RViz екран вибору дисплея

Найбільш часто використовуваним меню при використанні RViz, ймовірно, буде меню Displays 5. Це Меню дисплея використовується для вибору повідомлення для відображення на панелі 3D-виду, а опису кожного елемента наведені в таблиці 6-1.

Icon	Name	Description
	Axes	Displays the xyz axes.
	Camera	Creates a new rendering window from the camera perspective and overlays an image on top of it.
	DepthCloud	Displays a point cloud based on the Depth Map. It displays distance values acquired from sensors such as Kinect and Xtion with DepthMap and ColorImage topics as points with overlayed color obtained from the camera.
	Effort	Displays the force applied to each rotary joint of the robot.
	FluidPressure	Displays the pressure of fluids, such as air or water.
	Grid	Displays 2D or 3D grids.
	Grid Cells	Displays each cells of the grid. It is mainly used to display obstacles in the costmap of the navigation
	Group	This is a container for grouping displays. This allows us to manage the displays being used as one group.
	Illuminance	Displays the illuminance.
	Image	Displays the image in a new rendering window. Unlike the Camera display, it does not overlay the camera
	InteractiveMarkers	Displays Interactive Markers. We can change the position (x, y, z) and rotation (roll, pitch, yaw) with the mouse.
	LaserScan	Displays the laser scan value.
	Map	Displays the occupancy map, used in navigation, on top of the ground plane.
	Marker	Displays markers such as arrows, circles, triangles, rectangles, and cylinders provided by RViz.
	MarkerArray	Displays multiple markers.

Icon	Name	Description
	Odometry	Displays the odometry information in relation to the passage of time in the form of arrows. For example, as the robot moves, arrow markers are displayed showing the traveled path in a connected form according to the time intervals.
	Path	Displays the path of the robot used in navigation.
	Point Cloud	Displays point cloud data. This is used to display sensor data from depth cameras such as RealSense, Kinect, Xtion, etc. Since PointCloud2 is compatible with the latest Point Cloud Library (PCL), we can generally use PointCloud2.
	Point Cloud2	
	PointStamped	Displays a rounded point.
	Polygon	Displays a polygon outline. It is mainly used to simply display the outline of a robot on the 2D plane.
	Pose	Displays the pose (location + orientation) on 3D. The pose is represented in the shape of an arrow where the origin of the arrow is the position(x, y, z,) and the direction of the arrow is the orientation (roll, pitch, yaw). For instance, pose can be represented with the position and orientation of the 3D robot model, while it can be represented with the goal point.
	Pose Array	Displays multiple poses.
	Range	This is used to visualize the measured range of a distance sensor such as an ultrasonic sensor or an infrared sensor in the form of a cone.
	RelativeHumidity	Displays the relative humidity.
	RobotModel	Displays the robot model.
	TF	Displays the coordinate transformation TF used in ROS. It is displayed with the xyz axes much like the previously mentioned axes, but each axis expresses the hierarchy with an arrow according to the relative coordinates.
	Temperature	Displays the temperature.
	WrenchStamped	Displays the wrench, which is the torsion movement, in the form of 'arrow' (force) and 'arrow+circle' (torque).

Таблиця 12. Панель дисплеїв Rviz

6.2. Інструмент розробки графічного інтерфейсу ROS (rqt)

Крім інструменту 3D-візуалізації RViz, ROS надає різні графічні інструменти для розробки роботів. Наприклад, є графічний інструмент, який показує ієрархію кожного вузла у вигляді діаграми, тим самим показуючи стан поточного вузла і теми, і інструмент

побудови, який схематизує повідомлення у вигляді 2D-графіка. Починаючи з версії ROS Fuerte, більше 30 інструментів розробки графічного інтерфейсу були інтегровані в якості інструменту під назвою rqt6, який може бути використаний в якості комплексного інструменту графічного інтерфейсу. Крім того, RViz також був інтегрований як плагін art, що робить rqt важливим графічним інструментом для ROS.

Не тільки це, але і, як випливає з назви, rqt був розроблений на основі Qt, який є крос-платформним фреймворком, широко використовуваним для програмування GUI, що робить його дуже зручним для користувачів, щоб вільно розробляти і додавати плагіни. У цьому розділі ми дізнаємося про плагіни rqt 'rqt_image_view', 'rqt_graph', 'rqt_plot' і 'rqt_bag'.

Якщо ви встановили ROS за допомогою команди 'ros - [ROS_DISTRO] - desktop-full', то за замовчуванням буде встановлено rqt. Якщо ви не встановили версію ROS 'desktop-full' або з якоїсь причини не встановлено 'rqt', то наступна команда встановить 'rqt'.

```
$ sudo apt-get install ros-kinetic-rqt*
```

Команда для запуску rqt виглядає наступним чином. Ви можете просто ввести "rqt' на терміналі. Для вашої довідки ми також можемо запустити його за допомогою команди виконання вузла "rosrun rqt_gui rqt_gui'.

```
$ rqt
```

Якщо ми запустимо "rqt", то з'явиться графічний екран rqt, як показано на рис. 52. Якщо він запускається в перший раз, то буде відображатися тільки меню без будь-якого вмісту нижче. Це відбувається тому, що плагін, який є програмою, безпосередньо запускається "rqt", не був вказаний.

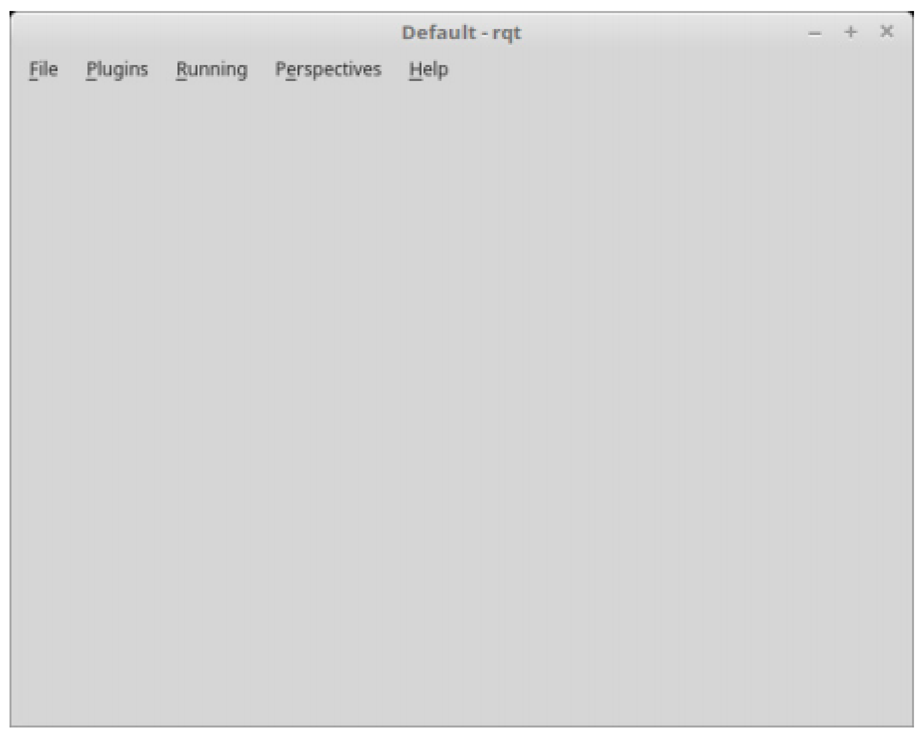


Рис. 52 Початковий екран rqt

Меню rqt виглядає наступним чином.

■ File - Меню Файл містить тільки підменю для закриття "rqt'.

■ Plugins - існує більше 30 плагінів. Виберіть плагін для використання.

■ Running - в даний час запущені плагіни відображаються, і вони можуть бути зупинені, коли вони не потрібні.

■ Perspectives - це меню зберігає робочі плагіни у вигляді набору і використовує їх пізніше для запуску тих ж плагінів.

У меню "rqt' вгорі, якщо ми виберемо [Plugins7 8], ми побачимо близько 30 плагінів. Ці плагіни виконують наступні ролі. Більшість з них є плагінами за замовчуванням 'rqt', які мають дуже корисні функції. Неофіційні плагіни також можуть бути додані, і при необхідності ми можемо додати користувальницькі плагіни " rqt", які ми також розробили для себе.

Action

■ Action Type Browser: це плагін для перевірки структури даних типу дій.

Configuration

■ Dynamic Reconfigure: це плагін для зміни значення параметра вузла.

■ Запуск це графічний плагін roslaunch, який корисний, коли ми не можемо згадати назву або склад roslaunch.

Introspection

■ Node Graph: це плагін для графічного представлення, який дозволяє нам перевірити діаграму відносин поточних запущених вузлів або потоків повідомлень.

■ Package Graph: це плагін для графічного представлення, який відображає залежності пакети.

■ Process Monitor: ми можемо перевірити PID (ідентифікатор процесора), використання процесора, використання пам'яті та кількість потоків поточних запущених вузлів.

Logging

■ Bag: це плагін, що стосується реєстрації даних ROS.

■ Console: це плагін для перевірки попереджень і повідомлень про помилки, що виникають у вузлах на одному екрані.

■ Logger Level: це інструмент для вибору реєстратора, який відповідає за публікацію журналів, і встановлення рівня реєстратора 9 для публікації певного журналу, такого як "Debug", "Info", "Warn", "Error" тощо. 'Fatal'. Це дуже зручно, якщо під час процесу налагодження обрана опція "налагодження".

Miscellaneous Tools

■ Python Console: це плагін для екрану консолі Python.

■Shell: це плагін, який запускає оболонку.

■ Web: це плагін, який запускає веб-браузер.

Robot Tools

■ Controller Manager: це плагін для перевірки стану, типу, інформації про апаратний інтерфейс контролера робота.

■ Diagnostic Viewer: це плагін для перевірки стану робота і помилки.

■ Moveit! Monitor: це плагін для перевірки MOVEit! дані, що використовуються для планування руху.

■ Robot Steering: це графічний інструмент для ручного управління роботом, і цей графічний інструмент корисний для дистанційного керування роботом.

■ Runtime Monitor: це плагін для перевірки попереджень або помилок вузлів в режимі реального часу.

Services

■ Service Caller: це графічний плагін, який підключається до працюючого сервера служби і запитує служби. Це корисно для тестування сервісу.

■ Service Type Browser: це плагін для перевірки структури даних типу служб.

Topics

■ Easy Message Publisher: це плагін, який може публікувати тему в графічному середовищі.

■ Topic Publisher: це графічний плагін, який може публікувати тему. Це корисно для тематичного тестування.

■ Topic Type Browser: це плагін, який може перевіряти структуру даних теми. Це корисно для перевірки типу теми.

■ Topic Monitor: це плагін, який перераховує використовувані в даний момент теми і перевіряє інформацію про обрану тему зі списку.

Visualization

■ Image View: це плагін, який може перевіряти дані зображення з камери. Це корисно для простого тестування даних камери.

■ Navigation Viewer: це плагін для перевірки положення або цільової точки робота в навігації.

■ Plot: це графічний плагін для побудови 2D-даних. Це корисно для схематизації 2D-даних.

■ Pose View: це плагін для відображення пози (положення+орієнтація) моделі робота або TF.

■ RViz: це плагін RViz, який є інструментом для 3D-візуалізації.

■ TF Tree: це плагін графічного типу подання, який показує взаємозв'язок кожного з них. координати, отримані з ТФ в деревоподібній структурі.

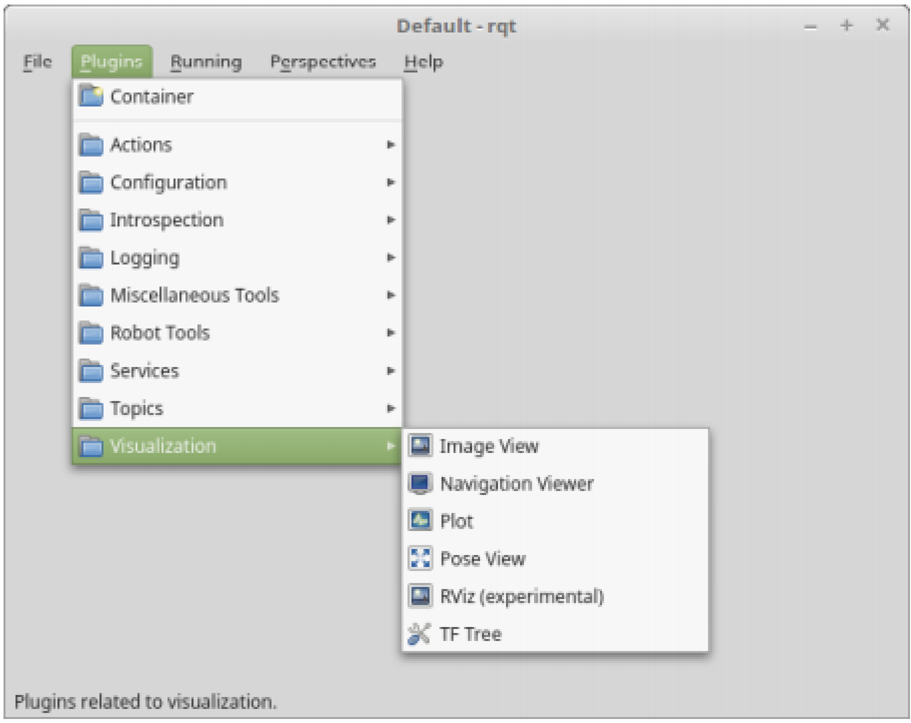


Рис. 53 Плагін rqt

Оскільки важко уявити всі плагіни, в цьому розділі ми дізнаємося про тих, які найбільш часто використовуються, а саме "rqt_image_view", "rqt_bag", 'rqt_graph' і "rqt_plot".

Це плагін 10 застосовується для відображення даних зображення з камери. Хоча це не процес обробки зображення, він все одно досить корисний для простої перевірки зображення. Камера USB зазвичай підтримує UVC, тому ми можемо використовувати пакет "uvc_camera" ROS. Спочатку встановіть пакет 'uvc_camera' за допомогою наведеної нижче команди.

```
$ sudo apt-get install ros-kinetic-uvc-camera
```

Підключіть USB-камеру до USB-порту ПК і запустіть 'uvc_camera_node' в пакеті 'uvc_camera' за допомогою наступної команди.

```
$ rosrun uvc_camera uvc_camera_node
```

Після завершення установки запустіть ' rqt ' за допомогою команди 'rqt', перейдіть в меню і виберіть [Plugins] → [Image View]. У полі вибору повідомлення, розташованому в лівому верхньому кутку, виберіть "/image_raw", щоб побачити зображення, як показано на рис. 6-10. Більш детальна інформація про Датчик камери буде представлена в розділі 8.3.

```
$ rqt
```

Крім вибору плагіна з меню rqt, ми також можемо використовувати спеціальну команду виконання, як показано нижче.

```
$ rqt_image_view
```

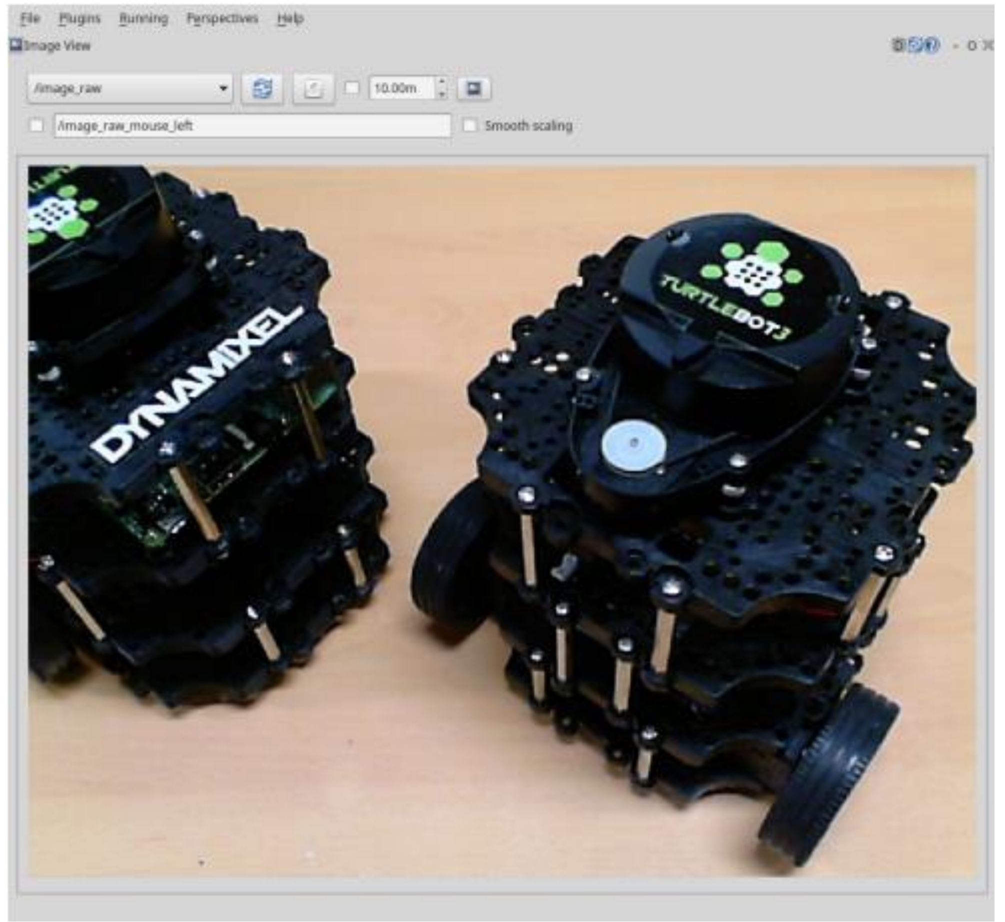


Рис. 54 Перевірка даних зображення USB-камери у вигляді зображення

rqt_graph 11-це інструмент, який показує кореляцію між активними вузлами та повідомленнями, переданими по дорожній мережі, у вигляді діаграми. Це дуже корисно для розуміння поточної структури мережі ROS. Інструкція дуже проста. Наприклад, для перевірки вузлів давайте запустимо 'turtlesim_node' і 'turtle_teleop_key' в пакеті 'turtlesim', описаному в розділі 3.3, і 'uvc_camera_node' в пакеті 'uvc_camera', описаному в розділі 3.3. Розділ 6.2.3. Кожен вузол повинен бути виконаний в окремому терміналі.

```
$ rosrun turtlesim turtlesim_node
$ rosrun turtlesim turtle_teleop_key
$ rosrun uvc_camera uvc_camera_node
$ rosrun image_view image_view image:=image_raw
```

Після виконання всіх вузлів запустіть ' rqt 'за допомогою команди 'rqt' і перейдіть в меню, щоб вибрати [Плагіни] → [Вузловий граф]. До Вашого відома, ми також можемо запустити його за допомогою' rqt_graph ' без необхідності вручну вибирати плагін з меню.

Кореляція між вузлами і темами при запуску rqt_graph показана на рис. 55.

```
$ rqt
```

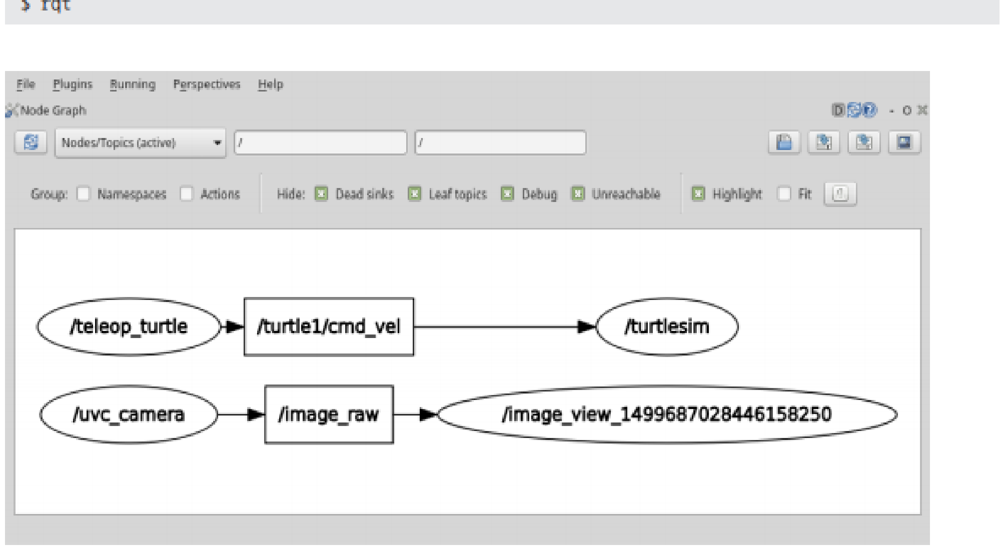


Рис. 55 Приклад rqt_graph

На рис. 55 кола представляють вузли (/teleop_turtle, / turtlesim) і квадрати (/turtle1 / cmd_vel, / image_raw) представляють тематичні повідомлення. Стрілка вказує на передачу повідомлення. У попередньому прикладі, коли ми виконували 'turtle_teleop_key' і 'turtlesim_node', вузол ' teleop_turtle' і вузол 'turtlesim' працювали відповідно. Ми можемо перевірити, що ці два вузла передають дані зі стрілками клавіш клавіатури у вигляді повідомлення про швидкість переміщення і швидкості обертання (назва теми: /turtle 1/cmd_vel).

Ми також можемо перевірити, що вузол 'uvc_camera' в пакеті 'uvc_camera' публікує '/повідомлення теми image_raw і вузол image_view_xxx підписуються на нього. На відміну від цього простого наприклад, фактичне програмування ROS складається з десятків вузлів, які передають різні тематичні повідомлення. У цій ситуації" rqt_graph " стає дуже корисним для перевірки кореляції вузлів в мережі ROS.

На цей раз давайте запустимо 'rqt_plot' з наступною командою замість вибору плагіна з першого меню. Для вашої довідки ми можемо запустити його за допомогою команди виконання вузла ' rosrun rqt_plot rqt_plot'.

```
$ rqt_plot
```

Як тільки "rqt_plot" буде запущений і запущений, натисніть на значок шестерінки в правому верхньому куті програма. Ми можемо вибрати опцію, як показано на рис. 56, де за замовчуванням

використовується параметр "MatPlot'. Крім MatPlot ми також можемо використовувати PyQtGraph і QwtPlot, тому зверніться до відповідного методу установки і використовуйте бібліотеку графіків за вашим вибором.

Наприклад, щоб використовувати PyQtGraph як графік за замовчуванням замість MatPlot, завантажте та встановіть останній файл ' python-pyqtgraph_0.9.xx-x_all.deb ' з адреси завантаження нижче. Якщо установка буде завершена, елемент PyQtGraph буде включений, і ви зможете використовувати PyQtGraph.

- http://www.pyqtgraph.org/downloads/

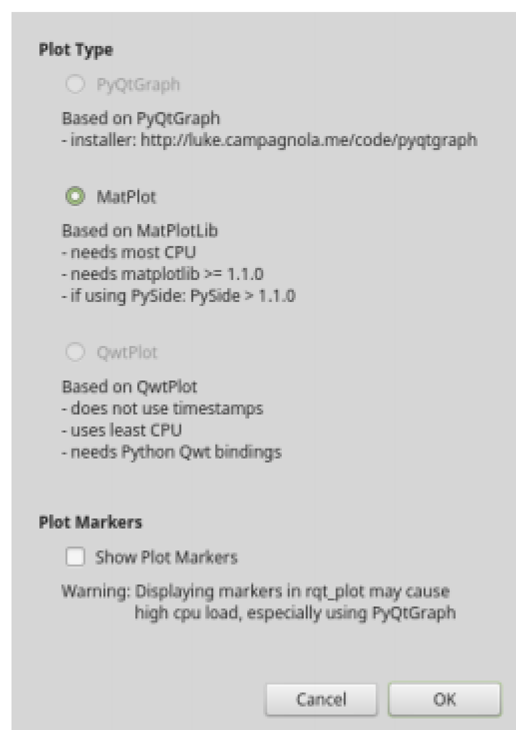


Рис. 56 Опція установки rqt_plot

rqt_plot -це інструмент для побудови 2D-даних. Інструмент побудови графіка отримує повідомлення ROS і відображає їх на 2D-координатах. Як приклад побудуємо координати x і y повідомлення пози вузла"turtlesim". Спочатку нам потрібно запустити' turtlesim_node ' пакета turtlesim.

```
$ rosrun turtlesim turtlesim_node
```

Введіть'/turtle1/ pose / ' в поле Тема у верхній частині інструменту 'rqt_plot', і він намалює '/turtle1/ pose / ' тема на площині 2D (вісь x: час, вісь y: значення даних). Крім того, ми можемо запустити його за допомогою наступної команди, вказавши тему для схематизації.

```
$ rqt_plot /turtle1/pose/
```

Потім запустіть "turtle_teleop_key" в пакеті "turtlesim", щоб ми могли переміщатися по черепасі на екрані.

```
$ rosrun turtlesim turtle_teleop_key
```

Як показано на рис. 56, ми можемо перевірити, що положення x, y, напрямок в тета-просторі, поступальна швидкість і швидкість обертання черепахи нанесені на графік. Як ми бачимо, це корисний інструмент для відображення координат 2D-даних. У цьому прикладі ми використовували turtlesim, але він також корисний для відображення 2D-даних вузлів, розроблених користувачами. Він особливо підходить для відображення значення датчика протягом певного періоду часу, такого як швидкість і прискорення.

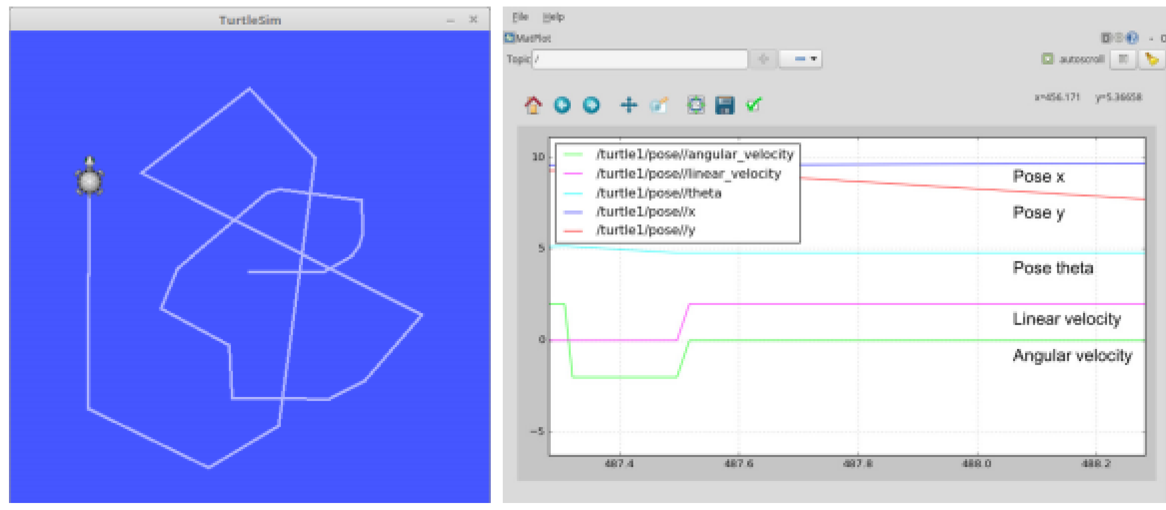


Рис. 56 Приклад rqt_plot

rqt_bag-це графічний інструмент для візуалізації повідомлення. "Росбаг", який ми розглянули в розділі " Розділ 5.4.8 rosbag: Ros log Information 'був текстовим інструментом, але' rqt_bag ' має додану функцію візуалізації, яка дозволяє нам відразу бачити зображення камери, що робить його дуже корисним для управління повідомленнями даних зображення. Перш ніж ми почнемо практикувати, ми повинні запустити все пов'язані вузли "turtlesim" і "uvc_camera", описані в інструментах "rqt_image_view' і 'rqt_graph'. Потім ми створюємо файл сумки з повідомленням' / image_raw ' камери і повідомленням '/turtle1 / cmd_vel' камери. 'turtlesim", використовуючи наступну команду.

У розділі 5.4 ми використовували програму "rosbag" для запису, відтворення і стиснення різних тематичних повідомлень ROS у вигляді файлу bag. "rqt_bag' - це графічна версія раніше представленого " rosbag", і так само, як rosbag, він також може записувати, відтворювати і стискати тематичні повідомлення. Крім того, оскільки це графічна програма, всі команди надаються у вигляді кнопок, тому вони прості у використанні, і ми також можемо дивитися зображення камери відповідно до зміни часу, як відеоредактор.

Як і в наступному прикладі, щоб скористатися функцією "rqt_bag", давайте збережемо зображення USB-камери у вигляді файлу сумки, а потім відтворимо його за допомогою "rqt_bag'.

```
$ rosrun uvc_camera uvc_camera_node
$ rosbag record /image_raw
$ rqt
```

Запустіть ' rqt 'за допомогою команди 'rqt', перейдіть в меню і виберіть [Plugins] → [Logging] →[Bag]. Потім виберіть піктограму завантажувального мішка у формі папки у верхньому лівому куті та завантажте файл"*. bag", який ми тільки що записали. Потім ми зможемо перевірити зображення камери відповідно до зміни часу, як показано на рис. 57. Ми також можемо збільшувати масштаб, відтворювати і перевіряти кількість даних з плином часу, а при клацанні правою кнопкою миші з'явиться опція "Publish", яка дозволить нам знову опублікувати повідомлення.

Рис. 57 Приклад rqt_bag

Тепер ми завершили установку та інструкції інструментів rqt. Оскільки ми не змогли пояснити всі плагіни в цьому розділі, ми рекомендуємо вам спробувати використовувати ці інструменти для себе, посилаючись на кілька прикладів, які ми бачили досі. Хоча ці інструменти можуть не мати прямого відношення до роботів або датчиків, як це роблять вузли ROS, при виконанні цих завдань вони можуть використовуватися в якості корисних додаткових інструментів для збереження, зміни та аналізу даних.

Розділ 7. Основи програмування ROS

Тепер, коли ви познайомилися з ROS, давайте дізнаємося про програмування ROS. Найгарячішими ключовими словами в попередніх розділах були повідомлення, теми, послуги, дії та параметри. Це тому, що ці терміни є ядром АФК. Вузол, що є мінімальною одиницею виконання, обмінюється вхідними і вихідними повідомленнями між вузлами за допомогою обміну повідомленнями по темам, службам, діям і параметрам. У цьому розділі ми познайомимося з програмуванням ROS на практичних прикладах.

7.1. Що потрібно знати перед програмуванням ROS

Повідомлення, що використовуються в ROS, слідують одиницям Сі, найбільш широко використовуваному стандарту в світі. Про це також йдеться в REP-01031. Наприклад, довжина в метрах, маса в кілограмах, час в секундах, струм в амперах, кут в радіанах, частота в Герцах, сила в Ньютонах, потужність у ватах, напруга в Вольтах і температура в Градусах Цельсія. Всі інші одиниці складаються з комбінації вищезазначених одиниць. Наприклад, поступальна швидкість виражається в метрах /сек, а швидкість обертання - в радіанах / сек. У той час як рекомендується використовувати повідомлення наданий ROS, не має значення, якщо ви створюєте абсолютно новий тип повідомлення, який ви перевизначили в міру необхідності. Однак вкрай важливо дотримуватися використання одиниці СІ, так як це дозволить іншим

користувачам використовувати користувальницьке повідомлення без перетворення одиниці.

REP (ROS Enhancement Proposals)

REP-це пропозиція, яка використовується при пропозиції правил, нових функцій і методів управління в спільноті ROS. Він використовується для демократичного створення правил ROS або узгодження змісту, необхідного для розробки, експлуатації та управління ROS. Як тільки пропозиція отримана, багато користувачів ROS можуть переглянути його і послатися на нього як на стандартний документ, створений в результаті взаємної співпраці. Документ REP можна знайти за адресою http://www.ros.org/reps/rep-0000.html.

Осі x, y і z в ROS використовують правило правої руки, як показано на рис.7-1. Фронт-це позитивне напрямок осі x, а вісь представлена червоним кольором (R). Ліва сторона-це позитивний напрямок осі y, а вісь представлена зеленим кольором (G). Нарешті, напрямок вгору є позитивним напрямком осі z, а вісь представлена синім кольором (B). Щоб легко запам'ятати це, ви можете вказати великим, вказівним і середнім пальцями у формі трьох осей. Вказівний палець-це вісь x, середній палець-вісь y, а великий палець-вісь z. порядок, згаданий вище, стає x, y, z, а порядок кольорів-RGB.

Ви можете використовувати праве правило 3 для напрямку обертання робота. Напрямок, в якому згинається ваша права рука, - це позитивний напрямок обертання. Наприклад, якщо робот обертається

від 12 у напрямку 9 годин, використовуючи радіан для кута повороту, робот обертається на + 1,5708 радіана по осі z.

Ці координатні представлення часто використовуються в програмуванні ROS і повинні бути запрограмовані у вигляді x: вперед, y: вліво, z: вгору.

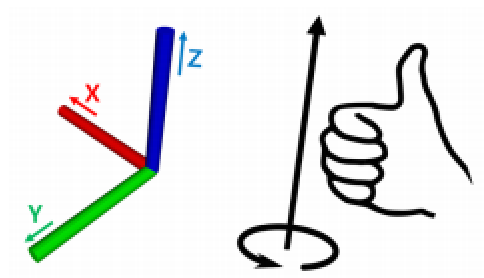


Рис. 58 Осі x, y, z і правило правої руки

ROS рекомендує розробникам дотримуватися посібника зі стилю програмування, щоб максимізувати повторне використання вихідного коду кожної програми. Це скорочує обсяг додаткової роботи, яку часто доводиться виконувати розробникам при роботі з вихідним кодом, покращує розуміння коду іншими співробітниками і полегшує перевірку коду між розробниками. Це не є обов'язковою вимогою, але багато користувачів ROS згодні і дотримуються цього правила. Тому я хотів би настійно закликати користувачів дотримуватися цього правила програмування.

Правила детально описані в Wiki (C++4, Python 5) для кожної мови. В цій книзі основне правило іменування 6 пояснюється нижче,

тому, будь ласка, ознайомтеся з цим правилом до програмування на ROS.

Type	Naming Rule	Example
Package	under_scored	Ex) first_ros_package
Topic, Service	under_scored	Ex) raw_image
File	under_scored	Ex) turtlebot3_fake.cpp
However, messages, services and action file names placed in the /msg and /srv folders follow CamelCased rules when using ROS messages and services. This is because the *.msg, *.srv, and *.action files are converted to header files and then used as structures or types (e.g. TransformStamped.msg, SetSpeed.srv)		
Namespace	under_scored	Ex) ros_awesome_package
Variable	under_scored	Ex) string table_name;
Type	CamelCased	Ex) typedef int32_t PropertiesNumber;
Class	CamelCased	Ex) class UrlTable
Structure	CamelCased	Ex) struct UrlTableProperties
Enumeration Type	CamelCased	Ex) enum ChoiceNumber
Function	camelCased	Ex) addTableEntry();
Method	camelCased	Ex) void setNumEntries(int32_t num_entries)
Constant	ALL_CAPITALS	Ex) const uint8_t DAYS_IN_A_WEEK = 7;
Macro	ALL_CAPITALS	Ex) #define PI_ROUNDED 3.0

Таблиця 13

7.2. Створення та запуск вузлів Publisher та Subscriber

Видавців і передплатників, які використовуються в передачі повідомлень ROS, можна порівняти зпередавачем і приймачем. У ROS передавач називається видавцем, а приймач-передплатник. Цей розділ призначений для створення простого файлу повідомлень, а також створення і запуску вузлів видавця і передплатник.

Наступна команда створює пакет ros_tutorials_topic. Цей пакет залежить від пакетів 'message_generation', 'std_msgs' і 'roscpp', оскільки вони додаються як параметри залежності, за якими слідує ім'я користувача пакета. Задля створення нового повідомлення буде потрібно пакет message_generation. 'std_msgs' - це стандартний пакет повідомлень ROS, а 'roscpp'- клієнтська бібліотека для використання C / C++ в ROS. Ці залежні пакети можуть бути включені при створенні пакета, але вони також можуть бути додані після створення ' package.xml - в папці з посилками.

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg ros_tutorials_topic message_generation std_msgs roscpp
```

Коли пакет створюється, папка пакета 'ros_tutorials_topic' створюється в папці' ~/папка catkin_ws / src. У цій папці пакета ' CMakeLists .txt - і ' package .xml-файли створюються разом з папками за замовчуванням. Ви можете перевірити його за допомогою команди "ls", як показано нижче, або перевірити внутрішню частину пакета за допомогою графічного інтерфейсу Nautilus, який схожий на файл Windows дослідник.

```
$ cd ros_tutorials_topic
$ ls
include          → Header File Folder
src              → Source Code Folder
CMakeLists.txt   → Build Configuration File
package.xml      → Package Configuration File
```

В 'package.XML файл, один з необхідних конфігураційних файлів ROS, являє собою XML-файл, що містить інформацію про

пакет, таку як ім'я пакета, автор, Ліцензія та залежні пакети. Давати відкриємо файл за допомогою редактора (наприклад, gedit, vim, emacs і т. д.) з наступною командою і змінимо його для поточного вузла.

```
$ gedit package.xml
```

Наступний код показує, як змінити ' package.xml-файл, відповідний створеному вами пакету. Особиста інформація буде включена в контент, так що ви можете змінювати її на свій розсуд. Докладний опис кожного варіанту див. у розділі 4.9.

```
ros_tutorials_topic/package.xml
<?xml version="1.0"?>
<package>
  <name>ros_tutorials_topic</name>
  <version>0.1.0</version>
  <description>ROS tutorial package to learn the topic</description>
  <license>Apache License 2.0</license>
  <author email="pyo@robotis.com">Yoonseok Pyo</author>
  <maintainer email="pyo@robotis.com">Yoonseok Pyo</maintainer>
  <url type="bugtracker">https://github.com/ROBOTIS-GIT/ros_tutorials/issues</url>
  <url type="repository">https://github.com/ROBOTIS-GIT/ros_tutorials.git</url>
  <url type="website">http://www.robotis.com</url>
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>message_generation</build_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>std_msgs</run_depend>
  <run_depend>message_runtime</run_depend>
  <export></export>
</package>
```

Catkin, яка є системою збірки ROS, використовує CMake. Тому середовище збірки описана в ' CMakeLists.TXT-файл в папці пакета.

Цей файл налаштовує створення виконуваного файлу, пріоритетну збірку пакета залежностей, Створення посилань і т. д.

```
$ gedit CMakeLists.txt
```

Нижче наведено модифікований код CMakeLists.txt для пакета, який ми створили. См. Розділ 4.9 для докладного опису кожного варіанту.

ros_tutorials_topic/CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
project(ros_tutorials_topic)

## A component package required when building the Catkin.
```

```
## Has dependency on message_generation, std_msgs, roscpp.
## An error occurs during the build if these packages do not exist.
find_package(catkin REQUIRED COMPONENTS message_generation std_msgs roscpp)

## Declaration Message: MsgTutorial.msg
add_message_files(FILES MsgTutorial.msg)

## an option to configure the dependent message.
## An error occurs duing the build if "std_msgs" is not installed.
generate_messages(DEPENDENCIES std_msgs)

## A Catkin package option that describes the library, the Catkin build dependencies,
## and the system dependent packages.
catkin_package(
   LIBRARIES ros_tutorials_topic
   CATKIN_DEPENDS std_msgs roscpp
)

## Include directory configuration.
include_directories(${catkin_INCLUDE_DIRS})

## Build option for the "topic_publisher" node.
## Configuration of Executable files, target link libraries, and additional dependencies.
add_executable(topic_publisher src/topic_publisher.cpp)
add_dependencies(topic_publisher ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
target_link_libraries(topic_publisher ${catkin_LIBRARIES})

## Build option for the "topic_subscriber" node.
add_executable(topic_subscriber src/topic_subscriber.cpp)
add_dependencies(topic_subscriber ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})
target_link_libraries(topic_subscriber ${catkin_LIBRARIES})
```

Наступна опція додається в CMakeLists.TXT файл.

```
add_message_files(FILES MsgTutorial.msg)
```

Наведена вище опція вказує на включення файлу повідомлення ' MsgTutorial.msg', який буде використовуватися у цьому прикладі

вузла при побудові пакета. Оскільки ' MsgTutorial. msg ' ще не створено, давайте створимо файл в наступному порядку:

```
$ roscd ros_tutorials_topic          → Move to package folder
$ mkdir msg                          → Create a new 'msg' folder in the package
$ cd msg                             → Move to the created 'msg' folder
$ gedit MsgTutorial.msg              → Create 'MsgTutorial.msg' file and modify contents
```

Зміст файлу повідомлення досить простий. У повідомленні є тимчасові змінні типу "stamp' і " int32". Крім цих двох типів, такі типи в наявності: основний повідомлення types7 наприклад, 'буль', 'int8', 'типу INT16', 'float32', 'рядок', 'час', 'тривалість', і 'common_msgs' 8, яка являє собою набір повідомлень, які часто використовуються в ROS. У цьому простому ми використовуємо time і int32.

```
ros_tutorials_topic/msg/MsgTutorial.msg

time stamp
int32 data
```

Розділення пакета повідомлень (msg, srv, action)

Зазвичай рекомендується створити окремий пакет для файлу повідомлення "msg' і службового файлу "srv', а не включати файл повідомлення в виконуваний вузол. Це відбувається тому, що, коли вузол передплатника і вузол видавця виконуються на різних комп'ютерах, і вузол видавця, і вузол передплатника залежать від одного і того ж повідомлення. Тому непотрібні вузли повинні бути встановлено, якщо файл повідомлення існує в пакеті. Якщо повідомлення створюється як незалежне пакет, пакет повідомлення може бути доданий до опції залежності, таким чином усуваючи

непотрібні залежності між пакетами. Однак ми включили файл повідомлення у виконуваний вузол в цій книзі, щоб спростити код.

Наступна опція була раніше налаштована в ' CMakeLists.TXT ' файл для створення виконуваного файл:

```
add_executable(topic_publisher src/topic_publisher.cpp)
```

Тобто, " topic_publisher.cpp 'файл вбудований в папку src" для створення виконуваного файлу "topic_publisher". Давайте створимо код, який виконує функції вузла видавця в наступному порядок:

```
$ roscd ros_tutorials_topic/src    → Move to the 'src' folder, which is the source folder of
the package
$ gedit topic_publisher.cpp        → Create or modify new source file
```

ros_tutorials_topic/src/topic_publisher.cpp

```
// ROS Default Header File
#include "ros/ros.h"
// MsgTutorial Message File Header
// The header file is automatically created when building the package.
#include "ros_tutorials_topic/MsgTutorial.h"

int main(int argc, char **argv)            // Node Main Function
{
  ros::init(argc, argv, "topic_publisher"); // Initializes Node Name
  ros::NodeHandle nh;      // Node handle declaration for communication with ROS system

  // Declare publisher, create publisher 'ros_tutorial_pub' using the 'MsgTutorial'
  // message file from the 'ros_tutorials_topic' package. The topic name is
  // 'ros_tutorial_msg' and the size of the publisher queue is set to 100.
  ros::Publisher ros_tutorial_pub =
nh.advertise<ros_tutorials_topic::MsgTutorial>("ros_tutorial_msg", 100);

  // Set the loop period. '10' refers to 10 Hz and the main loop repeats at 0.1 second intervals
  ros::Rate loop_rate(10);

  ros_tutorials_topic::MsgTutorial msg;    // Declares message 'msg' in 'MsgTutorial' message
                                           // file format
  int count = 0;                           // Variable to be used in message

  while (ros::ok())
  {
    msg.stamp = ros::Time::now();  // Save current time in the stamp of 'msg'
    msg.data  = count;             // Save the the 'count' value in the data of 'msg'

    ROS_INFO("send msg = %d", msg.stamp.sec);       // Print the 'stamp.sec' message
```

```
    ROS_INFO("send msg = %d", msg.stamp.nsec);      // Print the 'stamp.nsec' message
    ROS_INFO("send msg = %d", msg.data);            // Print the 'data' message

    ros_tutorial_pub.publish(msg);          // Publishes 'msg' message
    loop_rate.sleep();                      // Goes to sleep according to the loop rate defined above.

    ++count;                                // Increase count variable by one
  }

  return 0;
}
```

Нижче наведено варіант в ' CMakeLists.TXT ' файл для створення виконуваного файлу.

```
add_executable(topic_subscriber src/topic_subscriber.cpp)
```

Це означає, що 'topic_publisher.cpp 'файл побудований для створення виконуваного файлу 'topic_subscriber'. Давайте напишемо код, який виконує функції абонентського вузла в наступному порядку:

```
$ roscd ros_tutorials_topic/src      → Move to the 'src' folder, which is the source folder of
                                       the package
$ gedit topic_subscriber.cpp         → Create or modify new source file
```

ros_tutorials_topic/src/topic_subscriber.cpp

```cpp
// ROS Default Header File
#include "ros/ros.h"
// MsgTutorial Message File Header
// The header file is automatically created when building the package.
#include "ros_tutorials_topic/MsgTutorial.h"

// Message callback function. This is a function is called when a topic
// message named 'ros_tutorial_msg' is received. As an input message,
// the 'MsgTutorial' message of the 'ros_tutorials_topic' package is received.
void msgCallback(const ros_tutorials_topic::MsgTutorial::ConstPtr& msg)
{
  ROS_INFO("recieve msg = %d", msg->stamp.sec);     // Shows the 'stamp.sec' message
  ROS_INFO("recieve msg = %d", msg->stamp.nsec);    // Shows the 'stamp.nsec' message
  ROS_INFO("recieve msg = %d", msg->data);          // Shows the 'data' message
}

int main(int argc, char **argv)                     // Node Main Function
{
  ros::init(argc, argv, "topic_subscriber");        // Initializes Node Name

  ros::NodeHandle nh;      // Node handle declaration for communication with ROS system

  // Declares subscriber. Create subscriber 'ros_tutorial_sub' using the 'MsgTutorial'
  // message file from the 'ros_tutorials_topic' package. The topic name is
  // 'ros_tutorial_msg' and the size of the publisher queue is set to 100.
  ros::Subscriber ros_tutorial_sub = nh.subscribe("ros_tutorial_msg", 100, msgCallback);

  // A function for calling a callback function, waiting for a message to be
  // received, and executing a callback function when it is received
  ros::spin();

  return 0;
}
```

```
$ cd ~/catkin_ws      → Move to Catkin Workspace
$ catkin_make         → Run catkin build
```

Тепер давайте побудуємо файл повідомлення, вузол видавця і вузол передплатника в пакеті ros_tutorials_topic за допомогою наступної команди. Джерело пакета 'ros_tutorials_topic' знаходиться в' ~/catkin_ws/src/ros_tutorials_topic / src', а файл повідомлення пакета 'ros_tutorials_topic' знаходиться в '~ /catkin_ws/src/ros_tutorials_topic / msg'.

Вихідні файли вбудованого пакета будуть знаходитися в папках' /build 'i' / devel 'в папці' ~ / catkin_ws’. Конфігурація, що використовується в catkin build, зберігається в папці "/build", а виконувані файли - в папці "/devel / lib / ros_tutorials_topic", а файл заголовка повідомлення, автоматично згенерований з файлу повідомлення, - в папці "/devel/include / ros_tutorials_topic". Перевірте файли у кожній папці вище, щоб перевірити створений висновок.

А тепер займемося видавцем. Нижче наводиться команда для запуску вузла 'ros_tutorial_msg_publisher' пакета 'ros_tutorials_topic' за допомогою команди 'rosrun'. Обов'язково запустіть " roscore" з іншого терміналу, перш ніж запускати вузол видавця. Відтепер ми будемо вважати, що 'roscore' виконується до виконання вузла.

```
$ roscore
$ rosrun ros_tutorials_topic topic_publisher
```

При запуску видавця ви можете побачити екран виводу, показаний на рис. 58. Однак рядок, що відображається на екрані, є

даними у видавці за допомогою функції ROS_INFO (), яка аналогічна функції printf (), що використовується в поширених мовах програмування. Щоб дійсно опублікувати повідомлення по темі, ми повинні використовувати команду, яка діє як вузол передплатника, такий як вузол передплатника або rostopic.

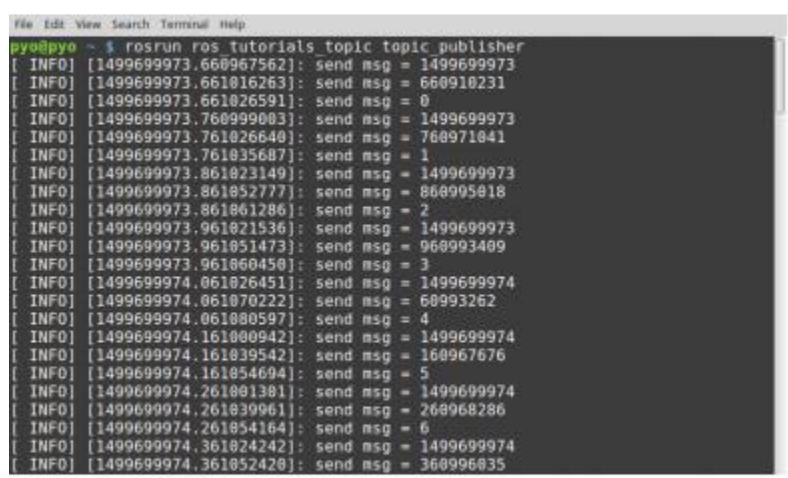


Рис. 58 Екран виконання вузла 'topic_publisher'

Потім давайте перевіримо повідомлення, опубліковане з вузла видавця. Іншими словами, прочитайте повідомлення на тему "ros_tutorial_msg". Ви можете побачити опубліковане повідомлення, як показано на малюнку Рис. 7-3.

```
$ rostopic echo /ros_tutorial_msg
```

```
File Edit View Search Terminal Help
pyo@pyo ~ $ rostopic echo /ros_tutorial_msg
stamp:
  secs: 1499700351
  nsecs: 684514825
data: 1713
---
stamp:
  secs: 1499700351
  nsecs: 784542724
data: 1714
---
stamp:
  secs: 1499700351
  nsecs: 884544453
data: 1715
---
stamp:
  secs: 1499700351
  nsecs: 984543934
data: 1716
---
stamp:
  secs: 1499700352
  nsecs:  84543178
```

Рис. 59 Отримано тему 'ros_tutorial_msg'

Нижче наводиться команда для запуску вузла topic_subscriber пакета ros_tutorials_topic за допомогою команди rosrun для запуску передплатника.

```
$ rosrun ros_tutorials_topic topic_subscriber
```

Коли абонент виконується, екран виводу відображається так, як показано на рис. 60. Отримано опубліковане повідомлення по темі "ros_tutorial_msg", і значення відображається на екрані.

```
File  Edit  View  Search  Terminal  Help
pyo@pyo ~ $ rosrun ros_tutorials_topic topic_subscriber
[ INFO] [1499700485.184875537]: recieve msg = 1499700485
[ INFO] [1499700485.184946471]: recieve msg = 184567102
[ INFO] [1499700485.184957742]: recieve msg = 3048
[ INFO] [1499700485.284812298]: recieve msg = 1499700485
[ INFO] [1499700485.284836776]: recieve msg = 284574255
[ INFO] [1499700485.284844492]: recieve msg = 3049
[ INFO] [1499700485.384811884]: recieve msg = 1499700485
[ INFO] [1499700485.384839629]: recieve msg = 384569171
[ INFO] [1499700485.384849957]: recieve msg = 3050
[ INFO] [1499700485.484795619]: recieve msg = 1499700485
[ INFO] [1499700485.484824179]: recieve msg = 484569717
[ INFO] [1499700485.484838747]: recieve msg = 3051
[ INFO] [1499700485.584792760]: recieve msg = 1499700485
[ INFO] [1499700485.584820628]: recieve msg = 584569677
[ INFO] [1499700485.584830560]: recieve msg = 3052
[ INFO] [1499700485.684824324]: recieve msg = 1499700485
[ INFO] [1499700485.684852121]: recieve msg = 684581217
[ INFO] [1499700485.684861556]: recieve msg = 3053
[ INFO] [1499700485.785495346]: recieve msg = 1499700485
[ INFO] [1499700485.785527583]: recieve msg = 785156898
[ INFO] [1499700485.785552403]: recieve msg = 3054
[ INFO] [1499700485.884855517]: recieve msg = 1499700485
[ INFO] [1499700485.884885781]: recieve msg = 884544763
```

Рис. 60 Екран, що показує виконання вузла topic_subscriber

Далі давайте перевіримо стан зв'язку виконуваних вузлів за допомогою команди " rqt " з розділу 6.2. Ви можете використовувати або команду "rqt_graph", або команду" rqt", як показано нижче. При виконанні 'rqt' виберіть [Plugins] → [Introspection] → [Node Graph] з меню і поточних запущених вузлів і повідомлень можна побачити, як показано на рис. 60.

```
$ rqt_graph or $ rqt
```

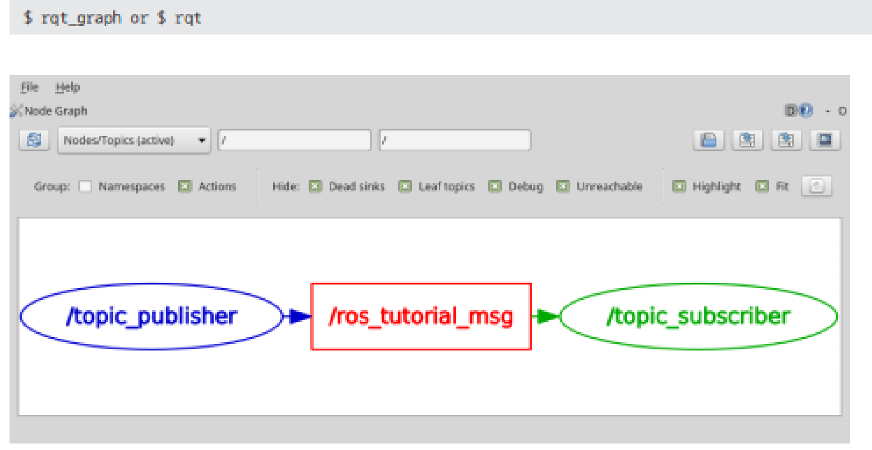


Рис. 1Схема підключення, намальована за допомогою 'rqt_graph'

На малюнку вище ми бачимо, що вузол видавця (topic_publisher) передає тему (ros_tutorial_msg), а тема приймається вузлом передплатника (topic_subscriber).

У цьому розділі ми створили видавець і передплатник вузлів, які використовуються в темізв'язку, і виконали їх, щоб дізнатися, як спілкуватися між вузлами. Приклад джерела можна знайти за наступною адресою github:

- https://github.com/ROBOTIS-GIT/ros_tutorials/tree/master/ros_tutorials_topic

Якщо ви хочете запустити його відразу ж, ви можете клонувати вихідний код за допомогою наступної команди у папці '~ / catkin_ws / src' і побудувати вихідний код. Потім запустіть вузли "topic_publisher" і " topic_subscriber’.

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/ROBOTIS-GIT/ros_tutorials.git
$ cd ~/catkin_ws
$ catkin_make
```

```
$ rosrun ros_tutorials_topic topic_publisher
```

```
$ rosrun ros_tutorials_topic topic_subscriber
```

7.3. Створення та запуск сервісних серверів та клієнтських вузлів

Службу можна розділити на два типи: сервер служби, який відповідає тільки при наявності запиту, і клієнт служби, який може відправляти як запити, так і відповідати на запити. На відміну від теми, СЕРВІС-це одноразове повідомлення. Тому, коли запит і відповідь служби будуть завершені, два підключених вузла будуть відключені.

Ці сервіси часто використовуються при запиті робота на виконання певної дії.

В якості альтернативи він використовується для вузлів, які вимагають, щоб певні події відбувалися за певних умов. Оскільки сервіс є єдиним випадком методу зв'язку, це дуже корисний метод, який може замінити тему з невеликою пропускною здатністю мережі.

У цьому розділі ми створимо простий файл служби і запустимо вузол сервера служби і вузол клієнт служби.

Наступна команда створює пакет 'ros_tutorials_service'. Цей пакет має залежність від пакетів 'message_generation', 'std_msgs' і 'roscpp', , тому опція залежності була додана. Пакет "message_generation" використовується для створення нового повідомлення. Пакет 'std_msgs' є стандартним пакетом повідомлень ROS, а пакет 'roscpp' дозволяє клієнтській бібліотеці використовувати C++ в ROS. Вони можуть бути включені при створенні пакета, але також можуть бути додані після створення ' package.xml ' файл

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg ros_tutorials_service message_generation std_msgs roscpp
```

При створенні пакета в папці "ros_tutorials_service" створюється папка "ros_tutorials_service". ~/папка catkin_ws / src. У цій папці пакета ' CMakeLists .txt - і ' package .xml-файли створюються разом з папками за замовчуванням. Ви можете перевірити його за допомогою команди "ls', як показано нижче.

```
$ cd ros_tutorials_service
$ ls
include             → Header File Folder
src                 → Source Code Folder
CMakeLists.txt      → Build Configuration File
package.xml         → Package Configuration File
```

В 'package.XML-файл-це один з необхідних конфігураційних файлів ROS. Це XML-файл, що містить інформацію про пакет, таку як ім'я пакета, автор, Ліцензія та залежні пакети. Давати відкриємо файл за допомогою редактора (наприклад, gedit, vim, emacs і т. д.) з наступною командою і змінимо його для передбачуваного вузла.

```
$ gedit package.xml
```

Наступний код показує, як змінити ' package.xml-файл для обслуговування пакета, який ви створюєте. Особиста інформація включена в контент, тому ви можете змінювати її в міру необхідності. Докладний опис кожного варіанту див. у розділі 4.9.

```
ros_tutorials_service/package.xml
<?xml version="1.0"?>
<package>
  <name>ros_tutorials_service</name>
  <version>0.1.0</version>
  <description>ROS tutorial package to learn the service</description>
  <license>Apache License 2.0</license>
```

```
  <author email="pyo@robotis.com">Yoonseok Pyo</author>
  <maintainer email="pyo@robotis.com">Yoonseok Pyo</maintainer>
  <url type="bugtracker">https://github.com/ROBOTIS-GIT/ros_tutorials/issues</url>
  <url type="repository">https://github.com/ROBOTIS-GIT/ros_tutorials.git</url>
  <url type="website">http://www.robotis.com</url>
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>message_generation</build_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>std_msgs</run_depend>
  <run_depend>message_runtime</run_depend>
  <export></export>
</package>
```

Система збірки ROS catkin використовує CMake. Таким чином, середовище збірки описана в ‘CMakeLists.TXT-файл в папці пакета. Цей файл налаштовує створення виконуваного файлу, пріоритет складання пакета залежностей, Створення посилань і т. д. відмінність від описаного вище пакета ros_ tutorials_topic полягає в тому, що пакет ros_tutorials_service додає новий вузол сервера служби, вузол клієнта служби і файл служби (*.srv), тоді як пакет ros_tutorials_topic додає вузол видавця, вузол передплатника і файл msg.

```
$ gedit CMakeLists.txt
```

```
ros_tutorials_service/CMakeLists.txt
cmake_minimum_required(VERSION 2.8.3)
project(ros_tutorials_service)

## A component package required when building the Catkin.
## Has dependency on message_generation, std_msgs, roscpp.
## Error occurs during the build if these packages are missing.
find_package(catkin REQUIRED COMPONENTS message_generation std_msgs roscpp)

## Serivce Declaration: SrvTutorial.srv
add_service_files(FILES SrvTutorial.srv)

## Configure the dependent message.
```

```
## An error occurs duing the build if "std_msgs" is not installed.
generate_messages(DEPENDENCIES std_msgs)

## A Catkin package option that describes the library, the Catkin build
## dependencies, and the system dependent packages.
catkin_package(
LIBRARIES ros_tutorials_service
CATKIN_DEPENDS std_msgs roscpp
)

## Configure the directory to Include
include_directories(${catkin_INCLUDE_DIRS})

## Build option for the "service_server" node.
## Configuration of Executable files, target link libraries, and additional
## dependencies.
add_executable(service_server src/service_server.cpp)
add_dependencies(service_server ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
target_link_libraries(service_server ${catkin_LIBRARIES})

## Build option for the "service_client" node.
add_executable(service_client src/service_client.cpp)
add_dependencies(service_client ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
target_link_libraries(service_client ${catkin_LIBRARIES})
```

Наступна опція додається в ' CMakeLists .TXT ' файл.

```
add_service_files(FILES SrvTutorial.srv)
```

Ця опція буде включати в себе 'SrvTutorial.srv ' при побудові пакета, який буде використовуватися в цьому вузлі. Давайте створимо файл 'SrvTutorial.srv' в наступному порядку:

```
$ roscd ros_tutorials_service       → Move to package folder
$ mkdir srv                         → Create a new 'srv' service folder in the package
$ cd srv                            → Move to the created 'srv' folder
$ gedit SrvTutorial.srv             → Create 'SrvTutorial.srv' file and modify contents
```

Давайте створимо тип" int64 "запитів на обслуговування" a " і 'b' і відповідь на обслуговування" result " наступним образ. '- - - "- це роздільник, який розділяє запит і відповідь. Структура аналогічна повідомленням теми, описаної вище, за винятком роздільника" - - -", який розділяє повідомлення запиту і відповіді.

ros_tutorials_service/srv/SrvTutorial.srv

```
int64 a
int64 b
---
int64 result
```

Наступна опція додається в ' CMakeLists.TXT ' файл.

```
add_executable(service_server src/service_server.cpp)
```

Це означає, що 'service_server.cpp ' файл побудований для створення виконуваного файлу "service_server". Давайте створимо код, який виконує функцію вузла сервера служби в наступному порядку:

```
$ roscd ros_tutorials_service/src  → Move to the 'src' folder, which is the source folder of
                                     the package
$ gedit service_server.cpp         → Create or modify the source file
```

ros_tutorials_service/src/service_server.cpp

```
// ROS Default Header File
#include "ros/ros.h"
// SrvTutorial Service File Header (Automatically created after build)
#include "ros_tutorials_service/SrvTutorial.h"

// The below process is performed when there is a service request
// The service request is declared as 'req', and the service response is declared as 'res'
bool calculation(ros_tutorials_service::SrvTutorial::Request &req,
               ros_tutorials_service::SrvTutorial::Response &res)
{
  // The service name is 'ros_tutorial_srv' and it will call 'calculation' function
  // upon the service request.
  res.result = req.a + req.b;
```

```
  // Displays 'a' and 'b' values used in the service request and
  // the 'result' value corresponding to the service response
  ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
  ROS_INFO("sending back response: %ld", (long int)res.result);

  return true;
}

int main(int argc, char **argv)                  // Node Main Function
{
  ros::init(argc, argv, "service_server");       // Initializes Node Name
  ros::NodeHandle nh;                            // Node handle declaration

  // Declare service server 'ros_tutorials_service_server'
  // using the 'SrvTutorial' service file in the 'ros_tutorials_service' package.
  // The service name is 'ros_tutorial_srv' and it will call 'calculation' function
  // upon the service request.
  ros::ServiceServer ros_tutorials_service_server = nh.advertiseService("ros_tutorial_srv",
calculation);

  ROS_INFO("ready srv server!");

  ros::spin();                                   // Wait for the service request

  return 0;
}
```

Для створення виконуваного файлу ‘CMakeLists.txt’ в файл додається наступна опція

```
add_executable(service_client src/service_client.cpp)
```

Коли service_client.cpp 'файл побудований, буде згенерований виконуваний файл "service_client'. Давайте створимо код, який виконує функцію вузла клієнта служби в наступному порядку:

```
$ roscd ros_tutorials_service/src          → Move to the "src" folder, which is the source
                                             folder of the package
$ gedit service_client.cpp                 → Create or modify the source file
```

ros_tutorials_service/src/service_client.cpp

```
#include "ros/ros.h"                          // ROS Default Header File
// SrvTutorial Service File Header (Automatically created after build)
#include "ros_tutorials_service/SrvTutorial.h"
#include <cstdlib>                                    // Library for using the "atoll" function

int main(int argc, char **argv)                       // Node Main Function
{
  ros::init(argc, argv, "service_client");            // Initializes Node Name

  if (argc != 3)                  // input value error handling
  {
    ROS_INFO("cmd : rosrun ros_tutorials_service service_client arg0 arg1");
    ROS_INFO("arg0: double number, arg1: double number");
    return 1;
  }

  ros::NodeHandle nh;      // Node handle declaration for communication with ROS system

  // Declares service client 'ros_tutorials_service_client'
  // using the 'SrvTutorial' service file in the 'ros_tutorials_service' package.
  // The service name is 'ros_tutorial_srv'
  ros::ServiceClient ros_tutorials_service_client =
nh.serviceClient<ros_tutorials_service::SrvTutorial>("ros_tutorial_srv");

  // Declares the 'srv' service that uses the 'SrvTutorial' service file
  ros_tutorials_service::SrvTutorial srv;

  // Parameters entered when the node is executed as a service request value are stored at 'a' and 'b'
  srv.request.a = atoll(argv[1]);
  srv.request.b = atoll(argv[2]);

  // Request the service. If the request is accepted, display the response value
  if (ros_tutorials_service_client.call(srv))
```

```
  {
    ROS_INFO("send srv, srv.Request.a and b: %ld, %ld", (long int)srv.request.a, (long
int)srv.request.b);
    ROS_INFO("receive srv, srv.Response.result: %ld", (long int)srv.response.result);
  }
  else
  {
    ROS_ERROR("Failed to call service ros_tutorial_srv");
    return 1;
  }
  return 0;
}
```

Створіть файл служби, вузол сервера служби і вузол клієнта служби в "ros_tutorials_service'. пакет з наступною командою. Джерело пакета 'ros_tutorials_service' знаходиться в' ~/catkin_ws/src/ros_tutorials_service / src', а файл служби знаходиться в '~/catkin_ws / src / ros_tutorials_service/srv’.

```
$ cd ~/catkin_ws && catkin_make  → Go to the catkin folder and run the catkin build
```

Вихідні дані збірки зберігаються в папках "~ / catkin_ws/build " і "~/catkin_ws / devel". Виконувані файли зберігаються в '~ / catkin_ws / devel / lib / ros_tutorials_service', а конфігурація збірки catkin - в '~ / catkin_ws / build'. Файл заголовка служби, який автоматично генерується з файлу повідомлення, зберігається у файлі "~/catkin_ws / devel/include / ros_tutorials_service'. Перевірте файли в кожному шляху вище, щоб перевірити створений висновок.

Сервер обслуговування, описаний у попередньому розділі, запрограмований на очікування запиту на обслуговування. Тому при

виконанні наступної команди сервер обслуговування буде запущений і буде чекати запиту на обслуговування. Обов'язково запустіть "roscore' перед запуском вузла.

```
$ roscore
$ rosrun ros_tutorials_service service_server
[INFO] [1495726541.268629564]: ready srv server!
```

Після запуску сервера служби запустіть клієнт служби з наступною командою.

```
$ rosrun ros_tutorials_service service_client 2 3
[INFO] [1495726543.277216401]: send srv, srv.Request.a and b: 2, 3
[INFO] [1495726543.277258018]: receive srv, srv.Response.result: 5
```

Параметри 2 і 3, введені за допомогою команди виконання, запрограмовані на передачу як значення запиту на обслуговування. В результаті А і в запитують послугу як значення 2 і 3 відповідно, а сума цих двох значень передається як значення відповіді. В цьому випадку параметр виконання використовується як запит на обслуговування, але насправді він може бути замінений командою або обчислюваним значенням, а змінна для тригера може бути використана в якості запиту на обслуговування.

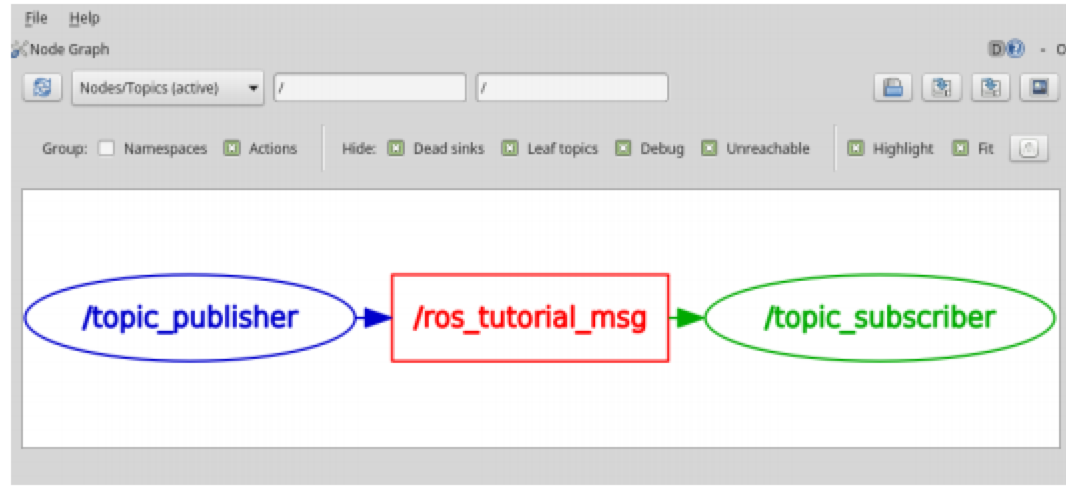


Рис. 61 Topic Publisher (left) and Topic Subscriber (right)

Зверніть увагу, що служба не може бути видно в "rqt_graph", оскільки це одноразова зв'язок, в той час як видавці тем і передплатники підтримують з'єднання, як показано на рис. 61.

Запит на обслуговування може бути виконаний шляхом запуску клієнтського вузла служби, такого як " service_client" з наведеного вище прикладу, але існує також метод, який використовує команду "rosservice call" або "Service Caller "з"rqt". Давайте подивимося, як використовувати "виклик россервісу'.

Напишіть відповідне ім'я Служби, наприклад '/ ros_tutorial_srv', після команди виклику rosservice, як показано в наведеній нижче команді. Далі йдуть необхідні параметри для запиту на обслуговування.

```
$ rosservice call /ros_tutorial_srv 10 2
result: 12
```

У попередньому прикладі ми встановили змінні типу 'int64' ' a ' і 'b' як запит, як показано в сервісному файлі нижче, тому ми ввели '10' і '2' в якості параметрів. Тип "int64' типу "12' повертається як "результат' відповіді служби.

```
int64 a
int64 b
---
int64 result
```

Нарешті, існує метод використання Rqt 'ServiceCaller', який являє собою графічний інструмент. По-перше, давайте запустимо " rqt", інструмент ROS GUI.

```
$ rqt
```

Потім виберіть [Plugins] → [Services] → [Service Caller] з меню програми 'rqt', і з'явиться наступний екран.

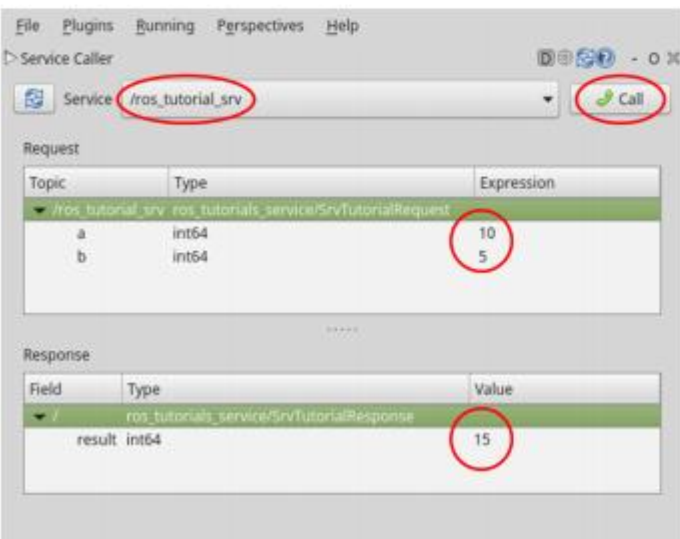


Рис. 62 Запит на обслуговування через плагін rqt 'Service Caller'

Якщо ви виберете ім'я служби в полі Служба вгорі, то побачите інформацію, необхідну для запиту на обслуговування, в полі запит. Щоб запросити послугу, введіть інформацію у вираз кожної інформації запиту. "10' було введено для" а", а "5' було введено для "b". При натисканні на значок <виклик> у вигляді зеленого телефону в правому верхньому куті буде виконано запит на обслуговування, а відповідь в нижній частині екрана покаже "результат' відповіді на обслуговування.

Описаний вище виклик rosservice має ту перевагу, що він виконується безпосередньо на терміналі, але для тих, хто не знайомий з командами Linux або ROS, ми рекомендуємо використовувати rqt 'Service Caller".

У цьому розділі ми створили сервер служби і клієнт служби і виконали їх, щоб дізнатися, як взаємодіяти між вузлами за допомогою служби. Вихідні коди для цього прикладу можна знайти за наступною адресою GitHub

- https://github.com/ROBOTIS-GIT/ros_tutorials/tree/master/ros_tutorials_service

Якщо ви хочете запустити приклад відразу ж, ви можете клонувати вихідний код за допомогою наступної команди в папці '~ / catkin_ws / src' і побудувати його. Потім запустіть вузли "service_server' і ' service_client’.

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/ROBOTIS-GIT/ros_tutorials.git
$ cd ~/catkin_ws
$ catkin_make
```

```
$ rosrun ros_tutorials_service service_server
```

```
$ rosrun ros_tutorials_service service_client 2 3
```

7.4. Writing and Running the Action Server and Client Node

У цьому розділі ми створимо і запустимо вузли action server і action client, а також розглянемо, що є третім методом передачі повідомлень, який ми обговорювали в розділі 4.2. Відмінно від тем і сервісів, дії дуже корисні для асинхронного, двонаправленого і більш складного програмування, де очікується тривалий час відгуку, після обробки запиту і проміжних зворотних зв'язків. Тут ми будемо використовувати приклад "actionlib", представлений в ROS Wiki.

Наступна команда створює пакет ros_tutorials_action. Цей пакет має залежність від пакетів 'message_generation', 'std_msgs', 'actionlib_msgs', 'actionlib', 'roscpp', тому була включена відповідна опція залежності.

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg ros_tutorials_action message_generation std_msgs actionlib_msgs actionlib roscpp
```

Велика частина процесу для цього, включаючи зміну файлу конфігурації пакета (package.xml), дуже схожий на описану вище тему і послугу. За винятком конкретних деталей, які повинні бути згадані в

цьому прикладі, буде надано тільки вихідний код, і ми пропустимо деталі.

```
$ roscd ros_tutorials_action
$ gedit package.xml
```

ros_tutorials_action/package.xml

```
<?xml version="1.0"?>
<package>
  <name>ros_tutorials_action</name>
  <version>0.1.0</version>
  <description>ROS tutorial package to learn the action</description>
  <license>BSD</license>
  <author>Melonee Wise</author>
  <maintainer email="pyo@robotis.com">pyo</maintainer>
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>actionlib</build_depend>
  <build_depend>message_generation</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>actionlib_msgs</build_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>actionlib</run_depend>
  <run_depend>std_msgs</run_depend>
  <run_depend>actionlib_msgs</run_depend>
  <run_depend>message_runtime</run_depend>

  <export></export>
</package>
```

Різниця між цим конфігураційним файлом і попередніми конфігураційними файлами для вузлів " ros_tutorials_topic ' і 'ros_tutorials_service' полягає в розширенні файлу повідомлення. У попередніх прикладах використовувалися файли' msg 'i' srv 'відповідно, а цей пакет' ros_tutorials_action ' додає файл дії (*.action).

Крім того, в якості прикладів вузлів були додані нові вузли сервера дій і клієнтські вузли. Крім того, оскільки ми використовуємо бібліотеку під назвою" Boost " крім ROS, була додана додаткова опція залежності.

```
$ gedit CMakeLists.txt
```

```
add_executable(action_server src/action_server.cpp)
add_dependencies(action_server ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
target_link_libraries(action_server ${catkin_LIBRARIES})

add_executable(action_client src/action_client.cpp)
add_dependencies(action_client ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
target_link_libraries(action_client ${catkin_LIBRARIES})
```

Наступна опція додається в CMakeLists.TXT файл.

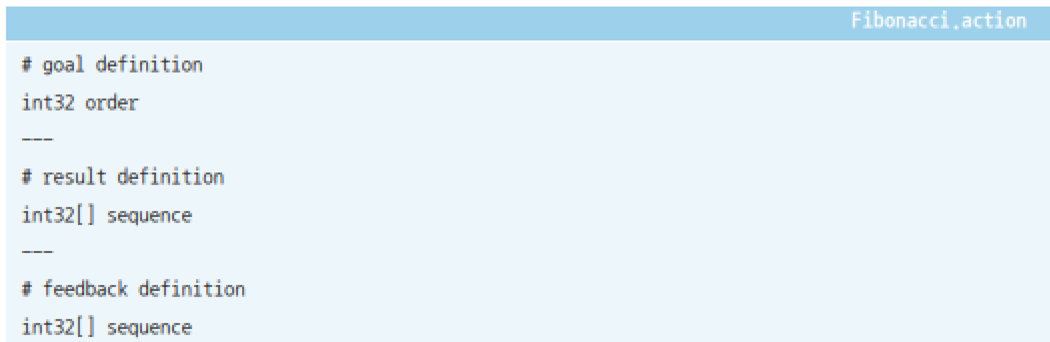
```
add_action_files(FILES Fibonacci.action)
```

Наведений вище параметр вказує на включення служби ".action", яка буде використовуватися в цьому вузлі при створенні пакета. Оскільки файл "Fibonacci.action" ще не створений, давайте створимо його в наступному порядку:

```
$ roscd ros_tutorials_action       → Move to package folder
$ mkdir action                     → Create a new action folder called 'action in the package
$ cd action                        → Move to the created 'action' folder
$ gedit Fibonacci.action           → Create 'Fibonacci.action' file and modify contents
```

У файлі дій три послідовних дефіса (---) використовуються в двох місцях в якості роздільників. Перший розділ-це повідомлення "мета', другий-повідомлення "результат', а третій-повідомлення "зворотній зв'язок". Основна відмінність полягає в тому, що зв'язок між повідомленням "мета" і повідомленням " результат" така ж, як і у вищезгаданому файлі "srv". Однак повідомлення " зворотній зв'язок"

використовується для проміжної передачі зворотного зв'язку під час виконання зазначеного процесу.



Fibonacci.action

```
# goal definition
int32 order
---
# result definition
int32[] sequence
---
# feedback definition
int32[] sequence
```

Five Basic Messages in Action

In addition to the Goal, Result, and Feedback message that can be found in an action file, the action file uses two additional messages: Cancel and Status. The Cancel message uses 'actionlib_msgs/GoalID' as a message that cancels the action execution from the action client or from a separate node while the action is being processed. The Status message can check the status of the current action according to State transitions[11] such as PENDING, ACTIVE, PREEMPTED, and SUCCEEDED[12].

Наступна опція налаштована на ' CMakeLists.TXT ' файл для створення виконуваного файлу:

```
add_executable(action_server src/action_server.cpp)
```

Тобто, 'action_server.cpp "файл побудований для створення виконуваного файлу "action_server". Давати напишемо код, який виконує роль вузла сервера дій, в наступному порядку:

```
$ roscd ros_tutorials_action/src   → Move to the 'src' folder, which is the source folder of
the package
$ gedit action_server.cpp          → Create or modify new source file
```

ros_tutorials_action/src/action_server.cpp

```
#include <ros/ros.h>                                 // ROS Default Header File
#include <actionlib/server/simple_action_server.h>   // action Library Header File
#include <ros_tutorials_action/FibonacciAction.h>    // FibonacciAction Action File Header

class FibonacciAction
{
protected:

  // Node handle declaration
  ros::NodeHandle nh_;

  // Action server declaration
  actionlib::SimpleActionServer<ros_tutorials_action::FibonacciAction> as_;
```

```
  // Use as action name
  std::string action_name_;

  // Declare the action feedback and the result to Publish
  ros_tutorials_action::FibonacciFeedback feedback_;
  ros_tutorials_action::FibonacciResult result_;

public:

  // Initialize action server (Node handle, action name, action callback function)
  FibonacciAction(std::string name) : as_(nh_, name, boost::bind(&FibonacciAction::executeCB,
this, _1), false), action_name_(name)
  {
    as_.start();
  }

  ~FibonacciAction(void)
  {
  }

  // A function that receives an action goal message and performs a specified
  // action (in this example, a Fibonacci calculation).
  void executeCB(const ros_tutorials_action::FibonacciGoalConstPtr &goal)
  {
    ros::Rate r(1);        // Loop Rate: 1Hz
    bool success = true;   // Used as a variable to store the success or failure of an action

    // Setting Fibonacci sequence initialization,
    // add first (0) and second message (1) of feedback.
    feedback_.sequence.clear();
    feedback_.sequence.push_back(0);
    feedback_.sequence.push_back(1);

    // Notify the user of action name, goal, initial two values of Fibonacci sequence
    ROS_INFO("%s: Executing, creating fibonacci sequence of order %i with seeds %i, %i",
action_name_.c_str(), goal->order, feedback_.sequence[0], feedback_.sequence[1]);

    // Action content
```

```
    for(int i=1; i<=goal->order; i++)
    {
      // Confirm action cancellation from action client
      if (as_.isPreemptRequested() || !ros::ok())
      {
        // Notify action cancellation
        ROS_INFO("%s: Preempted", action_name_.c_str());
        as_.setPreempted();       // Action cancellation
        success = false;          // Consider action as failure and save to variable
        break;
      }

  // Store the sum of current Fibonacci number and the previous number in the feedback
  // while there is no action cancellation or the action target value is reached.
  feedback_.sequence.push_back(feedback_.sequence[i] + feedback_.sequence[i-1]);
  as_.publishFeedback(feedback_);          // Publish feedback
  r.sleep();                               // sleep according to the defined loop rate.
}

    // If the action target value is reached,
    // transmit current Fibonacci sequence as the result value.
    if(success)
    {
      result_.sequence = feedback_.sequence;
      ROS_INFO("%s: Succeeded", action_name_.c_str());
      as_.setSucceeded(result_);
    }
  }
};

int main(int argc, char** argv)                     // Node Main Function
{
  ros::init(argc, argv, "action_server");           // Initializes Node Name
  // Fibonacci Declaration(Action Name: ros_tutorial_action)
  FibonacciAction fibonacci("ros_tutorial_action");
  ros::spin();                                      // Wait to receive action goal
  return 0;
}
```

Нижче наведено варіант в ' CMakeLists.TXT ' файл для створення для клієнтського вузла.

```
add_executable(action_client src/action_client.cpp)
```

Це означає, що 'action_client.cpp " файл створений для створення виконуваного файлу "action_client". Давайте напишемо код, який виконує функцію клієнтського вузла дії в наступному порядку:

```
$ roscd ros_tutorials_action/src          → Move to the "src" folder, which is the source
                                            folder of the package
$ gedit action_client.cpp                 → Create or modify new source file
```

ros_tutorials_action/src/action_client.cpp

```
#include <ros/ros.h>                                 // ROS Default Header File
#include <actionlib/client/simple_action_client.h>   // action Library Header File
#include <actionlib/client/terminal_state.h>         // Action Goal Status Header File
#include <ros_tutorials_action/FibonacciAction.h>    // FibonacciAction Action File Header

int main (int argc, char **argv)                     // Node Main Function
{
  ros::init(argc, argv, "action_client");            // Node Name Initialization

  // Action Client Declaration (Action Name: ros_tutorial_action)
  actionlib::SimpleActionClient<ros_tutorials_action::FibonacciAction> ac("ros_tutorial_action",
true);

  ROS_INFO("Waiting for action server to start.");
  ac.waitForServer(); //Wait until action server starts

  ROS_INFO("Action server started, sending goal.");
  ros_tutorials_action::FibonacciGoal goal; // Declare Action Goal
  goal.order = 20;         // Set Action Goal (Process the Fibonacci sequence 20 times)
  ac.sendGoal(goal);       // Transmit Action Goal

  // Set action time limit (set to 30 seconds)
  bool finished_before_timeout = ac.waitForResult(ros::Duration(30.0));
```

```
  // Process when action results are received within the time limit for achieving the action goal
  if (finished_before_timeout)
  {
    // Receive action target status value and display on screen
    actionlib::SimpleClientGoalState state = ac.getState();
    ROS_INFO("Action finished: %s",state.toString().c_str());
  }
  else
    ROS_INFO("Action did not finish before the time out.");  // If time out occurs

  //exit
  return 0;
}
```

Створіть файл дії, вузол сервера дій і вузол клієнта дій в "ros_tutorials_action". пакет з наступною командою. Джерело пакета 'ros_tutorials_action' знаходиться в' ~/catkin_ws/src/ros_tutorials_action / src', а файл дії знаходиться в " ~/catkin_ws / src/ros_tutorials_action/src/action".

```
$ cd ~/catkin_ws && catkin_make  → Go to the catkin folder and run the catkin build
```

Сервер дій, описаний у попередньому розділі, запрограмований на очікування без будь-якої обробки до тих пір, поки не буде досягнута "мета" дії. Тому при виконанні наступної команди сервер дій очікує встановлення "цілі". Обов'язково запустіть 'roscore' перед запуском вузла.

```
$ roscore
$ rosrun ros_tutorials_action action_server
```

Дія подібна до послуги, описаної в розділі 4.3, в тому, що існує дія "мета" і "результат", відповідні "запиту' і"відповіді". Однак дії мають повідомлення " зворотного зв'язку", відповідні проміжного

зворотного зв'язку в процесі. Це може виглядати як служба, але дуже схоже на Тему в її фактичному способі передачі повідомлень. Використання поточного повідомлення про дію можна перевірити за допомогою команд "rqt_graph" і "rostopic list", як показано нижчий.

```
$ rostopic list
/ros_tutorial_action/cancel
/ros_tutorial_action/feedback
/ros_tutorial_action/goal
/ros_tutorial_action/result
/ros_tutorial_action/status
/rosout
/rosout_agg
```

Для отримання додаткової інформації про кожне повідомлення додайте опцію "- v " до списку ростопіків. Це розділить теми, які будуть опубліковані і підписані, наступним чином:

```
$ rostopic list -v
Published topics:
 * /ros_tutorial_action/feedback [ros_tutorials_action/FibonacciActionFeedback] 1 publisher
 * /ros_tutorial_action/status [actionlib_msgs/GoalStatusArray] 1 publisher
 * /rosout [rosgraph_msgs/Log] 1 publisher
 * /ros_tutorial_action/result [ros_tutorials_action/FibonacciActionResult] 1 publisher
 * /rosout_agg [rosgraph_msgs/Log] 1 publisher

Subscribed topics:
 * /ros_tutorial_action/goal [ros_tutorials_action/FibonacciActionGoal] 1 subscriber
 * /rosout [rosgraph_msgs/Log] 1 subscriber
 * /ros_tutorial_action/cancel [actionlib_msgs/GoalID] 1 subscriber
```

Щоб візуально перевірити інформацію, використовуйте команду "rqt_graph", показану нижче. Рис. 63 показує зв'язок між сервером дій і клієнтом, а також повідомлення про дії, які передаються і приймаються двонаправлено. Тут повідомлення про дію представлено ім'ям "ros_tutorial_action / action_topics". При виборі дії

в меню групи всі п'ять повідомлень, що використовуються в дії, можна побачити, як показано на рис. 64. Тут ми бачимо, що дія в основному складається з 5 тем і вузлів, які публікують і підписуються на цю тему.

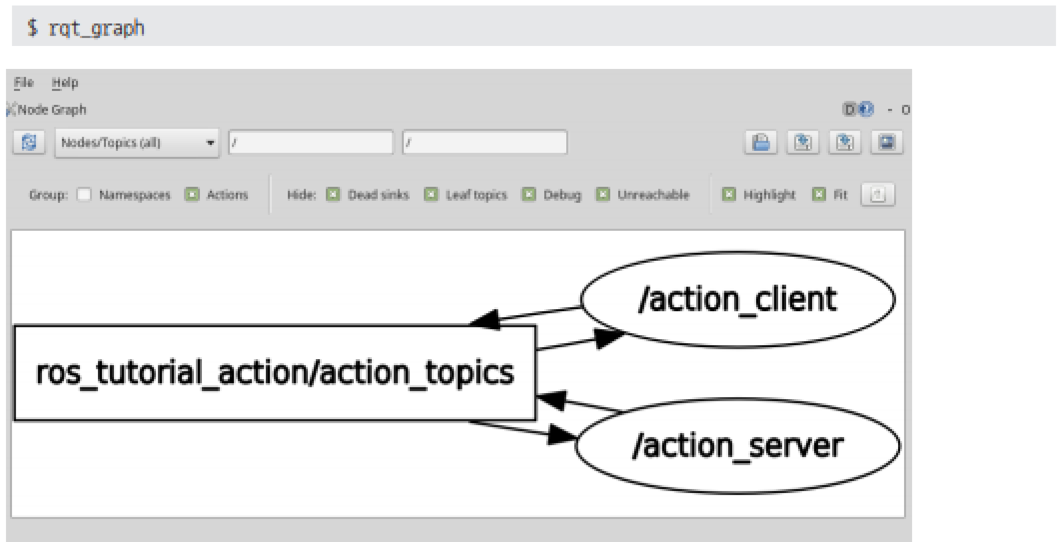


Рис. 63 Зв'язок між повідомленням про дію, яке передається і приймається двонаправлено, сервером дій і клієнтом

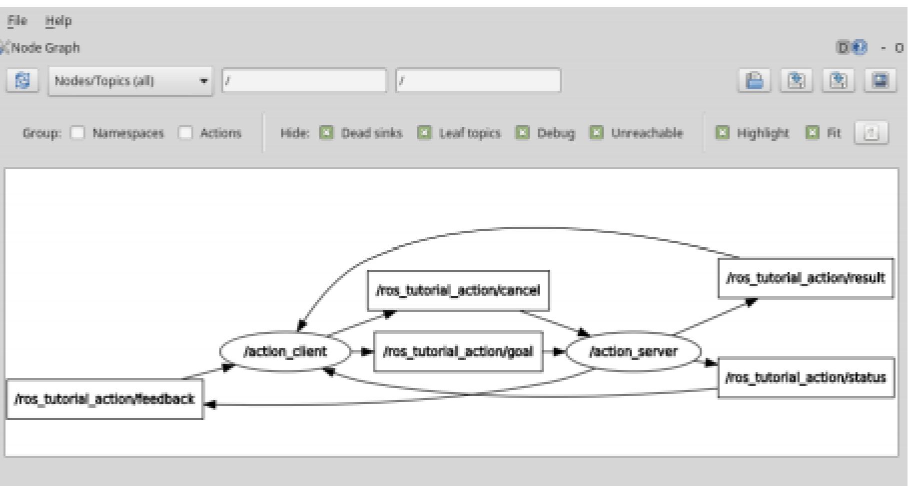


Рис. 64 Повідомлення, що використовуються в дії

Клієнт дій запускається за допомогою наступної команди. При запуску клієнта дій повідомлення "мета дії" встановлюється рівним 20.

```
$ rosrun ros_tutorials_action action_client
```

Встановивши цільове значення, сервер дій запустить послідовність Фібоначчі наступним чином. Команда rostopic 'rostopic echo / ros_tutorial_action / feedback" може використовуватися для отримання додаткових значень або результатів зворотного зв'язку.

```
$ rosrun ros_tutorials_action action_server
 [INFO] [1495764516.294367721]: ros_tutorial_action: Executing, creating fibonacci sequence of
order 20 with seeds 0, 1
 [INFO] [1495764536.294488991]: ros_tutorial_action: Succeeded
```

```
$ rosrun ros_tutorials_action action_client
 [INFO] [1495764515.999158825]: Waiting for action server to start.
 [INFO] [1495764516.293575887]: Action server started, sending goal.
 [INFO] [1495764536.295139830]: Action finished: SUCCEEDED
```

```
$ rostopic echo /ros_tutorial_action/feedback
header:
   seq: 42
   stamp:
     secs: 1495764700
     nsecs: 413836908
   frame_id: ''
 status:
   goal_id:
     stamp:
       secs: 1495764698
       nsecs: 413136891
     id: /action_client-1-1495764698.413136891
   status: 1
   text: This goal has been accepted by the simple action server
 feedback:
   sequence: [0, 1, 1, 2, 3]
 ---
```

У цьому розділі ми створили сервер дій і клієнтські вузли дій і виконали їх, щоб дізнатися, як взаємодіяти між вузлами. Відповідні джерела можна знайти в наступних розділах.

Адреса GitHub:

■ https://github.com/ROBOTIS-GIT/ros_tutorials/tree/master/ros_tutorials_action

Якщо ви хочете відразу запустити приклад, ви можете клонувати вихідний код за допомогою наступної команди в папці "catkin_ws / src" і створити пакет. Потім запустіть вузли "action_server' і ‘action_client’.

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/ROBOTIS-GIT/ros_tutorials.git
$ cd ~/catkin_ws
$ catkin_make
```

```
$ rosrun ros_tutorials_action action_server
```

```
$ rosrun ros_tutorials_action action_client
```

7.5. Using Parameters

Концепція параметра була описана кілька разів до цих пір зі списком параметрів.

Тому в цьому розділі давайте дізнаємося, як використовувати параметри на практиці. Посилатися на Розділ 4.1 для термінології параметрів і розділ 5.4 для команди ‘rospram.

Давайте змінимо ' service_server.cpp ' джерело на сервері служби і клієнтський вузол, створений в Розділ 7.3 використання

параметрів для виконання арифметичних операцій, а не просто додавання двох значень, введених в якості запиту на обслуговування. Змінити ' service_server.cpp-джерело в наступному порядку.

```
int g_operator = PLUS;

// The process below is performed if there is a service request
// The service request is declared as 'req', and the service response is declared as 'res'
bool calculation(ros_tutorials_service::SrvTutorial::Request &req,
                 ros_tutorials_service::SrvTutorial::Response &res)
{
  // The operator will be selected according to the parameter value and calculate 'a' and 'b',
  // which were received upon the service request.
  // The result is stored as the Response value.
  switch(g_operator)
  {
    case PLUS:
      res.result = req.a + req.b; break;
    case MINUS:
      res.result = req.a - req.b; break;
    case MULTIPLICATION:
      res.result = req.a * req.b; break;
    case DIVISION:
      if(req.b == 0)
      {
        res.result = 0; break;
      }
      else
      {
        res.result = req.a / req.b; break;
      }
    default:
      res.result = req.a + req.b; break;
  }

  // Displays the values of 'a' and 'b' used in the service request, and the 'result' value
  // corresponding to the service response.
  ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
  ROS_INFO("sending back response: [%ld]", (long int)res.result);
  return true;
}

int main(int argc, char **argv)                    // Node Main Function
```

```
{
  ros::init(argc, argv, "service_server");          // Initializes Node Name
  ros::NodeHandle nh;                               // Node handle declaration
  nh.setParam("calculation_method", PLUS);          // Reset Parameter Settings

  // Declare service server 'service_server' using the 'SrvTutorial' service file
  // in the 'ros_tutorials_service' package. The service name is 'ros_tutorial_srv' and
  // it is set to execute a 'calculation' function when a service is requested.
  ros::ServiceServer ros_tutorial_service_server = nh.advertiseService("ros_tutorial_srv",
calculation);
  ROS_INFO("ready srv server!");
  ros::Rate r(10);         // 10hz
  while (1)
  {
    // Select the operator according to the value received from the parameter.
    nh.getParam("calculation_method", g_operator);
    ros::spinOnce();  // Callback function process routine
    r.sleep();        // Sleep for routine iteration
   }
  return 0;
}
```

Оскільки більша частина вмісту аналогічна попереднім прикладам, давайте просто поглянемо на додаткові частини, необхідні для використання параметрів. Зокрема, методи "setParam" і " getParam", виділені жирним шрифтом, є найбільш важливими частинами при використанні параметрів. Оскільки це дуже прості методи, їх можна легко зрозуміти, просто прочитавши їх використання.

Наступний код встановлює параметр "calculation_method" В "PLUS". Оскільки слово "PLUS" визначено як " 1 " в коді в розділі 7.5.1, параметр "calculation_method" стає "1", і відповідь служби додасть отримані значення з запиту служби.

```
nh.setParam("calculation_method", PLUS);
```

Зверніть увагу, що параметри можуть бути встановлені в "цілі числа", "плаваючі", "логічні", "рядкові", "словники", "список" і так далі. Наприклад, "1' - це "ціле число", " 1.0 "- це "плаваюче", "internetofthings" - це" рядок"," true " - це логічне значення, "[1,2,3] "- це список цілих чисел, а "a: b, c: d' - це словник.

Наступне отримує значення параметра з "calculation_method" і встановлює його в якості значення " g_operator". В результаті" g_operator " з коду в розділі 7.5.1 перевіряє значення параметра кожні " 0,1 " секунди, щоб визначити, яку операцію використовувати для значень, отриманих через запит на обслуговування.

```
nh.getParam("calculation_method", g_operator);
```

Перебудуйте вузол сервера служби в пакеті "ros_tutorials_service" за допомогою наступної команди:

```
$ cd ~/catkin_ws && catkin_make
```

Коли збірка буде завершена, запустіть вузол" service_server "пакета" ros_tutorials_service ' з наступною командою.

```
$ roscore
$ rosrun ros_tutorials_service service_server
[INFO] [1495767130.149512649]: ready srv server!
```

Команда "список роспарама" відображає список параметрів, що використовуються в даний час в мережі ROS. У відображеному списку ми використовували параметр "/calculation_method".

```
$ rosparam list
/calculation_method
/rosdistro
/rosversion
/run_id
```

Встановіть параметри відповідно до наступної команди і переконайтеся, що обробка служби змінилася при кожному запиті однієї і тієї ж служби.

```
$ rosservice call /ros_tutorial_srv 10 5    → Input variables a and b for arithmetic operation
result: 15                                  → Resulting value from the default operation
$ rosparam set /calculation_method 2        → Subtraction
$ rosservice call /ros_tutorial_srv 10 5
result: 5
$ rosparam set /calculation_method 3        → Multiplication
$ rosservice call /ros_tutorial_srv 10 5
result: 50
$ rosparam set /calculation_method 4        → Division
$ rosservice call /ros_tutorial_srv 10 5
result: 2
```

Параметр 'calculation_method' можна змінити за допомогою команди 'rosparam set'. Зі зміненими параметрами ви можете побачити різні значення результатів з одним і тим же введенням " rosservice call /ros_tutorial_srv 10 5’. Як показано в наведеному вище прикладі, параметри в ROS можуть керувати потоком, налаштуванням і обробкою вузлів ззовні вузла. Це дуже корисна функція, тому ознайомтеся з цією функцією, навіть якщо вона вам не потрібна відразу

У цьому розділі ми змінили сервер служби і дізналися, як використовувати параметри.

Відповідний вихідний код був перейменований в пакет "ros_tutorials_parameter", щоб відрізнити його від раніше створеного вихідного коду служби, і його можна знайти в наступному

Адреса GitHub.

■ https://github.com/ROBOTIS-GIT/ros_tutorials/tree/master/ros_tutorials_parameter

Якщо ви хочете відразу запустити приклад, ви можете клонувати вихідний код за допомогою наступної команди в папці "catkin_ws / src" і створити пакет. Потім запустіть вузли "service_server' і "service_client".

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/ROBOTIS-GIT/ros_tutorials.git
$ cd ~/catkin_ws
$ catkin_make
```

```
$ rosrun ros_tutorials_parameter service_server_with_parameter
```

```
$ rosrun ros_tutorials_parameter service_client_with_parameter 2 3
```

7.6. Using roslaunch

'rosrun' - це команда, яка виконує лише один вузол, а "roslaunch" може запускати більше одного вузол. Інші функції команди "roslaunch" включають в себе можливість змінювати параметри пакета, перейменовувати ім'я вузла, Налаштування

ROSE_ROOT android_PACKAGE_PALPATION_PATH і змінювати змінні середовища.

"roslaunch' використовує файл"*. launch " для вибору виконуваних вузлів, який заснований на XML і надає Параметри, що залежать від тегів. Команда виконання - " roslaunch [ім'я пакета] [файл roslaunch]".

Щоб дізнатися, як використовувати roslaunch, перейменуйте раніше створені вузли "topic_publisher" і " topic_subscriber". Немає сенсу змінювати лише Імена, тому давайте запустимо два набори вузлів видавця і передплатника для зв'язку один з одним.

Спочатку напишіть файл"*. launch’. Файл, який використовується для roslaunch, має ім'я файлу розширення"*. launch",і ви повинні створити папку "launch' в папці пакета і помістити файл запуску в цю папка. Створіть папку за допомогою наступної команди і створіть новий файл з ім'ям ' union.запуск’.

```
$ roscd ros_tutorials_topic
$ mkdir launch
$ cd launch
$ gedit union.launch
```

Напишіть зміст "Союзу". Запустіть файл наступним чином.

```
union.launch
<launch>
  <node pkg="ros_tutorials_topic" type="topic_publisher" name="topic_publisher1"/>
  <node pkg="ros_tutorials_topic" type="topic_subscriber" name="topic_subscriber1"/>
  <node pkg="ros_tutorials_topic" type="topic_publisher" name="topic_publisher2"/>
  <node pkg="ros_tutorials_topic" type="topic_subscriber" name="topic_subscriber2"/>
</launch>
```

Теги, необхідні для запуску вузла за допомогою команди "roslaunch", описані в

тега <launch>. Тег < node> описує вузол, який повинен бути виконаний "roslaunch'. Варіанти включають в себе 'pkg', ' type' і 'name’'.

■ `*pkg*` - ім'я пакета

■ *"type"* - ім'я фактичного вузла, який буде виконаний (Ім'я вузла)

■ *'name*’ - ім'я (ім'я виконуваного файлу), що використовується при виконанні вузла, відповідного зазначеному вище "типу'. Ім'я зазвичай встановлюється таким же, як і тип, але при виконанні його можна налаштувати на використання іншого імені.

Як тільки файл' roslaunch 'буде створений, запустіть' union.запуск ' наступним чином. Зверніть увагу, що коли команда "roslaunch" запускає кілька вузлів, вихідні дані (інформація, помилка і т. д.) виконаних вузлів Не відображаються на екрані терміналу, що ускладнює налагодження. Якщо ви додасте опцію " -- screen", вихідні дані всіх вузлів, що працюють на цьому терміналі, будуть відображатися на екрані терміналу.

```
$ roslaunch ros_tutorials_topic union.launch --screen
```

Як буде виглядати екран, якщо ми його запустимо? По-перше, давайте поглянемо на вузли, запущені в даний час за допомогою наступної команди.

```
$ rosnode list
/rosout
/topic_publisher1
/topic_publisher2
/topic_subscriber1
/topic_subscriber2
```

В результаті вузол "topic_publisher" перейменовується і виконується як 'topic_publisher1' і "topic_publisher2". Вузол "topic_subscriber' також перейменовується і виконується як "topic_subscriber1' і "topic_subscriber2".

Проблема в тому, що на відміну від початкового наміру " запустити два вузли видавця і два вузла передплатника і змусити їх взаємодіяти з відповідними парами", ми можемо бачити через "rqt_graph" (рис.65), що кожен передплатник отримує тему від обох видавців. Це відбувається тому, що ми просто змінили ім'я вузла, який буде виконуватися, не змінюючи ім'я використовуваного повідомлення. Давайте виправимо цю проблему з тегом простору імен в "roslaunch".

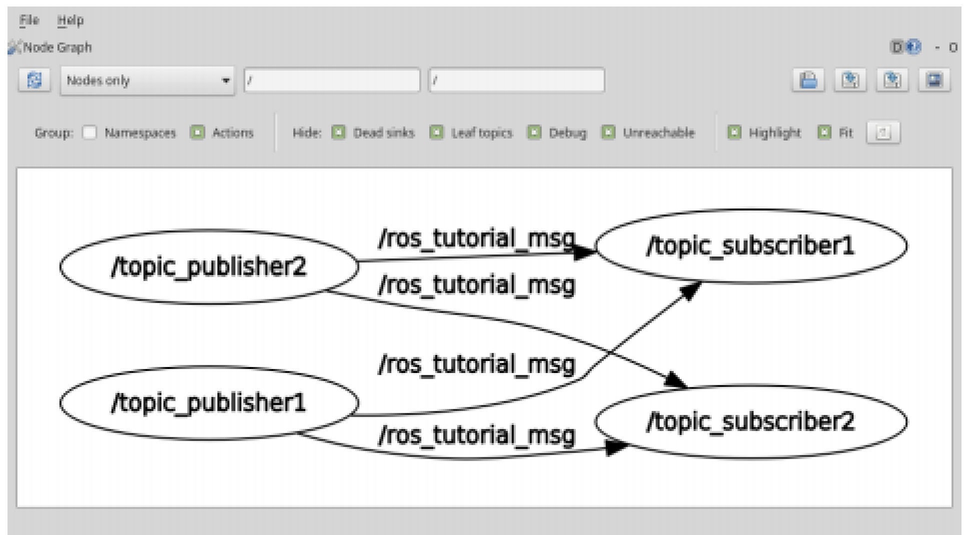


Рис. 65 Діаграма, що показує виконання декількох вузлів за допомогою roslaunch

Давайте змінимо " union.launch" файл, який ми створили раніше.

```
$ roscd ros_tutorials_topic/launch
$ gedit union.launch
```

```
ros_tutorials_topic/launch/union.launch
<launch>
  <group ns="ns1">
    <node pkg="ros_tutorials_topic" type="topic_publisher" name="topic_publisher"/>
    <node pkg="ros_tutorials_topic" type="topic_subscriber" name="topic_subscriber"/>
  </group>
  <group ns="ns2">
    <node pkg="ros_tutorials_topic" type="topic_publisher" name="topic_publisher"/>
    <node pkg="ros_tutorials_topic" type="topic_subscriber" name="topic_subscriber"/>
  </group>
</launch>
```

Тег < group> пов'язує певні вузли. Параметр " ns " відноситься до імені групи як до простору імен, а ім'я та повідомлення вузла, що належить групі, включені в ім'я, вказане "ns".

Ще раз візуалізуйте стан з'єднання і передачі повідомлень між вузлами за допомогою "rqt_graph'. На цей раз ми бачимо, що кожен вузол передає свою передбачувану пару, як показано на рис.66.

Рис. 66 Обмін повідомленнями при використанні простору імен

Тег запуску може бути застосований різними способами залежно від XML13, записаного у файлі запуск. Теги, що використовуються при запуску, такі.

Таблиця 13

<launch>	початок і кінець синтаксису roslaunch.
<node>	для виконання вузла. Пакет, ім'я вузла та ім'я виконання можуть бути змінений.

<machine>	можна задати ім'я, адресу, ross-road і ros-шлях до пакету системи, в якій запущений вузол.
< include>	додатковий файл запуску може бути включений з поточних / інших пакетів, які будуть запущені разом.
< remap>.	змінні ROS, такі як ім'я вузла та ім'я теми, що використовуються у вузлі, можуть бути замінені іншим ім'ям
< env>	встановіть змінні середовища, такі як шлях і IP (рідко використовувані).
<param>	Задайте ім'я параметра, тип, значення і т. д.
< rosparam>	Перевірте та Змініть інформацію про параметри, таку як завантаження, скидання та видалення, як команда "rosparam".
< group>	групуйте виконувані вузли.
<test>	використовується для тестування вузлів. Аналогічно <вузлу>, але з опціями, доступними для цілей тестування.
<arg>	Визначте змінну у файлі запуску, щоб параметр змінювався при виконанні, як показано нижче.

Внутрішні параметри можуть бути змінені ззовні при виконанні файлу запуску за допомогою тегів <param> і <arg>, які є змінною у файлі запуску. Ознайомтеся з цим параметром, так як це дуже корисний і широко використовуваний метод.

```
<launch>
  <arg name="update_period" default="10" />
  <param name="timing" value="$(arg update_period)"/>
</launch>
```

```
$ roslaunch my_package my_package.launch update_period:=30
```

Розділ 8. Робот. Датчик. Двигун (Robot. Sensor. Motor)

8.1. Robot Packages

Робот в основному підрозділяється на апаратне і програмне забезпечення. Механізм, двигуни, шестерні, схеми, датчики класифікуються як апаратні засоби. Мікропрограмне забезпечення мікроконтролера, яке управляє апаратним забезпеченням робота або керує ним, і прикладне програмне забезпечення, яке будує карту, переміщається, створює рух і сприймає навколишнє середовище на основі даних датчиків, класифікуються як програмне забезпечення.

ROS може бути класифікований як прикладне програмне забезпечення, і в залежності від спеціалізованих додатків він класифікується як пакет роботів 1, пакет датчиків 2 і пакет двигунів 3. Ці пакети надаються компаніями-роботами, такими як Willow Garage, ROBOT IS, Yujin Robot і Fetch Robotics, а також Open Robotics (OSRF, раніше Фонд робототехніки з відкритим вихідним кодом) та університетськими лабораторіями в області робототехніки. Індивідуальні розробники також розробляють і поширюють пакети, пов'язані з роботами, датчиками і двигунами ROS.

Шедевром пакету роботів, без сумніву, є PR2 і TurtleBot на рис. 67. Поміж них PR2-мобільний гуманоїдний робот, розроблений для

досліджень компанією Willow Garage, відповідальної за розробку ROS. До цих пір основний пакет інших роботів був отриманий з PR2, який є репрезентативним пакетом роботів ROS.

Хоча PR2 є загальним призначенням і його продуктивність вище, ціна не була досить конкурентоспроможною, щоб оживити АФК на ринку, тому TurtleBot був розроблений для збільшення межі АФК. Перший TurtleBot був заснований на платформі Create, яка є платформою робота-прибиральника iRobot. Для TurtleBot2 KOBUKI був прийнятий в якості мобільної платформи, яка була поліпшеною версією iClebo робота Yujin в Кореї. Тепер TurtleBot 3, мобільний робот на базі Dynamixel, розроблений у співпраці з ROBOTICS, Open Robotics, який буде широко висвітлений в ця книга. Детальна інформація про TurtleBot і використання пакетів роботів наведена в розділі 10.

Рис. 67 PR2(ліворуч), TurtleBot2(2-а ліворуч), TurtleBot3 (3 моделі праворуч)

На додаток до цих репрезентативних роботів існує більше 180 різних типів пакетів роботів, як показано на рис. 65. Ці цифри

обмежені роботами з випущеними джерелами пакетів, і їх число збільшиться, якщо будуть включені інші роботи з компаній, дослідницьких лабораторій, університетів і приватних осіб.

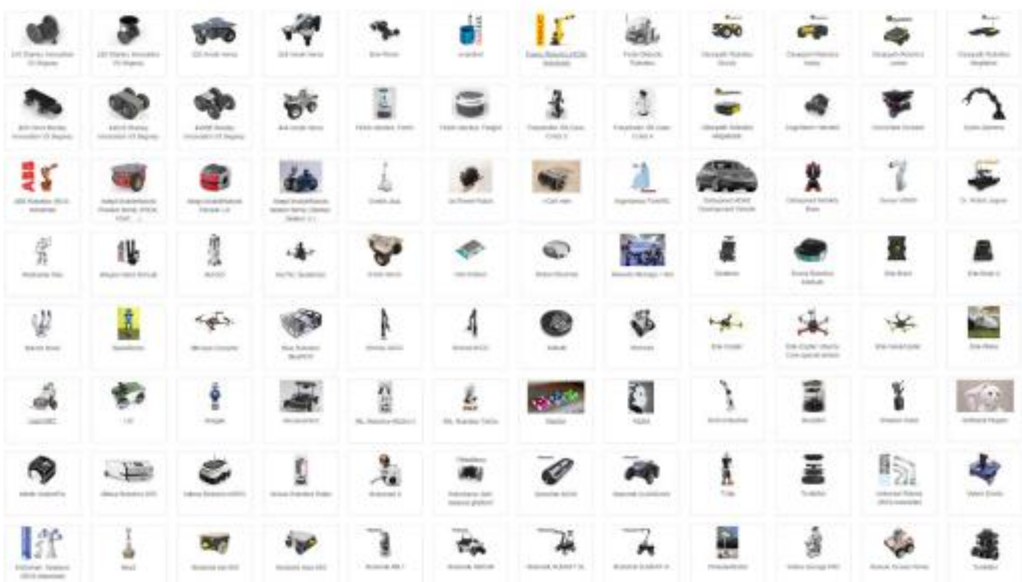


Рис. 68 Роботи, що працюють на ROS (http://robots.ros.org/)

Нижче наведені різні типи роботів, які використовуються майже в кожній області, і розкриті пакети роботів можна знайти за адресою http://robots.ros.org/.

- ■ Manipulator
- ■ Mobile robot
- ■ Autonomous car
- ■ Humanoid
- ■ UAV: Unmanned Aerial Vehicle
- ■ UUV: Unmanned Underwater Vehicle

Процедура встановлення пакета робота повинна бути дуже простою, якщо це офіційний пакет ROSE. По-перше, перевірте, чи вказано пакет робота, який ви збираєтеся використовувати, в Ros Wiki (http://robots.ros.org/) або використовуйте наведену нижче команду для пошуку всього списку пакетів ROS.

```
$ apt-cache search ros-kinetic
```

Іншим можливим методом є використання synaptic, програми диспетчера пакетів графічного інтерфейсу Linux, для пошуку слова " ros-kinetic". Процедура установки пакета робота повинна бути дуже простий, якщо це офіційний пакет ROS. Ось деякі приклади команд для установки пакета PR2.

```
$ sudo apt-get install ros-kinetic-pr2-desktop
```

Нижче наведена команда установки пакета TurtleBot 3

```
$ sudo apt-get install ros-kinetic-turtlebot3 ros-kinetic-turtlebot3-msgs ros-kinetic-
turtlebot3-simulations
```

Рекомендується використовувати останній вихідний код замість двійкової установки, так як TurtleBot 3 постійно оновлюється. Подробиці про цей метод описані в главі 10 “Mobil Robot”.

Навіть якщо пакет робота, який ви збираєтеся використовувати, не є офіційним пакетом, інформацію для установки можна знайти в Wiki пакета робота. Наприклад, Pioneer, широко відомий своїм мобільним роботом, пакет можна завантажити у вихідну папку catkin workspace з сховище, як показано нижче.

```
$ cd ~/catkin_ws/src  → Move to the source folder of catkin workspace
$ hg clone http://code.google.com/p/amor-ros-pkg/    → Download from repository
```

Таким чином, пакет робота можна завантажити або з сховища з відкритим вихідним кодом відповідно до методу установки, показаним у Вікі, або з офіційного пакету ROS.

Будь ласка, дотримуйтесь опису відповідного пакету роботів для використання кожного вузла, включеного в пакет. Пакет робота в основному включає в себе вузол приводу робота, вузол для збору і використання даних змонтованих датчиків і вузол дистанційного керування. Якщо робот є шарнірним роботом, він включає в себе вузол зворотної кінематики, в той час як вузол навігації включений для мобільного робота.

8.2. Sensor Packages

Датчики відіграють вирішальну роль у роботі. Існує безліч досліджень з вилучення значущої інформації з різних середовищ і розпізнавання середовища з використанням цієї інформації і передачі її роботу. Така інформація про навколишнє середовище дуже різноманітна, така як місце розташування, простір, погода, голос, інерція, вібрація, газ, поточна кількість, RFID, об'єкт, розпізнавання зовнішньої сили. Ця інформація використовується в якості важливих даних для робота при виконанні операції.

При створенні робота існує набагато більше, ніж просто додавання коліс або маніпулятора робота і управління ними за допомогою смартфона. Якщо ви завершили цей рівень, ви можете

сказати, що створили рухому машину. Робот може розглядатися як робот тільки тоді, коли він розпізнає навколишнє середовище і витягує значущу інформацію, і на основі цієї інформації він повинен бути в змозі скласти план або судження. Ось чому так важливі датчики.

Існують різні типи датчиків для отримання такої інформації, як різні середовища. Серед них типовим датчиком, використовуваним роботом, є датчик відстані. Як датчики відстані широко використовуються лазерні датчики відстані, такі як LDS(лазерний датчик відстані), лідар (виявлення світла і дальність) або LRF (лазерні далекоміри), а також інфрачервоні датчики, такі як RealSense, Kinect і Xtion. Крім того, існують різні датчики залежно від інформації, яку необхідно отримати, такі як кольорові камери, що використовуються для розпізнавання об'єктів, інерційні датчики, що використовуються для оцінки положення, мікрофони, що використовуються для розпізнавання голосу, і датчики крутного моменту, що використовуються для управління крутним моментом.

Проблема в тому, що існує безліч датчиків, які можна використовувати, як показано на рис.69. Існує обмеження на використання мікропроцесора для прийому різних даних датчиків. Наприклад, DS, 3D-сенсор, камера передають багато даних і вимагають високої обчислювальної потужності, тому це занадто багато для мікропроцесора. Для використання ПК для високопродуктивної обробки даних необхідні драйвери для пристроїв,

а також бібліотеки для таких процесів, як обробка хмар точок за допомогою OpenNI і OpenCV, а також обробка зображень

ROS надає середовище розробки, в якому можуть використовуватися драйвери та бібліотеки вищезазначених датчиків. Не всі датчики підтримуються пакетом ROS, але все більше і більше пакетів, пов'язаних з датчиками, збільшується. Датчики, що використовують один і той же протокол зв'язку, такі як I2C і UART, використовують єдиний метод зв'язку. Виробники датчиків активно підтримують пакети датчиків ROS, що прискорить підтримку ROS майбутніх сенсорних продуктів.

Рис. 69 Приклади датчиків, доступних в ROS

На Вікі-сторінці ROS sensor доступно кілька пакетів датчиків. Датчики класифікуються на 1D далекоміри, 2D далекоміри, 3D Датчики, оцінка пози(GPS + IMU), камери, сенсорні інтерфейси, розпізнавання звуку / мови, Датчики навколишнього середовища, сили

/ крутного моменту / торкання, захоплення руху, Джерело живлення, RFID і датчики представлені в їх категоріях.

Для отримання додаткової інформації про пакети датчиків див. вікі-сторінку датчика ROS, згадану вище. Зокрема, наступні пакети вважаються важливими в цій книзі.

Таблиця 14

1D Range Finders	інфрачервоні датчики відстані, які можна використовувати для створення недорогих роботів.
2D Range Finders	LDS зазвичай використовується в навігації.
3D Sensors	для 3D-вимірювань необхідні такі датчики, як RealSense від Intel, Kinect від Microsoft і XCTION від ASUS.
Audio/Speech Recognition	в даний час існує дуже мало областей, пов'язаних з розпізнаванням мови, але, схоже, вони постійно додаються.
Cameras	перераховані драйвери камер і різні пакети додатків, які широко використовуються для розпізнавання об'єктів, розпізнавання осіб, розпізнавання символів.
Sensor Interfaces	дуже мало датчиків підтримують протоколи USB і Web. Тим не менш, багато даних датчиків можуть бути легко отримані з мікропроцесора. Ці датчики

	можуть бути підключені до ROS через UART мікропроцесора або міні-ПК. Ці сенсорні інтерфейси вводяться.

Існують різні пакети датчиків, тому знайдіть датчики, які найбільше підходять для вашого проекту, і застосуйте їх. Найбільш часто використовувані камери, камери глибини і лазерні датчики відстані (LDS) будуть детально розглянуті в наступних розділах.

8.3. Camera

Камера відповідає оку робота. Зображення, отримані з камери, дуже корисні для розпізнавання навколишнього середовища навколо робота. Наприклад, розпізнавання об'єктів за допомогою зображення камери, розпізнавання облич, значення відстані, отримане з різниці між двома різними зображеннями за допомогою двох камер(стереокамера), візуальний слем монокамери, розпізнавання кольору з використанням інформації, отриманої з зображення, і відстеження об'єктів дуже корисні.

Існує багато видів камер, які будуть використовуватися для обробки зображень, але ми розглянемо USB-камеру. USB-камера означає, що вона підтримує USB-з'єднання. Інша назва-USB Video 200 ROS Robot Programming device Class (UVC)4. Офіційна назва- "UVC-камера", але зазвичай вона називається "USB-камера", яка зазвичай використовується в цьому розділі.

Станом на липень 2017 року остання версія UVC становить 1.55. Версія UVC 1.5 підтримує останню версію USB 3.0 і доступна

практично у всіх операційних системах, включаючи Linux, Windows і OS X. Він простий у використанні, широко поширений і дешевше, ніж інші камери. У цьому розділі ми розглянемо, як запустити USB-камеру і перевірити дані.

ROS надає різні пакети, пов'язані з USB-камерою. Більш детальну інформацію можна знайти в категорії "Сенсор/камера' Вікі-сайту ROS (http://wiki.ros.org/Sensors/Cameras). давайте розглянемо пакети, пов'язані з USB-камерами.

Таблиця 15

libuvc-камера	це пакет інтерфейсів для роботи камер зі стандартом UVC. (Розробник: Кен Тосселл)
uvc камера	це дуже зручний пакет з відносно докладними настройками камери. Крім того, якщо ви розглядаєте конфігурацію стереокамери, цей пакет ідеально підходить для цієї мети.
usb-cam	це дуже простий драйвер камери для Bosch. (Бенджамін Пітцер)
freenect-камера, openni-камера, openni2-камера	всі три пакети позначені як "камера", але ці пакети призначені для камер глибини, таких як Kinect або Xtion. Ці датчики увімкніть кольорову камеру, тому їх також називають камерами RGB-D. ці пакети необхідні для використання їх кольорових зображень.

camera1394	це драйвер для камер, що використовують стандарт IEEE 1394 FireWire.
prosilica-камера	використовується в камері PROSILICA компанії AVT, яка широко використовується в дослідницьких мета.
pointgrey-camera-driver	це драйвер для камери Point Gray дослідження Point Gray, яка широко використовується для досліджень.
camera-calibration:	Джеймс Боуман і Патрік Міхеліч розробили цей пакет калібрування камери, який застосовує функцію калібрування OpenCV. Багато пакетів, пов'язаних з камерою, вимагають цього пакету.

У цьому розділі давайте потренуємося з UVC-камерою 6, розробленою Кеном Тосселлом. Це один з найбільш часто використовуваних пакетів в пакетах USB-камер. Використання інших пов'язаних пакетів аналогічно, тому, якщо ви хочете використовувати інший пакет, перевірте сторінку Wiki для цього пакета.

Таблиця 16

USB Camera:	підключіть USB камеру до USB порту комп'ютера.
Camera Connection Information: показано	відкрийте нове вікно терміналу і перевірте правильність підключення за допомогою команди "lsusb", як камера, ви можете

нижче. Якщо у вас є загальний UVC	перевірити, чи підключена камера, як показано в підкресленому нижче повідомленні.

```
$ lsusb
Bus 004 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 002: ID 2109:0812 VIA Labs, Inc. VL812 Hub
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 005: ID 046d:c52b Logitech, Inc. Unifying Receiver
Bus 001 Device 006: ID 05e3:0608 Genesys Logic, Inc. Hub
Bus 001 Device 013: ID 046d:08ce Logitech, Inc. QuickCam Pro 5000
Bus 001 Device 012: ID 0c45:7603 Microdia
Bus 001 Device 002: ID 2109:2812 VIA Labs, Inc. VL812 Hub
Bus 001 Device 007: ID 8087:0a2a Intel Corp.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

uvc_camera Package Installation

```
$ sudo apt-get install ros-kinetic-uvc-camera
```

Image Related Package Installation

```
$ sudo apt-get install ros-kinetic-image-*
$ sudo apt-get install ros-kinetic-rqt-image-view
```

Running the uvc_camera node

Якщо ви запустите вузол" uvc_camera", Ви отримаєте наступне попереджувальне повідомлення про калібрування камери: "[WARN] [1423194481.257752159]: файл калібрування камери/home/xxx/.ros/camera_info/camera.yaml не знайдено". тому що файл калібрування відсутній. Давайте поки проігноруємо це, оскільки ми детально розглянемо калібрування в наступному розділі.

```
$ roscore
$ rosrun uvc_camera uvc_camera_node
```

Verify Topic Message

У наступному повідомленні теми показано, що інформація про камеру (/camera_info) і інформація про зображення (/image_raw) публікуються.

```
$ rostopic list
/camera_info
/image_raw
/image_raw/compressed
/image_raw/compressed/parameter_descriptions
/image_raw/compressed/parameter_updates
/image_raw/compressedDepth
/image_raw/compressedDepth/parameter_descriptions
/image_raw/compressedDepth/parameter_updates
/image_raw/theora
/image_raw/theora/parameter_descriptions
/image_raw/theora/parameter_updates
/rosout
/rosout_agg
```

У попередньому розділі ми підтвердили, що інформація про зображення публікується під час роботи вузла "uvc_camera_node". У цьому розділі ми використовуємо "image_view" і RViz для візуалізації інформації про зображення. Якщо ви не бачите зображення тут, значить, виникла проблема з драйвером камери або підключенням, тому перейдіть до попереднього розділу, щоб дізнатися, чи є проблема.

Visualization with image_view Node

По-перше, давайте запустимо вузол "image_view", щоб переглянути інформацію про зображення. Додана опція "image:=/image_raw", а "/ image_raw " - це опція для перегляду теми у вигляді зображення в списку тем. При виконанні зображення з камери відображається в невеликому вікні, як показано на рис. 70.

```
$ rosrun image_view image_view image:=/image_raw
```

Рис. 70 Перегляд зображення за допомогою вузла image_view

Visualization with rqt_image_view Node

Давайте розглянемо "rqt_image_view" в розділі 6.2. "image_view" додається в 'rqt_image_view' в якості плагіна rqt з графічним інтерфейсом. Запуск вузла "rqt_image_view" відображає зображення, показане на рис.8-5. На відміну від "image_view", ви можете вибрати тему в графічному інтерфейсі перегляду зображень, яка вже запущена.

```
$ rqt_image_view image:=/image_raw
```

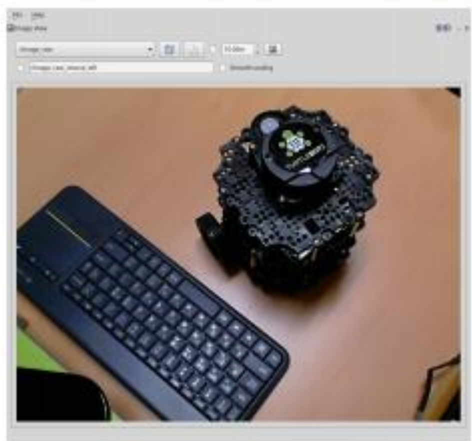


Figure 8-5 Перегляд зображень за допомогою вузла rqt_image_view

Visualization with RViz

Давайте запустимо інструмент візуалізації RViz. Докладний опис RViz див. у розділі 6.1.

```
$ rviz
```

Змініть параметр відображення при виконанні RViz. Натисніть кнопку [Додати] в лівому нижньому кутку RViz і виберіть [зображення] на вкладці [за типом відображення], як показано на рис. 71, щоб відкрити відображення зображення

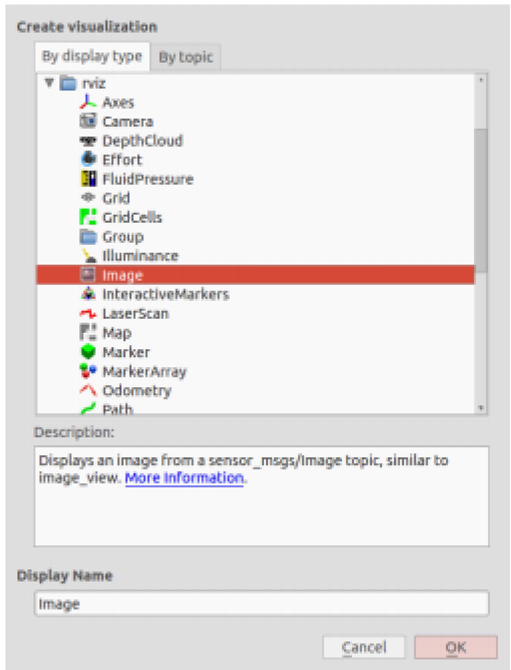


Рис. 71 Додавання відображення зображень в RViz

Змініть значення [тема зображення] в параметрі [зображення] на "/image_raw", після чого зображення відобразиться, як показано на рис. 72. Якщо зображення виглядає маленьким, Відрегулюйте розмір панелі зображень за допомогою миші.

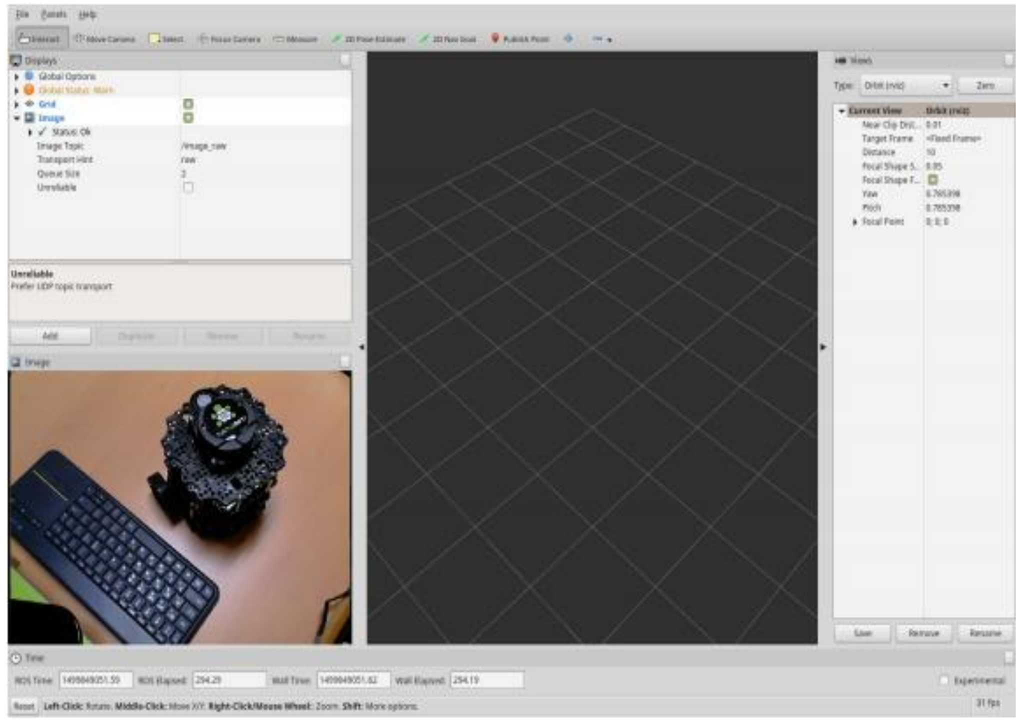


Рис. 72 Перегляд зображень за допомогою RViz

У попередньому розділі ми підключили USB-камеру до комп'ютера і зібрали зображення безпосередньо.

Однак, оскільки робот рухається, оператор не може слідувати за роботом, щоб побачити зображення з камери, прикріпленої до робота. У цьому розділі я поясню, як перевірити інформацію про зображення камери, встановленої на роботі, з іншого комп'ютера у віддаленому місці. Обов'язково уважно прочитайте і дотримуйтесь інструкцій.

Computer with Camera Attached

Майстром ROS може бути будь-який комп'ютер, але в даному прикладі комп'ютер, до якого підключена камера, є майстром ROS.

Перше, що вам потрібно зробити, це налаштувати мережеві змінні, такі як ROS_MASTER_URI та ROS_HOSTNAME. По-перше, давайте відкриємо файл bashrc за допомогою інструменту редагування документів, такого як gedit, sublime text, vim, emacs, nano.

```
$ gedit ~/.bashrc
```

При відкритті файлу bashrc буде багато налаштувань. Залиште попередні налаштування як є, перейдіть в нижню частину файлу bashrc і змініть змінні ROS_MASTER_URI і ROS_HOSTNAME, як показано нижче. Зверніть увагу, що IP-адреса (192.168.1.100) в наступному прикладі повинен бути IP-адресою комп'ютера, до якого підключена камера. При необхідності змініть IP-адресу. Щоб перевірити свою IP-адресу, ви можете використовувати команду "ifconfig", яка описана в розділі 3.2.

```
export ROS_MASTER_URI = http://192.168.1.100:11311
export ROS_HOSTNAME = 192.168.1.100
```

Потім запустіть roscore і запустіть вузол 'uvc_camera_node' в іншому вікні терміналу

```
$ roscore
```

```
$ rosrun uvc_camera uvc_camera_node
```

Remote Computer

Так само, як і головний комп'ютер, необхідно налаштувати віддалений комп'ютер. Відкрийте файл bashrc і змініть змінні ROS_MASTER_URI і ROS_HOSTNAME. Встановіть

ROS_MASTER_URI на IP-адресу комп'ютера, до якого підключена камера (майстер-ПК), і змініть IP-адресу ROS_HOSTNAME як віддалений комп'ютер (192.168.1.120-це IP-адреса віддаленого комп'ютера в цьому прикладі). Перевірте свою IP-адресу віддаленого комп'ютера за допомогою ifconfig і правильно налаштуйте параметри.

Потім запустіть "image_view'.

```
export ROS_MASTER_URI = http://192.168.1.100:11311
export ROS_HOSTNAME = 192.168.1.120
```

```
$ rosrun image_view image_view image:=/image_raw
```

У цьому розділі ми розглянули, як переглядати зображення, отримане з камери, встановленої на роботі, з віддаленого комп'ютера. Він може використовуватися в якості робота дистанційного зондування, робота відеоконференцзв'язку або веб-камери, тобто системи спостереження, яка являє собою цифрову камеру, здатну передавати зображення в мережу в режимі реального часу, оскільки робот може дистанційно управлятися і розпізнавати навколишнє середовище.

Коли вузол uvc_camera запущений, з'являється попереджувальне повідомлення про калібрування камери " [WARN][1423194481.257752159]: файл калібрування камери /home/xxx/.ros/camera_info/camera.yaml не знайдено". може з'явитися, що можна проігнорувати, якщо подальший процес обробки зображення не потрібно.

Однак калібрування камери необхідне, якщо ви вимірюєте відстань від зображень, отриманих за допомогою стереокамери, або обробляєте зображення для розпізнавання об'єктів.

Для отримання точної інформації про відстань від зображення, отриманого з камери, для кожної камери потрібна така інформація, як характеристики об'єктива, зазор між об'єктивом і датчиком зображення і кут нахилу датчика зображення. Це пов'язано з тим, що зображення камери являє собою проекцію тривимірного простору на двовимірне обличчя, і на цей процес проекції впливають характеристики кожної камери.

Наприклад, кожен об'єктив і датчик зображення відрізняються один від одного, зазор між об'єктивом і датчиком зображення відрізняється через апаратної структури камери, об'єктив і датчик зображення не ідеально вирівняні в процесі виробництва камери, що призводить до зміщення центру зображення і основної точки, а датчик зображення може бути злегка скручений або нахилений під кутом.

Калібрування камери-це процес виправлення цих відмінностей шляхом обчислення унікальних параметрів камери. Оскільки калібрування камери дуже важливе, важко описати деталі в цій книзі, тому, будь ласка, зверніться до інших матеріалів, які пояснюють процес обробки зображень OpenCV.

ROS пропонує пакет калібрування за допомогою калібрування камери OpenCV. Відкалібруйте камеру, як описано в наступних розділах.

Camera Calibration

Встановіть пакет калібрування камери і запустіть вузол "uvc_camera_node" наступним чином:

```
$ sudo apt-get install ros-kinetic-camera-calibration
$ rosrun uvc_camera uvc_camera_node
```

Далі, давайте перевіримо поточну інформацію про камеру.

Оскільки інформації про калібрування камери поки немає, будуть відображатися значення за замовчуванням.

```
$ rostopic echo /camera_info
header:
  seq: 7609
  stamp:
```

```
    secs: 1499873386
    nsecs: 558678149
  frame_id: camera
height: 480
width: 640
distortion_model: ''
D: []
K: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
R: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
P: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
binning_x: 0
binning_y: 0
roi:
  x_offset: 0
  y_offset: 0
  height: 0
  width: 0
  do_rectify: False
---
```

Prepare a Chessboard

Калібрування засноване на шахівниці, що складається з чорних і білих квадратів, як показано на рис.8-8. Ви можете завантажити шахову дошку 8x6 за наступною адресою. Роздрукуйте шахову дошку і закріпіть її на плоскій поверхні. Папір формату А4 або букви повинна бути хорошою. Для довідки, 8x6 має 9 горизонтальних квадратів, які складають 8 перетинів, і 7 вертикальних квадратів складають 6 перетинів, тому він називається шахівницею 8x6.

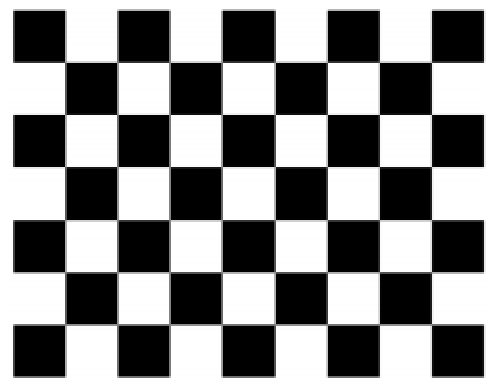


Рис. 73 Шахівниця для калібрування (8 x 6)

Calibration

Давайте виконаємо калібрування камери. Перш ніж ми почнемо, "-- size 8x6 "- це параметр ширини і висоти шахової дошки, тоді як "--square 0.024 " - це фактична довжина сторони одного квадрата. Ця ширина і висота квадрата можуть варіюватися від принтера до принтера, тому виміряйте фактичний розмір шахової дошки і введіть відповідні значення. У цьому випадку ширина і висота квадрата становлять 24 мм, тому наведений нижче приклад буде відображатися відповідним чином.

```
$ rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.024 image:=/image_raw
camera:=/camera
```

Після запуску вузла калібрування з'явиться графічний інтерфейс, як показано на рис. 74. Якщо ви наведете камеру на шахову дошку, калібрування почнеться негайно. У правій частині екрана графічного інтерфейсу ви побачите кольорові горизонтальні смуги, позначені як X, Y, розмір і нахил. Це умови калібрування для

правильного калібрування, і ви повинні налаштувати шахову дошку в різних напрямках щодо камери. У міру поліпшення умов смуги X, Y, розміру і перекосу стануть довшими і стануть зеленими.

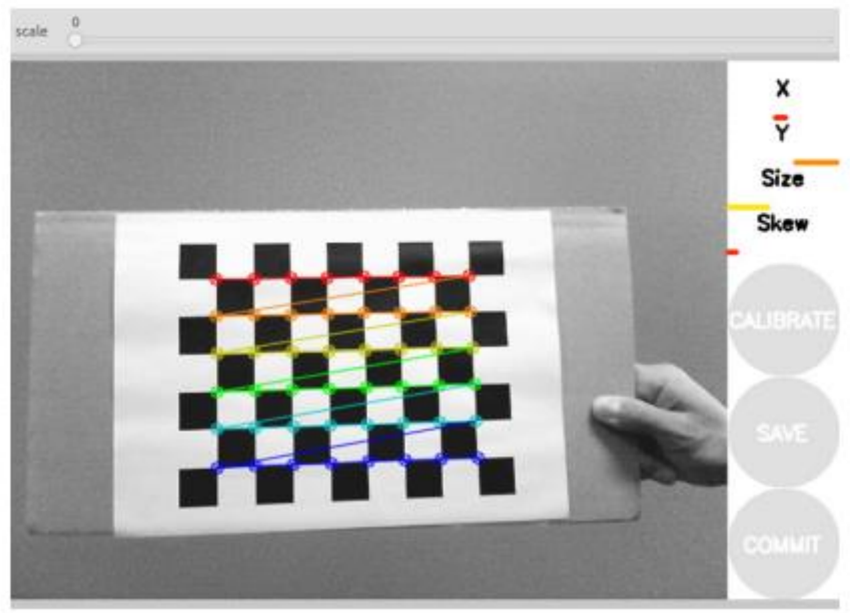


Рис. 74 Початковий стан графічного інтерфейсу калібрування

Кнопка" калібрування " активується, як показано на рис. 75, при зборі необхідних зображень для калібрування. Процес калібрування займає приблизно від 1 до 5 хвилин для розрахунку фактичної калібрування. Коли розрахунок буде завершено, натисніть кнопку Зберегти, щоб зберегти інформацію про калібрування камери. Збережена адреса відображається у вікні терміналу, де було виконано калібрування, і зберігається в папці / tmp, наприклад ' / tmp / calibrationdata.tar.gz".

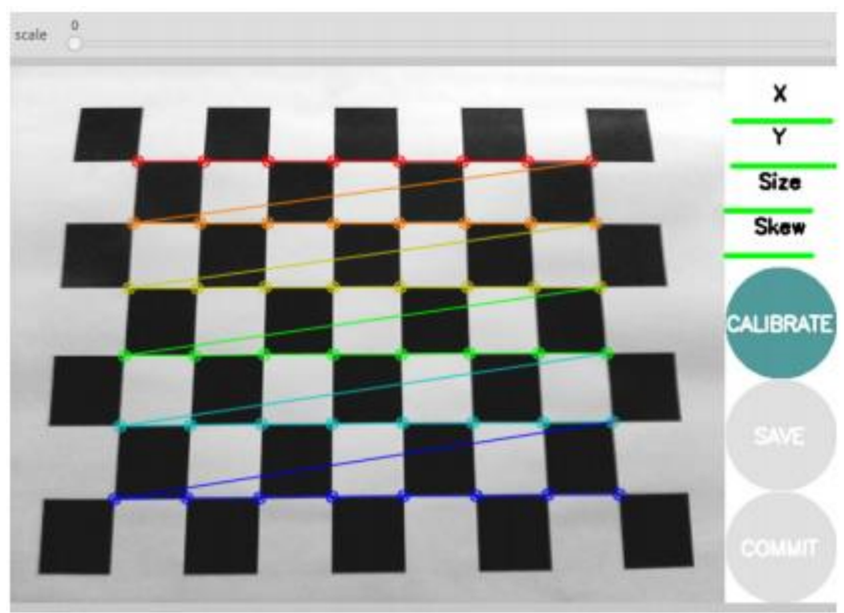


Рис. 75 Процес калібрування за допомогою графічного інтерфейсу калібрування

Create Camera Parameter File

Давайте створимо файл параметрів камери (camera.yaml) для ROS, що містить параметри калібрування камери. Розстебніть блискавку ' calibrationdata.tar.GZ 'файл, щоб побачити" ost.TXT ' файл, що містить параметри калібрування і файли зображень (*.png), що використовуються для процесу калібрування.

```
$ cd /tmp
$ tar -xvzf calibrationdata.tar.gz
```

Перейменувати ' ost.txt' в ' ost.ini' і створіть файл параметрів камери (camera.yaml), використовуючи вузол перетворення пакета "camera_calibration_parsers". Після створення файлу параметрів збережіть його в папці"~/. ros / camera_info/", як показано в

наступному прикладі, і пакети, пов'язані з камерою, що використовуються в ROS, будуть посилатися на цю інформацію.

```
$ mv ost.txt ost.ini
$ rosrun camera_calibration_parsers convert ost.ini camera.yaml
$ mkdir ~/.ros/camera_info
$ mv camera.yaml ~/.ros/camera_info/
```

Файл " camera.yaml "містить параметри, показані в наступному прикладі, і користувач може налаштувати значення "camera_name". Як правило, пакети, пов'язані з камерою, часто використовують "camera "як значення за замовчуванням, тому я змінив значення" name_camera" з "‘narrow_stereo” на "camera".

~/.ros/camera_info/camera.yaml


```
image_width: 640
image_height: 480
camera_name: camera
camera_matrix:
  rows: 3
  cols: 3
  data: [778.887262, 0, 302.058565, 0, 779.885146, 221.545303, 0, 0, 1]
distortion_model: plumb_bob
distortion_coefficients:
  rows: 1
  cols: 5
  data: [0.195718, -0.419555, -0.002234, -0.016098, 0]
rectification_matrix:
  rows: 3
  cols: 3
  data: [1, 0, 0, 0, 1, 0, 0, 0, 1]
projection_matrix:
  rows: 3
  cols: 4
  data: [794.464417, 0, 294.819501, 0, 0, 805.005371, 220.404173, 0, 0, 0, 1, 0]
```

‘camera.yaml " містить таку інформацію, як матриця камери (camera_matrix), коефіцієнт спотворення (distortion_coefficients), "rectification_matrix" для корекції стереокамери і матриці проекції. Опис кожного параметра можна знайти за адресою 'http://wiki.ros.org/image_pipeline/CameraInfo". нарешті, знову запустіть вузол "uvc_camera_node". Переконайтеся, що в цей час ви не отримуєте попереджень про файл калібрування.

```
$ rosrun uvc_camera uvc_camera_node
[INFO] [1499873830.472050095]: using default calibration URL
[INFO] [1499873830.472116471]: camera calibration URL:
file:///home/xxx/.ros/camera_info/camera.yaml
```

Крім того, якщо ви подивитеся на розділ "/camera_info", ви побачите, що параметри D, K, R і P заповнені, як показано в наступному прикладі.

```
$ rostopic echo /camera_info
header:
seq: 2213
```

```
stamp:
    secs: 1499874042
nsecs: 898227060
frame_id: camera
height: 480
width: 640
distortion_model: plumb_bob
D: [0.195718, -0.419555, -0.002234, -0.016098, 0.0]
K: [778.887262, 0.0, 302.058565, 0.0, 779.885146, 221.545303, 0.0, 0.0, 1.0]
R: [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
P: [794.464417, 0.0, 294.819501, 0.0, 0.0, 805.005371, 220.404173, 0.0, 0.0, 0.0, 1.0, 0.0]
binning_x: 0
binning_y: 0
roi:
x_offset: 0
y_offset: 0
height: 0
width: 0
do_rectify: False
---
```

8.4. Depth Camera

Камера глибини може називатися датчиком глибини в тій же категорії, що і LDS(лазерний датчик відстані), і якщо вона може отримувати кольорове зображення, її також можна назвати камерою RGB-D. його також можна назвати камерою Kinect в тому сенсі, що Microsoft досягла успіху в популяризації камери глибини за допомогою Kinect.

У цьому розділі ми будемо називати його глибинною камерою, щоб не плутати терміни.

Глибинні камери можуть бути розділені на різні типи в залежності від методу отримання інформації, наприклад, ToF7 (час польоту), структуроване світло 8 і Стерео 9.

ToF (час польоту)

Метод ToF випромінює інфрачервоні промені і вимірює відстань до моменту його повернення. У загальному випадку їх блок передачі і блок прийому являють собою пару (в деяких випадках конфігурація може відрізнятися), і відстань, виміряне кожним пікселем, зчитується. Причина того, що метод ToF є більш дорогим, ніж структуроване світло, використовуючи метод когерентної діаграми спрямованості, полягає в тому, що ціна обладнання збільшується в цьому структурному аспекті (останнім часом різні методи, такі як Розрахунок відстані з використанням різниці фаз, вводяться за конкурентоспроможною ціною).

Датчики ToF включають D-IMager від Panasonic, SwissRanger від Mesa Imaging, Fotonic - B70 від Fotonic, відеокамери і плати PMDtechnologies, серію DS DepthSense від SoftKinectic і недавно випущений Microsoft Kinect 2.

Рис. 76 Зліва: D-IMager, SwissRanger, CamBoard, Kinect2

Structured Light

Репрезентативні продукти на основі структурованого світла включають Microsoft Kinect і ASUS Xtion, які використовують когерентну діаграму спрямованості випромінювання (патент, цитований з патентом US20100225746, також званим когерентним випромінюванням). Ці датчики використовують систему PrimeSense PrimeSense на чіпі (SoC).

Рис. 77 Зліва Kinect, Xtion, Кармін, Датчик структури

Глибинна камера, що використовує PrimeSense soc PrimeSense, являє собою датчик, що складається з одного інфрачервоного проектора і однієї інфрачервоної камери, яка використовує когерентну діаграму спрямованості, відсутню в існуючому методі tof. Ця технологія почала привертати увагу після вирішення проблем з високою вартістю і зовнішніми перешкодами, а потім були випущені Carmine, Capri з SoC PrimeSense. Крім того, Kinect від Microsoft з тим же чіпом SoC став популярним в якості контролера Xbox. Після цього був випущений Asus Xtion, який був розроблений з урахуванням загального використання комп'ютера. Всі ці датчики оснащені soc PrimeSense.

Однак виникла проблема, коли Apple взяла на себе PrimeSense в грудні 2013 року.

Продукти PrimeSense Carmine і Capri більше не були доступні для покупки, Microsoft Kinect була припинена, а ASUS Xtion також була припинена пізніше. Датчик структури Occipital, який є останнім продуктом з PrimeSense SoC, продає свою продукцію як аксесуар Apple до сьогоднішнього дня, але ми не знаємо, що станеться в майбутньому. Ці популярні недорогі продукти тепер стали історією.

Стереокамера (див. 78), яка є одним з типів глибинних камер, була досліджена набагато більш давно, ніж попередні два типи, і її відстань обчислюється з використанням бінокулярного паралакса, як лівий і правий очі людини. Як випливає з назви, стереокамера оснащена двома датчиками зображення на певній відстані і обчислює значення сітки, використовуючи різницю між двома зображеннями, отриманими цими двома датчиками зображення.Репрезентативні продукти включають камеру Bumblebee Point Gray і Ojocamstereo WithRobot.

Існують різні типи стереокамер, і відмінним методом є налаштування інфрачервоного проектора, який випромінює інфрачервоні промені з когерентним малюнком, і двох датчиків інфрачервоного зображення, які приймають інфрачервоні промені для отримання відстані методом тріангуляції. Перший з подвійним зображенням сенсор називається пасивною стереокамерою, а остання з інфрачервоним проектором-активною стереокамерою. Однією з репрезентативних активних стереокамер є Intel RealSense, яка коштує

близько 100 доларів за модель R200. Це найдешевша з глибинних камер досі, невелика за розміром і схожа за продуктивністю на Xtion, описану вище. Серія D400-це нове покоління RealSense, яке широко використовується в області робототехніки через своїх невеликих розмірів, широкого кута огляду, зовнішнього використання, поліпшеного відстані зондування.

Рис. 78 Зліва направо Bumblebee, OjOcamStereo, RealSense

У цьому розділі використовується Intel RealSense R200 для установки і запуску драйверів для глибинних камер.

Встановлення пакетів, пов'язаних з Real Sense

Завантажте та встановіть драйвери Real Sense та виконувані пакети.

```
$ sudo apt-get install ros-kinetic-librealsense ros-kinetic-realsense-camera
```

Запустіть файл запуску r200_nodelet_default.

Запустіть ' r200_nodelet_default.запустіть файл, розташований у пакеті realsense_camera.

```
$ roscore
```

```
$ roslaunch realsense_camera r200_nodelet_default.launch
```

Якщо ви дотримувалися наведених вище інструкцій, але зіткнулися з проблемами установки пакета або проблемами роботи, вам може знадобитися використовувати певну конфігурацію для різних ядер Linux. Для отримання додаткової інформації зверніться до наступної адреси Wiki.

Набір зібраних 3D-даних відстані від камери глибини називається даними хмари точок, тому що він нагадує хмару за формою. Щоб візуалізувати дані хмари точок у графічному середовищі, запустіть RViz і змініть параметри відображення, показані нижче.

❶ Перейдіть до [Глобальні параметри] та змініть [фіксований кадр] на 'camera_depth_frame'.

❷ Натисніть кнопку [Додати] в лівому нижньому кутку RViz, потім виберіть [PointCloud2], щоб додати його. Встановіть тему 'камера / Глибина / точки' і виберіть розмір і колір.

❸ Після завершення всіх налаштувань ви можете побачити дані PCD, як показано на рис. 79. Оскільки колірна прив'язка встановлена до осі X, чим далі точка знаходиться від осі X, тим ближче колір стає до фіолетового.

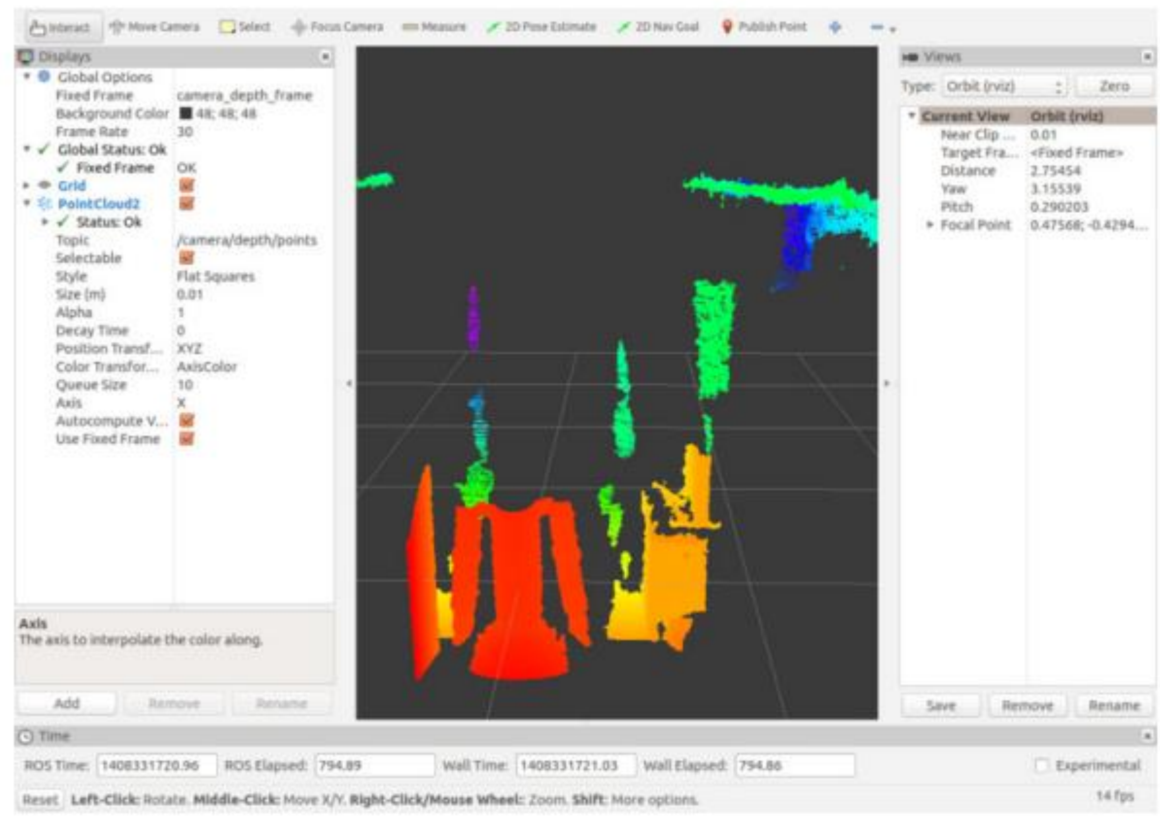


Рис. 79 Point cloud data visualized on the RViz's PointCloud2 display

Якщо ви використовуєте інші глибинні камери, перевірте наступну Вікі-адресу, щоб дізнатися, як керувати камерою та як використовувати пакет.

Датчики глибини діляться на LDS і глибинні камери за способом отримання інформації. ВСІ датчики відстані в цій категорії однакові в тому сенсі, що вони представляють відстань до об'єкта з точкою і мають справу з хмарою точок, яке являє собою сукупність точок. В якості набору API для використання цієї хмари точок

використовується PCL (Point Cloud Library), який виконує фільтрацію, сегментацію, реконструкцію поверхні, витяг фітинга і об'єктів з моделі.

OpenNI (Open Natural Interaction) -це драйвер і різні API-бібліотеки, розроблені PrimeSense Inc. спільно з Willow Garage і ASUS для використання продуктів PrimeSense. NI (природна взаємодія) означає спілкування між людьми та машинами, що походить від значення взаємодії, заснованого на людських почуттях, а не на клавіатурах та мишах. Більшість датчиків з soc PrimeSense використовують цей драйвер.

Подібні бібліотеки включають Microsoft Kinect Windows SDK і Libfreenect, який колись був відомий тим, що звільнив Kinect, вперше зламавши пристрій. Додаток до основного драйвера для управління даними хмари точок, OpenNI також включає в себе проміжне програмне забезпечення, таке як NITE, яке обробляє скелет людського тіла. Після того як Apple захопила PrimeSense, OpenNI був поставлений на межу утилізації, але Occipital тепер пропонує OpenNI у своєму репозиторії Github.

8.5. Лазерний Датчик Відстані

Лазерні датчики відстані (LDS) називаються різними назвами, такими як виявлення світла і далекомір (LiDAR), лазерний далекомір (LRF) і лазерний сканер. LDS-це датчик, який використовується для вимірювання відстані до об'єкта за допомогою лазера як його джерела. Датчик LDS має перевагу високої продуктивності, високої швидкості

збору даних в реальному часі, тому він має широкий спектр застосувань щодо вимірювання відстані. Це датчик, широко використовуваний в області роботів для розпізнавання об'єктів і людей, і датчик відстані на основі SLAM (distance-based sensor), а також широко використовуваний в безпілотних транспортних засобах завдяки його збору даних в реальному часі.

Типовими продуктами є серії Hokuyo URG, які широко використовуються в приміщенні, як показано на рис. 8-15, в той час як SICK широко використовується для зовнішнього використання. Серія HDL Velodyne оснащена декількома лазерними датчиками. Найбільшим недоліком цих датчиків є ціна. Ціни варіюються від продукту до продукту, але зазвичай вони коштують тисячі доларів, а серія HDL Velodyne коштує набагато більше. Китайські продукти (наприклад, RPLIDAR), які компенсують ці недоліки, виходять на ринок за низькою ціною близько 400 доларів США. Останнім часом очікується, що продукт LDS (HLS-LFCD2) вийде на ринок всього за 170 доларів США.

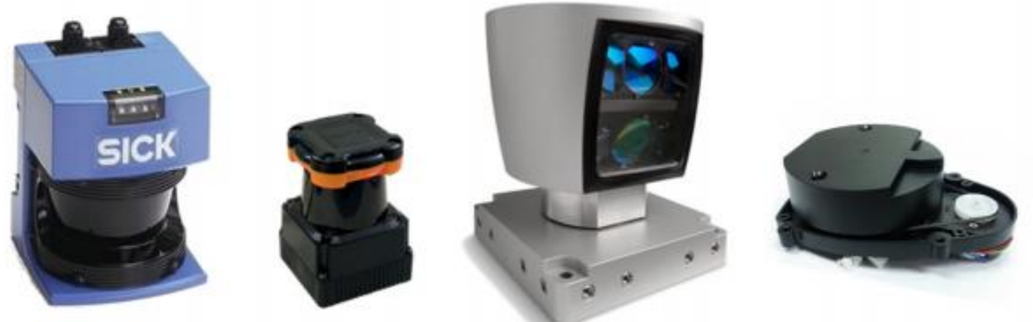


FIGURE 8-15 From left SICK LMS 210, Hokuyo UTM-30LX, Velodyne HDL-64e, HLS-LFCD LDS

Рис. 80 Зліва направо SICK LMS 210, Hokuyo UTM-30LX, Velodyne HDL-LFCD LDS

Датчик LDS обчислює різницю довжин хвиль при відображенні лазерного джерела від об'єкта. Проблема полягає в тому, що виробники використовують лише один лазерний джерело через проблеми з ціною та контролем. Для довідки, серія HDL Velodyne використовує всього 16 і 64 лазера, що значно підвищує ціну продукту. Тому більшість датчиків LDS використовують тільки один лазер. Типовий LDS складається з одного лазерного джерела, що відображає дзеркала і двигуна. Коли ви приводите в рух світлодіоди, ви можете почути звук обертового двигуна, тому що він обертає внутрішнє дзеркало і сканує лазер в горизонтальній площині. Зазвичай вимірюється від 180 до 360 градусів, залежно від продукту.

Ліве зображення на рис. 81 показує світлодіоди з лазером всередині і дзеркалом, нахиленим під кутом. Двигун обертає дзеркало, а датчик вимірює час повернення лазера (обчислює різницю в довжині

хвилі). Таким чином, датчик сканує об'єкти в горизонтальній площині навколо LDS, як показано на центральному зображенні. Однак точність знижується в міру збільшення відстані, як показано на правому зображенні.

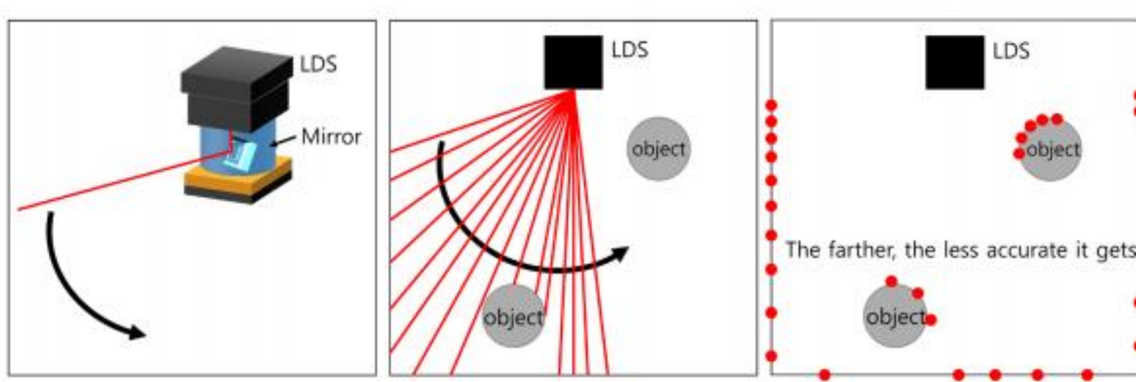


Рис. 81 Вимірювання дистанції з використанням LSD

Хоча користувачам не потрібно знати, як працює DS, важливо інформувати користувачів про можливі проблеми / попередження при використанні LDS.

По-перше, оскільки лазер використовується як джерело світла, сильний лазерний промінь може пошкодити око. Продукти класифікуються на основі лазерного джерела, тому важливо відзначити це при покупці продукту. Загалом, лазер класифікується від класу 1 до класу 4, і чим більше число, тим він небезпечніше. Клас 1-це безпечний продукт, який не має проблем з прямим зоровим контактом. Клас 22 збільшує ризик тривалого впливу. Описані вище LDS відповідають класу 1.

По-друге, оскільки він вимірює віддачу лазерного джерела, отже, марний, якщо нічого не відбивається. Іншими словами, прозоре скло, пластикові пляшки, скляні стаканчики мають тенденцію відображати або розсіювати лазерне джерело в багатьох напрямках. А для дзеркал світло відбивається назад в дзеркало, що робить його неточним виміром.

Оскільки сканується горизонтальна площина, датчик виявляє тільки об'єкти на горизонтальній площині. Іншими словами, вам потрібно знати, що це 2D-дані (у деяких випадках ldss повертається для вимірювання 3D-простору шляхом збору декількох 2D-розмірних даних).

Типові пакети LDS для ROS включають пакет 'sicks300', 'sicktoolbox', 'sicktoolbox_wrapper', який підтримує SICK LDS і 'hokuyo_node', пакет 'urg_node', який підтримує LDS Hokuyo і пакет velodyne, який підтримує LDS velodyne. Є також rplidar, який підтримує RPLIDAR і 'hls_lfcd_lds_driver', який підтримує LDS TurtleBot 3.

Установка пакета hks_fcd_lds_driver

У цьому розділі ми збираємося протестувати використання LDS (HLS-LFCD2) з HLDS (Hitachi LG dat Storage), тому Ви повинні встановити пакет 'hls_lfcd_lds_driver'.

```
$ sudo apt-get install ros-kinetic-hls-lfcd-lds-driver
```

Підключення LDS і зміна дозволу

HIS-LCD 2 використовує послідовний зв'язок, а для підключення датчика до ПК потрібен USB-конвертер. Коли датчик підключений до ПК, він розпізнається як "ttyUSB *' в системі Linux Ubuntu. (Зверніть увагу, що в цьому прикладі датчик розпізнається як "ttyUSB0").

```
$ ls -l /dev/ttyUSB*
crw-rw---- 1 root dialout 188, 0 Jul 13 23:25 /dev/ttyUSB0
```

У попередній команді дозвіл на читання і запис для ttyUSB0 не надається. Давайте встановимо і перевіримо дозвіл, використовуючи наступні команди.

```
$ sudo chmod a+rw /dev/ttyUSB0
$ ls -l /dev/ttyUSB*
crw-rw-rw- 1 root dialout 188, 0 Jul 13 23:25 /dev/ttyUSB0
```

З результату видно, що дозвіл було надано за допомогою команди chmod.

Запуск файлу запуску hlds_laser

Щоб запустити файл запуску 'hlds_laser', введіть наступну команду під час роботи roscore.

```
$ roslaunch hls_lfcd_lds_driver hlds_laser.launch
```

Перевірка даних. Сканування

Коли вузол 'hlds_laser' запущений, значення LDS передається в розділі '/ scan'. Ці дані можна прочитати за допомогою команди rostopic echo наступним чином.

```
$ rostopic echo /scan
header:
  seq: 49
  stamp:
    secs: 1499956463
    nsecs: 667570534
  frame_id: laser
angle_min: 0.0
angle_max: 6.28318548203
angle_increment: 0.0174532923847
time_increment: 2.98899994959e-05
scan_time: 0.0
range_min: 0.119999997318
range_max: 3.5
ranges: [0.0, 0.47200000286102295, 0.4779999852180481, 0.48399999737739563, 0.4909999966621399,
0.4970000088214874, 0.0, 0.5099999904632568,
```

У даних лазерного сканування " frame_id "встановлюється в" лазер", а кут вимірювання-в" 6.28318548203 радіан", рівний 360°. Прирощення кута вимірювання встановлюється рівним 1° (0,0174532923847 рад = 1дег), а мінімальне і максимальне відстані вимірювання становлять 0,11 метра і 3,5 метра відповідно. Ви також можете бачити, що дані вимірювання відстані для кожного кута публікуються у вигляді масиву "діапазони".

Тепер запустіть RViz, щоб перевірити інформацію про відстань LDS в графічному середовищі. Після запуску RViz змініть параметри відображення в наступному порядку:

```
$ rviz
```

❶ Встановіть ' Тип ' у вигляді у верхньому правому куті RViz на "TopDownOrtho", щоб дисплей був змінений на вигляд зверху, який малює інформацію про відстань на площині XY.

❷ У лівій колонці перейдіть в розділ [Глобальні параметри] і встановіть для параметра [фіксований кадр] значення "лазер'.

❸ Натисніть кнопку [Додати] в лівому нижньому кутку RViz, потім виберіть [осі] на дисплеї. Змініть параметри деталізації для довжини і радіуса, як показано на рис. 82.

❹ Натисніть кнопку [Додати] в лівому нижньому кутку RViz і виберіть [LaserScan] на дисплеї. Змініть детальні налаштування теми, колірного трансформатора і кольору, як показано на рис. 82.

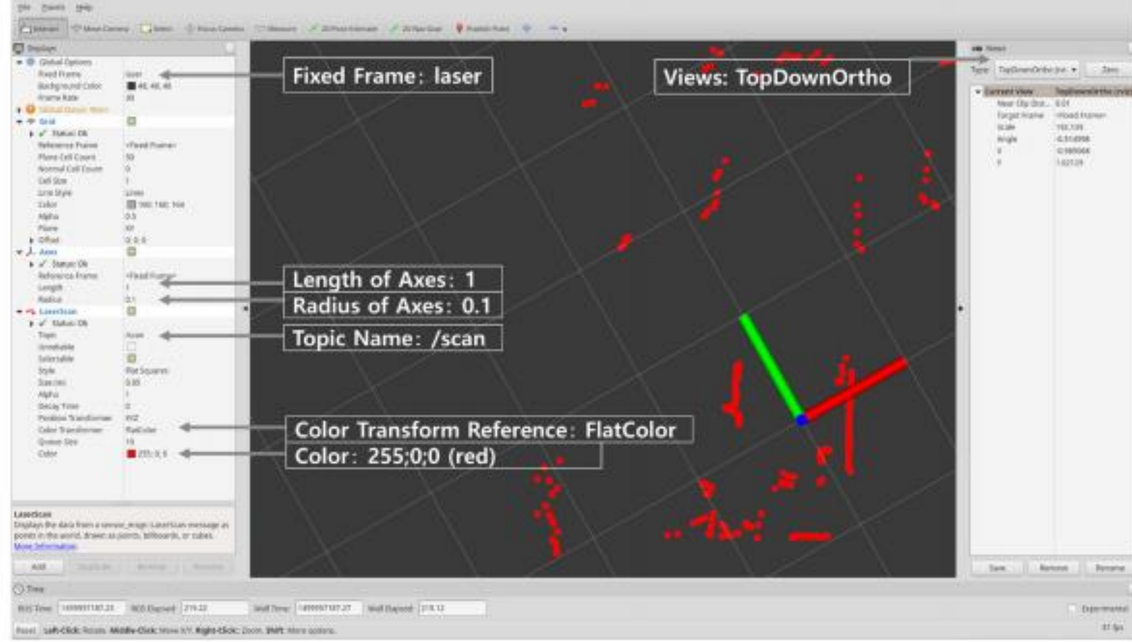


Рис. 82 Дисплей LaserScan на RViz

Після завершення всіх налаштувань можна побачити, що об'єкт сканується навколо осі Z координат, де осі x і y показані червоним і зеленим кольорами відповідно, як показано на рис. 82. Сіра сітка встановлена на 1 м, щоб виміряне значення відстані можна було порівняти з фактичним середовищем.

Конфігурація RViz може бути збережена у вигляді файлу. Давайте перевіримо дані лазерного датчика в RViz за допомогою наступної команди.

```
$ roslaunch hls_lfcd_lds_driver view_hlds_laser.launch
```

Існує нескінченне використання LDS, і SLAM (одночасна локалізація і відображення) є одним з найбільш відомих прикладів використання LDS. SLAM створює карту, розпізнаючи перешкоди навколо робота, і оцінює поточне положення робота на карті, як показано на рис. 83. Слем детально розглядається в главі 11.

В якості ще одного прикладу використання світлодіодів робот здатний виявляти різні об'єкти в навколишньому середовищі і реагувати на них в залежності від поточної обстановки, як показано на рис. 84. Практичне застосування LDS буде більш детально розглянуто в главах 10 і 11 за допомогою світлодіодів на роботі.

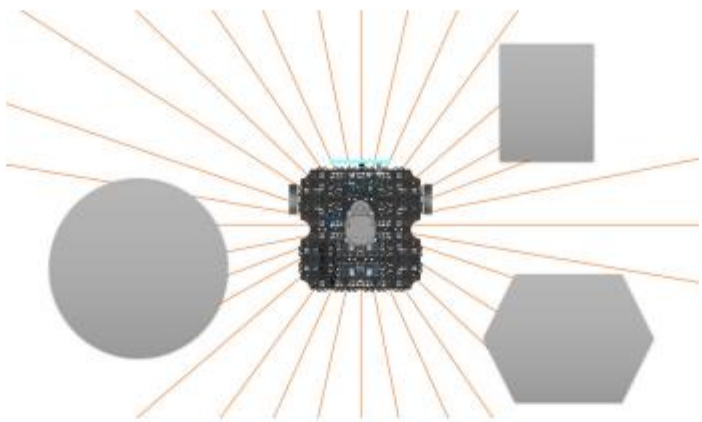


Рис. 83 Використання LSD: визначення переходів мобільного робота

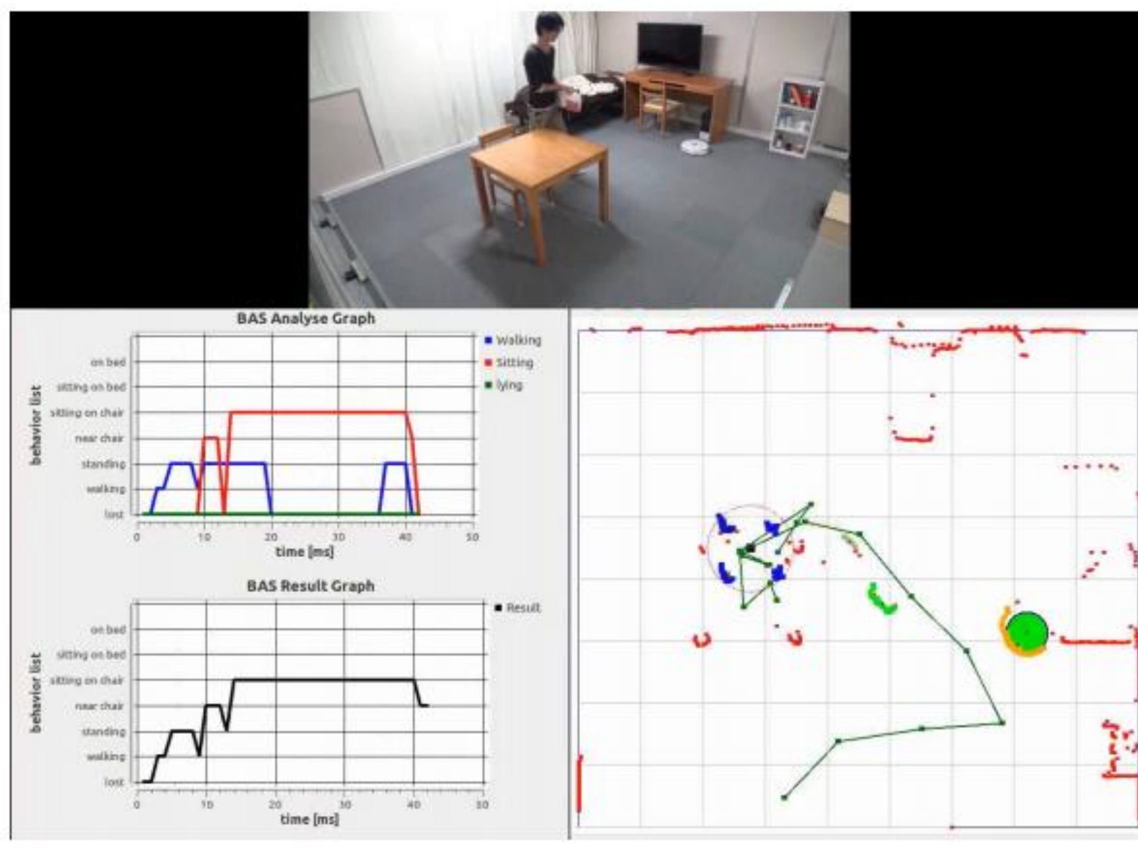


Рис. 84 Використання LSD: визначення людей і переміщення об'єктів

В якості ще одного прикладу використання світлодіодів робот здатний виявляти різні об'єкти в навколишньому середовищі і реагувати на них в залежності від поточної обстановки, як показано на рис. 84. Практичне застосування LDS буде більш детально розглянуто в главах 10 і 11 за допомогою світлодіодів на роботі.

8.6. Моторні пакети

торінка Motors, яка нещодавно була додана до ROS Wiki, - це набір пакетів двигунів та сервоконтролерів, підтримуваних ROS. В

даний час існують пакети, що підтримують "PhidgetMotorControl HC", "Roboteq Ax2550 Motor Controller' і 'ROBOTIS Dynamixel'.

Dynamixel-це інтегрований модуль, що складається з редуктора, контролера, двигуна і комунікаційної схеми. Серія Dynamixel пропонує Зворотні зв'язки для даних про положення, швидкість, температуру, навантаження, напругу і струм за допомогою методу послідовного ланцюга, який забезпечує просте дротове з'єднання між пристроями. На додаток до базового управління положенням також доступні управління швидкістю і крутним моментом (для конкретних моделей), які зазвичай використовуються в робототехніці.

Dynamixel широко використовуються в робототехніці через їх різних корисних функцій. Існує кілька методів використання динаміків для роботів, але будуть розглянуті два основні методи. Перший метод у розділі 13 використовує комунікаційний перетворювач U2D2 для доставки керуючих пакетів від ПК до Dynamixels, а другий метод у розділі 9 використовує вбудовані плати, такі як OpenCR, для безпосереднього управління Dynamixels. Для підтримки Dynamixels в цих різних середовищах розробки можна використовувати Dynamixel SDK, який підтримує різні мови програмування, такі як C, C++, C#, Python, Java, MATLAB і LabVIEW для трьох основних ОС(Linux, Windows, macOS). Dynamixel SDK також підтримує Arduino і поширюється у вигляді пакета ROS,

Dynamixel можна легко використовувати в ROS. Типовими пакетами, що підтримують Dynamixels, є "dynamixel_motor", "arbotix" і "dynamixel_workbench". Перші два пакети надаються користувачами спільноти, а останній пакет офіційно надається компанією ROBOTIS. 'dynamixel_workbench' використовує офіційний DYNAMIXEL SDK і підтримує різні функції, такі як налаштування Dynamixel на GUI, управління положенням / швидкістю / крутним моментом і багатопортовий приклад в ROS. TurtleBot3, який буде детально обговорюватися в цій книзі, також використовує Dynamixel в якості свого приводу. Давайте докладніше розглянемо ці двигуни в главі 9 "Вбудовані системи" і главі 10"мобільні роботи".

Рис. 85 Серія Dynamixel

8.7. Як використовувати публічні пакети

Скільки пакетів було випущено в ROS? Станом на липень 2017 року ROSE Kinetic надає близько 1600 пакетів (http://repositories.ros.org/status_page/ros_kinetic_default.html), а пакети, розроблені та випущені користувачами, складають приблизно

5000 (http://rosindex.github.io/stats/). у цьому розділі ви дізнаєтеся, як шукати серед загальнодоступних пакетів, а також встановлювати і використовувати потрібні вам пакети.

По-перше, введіть наведену нижче адресу в веб-браузері та натисніть "Кінетичний" серед версій ROS поруч із полем пошуку. Пакети, доступні для kinetic, яка є останньою версією ROS, будуть перераховані, як показано на рис. 86.

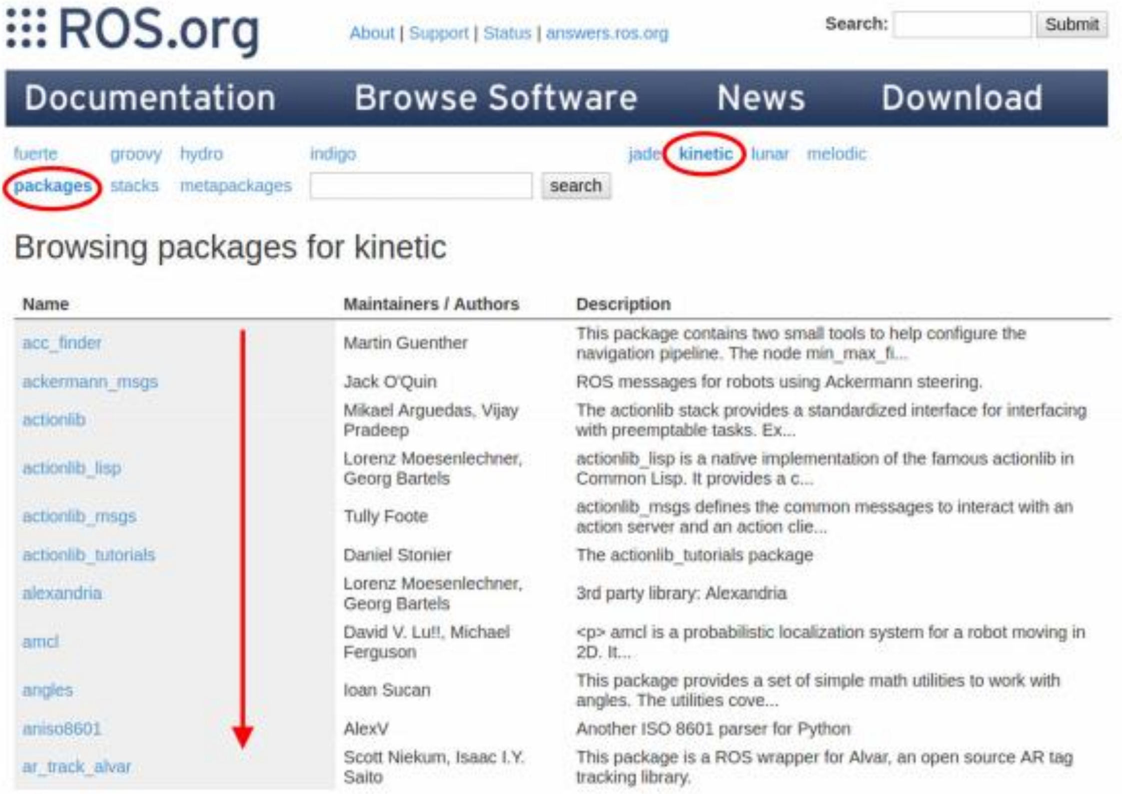


Рис. 87 Списки пакетів ROS

Це пакети, випущені як ROS Kinetic version. Кількість упаковок, мабуть, становить близько 1600. Indigo, попередня версія LTS, випустила понад 2900 пакетів. Більшість пакетів постійно

підтримуються в декількох версіях ROS, але деякі пакети не підтримуються постійно. Навіть якщо конкретний пакет не підтримується у вашій версії ROS, ROS має деяку сумісність з іншими версіями, тому кілька модифікацій дозволять вам використовувати цей пакет. У наступному розділі буде пояснено, як використовувати пакети.

Щоб знайти конкретний пакет серед опублікованих пакетів ROS, введіть ключове слово в поле пошуку на веб-сторінці 'http://wiki.ros.org /' і він покаже результат пошуку з відповідним пошуковим запитом на сайті. Наприклад, якщо ви введете " знайти об'єкт " і натиснете кнопку "Відправити", ви побачите інформацію або питання про різні пакети, які відповідають введеному вами ключовому слову.

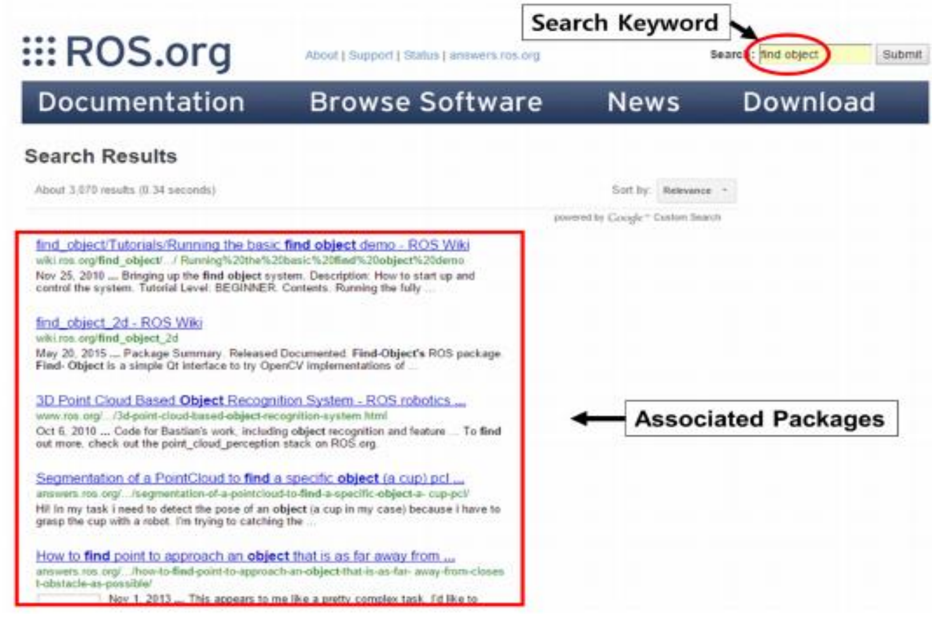
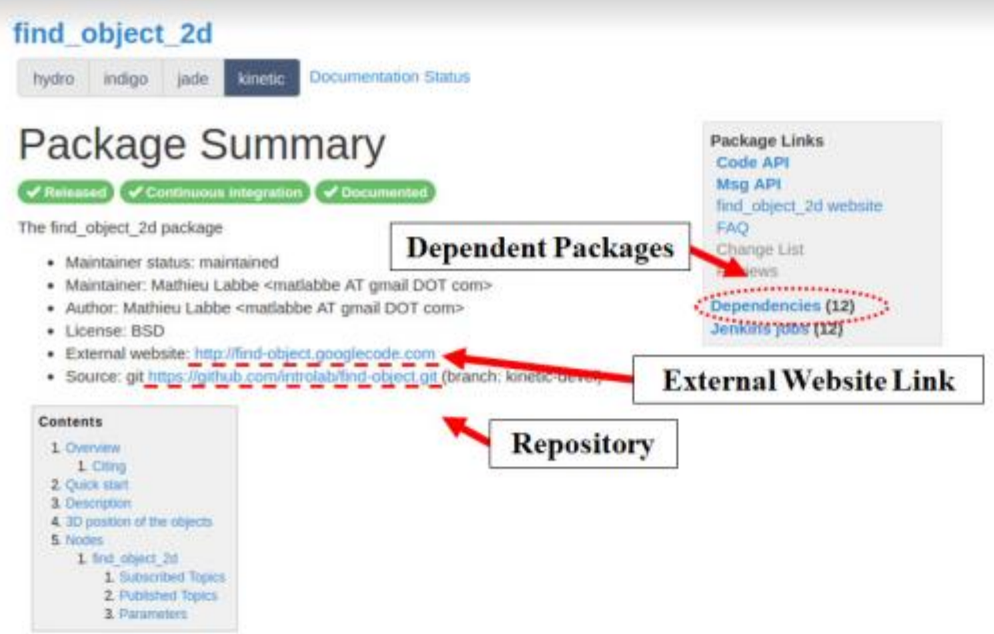


Рис. 88 Метод пошуку пакетів

Відобразиться відповідний пошуковому запиту пакет, як показано на рис. 88. Існує кілька пов'язаних пакетів, але тут ми будемо використовувати пакет 'find_object_2d' з другої' find_object_2d-ROS wiki ' вище. При натисканні кнопки '' find_object_2d-ROS Wiki ''відкриється Вікі-сторінка пакету'' find_object_2d'', як показано на рис. 89.

На цій сторінці ви можете побачити, чи є система збірки catkin або rosbuild, хто її створив і що це за Ліцензія з відкритим вихідним кодом. Давайте спочатку розглянемо інформацію про кінетичну версію, вибравши кнопку Kinetic вгорі. На сторінці кінетичної версії ви можете перевірити список залежних пакетів, натиснувши на посилання ''залежності' в меню статті, посилання на зовнішній веб-сайт проекту, адресу репозиторію пакета та інструкції з використання пакета. Особливо обов'язково перевірте наявність залежностей пакетів.



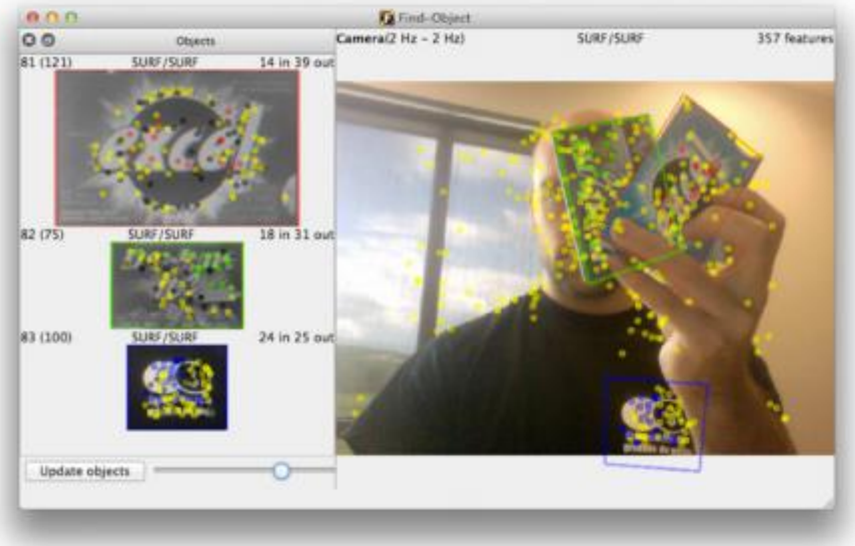


Рис. 89 Інформація про пакет

Якщо ви перевірите залежності пакетів на Вікі-сторінці пакета 'find_object_2d', то побачите, що цей пакет залежить в цілому від 12 різних пакетів.

- catkin
- cv_bridge
- genmsg
- image_transport
- message_filters
- pcl_ros
- roscpp
- rospy
- sensor_msgs
- std_msgs
- std_srvs
- tf

Використовуйте команду 'rospack list' або 'rospack find', щоб переконатися, що необхідні пакети встановлені.

Перевірте за допомогою команди 'rospack list'

```
$ rospack list
actionlib /opt/ros/kinetic/share/actionlib
actionlib_msgs /opt/ros/kinetic/share/actionlib_msgs
actionlib_tutorials /opt/ros/kinetic/share/actionlib_tutorials
```

Перевірте за допомогою команди 'rospack find' (якщо пакет встановлений)

```
$ rospack find cv_bridge
/opt/ros/kinetic/share/cv_bridge
```

Перевірте за допомогою команди 'rospack find' (якщо пакет не встановлений)

```
$ rospack find cv_bridge
[rospack] Error: package 'cv_bridge' not found
```

Якщо залежний пакет не встановлений, перевірте метод установки на кожній Вікі-сторінці і встановіть всі пакети залежностей.

```
$ sudo apt-get install ros-kinetic-cv-bridge
```

Крім того, пакет 'find_object_2d' знаходиться в розділі '2. Quick start' на сторінці Wiki (http://wiki.ros.org / find_object_2d) описується як вимагає пакета 'uvc_camera' (http://wiki.ros.org/uvc_ камера), тому встановіть пакет ''uvc_camera'.

```
$ sudo apt-get install ros-kinetic-uvc-camera
```

Якщо ви встановили всі пакети залежностей, то встановіть 'find_object_2d'. Типовими методами установки є двійкова установка, завантаження і побудова вихідного коду. В інформації про пакет на рис.8-23 натисніть на посилання на репозиторій і перейдіть за адресою Github, який приведе вас до інструкцій по установці.

Бінарна Установка

```
$ sudo apt-get install ros-kinetic-find-object-2d
```

Установка джерела

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/introlab/find-object.git
$ cd ~/catkin_ws/
$ catkin_make
```

аступні пакети не мають прямого відношення до ROS, але бібліотеки OpenCV і Qt використовуються в пакеті 'find_object_2d', тому вам необхідно їх встановити.

```
$ sudo apt-get install libopencv-dev       // Install OpenCV
$ sudo apt-get install libqt4-dev          // Install Qt
```

Запустіть пакет, як описано в інформації про пакет 'find_object_2d'. Спочатку запустіть "roscore" і запустіть вузол камери, використовуючи наступну команду в іншому вікні терміналу.

```
$ roscore
```

```
$ rosrun uvc_camera uvc_camera_node
```

Потім відкрийте інше вікно терміналу та запустіть вузол 'find_object_2d', як показано нижче.

```
$ rosrun find_object_2d find_object_2d image:=image_raw
```

Збережіть зображення з USB-камери в звичайному форматі файлу зображення, такому як PNG або JPEG, і перетягніть його в виконувану графічну програму. Тут ми використовували два зображення для виявлення, як показано на рис. 90.

find_object_2d ROS.org

FIGURE 8-24 Two sample images

Рис. 90 Два зразка зображень

Давайте тепер спробуємо виявити об'єкт. Підготуйте зображення, що включає як зареєстровані, так і незареєстровані об'єкти, і помістіть його перед камерою, як показано на рис. 91. В результаті ви можете бачити, що два об'єкти оточені прямокутниками і правильно виявлені.

Рис. 91 Два виявлені об'єкті

Ви також можете використовувати команду "rostopic echo" у вікні терміналу для перевірки теми "/object "або запустити вузол

"print_objects_detected", щоб побачити інформацію про виявлений об'єкт. При створенні нового пакета за допомогою цього пакета можна створити інший пакет Програми, якщо ви підпишетеся на координати виявлених об'єктів в якості теми.

```
$ rostopic echo /object
```

```
$ rosrun find_object_2d print_objects_detected
```

Пакети, випущені на ROS, швидко ростуть, коли ROS починає широко використовуватися.

Як пояснено в цьому розділі, якщо ви знаєте, як знайти і використовувати пакети, коли вони вам потрібні, ті, хто старанно працював над розробкою пакета, дозволять вам зробити ще один крок вперед і витратити більше часу на те, на чому вам дійсно потрібно зосередитися. Це основна ідея ROS. Накопичені знання виводять нас на більш високий рівень розвитку робототехніки.

У цьому розділі пояснюється, як використовувати пакети в ROS. Будь ласка, зверніться до ROS Wiki для отримання докладної інформації про те, як використовувати кожен пакет.

Розділ 9. Вбудована система

Вбудовану систему можна визначити як комп'ютер спеціального призначення, вбудований в систему, що вимагає управління пристроєм.

Вбудована система-це електронна система, що існує всередині пристрою у вигляді комп'ютерної системи, що виконує певні функції для управління машиною або іншою системою, що вимагає управління. Іншими словами, вбудовану систему можна визначити як комп'ютерну систему певного призначення, яка є частиною всього пристрою і служить мозком для систем, які необхідно контролювати.

Багато вбудовані пристрої використовуються для реалізації функцій роботів, як показано на рис. 92.

Зокрема, мікроконтролер, здатний керувати в реальному часі, необхідний для використання виконавчого механізму або датчика робота, а високопродуктивний процесор на базі комп'ютера необхідний для обробки зображень за допомогою камери або навігації, маніпуляцій.

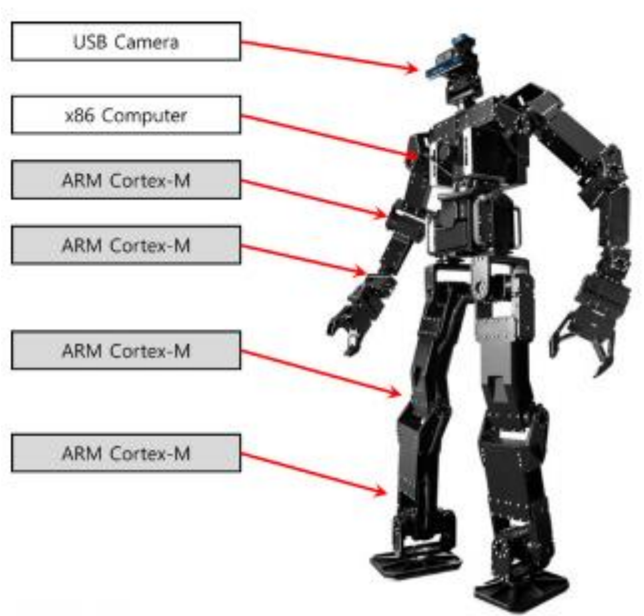


Рис. 92 Вбудована системна конфігурація робота

На рис. 93 показані різні системи від 8-бітного мікроконтролера до високопродуктивного ПК, і вам необхідно налаштувати вбудовану систему з потрібною продуктивністю відповідно до ваших потреб. ROS вимагає операційної системи, такої як Linux, яка управляється ПК або високопродуктивними процесорами серії ARM Cortex-A.

	8/16-bit MCU	32-bit MCU		ARM A-class	x86
		"small" 32-bit MCU	"big" 32-bit MCU		
Example Chip	Atmel AVR	ARM Cortex-M0	ARM Cortex-M7	Samsung Exynos	Intel Core i5
Example System	Arduino Leonardo	Arduino M0 Pro	SAM V71	ODROID	Intel NUC
MIPS	10's	100's	100's	1000's	10000's
RAM	1-32 KB	32 KB	384 KB	a few GB (off-chip)	2-16 GB (SODIMM)
Max power	10's of mW	100's of mW	100's of mW	1000's of mW	10000's of mW
Peripherals	UART, USB FS, ...	USB FS	Ethernet, USB HS	Gigabit Ethernet	USB SS, PCIe

Рис. 93 Види вбудованих дошок

Операційні системи, такі як Linux, не гарантують роботу в реальному часі, а для управління виконавчими механізмами і датчиками потрібні мікроконтролери, які підходять для управління в реальному часі.

У випадку TurtleBot3 Burger і Waffle Pi для керування приводом і датчиком використовується Мікроконтролер серії ARM Cortex-M7, а плата Raspberry Pi 3, яка використовується для Linux і ROS, підключається через USB і налаштовується так, як показано на рис. 94.

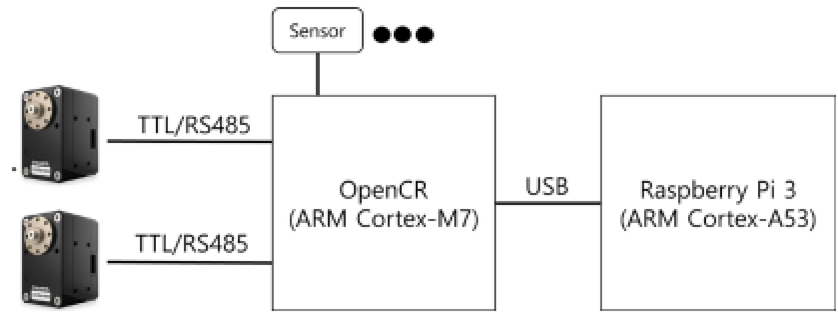


Рис. 94 Конфігурація вбудованої системи TurtleBot

9.1. Opencv

Opencv (Open-source Control Module for ROS) - це вбудована плата, що підтримує ПЗУ і використовується в якості основного контролера TurtleBot3. Апаратна інформація, така як схема, прошивка, дані gerber і вихідний код TurtleBot3, розкривається, і користувачі також можуть випускати і поширювати оригінальні / модифіковані коди.

STM32F746 ST використовується в якості основного мікроконтролера з вбудованим ядром ARM Cortex-M7, а обчислення з плаваючою комою підтримується апаратним забезпеченням, що робить його придатним для реалізації функцій, що вимагають високої продуктивності.

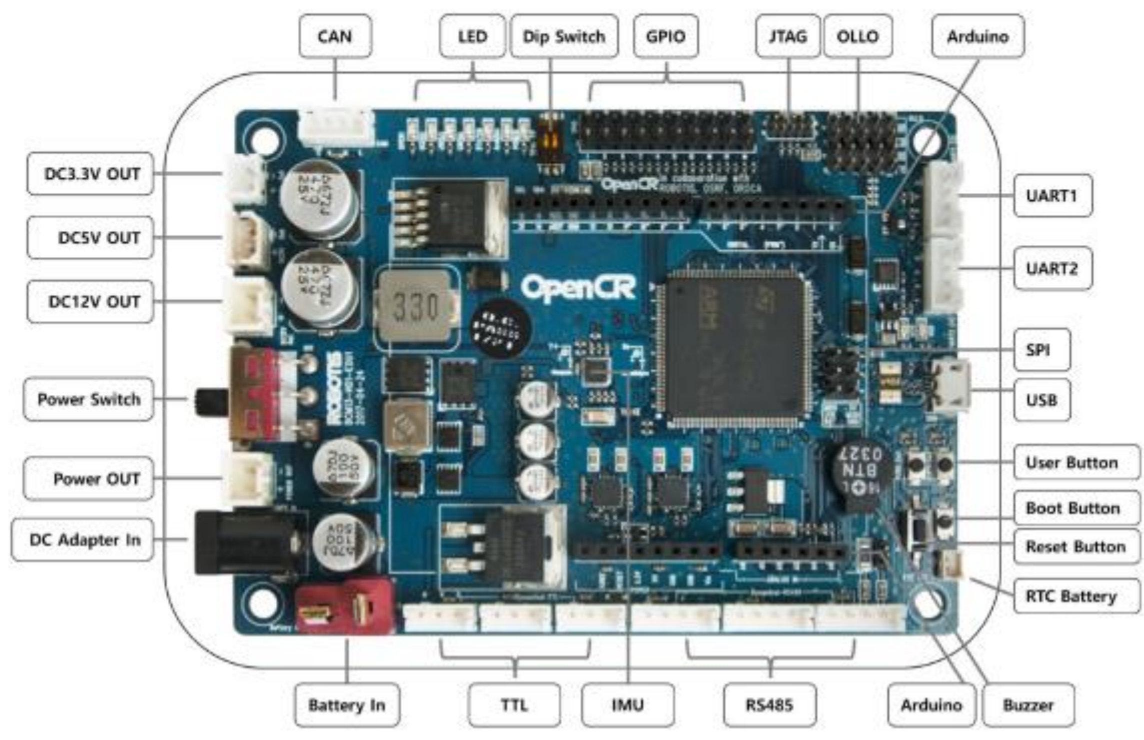


Рис. 95 Налаштування інтерфейсу OpenCR

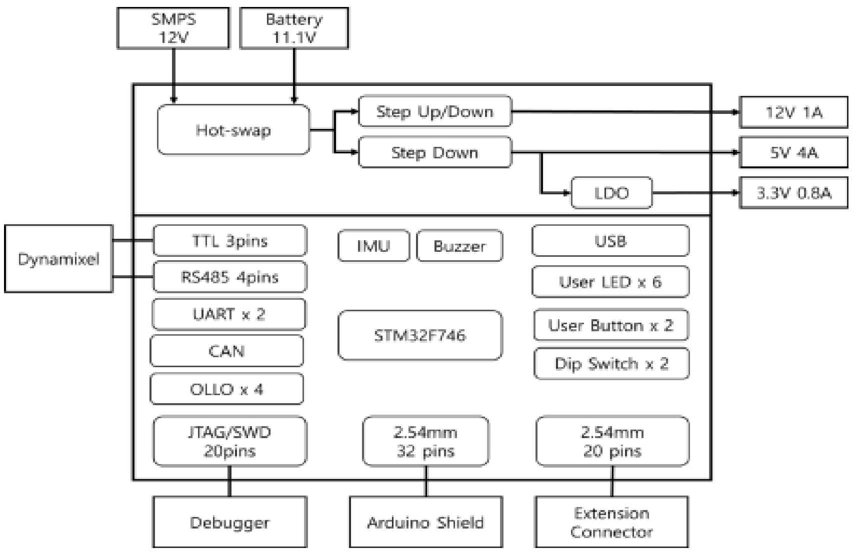


Рис. 96 Структурна схема OpenCR

Висока продуктивність

Мікросхема STM32F746, що використовується в OpenCR, являє собою високопродуктивний Мікроконтролер, що працює на частоті до 216 МГц з ядром Cortex-M7 у верхній частині мікроконтролерів ARM. Він також може бути використаний для обробки великих обсягів даних з використанням алгоритмів і різних периферійних пристроїв, що вимагають високої швидкості роботи.

Підтримка Arduino

Для тих, хто не знайомий з вбудованим середовищем розробки, середовище розробки Opencart просте у використанні за допомогою Arduino IDE. Opencv надає Arduino UNO сумісний інтерфейс, тому можна використовувати різні бібліотеки, вихідний код і екранують модулі, створені для середовища розробки Arduino. Оскільки відкрита плата додається і управляється менеджером плати в Arduino IDE, оновити прошивку дуже просто.

Opencv підтримує зв'язок як TTL, так і RS485, які є інтерфейсами за замовчуванням для DYNAMIXEL від ROBOTICS. Крім того, плата підтримує UART, SPI, I2C, can комунікаційний інтерфейс, а також додаткові контакти GPIO. Також можливо розробляти і налагоджувати прошивку з використанням обладнання JTAG, такого як STLink або JLink для професійних розробників, так як підтримується порт JTAG.

OpenCR включає в себе мікросхему MPU9250, в яку інтегровані триосьовий гіроскоп, триосьовий акселерометр і трьохосьовий магнітометричний датчик в одному чіпі, тому різні програми з використанням датчика IMU можна використовувати без додавання датчика. Висока швидкість читання і запису доступна, оскільки дані датчика передаються за допомогою зв'язку I2C або SPI.

Вихідна потужність

Коли відкривачка отримує вхідний джерело живлення 7 в ~ 24 В, підтримуються виходи 12 В (1А), 5 в (4а) і 3,3 В (800 мА). Його можна використовувати як джерело живлення SBC та Датчик, такий як Raspberry Pi, USB-камера, оскільки вона підтримує до 5V / 4A.

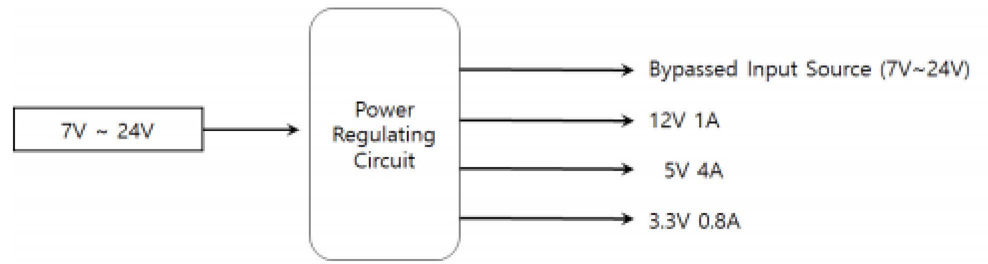


Рис. 97 Діаграма вихідної потужності

Гаряча заміна живлення

Коли джерело живлення SMPS (Switched-Mode Power Supply, який часто називають пристроєм, що перетворює змінний струм у постійний) підключений до OpenCR, блок живлення плати автоматично перемикається з акумулятора на SMPS. Аналогічно, якщо батарея підключена під час використання SMPS, а SMPS відключений, плата працює від батареї. Це дозволяє перемикатися на

живлення від батареї або SMPS без необхідності відключати живлення під час роботи Opencv.

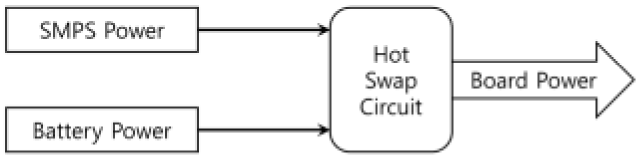


Рисунок 98. Конфігурація Hot-Swap

Відкритий вихідний код

Матеріали, необхідні для створення плати Opencv, є відкритими. Завантажувач, прошивка та друкована плата gerber, необхідні для виробництва обладнання, доступні на GitHub. Тому користувач може змінювати його в міру необхідності і виробляти.

Специфікація устаткування

Технічні характеристики обладнання Open CR наведені в таблиці 99.

Таблиця 17 Технічні характеристики обладнання для OpenCR

Items	Specifications
Microcontroller	STM32F746ZGT6 / 32-bit ARM Cortex®-M7 with FPU (216MHz, 462DMIPS)
Sensors	Gyroscope 3-Axis, Accelerometer 3-Axis, Magnetometer 3-Axis (MPU9250)
Programmer	ARM Cortex 10pin JTAG/SWD connector
	USB Device Firmware Upgrade (DFU)
	USB (Virtual COM Port)
Extension Ports	32 pins (L 14, R 18) *Arduino connectivity
	Sensor module x 4 pins
	Extension connector x 18 pins
Communication Ports	USB
	TTL (JST 3pin / Dynamixel)
	RS485 (JST 4pin / Dynamixel)
	UART x 2
	CAN
	SPI

Items	Specifications
LED Button Switch	LD2 (red/green) : USB communication
	User LED x 4 : LD3 (red), LD4 (green), LD5 (blue)
	User Button x 2
	User Switch x 2
Powers	External input source
	└ 5 V (USB VBUS), 7-24 V (Battery or SMPS)
	└ Default battery: LI-PO 11.1V 1,800mAh 19.98Wh
	└ Default SMPS: 12V 5A
	External output source
	└ 12V@1A, 5V@4A, 3.3V@800mA
	External battery connect for RTC (Real Time Clock)
	Power LED: LD1 (red, 3.3 V power on)
	Reset button x 1 (for power reset of board)
	Power on/off switch x 1
Dimensions	105(W) X 75(D) mm
Weight	60g

Карта флеш-Пам'яті

Всього 1 МБ флеш-пам'яті відкривача складається з області прошивки і області емуляції, яка емулює EEPROM, використовуваний завантажувачем і Arduino. Для емуляції бібліотеки EEPROM, використовуваної Arduino за допомогою флеш-пам'яті, і збільшення терміну служби флеш-пам'яті використовуються два сектори пам'яті.

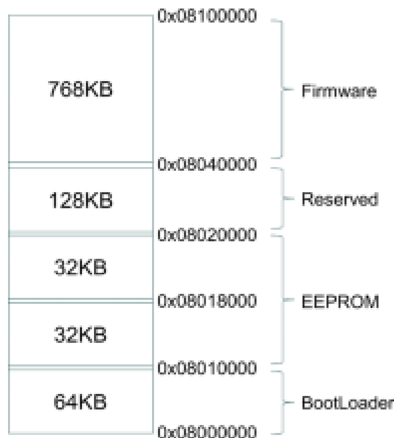


Рис. 99 Карта флеш -пам'яті

Датчик IMU

Датчик MPU 9250 від InvenSense розташований в центрі відкритої плати для точного вимірювання. MPU9250 має вбудовані датчики гіроскопа, акселерометра і магнітометра на одному чіпі. Орієнтація датчика показана на наступних малюнках 100 та 101.

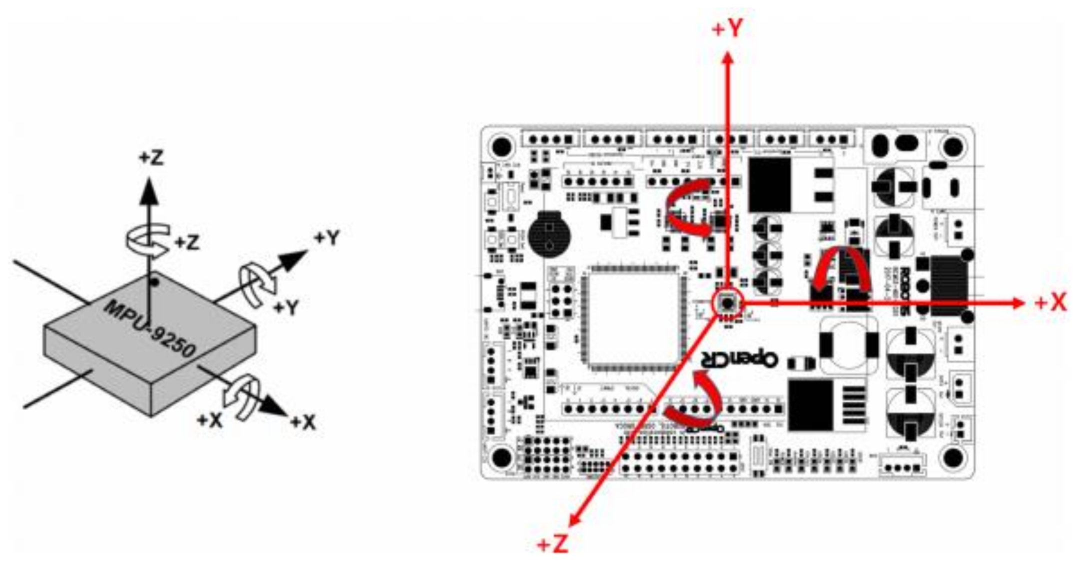


Рис. 100 Напрямок гіроскопа та акселерометра

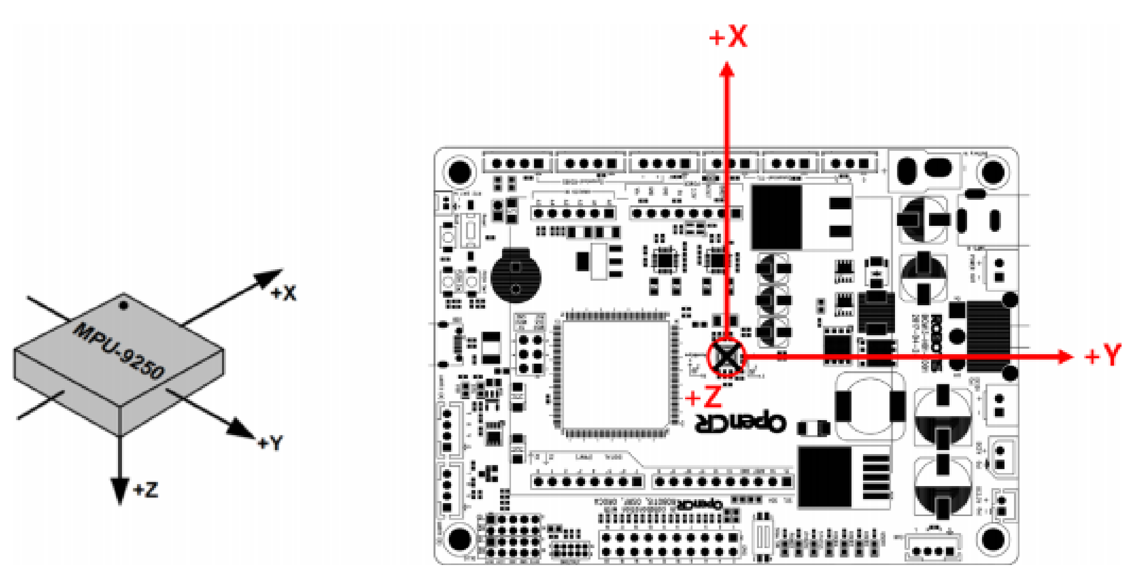


Рис. 101 Напрямок магнітометра

Середовище розробки за замовчуванням Open CRis Arduino IDE. Opener сумісний з Arduino, і додаткове обладнання, що розширює можливості, може підтримуватися окремою бібліотекою. Встановіть

Opencv з менеджера плати Arduino IDE і приступайте до Налаштування. Завантажте Arduino IDE, що розповсюджується з Arduino.cc і керувати правлінням через керуючого правлінням. Давайте налаштуємо середовище розробки в наступних розділах.

Дозвіл USB-порту

Для того щоб завантажити прошивку з Arduino IDE для відкриття, необхідно встановити дозвіл доступу USB за допомогою наступних команд. Наступна інструкція призначена для середовища розробки Linux. Відкрийте нове вікно терміналу за допомогою поєднання клавіш (Ctrl + Alt + t) і введіть наступні команди.

```
$ wget https://raw.githubusercontent.com/ROBOTIS-GIT/OpenCR/master/99-opencr-cdc.rules
$ sudo cp ./99-opencr-cdc.rules /etc/udev/rules.d/
$ sudo udevadm control --reload-rules
$ sudo udevadm trigger
```

Файл '99-opencr-cdc.rules ' містить параметри для зміни дозволу доступу до USB-порту і запобігання розпізнавання OpenCR як модему.

У Linux, коли послідовний пристрій підключено, система передає команду, щоб визначити, чи є пристрій модемом чи ні, і ця команда може викликати проблему, коли OpenCR підключений до системи Linux.

```
ATTRS{idVendor}=="0483" ATTRS{idProduct}=="5740", ENV{ID_MM_DEVICE_IGNORE}="1", MODE:="0666"
```

Переваги компілятора

GCC в Opencv використовує 32-бітні виконувані файли, тому, якщо у вас встановлена 64-бітна ОС, вам необхідно додати 32-бітну Сумісність в систему.

```
$ sudo apt-get install libncurses5-dev:i386
```

Установка Arduino IDE

Завантажте останню версію Arduino IDE з сайту завантаження Arduino. Opencv був протестований у версії 1.6.12 або вище і підтвердив, що він працює на останній версії 1.8.2. Якщо у вас є вища версія Arduino IDE, ви все ще можете використовувати останню версію, оскільки Arduino підтримує сумісність з нижчими версіями.

Скачайте останню версію, розпакуйте її в папку "~/tools " і приступайте до установки. Якщо у вас немає папки tools, створіть нову за допомогою команди 'cd ~ / & & mkdir tools'.

```
$ cd ~/tools/arduino-1.8.2
$ ./install.sh
```

Додайте шлях Arduino IDE до файлу сценарію оболонки, щоб ви могли запустити його з будь-якого місця. Файл сценарію оболонки можна редагувати за допомогою gedit або інших програм редагування тексту, таких як vim, emacs, nano, sublime text і visual studio code.

```
$ gedit ~/.bashrc
```

Додайте розташування шляху до нестисненого файлу в свій шлях і додайте наступну команду, щоб застосувати його.

```
export PATH=$PATH:$HOME/tools/arduino-1.8.2
```

```
$ source ~/.bashrc
```

Запустіть Arduino IDE

Після завершення установки запустіть програму з вікна терміналу з наступною командою.

```
$ arduino
```

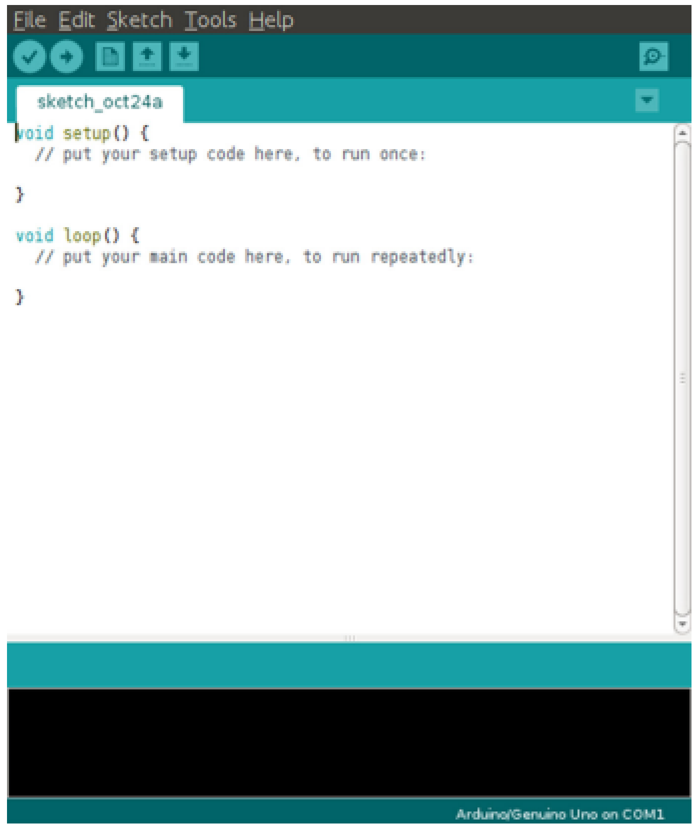


Рис. 102. Екран IDE від Arduino

Налаштування Opencv

Після завершення установки Arduino IDE вам потрібно додати плату, щоб ви могли зібрати і завантажити прошивку в OpenCR. У меню Arduino IDE виберіть файл → налаштування, введіть наступну адресу файлу конфігурації плати в поле Додаткові URL-адреси диспетчера плат на рис. 103 і натисніть кнопку "ОК".

package_opencr_index.json

Рис. 103 Введіть URL -адресу файлу конфігурації плати

Після введення URL-адреси файлу конфігурації плати виберіть Сервіс → Плата → Менеджер плат в меню Arduino IDE, як показано на рис. 104.

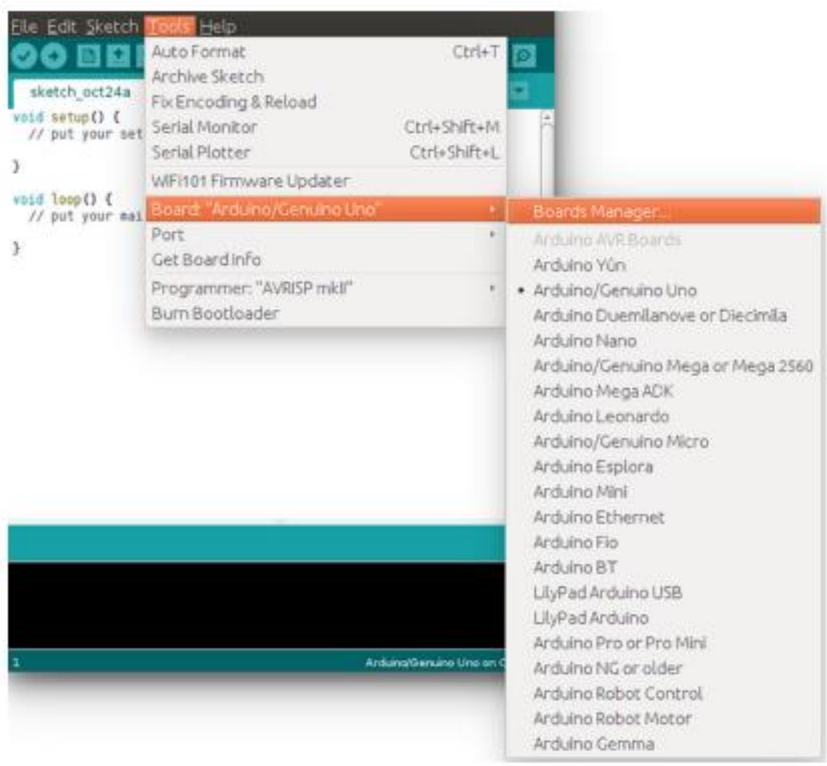


Рис. 104 Запустіть Boards Manager

Ви можете побачити OpenCR в кінці списку дощок. Якщо ви виберете "OpenCR by ROBOTIS" і встановіть його, відповідні файли будуть встановлені автоматично. Ви можете видалити поточну версію або переключитися на інші версії в диспетчері плат.

Type All Filter your search...

EMORO 2560 by **Inovatic-ICT**
Boards included in this package:
EMoRo 2560.
Online help
More info

AMEL-Tech Boards by **replaced by Arrow Boards**
Boards included in this package:
SmartEverything Fox.
Online help
More info

OpenCR by **ROBOTIS**
Boards included in this package:
OpenCR.
Online help
More info
Install

Close

Рис. 105 Board List

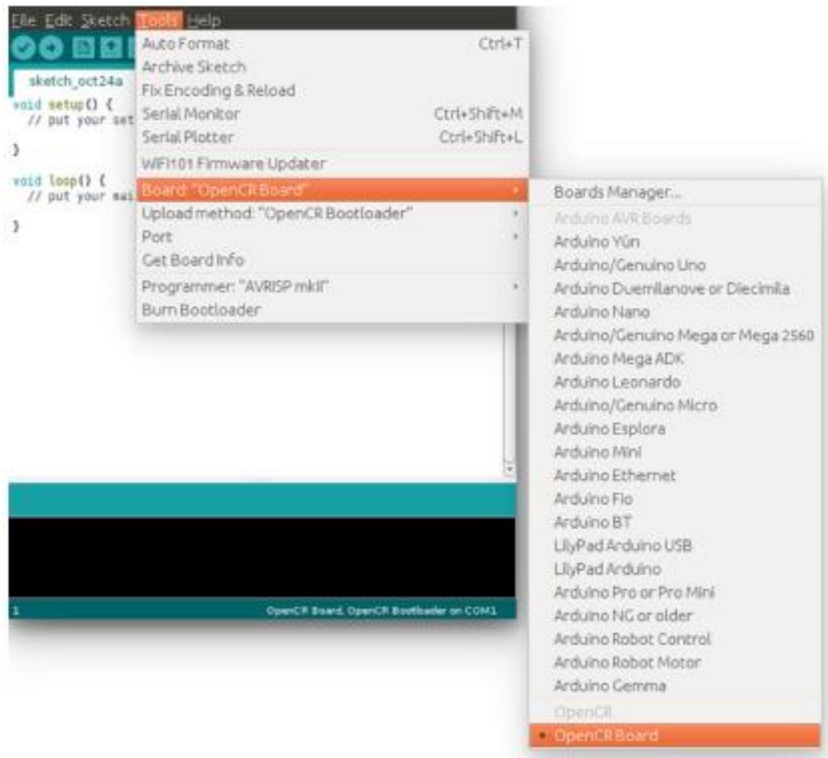


Рис. 106 Select Board

Коли плата OpenCR підключена до ПК, вона розпізнається як послідовний пристрій. Якщо ви виберете ім'я послідовного порту з меню Tools = Port, як показано на рис. 107, то тепер ви готові використовувати відкриту плату.

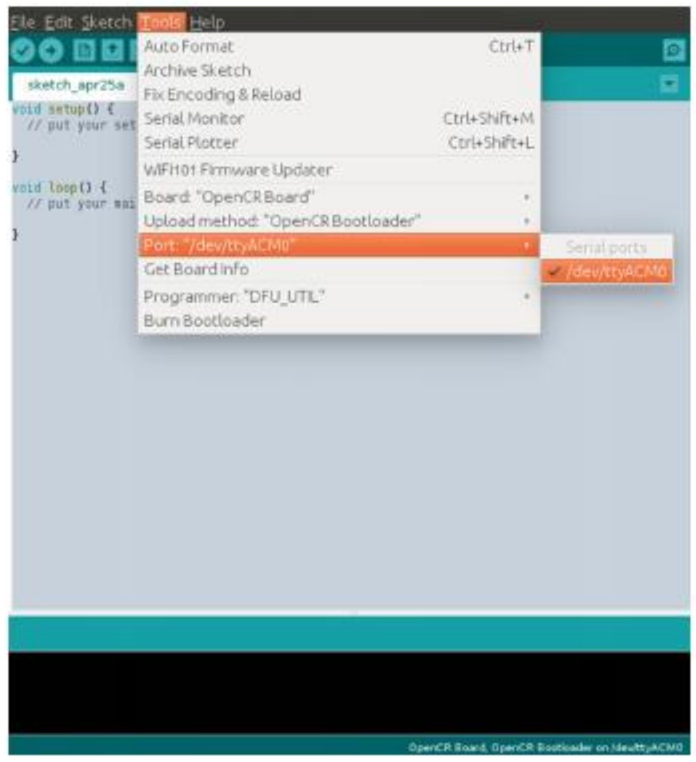


Рис. 107 Виберіть Communication port

Перевірте завантаження мікропрограми

Виберіть Файл → Новий, щоб створити новий файл, як показано на рис. 108. Виберіть плату та порт зв'язку та клацніть піктограму зі стрілкою вправо, щоб створити вихідний код та завантажити у OpenCR.

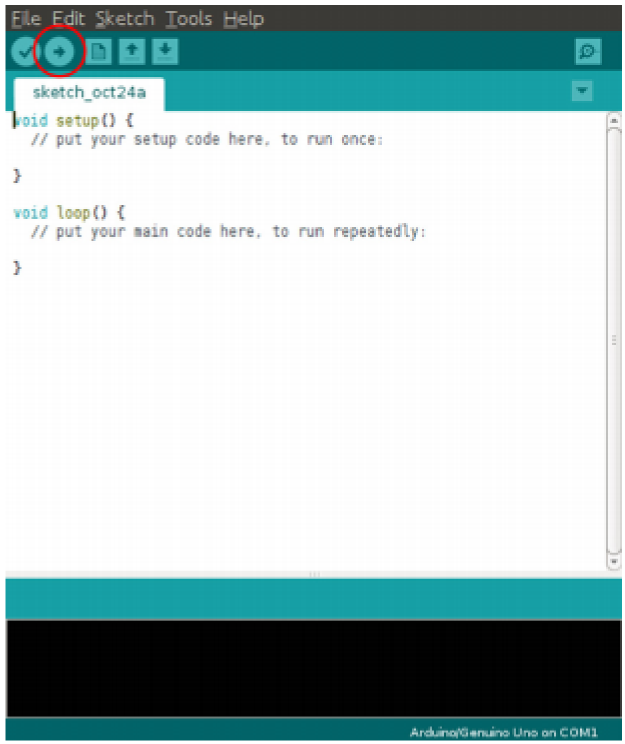


Рис. 108 Створення та завантаження Firmware

Як тільки вихідний код був скомпільований, Arduino IDE викликає завантажувач OpenCR і починає завантаження прошивки. У нижній частині вікна повідомлення з'явиться наступне повідомлення, і завантажена прошивка буде виконана.

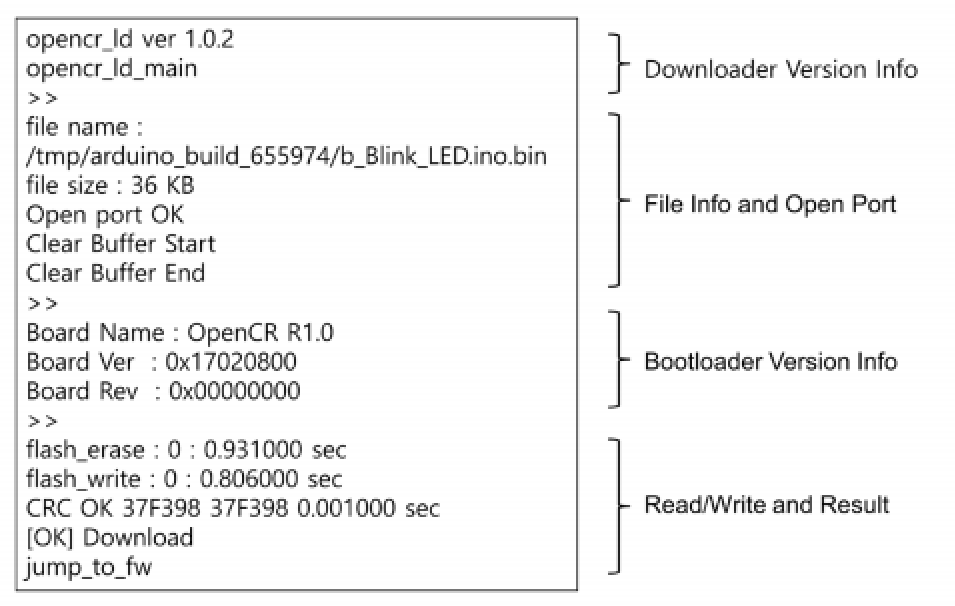


Рис. 109 Завантаження повідомлення

Режим відновлення прошивки

Якщо будь-яка проблема виникає під час роботи і прошивки і не може бути завантажена через цю проблему, прошивку можна завантажити примусово, виконавши завантажувач. Щоб запустити завантажувач, Натисніть і утримуйте кнопку push SW2 на платі і скиньте плату за допомогою кнопки RESET, і завантажувач завантажиться. Коли завантажувач працює, прошивку можна нормально завантажити.

Рис. 110 Режим відновлення Firmware

Оновити завантажувач

Коли потрібно завантажити завантажувач OpenCR, можна використовувати функцію DFU завантажувача, вбудованого в STM32F746. Режим DFU дозволяє користувачам оновлювати завантажувач, не маючи додаткового обладнання, такого як JTAG. Для довідки, завантажувач попередньо завантажується під час виготовлення плати, тому не так багато випадків, коли користувачам доводиться його оновлювати. Поки OpenCR підключений до ПК за допомогою USB, натисніть і утримуйте штифт BOOT0 і натисніть RESET, щоб активувати завантажувач, вбудований в STM32F746, і увійти в режим DFU.

Рис. 111 Кнопка режиму DFU

Ви можете перевірити, чи успішно ви увійшли в режим DFU, використовуючи команду "lsusb". Перебуваючи в режимі DFU, ви повинні побачити "STMicroelectronics STM Device in DFU Mode" в списку USB-пристроїв, як показано на рис. 111.

```
$ lsusb
```

```
$ lsusb
Bus 004 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 003: ID 2109:0812 VIA Labs, Inc. VL812 Hub
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 005: ID 046d:c52b Logitech, Inc. Unifying Receiver

Bus 001 Device 020: ID 0483:df11 STMicroelectronics STM Device in DFU Mode

Bus 001 Device 013: ID 05e3:0608 Genesys Logic, Inc. Hub
Bus 001 Device 012: ID 046d:08ce Logitech, Inc. QuickCam Pro 5000
Bus 001 Device 011: ID 0c45:7603 Microdia
Bus 001 Device 010: ID 2109:2812 VIA Labs, Inc. VL812 Hub
Bus 001 Device 007: ID 8087:0a2a Intel Corp.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Рис. 112 Перевірте режим DFU

Щоб активувати режим DFU, виберіть Tools → Programmer → DFU_UTIL в меню Arduino IDE, як показано на рис. 112.

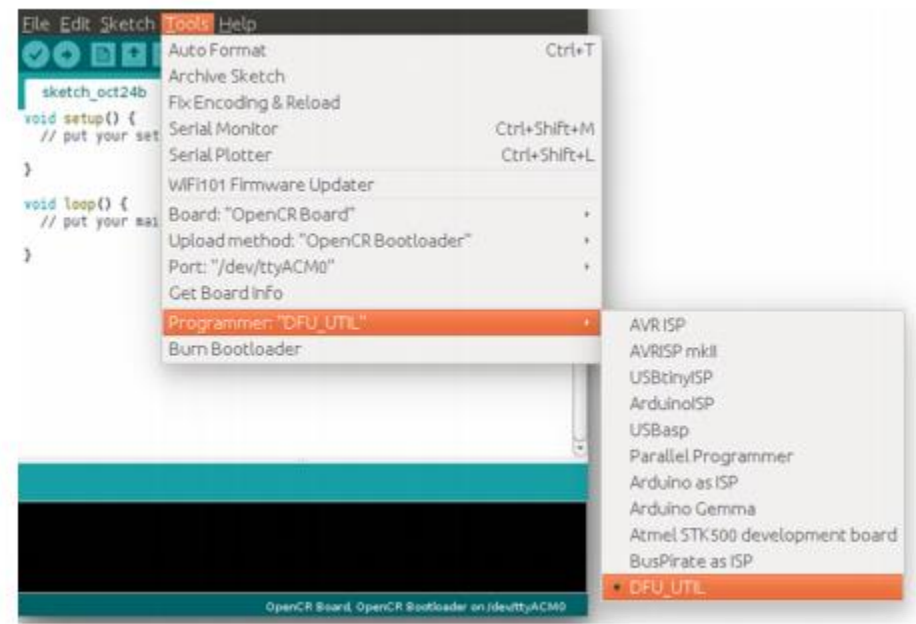


Рис. 113 Оберіть ‘Programmer’

Після зміни програміста вам потрібно вибрати Інструменти → Записати завантажувач, як показано на рис. 114, щоб оновити завантажувач. Після завершення оновлення потрібно скинути плату.

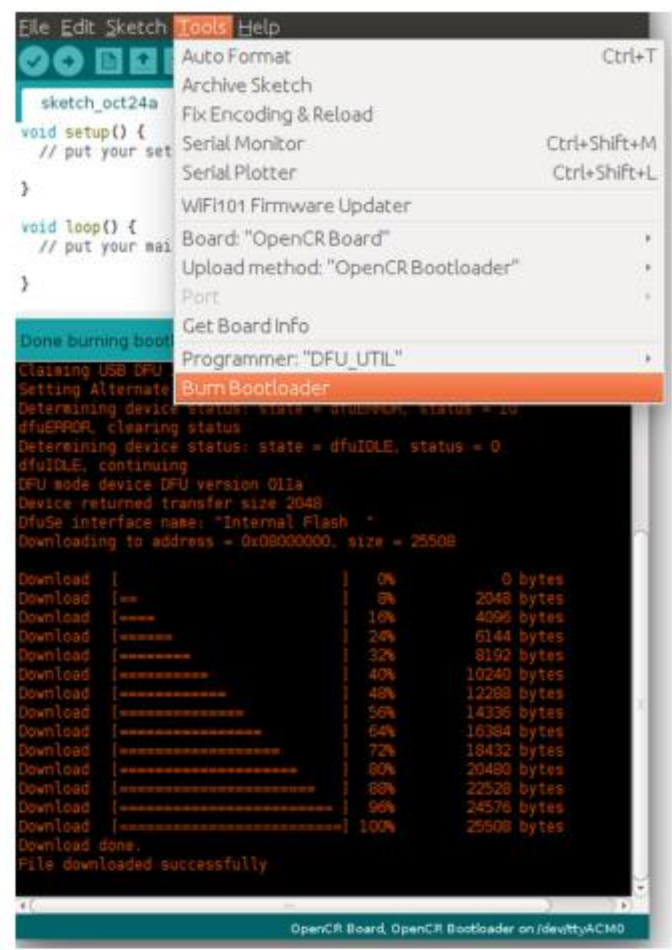


Рис. 114 Оновлення завантажувача

9.1.4. Приклади Opencv

Якщо ви додаєте пакет Opencv в Arduino IDE з диспетчера плат Arduino, ви можете використовувати приклад Opencv в меню File = Examples. Існує ряд прикладів, які допоможуть користувачам контролювати і вивчати використання обладнання, підключеного до Opencv. Давайте розглянемо деякі приклади додаткових функцій, що надаються OpenCR.

Рис. 115 Приклади OpenCR

СВІТЛОДІОД

OpenCR має 4 додаткові світлодіоди, показані на рис. 116, які користувачі можуть використовувати. Давайте відображатимемо інформацію за допомогою цих світлодіодів.

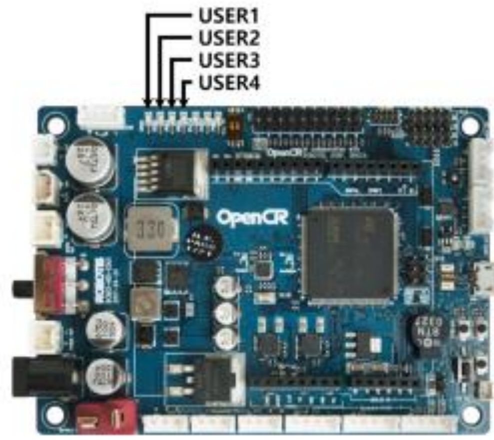


Рис. 116 Location of LEDs

Чотири світлодіоди визначаються як BBPIN_LED_USER_1, BB PIN_LED_USER_2, PIN_LED_USER_3, BB PIN_LED_USER_4, і в наступному прикладі світлодіод буде блимати послідовно.

```
blink_led
int led_pin = 13;
int led_pin_user[4] = { BDPIN_LED_USER_1, BDPIN_LED_USER_2, BDPIN_LED_USER_3, BDPIN_LED_USER_4 };

void setup() {
  pinMode(led_pin, OUTPUT);
  pinMode(led_pin_user[0], OUTPUT);
  pinMode(led_pin_user[1], OUTPUT);
```

```
  pinMode(led_pin_user[2], OUTPUT);
  pinMode(led_pin_user[3], OUTPUT);
}

void loop() {
  int i;

  digitalWrite(led_pin, HIGH);
  delay(100);
  digitalWrite(led_pin, LOW);
  delay(100);

  for( i=0; i<4; i++ )
  {
    digitalWrite(led_pin_user[i], HIGH);
    delay(100);
  }
  for( i=0; i<4; i++ )
  {
    digitalWrite(led_pin_user[i], LOW);
    delay(100);
  }
}
```

Зумер

OpenCR має вбудований зуммер, і одна з основних функцій Arduino може бути використана для створення звуку. Зуммер підключений до виводу, який визначається як BDPIN_BUZZER. Параметрами функції tone () є номер контакту, частота (Гц) і тривалість (мс).

```
buzzer
void setup()
{
}

void loop() {
  tone(BDPIN_BUZZER, 1000, 100);
  delay(200);
}
```

Вимірювання напруги

OpenCR може вимірювати напругу, що надходить від акумулятора або SMPS. Його можна виміряти за допомогою функції getPowerInVoltage (), яка повертає вхідну напругу як одиницю напруги.

```
read_voltage
void setup() {
  Serial.begin(115200);
}

void loop() {
  float voltage;

  voltage = getPowerInVoltage();

  Serial.print("Voltage : ");
  Serial.println(voltage);
}
```

Датчик IMU

Значення датчиків акселерометра і гіроскопа перетворюються в значення крену, тангажа і нишпорення плати за допомогою злиття датчиків. Коли клас IMU створюється як об'єкт і викликається функція update (), значення прискорення і гіроскопа періодично зчитуються з датчика. Цикл розрахунку за замовчуванням становить 200 Гц, але може бути змінений.

```
read_roll_pitch_yaw
#include <IMU.h>

cIMU    IMU;

void setup()
{
  Serial.begin(115200);

  IMU.begin();
}

void loop()
{
```

```
  static uint32_t pre_time;

  IMU.update();

  if( (millis()-pre_time) >= 50 )
  {
    pre_time = millis();

    Serial.print(IMU.rpy[0]);
    Serial.print(" ");
    Serial.print(IMU.rpy[1]);
    Serial.print(" ");
    Serial.println(IMU.rpy[2]);
  }
}
```

Dynamixel SDK

Для управління приводами DYNAMIXEL від ROBOTIS Opencv використовує модифікований DYNAMIXEL SDK C++ для Arduino. Використання DYNAMIXEL SDK таке ж, і існуючий вихідний код може бути частково змінений.

Dynamixel SDK можна завантажити за наступною адресою, і змінена версія вже завантажена в ваш OpenCR за замовчуванням.

Деякі приклади DYNAMIXEL SDK включені в бібліотеку Opencv і підтримують як протоколи 1.0, так і 2.0.

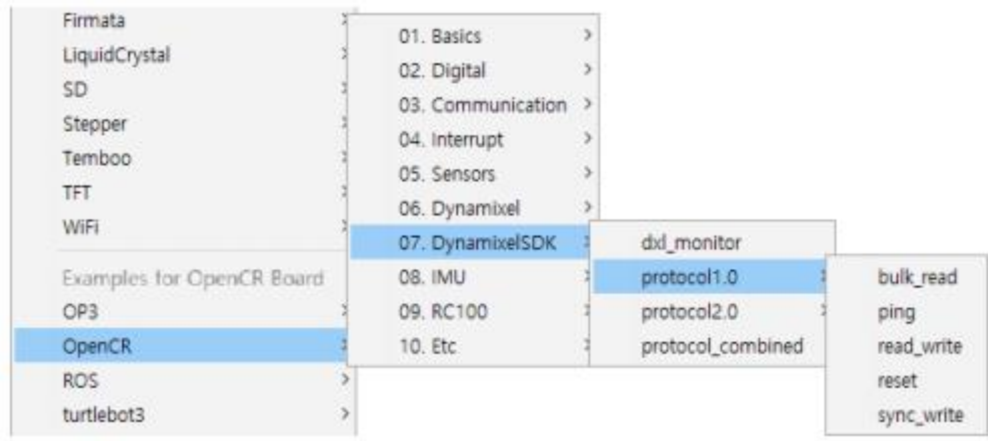


Рис. 117 Приклади Приклади протоколу DynamixelSDK 1.0

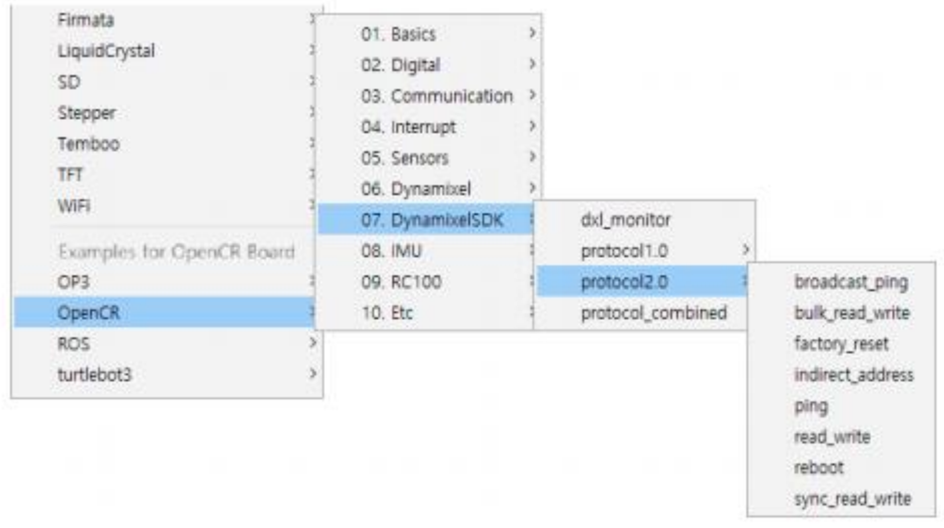


Рис. 118 Приклади DynamixelISDK Protocol 2.0

9.2. Rosserial

Rosserial-це пакет, який перетворює повідомлення, теми та служби ROS для використання в послідовному зв'язку. Як правило, мікроконтролери використовують послідовний зв'язок, такий як UART, а не TCP/IP, який використовується в якості зв'язку за замовчуванням в ROS. Тому, щоб перетворити повідомлення зв'язку між мікроконтролером і комп'ютером за допомогою ROS, rosserial повинен інтерпретувати повідомлення для кожного пристрою.

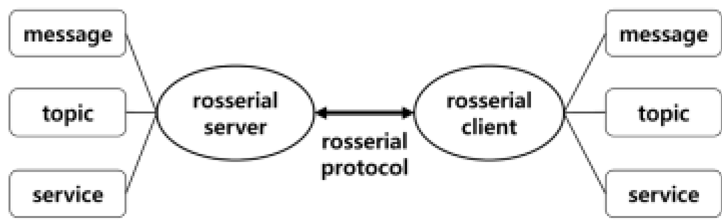


Рис. 119 Сервер rosserial (для ПК) та клієнсткьий сервер (для встанволеної системи)

На рис. 119 ПК, на якому працює ROS, є сервером rosserial, а Мікроконтролер, підключений до ПК, стає клієнтом rosserial. Як сервер, так і клієнт відправляють і отримують дані за допомогою протоколу rosserial, можна використовувати будь-яке обладнання, здатне відправляти і отримувати дані. Це дозволяє використовувати ROS-повідомлення з UART, що часто використовується в мікроконтролерах.

Наприклад, якщо аналоговий датчик, підключений до мікроконтролера, зчитується і перетворює своє аналогове значення в Цифрове, а потім передається в послідовний порт, то вузол" rosserial_server " комп'ютера отримує ці дані датчика і перетворює їх в тему, використовувану в ROS. І навпаки, якщо значення керування двигуном від іншого вузла отримано як тема в вузлі "rosserial_server", вузол "rosserial_server" перетворює це значення і відправляє його в мікроконтролер в послідовному форматі для управління підключеним двигуном.

У випадку загального комп'ютера, включаючи SBC, rosserial не може гарантувати контроль в реальному часі. Однак використання мікроконтролера в якості допоміжного апаратного контролера дозволяє здійснювати управління в режимі реального часу.

Сервер rosserial для ПК-Це вузол, який ретранслює зв'язок з протоколом rosserial між вбудованими пристроями і ПК, що працюють під управлінням ROS. Залежно від реалізованої мови програмування на даний момент підтримується до трьох вузлів.

Таблиця 18

rosserial_python	Цей пакет реалізований на мові Python і зазвичай використовується для використання rosserial.
rosserial_server	Хоча продуктивність була покращена з використанням мови C++, існують деякі

	функціональні обмеження в порівнянні з rosserial_python.
rosserial_java	Бібліотека rosserial_java використовується, коли потрібен модуль на основі Java або коли він використовується з Android SDK.

Бібліотека rosserial_client портована на вбудовану платформу мікроконтролера, щоб використовувати її в якості клієнта для rosserial. Бібліотека підтримує платформу Arduino. Тому будь-яка плата, що підтримує Arduino, може використовувати бібліотеку, а відкритий вихідний код дозволяє легко портувати її на інші платформи.

rosserial_arduino

Ця бібліотека призначена для плати Arduino, яка підтримує Arduino UNO і Leonardo board, але вона також може бути використана на інших платах за допомогою модифікації вихідного коду. Плата OpenCR, що використовується в TurtleBot3, має модифіковану бібліотеку rosserial_arduino.

Таблиця 19

rosserial_embedded linux	Це бібліотека, яку можна використовувати у вбудованому Linux.
rosserial_windows	Ця бібліотека підтримує операційну систему Windows і взаємодіє з додатками Windows.

rosserial_mbed	Ця бібліотека підтримує платформу mbed, яка є вбудованим середовищем розробки, і дозволяє використовувати плати mbed.
rosserial_tivac	Це бібліотека для використання на платі Launchpad виробництва TI.

Сервер rosserial і клієнт відправляють і отримують дані в пакетах на основі послідовного зв'язку. Протокол rosserial визначається на байтовому рівні і містить інформацію для синхронізації пакетів і перевірки даних.

Конфігурація пакетів

Пакет rosserial включає в себе поле заголовка для відправки та отримання стандартного повідомлення ROS і поле контрольної суми для перевірки достовірності даних.

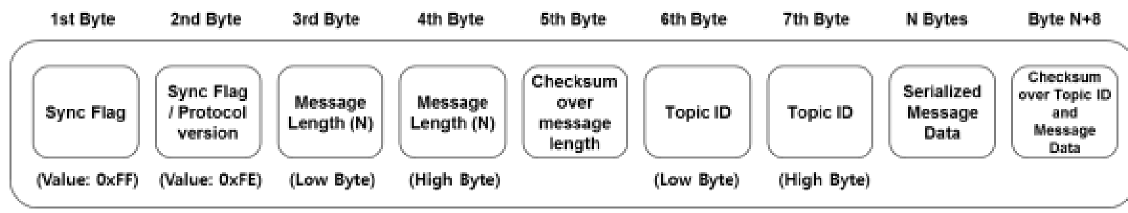


Рис. 120 Структура rosserial Packet

Прапор синхронізації

Цей прапорець байт завжди має значення 0xFF і вказує на початок пакета.

Прапор синхронізації / версія протоколу

Це поле вказує версію протоколу ROS, де Groovy-0xFF, а Hydro, Indigo, Jade, Kinetic-0xfe.

Довжина повідомлення

Це поле в 2 байти вказує довжину даних повідомлення, що передається через пакет. Спочатку йде молодший байт, а потім Старший.

Контрольна сума по довжині повідомлення

Контрольна сума перевіряє правильність довжини повідомлення і обчислюється наступним чином.

```
255 - ((Message Length Low Byte + Message Length High Byte) % 256)
```

TopicID

Поле ID складається з 2 байт і використовується в якості ідентифікатора для розрізнення типу повідомлення. Ідентифікатори тем від 0 до 100 зарезервовані для системних функцій. Основні ідентифікатори тем, що використовуються системою, показані нижче, і вони можуть бути відображені з 'rosserial_msgs/TopicInfo'.

```
uint16 ID_PUBLISHER=0
uint16 ID_SUBSCRIBER=1
uint16 ID_SERVICE_SERVER=2
uint16 ID_SERVICE_CLIENT=4
uint16 ID_PARAMETER_REQUEST=6
uint16 ID_LOG=7
uint16 ID_TIME=10
uint16 ID_TX_STOP=11
```

Серіалізовані Дані Повідомлень

Це поле даних містить серіалізовані повідомлення.

Контрольна сума за ідентифікатором теми та даними повідомлення

Ця контрольна сума призначена для перевірки ідентифікатора теми і даних повідомлення і розраховується наступним чином.

```
255 - ((Topic ID Low Byte + Topic ID High Byte + Data byte values) % 256)
```

Пакет запитів

Коли запускається сервер rosserial, він запитує у клієнта таку інформацію, як ім'я теми і тип. При запиті інформації використовується пакет запитів. Ідентифікатор теми пакета запиту дорівнює 0, а розмір даних-0. Дані в пакеті запиту показані нижче.

```
0xff 0xfe 0x00 0x00 0xff 0x00 0x00 0xff
```

Коли клієнт отримує пакет запиту, він надсилає повідомлення на сервер з наступними даними, а сервер надсилає та отримує повідомлення на основі цієї інформації.

```
uint16 topic_id
string topic_name
string message_type
string md5sum
int32 buffer_size
```

Хоча можна надсилати та отримувати стандартні повідомлення ROHS з вбудованою системою за допомогою rosserial, існують деякі апаратні обмеження вбудованої системи, які можуть викликати

проблему. Тому при створенні вузла з використанням rosserial необхідно враховувати ці обмеження.

Обмеження пам'яті

Мікроконтролери, що використовуються у вбудованій системі, мають значно меншу пам'ять в порівнянні зі стандартними ПК. Тому обсяг пам'яті необхідно враховувати заздалегідь, перш ніж визначати кількість видавців, передплатників і буферів передачі і прийому. Повідомлення, що перевищують розмір буфера передачі або прийому, не можуть бути оброблені, тому остерігайтеся розміру повідомлення. У разі Opencart він надає 1 МБ флеш-пам'яті і 320КБ SRAM, так що ви можете завантажити багато програм. Крім того, ви можете використовувати rosserial більш вільно, встановивши розмір буфера для серіалізації і десеріалізації рівним 1024 байтам для налаштування використання 25 видавців і 25 передплатників.

Float64

При використанні rosserial_arduino в якості клієнта rosserial мікроконтролер на платі Arduino не підтримує 64-бітне обчислення з плаваючою комою, тому при побудові бібліотеки 64-бітний тип даних автоматично перетворюється в 32-бітний. Якщо вбудована плата підтримує 64-бітний розрахунок програмування робота з плаваючою комою 260 ROS

, ви можете змінити код перетворення типу даних в 'make_libraries.py'. оскільки Opencv використовує процесор Cortex-M7 з FPU, він підтримує 64-бітне обчислення з плаваючою комою.

Рядок

Через обмеження пам'яті мікроконтролера рядкові дані не зберігаються в рядковому повідомленні, а в повідомленні зберігається тільки покажчик на певний рядок. У наступному прикладі показано, як використовувати рядкове повідомлення.

```
std_msgs::String str_msg;
unsigned char hello[13] = "hello world!";
str_msg.data = hello;
```

Масив

Подібно до рядків, масив використовує покажчик для обробки фактичних даних у масиві, а кінець даних масиву невідомий. Тому довжина масиву повинна бути включена при передачі і прийомі повідомлень.

Бодрат

ЗВ'ЯЗОК UART зі швидкістю 115200bps може стати повільніше для обробки повідомлень, коли кількість повідомлень збільшується. Opencv використовує віртуальну послідовну зв'язок USB для забезпечення високошвидкісного зв'язку.

Щоб використовувати rosserial, встановіть необхідні пакети для ROS і використовуйте клієнтську бібліотеку платформи використовуваного пристрою. Виконайте наступні дії, щоб

встановити необхідні пакети і дізнатися, як використовувати їх з платформою Arduino.

Установка пакета

Встановіть пакети підтримки rosserial і Arduino за допомогою наступної команди. Крім того, є "ros-kinetic-rosserial-windows", "ros-kinetic-rosserial-xbee" і "ros-kinetic-rosserial-embeddedlinux", але при необхідності встановіть додаткові пакети.

```
$ sudo apt-get install ros-kinetic-rosserial ros-kinetic-rosserial-server ros-kinetic-rosseria
arduino
```

Створення бібліотеки

Щоб використовувати завантажені пакети в середовищі Arduino, вам потрібно створити бібліотеку rosserial для Arduino. Перемістіться в папку libraries IDE Arduino, як показано нижче, і видаліть раніше створені файли бібліотек. Виконати 'make_libraries.py' файл в пакеті 'rosserial_arduino' для створення 'ros_lib'. Оскільки OpenCR вже включає бібліотеку ros_lib для TurtleBot3, наступна процедура не потрібна для TurtleBot 3.

```
$ cd ~/Arduino/libraries/
$ rm -rf ros_lib
$ rosrun rosserial_arduino make_libraries.py .
```

Змінити Комунікаційний Порт

У разі використання бібліотеки Arduino ROS за замовчуванням визначається певний комунікаційний порт. Загальні плати Arduino

використовують послідовний об'єкт у класі HardwareSerial, тому для зміни комунікаційного порту необхідно змінити вихідний код згенерованої бібліотеки. NodeHandle визначається за допомогою класу ArduinoHardware у файлі ' ros.h' в папці бібліотеки, і клас ArduinoHardware повинен бути змінений для зміни комунікаційного порту.

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

namespace ros
{
  /* Publishers, Subscribers, Buffer Sizes */
  typedef NodeHandle_<ArduinoHardware, 25, 25, 1024, 1024> NodeHandle;
}
```

У випадку OpenCR SERIAL_CLASS визначається як USBSerial у файлі ' ArduinoHardware.h', так що він може взаємодіяти через USB-порт. Якщо потрібен інший комунікаційний порт, класи serialport і об'єкт можуть бути змінені.

```
ArduinoHardware.h
#define SERIAL_CLASS USBSerial
__contents omitted__
class ArduinoHardware
{
  iostream = &Serial;
```

```
__contents omitted__
}
```

Opencv надає приклади rosserial для Opencart разом з бібліотекою rosserial. Як показано на рис. 121, можна знайти приклади введення і виведення, такі як світлодіод, кнопка і датчик IMU, і будуть додані додаткові приклади.

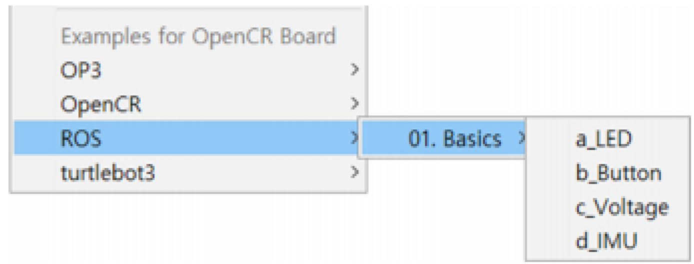


Рис. 121 Приклади rosserial для OpenCR

Ви повинні спочатку запустити roscore, перш ніж намагатися виконати наступні приклади.

СВІТЛОДІОД

Чотири світлодіоди визначаються у абонента «led_out» за допомогою «std_msg / Byte», що є стандартним типом даних ROS. Коли викликається функція зворотного виклику у абонента, відповідний світлодіод на біт 0 ~ 3 отриманого повідомлення відповість, увімкнувши або вимкнувши світло, коли біт дорівнює 1 або 0 відповідно.

a_LED.ino

```
#include <ros.h>
#include <std_msgs/String.h>
#include <std_msgs/Byte.h>

int led_pin_user[4] = { BDPIN_LED_USER_1, BDPIN_LED_USER_2, BDPIN_LED_USER_3, BDPIN_LED_USER_4 };

ros::NodeHandle nh;

void messageCb( const std_msgs::Byte& led_msg) {
  int i;

  for (i=0; i<4; i++)
```

```
  {
    if (led_msg.data & (1<<i))
    {
      digitalWrite(led_pin_user[i], LOW);
    }
    else
    {
      digitalWrite(led_pin_user[i], HIGH);
    }
  }
}

ros::Subscriber<std_msgs::Byte> sub("led_out", messageCb );

void setup() {
  pinMode(led_pin_user[0], OUTPUT);
  pinMode(led_pin_user[1], OUTPUT);
  pinMode(led_pin_user[2], OUTPUT);
  pinMode(led_pin_user[3], OUTPUT);

  nh.initNode();
  nh.subscribe(sub);
}

void loop() {
  nh.spinOnce();
}
```

Завантажте прошивку за допомогою Arduino IDE і запустіть сервер rosserial за допомогою rosserial_python наступним чином. Якщо OpenCR підключений до іншого USB-порту, відмінного від 'ttyACM0', змініть команду виконання '_port: = /dev/ttyACM0', щоб виправити послідовний порт. Як тільки сервер rosserial буде запущений, сервер rosserial і відкритий клієнт будуть відправляти і отримувати пакети через послідовний порт USB.

```
$ rosrun rosserial_python serial_node.py __name:=opencr _port:=/dev/ttyACM0 _baud:=115200
[INFO] [1495609829.326019]: ROS Serial Python Node
[INFO] [1495609829.336151]: Connecting to /dev/ttyACM0 at 115200 baud
[INFO] [1495609831.454144]: Note: subscribe buffer size is 1024 bytes
[INFO] [1495609831.454994]: Setup subscriber on led_out [std_msgs/Byte]
```

Відкрийте новий термінал і введіть команду "rostopic list", щоб перевірити тему "led_out".

```
$ rostopic list
/diagnostics
/led_out
/rosout
/rosout_agg
```

Наступні команди "rostopic pub' запишуть значення в повідомлення і опублікують його в темі "led_out".

```
$ rostopic pub -1 led_out std_msgs/Byte 1      → USER1 LED On
$ rostopic pub -1 led_out std_msgs/Byte 2      → USER2 LED On
$ rostopic pub -1 led_out std_msgs/Byte 4      → USER3 LED On
$ rostopic pub -1 led_out std_msgs/Byte 8      → USER4 LED On
$ rostopic pub -1 led_out std_msgs/Byte 0      → LED Off
```

Кнопка

У цьому прикладі також використовується тип даних "std_msgs / Byte", як і в прикладі LED, але "button" оголошується як ім'я теми для вузла "pub_button' для публікації статусу кнопки на сервері. Наступний приклад коду публікує натиснуте або відпущене стан кнопок SW1 і SW2 відкритої плати у вигляді байтового значення. Стан кнопки публікується з регулярним інтервалом в 50 мс.

b_Button.ino

```
#include <ros.h>
#include <std_msgs/Byte.h>

ros::NodeHandle nh;

std_msgs::Byte button_msg;
ros::Publisher pub_button("button", &button_msg);

void setup()
{
  nh.initNode();
  nh.advertise(pub_button);

  pinMode(BDPIN_PUSH_SW_1, INPUT);
```

```
  pinMode(BDPIN_PUSH_SW_2, INPUT);
}

void loop()
{
  uint8_t reading = 0;
  static uint32_t pre_time;


  if (digitalRead(BDPIN_PUSH_SW_1) == HIGH)
  {
    reading |= 0x01;
  }
  if (digitalRead(BDPIN_PUSH_SW_2) == HIGH)
  {
    reading |= 0x02;
  }

  if (millis()-pre_time >= 50)
  {
    button_msg.data = reading;
    pub_button.publish(&button_msg);
    pre_time = millis();
  }

  nh.spinOnce();
}
```

Завантажте приклад кнопки і запустіть rosserial_python з ПК наступним чином, і на екрані з'явиться повідомлення про налаштування видавця кнопок.

```
$ rosrun rosserial_python serial_node.py __name:=opencr _port:=/dev/ttyACM0 _baud:=115200
[INFO] [1495609931.875745]: ROS Serial Python Node
[INFO] [1495609931.885488]: Connecting to /dev/ttyACM0 at 115200 baud
[INFO] [1495609934.000344]: Note: publish buffer size is 1024 bytes
[INFO] [1495609934.001180]: Setup publisher on button [std_msgs/Byte]
```

Переконайтеся, що у вас є кнопка тема з командою "список ростопіків".

```
$ rostopic list
/button
/diagnostics
/rosout
/rosout_agg
```

Відкрийте нове вікно терміналу і введіть наступну команду для відображення стану опублікованої кнопки кожні 50 мс.

```
$ rostopic echo button
data: 1
---
data: 1
---
data: 1
---
data: 1
---
data: 1
```

Виміряйте Вхідну Напругу

Як показано на рис. 122, Вхідна напруга може бути виміряна шляхом зчитування значення АЦП 3,3 В з регульованим рівнем

вхідної напруги після схеми дільника напруги. Відрегульоване Вхідна напруга (Vout) можна розрахувати за формулою дільника напруги; Vout = (10/57) * Вхідна напруга. Тому фактичну вхідну напругу можна розрахувати, розділивши Vout на 0,1754 (≈10/57). У Opencart вони отримують потужність у функції Voltage (), що виконує перетворення, тому Ви можете використовувати цю функцію для обчислення фактичної вхідної напруги за зчитаним значенням АЦП.

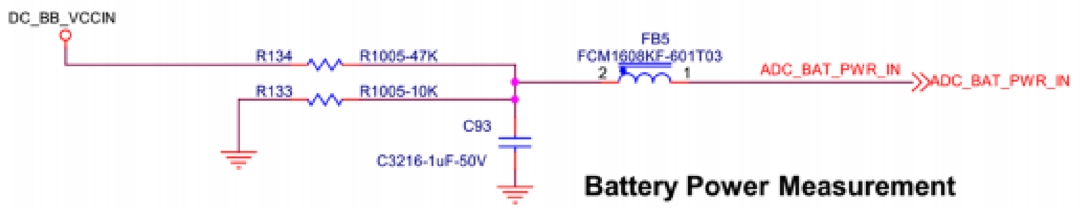


Рис. 122 Схема вимірювання напруги

Щоб представити точну напругу, в прикладі вимірювання напруги використовується тип даних 'std_msgs / Float32'. Оголосіть вузол видавця ' pub_voltage "з темою" voltage", і виміряна Вхідна напруга буде публікуватися кожні 50 мс.

c_Voltage.ino

```
#include <ros.h>
#include <std_msgs/Float32.h>

ros::NodeHandle nh;

std_msgs::Float32 voltage_msg;
ros::Publisher pub_voltage("voltage", &voltage_msg);

void setup()
{
  nh.initNode();
  nh.advertise(pub_voltage);
}

void loop()
{
  static uint32_t pre_time;


  if (millis()-pre_time >= 50)
  {
    voltage_msg.data = getPowerInVoltage();
    pub_voltage.publish(&voltage_msg);
    pre_time = millis();
  }

  nh.spinOnce();
}
```

Запустіть сервер rosserial з наступною командою.

```
$ rosrun rosserial_python serial_node.py __name:=opencr _port:=/dev/ttyACM0 _baud:=115200
[INFO] [1495609160.098041]: ROS Serial Python Node
[INFO] [1495609160.108219]: Connecting to /dev/ttyACM0 at 115200 baud
[INFO] [1495609162.224307]: Note: publish buffer size is 1024 bytes
[INFO] [1495609162.225184]: Setup publisher on voltage [std_msgs/Float32]
```

Щоб перевірити вхідну напругу OpenCR, відкрийте нове вікно терміналу і прочитайте значення в розділі "напруга' за допомогою команди "rostopic echo".

```
$ rostopic echo voltage
data: 12.1300001144
---
data: 12.1099996567
---
data: 12.1300001144
---
data: 12.1099996567
```

IMU

Для передачі значення датчика IMU зі стандартним типом повідомлення ROS використовується повідомлення 'sensor_msgs / Imu'. У наступному прикладі обчислене відношення за допомогою бібліотеки датчиків IMU перетворюється на тип повідомлення "sensor_msgs / Imu" і публікується повідомлення через тему. Згенеруйте tf за допомогою 'base_link', щоб встановити його в якості еталонного tf для датчика IMU, і підключіть 'base_link' до датчика IMU, щоб зміна відношення можна було відстежувати за значенням 'base_link'.

d_IMU.ino

```
#include <ros.h>
#include <sensor_msgs/Imu.h>
#include <tf/tf.h>
#include <tf/transform_broadcaster.h>
#include <IMU.h>



ros::NodeHandle nh;

sensor_msgs::Imu imu_msg;
ros::Publisher imu_pub("imu", &imu_msg);



geometry_msgs::TransformStamped tfs_msg;
tf::TransformBroadcaster tfbroadcaster;

cIMU imu;
```

```
void setup()
{
  nh.initNode();
  nh.advertise(imu_pub);
  tfbroadcaster.init(nh);

  imu.begin();
}

void loop()
{
  static uint32_t pre_time;

  imu.update();

  if (millis()-pre_time >= 50)
  {
    pre_time = millis();

    imu_msg.header.stamp    = nh.now();
    imu_msg.header.frame_id = "imu_link";

    imu_msg.angular_velocity.x = imu.gyroData[0];
    imu_msg.angular_velocity.y = imu.gyroData[1];
    imu_msg.angular_velocity.z = imu.gyroData[2];
    imu_msg.angular_velocity_covariance[0] = 0.02;
    imu_msg.angular_velocity_covariance[1] = 0;
    imu_msg.angular_velocity_covariance[2] = 0;
    imu_msg.angular_velocity_covariance[3] = 0;
    imu_msg.angular_velocity_covariance[4] = 0.02;
    imu_msg.angular_velocity_covariance[5] = 0;
    imu_msg.angular_velocity_covariance[6] = 0;
    imu_msg.angular_velocity_covariance[7] = 0;
    imu_msg.angular_velocity_covariance[8] = 0.02;

    imu_msg.linear_acceleration.x = imu.accData[0];
    imu_msg.linear_acceleration.y = imu.accData[1];
    imu_msg.linear_acceleration.z = imu.accData[2];
```

```
imu_msg.linear_acceleration_covariance[0] = 0.04;
imu_msg.linear_acceleration_covariance[1] = 0;
imu_msg.linear_acceleration_covariance[2] = 0;
imu_msg.linear_acceleration_covariance[3] = 0;
imu_msg.linear_acceleration_covariance[4] = 0.04;
imu_msg.linear_acceleration_covariance[5] = 0;
imu_msg.linear_acceleration_covariance[6] = 0;
imu_msg.linear_acceleration_covariance[7] = 0;
imu_msg.linear_acceleration_covariance[8] = 0.04;

imu_msg.orientation.w = imu.quat[0];
imu_msg.orientation.x = imu.quat[1];
imu_msg.orientation.y = imu.quat[2];
imu_msg.orientation.z = imu.quat[3];

imu_msg.orientation_covariance[0] = 0.0025;
imu_msg.orientation_covariance[1] = 0;
imu_msg.orientation_covariance[2] = 0;
imu_msg.orientation_covariance[3] = 0;
imu_msg.orientation_covariance[4] = 0.0025;
imu_msg.orientation_covariance[5] = 0;
imu_msg.orientation_covariance[6] = 0;
imu_msg.orientation_covariance[7] = 0;
imu_msg.orientation_covariance[8] = 0.0025;

imu_pub.publish(&imu_msg);

tfs_msg.header.stamp    = nh.now();
tfs_msg.header.frame_id = "base_link";
tfs_msg.child_frame_id  = "imu_link";
tfs_msg.transform.rotation.w = imu.quat[0];
tfs_msg.transform.rotation.x = imu.quat[1];
tfs_msg.transform.rotation.y = imu.quat[2];
tfs_msg.transform.rotation.z = imu.quat[3];

tfs_msg.transform.translation.x = 0.0;
tfs_msg.transform.translation.y = 0.0;
tfs_msg.transform.translation.z = 0.0;
```

```
    tfbroadcaster.sendTransform(tfs_msg);
  }


  nh.spinOnce();
}
```

Завантажте приклад і запустіть сервер rosserial з наступною командою для створення видавців 'imu' i'tf'.

```
$ rosrun rosserial_python serial_node.py __name:=opencr _port:=/dev/ttyACM0 _baud:=115200
[INFO] [1495611663.941723]: ROS Serial Python Node
[INFO] [1495611663.946220]: Connecting to /dev/ttyACM0 at 115200 baud
[INFO] [1495611666.075668]: Note: publish buffer size is 1024 bytes
[INFO] [1495611666.076638]: Setup publisher on imu [sensor_msgs/Imu]
[INFO] [1495611666.146240]: Setup publisher on /tf [tf/tfMessage]
```

Відкрийте нове вікно терміналу і перевірте дані Повідомлення теми " imu ' наступним чином.

```
$ rostopic echo /imu
header:
  seq: 686
  stamp:
    secs: 1495611700
    nsecs: 369472074
  frame_id: imu_link
orientation:
  x: 0.0232326872647
  y: -0.0115436725318
  z: -4.04381135013e-05
  w: 0.999659180641
orientation_covariance: [0.0024999999441206455, 0.0, 0.0, 0.0, 0.0024999999441206455, 0.0, 0.0,
0.0, 0.0024999999441206455]
angular_velocity:
  x: 0.0
  y: 0.0
  z: 0.0
angular_velocity_covariance: [0.019999999552965164, 0.0, 0.0, 0.0, 0.019999999552965164, 0.0,
0.0, 0.0, 0.019999999552965164]
linear_acceleration:
```

```
  x: 370.0
  y: 754.0
  z: 16228.0
linear_acceleration_covariance: [0.03999999910593033, 0.0, 0.0, 0.0, 0.03999999910593033, 0.0,
0.0, 0.0, 0.03999999910593033]
---
```

Щоб візуалізувати дані IMU, використовуйте RViz, який може графічно представляти дані IMU в просторі 3DD. Введіть наступну команду для запуску RViz.

```
$ rviz
```

Перейдіть в розділ Глобальні параметри → фіксований кадр і виберіть "base_link" на панелі параметрів Rviz Displays, як показано на рис. 123. Потім натисніть кнопку Додати в нижній частині панелі параметрів дисплеїв, щоб додати " осі "до елемента опції, і виберіть" imu_link " для системи відліку, щоб побачити зміну осі на екрані відповідно до ставлення OpenCR.

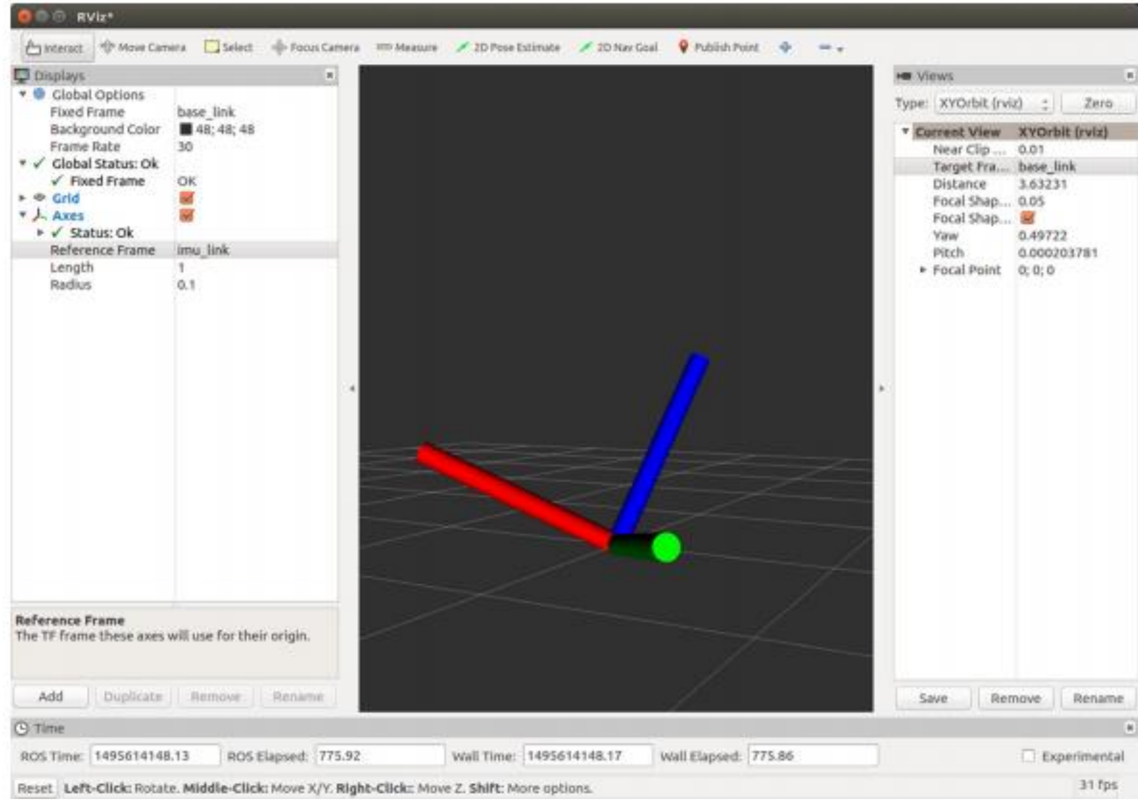


Рис. 123 Візуалізовані дані IMU у Rviz

9.3. Прошивка TurtleBot 3

OpenCR має вбудовану бібліотеку rosserial для TurtleBot3 і може завантажити прошивку TurtleBot3 як приклад. Оскільки файли прикладів надаються в якості вихідних кодів, вони можуть бути змінені користувачами. Прошивка TurtleBot 3 поширюється через менеджер плати Arduino IDE. Тому, якщо версія Board manager змінена, оновіть board manager і завантажте останню версію прошивки.

Як показано на рис. 124, прошивку TurtleBot3 можна завантажити, вибравши Файл → приклади → turtlebot3 → turtlebot3_burger → turtlebot3_core, а також завантажити на плату за допомогою кнопки "завантажити" в Arduino IDE.

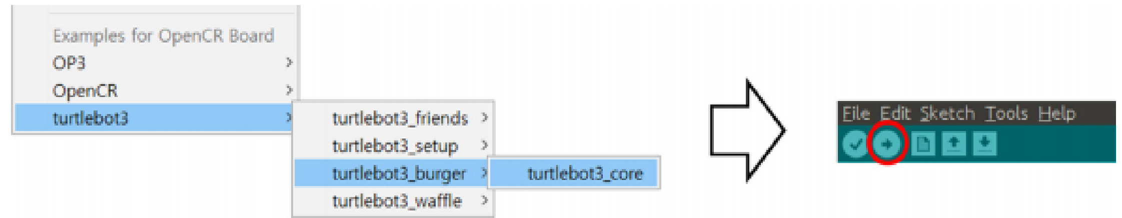


Рис. 124 Прошивка для Burger TurtleBot3

Для бургера TurtleBot3 і вафлі (Waffle Pi) існує різниця між монтажним положенням і радіусом повороту приводу Dynamixel. Конкретні значення налаштувань були використані для обох моделей у файлі turtlebot3_core_config.H. при зміні монтажного положення приводу Dynamixel необхідно також змінити значення параметра.

```
turtlebot3_core_config.h
#define WHEEL_RADIUS            0.033          // meter
#define WHEEL_SEPARATION        0.160          // meter
#define TURNING_RADIUS          0.080          // meter
#define ROBOT_RADIUS            0.105          // meter
```

Щоб використовувати інформацію в TurtleBot3 Burger, запустіть вузол 'rosserial_python' з наступною командою. Коли це робиться, створюється вузол " turtlebot3_core "і команда переміщення підписується з розділу" /cmd_vel", як показано на рис. 125, в той час як інформація про одометрії (/odom), інформація про іду (/imu), інформація про Датчик (/sensor_state) публікуються. Ті ж теми використовуються для TurtleBot 3 Waffle і Waffle Pi. Більш детальну інформацію див. у главі 10.

```
$ rosrun rosserial_python serial_node.py __name:=turtlebot3_core _port:=/dev/ttyACM0
[INFO] [1500275719.375458]: ROS Serial Python Node
[INFO] [1500275719.380338]: Connecting to /dev/ttyACM0 at 57600 baud
[INFO] [1500275721.496849]: Note: publish buffer size is 1024 bytes
[INFO] [1500275721.497162]: Setup publisher on sensor_state [Turtlebot3_msgs/SensorState]
[INFO] [1500275721.499622]: Setup publisher on imu [sensor_msgs/Imu]
[INFO] [1500275721.502328]: Setup publisher on cmd_vel_rc100 [geometry_msgs/Twist]
[INFO] [1500275721.507266]: Setup publisher on odom [nav_msgs/Odometry]
[INFO] [1500275721.511984]: Setup publisher on joint_states [sensor_msgs/JointState]
[INFO] [1500275721.568189]: Setup publisher on /tf [tf/tfMessage]
[INFO] [1500275721.571585]: Note: subscribe buffer size is 1024 bytes
[INFO] [1500275721.571865]: Setup subscriber on cmd_vel [geometry_msgs/Twist]
[INFO] [1500275721.573235]: Start Calibration of Gyro
[INFO] [1500275724.046148]: Calibrattion End
```

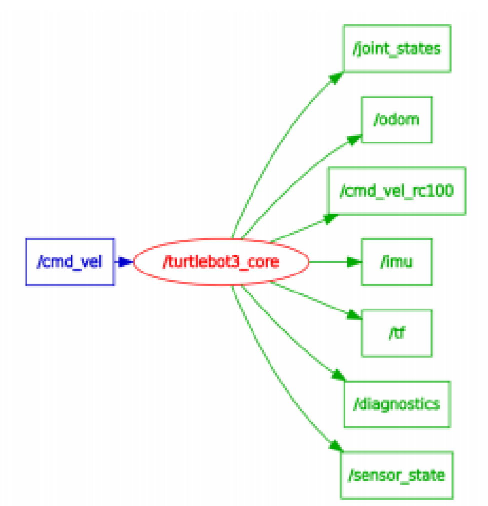


Рис. 125 Публікація та опис теми вузла turtlebot3_core

Прошивку TurtleBot3 Waffle і Waffle Pi можна завантажити, вибравши Файл → приклади →turtlebot3 → turtlebot3_waffle → turtlebot3_core і натиснувши кнопку Завантажити з Arduino IDE, як показано на рис. 126.

Рис. 126 TurtleBot3 Waffle та Waffle Pi Firrmware

```
turtlebot3_core_config.h
#define WHEEL_RADIUS              0.033        // meter
#define WHEEL_SEPARATION          0.287        // meter
#define TURNING_RADIUS            0.1435       // meter
#define ROBOT_RADIUS              0.220        // meter
```

Диски Dynamixel, що використовуються в TurtleBot3, мають заводські ідентифікатори та конфігурації. Якщо ви замінюєте привід або змінюєте налаштування для іншої мети, ви повинні переналаштувати значення налаштування для повторного використання з TurtleBot3. Приклад зміни налаштування приводу включений, тому давайте змінимо значення Налаштування.

Завантажити інсталяційну прошивку

Як показано на рис. 127, В меню прикладу перейдіть в turtlebot3 → turtlebot3_setup →turtlebot3_setup_motor, завантажте прошивку на плату OpenCR і продовжите процес налаштування. Після завершення налаштування завантажте відповідну прошивку TurtleBot 3 в OpenCR.

Рис. 127 Приклад налаштування приводу TurtleBot3

Натисніть кнопку завантажити на Arduino IDE для завантаження, а після завершення завантаження натисніть значок послідовного монітора в правому верхньому куті програми, як показано на рис. 128. Підключіть динамік до відкривачки. Зверніть увагу, що ця прошивка працює тільки з одним Dynamixel, тому вам потрібно підключати тільки один Dynamixel за раз.

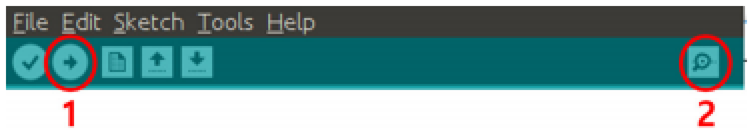


Рис. 128 Завантаження та запуск монітору послідовного порту

Змінити налаштування Dynamixel

При запуску послідовного монітора відображається меню Налаштування Dynamixel, як показано на рис. 129. TurtleBot3 складається з двох дисків Dynamixel зліва і справа відповідно, тому виберіть Dynamixel в залежності від положення збірки. Щоб налаштувати лівий двигун, введіть "1'.

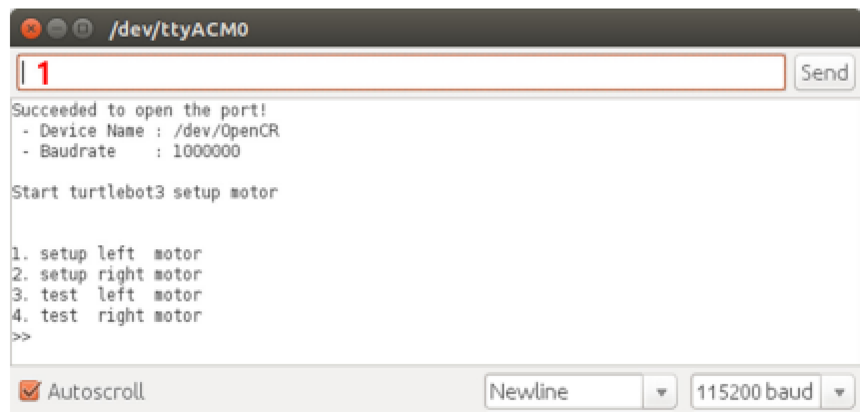


Рис. 129 Монітор налаштування приводу TurtleBot3

Щоб запобігти помилкам введення, знову відображається меню підтвердження. Щоб продовжити внесення змін, введіть "Y'.

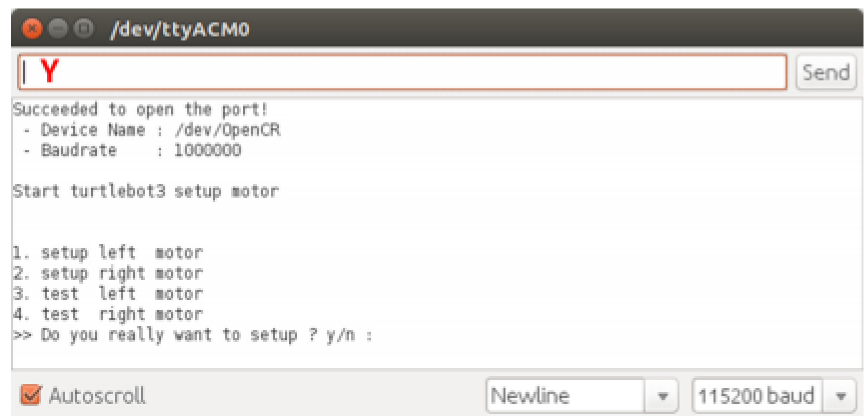


Рис. 130 Меню підтвердження налаштувань

Якщо ви введете "Y', інструмент налаштування почне пошук підключеного Динамікса, використовуючи різні параметри та ідентифікатор. Якщо динаміксель знайдений, він буде скинутий для

конфігурації TurtleBot 3. Після завершення налаштування виводиться повідомлення "OK".

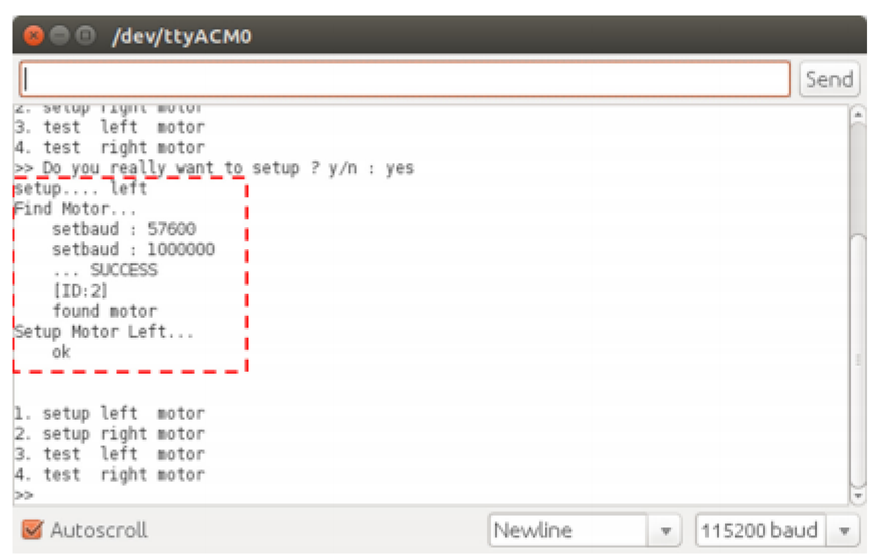


Рис. 131 Setup complete message

Тест Dynamixel

Завершіть процедуру налаштування та переконайтеся, що зміна була зроблена правильно. Якщо вибрати одне з тестових меню для двигуна, то підключений Динаміксель з правильною конфігурацією буде повторювати обертання за годинниковою стрілкою і проти годинникової стрілки. Щоб завершити тест, знову натисніть клавішу Enter. Щоб перевірити лівий динамік, введіть "3", Як показано на рис. 132, і введіть "4' для правого динаміка.

```
/dev/ttyACM0
3                                                        Send
4. test  right motor
>> Do you really want to setup ? y/n : yes
setup.... left
Find Motor...
    setbaud : 57600
    setbaud : 1000000
    ... SUCCESS
    [ID:1]
    found motor
Setup Motor Left...
    ok


1. setup left  motor
2. setup right motor
3. test  left  motor
4. test  right motor
>> test.... left
Test Motor Left...
Autoscroll                          Newline        115200 baud
```

Рис. 132 Тестове меню Dynamixel

Ми обговорювали, як інтегрувати ROS у вбудовану систему. Вбудована система тісно пов'язана з роботами, які вимагають управління в режимі реального часу. Навчившись інтегрувати ROS, ви допоможете своїм майбутнім розробкам роботів. Глави 10, 11, 12 і 13 будуть присвячені практичним прикладам роботів mobilee, що використовують вбудовану систему, описану в цьому розділі.

Розділ 10. Мобільні роботи

10.1. Робот, Підтримуваний ROS

Роботів, підтримуваних ROS, можна знайти на сторінці Wiki (http://robots.ros.org/). на базі ROS розроблено близько 180 роботів. Деякі з них включають в себе користувальницькі роботи, які публічно випускаються розробниками, і це помітний список, враховуючи, що одна система підтримує таких різноманітних роботів. Найвідомішими роботами серед них є PR2, розроблені Willow Garage і TurtleBot. Обидва робота є стандартними платформами ROHS, в розробці яких брали участь Willow Garage або Open Robotics (раніше OSRF). У цьому розділі ми зосередимося на TurtleBot, який був розроблений як платформа початкового рівня.

10.2. Серія TurtleBot3

TurtleBot-це стандартний платформний робот ROS. Черепаха походить від робота-черепахи, який був приведений в дію освітньою мовою комп'ютерного програмування" Логотип " в 1967 році. Крім того, вузол turtlesim, який вперше з'являється в базовому підручнику ROS, являє собою програму, що імітує командну систему програми Logo turtle. Він також використовується для створення значка Черепахи як символу ROS, як показано на рис. 133. Дев'ять точок, використаних в логотипі ROS, походять від заднього панцира черепахи. TurtleBot, який походить від черепахи логотипу, призначений для легкого навчання люди, які новачки в ROS через TurtleBot, а також навчають мови комп'ютерного програмування за

допомогою логотипу. З тих пір TurtleBot став стандартною платформою ROS, яка є найпопулярнішою платформою серед розробників і студентів.

Рис. 133 Символи кожної версії ROS

Існує 3 версії серії TurtleBot (див. Рис. 134). TurtleBot1 був розроблений Таллі (менеджером платформи Open Robotics) і Мелоні (генеральним директором Fetch Robotics) з Willow Garage поверх дослідницького робота iRobot Roomba Create для розгортання ROS. Він був розроблений в 2010 році і продається з 2011 року. У 2012 році TurtleBot 2 був розроблений компанією Yujin Robot на базі дослідницького робота iClebo Kobuki. У 2017 році TurtleBot3 був розроблений з функціями, що доповнюють відсутні функції його попередників і вимоги користувачів. TurtleBot3 використовує інтелектуальний привід ROBOTIS' Dynamixel ' для водіння.

Рис. 134 Зліва направо TurtleBot1, TurtleBot2, TurtleBot3 (останні три моделі)

TurtleBot3-це невеликий, доступний за ціною, програмований мобільний робот на базі ПЗУ для використання в освіті, дослідженнях, хобі та прототипуванні продуктів. Мета TurtleBot3-різко зменшити розмір платформи і знизити ціну, не жертвуючи її функціональністю і якістю, в той же час пропонуючи розширюваність. TurtleBot 3 може бути налаштований різними способами в залежності від того, як ви реконструюєте механічні деталі і використовуєте додаткові деталі, такі як комп'ютер і датчик. Крім того, TurtleBot3 еволюціонує з економічним і малогабаритним SBC, який підходить для надійної вбудованої системи, 360-градусного датчика відстані і технології 3D-друку.

10.3. Обладнання TurtleBot3

Існують три офіційні моделі TurtleBot3, TurtleBot3 Burger, Waffle і Waffle Pi, як показано на рис. 135, і ця книга буде в основному обговорюватися на основі Turtlebot3 Burger, якщо не вказано інше. Крім того, TurtleBot3 підтримує різні структури та обладнання, такі як Монстр, Танк, носій, і ці варіації називаються TurtleBot3 + [суфікс]. Основними компонентами TurtleBot3 є приводи, SBC для управління ROS, датчик для шолома і навігації, реструктурований механізм, вбудована плата OpenCR, використовувана в якості субконтролера, зірочки, які можуть використовуватися з шинами і гусеницями, а також 3-елементна літій-полі батарея. TurtleBot3

Вафля відрізняється від бургера формою платформи, на якій зручно монтувати компоненти, використанням приводів з більш

високим крутним моментом, високопродуктивним SBC з процесором Intel, глибинною камерою RealSense для 3D-розпізнавання від Intel. TurtleBot3 Waffle Pi має таку ж форму, що і модель Waffle, але ця модель використовується Raspberry Pi як модель Burger, а камера theRaspberry Pi робить її більш доступною.

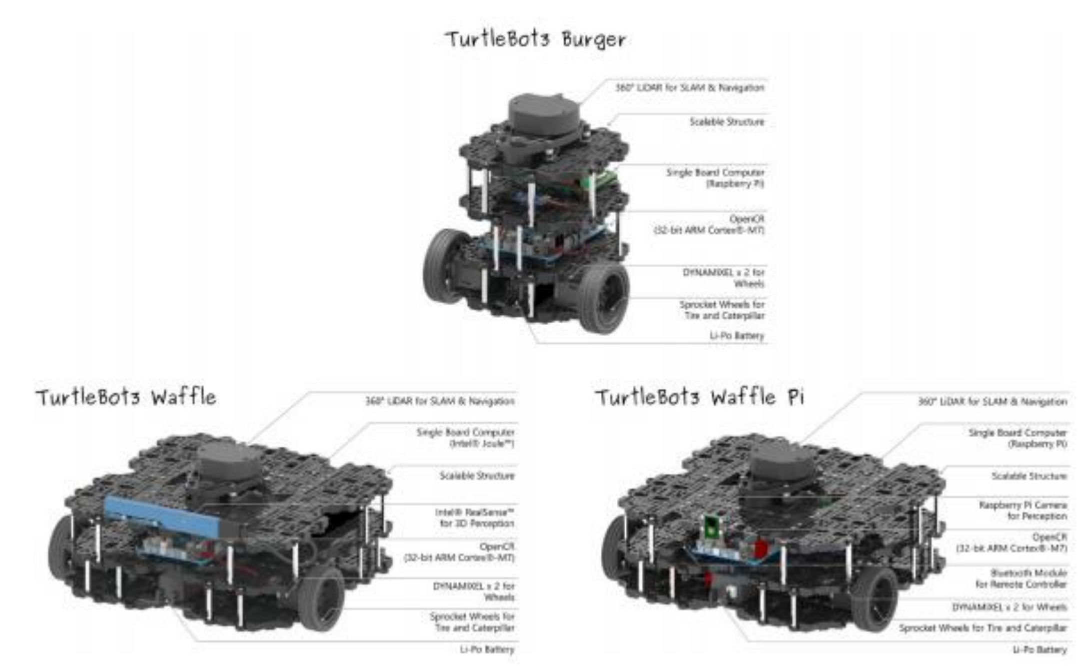


Рис. 135 Апаратна конфігурація TurtleBot3

Файли проектування 3D CAD TurtleBot3 доступні через хмарний інструмент 3D CAD "Onshape" і дозволяють всім членам проектної групи отримувати доступ до загальних файлів проектування за допомогою смартфонів, планшетів і ПК, незалежно від операційної системи. Ви можете не тільки перевірити кожен компонент TurtleBot3 за допомогою веб-браузера, але і завантажити деталі в свій

репозиторій і зробити свої власні деталі, змінивши дизайн. Після завантаження файлу STL деталі можна роздрукувати за допомогою 3D-принтера. Файли для кожної моделі можна знайти в розділі Open Sourcee, представленому в якості додатку на офіційній Вікі-сторінці TurtleBot3.

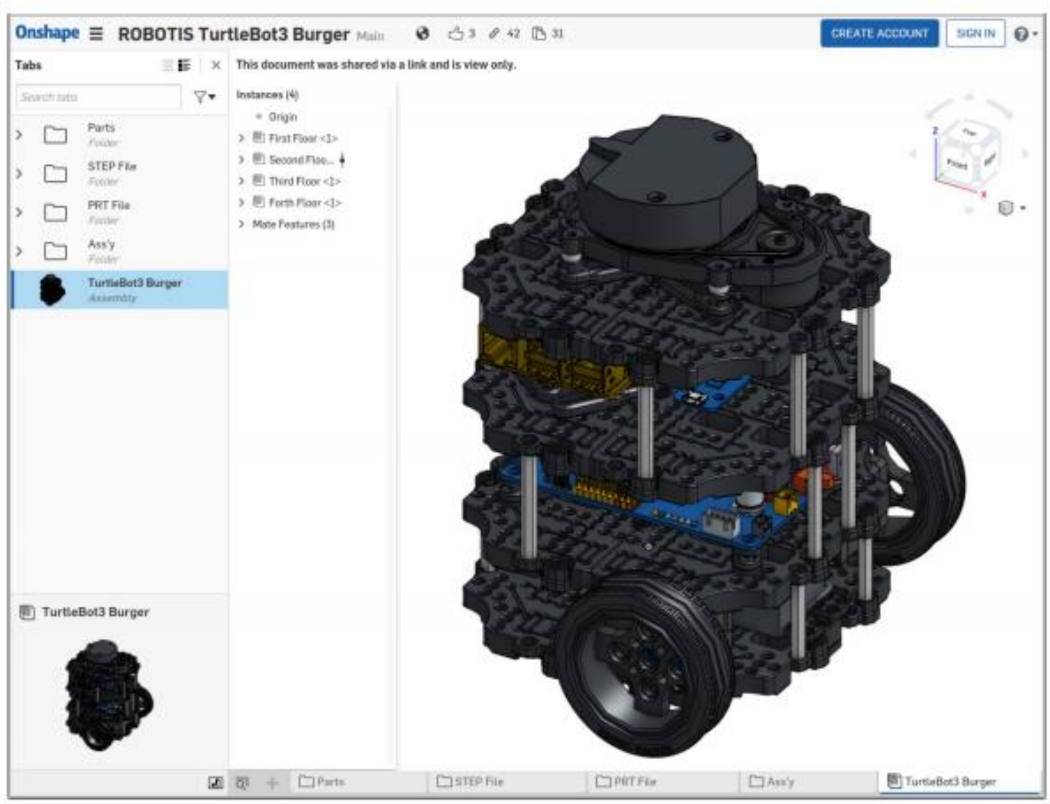


Рис. 136 Обладнання з відкритим кодом TurtleBot3

Офіційний TurtleBot3 Wiki

Вищезгадані апаратні деталі TurtleBot3 і основний зміст, описаний в цьому розділі, також можна знайти в офіційній Вікі

TurtleBot3 за наступним посиланням. Щоб дізнатися ROS за допомогою TurtleBot 3, зверніться до посилання нижче.

http://turtlebot3.robotis.com

Апаратне забезпечення з відкритим вихідним кодом для TurtleBot 3

Файли апаратного дизайну та програмне забезпечення TurtleBot 3 відкриті для публіки. Якщо вам потрібні апаратні файли для кожної моделі TurtleBot3 і для Opencv, що використовуються в якості субконтролера TurtleBot 3, зверніться до списку і посиланнях нижче. Якщо не вказано інше, все обладнання є відкритим вихідним кодом і відповідає ліцензії hardware Statement of Principles and Definition v1.0.

Opencv: https://github.com/ROBOTIS-GIT/OpenCR-Hardware

TurtleBot3 Burger: http://www.robotis.com/service/download.php?no=676

TurtleBot3 Waffle: http://www.robotis.com/service/download.php?no=677

TurtleBot3 Waffle Pi: http://www.robotis.com/service/download.php?no=678

TurtleBot3 Friends OpenManipulator Chain: http://www.robotis.com/service/download.php?no=679

TurtleBot3 Friends Segway: http://www.robotis.com/service/download.php?no=680

TurtleBot3 Friends Conveyor: http://www.robotis.com/service/download.php?no=681

TurtleBot3 Friends Monster: http://www.robotis.com/service/download.php?no=682

TurtleBot3 Friends Tank: http://www.robotis.com/service/download.php?no=683

TurtleBot3 Friends Omni: http://www.robotis.com/service/download.php?no=684

TurtleBot3 Friends Mecanum: http://www.robotis.com/service/download.php?no=685

TurtleBot3 Friends Bike: http://www.robotis.com/service/download.php?no=686

TurtleBot3 Friends Road Train: http://www.robotis.com/service/download.php?no=687

TurtleBot3 Friends Real TurtleBot: http://www.robotis.com/service/download.php?no=688

TurtleBot3 Friends Carrier: http://www.robotis.com/service/download.php?no=689

10.4. Програмне забезпечення TurtleBot3

Програмне забезпечення TurtleBot3 складається з прошивки (FW) плати Opencart, використовуваної в якості субконтролера, і 4 пакетів ROS. Прошивка TurtleBot3 також називається "turtlebot3_core" в тому сенсі, що це ядро TurtleBot3, яке вже було описано в главі 9 Embedded System. Він використовує OpenCR як субконтролер для

оцінки місця розташування робота шляхом обчислення значення кодера Dynamixel, який є приводним двигуном TurtleBot3, або для управління швидкістю відповідно до команди, опублікованої програмним забезпеченням верхнього рівня. Крім того, прискорення та кутова швидкість отримуються від 3-осьового датчика прискорення та 3-осьового гіроскопа, встановленого на OpenCR для оцінки напрямку руху робота, а стан акумулятора також вимірюється та передається через теми.

Пакет Ros TurtleBot3 включає в себе 4 пакети, які є "turtlebot3", 'turtlebot3_msgs', 'turtlebot3_simulations' і 'turtlebot3_applications'. Пакет "turtleBot3" містить модель робота TurtleBot3, пакет SLAM і навігації, пакет дистанційного керування і пакет bringup. Пакет turtlebot3_msgs містить файли повідомлень, що використовуються в turtlebot3, пакет turtlebot3_simulations містить пакети, пов'язані з симуляцією, а пакет turtlebot3_applications містить додатки.

рограмне забезпечення з відкритим вихідним кодом для TurtleBot 3

Програмне забезпечення TurtleBot 3 розкривається громадськості як відкритий вихідний код. Завантажувач OpenCR, який використовується в якості субконтролера TurtleBot3, прошивка для розробки з Arduino IDE і прошивка для управління TurtleBot3, а також пакети ROS (turtlebot3, turtlebot3_msgs, turtlebot3_simulations, turtlebot3_applications) також доступні в якості відкритого вихідного

коду. Ліцензії на програмне забезпечення з відкритим кодом відрізняються для кожного джерела, але в основному відповідають Apache license 2.0, а деякі програми використовують 3-клаузульну ліцензію BSD і GPLv3.

https://github.com/ROBOTIS-GIT/OpenCR

https://github.com/ROBOTIS-GIT/turtlebot3

https://github.com/ROBOTIS-GIT/turtlebot3_msgs

https://github.com/ROBOTIS-GIT/turtlebot3_simulations

https://github.com/ROBOTIS-GIT/turtlebot3_applications

10.5. Середовище розробки TurtleBot 3

Середовище розробки TurtleBot3 можна розділити на віддалений ПК, який виконує дистанційне керування, SLAM, навігаційний пакет, і КОМП'ЮТЕР TurtleBot, який управляє компонентами робота і збирає сенсорну інформацію, як показано на рис.137. Обидва ПК дуже схожі з точки зору середовища розробки, але використовувані ними пакети налаштовуються по-різному в залежності від продуктивності і призначення ПК. Базове середовище розробки для обох ПК вимагає Linux (тут Ubuntu 16.04 і сумісні Linux mint і Ubuntu MATE) в якості базової операційної системи, а також ROS (Kinetic Kame). Докладніше про налаштування середовища розробки ROS див.розділ 3. Крім того, зверніться до наступного веб-сайту для налаштування PC, TurtleBot і OpenCR.

http://turtlebot3.robotis.com

Якщо у вас встановлені Linux і ROS, ви можете встановити програмне забезпечення, пов'язане з TurtleBot3. Всі ці методи установки описані у вищезгаданій turtlebot3 wiki, але тут ми коротко підсумовуємо їх і пояснюємо тільки метод установки. Давайте встановимо залежні пакети та пакети TurtleBot3 на операційний комп'ютер (цей комп'ютер буде називатися віддаленим ПК), який керує TurtleBot3. Однак ми виключили пакет turtlebot3_applications, який містить різні приклади, не згадані в цій книзі.

Installation command for Dependent Packages (Remote PC)

```
$ sudo apt-get install ros-kinetic-joy ros-kinetic-teleop-twist-joy ros-kinetic-teleop-twist-keyboard ros-kinetic-laser-proc ros-kinetic-rgbd-launch ros-kinetic-depthimage-to-laserscan ros-kinetic-rosserial-arduino ros-kinetic-rosserial-python ros-kinetic-rosserial-server ros-kinetic-rosserial-client ros-kinetic-rosserial-msgs ros-kinetic-amcl ros-kinetic-map-server ros-kinetic-move-base ros-kinetic-urdf ros-kinetic-xacro ros-kinetic-compressed-image-transport ros-kinetic-rqt-image-view ros-kinetic-gmapping ros-kinetic-navigation
```

TurtleBot3 Package Installtion (Remote PC)

```
$ cd ~/catkin_ws/src/
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3.git
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git
$ cd ~/catkin_ws && catkin_make
```

Потім встановіть пакети TurtleBot 3 і залежні пакети, такі як пакет датчиків, в SBC off TurtleBot3.

Installation command for Dependent Package (TurtleBot3)

```
$ sudo apt-get install ros-kinetic-joy ros-kinetic-teleop-twist-joy ros-kinetic-teleop-twist-keyboard ros-kinetic-laser-proc ros-kinetic-rgbd-launch ros-kinetic-depthimage-to-laserscan ros-kinetic-rosserial-arduino ros-kinetic-rosserial-python ros-kinetic-rosserial-server ros-kinetic-rosserial-client ros-kinetic-rosserial-msgs ros-kinetic-amcl ros-kinetic-map-server ros-kinetic-move-base ros-kinetic-urdf ros-kinetic-xacro ros-kinetic-compressed-image-transport ros-kinetic-rqt-image-view ros-kinetic-gmapping ros-kinetic-navigation
```

TurtleBot3 Package Installtion (TurtleBot3)

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3.git
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git
$ git clone https://github.com/ROBOTIS-GIT/hls_lfcd_lds_driver.git
$ cd ~/catkin_ws && catkin_make
```

Рис. 137 налаштування пульта дистанційного керування для TurtleBot3

Після установки всього програмного забезпечення важливо налаштувати Мережеве середовище, як показано на рис. 10-5. Для

отримання детальної інформації про те, як змінити налаштування ROS_HOSTNAME та ROS_MASTER_URI, зверніться до розділу 3.2 та розділу 8.3. TurtleBot3 використовує настільний ПК або ноутбук як віддалений ПК, який діє як майстер для запуску roscore і приймає на себе такі складні елементи управління процесом, як SLAM і навігація. SBC в TurtleBot3 відповідає за роботу компонентів робота і збір даних датчиків. Нижче наведено приклад налаштування віддаленого управління, коли Ros Master працює на віддаленому ПК.

Перевірте IP-адресу віддаленого комп'ютера

У вікні терміналу використовуйте команду "ifconfig" для перевірки IP-адреси віддаленого комп'ютера (наприклад, 192.168.7.100).

Налаштування ROS_HOSTNAME і ROS_MASTER_URI на віддаленому ПК

Змініть налаштування ROS_HOSTNAME та ROS_MASTER_URI у файлі'~/. bashrc ' наступним чином:

```
export ROS_HOSTNAME=192.168.7.100
export ROS_MASTER_URI=http://${ROS_HOSTNAME}:11311
```

Перевірте IP-адресу TurtleBot 3

Перевірте IP-адресу TurtleBot3 за допомогою команди "ifconfig' у вікні терміналу. Припустимо, IP-адреса TurtleBot3-192.168.7.200. В якості запобіжного заходу TurtleBot повинен знаходитися в тій же мережевій зоні, що і віддалений комп'ютер.

Налаштування ROS_HOSTNAME та ROS_MASTER_URI SBC TurtleBot3 змінюють налаштування ROS_HOSTNAME та ROS_MASTER_URI у файлі'~/.bashrc ' наступним чином

```
export ROS_HOSTNAME=192.168.7.200
export ROS_MASTER_URI=http://192.168.7.100:11311
```

Тепер ми завершили розробку середовища розробки TurtleBot 3. У наступному розділі давайте керувати TurtleBot3 за допомогою різних пакетів ROS, починаючи з пульта дистанційного керування.

10.6. TurtleBot 3 Пульт Дистанційного Керування

Давайте поглянемо на пульт дистанційного керування TurtleBot3. Майже всі пристрої, які можуть бути підключені до ПК, такі як клавіатура, контролер Bluetooth RC100, джойстик PS3, джойстик XBOX 360, Wii Remote, Nunchuk, Android app, LEAP Motion, можуть бути використані для управління TurtleBot3. Для отримання додаткової інформації зверніться до розділу телеоперація за адресою 'http://turtlebot3.robotis.com/'. у цьому розділі ми будемо використовувати найбільш часто використовувану клавіатуру і джойстик PS3, який часто використовується для управління роботом.

Запуск roscore (віддалений комп'ютер)

На віддаленому КОМП'ЮТЕРІ виконайте наступну команду для запуску roscore. Роско повинен бути виконаний тільки один раз.

```
$ roscore
```

Запустіть файл turtlebot 3_robot.запуск [TurtleBot]

У TurtleBot запустіть файл запуску ' turtlebot3_robot.запуск ' наступним чином. Цей файл запуску виконує 'turtlebot3_core", який відповідає за зв'язок з OpenCR, контролером TurtleBot3, і вузлом" hls_lfcd_lds_driver", який управляє LDS, який є датчиком відстані 360 градусів.

```
$ roslaunch turtlebot3_bringup turtlebot3_robot.launch --screen
```

--screen option

Roslaunch може виконувати кілька вузлів одночасно. Однак повідомлення від кожного вузла за замовчуванням не відображаються. При необхідності ви можете використовувати опцію '-- screen', щоб побачити всі приховані повідомлення під час роботи. Якщо ви використовуєте roslaunch, рекомендується додати цю опцію.

Запустіть файл turtlebot 3_teleop_key.launch [віддалений ПК]

Запустіть файл turtlebot3_teleop_key.launch з віддаленого комп'ютера.

```
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch --screen
```

При запуску цього файлу запуску виконується вузол turtlebot3_teleop_keyboard' і у вікні терміналу з'являється наступне повідомлення. Цей вузол приймає ключові входи" w"," a"," d", " x " і передає роботу поступальну і обертальну швидкості в м/с і рад/с відповідно. Клавіша "Пробіл' і " s 'скинуть поступальну і обертальну швидкість на "0", щоб зупинити рух TurtleBot3.

```
Control Your TurtleBot3!
---------------------------
Moving around:
        w
   a    s    d
        x

w/x : increase/decrease linear velocity
a/d : increase/decrease angular velocity
space key, s : force stop

CTRL-C to quit
```

Якщо ви хочете використовувати джойстик PS3 замість клавіатури для телеоперації TurtleBot 3, Встановіть залежний пакет для джойстика на віддаленому ПК наступним чином. Потім запустіть файл " teleop.launch "в пакеті "teleop_twist_joy", щоб керувати роботом за допомогою джойстика PS3. Джойстик PS3 повинен бути підключений до віддаленого ПК через Bluetooth.

```
$ sudo apt-get install ros-kinetic-joy ros-kinetic-joystick-drivers ros-kinetic-teleop-twist-joy
$ roslaunch teleop_twist_joy teleop.launch --screen
```

Давайте візуалізуємо стан робота на RViz. Перед виконанням RViz 3D-модель повинна бути встановлена як Burger. Використовуйте наведену нижче команду для позначення 3D-моделі TurtleBot 3 Burger. Якщо ви використовуєте Turtlebot3 Waffle або Waffle Pi, ви повинні встановити параметр TURTLEBOT3_MODEL як "waffle' або

'waffle_pi' замість "burger". Потім виконайте ' turtlebot3_model.запустіть ' файл, і rviz буде завантажений.

```
$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_bringup turtlebot3_remote.launch
$ rosrun rviz rviz -d `rospack find turtlebot3_description`/rviz/model.rviz
```

Коли rviz буде виконаний, 3D-Модель TurtleBot3 Burger буде відображатися на початку координат RGB разом з TF кожного суглоба робота, як показано на рис. 138. Крім того, дані про відстань від датчика 360 degreeLEDSS можуть відображатися навколо робота у вигляді червоних точок.

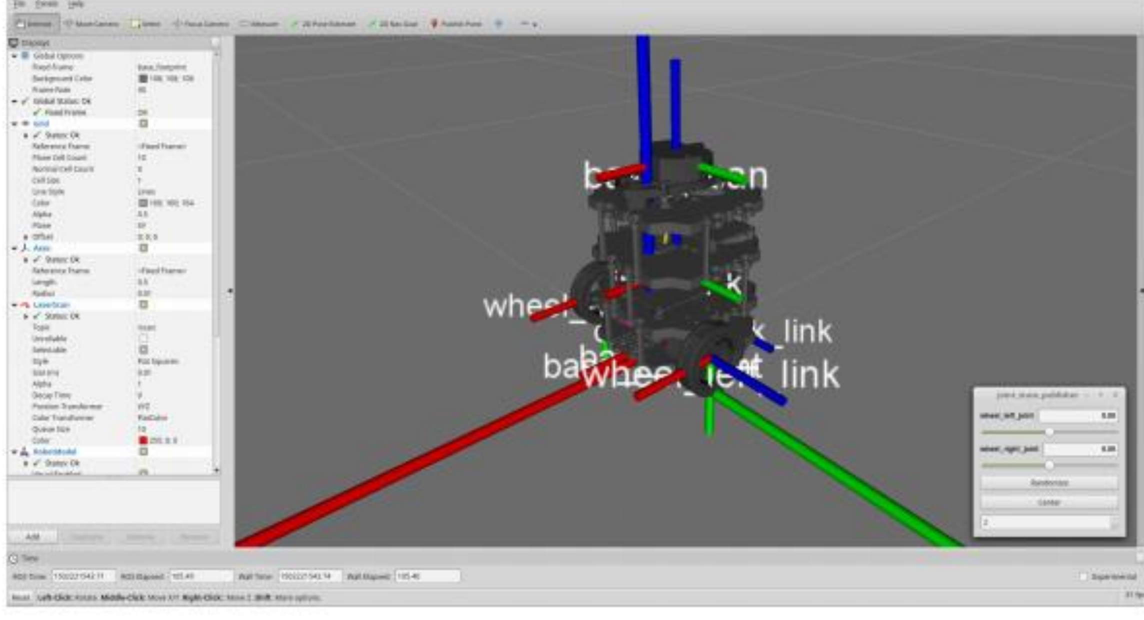


Рис. 138 Візуалізація TurtleBot3

Для того щоб управляти TurtleBot3, деякі роботи повинні виконуватися з локального TurtleBot3 SBC, що буде незручно для

користувача працювати взад і вперед на двох комп'ютерах. Для вирішення цієї проблеми рекомендується віддалений доступ до TurtleBot 3 SBC з віддаленого комп'ютера за допомогою SSH. Це дозволяє виконувати команди на TurtleBot 3 SBC з віддаленого ПК. Ось приклад того, як віддалено підключитися до TurtleBot3 SBC з віддаленого ПК. Додаткові відомості див. в примітках по SSH.

```
$ ssh turtlebot@192.168.7.200
```

SSH(Secure Shell)

SSH відноситься до Програми або протоколу, який дозволяє вам увійти на інший комп'ютер в мережі і виконувати команди на віддаленій системі і копіювати файли в іншу систему. Він часто використовується при підключенні до віддаленого комп'ютера і відправляє команду з вікна терміналу в Linux. Для цього ssh-додаток має бути встановлено наступним чином.

```
$ sudo apt-get install ssh
```

Щоб підключитися до віддаленого комп'ютера (в даному випадку TurtleBot3), використовуйте наступну команду для підключення у вікні терміналу. Як тільки з'єднання встановлено з ПК, команди можна вводити точно так же, як при використанні локального комп'ютера.

```
$ ssh username @ ip address of the remote PC
```

У випадку Raspberry Pi (TurtleBot3 Burger і Waffle Pi), так як SSH-сервер Ubuntu MATE 16.04.x і Raspbian за замовчуванням вимкнено. Якщо ви хочете включити SSH, будь ласка, зверніться до документів нижче.

https://www.raspberrypi.org/documentation/remote-access/ssh/

https://ubuntu-mate.org/raspberry-pi/

10.7. TurtleBot3 Тема

Якщо ви запускаєте roscore тільки на віддаленому ПК і не запускаєте ніяких інших вузлів, команда 'rostopic list' поверне '/rosout' і '/ rosout_agg'. Давайте запустимо TurtleBot3, запустивши ' turtlebot3_robot.запустіть файл' у вікні терміналу TurtleBot3 SBC, як ми це зробили для пульта дистанційного керування TurtleBot вище. Коли файл запуску робота TurtleBot3 буде виконаний, вузол turtlebot3_core і вузол turtlebot3_lds будуть запущені, і повідомлення, опубліковані з кожного вузла, такі як стан з'єднання, виконавчі механізми і IMU, можуть бути отримані у вигляді тем.

```
$ roslaunch turtlebot3_bringup turtlebot3_robot.launch --screen
```

Наприклад, наступна команда "rostopic list" може перевірити, що різні теми публікуються або підписуються:

```
$ rostopic list
/cmd_vel
/cmd_vel_rc100
/diagnostics
/imu
/joint_states
```

```
/odom
/rosout
/rosout_agg
/rpms
/scan
/sensor_state
/tf
```

Крім того, запустіть " turtlebot3_teleop_key.launch " на віддаленому комп'ютері, як ви це зробили для пульта дистанційного керування TurtleBot3:

```
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch --screen
```

Щоб отримати більш детальну інформацію про вузол і тему, запустіть 'rqt_graph', як показано в наступному прикладі. Потім ви можете перевірити теми, опубліковані та підписані TurtleBot 3, Як показано на рис.139.

```
$ rqt_graph
```

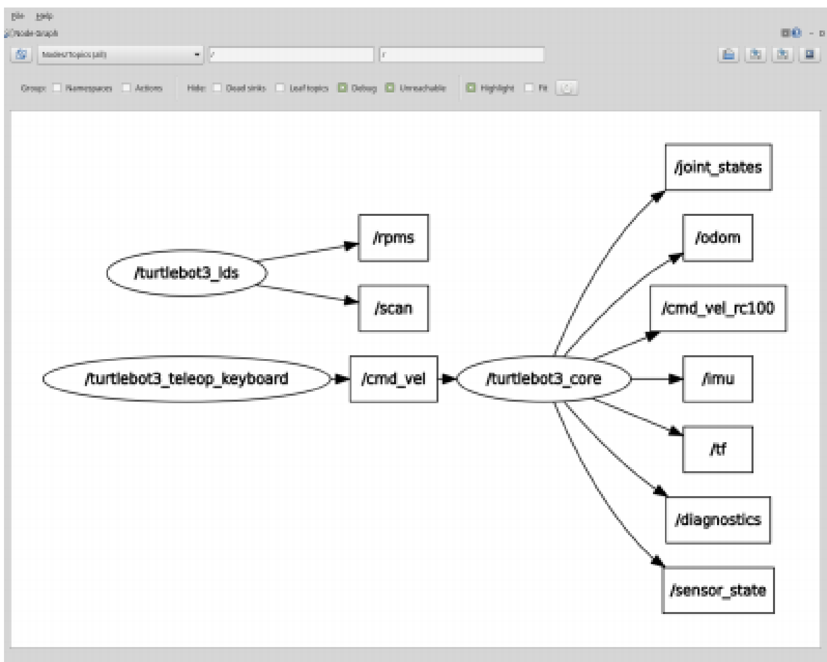


Рис. 139 Вузол і темаTurtleBot3

Теми, згадані вище, можна розділити на теми підписки, отримані TurtleBot3, і Теми Публікації, передані з TurtleBot3. Серед них теми підписки показані в таблиці нижче. Вам не потрібно знати всі теми підписки, але було б гарною практикою навчитися їх використовувати. Перш за все, давайте поглянемо на "cmd_vel'. Це корисна тема для управління роботом, і користувач може керувати прямим, зворотним, лівим і правим обертанням робота за допомогою цієї теми.

Таблиця 20

Subscribed Topics of the TuttleBot3

Topic Name	Format	Function
motor_power	std_msgs/Bool	Dynamixel Torque On/Off
reset	std_msgs/Empty	Reset Odometry and IMU Data
sound	turtlebot3_msgs/Sound	Output Beep Sound
cmd_vel	geometry_msgs/Twist	Control the translational and rotational speed of the robot. unit in m/s, rad/s (actual robot control)

* Topics used in TurtleBot3 may change depending on the purpose.

Для раніше згаданих тем підписки робот отримує і обробляє теми, опубліковані користувачем. Важко перевірити кожну тему в цій книзі, тому давайте просто скористаємося кількома темами підписки. У наступному прикладі двигун зупиняється командою "rostopic pub' у вікні терміналу.

```
$ rostopic pub /motor_power std_msgs/Bool "data: 0"
```

Далі, давайте контролювати швидкість TurtlebBot3. X і y, що використовуються тут, є поступальними швидкостями, а одиниця виміру-це стандарт RoHS m/S. A z - швидкість обертання в рад/с. Коли значення x дорівнює 0,02, як показано в наступному прикладі, TurtleBot3 просувається зі швидкістю 0,02 м/с в позитивному напрямку осі x.

```
$ rostopic pub /cmd_vel geometry_msgs/Twist "linear:
  x: 0.02
  y: 0.0
  z: 0.0
angular:
  x: 0.0
```

```
  y: 0.0
  z: 0.0"
```

Коли значення z задано рівним 1,0, як показано в наступному прикладі, TurtleBot 3 буде обертатися проти годинникової стрілки зі швидкістю 1,0 рад/с.

```
$ rostopic pub /cmd_vel geometry_msgs/Twist "linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 1.0"
```

Теми, які публікує TurtleBot3, - це діагностика, налагодження, пов'язані з датчиками теми, такі як "joint_states", "sensor_state", 'odom', 'version_info' і 'tf'.

Вам не потрібно знати всі опубліковані теми, але було б гарною практикою навчитися їх використовувати. Зокрема, "odom" для інформації про одометрію, "tf "для інформації про перетворення,"

joint_states " для спільної інформації та пов'язані з сенсорною інформацією теми необхідні при використанні TurtleBot3 далі.

Таблиця 21

Topic Name	Format	Function
sensor_state	turtlebot3_msgs/SensorState	Topic that contains the values of the sensors mounted on the TurtleBot3
battery_state	sensor_msgs/BatteryState	Contains battery voltage and status
scan	sensor_msgs/LaserScan	Topic that confirms the scan values of the LiDAR mounted on the TurtleBot3
imu	sensor_msgs/Imu	Topic that includes the attitude of the robot based on the acceleration and gyro sensor.
odom	nav_msgs/Odometry	Contains the TurtleBot3's odometry information based on the encoder and IMU
tf	tf2_msgs/TFMessage	Contains the coordinate transformation such as base_footprint and odom

Topic Name	Format	Function
joint_states	sensor_msgs/JointState	Checks the position (m), velocity (m/s) and effort (N·m) when the wheels are considered as joints.
diagnostics	diagnostic_msgs/DiagnosticArray	Contains self diagnostic information
version_info	turtlebot3_msgs/VersionInfo	Contains the TurtleBot3 hardware, firmware, and software information
cmd_vel_rc100	geometry_msgs/Twist	This topic is published when the Bluetooth-based controller, RC100, is connected to control the velocity (m/s) and angular speed (rad/s) of mobile robot.

* Topics used in TurtleBot3 may change depending on the purpose.

Опубліковані теми, згадані вище, передають значення датчика робота, стан двигуна і положення робота за темами. У цьому розділі ви підпишетеся на деякі теми і перевірите поточний стан робота.

Тема "sensor_state" в основному присвячена аналоговим датчикам, підключеним до вбудованої плати OpenCR. Ви можете отримати таку інформацію, як бампер, скеля, кнопка, left_encoder, right_encoder в якості наступного прикладу.

```
$ rostopic echo /sensor_state
stamp:
  secs: 1500378811
  nsecs: 475322065
bumper: 0
cliff: 0
button: 0
left_encoder: 35070
right_encoder: 108553
battery: 12.0799999237
---
```

Тема "одом" може бути використана для отримання інформації про одометрію, яка записує інформацію про водіння. У TurtleBot 3 Основна інформація про одометрію може бути отримана на основі гіроскопа і кодера. Одометрія необхідна для навігації.

```
$ rostopic echo /odom
header:
  seq: 30
  stamp:
    secs: 1500379033
    nsecs: 274328964
  frame_id: odom
child_frame_id: ''
pose:
  pose:
    position:
      x: 3.55720114708
      y: 0.655082702637
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.113450162113
      w: 0.993543684483
  covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0]
twist:
  twist:
    linear:
      x: 0.0
      y: 0.0
      z: 0.0
    angular:
      x: 0.0
      y: 0.0
      z: -0.00472585950047
  covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0]
---
```

Тема " tf '- це інформація про переміщення і обертання кожного суглоба робота, перетвореного в відносну координату до "base_footprint", наприклад перетворення координат між "base_footprint', який є центральним положенням робота на площині XY, і odom, який є інформацією одометрії.

```
$ rostopic echo /tf
transforms:
  -
    header:
      seq: 0
      stamp:
        secs: 1500379130
        nsecs: 727869913
      frame_id: odom
    child_frame_id: base_footprint
    transform:
      translation:
        x: 3.55720019341
        y: 0.655082404613
        z: 0.0
      rotation:
        x: 0.0
        y: 0.0
        z: 0.112961538136
        w: 0.993599355221
---
```

Ви також можете використовувати плагін 'tf_tree' в 'rqt', як показано на рис.10-8, для візуалізації інформації tf в графічному середовищі. На рис. 10-8, оскільки інформація про модель робота відсутня і, отже, кожна координата не пов'язана, але при виконанні

перетворення координат з інформацією про модель робота інформація про з'єднання кожного з'єднання може бути використана, як показано на рис. 1.

```
$ rosrun rqt_tf_tree rqt_tf_tree
```

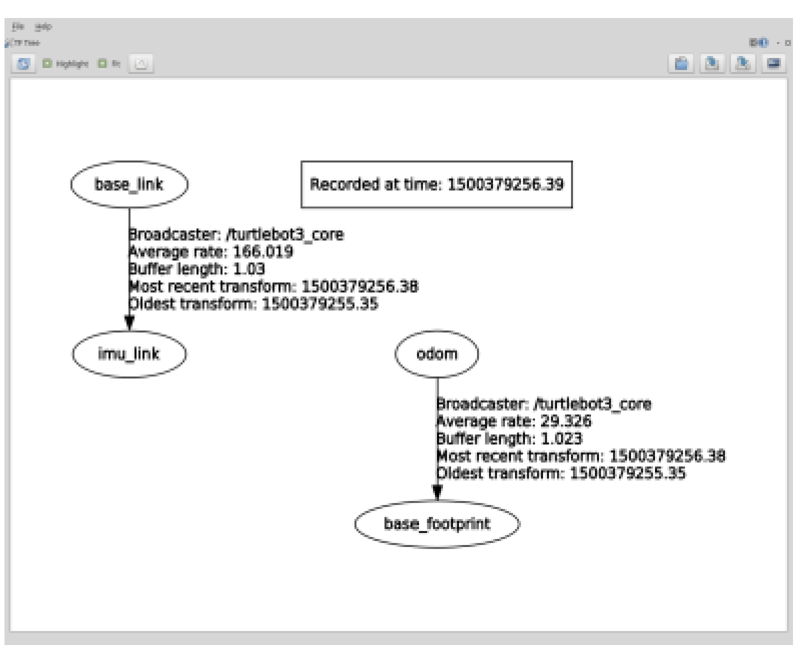


Рис. 140 Візуалізація geретворення координат через tf_tree

Ми завершили тематичний розділ. ROS використовує тему, службу та дію для зв'язку між вузлами, які є процесорами. Зокрема, тема є найбільш широко використовуваним методом передачі повідомлень.

10.8. Моделювання TurtleBot 3 з використанням RViz

TurtleBot3 підтримує середовище розробки, яка може бути запрограмована і розроблена за допомогою віртуального робота в

процесі моделювання. Для цього існують два середовища розробки: одна використовує інструмент 3D-візуалізації "RViz", а інша-3D-симулятор робота "Gazebo'.

У цьому розділі ми спочатку розглянемо, як використовувати RViz. RViz дуже корисний для управління TutleBot3 і тестування SLAM і навігації за допомогою метапакета "turtlebot3_simulations'. Щоб використовувати віртуальне моделювання з цим метапакетом, спочатку слід встановити пакет turtlebot3_fake. Про це йдеться в розділі 10. 5 середовище розробки TurtleBot3'. Якщо ви вже встановили пакет, переходите до наступного розділу.

Щоб запустити віртуального робота, виконайте команду turtlebot3_fake.запустіть ' файл у пакеті turtlebot3_fake, як показано нижче.

```
$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_fake turtlebot3_fake.launch
```

Наведена вище команда завантажує файл 3D-моделювання в пакет turtlebot3_description і виконує вузол turtlebot3_fake_node, який публікує підроблені теми як фактичні публікації робота, і вузол robot_state_publisher, який публікує перетворення кожного колеса в тему tf. Однак сенсорна інформація не може бути використана в RViz, слід використовувати 3D-симулятор, заснований на фізичному движку "альтанка". Як буде пояснено в наступному розділі, ми

розглянемо Odometryy і TF, які можна перевірити під час простого руху.

Щоб візуалізувати TurtleBot3 на RViz, запустіть RViz, перейдіть до [Global Options] → [fixed frame] і виберіть '/odom'. Потім натисніть кнопку " Додати "в лівому нижньому кутку вікна, щоб додати" RobotModel "і відобразити файл 3D-моделі, завантажений з"turtlebot3_fake".запуск' в центрі екрану, як показано на рис.141.

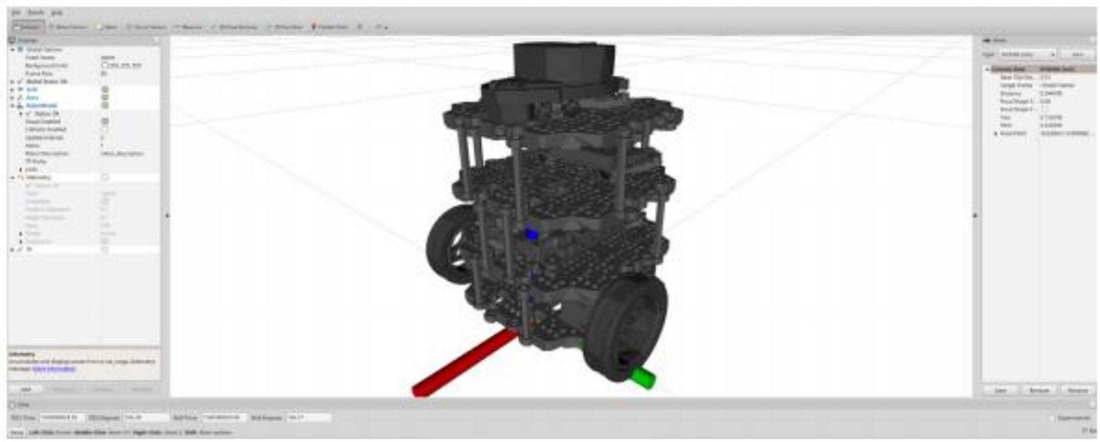


Рис. 141 Завантаження віртуального робота

Тепер давайте запустимо віртуального робота. Запустіть файл turtlebot3_teleop_key.launch з пакета turtlebot3_teleop, який дозволяє віддалено керувати роботом з клавіатури.

```
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

При виконанні файлу turtlebot3_teleop_key.launch буде запущений вузол turtlebot3_teleop_keyboard. У 'turtlebot3_fake_node'

поступальна швидкість і швидкість обертання отримані з теми '/cmd_vel', опублікованої вузлом 'turtlebot3_teleop_keyboard' для віртуальної роботи TurtleBot3. У вікні терміналу, де запущений файл turtlebot3_teleop_key.launch, використовуйте клавіші нижче для управління роботом.

Таблиця 22

клавіша w	Вперед (+0,01 кроку, одиниця виміру = м / сек)
клавіша x	Назад (-0.01 крок, одиниця виміру = м / сек)
клавіша a	Поворот у напрямку CCW (+0.1 крок, одиниця виміру = рад / сек)
клавіша d	Поворот у напрямку CW (-0.1 крок, одиниця виміру = рад / сек)
пробіл або клавіша s	скидання швидкості перекладу і обертання до 0
Ctrl + c	завершити вузол

Тепер, коли ми практикувалися в управлінні віртуальним роботом, давайте перевіримо інші значення теми. Наприклад, як показано на рис. 142, turtlebot3_fake_node отримує команду speed для генерації одометричної інформації. Вузол публікує 'odometry', 'joint_state'' і '' tf '' через тему, щоб їх можна було використовувати для візуалізації руху TurtleBot3 в RViz.

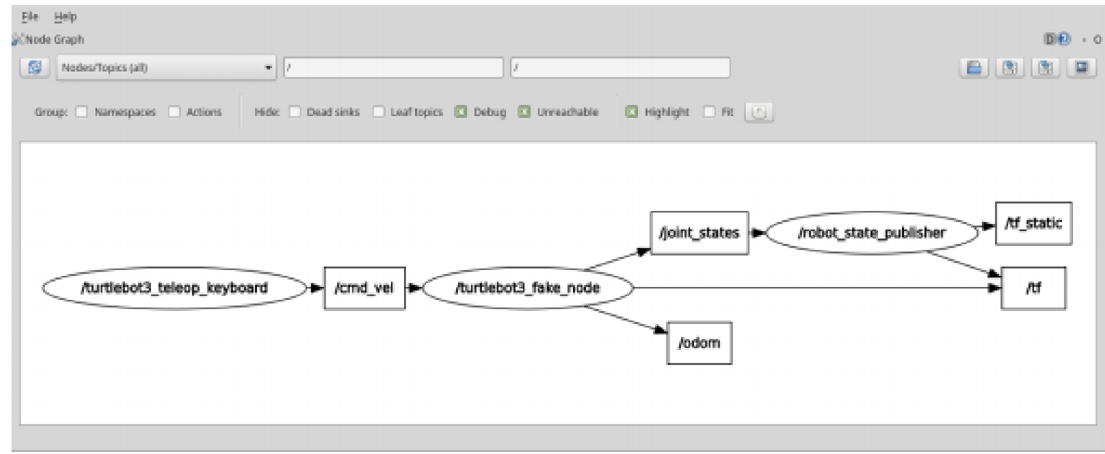


Рис. 142 Візуалізовані вузли та теми за допомогою rqt_graph

Давайте спочатку переконаємося, що інформація про одометрію генерується і публікується належним чином. Хоча команда "rostopic echo / odom" у вікні терміналу може перевірити цю інформацію, давайте візуалізуємо інформацію про одометрію за допомогою RViz. Натисніть кнопку "Додати" в лівому нижньому кутку RViz, потім виберіть вкладку "по темі", Як показано на рис. 143, і додайте "Одометрію", вибравши її. На екрані з'являється червона стрілка, яка вказує одометрію в прямому напрямку горлиці. Зніміть прапорець " Коваріація "в розділі" Одометрія " і відрегулюйте розмір валу і розмір головки, так як початкова стрілка занадто велика для робота.

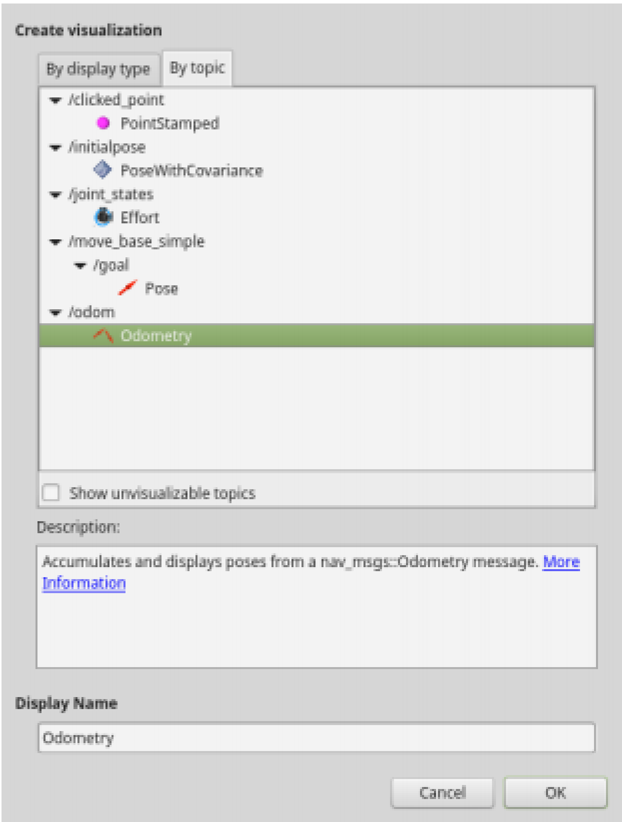


Рис. 143 Adding the odometry display to verify the odom topic

Тепер давайте перейдемо до віртуального TurtleBot3 за допомогою вузла turtlebot3_teleop_keyboard. Як показано на рис. 144, на траєкторії руху робота відображаються червоні стрілки. Ця одометрія є дуже простою інформацією, яка вказує, де в даний момент знаходиться робот. У наведеній вище практиці ми перевірили, що інформація про одометрію відображається правильно.

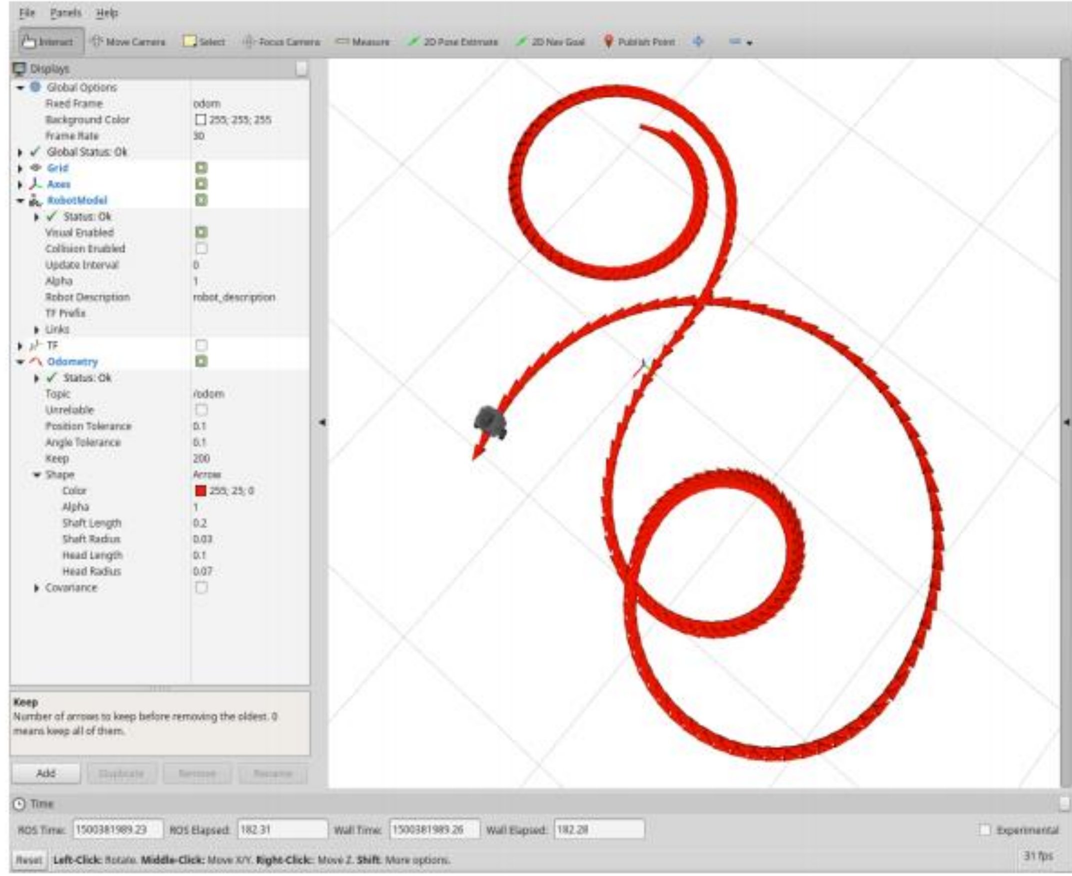


Рис. 144 Movement and odometry information of the virtual TurtleBot Burger

Тема tf, що містить відносні координати компонентів TurtleBot3, може бути перевірена за допомогою команди rostopic, як і раніше, але давайте візуалізуємо її за допомогою RViz, як odom, і візуалізуємо ієрархію за допомогою 'rqt_tf_tree'.

Натисніть кнопку Додати в лівому нижньому кутку RViz і виберіть "TF". Це буде відображати "odom', 'base_footprint', 'imu_link', 'wheel_left_link', 'wheel_right_link" і т. д., як показано на рис. 145. Давайте знову перемістимо віртуальну TurtleBot3 за допомогою вузла

turtlebot3_teleop_keyboard. Коли turtlebot3 рухається, ви можете бачити, як обертаються "wheel_left_link' і "wheel_right_link".

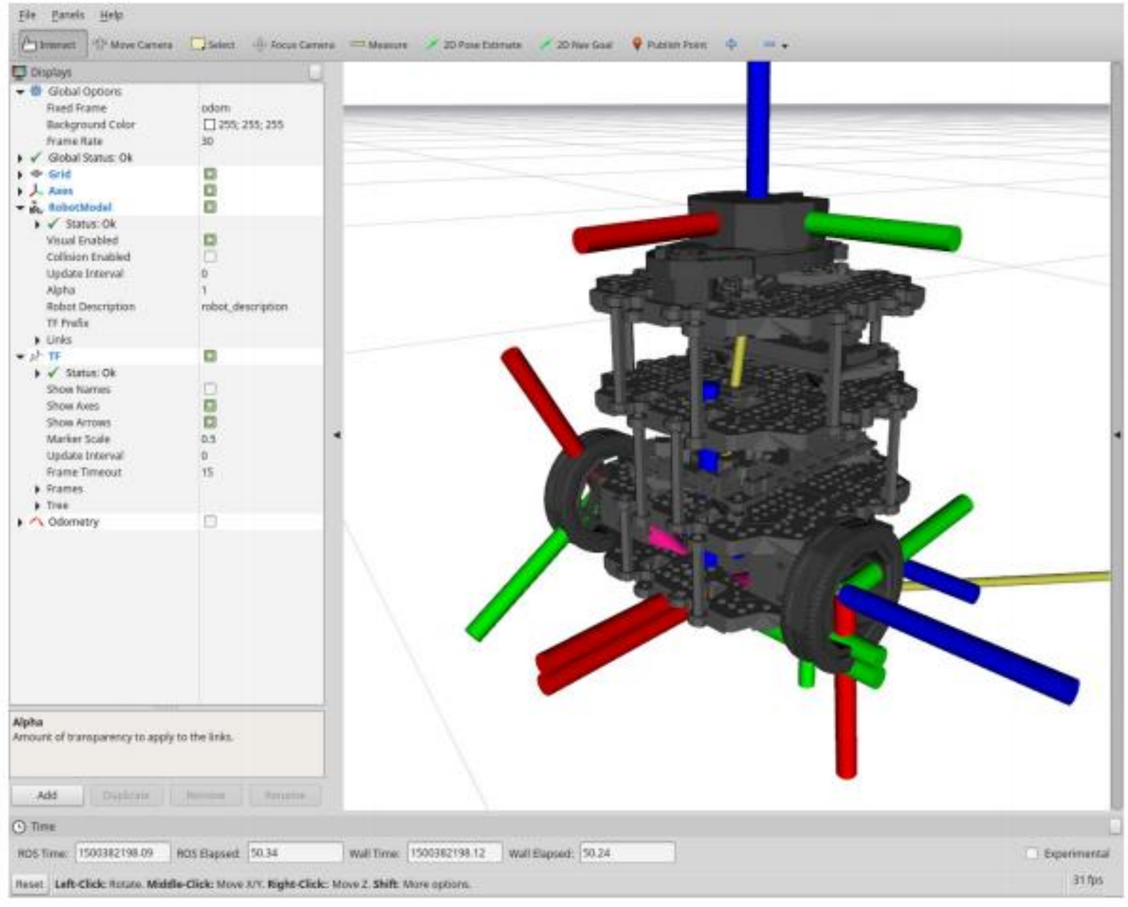


Рис. 145 Візуалізація tf-тем в Rviz

Тепер запустіть 'rqt_tf_tree' з наступною командою. Ми бачимо, що компоненти TurtleBot 3 відносно трансформовані, як показано на рис. 145. Аналогічно можна виразити положення датчиків, які можуть бути встановлені на роботі. Це буде детально розглянуто в наступному розділі.

```
$ rosrun rqt_tf_tree rqt_tf_tree
```

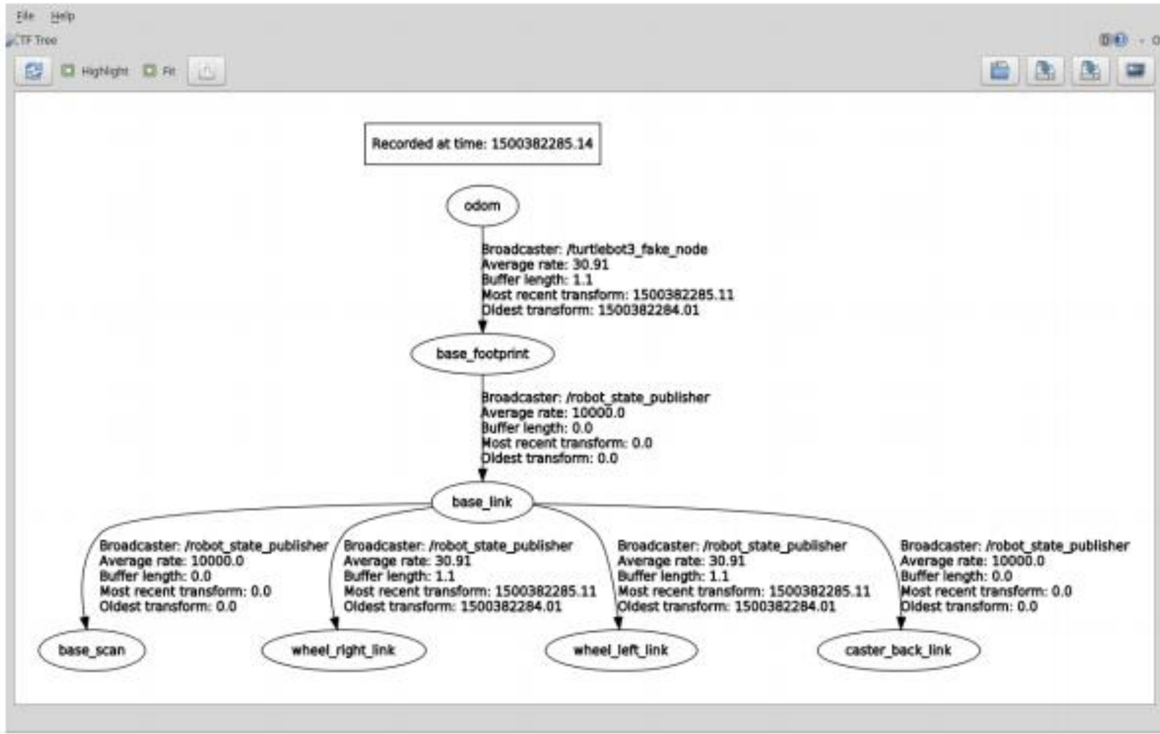


Рис. 146 Візуалізація tf-тем в rqt_tf_tree

10.9. TurtleBot3: моделювання з використанням Альтанки

Gazebo-це 3D-симулятор, який надає роботів, датчики, моделі навколишнього середовища для 3D-моделювання, необхідного для розробки роботів, і пропонує реалістичне моделювання за допомогою свого фізичного движка. Gazebo є одним з найпопулярніших симуляторів з відкритим кодом в останні роки і був обраний в якості офіційного симулятора DARPA Robotics Challenge в США. Це дуже

популярний симулятор в області робототехніки через його високу продуктивність, хоча він і є відкритим вихідним кодом. Крім того, Gazebo розробляється та розповсюджується компанією Open Robotics, яка відповідає за ROS та її спільноту, тому вона дуже сумісна з ROS.

Нижче наведені характеристики *Gazebo.*

Таблиця 23

Характеристики Gazebo

Моделювання динаміки	у перші дні розробки підтримувався тільки ODE(Open Dynamics Engine). Однак починаючи з версії 3.0 для задоволення потреб різних користувачів використовуються різні фізичні движки, такі як Bullet, Simbody і DART.
3d-графіка	Gazebo використовує OGRE (Open-source Graphics Rendering Engines), який часто використовується в іграх, не тільки модель робота, але і світло, тінь і текстура можуть бути реалістично намальовані на екрані.
Датчики і моделювання шуму	підтримуються лазерний далекомір(LRF), 2D / 3D камера, глибинна камера, контактний датчик, датчик сили і крутного моменту і багато іншого, і шум може бути застосований до даних датчика, аналогічним фактичної навколишньому середовищу.

Плагіни	API-інтерфейси надаються для того, щоб користувачі могли створювати роботів, датчики, управління навколишнім середовищем в якості плагіна.
Модель робота	PR2, PIONEER2 DX, iRobot Create і TurtleBot вже підтримуються у вигляді SDF-файлу моделі альтанки, і користувачі можуть додавати своїх власних роботів за допомогою SDF-файлу.
Передача даних по протоколу TCP / IP	симуляція може бути запущена на віддаленому сервері, і використовується Протобуф Google, передача повідомлень на основі сокетів.
Cloud Simulation	Gazebo надає середовище Cloud simulation CloudSim для використання в хмарних середовищах, таких як Amazon, Softlayer і OpenStack.
Інструмент командного рядка	для перевірки та контролю стану симуляції підтримуються як GUI, так і cui інструменти.

Остання версія Gazebo-8.0, а всього п'ять років тому вона була 1.9. Поточна версія-це прийняте за замовчуванням додаток ROS Kinetic Kame, що використовується в цій книзі. Якщо ROS встановлено відповідно до інструкцій у розділі "3.1 встановлення ROS", альтанку можна використовувати без додаткової установки.

А тепер бігом в альтанку. Якщо немає ніяких проблем, ви можете бачити, що альтанка працює так, як показано на рис. 147. На даний момент Gazebo можна розглядати як самостійний симулятор, так як він не пов'язаний з ROS.

```
$ gazebo
```

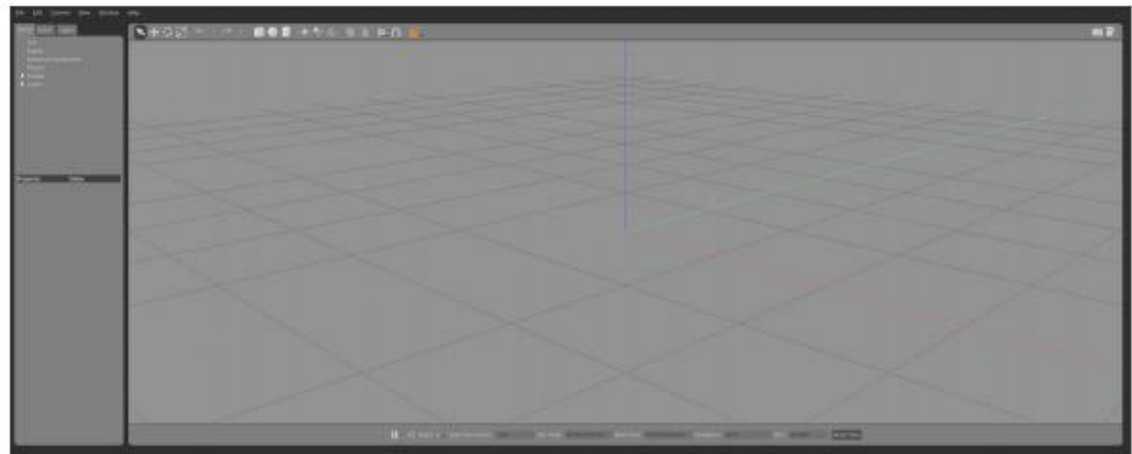


Рис. 147 Початковий екран Gazebo

Давайте встановимо залежні пакети для запуску TurtleBot3 на симуляторі Альтанки. Метапакет "gazebo_ros_pkgs", що з'єднує Gazebo і ROS, вже встановлений, а також потрібен пакет 3D-моделювання для TurtleBot3" turtlebot3_gazebo", але він вже встановлений в середовищі розробки"10.5 TurtleBot3".

Нижче наведено команду для встановлення файлу 3D-моделі в Burger або Waffle, Waffle Pi. Приклад команди заснований на вафлі, яка може отримати інформацію про камеру. Встановіть змінну моделі

TURTLEBOT3_ у Значення 'waffle' за допомогою наступної команди. Якщо команда записана у файлі'~/.bashrc', то вам не потрібно кожен раз встановлювати модель при відкритті нового вікна терміналу.

```
$ export TURTLEBOT3_MODEL=waffle
```

Тепер запустіть файл запуску, як показано в наступному прикладі. Вузли "gazebo', "gazebo_gui"," mobile_base_nodelet_manager"," robot_state_publisher "i" spawn_mobile_base " виконуються разом, і на екрані Gazebo з'являється вафля TurtleBot3, як показано на рис. 148. Gazebo-це 3D-симулятор, який використовує багато ресурсів процесора, графічного процесора та оперативної пам'яті завдяки використанню фізичного двигуна та графічних ефектів. Залежно від апаратних характеристик ПК завантаження програми може зайняти значну кількість часу.

```
$ roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
```

Як показано на наступному малюнку, ви можете бачити, що тільки робот відображається в порожній площині.

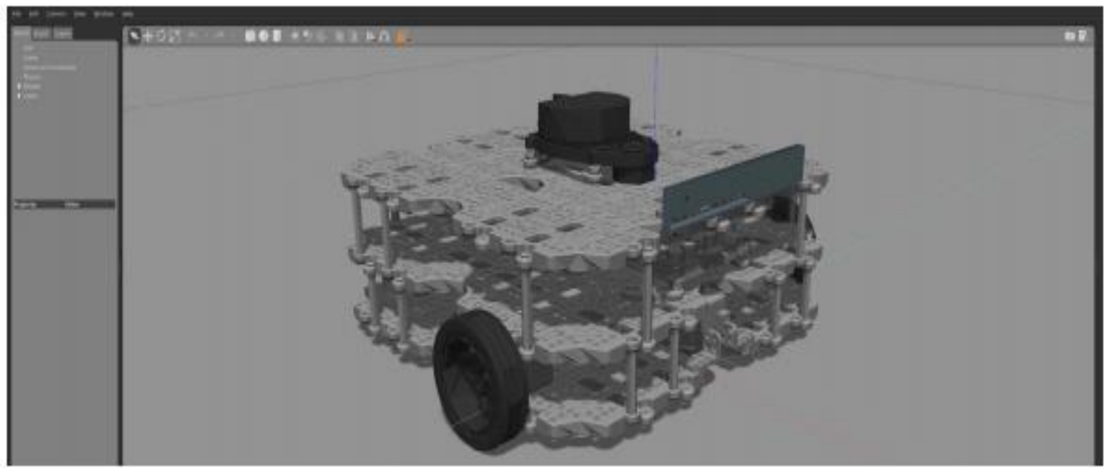


Рис. 148 3D -вид TurtleBot3 на Gazebo

У наведеному вище прикладі в альтанку завантажується тільки робот. Щоб виконати фактичне моделювання, користувач може вказати середовище або завантажити модель середовища, надану Gazebo. Модель середовища може бути додана в Gazebo, натиснувши на кнопку "Вставити" у верхній частині екрана і вибравши файл. Існують різні моделі роботів і об'єктів, а також моделі навколишнього середовища, тому давайте додамо їх при необхідності.

У цій інструкції ми будемо використовувати надану середу. Закрийте поточний активний екран Gazebo, натиснувши кнопку " X " у верхньому лівому куті екрана або введіть [Ctrl + c] у вікні терміналу, де ви запустили Gazebo.

Запустіть 'turtlebot3_world".запустіть файл наступним чином. Світ "turtlebot3_world".файл запуску буде завантажений з уже створеною нами моделлю середовища turtlebot3.world. Модель

навколишнього середовища turtlebot3.world була створена з символу серії TurtleBot, як показано на рис. 149. Якщо ви хочете створити свою власну модель середовища, перевірте файл ' / models / turtlebot3.world 'в пакеті 'turtlebot3_gazebo', щоб побачити, як він налаштований.

```
$ roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

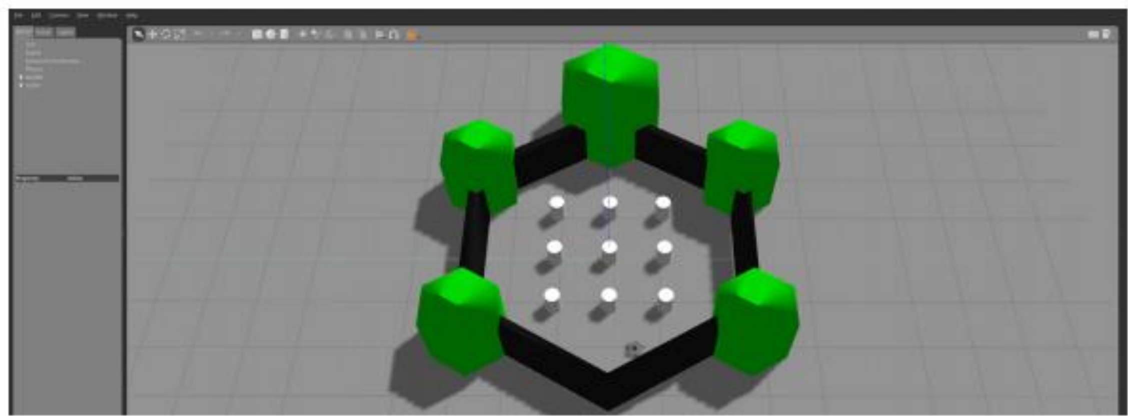


Рис. 149 TurtleBot3 та модель середовища

Тепер запустіть файл запуску віддаленого управління в наступному прикладі, щоб керувати віртуальним TurtleBot3 в середовищі Gazebo за допомогою клавіатури.

```
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

Альтанка досі виглядає досить схожою на симуляцію RViz з попереднього розділу. Альтанка, однак, не тільки призначена для того, щоб виглядати як віртуальний робот, але і може віртуально перевіряти зіткнення, обчислювати положення і використовувати датчик IMU і камеру. Нижче наведено приклад використання цього файлу запуску.

Коли файл виконується, віртуальний TurtleBot3 переміщається випадковим чином в завантаженому середовищі і уникає перешкод, перш ніж врізатися в стіну, як показано на рис. 150. Це відмінний приклад навчання Gazebo.

```
$ export TURTLEBOT3_MODEL=waffle
$ roslaunch turtlebot3_gazebo turtlebot3_simulation.launch
```

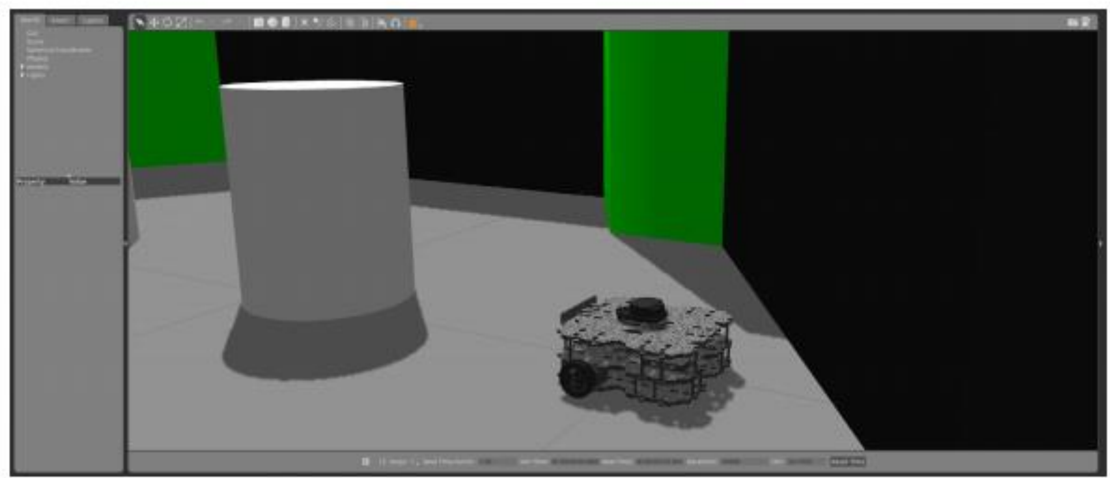


Рис. 150 TurtleBot3 автоматично рухається та уникає перешкод на Gazebo

На додаток до цього, давайте запустимо RViz з наступною командою. Як показано на рис. 151, RViz може візуалізувати положення робота, що працює в альтанці, дані датчика відстані і зображення камери. Цей результат моделювання майже ідентичний роботі реального робота в середовищі modell, розробленої точно так же, як і в Gazebo.

```
$ export TURTLEBOT3_MODEL=waffle
$ roslaunch turtlebot3_gazebo turtlebot3_gazebo_rviz.launch
```

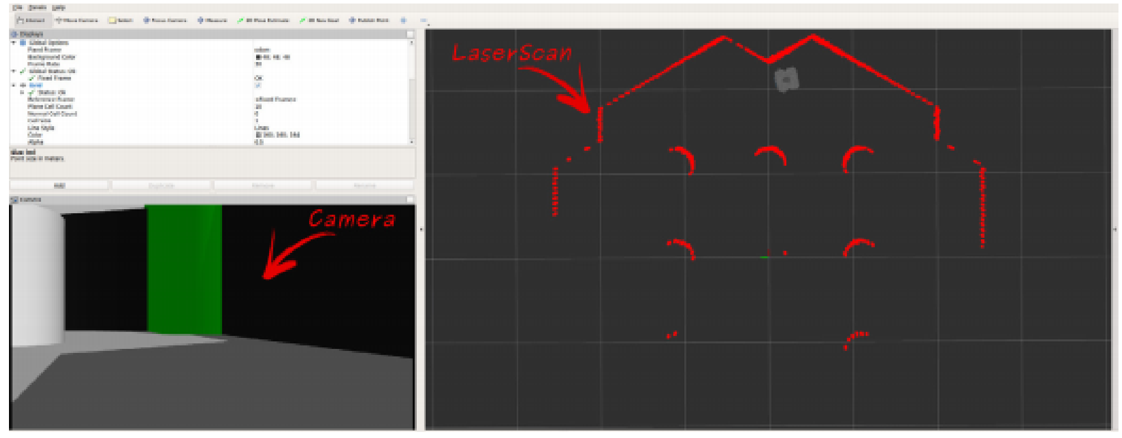


Рис. 151 Візуалізація віртуальних даних LiDAR та зображення камери в Rviz

У цьому розділі пояснюється використання команди для віртуального шолома і навігації. У наступному розділі ми розглянемо SLAM для створення карти з використанням фактичного TurtleBot3 і навігацію для переміщення до певного пункту призначення на даній карті. Після ознайомлення з SLAM і навігацією по альтанці спробуйте виконати фактичну операцію в наступному розділі.

Процедура виконання Virtual SLAM

Коли ви запускаєте залежні пакети і переміщаєте робота в віртуальний простір і створюєте карту, як показано нижче, ви можете створити карту, як показано на рис. 152, і остаточна карта буде виглядати так, як показано на рис. 153.

Launch Gazebo

```
$ export TURTLEBOT3_MODEL=waffle
$ roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

Launch SLAM

```
$ export TURTLEBOT3_MODEL=waffle
$ roslaunch turtlebot3_slam turtlebot3_slam.launch
```

Execute RViz

```
$ export TURTLEBOT3_MODEL=waffle
$ rosrun rviz rviz -d `rospack find turtlebot3_slam`/rviz/turtlebot3_slam.rviz
```

Remotely Control TurtleBot3

```
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

Save the Map

```
$ rosrun map_server map_saver -f ~/map
```

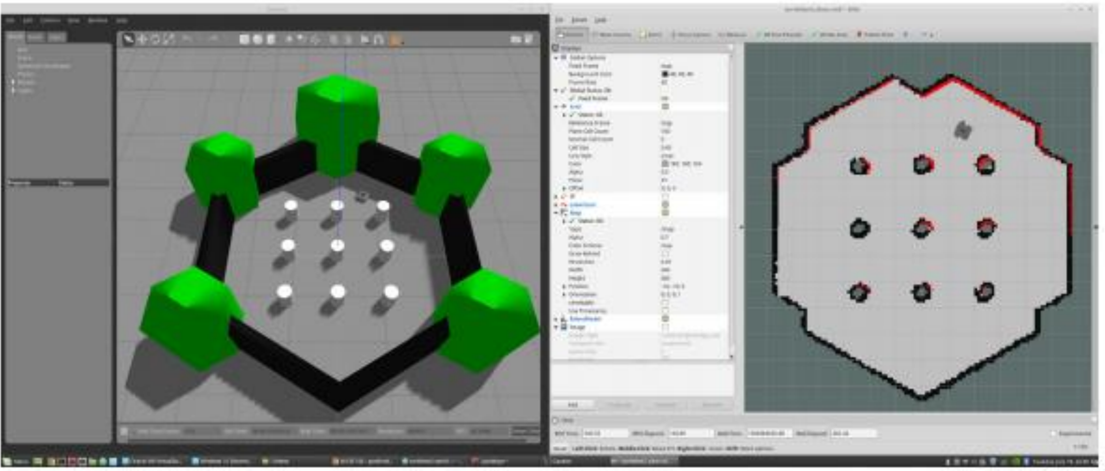


Рис. 152 Запуск SLAM на Gazebo (зліва: Gazebo, справа: Rviz)

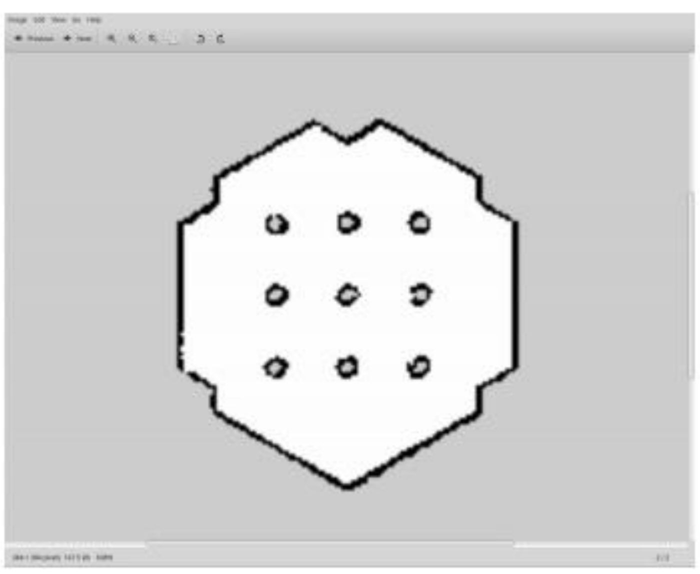


Рис. 153 Сформована карта середовища Gazebo

Процедура Виконання Віртуальної Навігації

Завершіть всі програми, які були виконані під час практики віртуального шолома, і виконайте пов'язані пакети в наступній інструкції, робот з'явиться на раніше згенерованої карті. Після установки початкового положення робота на карті встановіть пункт призначення для запуску навігації, як показано на рис. 154. Початкове положення потрібно встановити тільки один раз.

Execute Gazebo

```
$ export TURTLEBOT3_MODEL=waffle
$ roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

Execute Navigation

```
$ export TURTLEBOT3_MODEL=waffle
$ roslaunch turtlebot3_navigation turtlebot3_navigation.launch map_file:=$HOME/map.yaml
```

Execute RViz and Set Destination

```
$ export TURTLEBOT3_MODEL=waffle
$ rosrun rviz rviz -d `rospack find turtlebot3_navigation`/rviz/turtlebot3_nav.rviz
```

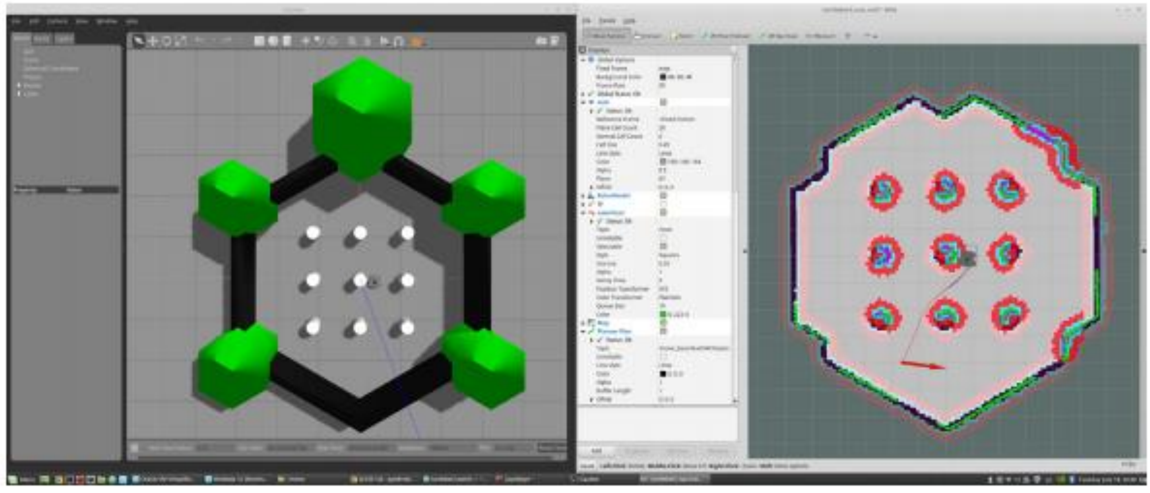


Рис. 154 Запуск навігації на Gazebo (зліва: Gazebo, справа: Rviz)

Були введені два методи моделювання пакета TurtleBot 3. Один з них-використовувати RViz, інструмент 3D-візуалізації ROS, а інший-використовувати Gazebo, 3D-симулятор робота. Моделювання-це чудовий інструмент для користувачів, оскільки він дозволяє

виконувати завдання програмування дуже близько до реального середовища з роботом.

Моделювання TurtleBot

TurtleBot підтримує три типи моделювання: *stage, stdr і Gazebo*. Зверніться до вікі нижче, щоб виконати різні симуляції з віртуальним роботом.

http://wiki.ros.org/TurtleBot_stdr

http://wiki.ros.org/TurtleBot_gazebo

http://wiki.ros.org/TurtleBot_stage

Розділ 11. SLAM та навігація

11.1. Навігація та компоненти

Можливо, було б простіше зрозуміти навігацію як GPS-навігацію в повсякденному житті. Якщо ви задаєте пункт призначення на навігаційному пристрої, навігація дозволяє перевірити відстань і час у дорозі від поточного місця розташування до пункту призначення, а також задати конкретні переваги та інформацію, такі як місця для зупинки і переважні дороги по шляху.

Навігаційна система має відносно коротку історію. У 1981 році японський автовиробник Honda вперше запропонував аналогову систему, засновану на трьохосьовому гіроскопі і плівковій карті під назвою "Електрогрирокатор". Згодом Etak Navigator, електронна навігаційна система, що працює з електронним компасом і датчиками, прикріпленими до коліс, була представлена американською автомобільною компанією Etak. Тим не менш, установка датчика і електронного компаса на автомобіль була важким тягарем для автомобільних цін стали і проблеми надійності навігаційної системи. З 1970-х років Сполучені Штати розробляють супутникові системи позиціонування для військових цілей, а в 2000-х роках стали доступні 24 супутника GPS (Global Positioning System) загального призначення, і почали поширюватися навігаційні системи на основі тріангуляції, що використовують ці супутники.

Повернемося тепер до роботів. Основою і родзинкою мобільного робота, без сумніву, є навігація. Навігація в робототехніці

невіддільна і необхідна. Навігація-це рух робота до певного місця призначення, що не так просто, як здається. Але важливо знати, де знаходиться сам робот, і мати карту даного середовища. Важливо також знайти оптимальний маршрут серед різних варіантів маршруту та уникнути таких перешкод, як стіни та меблі. Це нелегка місія.

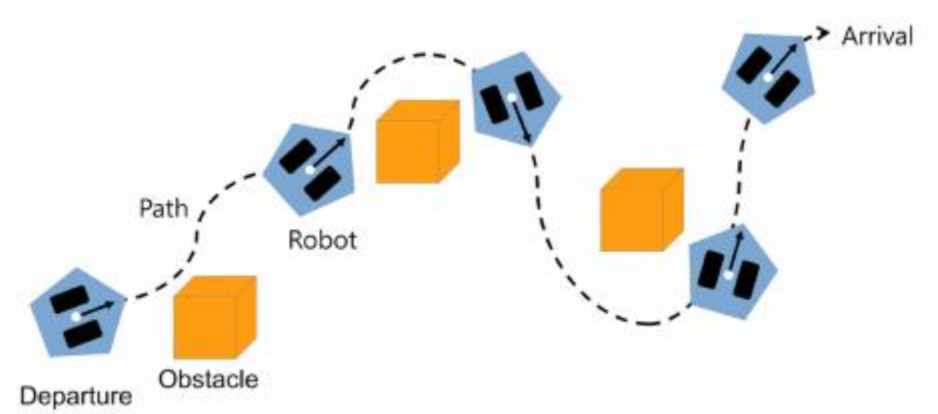


Рис. 155 Навігація

Що потрібно для реалізації навігації в роботах? Це може варіюватися в залежності від навігаційного алгоритму,і в якості *основних функцій* можуть знадобитися наступні:

- ✓ *Карта*
- ✓ *Поза робота*
- ✓ *Сприйняття*
- ✓ *Розрахунок шляху і водіння*

Перша істотна особливість навігації-це карта. Навігаційна система оснащена дуже точною картою з моменту покупки, і модифіковану карту можна періодично завантажувати, щоб її можна

було направляти до місця призначення на основі карти. Але чи буде доступна карта кімнати, де буде розміщений сервісний робот? Як і навігаційна система, роботові потрібна карта, тому нам потрібно створити карту і дати її роботу, або робот повинен бути в змозі створити карту сам.

SLAM (одночасна локалізація і картографування) розроблений для того, щоб робот міг створювати карту за допомогою людини або без нього. Це метод створення карти, в той час як робот досліджує невідомий простір, виявляє його оточення і оцінює його поточне місце розташування, а також створює карту.

По-друге, робот повинен вміти вимірювати і оцінювати свою позу (положення + орієнтація). У випадку автомобіля GPS використовується для оцінки його пози. Однак GPS не можна використовувати в приміщенні, і навіть якщо він може бути використаний, GPS з великими помилками не може бути використаний для роботів. В даний час використовується високоточна система, така як DGPS, але це також марно в приміщенні, а також занадто дорого для загального призначення. Щоб подолати цю проблему, були введені різні методи, такі як розпізнавання маркерів і оцінка місця розташування в приміщенні. Однак з точки зору вартості та точності він все ще недостатній для загального використання.

В даний час найбільш широко використовуваним методом оцінки пози в приміщенні для сервісних роботів є dead reckoning, який

є відносною оцінкою пози, але він використовується вже давно і складається з недорогих датчиків і може отримати певний рівень точного результату оцінки пози. Величина руху робота вимірюється обертанням колеса.

Однак існує похибка між розрахованим відстанню при обертанні колеса і фактичним відстанню переміщення. Тому інерціальна інформація за допомогою датчика іду можна зменшити похибку шляхом компенсації похибки положення і орієнтації між розрахунковим значенням і фактичним значенням.

Поза (Позиція + Орієнтація)

ROS визначає позу як комбінацію положення робота (x, y, z) і орієнтації (x, y, z, w). Як описано в розділі 4.5 TF, орієнтація описується x, y, z і w у формі кватерніона, тоді як положення описується трьома векторами, такими як x, y і z. для отримання більш докладної інформації про повідомлення "поза" зверніться за наступною адресою. Існують і інші технічні терміни, які неявно включають інформацію про напрямок, тому не плутайте їх з цими термінами.

http://docs.ros.org/api/geometry_msgs/html/msg/Pose.html

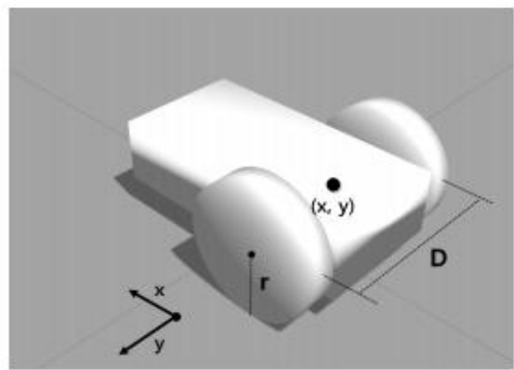


Рис. 156 Інформація, необхідна для розрахунків (центр (x,y) відстань між колесами D та радіус колес r)

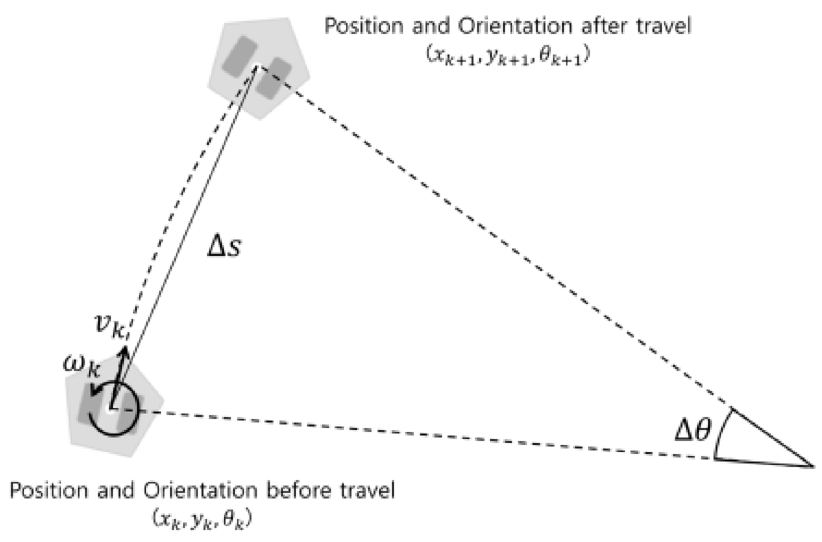


Рис. 157 Dead Reckoning (Мертвий розрахунок)

Ось коротке пояснення про мертвий розрахунок. Коли є мобільний робот, як показано на рис. 156, нехай D-відстань між колесами, а r-радіус колеса. Припускаючи, що робот пройшов дуже коротку відстань за час Te, швидкість обертання (vl, vr) лівого і

правого коліс обчислюється так, як показано в рівняннях (11-1) і (11-2) з величиною обертання лівого і правого двигуна (поточне значення кодера Elc, Erc і попереднє значення кодера Elp, Erp).

$$v_l = \frac{(E_l c - E_l p)}{T_e} \cdot \frac{\pi}{180} \text{ (radian / sec)} \qquad \text{(Equation 11-1)}$$

$$v_r = \frac{(E_r c - E_r p)}{T_e} \cdot \frac{\pi}{180} \text{ (radian / sec)} \qquad \text{(Equation 11-2)}$$

За рівняннями 11-3 і 11-4 обчислюється швидкість лівого і правого колеса (Vl, Vr). З швидкості лівого і правого коліс можна отримати лінійну швидкість (vk) і кутову швидкість (~k) робота, як показано в рівняннях 11-5 і 11-6.

$$V_l = v_l \cdot \mathrm{r} \text{ (meter/sec)} \qquad \text{(Equation 11-3)}$$

$$V_r = v_r \cdot \mathrm{r} \text{ (meter/sec)} \qquad \text{(Equation 11-4)}$$

$$v_k = \frac{(V_r + V_l)}{2} \text{ (meter/sec)} \qquad \text{(Equation 11-5)}$$

$$\omega_k = \frac{(V_r - V_l)}{D} \text{ (radian/sec)} \qquad \text{(Equation 11-6)}$$

Нарешті, використовуючи ці значення, ми можемо отримати положення(x(k+1), y (k+1)) і орієнтацію(і (k+1)) робота з рівнянь 11-7 - 11-10.

$$\Delta s = v_k T_e \qquad \Delta\theta = \omega_k T_e \qquad \text{(Equation 11–7)}$$

$$x_{(k+1)} = x_k + \Delta s \cos\left(\theta_k + \frac{\Delta\theta}{2}\right) \qquad \text{(Equation 11–8)}$$

$$y_{(k+1)} = y_k + \Delta s \sin\left(\theta_k + \frac{\Delta\theta}{2}\right) \qquad \text{(Equation 11–9)}$$

$$\theta_{(k+1)} = \theta_k + \Delta\theta \qquad \text{(Equation 11–10)}$$

По-третє, для з'ясування того, чи є перешкоди, такі як стіни і предмети, потрібні датчики. Використовуються різні типи датчиків, такі як датчики відстані та датчики зору. Датчик відстані використовує лазерні датчики відстані (LDS, LRF, LiDAR), ультразвукові датчики та інфрачервоні датчики відстані. Датчик зору включає в себе стереокамери, моно-камери, всеспрямовані камери, а останнім часом RealSense, Kinect, Xtion, які широко використовуються в якості глибинних камер, використовуються для ідентифікації перешкод.

Остання істотна особливість навігації - це розрахунок і проходження оптимального маршруту до місця призначення. Це називається пошуком і плануванням шляху, і існує безліч алгоритмів, які виконують це завдання, таких як алгоритм a*, потенційне поле, Фільтр частинок і RRT (швидко досліджуване випадкове дерево).

У цьому розділі ми коротко підсумували SLAM і компоненти навігації, але це все ще важко і широко зрозуміти. Вимірювання та оцінка пози робота були пояснені в попередньому розділі. Вимірювання перешкод, таких як стіна і об'єкти, описано в главі 8

роботи, датчики і двигуни. Тепер давайте розглянемо SLAM для створення карти і навігацію за допомогою згенерованої карти.

11.2. Практика SLAM

Перш ніж описувати теорію SLAM, давайте подивимося, як використовувати SLAM з TurtleBot3. Файл bag, який можна використовувати для створення карти, знаходиться в репозиторії GitHub, тому рекомендується спробувати цей приклад. Теорія СЛЕМА буде розглянута в розділі 11.4 після завершення практики СЛЕМА.

Пакети mapping, cartographer і rtabmap широко використовуються для SLAM, і в цьому розділі ми будемо використовувати пакет gmapping. При використанні картографування корисно розуміти деякі апаратні обмеження, хоча вони і не пов'язані з мобільними роботами.

Рух

Платформа повинна бути здатна рухатися за допомогою команд лінійної швидкості по осях X, Y і тета-кутової швидкості. Наприклад, робот складається з двох двигунів, таких як диференційний привід mobilee robot або всесезонний робот, який має більше трьох приводних валів.

Одометрія

Інформація про одометрію повинна бути доступна. Пройдена відстань повинна бути виміряна шляхом обчислення за допомогою мертвого рахунку або компенсуючої пози з інерційними даними або

оцінки швидкості переміщення і кутової швидкості за допомогою датчика іду, щоб робот міг обчислити і оцінити свою поточну позу.

Датчик вимірювання відстані

Для шолома і навігації робот повинен мати такі датчики, як світлодіоди (лазерний далекомір), LRF (лазерний далекомір) або лідар, які можуть вимірювати відстань до перешкоди в площині XY. Глибинні камери, такі як RealSense, Kinect і Xtion, також можуть перетворювати 3D-інформацію в 2D-інформацію площини XY. Іншими словами, необхідно встановити датчик, здатний вимірювати відстань на площині XY. Ультразвукові датчики, датчики PSD і візуальні сигнали тривоги також можуть бути розглянуті, але вони не будуть розглянуті в цій книзі.

Форма робота

Роботи з правильними багатокутними або круглими формами розглядаються в цьому розділі. Трансформовані роботи, які Довгі на одній осі, роботи, занадто великі, щоб пройти між дверима, двоногі гуманоїдні роботи, багатосуглобові мобільні роботи і літаючі роботи не розглядаються для шолома. У цій главі ми будемо використовувати TurtleBot3, офіційну платформу ROS, яку ми обговорювали в главі 10. TurtleBot 3 на рис. 157 задовольняє всі чотири обмеження SLAM, згадані вище.

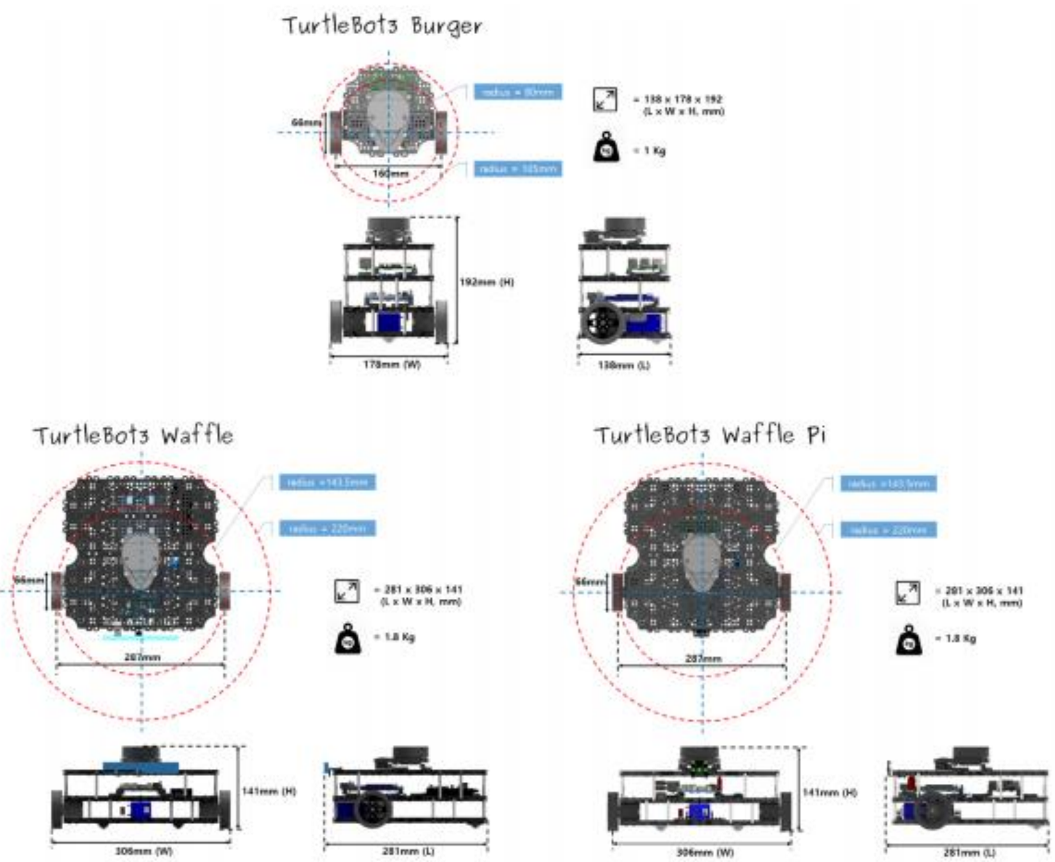


Рис. 158 Форма та розмір TurtleBot3 Burger, Waffle and Waffle Pie

Хоча середовище з підтримкою SLAM не вказано, існують певні обмеження, пов'язані з характеристиками алгоритму Gmapping:

❶ квадратна кімната без перешкод;

❷ довгий коридор без будь-яких відмінних об'єктів;

❸ окуляри, які не відображають лазерне або інфрачервоне світло,

❹ дзеркала, які розсіюють світло;

❺ широкі і відкриті середовища, де інформація про перешкоди не може бути отримана, такі як озеро або море.

У наведеному в книзі прикладі навколишнє середовище задається у вигляді лабіринту з сітковою структурою, довжина якої не може бути виміряна, як показано на рис.159.

Рис. 159 Цільове середовище вимірювання

Пакети ROS, пов'язані з SLAM, що використовуються в цьому розділі, - це Метапакет TurtleBot3 і пакет gmapping в метапакеті slam_gmapping і пакет map_server в метапакеті навігації. Ці пакети вже були встановлені в середовищі розробки 10.5 TurtleBot3. Оскільки цей розділ є наступним вправою, буде описана тільки процедура виконання. Опис кожного пакета буде детально описано в наступному розділі. Щоб уникнути плутанини, теги [Remote PC] і [TurtleBot] будуть використовуватися для вказівки того, де повинна бути застосована команда.

Послідовність виконання SLAM виглядає наступним чином. Цей приклад описує команди для TurtleBot 3 Waffle як посилання. Якщо ви використовуєте Burger або Waffle Pi, просто змініть " TURTLEBOT3_MODEL "в команді з" waffle"на 'burger' або 'waffle_pi".

Roscore

Запустіть 'roscore' на [віддаленому ПК]

```
$ roscore
```

Запуск Робота

У [TurtleBot] запустіть ' turtlebot3_robot.запустіть файл для виконання вузлів turtlebot3_core і turtlebot3_lds.

```
$ roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

Пакет run SLAM

Виконайте запуск файлу turtlebot3_slam.launch в [віддаленому ПК]. Пакет turtlebot3_slam складається всього з одного файлу запуску. Файл запуску виконує вузол robot_state_publisher, який публікує тривимірну інформацію про положення і орієнтацію обох коліс і кожного з'єднання в TF, а також вузол slam_gmapping для побудови карти. Крім того, "robot_model" налаштований на завантаження URDF (Unified Robot Description Format), який описує зовнішній вигляд робота.

```
$ export TURTLEBOT3_MODEL=waffle
$ roslaunch turtlebot3_slam turtlebot3_slam.launch
```

Запустіть RViz

Давайте запустимо інструмент візуалізації RViz, щоб ви могли візуально перевірити результати під час SLAM. При запуску RViz з наступними опціями плагіни відображення додаються з самого початку.

```
$ export TURTLEBOT3_MODEL=waffle
$ rosrun rviz rviz -d `rospack find turtlebot3_slam`/rviz/turtlebot3_slam.rviz
```

Зберегти Повідомлення Теми

Користувач може безпосередньо керувати роботом і виконувати операцію SLAM. Теми' /scan 'і' / tf', опубліковані під час операції, можуть бути збережені у вигляді файлу ' scan_data.bag', як показано в наведеній нижче команді. Ви можете прочитати цей файл, щоб створити карту пізніше, або ви можете відтворити теми "/scan " і "/tf " під час експерименту, коли ви працюєте над картою, не повторюючи експеримент. Думайте про це як про зберігання даних теми з експерименту (теми "/scan' і '/tf'). Опція' - O ' наступної команди дозволяє вказати ім'я вихідного файлу і зберегти файл bag як 'scan_data.bag’. Збереження повідомлень теми не є обов'язковим в іcламі, тому Ви можете пропустити його, якщо вам не потрібно зберігати повідомлення.

```
$ rosbag record -O scan_data /scan /tf
```

Управління роботом

Наступна команда дозволяє користувачеві керувати роботом для виконання операції SLAM вручну. Важливо уникати енергійних рухів, таких як занадто швидка зміна швидкості або занадто швидке обертання. При побудові карти за допомогою робота робот повинен сканувати кожен куточок навколишнього середовища для вимірювання. Це вимагає деякого досвіду, щоб побудувати чисту карту, тому давайте практикувати SLAM кілька разів, щоб створити ноу-хау.

```
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

Створити карту

Тепер, коли ви зробили всю роботу, давайте запустимо "map_saver_node', щоб створити карту. Карта малюється на основі одометрії робота, інформації tf та інформації сканування датчика при русі робота. Ці дані можна побачити в rviz з попереднього прикладу. Створена карта зберігається в каталозі, в якому виконується "map_saver'. Якщо ви не вкажете ім'я файлу, він буде збережений як файл map.pgm і map.yaml, що містить інформацію про карту.

Опція '- f ' відноситься до папки та імені файлу, в якому зберігається файл карти. Якщо в якості опції використовується '~ / map', то ' map.pgm ' і ' map.yaml ' будуть збережені в папці map домашньої папки користувача (~ /).

```
$ rosrun map_server map_saver -f ~/map
```

Використовуйте описаний вище процес для створення вашої карти. Вузли та теми, необхідні для відображення, можуть бути отримані за допомогою "rqt_graph", як показано на рис.160. Процес картографування показаний на рис. 162, а завершена карта - на рис. 161. Ми можемо підтвердити, що вищезгадане експериментальне середовище правильно намальоване на карті.

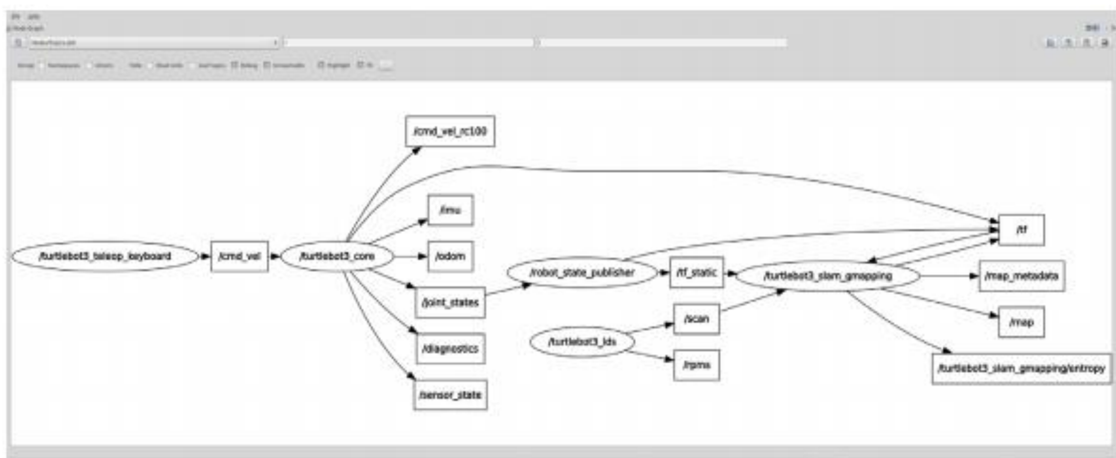


Рис. 160 Вузли та теми, необхідні для SLAM

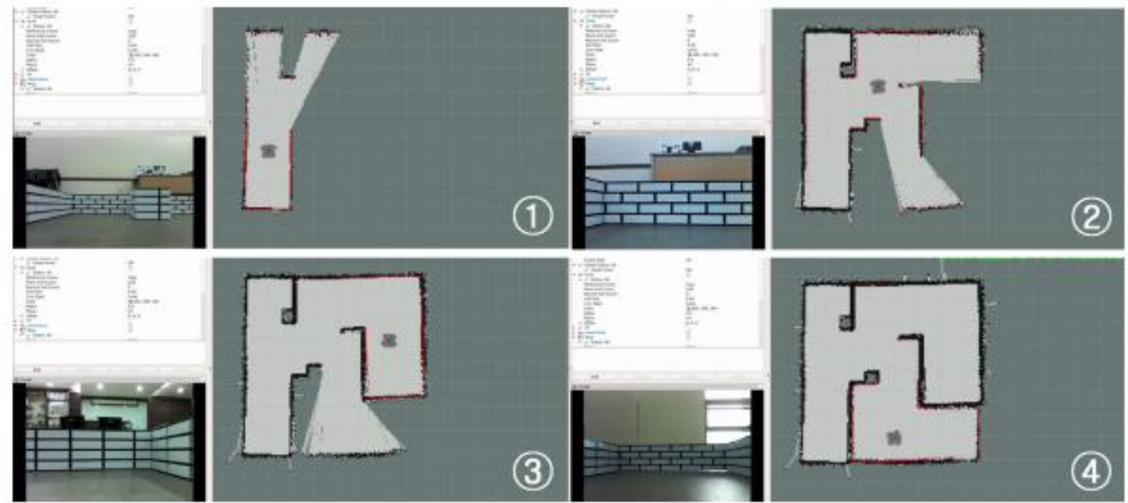


Рис. 161 SLAM працює для відображення (SLAM running for mapping)

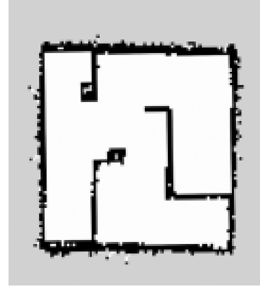


Рис. 162 Завершена карта

Давайте тренуємося в шоломі без TurtleBot3 і датчика LDS. Для цього потрібно записаний bigfile, який можна завантажити за допомогою наступної команди.

```
$ wget https://raw.githubusercontent.com/ROBOTIS-GIT/turtlebot3/master/turtlebot3_slam/bag/
TB3_WAFFLE_SLAM.bag
```

Інша частина процедури аналогічна наведеним вище інструкціям SLAM. Однак параметр команди "rosbag" необхідно змінити з 'save' на 'play'. Тоді він буде вести себе в тій же манері, що і реальний експеримент.

```
$ roscore
```

```
$ export TURTLEBOT3_MODEL=waffle
```

```
$ roslaunch turtlebot3_bringup turtlebot3_remote.launch
```

```
$ export TURTLEBOT3_MODEL=waffle
$ rosrun rviz rviz -d `rospack find turtlebot3_slam`/rviz/turtlebot3_slam.rviz
```

```
$ roscd turtlebot3_slam/bag
$ rosbag play ./TB3_WAFFLE_SLAM.bag
```

```
$ rosrun map_server map_saver -f ~/map
```

У наступному розділі описуються вихідні коди пакетів і способи налаштування пакетів, які ви запустили в попередньому прикладі.

11.3. Додаток SLAM

У цьому розділі ми розглянемо пакети ROS, що використовуються в SLAM, і дізнаємося, як їх створювати і налаштовувати. Будуть розглянуті Метапакет TurtleBot3, пакет gmapping в метапакеті slam_gmapping і пакет map_server в метапакеті навігації. У цьому розділі наведено технічні деталі SLAM, щоб його

можна було застосувати до робота. Теорія шолома розглядається в розділі 11.4.

Цей розділ заснований на платформі робота TurtleBot3 і її датчику, але він може бути реалізований на індивідуальному роботі без обмеження конкретною платформою робота або датчиком. Якщо ви хочете створити власну робототехнічну платформу або створити власну робототехнічну платформу Turtle bot 3, цей розділ буде корисним.

Перш за все, про карти потрібно розповісти набагато більше, оскільки карта-це результат, який ми шукаємо в цьому розділі. Якщо ми дамо паперову карту роботу, як ви думаєте, чи зможе робот її зрозуміти? Швидше за все, ні. Роботу потрібно оцифровка, щоб зрозуміти і обчислити інформацію. Визначення карти для навігації роботів обговорювалося вже давно і обговорюється досі. Зокрема, сучасні карти включають в себе не тільки двовимірну, але і тривимірну інформацію, а іноді навіть сегменти об'єкта, не пов'язаного з навігаційною інформацією.

У цьому розділі ми будемо використовувати двовимірну сіткову карту зайнятості (OGM), яка зазвичай використовується в спільноті ROS. Карта, отримана з попереднього розділу, Як показано на рис. 163, Біла-це вільна область, в якій робот може рухатися, чорна-зайнята область, в якій робот не може рухатися, а сіра-невідома область.

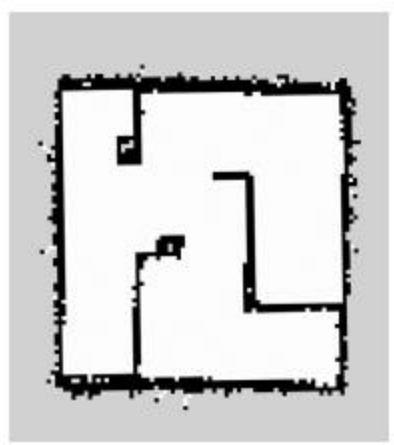


Рис. 163 Карта сітки зайнятості (Occupancy Grid Map)

Область на карті представлена значеннями відтінків сірого в діапазоні від " 0 " до "255". Це значення отримується за допомогою апостеріорної ймовірності теореми Байєса, яка обчислює ймовірність зайнятості, що представляє стан зайнятості. Імовірність зайнятості "occ' виражається як "occ = (255-color_avg) / 255.0'. Якщо зображення 24-бітне, то " color_avg = (значення відтінків сірого в одній комірці / 0xFFFFFF x 255)'. Чим ближче цей "occ' до 1, тим вище ймовірність того, що він зайнятий, і чим ближче до "0", тим менше ймовірність того, що він буде зайнятий.

Коли ймовірність зайнятості публікується у вигляді повідомлення ROS (nav_msgs/Accountability Grid), вона перевизначається до цілого числа [0 ~ 100]. Область, близька до" 0', є вільною областю, визначеною як незайнята область, тоді як "100"

визначається як зайнята область, а "-1' особливо визначається для невідомої області.

У ROS фактична карта зберігається у форматі файлу '*.pgm' (portable graymap format), а файл '*.yaml ' містить інформацію про карту. Наприклад, якщо ми перевіримо інформацію про карту (map.yaml), збережену в розділі 11.2, параметр image визначає ім'я файлу карти, а дозвіл-дозвіл карти в метрах / пікселях.

```
image: map.pgm
resolution: 0.050000
origin: [-10.000000, -10.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

Тобто кожен піксель може бути перетворений в 5 см. початок координат-це початок координат карти, і кожне значення являє собою X, y і YAW відповідно. Нижній лівий кут карти являє x = - 10m, y = - 10m. заперечення інвертує чорно-білий колір. Колір кожного пікселя визначається ймовірністю зайнятості. Якщо ймовірність зайнятості перевищує поріг зайнятості (occupied_thresh), Піксель виражається як зайнята область чорним кольором. В іншому випадку піксель буде виражений як вільна область білого кольору.

На рис. 164 показано результат створення великої карти за допомогою TurtleBot 3. Знадобилося близько години, щоб створити карту з відстанню переміщення близько 350 метрів.

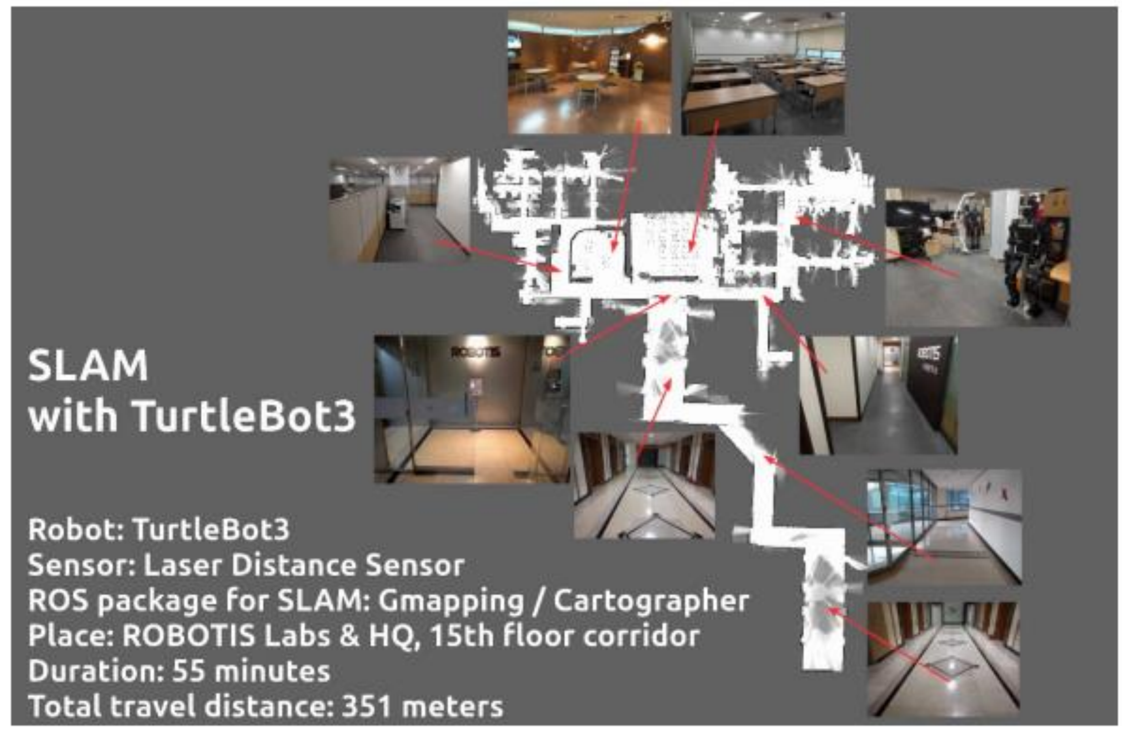


Рис. 164 Карта великої площі зайнятості, створена TurtleBot3 (Large area occupancy grid map created by TurtleBot3)

Тепер, коли ви знаєте про карту, давайте розглянемо матеріали, необхідні для створення карти за допомогою SLAM. Перш за все, нам потрібно визначити, що нам потрібно при створенні карти. Перше, що вам потрібно, - це значення відстані. Це означає, що робот, будучи центром вимірювання, повинен мати можливість отримувати значення відстані від певних об'єктів. Наприклад, така інформація, як "диван знаходиться на відстані 2 м від робота". Прикладом такої інформації є дані про відстань, скановані з площини XY за допомогою таких датчиків, як LEDSS і глибинна камера.

По-друге, це значення пози, яке позначає інформацію про позу датчика, прикріпленого до робота. Таким чином, величина пози датчика залежить від одометрії робота. Необхідно надати інформацію про одометрії для розрахунку значення пози.

На рис. 165 відстань, виміряна за допомогою LDS, називається "скануванням" в ROS, а інформація про позу (положення + орієнтація) залежить від відносної координати, тому вона називається "tf' (перетворення). Як показано на рис. 165, ми запускаємо SLAM на основі двох фрагментів інформації, 'scan' і 'tf', і створюємо потрібну карту.

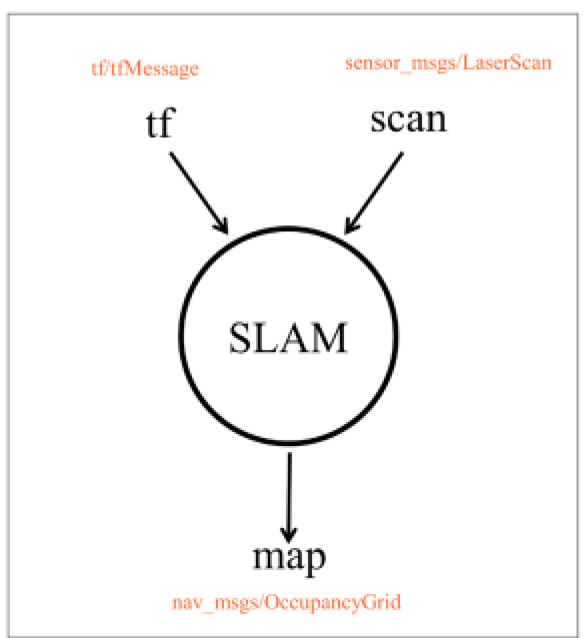


Рис. 165 Сканування та tf дані, необхідні в SLAM, та відношення до карти (Scan and tf data required in SLAM and the relation to the map)

Щоб створити карту за допомогою SLAM, на додаток до вузла turtlebot3_core був створений пакет turtlebot3_slam. Цей пакет не має вихідного файлу, але пакети, необхідні для SLAM, запускаються при виконанні файлу запуску. Потік процедури SLAM проілюстрований на рис.166, і кожен процес описаний нижче.

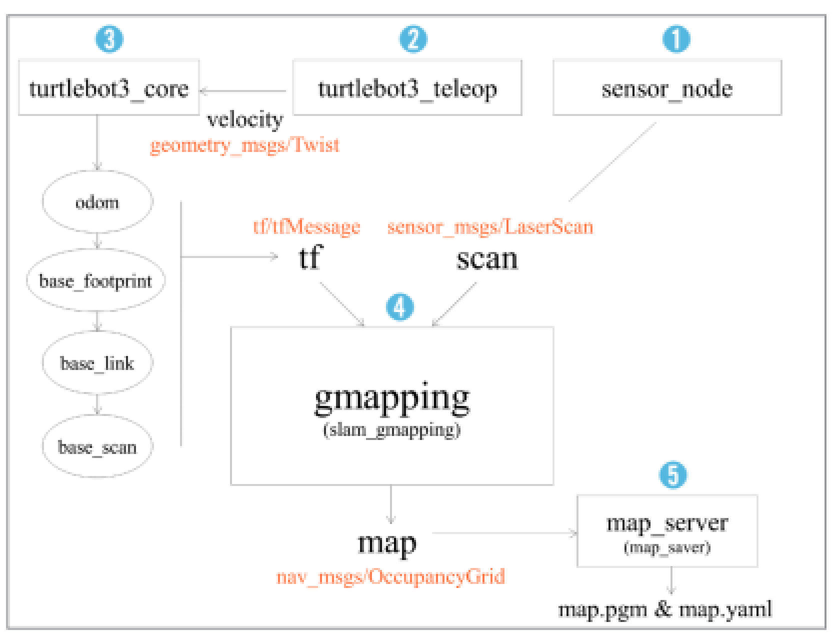


Рис. 166 Блок-схема turtlebot3_slam

Таблиця 23

sensor_node(приклад: turtlebot3_lds)	Вузол " turtlebot3_lds "запускає датчик LDS і відправляє інформацію" сканування", необхідну для SLAM, у вузол "slam_gmapping'.

turtlebot3_teleop(приклад: turtlebot3_teleop_keyboard)	Вузол turtlebot3_teleop_keyboard-це вузол, який може приймати введення з клавіатури і керувати роботом. Цей вузол надсилає команду translation and rotation speed вузлу turtlebot3_core.
turtlebot3_core	Вузол "turtlebot3_core" отримує команду перекладу і швидкості обертання і переміщує робота. У той час як вузол публікує "odom", який є вимірюваною і оціненою позою робота, він також публікує перетворену відносну координату "odom" у формі " tf " в порядку odom →base_footprint → base_link → base_scan.
turtlebot3_slam_gmapping	Вузол turtlebot3_slam_gmapping створює карту на основі інформації сканування від датчика вимірювання відстані та інформації tf, яка є значенням пози датчика.
map_saver	Вузол "map_saver' в пакеті "map_server' створює файл " map.pgm'

	і файл " map.yaml", який є інформаційним файлом для карти.

Дві частини інформації, що використовуються в SLAM, - це значення відстані і поза, в якій вимірюється значення відстані. Значення відстані може бути отримано з вузла датчика, а значення пози датчика може бути отримано шляхом обчислення положення датчика. Датчик встановлюється у фіксованому місці робота, і робот рухається відповідно до команди дистанційного керування. Тобто робот і датчик фізично фіксовані, а поза (положення + орієнтація) датчика змінюється відповідно до позою робота. Простіше думати про це як про відносне перетворення координат, яке в ROS називається"tf". Наступна команда візуалізує відносні координати в деревоподібній структурі.

```
$ rosrun rqt_tf_tree rqt_tf_tree
```

Якщо ви виконаєте наведену вище команду, ви можете перевірити відносне перетворення координат (TF) робота і датчика за допомогою "TF tree viewer", як показано на рис. 167. Іншими словами, якщо ми зосередимося на позі LDS в позі робота, інформація про позу буде відносно пов'язана в порядку odom → base_footprint → base_link → base_scan. Керована поступальна швидкість і швидкість обертання від вузла turtlebot3_teleop_keyboard керують роботом, в той час як поза робота вимірюється відповідно до мертвим рахунком. Під час цієї процедури "одом" публікується як "тф'. Фізично фіксований

base_footprint → base_link → base_scan, який описує кожне перетворення координат, як описано у файлі ' / urdf / turtlebot3_waffle.urdf.xacro 'пакета ' turtlebot3_description', і періодично публікує 'tf' через вузол 'robot_state_publisher'.

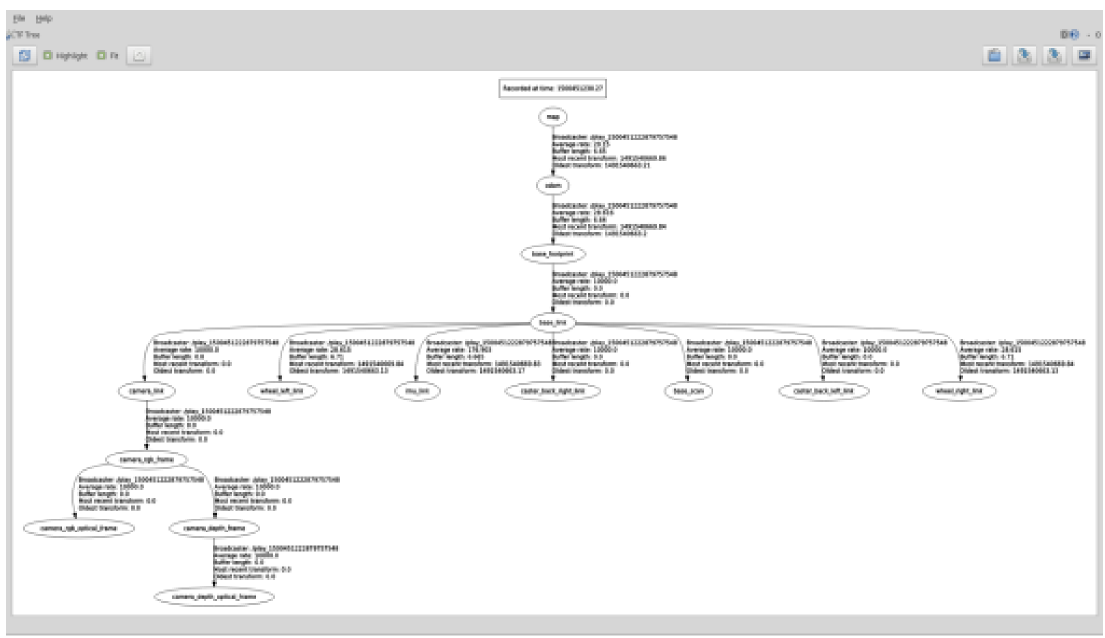


Рис. 167 Відносний статус перетворення координат деталей карти та роботів (Relative Coordinate Transformation Status of Map and Robots Parts)

Вміст файлу turtlebot3_slam.запуск у пакеті turtlebot3_slam виглядає наступним чином. Стартовий файл розділений на два основних розділи, де перша частина містить ' turtlebot3_remote.запустіть файл', і друга частина виконає вузол 'turtlebot3_slam_gmapping'.

turtlebot3_slam/launch/turtlebot3_slam.launch

```
<launch>
  <!-- Turtlebot3 -->
  <include file="$(find turtlebot3_bringup)/launch/turtlebot3_remote.launch" />

  <!-- Gmapping -->
  <node pkg="gmapping" type="slam_gmapping" name="turtlebot3_slam_gmapping" output="screen">
```

```
    <param name="base_frame" value="base_footprint"/>
    <param name="odom_frame" value="odom"/>
    <param name="map_update_interval" value="2.0"/>
    <param name="maxUrange" value="4.0"/>
    <param name="minimumScore" value="100"/>
    <param name="linearUpdate" value="0.2"/>
    <param name="angularUpdate" value="0.2"/>
    <param name="temporalUpdate" value="0.5"/>
    <param name="delta" value="0.05"/>
    <param name="lskip" value="0"/>
    <param name="particles" value="120"/>
    <param name="sigma" value="0.05"/>
    <param name="kernelSize" value="1"/>
    <param name="lstep" value="0.05"/>
    <param name="astep" value="0.05"/>
    <param name="iterations" value="5"/>
    <param name="lsigma" value="0.075"/>
    <param name="ogain" value="3.0"/>
    <param name="srr" value="0.01"/>
    <param name="srt" value="0.02"/>
    <param name="str" value="0.01"/>
    <param name="stt" value="0.02"/>
    <param name="resampleThreshold" value="0.5"/>
    <param name="xmin" value="-10.0"/>
    <param name="ymin" value="-10.0"/>
    <param name="xmax" value="10.0"/>
    <param name="ymax" value="10.0"/>
    <param name="llsamplerange" value="0.01"/>
    <param name="llsamplestep" value="0.01"/>
    <param name="lasamplerange" value="0.005"/>
    <param name="lasamplestep" value="0.005"/>

    </node>
</launch>
```

По-перше, давайте подивимося на 'turtlebot3_remote.запустіть ' файл. Цей файл містить задану користувачем модель робота для завантаження і виконує вузол robot_state_publisher, який публікує інформацію про стан робота обох коліс і кожного з'єднання в TF.

```
turtlebot3_bringup/launch/turtlebot3_remote.launch
<launch>
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger, waffle, waffle_pi]"/>

  <include file="$(find turtlebot3_bringup)/launch/includes/description.launch.xml">
    <arg name="model" value="$(arg model)" />
  </include>

  <node pkg="robot_state_publisher" type="robot_state_publisher" name="robot_state_publisher"
output="screen">
    <param name="publish_frequency" type="double" value="50.0" />
  </node>
</launch>
```

Вузол "turtlebot3_slam_gmapping" дозволяє перейменувати вузол "slam_gmapping" в пакеті gmapping при запуску вузла. Щоб цей вузол працював правильно, вам потрібно змінити різні параметри вашого робота і датчика. Наступні налаштування призначені для вафлі TurtleBot 3. Якщо ви хочете використовувати іншого робота, крім TurtleBot3, будь ласка, зверніться до наступного пояснення і змініть налаштування відповідно до вашим роботом і датчиком.

slam_gmapping Node Setting

```
<param name="base_frame" value="base_footprint"/>   The frame attached to the mobile base
<param name="odom_frame" value="odom"/>             The frame attached to the odometry system
<param name="map_update_interval" value="2.0"/>     Map update interval (sec)
<param name="maxUrange" value="4.0"/>      Max Range of laser sensor to use (meter)
<param name="minimumScore" value="100"/>   Min Score considering the results of scan matching
<param name="linearUpdate" value="0.2"/>   Min travel distance required for processing
<param name="angularUpdate" value="0.2"/>  Min rotation angle required for processing
<param name="temporalUpdate" value="0.5"/> If the last scan time exceeds this update time, the
scan is performed. Negative values will be ignored and not used.
<param name="delta" value="0.05"/>         Map Resolution: Distance / Pixel
<param name="lskip" value="0"/>            Number of beams to skip in each scan
<param name="particles" value="120"/>      Number of particles in particle filter
<param name="sigma" value="0.05"/>         Standard deviation of laser-assisted search
<param name="kernelSize" value="1"/>       Window size of laser-assisted search
<param name="lstep" value="0.05"/>         Initial search step (translation)
<param name="astep" value="0.05"/>         Initial search step (rotation)
<param name="iterations" value="5"/>       Number of scan-matching iterations
<param name="lsigma" value="0.075"/>       The sigma of a beam used for likelihood computation
```

```
<param name="ogain" value="3.0"/>          Gain to be used while evaluating the likelihood
<param name="srr" value="0.01"/>           Odometry error (translation→ translation)
<param name="srt" value="0.02"/>           Odometry error (translation→ rotaion)
<param name="str" value="0.01"/>           Odometry error (rotation → translation)
<param name="stt" value="0.02"/>           Odometry error (rotation → rotation)
<param name="resampleThreshold" value="0.5"/>Resampling threshold value
<param name="xmin" value="-10.0"/>         Initial map size (min x)
<param name="ymin" value="-10.0"/>         Initial map size (min y)
<param name="xmax" value="10.0"/>          Initial map size (max x)
<param name="ymax" value="10.0"/>          Initial map size (max y)
<param name="llsamplerange" value="0.01"/> Translational sampling range for the likelihood
<param name="llsamplestep" value="0.01"/>  Translational sampling step for the likelihood
<param name="lasamplerange" value="0.005"/> Angular sampling range for the likelihood
<param name="lasamplestep" value="0.005"/> Angular sampling step for the likelihood
```

Весь зміст, необхідний для картографування, було пояснено. Наступний розділ присвячений теорії SLAM.

11.4. Теорія SLAM

SLAM (одночасна локалізація і картографування) означає дослідження і картографування невідомого середовища при оцінці пози самого робота за допомогою встановлених на ньому датчиків. Це ключова технологія навігації, така як автономне водіння.

Для оцінки пози зазвичай використовуються енкодери і інерціальні вимірювальні блоки (іду). Енкодер обчислює приблизну позу робота за допомогою мертвого рахунку, який вимірює величину обертання ведучого колеса. Цей процес супроводжується досить великою похибкою оцінки, і інерціальна інформація, виміряна інерціальним датчиком, компенсує похибка обчисленої пози. Залежно від мети позу можна оцінити і без енкодера, але тільки за допомогою інерціального датчика.

Ця оціночна поза може бути ще раз скоригована за допомогою навколишнього екологічної інформації, отриманої за допомогою датчика відстані або камери, використовуваної при створенні карти. Ця методологія оцінки пози включає фільтр Калмана, локалізацію Маркова, локалізацію Монте-Карло з використанням фільтра частинок і так далі.

Для картографування часто використовуються датчики відстані, такі як ультразвукові датчики, світлові Детектори, радіодетектори, лазерні далекоміри та інфрачервоні сканери. На

додаток до датчика відстані, камери також використовуються для вимірювання відстані, такі як стереокамера. Крім того, є візуальна SLAM за допомогою загальної камери.

Крім того, було запропоновано метод розпізнавання навколишнього середовища шляхом прикріплення маркерів. Наприклад, цей метод має маркери на стелі, щоб камера могла їх розрізнити. Останнім часом камери глибини (RealSense, Kinect, Xtion і т.д.) широко використовуються для отримання інформації про відстань, яка так само точна, як і датчики відстані.

Оцінка пози є дуже важливою областю досліджень в робототехніці і активно вивчається до цих пір. Якщо ціна робота може бути правильно оцінена, такі завдання, як SLAM, який являє собою побудову карти на основі пози, можуть бути легко виконані. Однак існує багато проблем, таких як невизначеність інформації спостереження датчика, і властивість реального часу має бути забезпечено для того, щоб працювати в реальному середовищі. Для вирішення цього завдання були вивчені різні методи оцінки місця розташування. У цьому розділі методологія фільтра Калмана та фільтра частинок обговорюється як загальні приклади оцінки пози.

Фільтр Калмана

Фільтр Калмана, який використовувався в проекті НАСА "Аполлон", був розроблений доктором Рудольфом Е.Калманом, який з тих пір прославився цим алгоритмом. Його фільтр був рекурсивним

фільтром, який відстежує стан об'єкта в лінійній системі з шумом. Фільтр заснований на байєсівській ймовірності, яка передбачає модель і використовує цю модель для передбачення поточного стану з попереднього. Потім помилка між прогнозованим значенням попереднього кроку та фактичним вимірюваним поточним значенням, отриманим вимірювальним приладом, використовується для виконання кроку оновлення оцінки більш точного значення стану. Фільтр повторює описаний вище процес і підвищує точність. Цей процес спрощений, як показано на рис. 168.

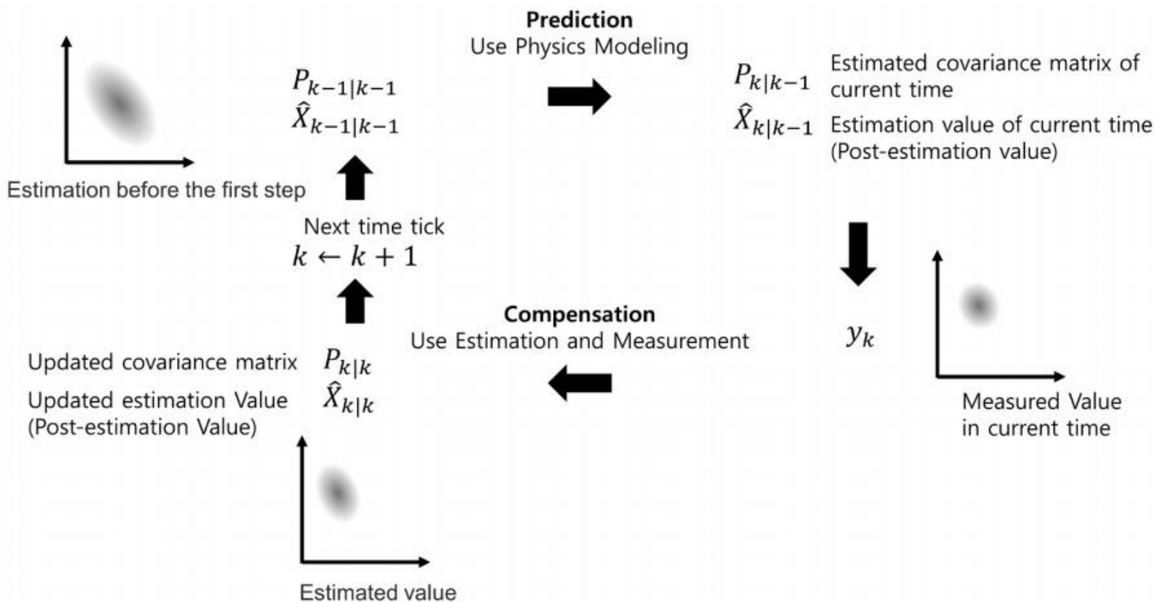


Рис. 168 Основна концепція фільтру Калмана

Однак фільтр Калмана застосовується лише до лінійних систем. Більшість наших роботів і датчиків є нелінійними системами

та EKF (розширений фільтр Калмана), модифікований фільтром Калмана широко використовуваний. Крім того, існує безліч варіантів KF, таких як UKF (фільтр Калмана без запаху), що покращило точність EKF, та фільтр Fast Kalman, який покращив швидкість, та вони досліджуються і сьогодні. Фільтр Калмана також часто використовується з іншими алгоритмами, такими як Rao-Blackwelled Particle Filter (RBPF), який використовується з фільтрами для частинок.

Фільтр частинок

Фільтр частинок - найпопулярніший алгоритм відстеження об'єктів. Типовими прикладами є локалізація Монте Карло з використанням фільтрів часток. Описаний раніше фільтр Калмана гарантує точність лише для лінійної системи та системи, до якої застосовується гауссовий шум. Більшість проблем в реальному світі є нелінійними системами.

Оскільки роботи та датчики також нелінійні, фільтри часток часто використовуються для оцінки пози. Якщо фільтр Калмана є аналітичним методом, який приймає систему як лінійну і здійснює пошук параметрів лінійним рухом, фільтр частинок - це техніка для прогнозування моделювання на основі методу спроб і помилок. Фільтр для частинок отримав свою назву, оскільки розрахункове значення, породжене розподілом ймовірностей у системі,

представляється як частинки. Це також називається послідовним методом Монте-Карло (SMC) або методом Монте-Карло.

Фільтр частинок, як і інші алгоритми оцінки пози, оцінює позу об'єкта припускаючи, що помилка включена у вхідну інформацію. При використанні SLAM, робота значення одометрії та значення вимірювання за допомогою датчика відстані використовуються для оцінки поточної пози робота.

У методі фільтрування частинок невизначена поза описується групою частинок, що називаються зразками. Ми переміщуємо частинки в нове передбачуване положення та орієнтацію на основі роботи моделі руху робота та ймовірності та вимірюємо вагу кожної частинки відповідно до фактичного значення вимірювання та поступово зменшуємо шум, щоб оцінити точну позу.

У випадку з мобільним роботом, кожна частинка представлена у вигляді 'частинка = поза (x, y, i)', вага і кожна частинка являє собою довільну дрібну частинку, що представляє передбачуване положення та орієнтацію робота виражається x, y та i робота та вагою кожної частинки.

Цей фільтр частинок проходить наступні 5 процедур. За винятком ініціалізації в кроці 1, кроки 2–5 неодноразово виконуються для оцінки пози робота. Іншими словами, це метод оцінки пози робота шляхом оновлення розташування частинок, якими показано

ймовірність розташування робота на координатній площині X, Y на основі виміряного значення датчика.

Таблиця 24

❶ Ініціалізація	Оскільки початкова поза (положення, орієнтація) робота невідома, частинки випадково розташовані в межах діапазону, де позу можна отримати з N частинками. Кожна з початкових частинок важить 1 / N, а сума ваги частинок дорівнює 1. N емпірично визначається, зазвичай сотнями. Якщо початкове положення відоме, частинки розміщуються поблизу робота.
❷ Прогнозування	На основі системної моделі, що описує рух робота, він рухає кожну частинку як кількість спостережуваних рухів з інформацією про одометрію та шумом.
❸ Оновлення	На основі виміряної інформації датчика обчислюється ймовірність кожної частинки та значення ваги кожної частинки оновлюється на основі розрахованої ймовірності.
❹ Оцінка пози	Положення, орієнтація та вага всіх частинок використовуються для обчислення середньої

	ваги, середнє та максимальне значення ваги для оцінки пози роботи.
❺ Передискретизація	Крок генерації нових частинок полягає у видаленні легших частинок та у створенні нових частинок, які успадковують інформацію про позу зважених частинок. Кількість частинок N повинно підтримуватися.

Крім того, якщо кількість зразків є достатньою, фільтр частинок може бути більш точним ніж оцінка пози EKF або UKF, що покращило фільтр Калмана. Однак якщо цифр недостатньо, розрахунок може бути неточним. SLAM на основі Rao-Blackwelled Particle Filter (RBPF), який одночасно використовується як фільтр для частинок, так і фільтр Калмана, також широко икористовується як підхід до вирішення цієї проблеми.

Фільтр частинок

Щоб дізнатись більше про фільтри частинок, зверніться до книги „Імовірнісна робототехніка”, яка використовується як підручник в галузі робототехніки - Себастьян Трен (професор Стенфорда, співробітник Google, засновник Udacity). Я сильно рекомендую цю книгу всім, хто хоче вивчати робототехніку.

http://www.probabilistic-robotics.org/

https://www.udacity.com/course/cs373

OpenSLAM and Gmapping

Як пояснювалося, SLAM - це широко досліджена галузь робототехніки. Така інформація може бути знайдена в останніх академічних журналах та презентаціях, і багато з цих досліджень є відкритими, зібраними групою OpenSLAM і може бути знайдена на OpenSLAM.org. Цей сайт є обов'язковим для відвідування.

Тут також представлено gmapping, яке ми використовували в розділі 11.4, і спільнота ROS використовує його широко в SLAM. Щодо gmapping представлено дві статті. Одна була опублікована в ICRA 2005, а інша була опублікована в журналі "Robotics, IEEE Transactions on" у 2007 році.

Ці статті описують, як зменшити кількість частинок, щоб зменшити обчислювальний обсяг і реалізувати роботу в режимі реального часу. Основним підходом є використання описаного фільтра частинок Rao-Blackwellized вище. Зверніться до статті, щоб отримати докладнішу інформацію, а приблизний опис можна побачити під час обговорення фільтрів для частинок у розділі 11.4.2.

[1] Grisetti, Giorgio, Cyrill Stachniss, and Wolfram Burgard, Удосконалення сітчастого шлему з фільтрами частинок, що містять раоблаквелі, шляхом адаптивних пропозицій та вибіркової передискретизації, Праці 2005 Міжнародна конференція IEEE з робототехніки та автоматизації, с. 2432-2437, 2005.

[2] Grisetti, Giorgio, Cyrill Stachniss та Wolfram Burgard, Удосконалені методи для картографування сітки з рао-чорно-чорні фільтри для частинок, IEEE транзакції з робототехніки, том 23, No.1, с.34-46, 2007

11.5. Навігаційна практика

Перш ніж пояснювати навігацію, я поясню, як виконувати навігацію за допомогою TurtleBot3. Теорія навігації буде обговорена в розділі 11.7 після того, як буде обговорено застосування навігаційної програми.

Пакети ROS, пов'язані з навігацією, використані в цьому розділі, - це метапакет turtlebot3, попередня практика SLAM, "переміщення_бази" в навігаційному метапакеті, amcl та паети "map_server”. Встановлення цих пакетів було завершено під час попереднього SLAM розділу. Оскільки цей розділ є практичною практикою, я опишу лише метод виконання. Опис кожної упаковки наведено в наступному розділі.

Порядок виконання для навігації такий. У цьому прикладі ми будемо використовувати TurtleBot3 Waffle як довідковий. Якщо ви використовуєте Burger або Waffle Pi, просто змініть параметр "TURTLEBOT3_MODEL" у команді з "waffle" на "burger" або "waffle_pi".

roscore

Запустіть roscore на [віддаленому ПК].

```
$ roscore
```

Запуск робота

З [TurtleBot] запустіть файл „turtlebot3 robot.launch“ для виведення “turtlebot3_core“ та вузли ‘turtlebot3 lds’.

```
$ roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

Виконати навігаційний пакет

Запустіть файл ‘turtlebot3_navigation.launch’ на [віддаленому ПК]. Пакет „Turtlebot3_navigation“ складається з декількох запускових файлів. Коли пакет виконується, наступні вузли будуть запущені разом із інформацією про 3D-модель TurtleBot3.

■ Вузол „robot_state_publisher“ для публікації тривимірного положення та інформації про орієнтацію обох коліс та шарнірів у TF

■ Вузол ‘map_server’, який завантажує карту

■ Вузол AMCL (Адаптивна локалізація Монте-Карло)

■ Вузол ‘move_base’.

```
$ export TURTLEBOT3_MODEL=waffle
$ roslaunch turtlebot3_navigation turtlebot3_navigation.launch map_file:=$HOME/map.yaml
```

Виконайте Rviz

Давайте запустимо RViz, інструмент візуалізації ROS, який забезпечує візуальне підтвердження цільової пози призначення та результати в навігації. Коли ви запускаєте RViz з наступними опціями, це дуже зручно додавати модулі відображення з нуля.

```
$ rosrun rviz rviz -d `rospack find turtlebot3_navigation`/rviz/turtlebot3_nav.rviz
```

Після запуску наведеної вище команди ви побачите екран, показаний на рис. 169. На правій карті ви побачите безліч зелених стрілок, які є частинками описаного фільтра частинок в теорії SLAM. Це буде пояснено пізніше, але навігація також використовує фільтри частинок. Робот є посеред зелених стрілок.

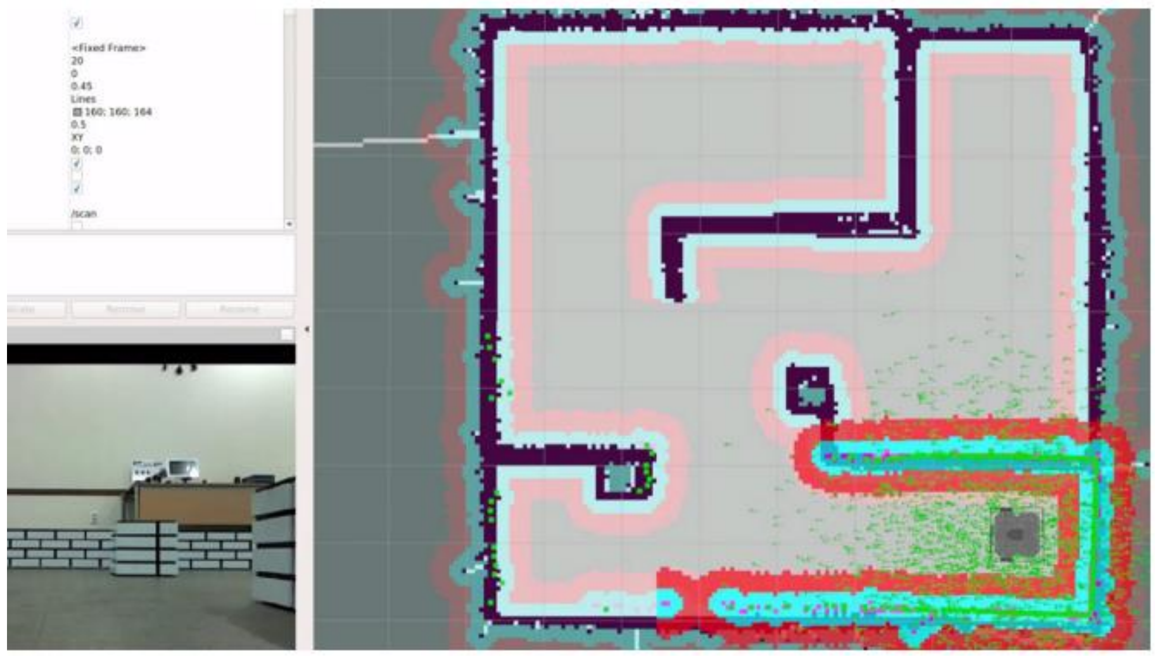


Рис. 169 Частинки, видимі в Rviz (зелені стрілки навколо робота)

Оцініть початкову позу

По-перше, слід виконати початкову оцінку пози робота. Коли ви натискаєте [2D Pose Estimate] в меню RViz з'являється дуже велика зелена стрілка. Перемістіть її в позу, де фактично знаходиться робот на даній карті, і, утримуючи ліву кнопку миші, перетягніть зелену стрілка в напрямку, куди спрямована передня частина робота. Це

команда для оцінки пози робота на ранній стадії. Потім рухайте робота вперед і назад за допомогою інструментів, такі як вузол ‘turtlebot3_teleop_keyboard’ для збору інформації про навколишнє середовище та дізнатися, де на карті зараз знаходиться робот. Коли цей процес завершено, робот оцінює своє фактичне положення та орієнтацію, використовуючи положення і орієнтацію, вказану зеленою стрілкою як початкова поза.

Встановлення пункту призначення та переміщення робота

Коли все буде готово, спробуємо команду переміщення з навігаційного графічного інтерфейсу. Якщо натиснути [2D Nav Goal] в меню RViz з'являється дуже велика зелена стрілка. Ця зелена стрілка є маркером який може вказати пункт призначення робота. Корінь стрілки - це позиція «x» та «y» робота, а орієнтація, вказана стрілкою, є напрямком «тета» робота. Клацніть цю стрілку в місці, де робот рухатиметься, і перетягніть його, щоб встановити орієнтацію. Робот створить шлях, щоб уникнути перешкод до пункту призначення на основі карти (див. рис. 170).

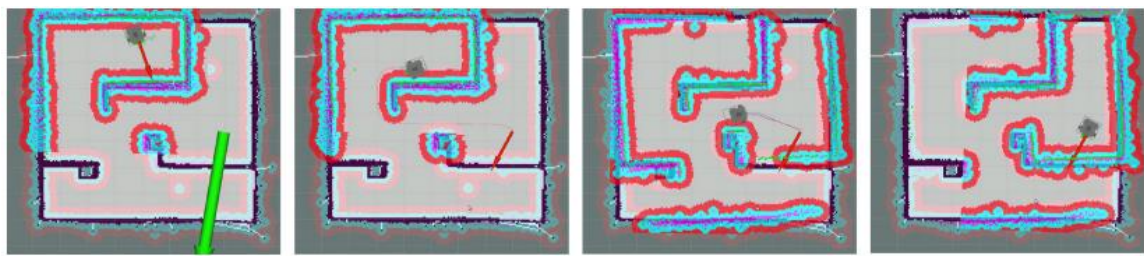


Рис. 170 Налаштування призначення (великі стрілки) і те, як рухається робот (Destination settings (big arrows) and how the robot is moving)

У наступних розділах описуються деталі джерела та способи налаштування пакунків, які ви маєте запустити раніше. Він буде поділений на практику, застосування та теорію лише як приклад SLAM.

11.6. Додаток навігації

Розділ 11.5 був ручною практикою навігації, і цей розділ досліджує пакет ROS, який використовується в навігації та як її створити та налаштувати. Ми розглянемо метапакет TurtleBot3 і вузол «turtlebot3_lds» драйвера LDS, інформація про тривимірну модель TurtleBot3 (turtlebot3_description), вузол ‘map_server’, який завантажує раніше створену карту, Адаптивний вузол локалізації Монте-Карло та вузол ‘move_base’. Мета цього розділу полягає у застосуванні навігації на вашому роботі.

У цьому розділі ми пояснимо на основі роботизованої платформи TurtleBot3 та датчика LDS, що використовується разом з ним. Розуміючи цю інструкцію, ви можете самостійно виконувати навігацію робота, який не обмежується конкретною платформою або конкретними датчиками. Якщо ви хочете створити свою власну платформу робота або побудуйте спеціальний робот на основі робочої платформи TurtleBot3, цей підручник буде корисним.

Навігація полягає в переміщенні робота з одного місця до вказаного пункту призначення в даному навколишньому середовищі. Для цього потрібно скласти карту, яка містить геометричну інформацію про меблі, предмети і стіни даного середовища. Як

описано в попередньому розділі SLAM, карта була створена з інформацією про відстань, отриману датчиком, та інформацією про позу самого робота.

Навігація дозволяє роботові перейти від поточної пози до призначеної цілі на карті за допомогою карти, кодера робота, інерційного датчика та датчика відстані. Процедура для виконання цього завдання є наступною.

Зондування

На карті робот оновлює інформацію про свою одометрію за допомогою кодера та інерційного датчика (Датчик IMU) і вимірює відстань від пози датчика до перешкоди (стіни, предмет, меблі тощо).

Локалізація / оцінка пози

На основі величини обертання колеса від кодера, інформації про інерцію від IMU датчика та інформації про відстань від датчика до перешкоди, оцінка локалізації / пози поточного робота виконується на створеній раніше карті. Тут багато методів оцінки пози, але в цьому розділі ми будемо використовувати метод локалізації фільтра частинок та адаптивну локалізацію Монте-Карло (AMCL), що є варіантом Monte Carlo Localization (MCL).

Планування руху

Планування руху, яке також називають плануванням шляху, створює траєкторію з поточного позувати до цільової пози, зазначеної на карті. Створений план шляху включає глобальний шлях планування

на цілій карті та місцеве планування шляху для менших територій навколо робота. Ми плануємо використовувати пакети планування маршрутів ‘move_base’ та ‘nav_core’ у ROS на основі Dynamic Віконний підхід (DWA), який є алгоритмом уникнення перешкод.

Уникнення пересування / перешкод

Якщо команда, дана роботові, базується на основі траєкторії, створеної рухом Плануючи, робот рухається до пункту призначення за запланованим шляхом. Оскільки зондування, оцінка пози та планування руху все ще виконуються під час руху, перешкод або рухомих об’єктів, які раптово з’являються, можна буде уникнути, використовуючи підхід динамічного вікна (DWA).

Рис. 171 ілюструє взаємозв'язок між основними вузлами та темами для запуску навігаційного пакету ROS. Ми зупинимось на інформації, необхідній для навігації. Назва теми і тип повідомлення теми відображаються окремо при описі теми на малюнку 171. Наприклад, у випадку одометрії ‘/ odom’ - це назва теми, а ‘nav_msgs / Odometry’ - форма повідомлення теми.

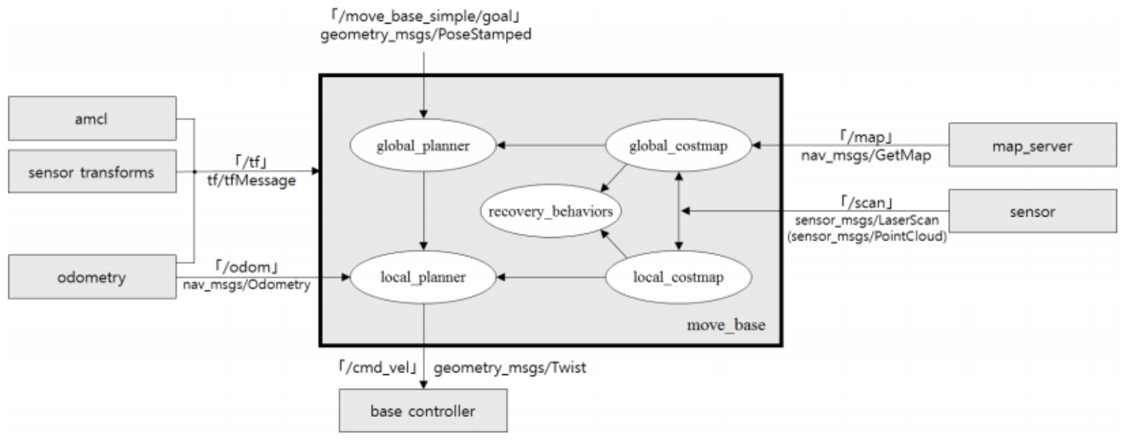


Рис. 171 Взаємозв'язок між основними вузлами та темами у конфігурації навігаційних пакетів

Одометрія ('/ odom', nav_msgs / Odometry)

Інформація про одометрію робота використовується для планування місцевого шляху шляхом отримання такої інформації, як поточна швидкість робота, створює локальний шлях або уникає перешкод.

Перетворення координат ('/ tf', tf / tfMessage)

Оскільки поза датчика робота змінюється залежно від апаратної конфігурації робота, ROS використовує відносне перетворення координат (TF). Це просто описує відносні x, y, z координати датчика від одометрії робота. Наприклад, координати датчика можна отримати за допомогою odom → base_footprint → base_link → base_scan перетворення та опубліковані за темою. Вузол 'move_base' отримує тему та виконує шлях планування з позою робота і датчика.

Датчик відстані (‘/ scan’, sensor_msgs / LaserScan або sensor_msgs / PointCloud)

Це стосується значення відстані, виміряного від датчика. LDS та RealSense, Kinect, Xtion є загальновживаними. Цей датчик відстані використовується для оцінки поточної пози робота або планування руху робота за допомогою AMCL (адаптивна локалізація Монте-Карло).

Карта (‘/ map’, nav_msgs / GetMap)

Навігація використовує сіткову карту заповнення. У цьому посібнику ми використовуватимемо "map.pgm" та "map.yaml" створений із попереднього розділу за допомогою пакета ‘map_server’.

Цільові координати (‘/ move_base_simple / goal’, geometry_msgs / PoseStamped)

Цільові координати задаються користувачем. Додаткова команда координати цілі пакет можна створити за допомогою такого пристрою, як планшет, але в цьому розділі ми будемо використовувати RViz для встановлення координати цілі. Координати цілі складаються з двовимірних координат (x, y) і напрямок θ.

Команда швидкості (‘/ cmd_vel’, geometry_msgs / Twist)

Робота можна перенести до координати призначення, опублікувавши команду швидкості, що йде за запланованим шляхом.

Як описано в розділі 11.5, якщо файли „turtlebot3_robot.launch“ та „turtlebot3_navigation.launch“ запущені на [черепаха], робот готовий до навігації.

```
$ roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

```
$ export TURTLEBOT3_MODEL=waffle
$ roslaunch turtlebot3_navigation turtlebot3_navigation.launch map_file:=$HOME/map.yaml
```

Коли ви виконуєте "rqt_graph", який візуалізує інформацію про вузол і тему, що працює на ROS інформація може бути візуально показана на рис. 171. Як ви можете бачити на схемі, інформація, необхідна для навігації, описаної вище, публікується та підписується як назви тем «/ odom», «/ tf», «/ scan», «/ map» та «/ cmd_vel». "move_base_simple/goal" опубліковано, коли координата призначення вказана з RViz.

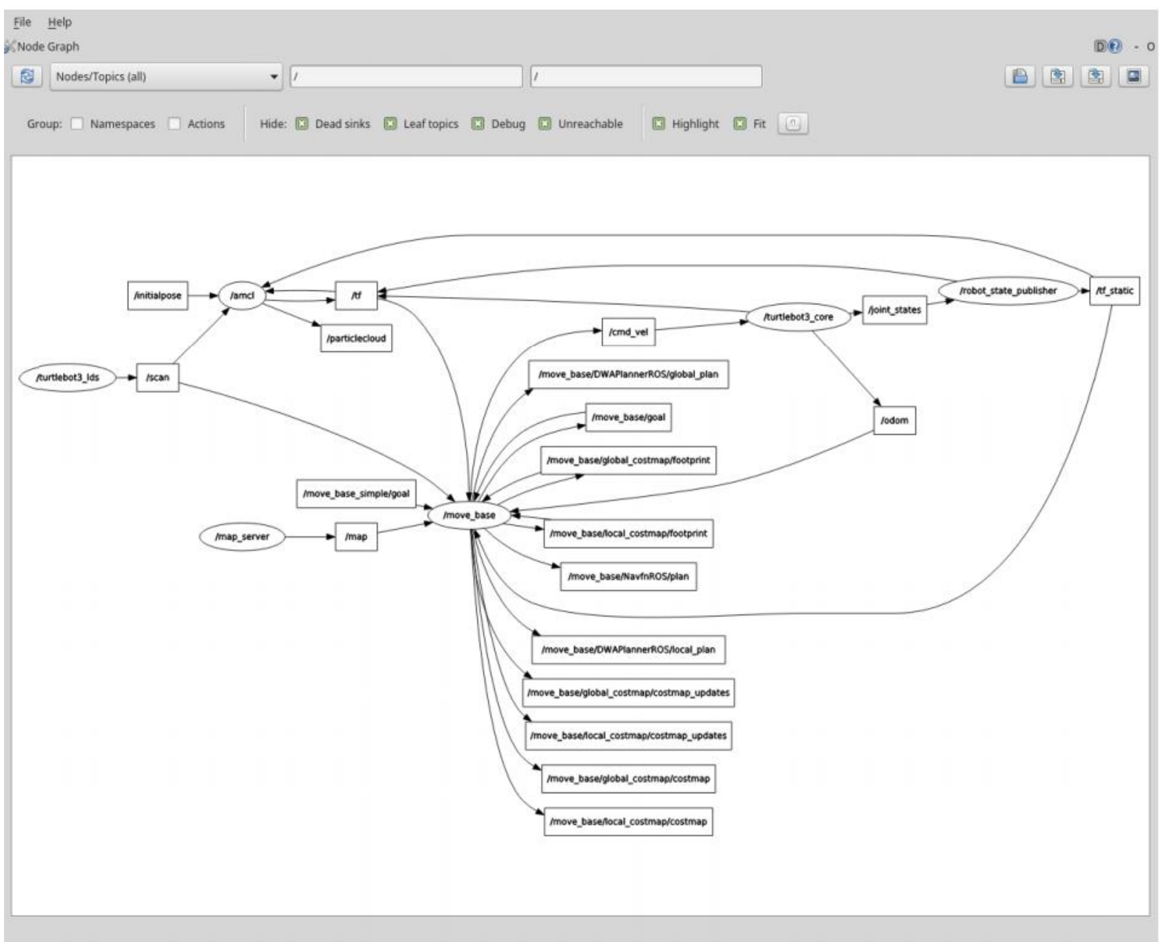


Рис. 171 Вузол і тематичний стан навігації turtlebot3 (Node and Topic state of turtlebot3_navigation)

Пакет ‘turtlebot3_navigation’ містить файл запуску, який запускає пов’язані з навігацією вузли та пакети, а також файли конфігурації, такі як файл xml, файл yaml, який налаштовує різні параметри, файл карти та файл конфігурації rviz. Нижче наведено деталі цих файлів.

Таблиця 25

/launch/turtlebot3_navigation.launch	Файл ‘turtlebot3_navigation.launch’ запускає всі пакунки, пов’язані з навігацією.
/launch/amcl.launch.xml	Файл ‘amcl.launch.xml’ містить різні параметри Adaptive Monte Carlo Localization (AMCL) і використовується з файлом ‘turtlebot3_navigation.launch’.
/param/move_base_params.yaml	Цей конфігураційний файл налаштовує параметр "move_base", який контролює планування руху.
/param/costmap_common_params_burger.yaml /param/costmap_common_params_waffle.yaml /param/global_costmap_params.yaml /param/local_costmap_params.yaml	Навігація використовує карту сітки заповнення, описану в розділі

11.3.1.

Виходячи з цієї заповнюваності карта сітки, кожен піксель обчислюється як перешкода, нерухома площа та рухома зона пози робота та навколишню інформацію, отриману від датчика. У цьому розрахунку значення застосовується концепція калькуляції витрат. Наведені вище файли задають параметри калькуляції витрат.

'Costmap_common_params.yaml' має загальні параметри, де '*global_costmap_params. файл yaml*'необхідний для глобального планування руху області, тоді як файл' *local_costmap_params.yaml '* необхідний для планування руху місцевих територій. З'являється *'costmap_common_params.yaml'* з суфіксом Burger або Waffle відповідно до моделі робота і файл містить різні інформація для кожної моделі. Однак модель TurtleBot3 Waffle Pi така ж, як і TurtleBot3 Waffle, крім камери. Тож модель TurtleBot3 Waffle Pi використовує 'waffle' суфікс для використання налаштувань TurtleBot3 Waffle.

/param/dwa_local_planner_params.yaml

«Dwa_local_planner» - це пакет, який в кінцевому підсумку передає команду швидкості роботові та встановлює параметри для нього.

base_local_planner_params.yaml

Цей файл містить значення конфігурації для «base_local_planner», однак він не використовується, оскільки

turtlebot3 використовує замість цього "dwa_local_planner". Це тому, що параметр в вузлі ‘move_ base’ був заздалегідь змінений таким чином:

```
<param name="base_local_planner" value="dwa_local_planner/DWAPlannerROS" />
```

/maps/map.pgm

/maps/map.yaml

Збережіть і використовуйте раніше створену карту сітки заповнення в папці «/ maps».

/rviz/turtlebot3_nav.rviz

Цей файл містить інформацію про налаштування RViz. Цей файл використовується для завантаження Grid, RobotModel, TF, LaserScan, Map, Global Map, Local Map та AMCL Particles з модуля RViz display.

Наступний файл ‘turtlebot3_navigation.launch’ містить деталі моделі робота, ‘robot_state_publisher ', виконання та конфігурація сервера серверів карт, AMCL та' move_base '.

```
---------------------------------------------------------
/launch/turtlebot3_navigation.launch
<launch>
<arg name="model" default="$(env TURTLEBOT3_MODEL)"
doc="model type [burger, waffle, waffle_pi]"/>

 <!-- Turtlebot3 -->
```

```
<include file="$(find
turtlebot3_bringup)/launch/turtlebot3_remote.launch" />
 <!-- Map server -->
 <arg name="map_file" default="$(find
turtlebot3_navigation)/maps/map.yaml"/>
 <node name="map_server" pkg="map_server" type="map_server"
args="$(arg map_file)">
 </node>
 <!-- AMCL -->
 <include file="$(find
turtlebot3_navigation)/launch/amcl.launch.xml"/>
 <!-- move_base -->
 <arg name="cmd_vel_topic" default="/cmd_vel" />
 <arg name="odom_topic" default="odom" />
 <node pkg="move_base" type="move_base" respawn="false"
name="move_base" output="screen">
 <param name="base_local_planner"
value="dwa_local_planner/DWAPlannerROS" />
 <rosparam file="$(find
turtlebot3_navigation)/param/costmap_common_params_$(arg
model).yaml"
command="load" ns="global_costmap" />
```

```
 <rosparam                                              file="$(find
turtlebot3_navigation)/param/costmap_common_params_$(arg
model).yaml"
command="load" ns="local_costmap" />
 <rosparam                                              file="$(find
turtlebot3_navigation)/param/local_costmap_params.yaml"
command="load" />
 <rosparam                                              file="$(find
turtlebot3_navigation)/param/global_costmap_params.yaml"
command="load" />
 <rosparam                                              file="$(find
turtlebot3_navigation)/param/move_base_params.yaml"
command="load" />
 <rosparam                                              file="$(find
turtlebot3_navigation)/param/dwa_local_planner_params.yaml"
command="load" />
```

```
 <remap from="cmd_vel" to="$(arg cmd_vel_topic)"/>
 <remap from="odom" to="$(arg odom_topic)"/>
 </node>
</launch>
--------------------------------------------------------------------------------------
---
```

Модель робота і TF

У цьому розділі пояснюється завантаження робочої 3D-моделі TurtleBot3 із "turtlebot3_description" упаковує та публікує стан робота, наприклад спільну інформацію через «robot_state_publisher» у формі відносного перетворення координат, "tf". Точніше, «tf» одометрії - це опубліковано з "turtlebot3_core", а інші координати відносно трансформовані (odom → base_footprint → base_link → base_scan) на основі перетворення координат, описаного в імпортовану модель робота та опублікуйте у "tf". Завдяки цим процесам, 3D-модель робота можна побачити в RViz та позу вимірювання величини відстані, отриманої з датчика можна знайти за допомогою "tf".

```
<!-- Turtlebot3 -->
<include file="$(find turtlebot3_bringup)/launch/turtlebot3_remote.launch" />
```

```
--------------------------------------------
turtlebot3_bringup/launch/turtlebot3_remote.launch


<launch>
 <arg name="model" default="$(env TURTLEBOT3_MODEL)"
doc="model type [burger, waffle,
waffle_pi]"/>
 <include file="$(find
turtlebot3_bringup)/launch/includes/description.launch.xml">
 <arg name="model" value="$(arg model)" />
```

```
 </include>
 <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher"
output="screen">
 <param name="publish_frequency" type="double" value="50.0" />
 </node>
</launch>
---------------------------------------------------------------
```

Сервер карт

Вузол map_server завантажує інформацію про карту (map.yaml) та карту (map.pgm), що зберігаються у папці ‘turtlebot3_navigation / maps /’. Карта опублікована у вигляді теми вузол ‘map_server’.

```
<!-- Map server -->
<arg name="map_file" default="$(find turtlebot3_navigation)/maps/map.yaml"/>
<node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)">
</node>
```

AMCL(Adaptive Monte Carlo Localization)

Запустіть вузол amcl для AMCL та встановіть відповідні параметри. Це детально висвітлено у розділі 11.6.5.

```
<include file="$(find turtlebot3_navigation)/launch/amcl.launch.xml"/>
```

move_base

Встановіть параметри, пов’язані з картою витрат, необхідні для планування руху, та встановіть параметри для „dwa_local_planner ', який передає роботові команду швидкості переміщення та встановлює

параметри для "Move_base", який контролює планування руху. Детальні пояснення доступні в розділі 11.6.6.

```
<arg name="cmd_vel_topic" default="/cmd_vel" />
<arg name="odom_topic" default="odom" />
<node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">
  <param name="base_local_planner" value="dwa_local_planner/DWAPlannerROS" />

  <rosparam file="$(find turtlebot3_navigation)/param/costmap_common_params_$(arg model).yaml"
command="load" ns="global_costmap" />
  <rosparam file="$(find turtlebot3_navigation)/param/costmap_common_params_$(arg model).yaml"
command="load" ns="local_costmap" />
  <rosparam file="$(find turtlebot3_navigation)/param/local_costmap_params.yaml" command="load" />
  <rosparam file="$(find turtlebot3_navigation)/param/global_costmap_params.yaml" command="load" />
  <rosparam file="$(find turtlebot3_navigation)/param/move_base_params.yaml" command="load" />
  <rosparam file="$(find turtlebot3_navigation)/param/dwa_local_planner_params.yaml"
command="load" />

  <remap from="cmd_vel" to="$(arg cmd_vel_topic)"/>
  <remap from="odom" to="$(arg odom_topic)"/>
</node>
```

Давайте встановимо докладні параметри для "turtlebot3_navigation", про які ми вже говорили раніше.

AMCL (Adaptive Monte Carlo Localization)

Файл ‘amcl.launch.xml’ містить параметри AMCL і використовується з ‘turtlebot3_ navigation.launch’ , описаний вище. Опис AMCL буде розглянуто в навігаційній теорії.

```
-----------------------------------------------------
turtlebot3_navigation/launch/amcl.launch.xml
<launch>
 <!-- if true, AMCL receives map topic instead of service call. -->
 <arg name="use_map_topic" default="false"/>
 <!-- topic name for the sensor values from the distance sensor. -->
 <arg name="scan_topic" default="scan"/>
 <!-- used as the initial x-coordinate value of the Gaussian distribution in
initial pose
estimation.-->
 <arg name="initial_pose_x" default="0.0"/>
 <!-- used as the initial y-coordinate value of the Gaussian distribution in
the initial pose
estimation.-->
 <arg name="initial_pose_y" default="0.0"/>
 <!-- used as the initial yaw coordinate value of the Gaussian distribution
in the initial pose
estimation. -->
 <arg name="initial_pose_a" default="0.0"/>
 <!-- execute the amcl node by referring to the parameter settings below.
-->
 <node pkg="amcl" type="amcl" name="amcl">
 <!-- filter related parameter -->
```

```
<!-- min number of particles allowed -->
<param name="min_particles" value="500"/>
<!-- max number of particles allowed (the higher the better; set based on PC performance) -->
<param name="max_particles" value="3000"/>
<!-- max error between the actual distribution and the estimated distribution -->
<param name="kld_err" value="0.02"/>
<!-- translational motion required for filter update (meter) -->
<param name="update_min_d" value="0.2"/>
<!-- rotational motion required for filter update (radian) -->
<param name="update_min_a" value="0.2"/>
<!-- resampling interval -->
<param name="resample_interval" value="1"/>
<!-- conversion allowed time (by sec) -->
<param name="transform_tolerance" value="0.5"/>
<!-- index drop rate(slow average weight filter), deactivated if 0.0 -->
<param name="recovery_alpha_slow" value="0.0"/>
<!-- index drop rate(fast average weight filter), deactivated if 0.0 -->
<param name="recovery_alpha_fast" value="0.0"/>
<!-- refer to above initial_pose_x -->
<param name="initial_pose_x" value="$(arg initial_pose_x)"/>
<!-- refer to above initial_pose_y -->
```

```
<param name="initial_pose_y" value="$(arg initial_pose_y)"/>
<!-- refer to above initial_pose_a -->
<param name="initial_pose_a" value="$(arg initial_pose_a)"/>
<!-- max period to visually displaying scan and path info -->
<!-- example: 10Hz = 0.1sec, deactivated if -1.0 -->
<param name="gui_publish_rate" value="50.0"/>
<!-- same as the explanation for use_map_topic -->
<param name="use_map_topic" value="$(arg use_map_topic)"/>
<!--distance sensor parameter -->
<!-- change the sensor topic name -->
<remap from="scan" to="$(arg scan_topic)"/>
<!-- max distance of laser sensing distance (meter) -->
<param name="laser_max_range" value="3.5"/>
<!-- max number of laser beams used during filter update -->
<param name="laser_max_beams" value="180"/>
<!-- z_hit mixed weight of sensor model (mixture weight) -->
<param name="laser_z_hit" value="0.5"/>
<!-- z_short mixed weight of sensor model (mixture weight) -->
<param name="laser_z_short" value="0.05"/>
<!-- z_max mixed weight of sensor model (mixture weight) -->
<param name="laser_z_max" value="0.05"/>
<!-- x_rand mixed weight of sensor model (mixture weight) -->
<param name="laser_z_rand" value="0.5"/>
```

```
<!-- standard deviation of Gaussian model using z_hit of sensor -->
<param name="laser_sigma_hit" value="0.2"/>
<!-- index drop rate parameter for z_short of sensor -->
<param name="laser_lambda_short" value="0.1"/>
<!-- max distance and obstacle for likelihood_field method sensor -->
<param name="laser_likelihood_max_dist" value="2.0"/>
<!-- sensor type(select likelihood_field or beam) -->
<param name="laser_model_type" value="likelihood_field"/>
<!-- parameter related to odometry -->
<!-- robot driving methods. "diff" or "omni" can be selected -->
<param name="odom_model_type"value="diff"/>
<!-- estimated rotational motion noise of the odometry during rotational
motion -->
<param name="odom_alpha1" value="0.1"/>
<!-- estimated rotational motion noise of the odometry during
translation motion -->
<param name="odom_alpha2" value="0.1"/>
<!-- estimated translation motion noise of the odometry during
translation motion -->
<param name="odom_alpha3" value="0.1"/>
<!-- estimated translation motion noise of the odometry during
rotational motion -->
<param name="odom_alpha4" value="0.1"/>
```

```
<!-- odometry frame -->
<param name="odom_frame_id" value="odom"/>
<!-- robot base frame -->
<param name="base_frame_id" value="base_footprint"/>
</node>
</launch>
----------------------------------------------------------------
```

move_base

Це файл налаштування параметрів "move_base", який контролює планування руху.

```
-----------------------------------------------
turtlebot3_navigation/param/move_base_params.yaml
# choosing whether to stop the costmap node when move_base is inactive
shutdown_costmaps: false
# cycle of control iteration (in Hz) that orders the speed command to the
robot base
controller_frequency: 3.0
# maximum time (in seconds) that the controller will listen for control
information before the
space-clearing operation is performed
controller_patience: 1.0
# repetition cycle of global plan (in Hz)
planner_frequency: 2.0
```

```
# maximum amount of time (in seconds) to wait for an available plan
before the space-clearing
operation is performed
planner_patience: 1.0
# time (in sec) allowed to allow the robot to move back and forth before
executing the recovery
behavior.
oscillation_timeout: 10.0
# oscillation_timeout is initialized if you move the distance below the
distance (in meter) that
the robot should move so that it does not move back and forth.
oscillation_distance: 0.2
# Obstacles farther away from fixed distance are deleted on the map
during costmap initialization
of the restore operation
conservative_reset_dist: 0.1
-------------------------------------------------------------
```

costmap

Навігація використовує сіткову карту заповнення. На основі цієї сіткової карти заповнення кожен піксель становить обчислюється як перешкода, нерухлива площа та рухома площа на основі пози роботи робота навколишня інформація, отримана від датчика. У цьому розрахунку поняття калькуляції витрат застосовується. Параметр

конфігурації карти витрат складається з ‘*costmap_common_params. файл yaml, файл „global_costmap_params.yaml“* для глобального планування руху області та *„local_файл costmap_params.yaml* ’для планування руху місцевих територій.

"Costmap_common_params_" файл burger.yaml 'використовується для бургерів, а файл *costmap_common_params_waffle.yaml* 'використовується для Waffle. Однак модель TurtleBot3 Waffle Pi така ж, як модель TurtleBot3 Waffle крім камери. Тож модель TurtleBot3 Waffle Pi використовує суфікс ‘waffle’, щоб використовувати налаштування моделі TurtleBot3 Waffle.

Далі наведено налаштування параметрів для Burger TurtleBot3.

```
-------------------------
turtlebot3_navigation/param/costmap_common_params_burger.yaml

# Indicate the object as an obstacle when the distance between the robot
and obstacle is within
this range.
obstacle_range: 2.5
# sensor value that exceeds this range will be indicated as a freespace
raytrace_range: 3.5
```

```
# external dimension of the robot is provided as polygons in several points
footprint: [[-0.110, -0.090], [-0.110, 0.090], [0.041, 0.090], [0.041, -
0.090]]
# radius of the robot. Use the above footprint setting instead of
robot_radius.
# robot_radius: 0.105
# radius of the inflation area to prevent collision with obstacles
inflation_radius: 0.15
# scaling variable used in costmap calculation. Calculation formula is as
follows.
# scaling
# exp(-1.0 * cost_scaling_factor *(distance_from_obstacle -
inscribed_radius)) *(254 - 1)
cost_scaling_factor: 0.5
# select costmap to use between voxel(voxel-grid) and
costmap(costmap_2d)
map_type: costmap
# tolerance of relative coordinate conversion time between tf
transform_tolerance: 0.2
# specify which sensor to use
observation_sources: scan
# set data type and topic, marking status, minimum obstacle for the laser
scan
```

```
scan: {data_type: LaserScan, topic: scan, marking: true, clearing: true}
------------------------------------------------------
```

Нижче наведено параметри для програми TurtleBot3 Waffle. На відміну від бургера TurtleBot3 Burger, waffle's "Footprint" і "inflation_radius", що запобігає зіткненню з перешкодою, різні. Всі інші значення однакові, а для опису параметрів зверніться до опису Burger.

```
------------------
turtlebot3_navigation/param/costmap_common_params_waffle.yaml
obstacle_range: 2.5
raytrace_range: 3.5
footprint: [[-0.205, -0.145], [-0.205, 0.145], [0.077, 0.145], [0.077, -0.145]]
inflation_radius: 0.20
cost_scaling_factor: 0.5
map_type: costmap
transform_tolerance: 0.2
observation_sources: scan
scan: {data_type: LaserScan, topic: scan, marking: true, clearing: true}
--------------------------------------------------


-------------------------------
turtlebot3_navigation/param/global_costmap_params.yaml
```

```
global_costmap:
 global_frame: /map                      # set map frame
 robot_base_frame: /base_footprint       # set robot's base frame
 update_frequency: 2.0            # update frequency
 publish_frequency: 0.1           # publish frequency
 static_map: true                 # setting whether or not to use given map
 transform_tolerance: 1.0                # transform tolerance time
```

---

turtlebot3_navigation/param/local_costmap_params.yaml

```
local_costmap:
 global_frame: /odom                     # set map frame
 robot_base_frame: /base_footprint       # set robot's base frame
 update_frequency: 2.0            # update frequency
 publish_frequency: 0.5           # publish frequency
 static_map: false                # setting whether or not to use given map
 rolling_window: true                    # local map window setting
 width: 3.5                       # area of local map window
 height: 3.5                      # height of local map window
 resolution: 0.05                 # resolution of local map window
(meter/cell)
 transform_tolerance: 1.0                # transform tolerance time
```

```
-------------------------------------------------------------------
```

dwa_local_planner

Пакет "dwa_local_planner" в кінцевому підсумку публікує команду швидкості для робота і цей файл встановлює для нього параметри.

```
-----------------------------------
turtlebot3_navigation/param/dwa_local_planner_params.yaml
DWAPlannerROS:

 # robot parameters
 max_vel_x: 0.18          # max velocity for x axis(meter/sec)
 min_vel_x:-0.18          # min velocity for x axis (meter/sec)
 max_vel_y: 0.0           # Not used. applies to omni directional robots
only
 min_vel_y: 0.0           # Not used. applies to omni directional robots
only
 max_trans_vel: 0.18            # max translational velocity(meter/sec)
 min_trans_vel: 0.05            # min translational velocity (meter/sec),
negative value for reverse
 # trans_stopped_vel: 0.01      # translation stop velocity(meter/sec)
```

```
 max_rot_vel: 1.8          # max rotational velocity(radian/sec)
 min_rot_vel: 0.7          # min rotational velocity (radian/sec)
 # rot_stopped_vel: 0.01  # rotation stop velocity (radian/sec)
 acc_lim_x: 2.0            # limit for x axis acceleration(meter/sec^2)
 acc_lim_y: 0.0            # limit for y axis acceleration (meter/sec^2)
 acc_lim_theta: 2.0                # theta axis angular acceleration limit
(radian/sec^2)

# Target point error tolerance
 yaw_goal_tolerance: 0.15        # yaw axis target point error tolerance
(radian)
 xy_goal_tolerance: 0.05 # x, y distance Target point error tolerance
(meter)

 # Forward Simulation Parameter
 sim_time: 3.5                    # forward simulation trajectory time
 vx_samples: 20            # number of sample in x axis velocity space
 vy_samples: 0             # number of sample in y axis velocity space
 vtheta_samples: 40               # number of sample in theta axis velocity
space
 # Trajectory scoring parameter (trajectory evaluation)
 # Score calculation used for the trajectory evaluation cost function is as
follows.
```

```
# cost =
# path_distance_bias * (distance to path from the endpoint of the trajectory in meters)
# + goal_distance_bias * (distance to local goal from the endpoint of the trajectory in meters)
# + occdist_scale * (maximum obstacle cost along the trajectory in obstacle cost (0-254))
path_distance_bias: 32.0           # weight value of the controller that follows the given path
goal_distance_bias: 24.0           # weight value for the goal pose and control velocity
occdist_scale: 0.04                # weight value for the obstacle avoidance
forward_point_distance: 0.325      # distance between the robot and additional scoring point (meter)
stop_time_buffer: 0.2    # time required for the robot to stop before collision (sec)
scaling_speed: 0.25              # scaling Speed (meter/sec)
max_scaling_factor: 0.2 # max scaling factor
# Oscillation motion prevention paramter
# distance the robot must move before the oscillation flag is reset
oscillation_reset_dist: 0.05
# Debugging
```

```
 publish_traj_pc: true              # debugging setting for the movement
trajectory
 publish_cost_grid_pc: true       # debugging setting for costmap
 global_frame_id: odom  # ID setting for global frame
-------------------------------------------------------
```

Map

Створена раніше карта сітки заповнюваності зберігається у *папці / / maps.* Інших параметри конфігурації немає.

```
/maps/map.pgm
/maps/map.yaml
```

turtlebot3_nav.rviz

Цей файл містить інформацію про налаштування RViz і викликає модулі відображення RViz, такі як Grid, модель робота, TF, LaserScan, карта, глобальна карта, локальна карта та частинки Amcl. Це рекомендується використовувати наступний файл без будь-якої конфігурації. Ви можете використовувати параметри під час запуску команди, як показано нижче.

```
$ rosrun rviz rviz -d `rospack find turtlebot3_navigation`/rviz/turtlebot3_nav.rviz
```

Пояснено деталі використання навігаційного пакету. У наступному розділі ми обговоримо теорію калькуляції витрат, Adaptive Monte Carlo Localization (AMCL), та Dynamic Window Approach (DWA).

11.7. Теорія навігації

Поза робота оцінюється на основі одометрії, отриманої від кодера, та інерційного датчик (датчик IMU). А відстань між роботом і перешкодою отримується за допомогою датчика відстані, встановленого на роботі. Поза робота і датчика, інформація про перешкоди та карта сітки заповнення, отримана в результаті SLAM, використовується для завантаження статичної карти та використовує орієнтовану, вільну та невідому зони для навігації.

У навігації, карта витрат обчислює площу перешкоди, можливу зону зіткнення та площу зони руху робота на основі згаданих чотирьох факторів. Залежно від типу навігації, карти витрат можна розділити на два види. Одним із них є глобальна карта витрат, яка встановлює план шляху для навігації глобальна область фіксованої карти. Інший - „local_costmap", який використовується для планування шляху та уникнення перешкод на обмеженій ділянці навколо робота. Хоча їх цілі є різні, обидві карти витрат представлені однаково.

Калькуляція витрат виражається як значення від «0» до «255». Показано значення значення на рис. 172, і для короткого підсумовування значення використовується для визначення того, чи робот рухається або стикається з перешкодою. Розрахунок кожної площі залежить від карти витрат, параметрів конфігурації, зазначені в розділі 11.6.

- 000: Вільна зона, де робот може вільно пересуватися
- 001 ~ 127: Області з низькою ймовірністю зіткнення

■ 128 ~ 252: Області високої ймовірності зіткнення

■ 253 ~ 254: Область зіткнення

■ 255: Окупована зона, де робот не може рухатися

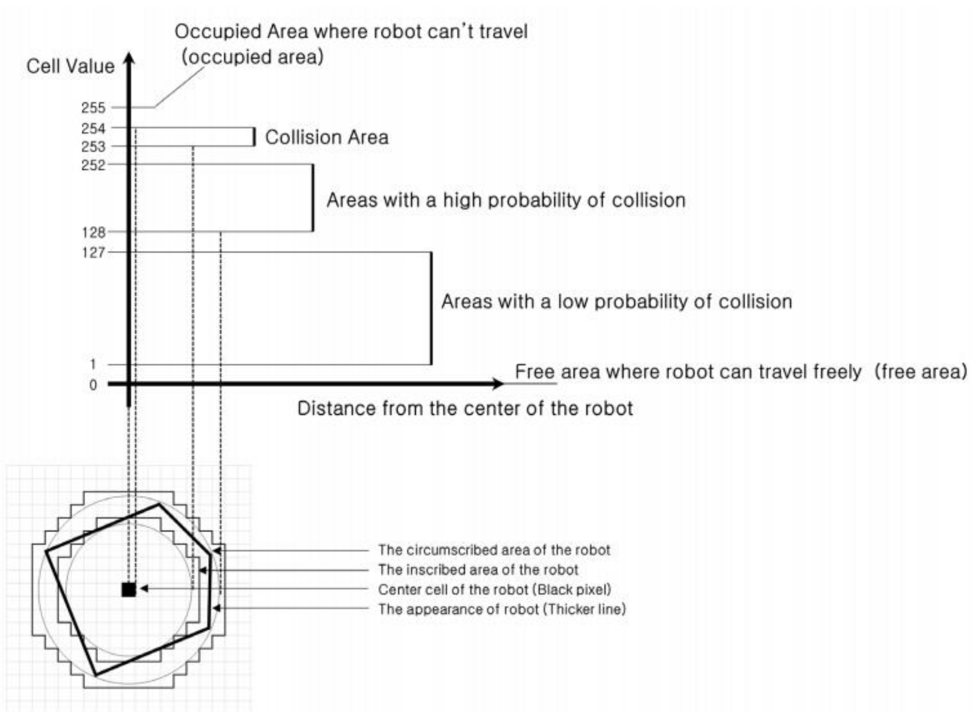


Рис. 172 Взаємозв'язок між відстанню до перешкоди та значенням карти витрат

Наприклад, фактична карта витрат виражається, як показано на рис. 173. Детальніше, є модель робота посередині і прямокутна коробка навколо нього, що відповідає зовнішній поверхні робота. Коли ця контурна лінія торкається стіни, робот також вріжеться у стіну. Зелений колір відображає перешкоду із значенням датчика відстані, отриманим від лазерного датчика. Якщо карта витрат сірого

стає темнішою, імовірно, це буде область зіткнення. Те саме працює у випадку кольорового подання. Рожева область є фактичною перешкодою, а світло-блакитна - точка, де центр робота потрапляє в цю область, і межа проводиться товстою червоною лінією. Колір не має важливого значення, оскільки користувач може змінити його в RViz.

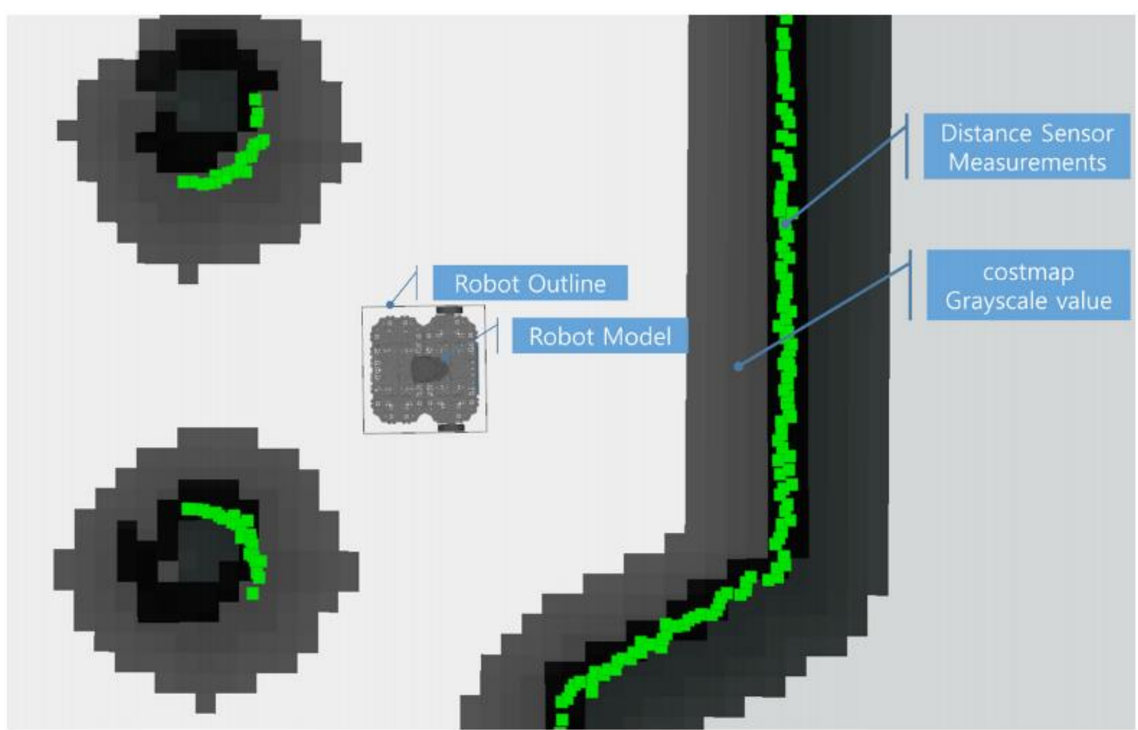


Рис. 173 Представлення карт витрат (сіра шкала)

Як згадувалося в розділі 11.4 Фільтр частинок теорії SLAM, локалізація Монте-Карло (MCL) алгоритм оцінки пози широко використовується в області оцінки пози. AMCL (Adaptive Monte Carlo

Localization) можна розглядати як вдосконалений варіант оцінки пози Монте-Карло, що покращує продуктивність в реальному часі за рахунок скорочення часу виконання при меншій кількості вибірки в алгоритмі оцінки пози Монте-Карло. Отже, давайте розглянемо основний алгоритм оцінки пози Монте-Карло (MCL).

Кінцевою метою оцінки пози Монте-Карло (MCL) є визначення, де знаходиться робот, розташований в заданому середовищі. Тобто ми повинні отримати на карті x, y та θ робота. Для цього MCL обчислює ймовірність того, що робот може бути розташований. По-перше, позицію і орієнтацію (x, y, θ) робота в момент часу t позначають xt та отриману інформацію про відстань від датчика відстані до часу t позначається $z_{0...t} = \{z_0, z_1, ..., z_t\}$, і рух інформації, отриманої від кодера до часу t, становить $u_{0...t} = \{u_0, u_1, ..., u_t\}$. Тоді ми можемо обчислити переконання (задня ймовірність з використанням формули байєсівського оновлення) з наступним рівнянням.

$$bel(x_t) = p(x_t | z_{0...t}, u_{0...t}) \qquad \text{(Equation 11–11)}$$

Оскільки робот може мати апаратні помилки, встановіть модель датчика та рух модель. Потім обробіть передбачення та оновлення фільтра Байєса наступним чином.

На етапі прогнозування обчислюється положення $bel'(x_t)$ робота в наступний часовий проміжок за допомогою моделі руху

$p(x_t \mid x_{t-1}, u_{t-1})$ робота і вірогідність $bel(x_{t-1})$ на попереднє положення та інформацію про рух u, отриману від кодера.

$$bel'(x_t) = \int p(x_t \mid x_{t-1}, u_{t-1}) bel(x_{t-1}) dx_{t-1} \qquad \text{(Equation 11–12)}$$

Далі наведено крок оновлення. Цього разу модель датчика $p(z_t \mid x_t)$, ймовірність $bel(x_t)$ та константа нормування eta (η_t) використовується для отримання більш точної ймовірності $bel'(x_t)$ на основі інформації про датчик.

$$bel(x_t) = \eta_t p(z_t \mid x_t) bel'(x_t) \qquad \text{(Equation 11–13)}$$

Далі наводиться процедура оцінки положення шляхом генерування N частинок за допомогою фільтра частинок, використовуючи обчислену імовірність $bel(x_t)$ поточного положення. Будь ласка, зверніться до опису фільтра частинок у розділі 11.4 Теорія SLAM. У MCL використовується термін "зразок" замість частинки, і проходить через SIR (зважування важливості повторного відбору проб) процес. Перший - це процес відбору проб. Тут новий набір зразків $x_t^{'}$ витягується за допомогою моделі руху робота $p(x_t \mid x_{t-1}, u_{t-1})$ з імовірністю $bel(x_{t-1})$ попередньої позиції. Зразок "i" $x_t^{'(i)}$ серед набору вибірки $x_t^{'}$, інформація про відстань z_t, та використовується константа нормування η щоб отримати вагу $\omega_t^{(i)}$.

$$\omega_t^{(i)} = \eta p(z_t \mid x_t^{'(i)}) \qquad \text{(Equation 11–14)}$$

Нарешті, в процесі передискретизації ми створюємо N зразків нового X_t набору для вибірки (частинок) з використанням зразка $x_t^{'(i)}$ і ваги $\omega_t^{(i)}$.

$$X_t = \{x_t^{(j)} \mid j = 1 \dots N\} \sim \{x_t^{'(i)},\ \omega_t^{(i)}\} \qquad \text{(Equation 11–15)}$$

Повторюючи вищезазначений процес SIR під час переміщення частинок, передбачуванння положення робота збільшується в точності. Наприклад, як показано на рис. 174, ми можемо побачити процес збіжного місця розташування з «t1» на «t4». Весь цей процес був названий «ймовірнісною робототехнікою» Професора Себастьяна Трун, який називають підручником імовірнісного поля в робототехніці. Якщо дозволяє час, я рекомендую вам поглянути на цю книгу.

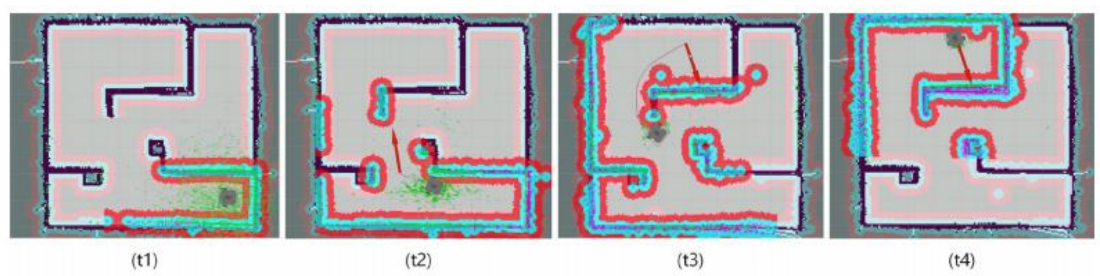


Рис. 174 Процес AMCL для оцінки пози роботів (AMCL process for robot pose estimation)

Dynamic Window Approach (DWA) - популярний метод планування уникнення перешкод та уникання перешкод. Це метод

вибору швидкості, яка може швидко досягти цільової точки уникаючи перешкод, які можуть зіткнутися з роботом у просторі швидкісного пошуку. В ROS, планувальник траєкторії використовувався для місцевого планування шляху, але DWA замінюється через його чудову продуктивність.

По-перше, робот знаходиться не в координатах осей x та y, а в просторі пошуку швидкості за допомогою поступальна швидкість v і швидкість обертання ω як осі, як показано на рис. 175. У цьому просторі, робот має максимально допустиму швидкість через апаратні обмеження, і це називається Динамічне вікно.

v: Translational velocity (meter/sec)
ω: Rotational velocity (radian/sec)
V_s: Maximum velocity area
V_a: Permissible velocity area
V_c: Current velocity
V_r: Speed area in Dynamic Window
a_{max}: Maximum acceleration / deceleration rate
$G_{(v,\omega)} = \sigma(\alpha \cdot heading(v,\omega) + \beta \cdot dist(v,\omega) + \gamma \cdot velocity(v,\omega))$: Objective function
$heading(v,\omega)$: 180 - (difference between the direction of the robot and the direction of the target point)
$dist(v,\omega)$: Distance to the obstacle
$velocity(v,\omega)$: Selected velocity
α, β, γ: Weighting constant
σ(x): Smooth Function

цьому динамічному вікні цільова функція $G(v, \omega)$ використовується для обчислення поступальної швидкості v і швидкість обертання ω, яка максимізує цільову функцію, яка розглядає напрямок, швидкість і зіткнення робота. Якщо побудувати

її, ми можемо знайти оптимальну швидкість серед різних варіантів v та ω до пункту призначення, як показано на рис. 176.

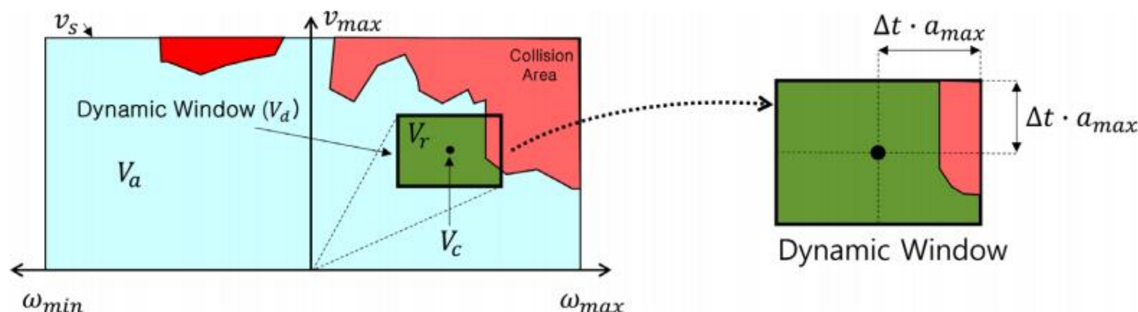


Рис. 175 Простір пошуку швидкості робота та вікно Dynomixel (Robot's velocity search space and dynamixel window)

На цьому завершується вправа, застосування та теорія SLAM в навігації. Хоча це в основному описували як мобільну робот-платформу з TurtleBot3, те саме можна застосувати до іншого робота. Не соромтеся застосовувати його до іншого робота або розробити власного.

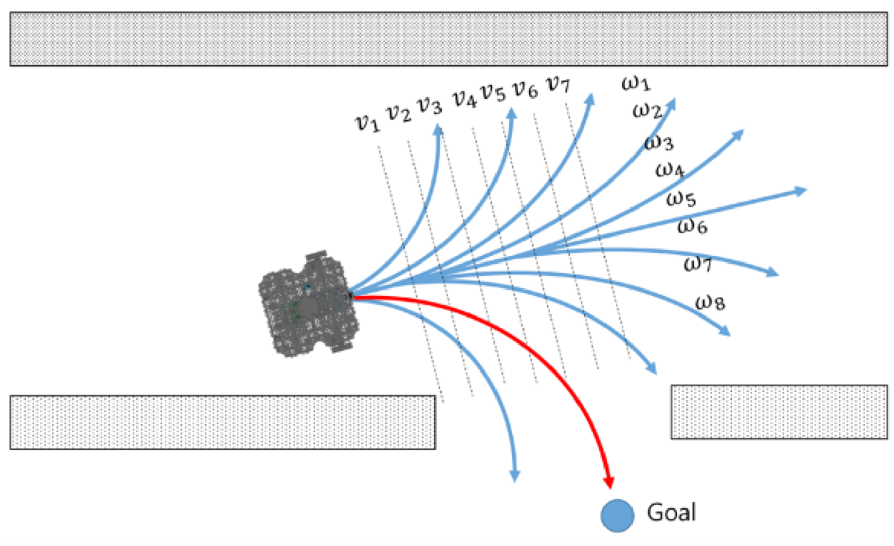


Рис. 176 Швидкість перекладу v та швидкість обертання ω

Розділ 12. Робот служби доставки

SLAM і навігація застосовуються в різних середовищах, таких як робототехніка для автомобілів, фабрики та виробничі лінії та сервісні роботи для перевезення та доставки предметів. Його технологія так швидко покращується. У розділах 10 та 11 висвітлено SLAM та навігаційні пакети. У главі 12, давайте побудуємо реального робочого робота сервісу на основі знань, які ми отримали в попередніх розділах.

12.1. Конфігурація робота служби доставки

Дизайн системи робота служби доставки можна розділити на «Сервісне ядро», «Сервісне обслуговування» Master 'та' Service Slave ', як показано на рис. 177.

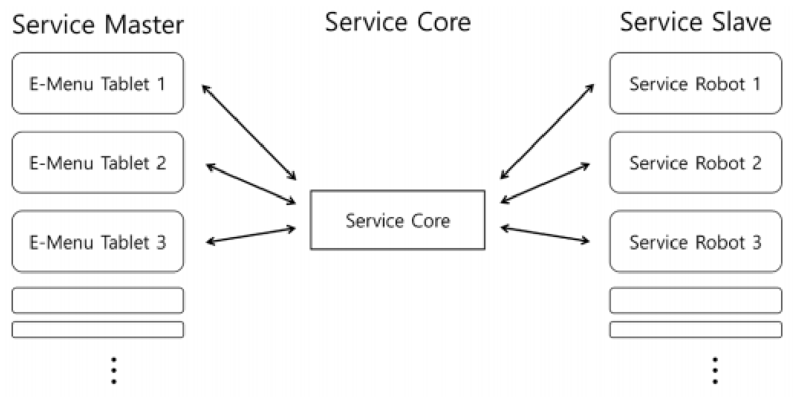


Рис. 177 Приклад проектування системи робота -служби доставки

Сервісне ядро

Сервісне ядро - це різновид бази даних, яка управляє статусом замовлення клієнтів та послугою статус роботи робота. Отримавши замовлення від замовника, воно відіграє ключову роль у обробці замовлення та планування процесу обслуговування робота. Кожен конкретний порядок і система обробки унікальна і впливає на замовлення інших клієнтів, отже, на кожне ядро послуги має існувати як одне ядро.

Майстер служби

Майстер служби отримує замовлення клієнта та передає деталі замовлення ядру служби. Він також відображає список товарів, які можна замовити, та повідомляє службу про стан послуги замовника. Щоб це зробив майстер служби, база даних сервісного ядра повинна бути синхронізована.

Service Slave

Платформа робота та фізичний об'єкт, який обробляє замовлення, ведений сервіс оновлює статус обслуговування замовлення до сервісного ядра в режимі реального часу.

Головний сервіс, ведений сервіс та сервісне ядро можуть бути лише одним комп'ютером або стільки, скільки розподілено на одинкомп'ютер. Однак один комп'ютер може бути не в змозі обробити всю роботу, коли весь процес виконується на одному комп'ютері, а з іншого - сильний розподіл на кожному комп'ютері може призвести до перевантаження бездротового зв'язку. Таким чином, дбайливий розподіл є критичним для правильної обробки. Крім того, оскільки ROS 1.x - це система, створена для керувати одним роботом, щоб використовувати ROS на декількох комп'ютерах, має бути наступна команда для зменшення різниці в часі між кожним комп'ютером.

```
$ sudo ntpdate ntp.ubuntu.com
```

Наприклад, як на рис. 178, я використовував наступні пристрої для реалізації служби доставки робота системи.

■ Сервісне ядро: NUC i5 (3 комп'ютери для виконання роскорду та сервісного ядра)

■ Сервісний майстер: 3 SAMSUNG ПРИМІТКА 10.4 ОС Android

■ Службовий ведений: 3 Intel Joule 570x з TurtleBot3 Carrier (waffle TurtleBot3, налаштовані для робота доставки)

Рис. 178 Фактичний образ роботів служби доставки та їх операційних систем

Як тільки загальна концепція буде завершена, вам доведеться вирішити, якими повинні бути повідомлення, які обмінюються між вузлами в кожній області, як на рис. 179.

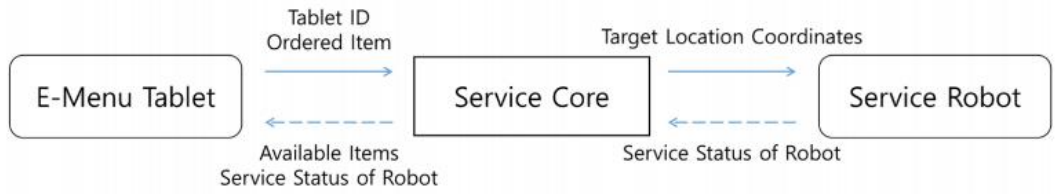


Рис. 179 Приклад повідомлень у кожній області, що передаються роботом служби доставки

Незважаючи на те, що один майданчик може наглядати за різними роботами, для спрощення системної структури - одна площадка на робота було виконано пару для виконання служби.

Колодка, яка використовується для розміщення замовлення, повинна надіслати інформація про ідентифікатор колодки та вибрані елементи до сервісного ядра. На основі ідентифікатора колодки, сервісне ядро визначає, якому роботові призначити виконання роботи, і передає відповідний цільовий сайт для замовленого товару для виконання послуги.

На основі алгоритму планування шляху службовий робот переміщається на отриману цільову ділянку. Виконуючи цю місію, робот передає свій службовий статус, включаючи зіткнення з перешкодою, збій маршруту, прибуття цілі та незалежно від того, чи замовлений товар набувається. Ядро сервісу обробляє таку інформацію, як статус на дублікаті замовлення, що стосуються замовлених предметів, замовлення, отримані від робота, поки робот знаходиться на службі показу замовнику через майданчик. Кожна інформація, надіслана та отримана під час цього процесу використовує спеціальні нові файли повідомлень або файли srv.

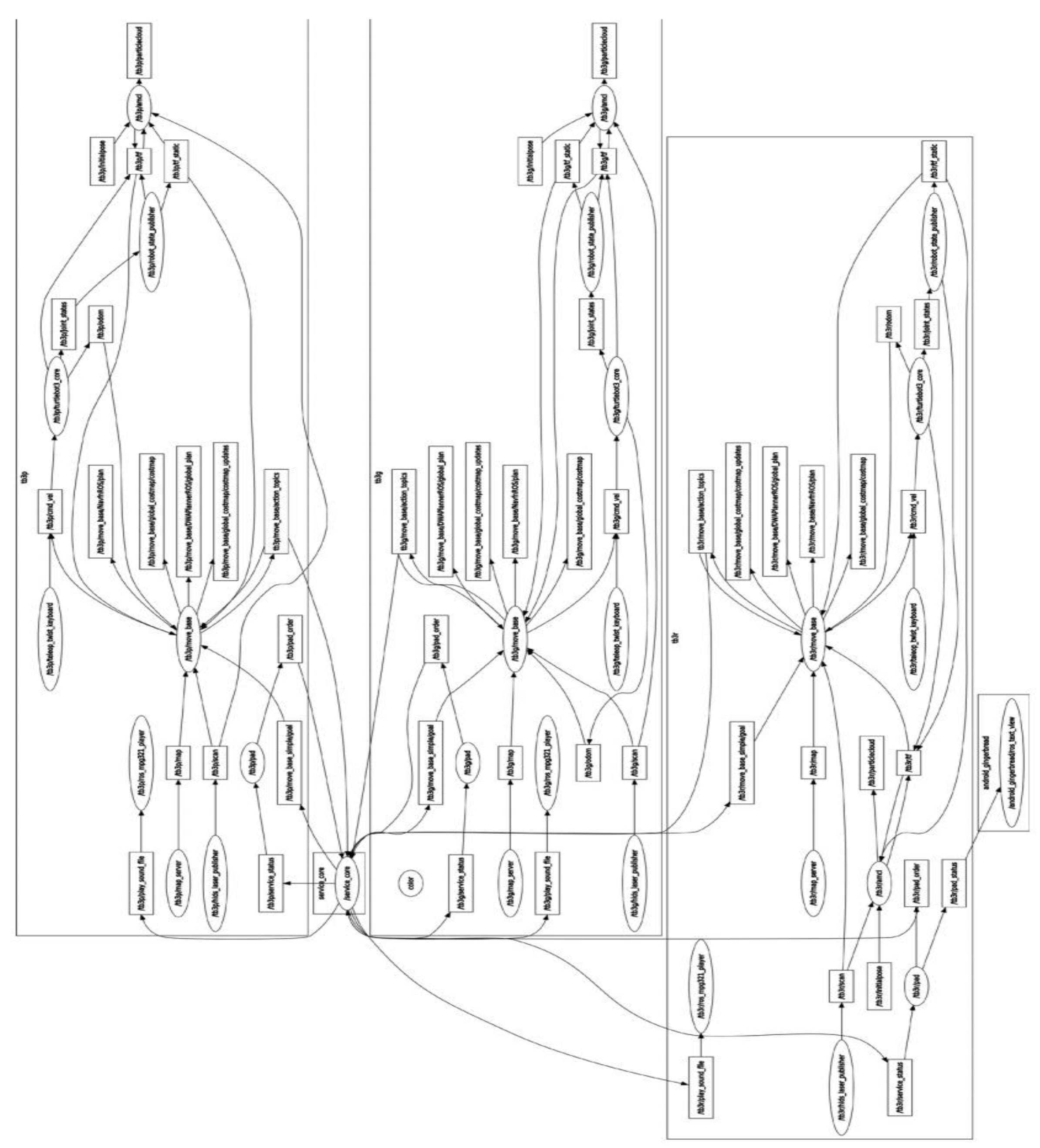


Рис.180 Приклад структури вузла робота служби доставки

Перш ніж вдаватися до деталей про кожен вузол робота, давайте розберемось із загальною концепцією спочатку. На рис. 180 показано співвідношення всіх вузлів та тем служби доставки для побудови в главі 12. Графік складається з 3 груп та ‘service_core’. Сервісні вузли, пов’язані з роботами, поєднуються з просторами імен „tb3p“, „tb3g“, „tb3r“ у кожній групі. "Service_core" повинен контролювати три групи і тому не повинно залежати від будь-якої групи. Якщо ви хочете налаштувати систему, яка дозволяє за допомогою однієї подушки керувати групою роботів, а не спарювати кожного робота до однієї подушечки, вам потрібно створити окремі групи для кожної колодки та кожен робот-сервіс під окремим простір імен.

Причина, по якій групується простір імен1 використовується при розробці послуги системи робота полягає в тому, що коли більшість роботів або комп'ютерів хочуть використовувати один тип ROS-пакету, та водночас неможливо виконати дубльовані назви вузлів і тем, зареєстрованих в ROS master.

Серед тем, згрупованих у три простори імен, рис. 181 показує графік вузла, на якому фокусується теми, які безпосередньо *‘service_core’* відправляє і отримує. *‘service_core’* отримує замовлення через тема під назвою *‘/ pad_order’* та отримує пошук шляху робота і успішного досягнення призначення через *‘/ move_base/action_topics’*. Це також надає статус послуги через тему *«/ service_status»,* проходить

розташування записаного голосовго файлу через *«/ play_sound_file»* та координує цільову точку робота через *'/ move_base_simple / goal'*.

Рис. 182 - блок-схема підготовки служби доставки. Навігація використовує карту зроблену за допомогою SLAM. При подачі продумайте, яке місце використовувати як місце обслуговування. Посаду слід ідентифікувати та записати як значення координат на створеній карті, так і значення координат будуть використовуватися для збору параметрів ROS та конфігурації в 'service_core' для подальшого опису. Для мого прикладу мені потрібні координатні значення місцезнаходження для розміщення замовника та розташування для робота, щоб отримати замовлений предмет. Див. Розділи 10 та 11 або подробиці щодо SLAM та навігації.

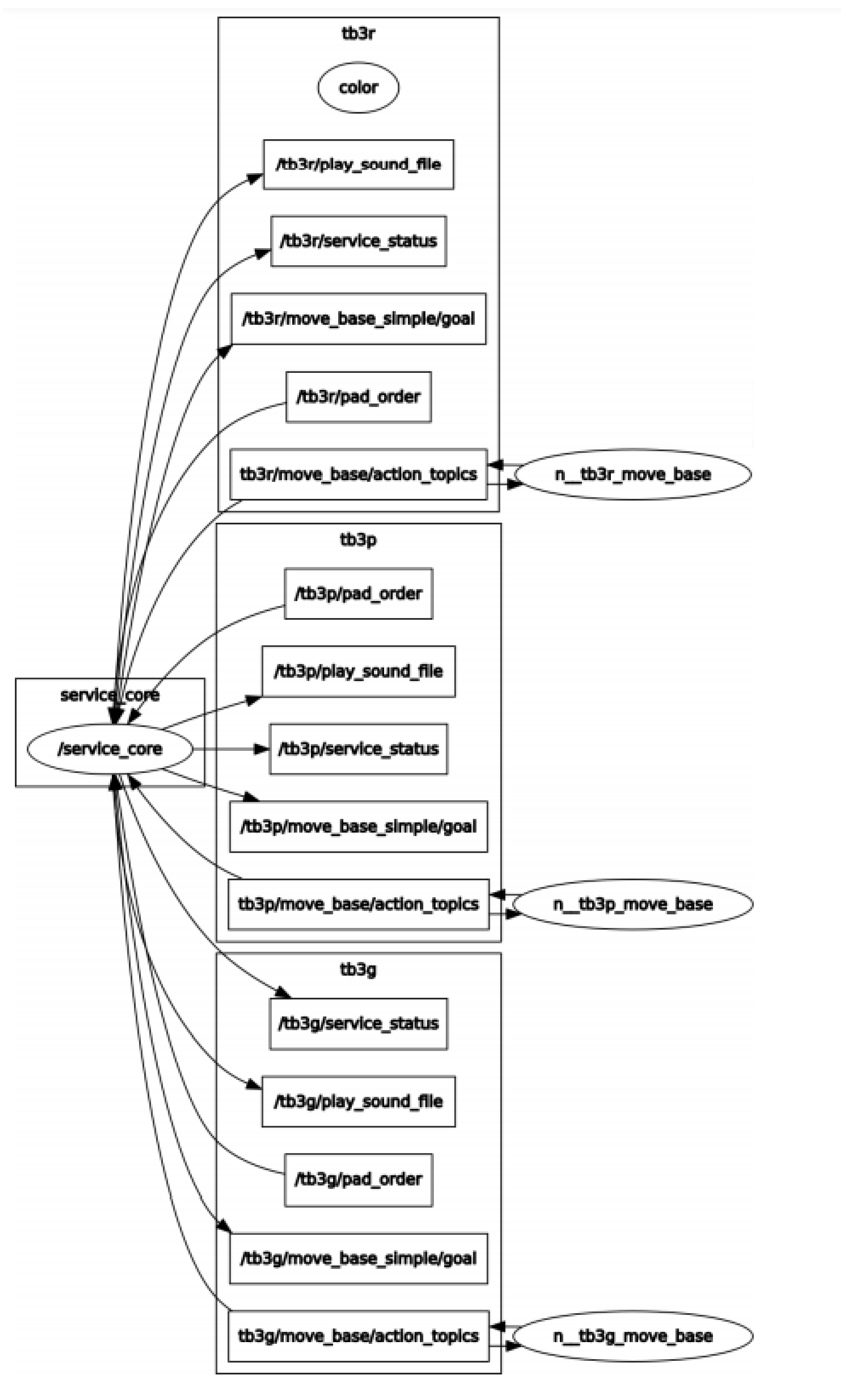

tb3r
color
/tb3r/play_sound_file
/tb3r/service_status
/tb3r/move_base_simple/goal
/tb3r/pad_order
tb3r/move_base/action_topics
n__tb3r_move_base
tb3p
/tb3p/pad_order
/tb3p/play_sound_file
service_core
/service_core
/tb3p/service_status
/tb3p/move_base_simple/goal
tb3p/move_base/action_topics
n__tb3p_move_base
tb3g
/tb3g/service_status
/tb3g/play_sound_file
/tb3g/pad_order
/tb3g/move_base_simple/goal
tb3g/move_base/action_topics
n__tb3g_move_base

Рис. 181 Список тем, які вузол service_core надсилає та отримує

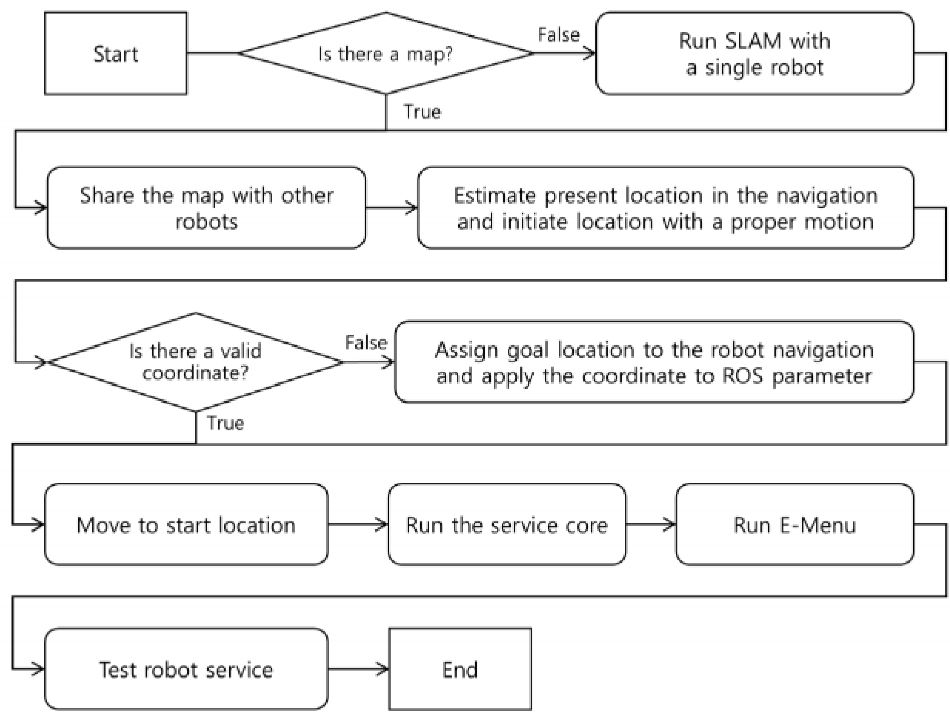


Рис. 182 Блок -схема підготовки служб доставки

Вихідний код, необхідний для створення робота служби доставки, описаний тут та є доступним за адресою нижче. Перейдемо до конфігурації та джерел вузлів.

https://github.com/ROBOTIS-GIT/turtlebot3_deliver

На рис. 183 показано основну структуру джерела сервісного ядра. Цей вузол починається з *main ()* і спочатку встановлює параметр *ROS* через f*nInitParam ()*. Згодом отримує замовлення від клієнтів через функції *pad* та *cbReceivePadOrder ()* та *cbCheckArrivalStatus (),* які отримують дані про статус прибуття цільової позиції робота,

оголошуються для виконання як тема передплати та через функції *fnPubServiceStatus (), fnPubPose (),* стан служби та цільові позиції значень координат робота. Цей процес виконується в нескінченний цикл, доки не буде натиснута клавіша [Ctrl + c].

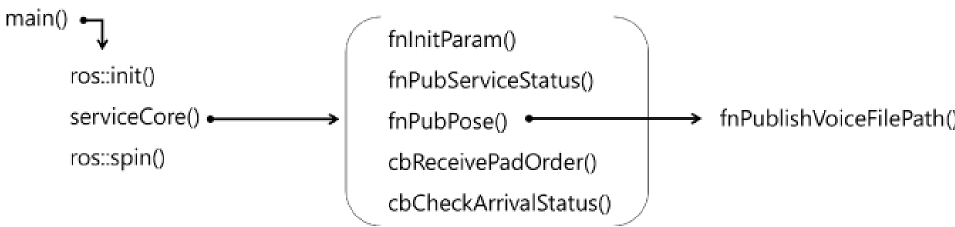


Рис. 183 Основна структура ядра обслуговування

При виконанні функції *service Core ()* викликається функція *fn Init Param ()* і визначаються видавець і передплатник для передачі і прийому різних даних. *'service_core'* повинен обробляти теми для трьох роботів, трьох майданчиків. Таким чином, визначено три типи видавця і передплатника. Кожна з їхніх робіт описана нижче.

Таблиця 26

Функція ServiceCore () у service_core.cpp

Елемент	**Опис**
pubServiceStatusPad	Видавець, який публікує статус служби доставки в pad
pubPlaySound	Видавець, який публікує розташування записаних голосових файлів

subPadOrder	Абоненти, які підписуються на замовлення клієнтів з колодки
subArrivalStatus	Абонент, який передплачує прибуття робота

```
---------------------------------- /turtlebot3_carrier/src/service_core.cpp
(parts of excerpt)
ServiceCore()
{
 fnInitParam();
 pubServiceStatusPadTb3p =
nh_.advertise<turtlebot3_carrier::ServiceStatus>("/tb3p/
service_status", 1);
 pubServiceStatusPadTb3g =
nh_.advertise<turtlebot3_carrier::ServiceStatus>("/tb3g/
service_status", 1);
 pubServiceStatusPadTb3r =
nh_.advertise<turtlebot3_carrier::ServiceStatus>("/tb3r/
service_status", 1);
 pubPlaySoundTb3p =
nh_.advertise<std_msgs::String>("/tb3p/play_sound_file", 1);
 pubPlaySoundTb3g =
nh_.advertise<std_msgs::String>("/tb3g/play_sound_file", 1);
```

```
 pubPlaySoundTb3r =
nh_.advertise<std_msgs::String>("/tb3r/play_sound_file", 1);
 pubPoseStampedTb3p =
nh_.advertise<geometry_msgs::PoseStamped>("/tb3p/move_base_simpl
e/goal", 1);
 pubPoseStampedTb3g =
nh_.advertise<geometry_msgs::PoseStamped>("/tb3g/move_base_simpl
e/goal", 1);
 pubPoseStampedTb3r =
nh_.advertise<geometry_msgs::PoseStamped>("/tb3r/move_base_simpl
e/goal", 1);
 subPadOrderTb3p = nh_.subscribe("/tb3p/pad_order", 1,
&ServiceCore::cbReceivePadOrder, this);
 subPadOrderTb3g = nh_.subscribe("/tb3g/pad_order", 1,
&ServiceCore::cbReceivePadOrder, this);
 subPadOrderTb3r = nh_.subscribe("/tb3r/pad_order", 1,
&ServiceCore::cbReceivePadOrder, this);
 subArrivalStatusTb3p = nh_.subscribe("/tb3p/move_base/result", 1,
&ServiceCore::cbCheckArrivalStatusTB3P, this);
 subArrivalStatusTb3g = nh_.subscribe("/tb3g/move_base/result", 1,
&ServiceCore::cbCheckArrivalStatusTB3G, this);
 subArrivalStatusTb3r = nh_.subscribe("/tb3r/move_base/result", 1,
&ServiceCore::cbCheckArrivalStatusTB3R, this);
```

```
ros::Rate loop_rate(5);
while (ros::ok())
{
fnPubServiceStatus();
fnPubPose();
ros::spinOnce();
loop_rate.sleep();
}
}
-------------------------------------------------------------------------------------
-------
```

Функція *Fn Init Param ()* отримує дані про цільову позу (положення + орієнтація) робота на карті з заданого файлу параметрів. У цьому прикладі робот повинен мати можливість переміщатися на три позиції, де клієнти можуть розмістити замовлення, а також мати можливість переміщатися на три позиції, де замовлені товари можуть бути придбані, тому необхідно зберігати інформацію про шість координат місць розташування на карті. Структура функції *Fn Init Param* () виглядає наступним чином:

Таблиця 27

Функція fnInitParam () у service_core.cpp

Елемент	**Опис**
poseStampedTable	Координати, де клієнт розміщує і отримує замовлення
poseStampedCounter	Координати місця завантаження товару на робота

```
------------------------------------- /turtlebot3_carrier/src/service_core.cpp
(parts of excerpts)
void fnInitParam()
{
 nh_.getParam("table_pose_tb3p/position", target_pose_position);
 nh_.getParam("table_pose_tb3p/orientation", target_pose_orientation);
 poseStampedTable[0].header.frame_id = "map";
 poseStampedTable[0].header.stamp = ros::Time::now();
 poseStampedTable[0].pose.position.x = target_pose_position[0];
 poseStampedTable[0].pose.position.y = target_pose_position[1];
 poseStampedTable[0].pose.position.z = target_pose_position[2];
 poseStampedTable[0].pose.orientation.x = target_pose_orientation[0];
 poseStampedTable[0].pose.orientation.y = target_pose_orientation[1];
 poseStampedTable[0].pose.orientation.z = target_pose_orientation[2];
 poseStampedTable[0].pose.orientation.w = target_pose_orientation[3];
```

```
nh_.getParam("table_pose_tb3g/position", target_pose_position);
nh_.getParam("table_pose_tb3g/orientation", target_pose_orientation);
poseStampedTable[1].header.frame_id = "map";
poseStampedTable[1].header.stamp = ros::Time::now();
poseStampedTable[1].pose.position.x = target_pose_position[0];
poseStampedTable[1].pose.position.y = target_pose_position[1];
poseStampedTable[1].pose.position.z = target_pose_position[2];
poseStampedTable[1].pose.orientation.x = target_pose_orientation[0];
poseStampedTable[1].pose.orientation.y = target_pose_orientation[1];
poseStampedTable[1].pose.orientation.z = target_pose_orientation[2];
poseStampedTable[1].pose.orientation.w = target_pose_orientation[3];
nh_.getParam("table_pose_tb3r/position", target_pose_position);
nh_.getParam("table_pose_tb3r/orientation", target_pose_orientation);
poseStampedTable[2].header.frame_id = "map";
poseStampedTable[2].header.stamp = ros::Time::now();
poseStampedTable[2].pose.position.x = target_pose_position[0];
poseStampedTable[2].pose.position.y = target_pose_position[1];
poseStampedTable[2].pose.position.z = target_pose_position[2];
poseStampedTable[2].pose.orientation.x = target_pose_orientation[0];
poseStampedTable[2].pose.orientation.y = target_pose_orientation[1];
poseStampedTable[2].pose.orientation.z = target_pose_orientation[2];
poseStampedTable[2].pose.orientation.w = target_pose_orientation[3];
nh_.getParam("counter_pose_bread/position", target_pose_position);
```

```
 nh_.getParam("counter_pose_bread/orientation",
target_pose_orientation);
 poseStampedCounter[0].header.frame_id = "map";
 poseStampedCounter[0].header.stamp = ros::Time::now();
 poseStampedCounter[0].pose.position.x = target_pose_position[0];
 poseStampedCounter[0].pose.position.y = target_pose_position[1];
 poseStampedCounter[0].pose.position.z = target_pose_position[2];
 poseStampedCounter[0].pose.orientation.x =
target_pose_orientation[0];
 poseStampedCounter[0].pose.orientation.y =
target_pose_orientation[1];
 poseStampedCounter[0].pose.orientation.z = target_pose_orientation[2];
 poseStampedCounter[0].pose.orientation.w =
target_pose_orientation[3];
 nh_.getParam("counter_pose_drink/position", target_pose_position);
 nh_.getParam("counter_pose_drink/orientation",
target_pose_orientation);
 poseStampedCounter[1].header.frame_id = "map";
 poseStampedCounter[1].header.stamp = ros::Time::now();
 poseStampedCounter[1].pose.position.x = target_pose_position[0];
 poseStampedCounter[1].pose.position.y = target_pose_position[1];
 poseStampedCounter[1].pose.position.z = target_pose_position[2];
```

```
 poseStampedCounter[1].pose.orientation.x =
target_pose_orientation[0];
 poseStampedCounter[1].pose.orientation.y =
target_pose_orientation[1];
 poseStampedCounter[1].pose.orientation.z = target_pose_orientation[2];
 poseStampedCounter[1].pose.orientation.w =
target_pose_orientation[3];
 nh_.getParam("counter_pose_snack/position", target_pose_position);
 nh_.getParam("counter_pose_snack/orientation",
target_pose_orientation);
 poseStampedCounter[2].header.frame_id = "map";
 poseStampedCounter[2].header.stamp = ros::Time::now();
 poseStampedCounter[2].pose.position.x = target_pose_position[0];
 poseStampedCounter[2].pose.position.y = target_pose_position[1];
 poseStampedCounter[2].pose.position.z = target_pose_position[2];
 poseStampedCounter[2].pose.orientation.x =
target_pose_orientation[0];
 poseStampedCounter[2].pose.orientation.y =
target_pose_orientation[1];
 poseStampedCounter[2].pose.orientation.z = target_pose_orientation[2];
 poseStampedCounter[2].pose.orientation.w =
target_pose_orientation[3];
}
```

```
--------------------------------------------------------------------------------
-
```

Значення параметрів, зазначених у файлі *target_pose.yaml*, є координатами на карті, необхідними роботу для виконання своїх послуг.

Існують різні способи отримання значень координат, і найпростіший спосіб-використовувати команду "*rostopic echo*" для отримання значення пози під час навігації. Однак ці координати змінюються з кожним відображенням через SLAM, тому уникайте Переписування карти під час виконання фактичної навігації, щоб зменшити позиційні зміни об'єктів і перешкод, щоб ви могли продовжувати працювати з тією ж картою.

```
----------------------------------------------
/turtlebot3_carrier/param/target_pose.yaml
table_pose_tb3p:
 position: [-0.338746577501, -0.85418510437, 0.0]
 orientation: [0.0, 0.0, -0.0663151963596, 0.997798724559]
table_pose_tb3g:
 position: [-0.168751597404, -0.19147400558, 0.0]
 orientation: [0.0, 0.0, -0.0466624033917, 0.998910716786]
table_pose_tb3r:
```

```
 position: [-0.251043587923, 0.421476781368, 0.0]
 orientation: [0.0, 0.0, -0.0600887022438, 0.998193041382]
counter_pose_bread:
 position: [-3.60783815384, -0.750428497791, 0.0]
 orientation: [0.0, 0.0, 0.999335763287, -0.0364421763375]
counter_pose_drink:
 position: [-3.48697376251, -0.173366710544, 0.0]
 orientation: [0.0, 0.0, 0.998398746904, -0.0565680314445]
counter_pose_snack:
 position: [-3.62247490883, 0.39046728611, 0.0]
 orientation: [0.0, 0.0, 0.998908838216, -0.0467026009308]
------------------------------------------------------------------------------------------
-----------------
```

Далі, в залежності від того, в якій сервісній процедурі знаходиться поточний робот, функція *fnPubPose()* використовується для встановлення наступної цільової позиції, коли робот досягає цільової позиції. Коли робот завершує обслуговування, всі параметри скидаються.

Таблиця 28

Функції fnPubPose () у service_core.cpp

Елемент	**Опис**
is_robot_reached_target	Чи досяг робот своєї навігаційної мети

is_item_available	Наявність предметів
item_num_chosen_by_pad	Артикул замовлення
robot_service_sequence	Процес, який виконує робот:
	0-очікування замовлення клієнта
	1-відразу після отримання замовлення клієнта
	2-збираюся отримати замовлений товар
	3-Завантаження замовлених товарів
	4-переміщення до місця знаходження клієнта
	5-Доставка товарів клієнту
fnPublishVoiceFilePath()	Функція публікації шляху до записаного голосового файлу
ROBOT_NUMBER	Номер робота (визначається користувачем)

```
---------------------------------- /turtlebot3_carrier/src/service_core.cpp
(parts of excerpt)
void fnPubPose()
{
 if (is_robot_reached_target[ROBOT_NUMBER])
 {
 if (robot_service_sequence[ROBOT_NUMBER] == 1)
```

```
{
fnPublishVoiceFilePath(ROBOT_NUMBER, "~/voice/voice1-2.mp3");
robot_service_sequence[ROBOT_NUMBER] = 2;
}
else if (robot_service_sequence[ROBOT_NUMBER] == 2)
{

pubPoseStampedTb3p.publish(poseStampedCounter[item_num_chosen
_by_pad[ROBOT_NUMBER]]);
is_robot_reached_target[ROBOT_NUMBER] = false;
robot_service_sequence[ROBOT_NUMBER] = 3;
}
else if (robot_service_sequence[ROBOT_NUMBER] == 3)
{
fnPublishVoiceFilePath(ROBOT_NUMBER, "~/voice/voice1-3.mp3");
robot_service_sequence[ROBOT_NUMBER] = 4;
}
else if (robot_service_sequence[ROBOT_NUMBER] == 4)
{

pubPoseStampedTb3p.publish(poseStampedTable[ROBOT_NUMBER]
);
is_robot_reached_target[ROBOT_NUMBER] = false;
```

```
robot_service_sequence[ROBOT_NUMBER] = 5;
}
else if (robot_service_sequence[ROBOT_NUMBER] == 5)
{
fnPublishVoiceFilePath(ROBOT_NUMBER, "~/voice/voice1-4.mp3");
robot_service_sequence[ROBOT_NUMBER] = 0;
is_item_available[item_num_chosen_by_pad[ROBOT_NUMBER]] =
1;
item_num_chosen_by_pad[ROBOT_NUMBER] = -1;
}
}
}
… omitted …
--------------------------------------------------------------------------------------
-----
```

Функція *cbReceivePadOrder()* отримує номер майданчика, використовуваного в момент замовлення, і номер товару замовленого товару, щоб визначити, чи доступна послуга.

Якщо служба доступна, то для запуску служби *"robot_service_sequence'* встановлюється значення "1". Вихідний код цієї функції виглядає наступним чином.

Таблиця 29

Функція cbReceivePadOrder () у service_core.cpp

Елемент	**Опис**
pad_number	Номер майданчика, замовленої у (Номер робота для обслуговування)
item_number	Артикул замовлення

```
----------------------------------- /turtlebot3_carrier/src/service_core.cpp
(parts of excerpt)
void cbReceivePadOrder(const turtlebot3_carrier::PadOrder padOrder)
{
 int pad_number = padOrder.pad_number;
 int item_number = padOrder.item_number;
 if (is_item_available[item_number] != 1)
 {
 ROS_INFO("Chosen item is currently unavailable");
 return;
 }
 if (robot_service_sequence[pad_number] != 0)
 {
 ROS_INFO("Your TurtleBot is currently on servicing");
 return;
 }
```

```
 if (item_num_chosen_by_pad[pad_number] != -1)
 {
 ROS_INFO("Your TurtleBot is currently on servicing");
 return;
 }
 item_num_chosen_by_pad[pad_number] = item_number;
 robot_service_sequence[pad_number] = 1; // just left from the table
 is_item_available[item_number] = 0;
}
--------------------------------------------------------------------------------
```

Функція *cbCheckArrivalStatus(),* показана нижче, перевіряє рухомий стан робота підписувача. Блок коду *" else* " обробляє, якщо робот стикається з перешкодою під час своєї навігації або не може знайти свій шлях в алгоритмі.

```
--------------------------- /turtlebot3_carrier/src/service_core.cpp (parts of excerpt)
void cbCheckArrivalStatusTB3P(const move_base_msgs::MoveBaseActionResult rcvMoveBaseActionResult)
{
 if (rcvMoveBaseActionResult.status.status == 3)
 {
```

```
 is_robot_reached_target[ROBOT_NUMBER_TB3P] = true;
 }
 else
 {
 …omitted…
 }
}
void cbCheckArrivalStatusTB3G(const move_base_msgs::MoveBaseActionResult rcvMoveBaseActionResult)
{
…omitted…
}
void cbCheckArrivalStatusTB3R(const move_base_msgs::MoveBaseActionResult rcvMoveBaseActionResult)
{
…omitted…
}
--------------------------------------------------------------------------------------
-
```

Функція *fnPublishVoicePath()*, показана в наступному вихідному коді, публікує місце розташування попередньо записаного

голосового файлу у вигляді рядка. Для того щоб дійсно відтворити голос на ROS, вам знадобиться додатковий пакет ROS.

```
---------------------- /turtlebot3_carrier/src/service_core.cpp (parts of excerpt)
void fnPublishVoiceFilePath(int robot_num, const char* file_path)
 {
 std_msgs::String str;
 str.data = file_path;
 if (robot_num == ROBOT_NUMBER_TB3P)
 {
 pubPlaySoundTb3p.publish(str);
 }
 else if (robot_num == ROBOT_NUMBER_TB3G)
 {
 pubPlaySoundTb3g.publish(str);
 }
 else if (robot_num == ROBOT_NUMBER_TB3R)
 {
 pubPlaySoundTb3r.publish(str);
 }
 }
```

```
-----------------------------------------------------------------------------------------
-------
```

У цьому прикладі вузол service master працює на планшетному комп'ютері з ОС Android. Однак вузли також можуть бути запрограмовані на прийом замовлень через термінал. Детальніше про програмуванні ROS Java2 з використанням платформи Android OS буде розказано в наступному розділі. Джерело головного вузла служби, описаний нижче, створюється шляхом застосування "*android_tutorial_pubsub*", простий приклад публікації теми та підписки, наданого *ROS Java.*

```
---------------------------------------------------
turtlebot3_carrier_pad/ServicePad.java
package org.ros.android.android_tutorial_pubsub;
import org.ros.concurrent.CancellableLoop;
import org.ros.message.MessageListener;
import org.ros.namespace.GraphName;
import org.ros.node.AbstractNodeMain;
import org.ros.node.ConnectedNode;
import org.ros.node.topic.Publisher;
import org.ros.node.topic.Subscriber;
import javax.security.auth.SubjectDomainCombiner;
public class ServicePad extends AbstractNodeMain {
```

```
private String pub_pad_order_topic_name;
private String sub_service_status_topic_name;
private String pub_pad_status_topic_name;
private int robot_num = 0;
private int selected_item_num = -1;
private boolean jump = false;
private int[] item_num_chosen_by_pad = {-1, -1, -1};
private int[] is_item_available = {1, 1, 1};
private int[] robot_service_sequence = {0, 0, 0};
public boolean[] button_pressed = {false, false, false};
public ServicePad() {
this.pub_pad_order_topic_name = "/tb3g/pad_order";
this.sub_service_status_topic_name = "/tb3g/service_status";
this.pub_pad_status_topic_name = "/tb3g/pad_status";
}
public GraphName getDefaultNodeName() {
return GraphName.of("tb3g/pad");
}
public void onStart(ConnectedNode connectedNode) {
final Publisher pub_pad_order =
connectedNode.newPublisher(this.pub_pad_order_topic_name,
"turtlebot3_carrier/PadOrder");
```

```
final Publisher pub_pad_status = connectedNode.newPublisher(this.
pub_pad_status_topic_name,
"std_msgs/String");
final Subscriber<turtlebot3_carrier.ServiceStatus> subscriber =
connectedNode.newSubscriber(this.sub_service_status_topic_name, "
turtlebot3_carrier/
ServiceStatus");
subscriber.addMessageListener(new
MessageListener<turtlebot3_carrier.SerivceStatus>() {
@Override
public void onNewMessage(turtlebot3_carrier.SerivceStatus
serviceStatus)
{
item_num_chosen_by_pad = serviceStatus.item_num_chosen_by_pad;
is_item_available = serviceStatus.is_item_available;
robot_service_sequence = serviceStatus.robot_service_sequence
}
});
connectedNode.executeCancellableLoop(new CancellableLoop() {
protected void setup() { }
protected void loop() throws InterruptedException
{
```

```
str_msgs.String padStatus =
(str_msgs.String)pub_pad_status.newMessage();
turtlebot3_carrier.PadOrder padOrder =
(turtlebot3_carrier.PadOrder)pub_pad_order.newMessage();
String str = "";
if (button_pressed[0] || button_pressed[1] || button_pressed[2])
{
jump = false;
if (button_pressed[0])
{
selected_item_num = 0;
str += "Burger was selected";
button_pressed[0] = false;
}
else if (button_pressed[1])
{
selected_item_num = 1;
str += "Coffee was selected";
button_pressed[1] = false;
}
else if (button_pressed[2])
{
selected_item_num = 2;
```

```
str += "Waffle was selected";
button_pressed[2] = false;
}
else
{
selected_item_num = -1;
str += "Sorry, selected item is now unavailable. Please choose another item.";
}
if (is_item_available[selected_item_num] != 1)
{
str += ", but chosen item is currently unavailable.";
jump = true;
}
else if (robot_service_sequence[robot_num] != 0)
{
str += ", but your TurtleBot is currently on servicing";
jump = true;
}
else if (item_num_chosen_by_pad[robot_num] != -1)
{
str += ", but your TurtleBot is currently on servicing";
jump = true;
```

```
}
padStatus.setData(str);
pub_pad_status.publish(padStatus);
if(!jump)
{
padOrder.pad_number = robot_num;
padOrder.item_number = selected_item_num;
pub_pad_order.publish(padOrder);
}
}
Thread.sleep(1000L);
}
});
}
}
-----------------------------------------------------------------------------------------------
```

У коді клас *MainActivity* отримує і використовує екземпляр класу *ServicePaid*. Крім того, це майже те ж саме, що і загальний клас MainActivity, тому ми опустимо докладне пояснення і розглянемо тільки ту частину, яка відповідає службі доставки.

Номер робота, керованого планшетним ПК, на який завантажується цей приклад, позначається як *robot_num* в якості наступного вихідного коду, а номер обраного продукту на планшетному ПК ініціалізується як"-1'.

```
---------------------------------------
private int robot_num = 0;
private int selected_item_num = -1;
---------------------------------------
```

Щоб уникнути дублювання замовлень, планшетний комп'ютер повинен бути синхронізований зі статусом замовлення, зберігаються в сервісному ядрі, до отримання замовлення клієнта. Наступний вихідний код оголошується в сервісному ядрі в тому ж форматі, що і масив, який записує статус замовлення.

```
-------------------------------------------
private int[] item_num_chosen_by_pad = {-1, -1, -1};
private int[] is_item_available = {1, 1, 1};
private int[] robot_service_sequence = {0, 0, 0};
-------------------------------------------
```

При створенні екземпляра класу *Servicebase* в класі MainActivity ім'я теми буде вказано наступним чином. У цьому прикладі кожна пара pad і robot задається як простір імен групи, тому

перед кожним ім'ям теми необхідно написати ім'я, яке використовується в цьому просторі ім'я.

```
------------------------------------------
public ServicePad() {
this.pub_pad_order_topic_name = "/tb3g/pad_order";
this.sub_service_status_topic_name          =          "/tb3g/service_status";
this.pub_pad_status_topic_name = "/tb3g/pad_status"; }
------------------------------------------
```

Вказує ім'я вузла, яке з'являється в ROS. Ім'я вузла також має бути записано разом з простором імен групи.

```
------------------------------------------
public GraphName getDefaultNodeName() {
return GraphName.of("tb3g/pad");
}
------------------------------------------
```

З сервісного ядра користувач отримує умови обслуговування, такі як номер замовленого товару, доступність вибору товару і статус обслуговування робота.

```
------------------------------------------
subscriber.addMessageListener(new
MessageListener<turtlebot3_carrier.ServiceStatus>() {
 @Override
```

```
public void onNewMessage(turtlebot3_carrier.ServiceStatus
serviceStatus)
 {
 item_num_chosen_by_pad = serviceStatus.item_num_chosen_by_pad;
 is_item_available = serviceStatus.is_item_available;
 robot_service_sequence = serviceStatus.robot_service_sequence;
 }
});
-------------------------------------------
```

Ця частина обробляє замовлення клієнта на основі загальної ситуації обслуговування, синхронізованої з ядром обслуговування. Точно так само перевіряє, чи дублюється замовлення, і, якщо послуга доступна, публікує замовлення. На рис. 184 та рис. 185 показані приклади успішних і невдалих замовлень, відповідно, коли клієнт вибирає товар, намальований на планшеті.

```
-------------------------------------------
protected void loop() throws InterruptedException
{
 std_msgs.String padStatus = (std_msgs.String)
pub_pad_status.newMessage();
 turtlebot3_carrier.PadOrder padOrder = (turtlebot3_carrier.PadOrder)
 pub_pad_order.newMessage();
 String str = "";
```

```
if (button_pressed[0] || button_pressed[1] || button_pressed[2])
{
jump = false;
if (button_pressed[0])
{
selected_item_num = 0;
str += "Burger was selected";
button_pressed[0] = false;
}
else if (button_pressed[1])
{
selected_item_num = 1;
str += "Coffee was selected";
button_pressed[1] = false;
}
else if (button_pressed[2])
{
selected_item_num = 2;
str += "Waffle was selected";
button_pressed[2] = false;
}
else
{
```

```
selected_item_num = -1;
str += "Sorry, selected item is now unavailable. Please choose another
item.";
}
if (is_item_available[selected_item_num] != 1)
{
str += ", but chosen item is currently unavailable.";
jump = true;
}
else if (robot_service_sequence[robot_num] != 0)
{
str += ", but your TurtleBot is currently on servicing";
jump = true;
}
else if (item_num_chosen_by_pad[robot_num] != -1)
{
str += ", but your TurtleBot is currently on servicing";
jump = true;
}
padStatus.setData(str);
pub_pad_status.publish(padStatus);
if(!jump)
{
```

```
 padOrder.pad_number = robot_num;
 padOrder.item_number = selected_item_num;
 pub_pad_order.publish(padOrder);
 }
 }
 Thread.sleep(1000L);
}
------------------------------------------
```

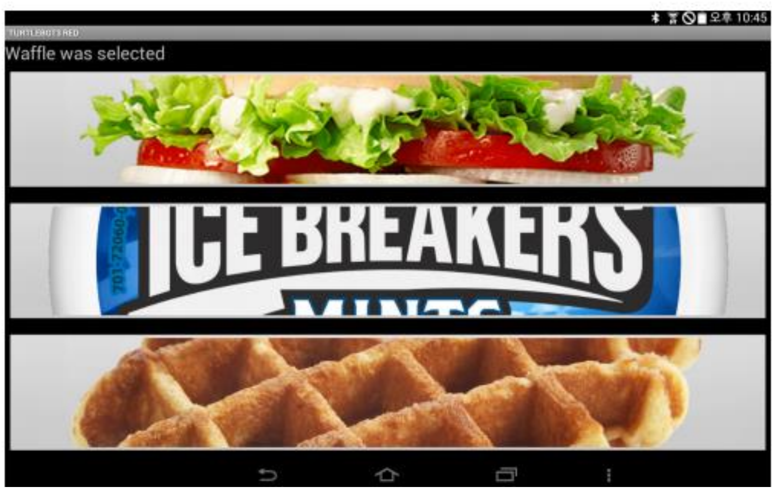


Рис. 184 Приклад 1 меню, відображеного у (коли вибрано недоступний пункт) блокноті (коли замовлення успішно отримано)

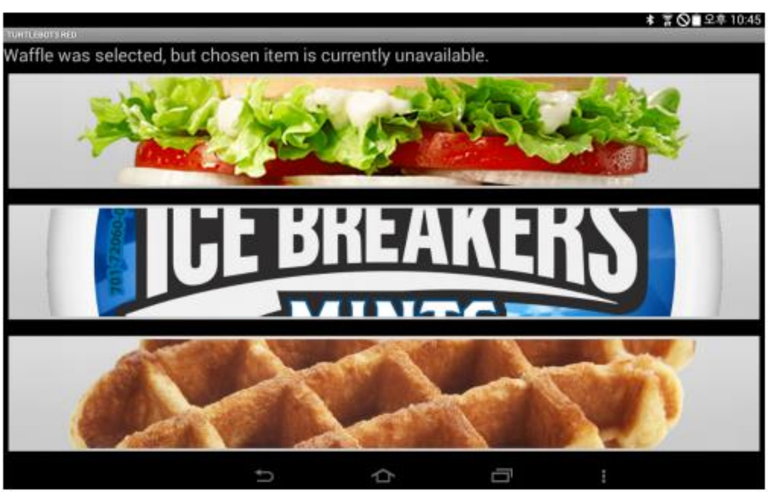


Рис. 185 Приклад 2 меню, відображеного у блокноті (коли вибрано недоступний пункт)

Вузли, контрольовані в сервісному підпорядкованому вузлі, безпосередньо пов'язані з керуванням роботом. У цьому прикладі використовується TurtleBot 3 Carrier 3 і вузли, які виконуються при виконанні SLAM і навігація, описані в розділах 10 і 11, є основними вузлами. Тут ми опишемо вихідний код TurtleBot3, який був змінений для розробки носія TurtleBot3. У цьому в основному використовуються пакети, показані на рис. 186. Кожна стрілка являє собою підпакет. Деякі з цих пакетів повинні бути змінені, щоб зробити робота служби доставки.

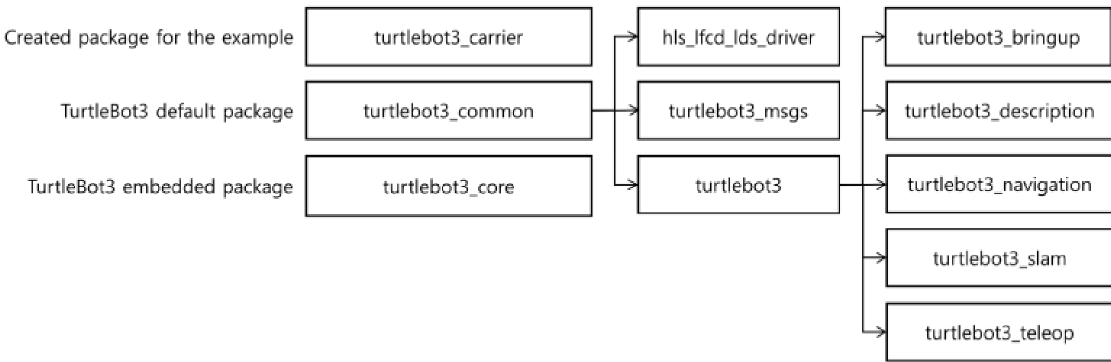


Рис. 186 Список використаних пакетів

Нижче описується модифіковане джерело для створення робота служби доставки. *poll()* функція обробляє значення вимірювання відстані прилеглих об'єктів за допомогою лазерного датчика LDS(HLSLFCD2). TurtleBot3 Carrier має стовпи навколо LDS, і він складається для служб доставка. Оскільки він розпізнає ці стовпи при роботі LDS, це вплине на результати SLAM або навігації. Тому, коли об'єкт виявлений на відстані менше визначеного значення, TurtleBot3 carrier встановить виявлене значення як "0". Пізніше, в навігації алгоритм, значення відстані " 0 "розпізнається як " немає об'єкта", і існування стовпа не впливає на SLAM або навігацію.

```
------------------------------------------------
hld_lfcd_lds_driver/src/hlds_laser_publisher.cpp
void LFCDLaser::poll(sensor_msgs::LaserScan::Ptr scan)
{
…omitted…
```

```
while (!shutting_down_ && !got_scan)
{
… omitted …
```

3

http://emanual.robotis.com/docs/en/platform/turtlebot3/friends/#turtlebot3-friends-carrier

```
if(start_count == 0)
{
if(raw_bytes[start_count] == 0xFA)
{
start_count = 1;
}
}
else if(start_count == 1)
{
if(raw_bytes[start_count] == 0xA0)
{
… omitted …
//read data in sets of 6
for(uint16_t i = 0; i < raw_bytes.size(); i=i+42)
{
if(raw_bytes[i] == 0xFA && raw_bytes[i+1] == (0xA0 + i / 42))
{
```

```
… omitted …
for(uint16_t j = i+4; j < i+40; j=j+6)
{
index = (6*i)/42 + (j-6-i)/6;
// Four bytes per reading
uint8_t byte0 = raw_bytes[j];
uint8_t byte1 = raw_bytes[j+1];
uint8_t byte2 = raw_bytes[j+2];
uint8_t byte3 = raw_bytes[j+3];
// Remaining bits are the range in mm
uint16_t intensity = (byte1 << 8) + byte0;
uint16_t range = (byte3 << 8) + byte2;
scan->ranges[359-index] = range / 1000.0;
scan->intensities[359-index] = intensity;
}
}
}
/// Add pillars ///
for(uint16_t deg = 0; deg < 360; deg++)
{
if(scan->ranges[deg] < 0.15)
{
scan->ranges[deg] = 0.0;
```

```
 scan->intensities[deg] = 0.0;
 }
 }
/// end of addition ///
 scan->time_increment = motor_speed/good_sets/1e8;
 }
 else
 {
 start_count = 0;
 }
 }
 }
}
----------------------------------------------------------------------------------
--
```

turtlebot3_core – це прошивка, встановлена на платі управління OpenCR, використовуваної TurtleBot3, і ' turtlebot3_motor_driver.cpp-це джерело, яке безпосередньо керує приводом, що використовується в Горлиця 3. Фактичний сервісний робот рухається з завантаженими об'єктами, тому для безпечного переміщення потрібно належний контроль. Тому ми додали управління профілем Dynamixels, яке не входить в джерело 'turtlebot3_motor_driver.cpp’. Тут значення ADDR_X_PROFILE_ ACCELERATION дорівнює 108. Для отримання

додаткової інформації про привід, будь ласка, зверніться до керівництва Dynamixel (http://emanual.robotis.com/).

```
----------------------- turtlebot3_core(for TurtleBot3
Waffle)/turtlebot3_motor_driver.cpp
bool Turtlebot3MotorDriver::init(void)
{
… omitted …
 // Enable Dynamixel Torque
 setTorque(left_wheel_id_, true);
 setTorque(right_wheel_id_, true);
 /// begin addition ///
 // Set Dynamixel Profile Acceleration
 setProfileAcceleration(left_wheel_id_, 15);
 setProfileAcceleration(right_wheel_id_, 15);
 /// end of addition ///
 … omitted …
 return true;
}
bool Turtlebot3MotorDriver::setTorque(uint8_t id, bool onoff)
{
… omitted …
}
```

```
bool Turtlebot3MotorDriver::setProfileAcceleration(uint8_t id, uint32_t
value)
{
 uint8_t dxl_error = 0;
 int dxl_comm_result = COMM_TX_FAIL;
 dxl_comm_result = packetHandler_->write4ByteTxRx(portHandler_,
id, ADDR_X_PROFILE_ACCELERATION,
value, &dxl_error);
 if(dxl_comm_result != COMM_SUCCESS)
 {
 packetHandler_->printTxRxResult(dxl_comm_result);
 }
 else if(dxl_error != 0)
 {
 packetHandler_->printRxPacketError(dxl_error);
 }
}
------------------------------------------------------------------------------------
------
```

Файл ' *turtlebot3_navigation.launch* ' запускає вузли, які виконуються при використанні навігації в TurtleBot3. Як описано раніше, вузли та теми ROS повинні бути згруповані в простір імен

груп, щоб використовувати ідентичний пакет ROS в багатьох роботах одночасно.

У наведеному вище вихідному коді запуску ми згрупували вузли з простором імен 'tb3g' і використовували функцію remap для отримання повідомлень від інших вузлів, які не згруповані в тому ж просторі імен.

Цей метод також використовується, коли ім'я теми не може бути змінено у вихідному коді. Зверніть увагу, що ця зміна повинна бути зроблена не тільки в файлі запуску, але і у всіх областях, де потрібно групування, наприклад у файлі RViz.

```
----------------------------
turtlebot3/turtlebot3_navigation/turtlebot3_navigation.launch
<launch>
<!—- addition starts -->
 <group ns="tb3g">
 <remap from="/tf" to="/tb3g/tf"/>
 <remap from="/tf_static" to="/tb3g/tf_static"/>
<!-- addition ends -->
 <arg name="model" default="waffle" doc="model type [burger, waffle]"/>
 … omitted …
 </node>
<!—- addition starts -->
```

```
 </group>
<!-- addition ends -->
</launch>
--------------------------------------------------------------------------------
```

Якщо до цього моменту все йшло гладко, побудова системи, показаної на рис. 187 буде успішною.

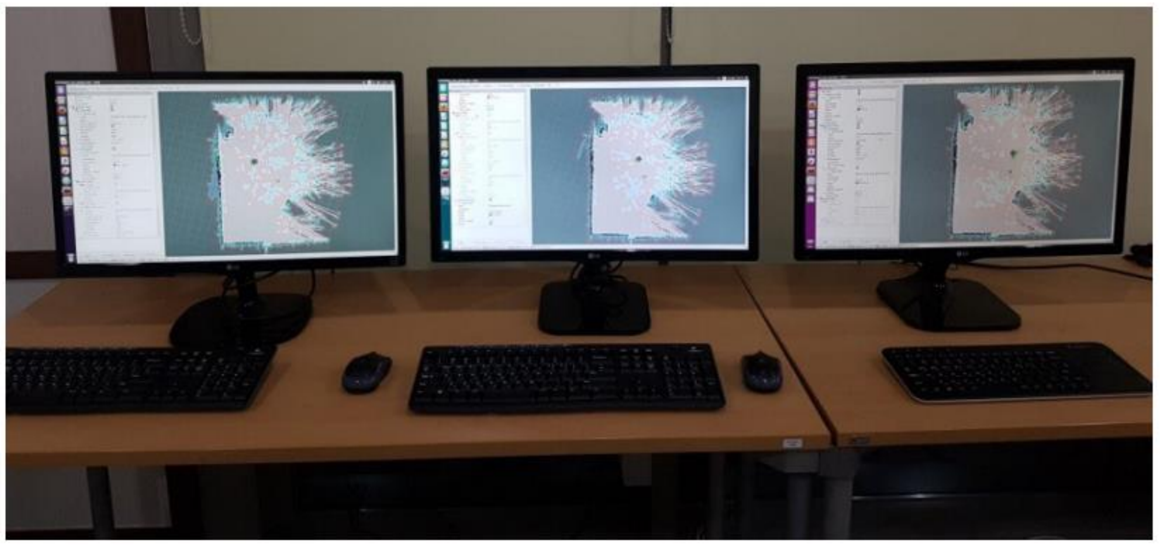


Рис. 187 Навігаційний вигляд кожного робота, що відображається на Rviz

У попередньому розділі ми розповіли про розміщення замовлень за допомогою майданчика, наприклад, меню, яке працює на майданчику на рис. 184. У цьому розділі ми встановимо Android Studio

IDE 4 на Linux і створимо середовище розробки Ros Java. Потім пояснимо простий приклад ROS Java.

Нижче описано, як встановити необхідні пакети для установки Android Studio IDE і застосування середовища ROSJava. ROSJava відноситься до клієнтської бібліотеки ROS, що працює на мові Java. Давайте спочатку встановимо необхідне середовище для використання Java. Необхідні пакети - це Java SE Development Kit(JDK), і вам потрібно вказати місце, де ви хочете його запускати. Ця книга описує, як завантажити JDK 8, але ви повинні змінити його, коли версія JDK буде оновлена.

```
---------------------------------------------------------------
$ sudo apt-get install openjdk-8-jdk
$ echo export PATH=${PATH}:/opt/android-sdk/tools:/opt/android-sdk/platform-tools:/opt/androidstudio/bin >> ~/.bashrc
$ echo export ANDROIS_HOME=/opt/android-sdk >> ~/.bashrc
$ source ~/.bashrc
---------------------------------------------------------------
```

Наступна команда завантажує необхідні інструменти для будівництва в ROS Java. Після цього встановіть і зберіть пакет, що містить систему ROS Java і приклади. Папка "android_core' - це папка робочого простору під назвою "catkin_ws' на ROS.

```
------------------------------------------------------------
$ sudo apt-get install ros-kinetic-rosjava-build-tools
$ mkdir -p ~/android_core
$ wstool init -j4 ~/android_core/src
https://raw.github.com/rosjava/rosjava/kinetic/
android_core.rosinstall
$ source /opt/ros/kinetic/setup.bash
$ cd ~/android_core
$ catkin_make
------------------------------------------------------------
```

Ось як встановити IDE Android Studio. Щоб уникнути плутанини, я буду використовувати однакові імена і розташування папок і файлів, описаних в Ros Android. Ці пакети є додатковими пакетами для установки mksdcard, які дозволяють використовувати SD-карту в IDE Android Studio для реалізації віртуального пристрою. Якщо ви не встановите ці пакети, функція mksdcard не буде встановлена.

```
------------------------------------------------------------
$ sudo apt
get install lib32z1 lib32ncurses5 lib32stdc++6
```

Ця книга встановлює IDE і SDK Android Studio в папку '/ opt', яка є папкою установки, рекомендованої Ros Android 5. Щоб встановити Програму в папку "/opt", вам необхідно змінити права користувача "/ opt ' на запис.

```
------------------------------------------------------------
$ sudo chown
R $USER:$USER /opt
------------------------------------------------------------
```

Інсталяційний файл IDE Android Studio можна знайти тут. https://developer.android.com/studio/index.html#download

Коли інсталяційний файл буде завантажений, розпакуйте його в файл '/ opt / android-studio'. Після розпакування папка налаштовується так, як показано на рис. 188.

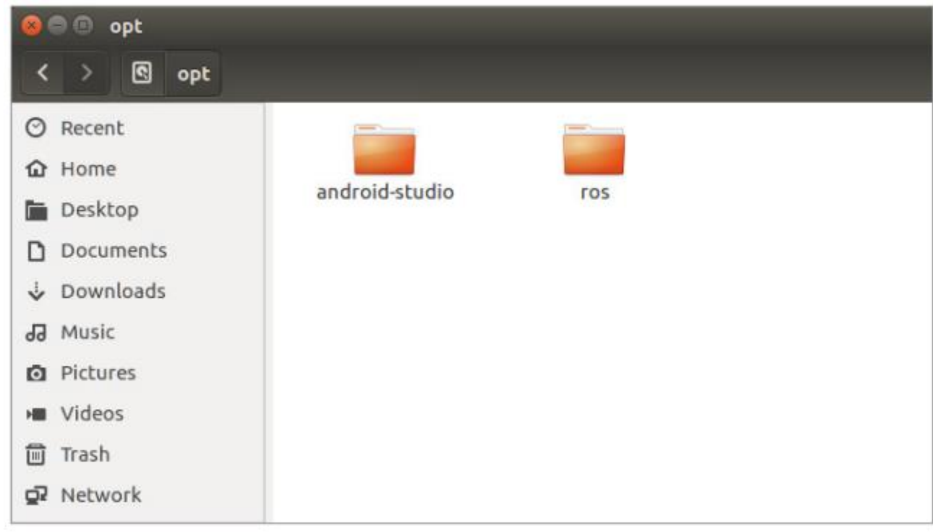


Рис. 188 Декомпресована IDE Android Studio

Після завершення вилучення введіть наступну команду, щоб почати установку IDE.

```
------------------------------------------------------------
$ /opt/android
studio/bin/studio.sh
------------------------------------------------------------
```

Коли з'явиться вікно, показане на рис. 189, натисніть кнопку "Виконати без імпорту", щоб перейти до ‘custom install’

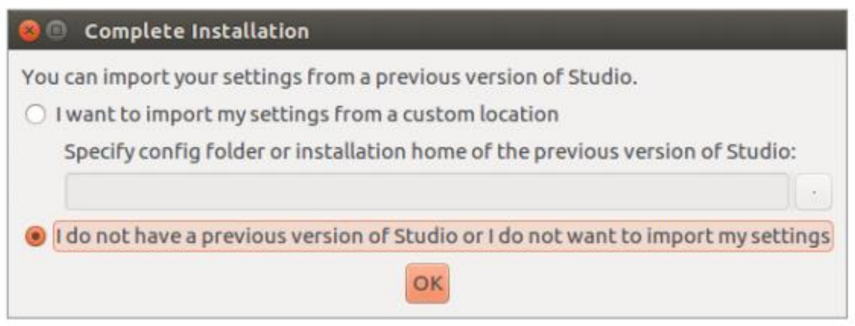


Рис. 189 Перше вікно, яке з’явилося під час інсталяції

Після цього ви повинні побачити вікно установки Android SDK, як показано на рис. 190. Зверніть увага, що вам потрібно вибрати місце установки в '/ opt / android-sdk'. Якщо у вас немає папки "android-sdk", створіть її в потрібному місці і продовжуйте.

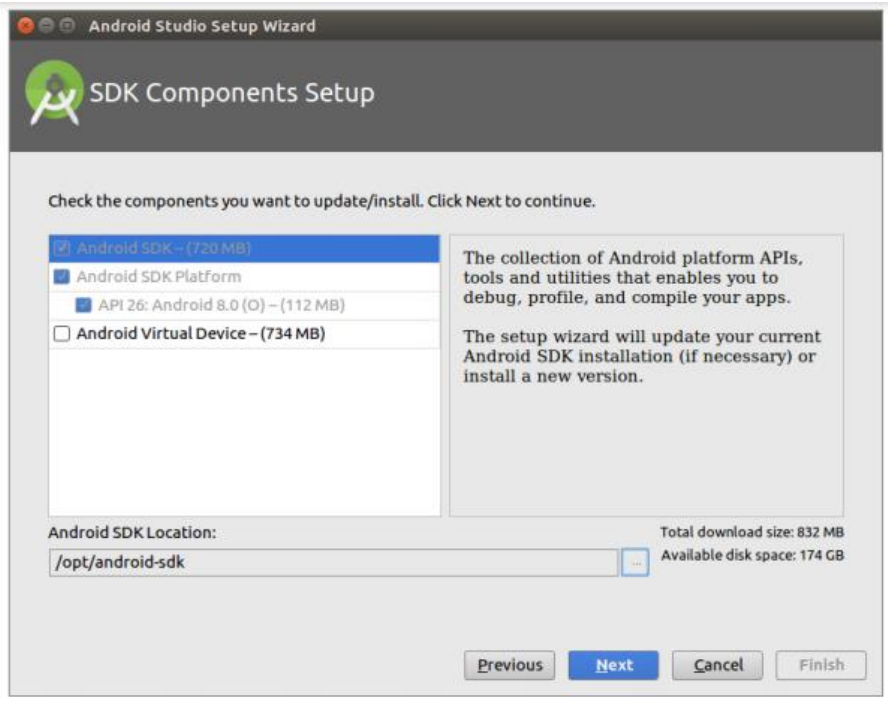


Рис. 190 Екран встановлення Android SDK

Якщо установка завершена, вона закінчується, як показано на рис. 191.

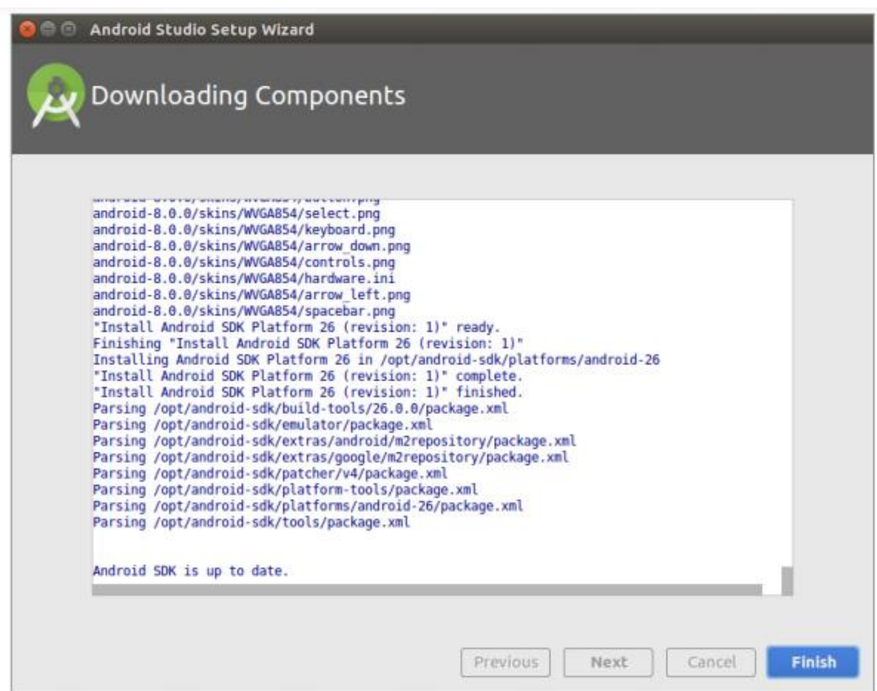


Рис. 191 Екран завершення установки

Коли IDE Android Studio буде успішно встановлена, з'явиться вікно " Ласкаво просимо в Android Studio", як показано на рис. 192.

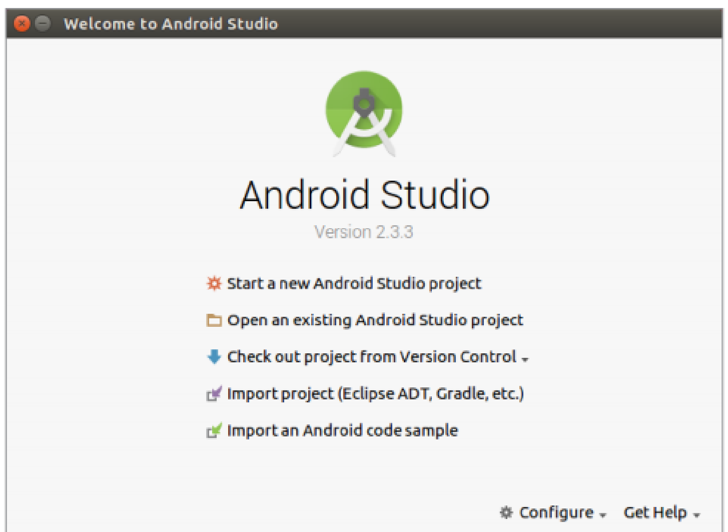


Рис. 192 Вітаємо в Android Studio

Тепер перейдіть до оновлення Android SDK через Configure → SDK Manager.

На рис. 193 доступні наступні SDK для оновлення: 10 (Gingerbread), 13 (Honeycomb), 15 (Ice cream) і 18 (Jellybean), і кожне число представляє рівень API SDK.

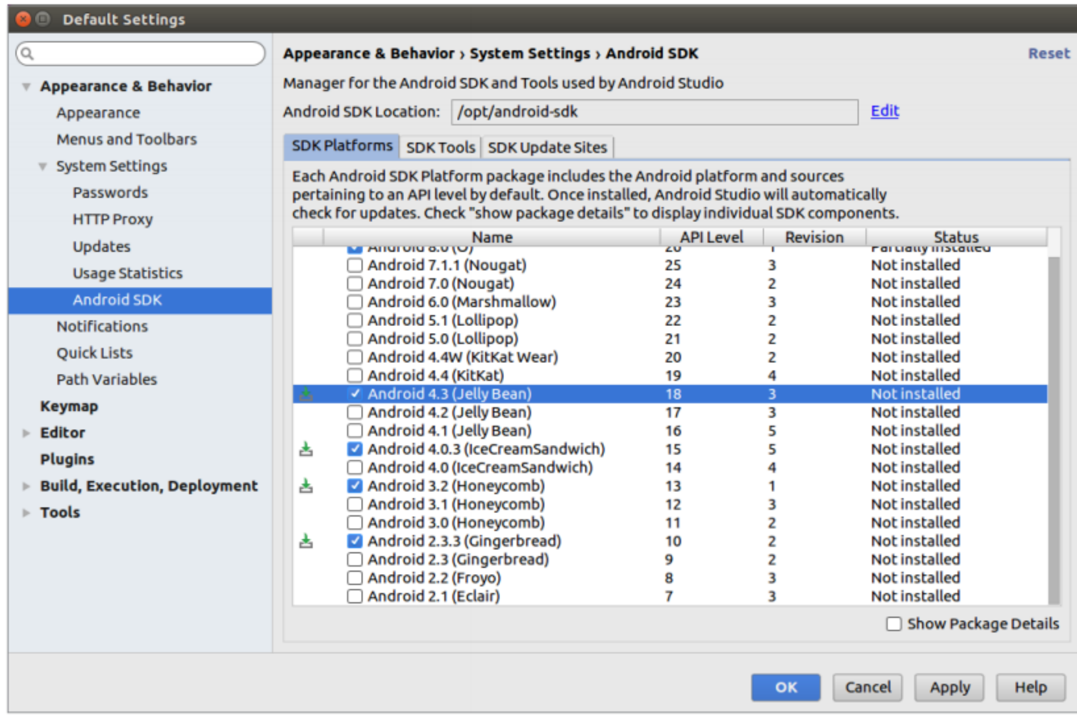


Рис. 193 Налаштування Android SDK

Після завершення налаштування натисніть на існуючий проект Android Studio у вікні і імпортуйте раніше встановлений "android_core", як показано на рис. 194. Коли ви натиснете кнопку ОК, з'явиться вікно IDE, показане на рис. 195, і почнеться збірка імпортованого джерела.

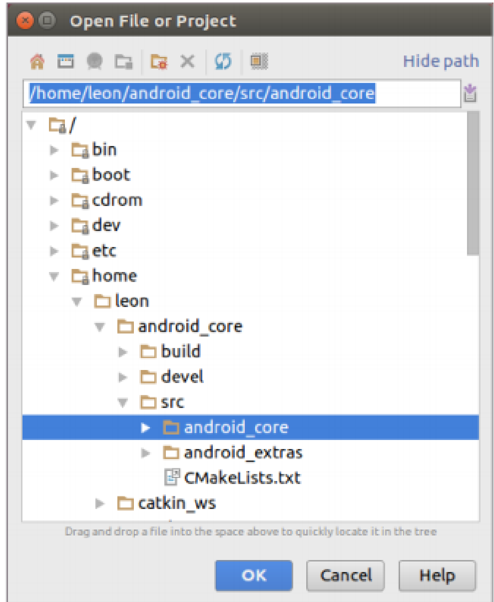


Рис. 194 Імпортний проект

Рис. 195 Екран імпорту IDE студії Android

Далі давайте запустимо приклад "android_tutorial_pubsub". Виберіть "android_tutorial_pubsub' у вікні вибору проекту у верхній частині вікна і натисніть кнопку відтворення поруч з ним, щоб знайти пристрій для виконання програми. Виберіть відповідний пристрій, як показано на рис. 196, і натисніть кнопку "ОК".

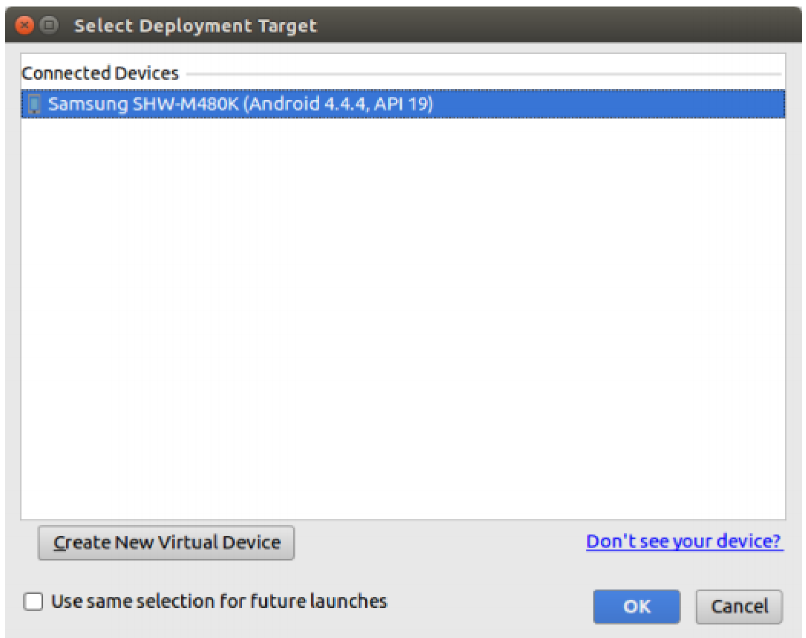


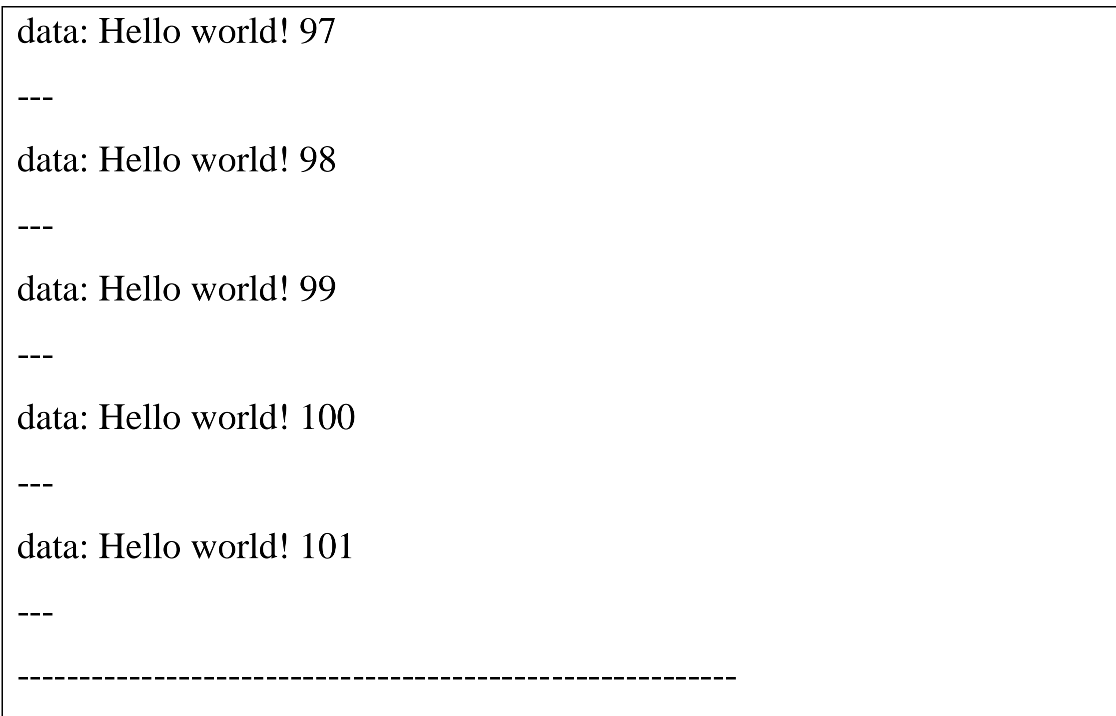
Рис. 196 Вікно вибору пристрою

Коли завантаження вихідного коду на пристрої успішно завершено, IP-адреса ROS master (комп'ютера, на якому працює roscore) вводиться на Термінал, як показано на рис. 197. Введіть відповідну IP-адресу та натисніть кнопку Підключитися.

Рис. 197 Вікно налаштувань IP ROS

Коли виконується "android_tutorial_pubsub", пристрій публікує " Hello world!", а тоді збільшує число як тип "std_msgs:: String'. Тепер давайте поглянемо на рядок, який пристрій публікує на вашому комп'ютері. Якщо ви подивитеся на тему "/ chatter "за допомогою команди" rostopic echo", як описано раніше, ви побачите, що ця тема публікується наступним чином.

```
---------------------------------------------------------
$ rostopic echo /chatter
data: Hello world! 96
---
```



```
data: Hello world! 97
---
data: Hello world! 98
---
data: Hello world! 99
---
data: Hello world! 100
---
data: Hello world! 101
---
------------------------------------------------------------
```

Ми розглянули SLAM і сервісного робота, які застосували навігацію в попередньому розділі. Як описано в цьому розділі, побудувати сервісного робота нескладно. Я закінчую цю главу надією, що ця книга допоможе вам побудувати гарного робота.

Розділ 13. Маніпулятор

13.1. Введення маніпулятора

Маніпулятори-це роботи, призначені для використання на заводах, де виконуються прості повторювані завдання. Він призначений для заміни небезпечної роботи або для заміни повторюваних завдань, і останнім часом було проведено багато досліджень, спрямованих на співпрацю з людиною.

У міру активізації досліджень взаємодії людини і робота (HRI), маніпулятори почали використовуватися у різних областях (медіа-мистецтво, VR тощо), а також на фабриках і надаванню нового досвіду широкій публіці. Об'єднавши цифровий привід і технологію 3D-друку, маніпулятори стали більш доступними для широкої публіки, і вони ростуть як великий гравець в індустрії виробників і освіти. З іншого боку, багато турбуються і бояться, що поєднання маніпуляторів і штучного інтелекту відніме у них роботу.

Маніпулятори давно стали інструментом збагачення суспільства і досі допомагають людям у багатьох різних областях. У майбутньому, якщо розвиток маніпулятора зможе проникнути в наше життя, не виходячи з її суті, очікується, що маніпулятор стане частиною нашого життя точно так само, як роботи-пилососи.

Ми введемо структурний опис маніпулятора і бібліотеку для маніпулятора, підтримувану ROS. Відкритий маніпулятор від

ROBOTICS є одним з маніпуляторів, що підтримують ROS, і має ту перевагу, що може легко виготовлятися за низькою ціною за допомогою приводів Dynamixel з 3D-друкованими деталями. За допомогою цього маніпулятора я представлю і поясню, як використовувати 3D-симулятор Gazebo, який працює з ROS та інтегрованою бібліотекою для маніпуляторів під назвою “Рухайся!. Нарешті”, я збираюся поговорити про сумісність між OpenManipulator, TurtleBot3 Waffle, Waffle Pi і як налаштувати і контролювати саму платформу.

Основна структура маніпулятора складається з основи, ланки, з'єднання і кінцевого ефектора, як показано на рис. 198.

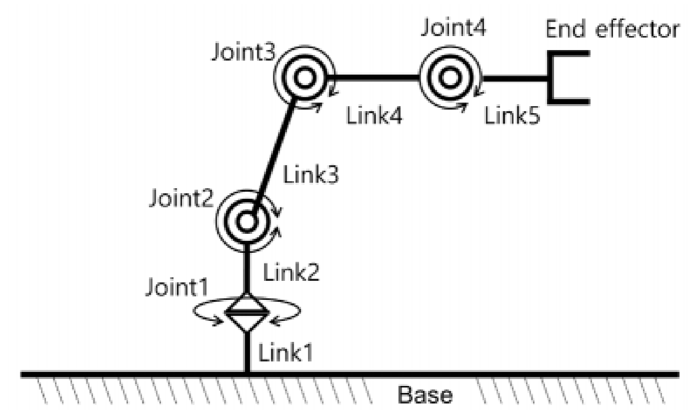


Рис. 198 Основна структура маніпулятора

Маніпулятор зазвичай має фіксовану форму з одного боку, і фіксована частина називається підставою. Підстава маніпулятора є найбільш жорсткою частиною, так як сила, прикладена до нього, пропорційна довжині і швидкості кінцевого ефектора. Підставою

також може бути об'єкт з рухом на зразок мобільного робота, який доповнює ступінь свободи маніпулятора.

Маніпулятор, заснований на "базі', складається з каскаду ланок і з'єднань. Посилання зазвичай має один суглоб, але може мати більш одного суглоба. Шарніри являють собою вісь обертання і в основному складаються з електродвигунів. Завдяки обертанню цього електродвигуна з'єднання виробляє рух ланки. По руху суглобів їх можна розділити на револьверні, призматичні, гвинтові, циліндричні, універсальні і сферичні. В останні роки публіці стали відомі з'єднання, що використовують гідравлічні, а не електричні двигуни, і активно ведуться дослідження з пошуку нових типів з'єднань, здатних замінити електродвигуни.

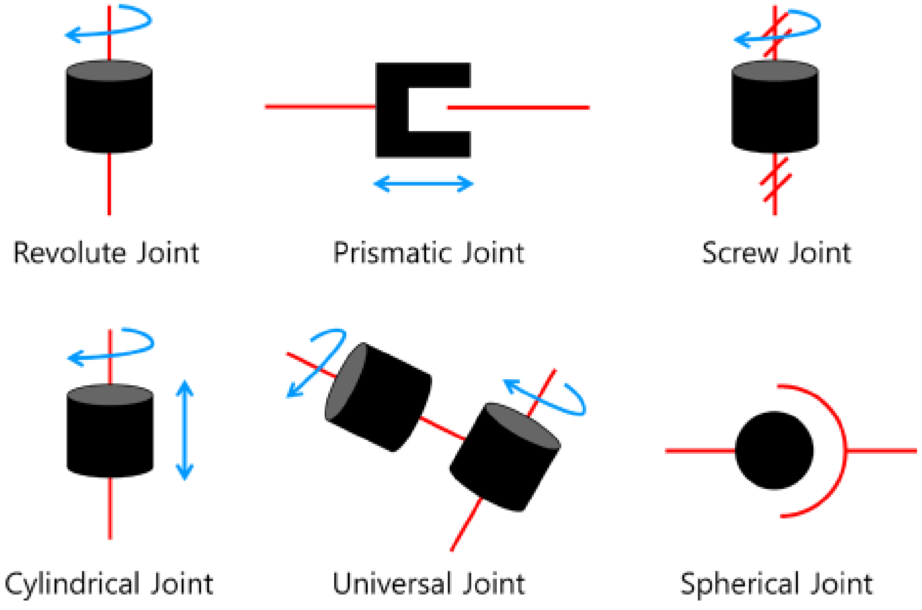


Рис. 199 Види суглобів

На підставці є каскад ланок і з'єднань, а в кінці-кінцевий ефектор. ‘Кінцевий ефектор " часто є захопленням через характеристики маніпулятора, який призначений для захоплення і переміщення об'єктів. Як показано на рис. 200, розміри і форма захоплень розрізняються в залежності від призначення, і було зроблено багато спроб забезпечити захоплення формою і розміром, які варіюються в залежності від мети використання, і було зроблено багато спроб забезпечити захоплення, здатний вибирати різні форми.

Рис. 200 Види затискачів (зліва направо: ROBOTIQ, ROBOTIS, Cornell University)

Спосіб управління маніпулятором можна класифікувати на ***спільне управління простором*** і ***управління простором завдань***.

Управління простором суглоба-це метод обчислення координати кінця маніпулятора шляхом введення кута повороту кожного суглоба, як показано на рис. 201. Координата кінця маніпулятора (X, Y, Z, i, z, and }) може бути отримана за допомогою прямої кінематики.

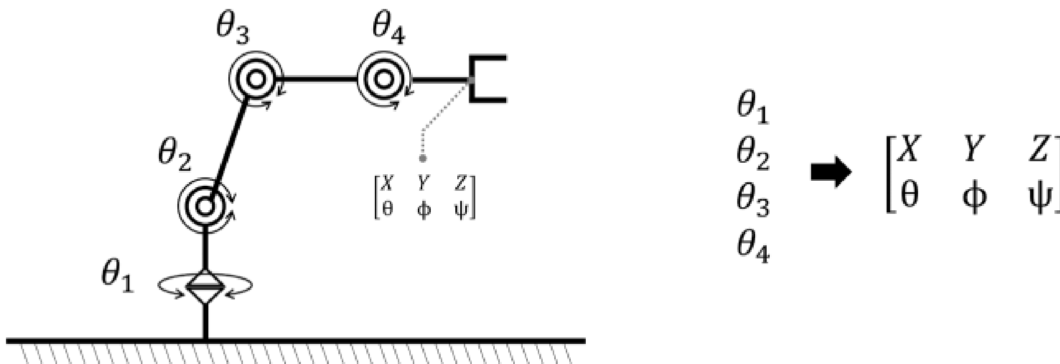


Рис. 201 Передня кінематика

Як показано на рис. 202, завдання просторового управління має вхід координати кінця маніпулятора і виходи обертання кожного суглоба, на відміну від спільного просторового управління. Положення об'єкта в просторі завдання включає його положення і орієнтацію. Оскільки ми живемо в тривимірному світі, положення об'єкта можна виразити як X, Y і Z, а його орієнтацію - як θ(roll), ϕ(pitch) і ψ (yaw). Давайте візьмемо як приклад чашку на столі. Чашка на столі (припускаючи, що центр маси чашки знаходиться в центрі чашки) можна сказати, що навіть якщо положення нерухоме, то його поза може бути змінена лежачи або обертаючи напрямок чашки. Іншими словами, при математичній інтерпретації це означає, що існує 6 невідомих, тому, якщо є 6 рівнянь, ви можете знайти єдине рішення.

Однак не всі маніпулятори мають шість ступенів свободи. Більш ефективно проектувати ступінь свободи відповідно до мети і середовища, в яких використовується маніпулятор. Ступінь обертання кожного суглоба в залежно від кінця маніпулятора може бути отриманий за допомогою зворотної кінематики.

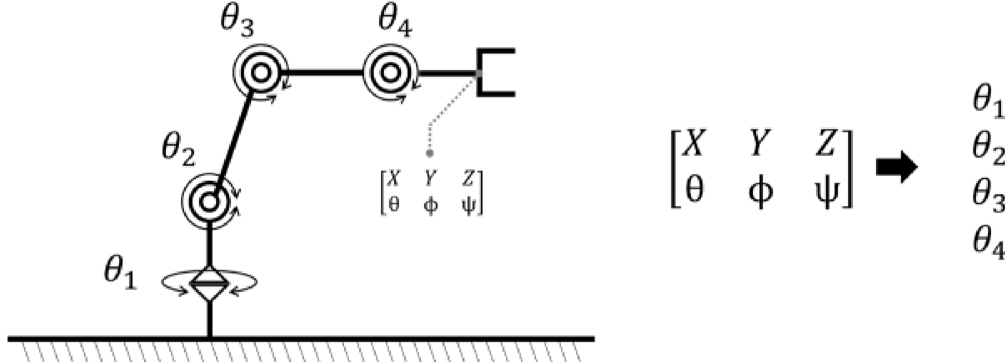


Рис. 202 Зворотня кінематика

ROS привернула увагу багатьох користувачів завдяки масштабованості та гнучкості з відкритим вихідним кодом. У міру того як все більше користувачів використовують ОС, платформи, що підтримують стрижні, поступово збільшуються, і тепер є компанії, які збирають і продають ці платформи. Крім того, платформи, створені приватними особами для досліджень або хобі, реєструються в офіційному пакеті ROS і представляються багатьом користувачам. Існує близько 180 платформ, підтримуваних ROS, і ви можете перевірити їх на Ros Robots.

Як правило, є промисловий маніпулятор ABB, який підтримує ROS-INDUSTRIAL, і JACO Kinova, який широко використовується для досліджень, також підтримує ROS. У Кореї є маніпулятор ROBOTIS-H, який підтримує ROS. Див. рис. 203 для кожного маніпулятора.

Рис. 203 Різні маніпулятори з підтримкою ROD (зліва направо: ABB, ROBOTIS, Kinova)

13.2. Моделювання та симуляція відкритого маніпулятора

ROS надає корисні інструменти для маніпуляторів.

Перше, потрібно створити файл Unified Robot Description Format (URDF) у вигляді розширюваної розмітки мови (XML) як інструмент візуалізації для моделювання роботів. Ви можете створити простий маніпулятор або спеціально розроблені деталі для вашого формату файлу URDF і побачити їх в інструменті візуалізації ROS RViz.

Друге, це 3D-симулятор Gazebo, який може імітувати реальне робоче середовище.

Середовище моделювання Gazebo, як і URDF, може бути легко створене за допомогою моделювання файлу формату опису (SDF) з

використанням XML. Gazebo також підтримує функції Ros-Control і плагінів для управління різними датчиками і роботами.

Третє, Рухай! - це інтегрована бібліотека і потужний інструмент для маніпуляторів, який надає відкриті бібліотеки, такі як Бібліотека кінематики і динаміки (KDL) і відкрита Бібліотека планування руху (OMPL), яка допомагає вам ідентифікувати різні функції маніпулятора, такі як розрахунок зіткнень, планування руху і демонстрація вибору і розміщення.

Давайте розглянемо, як використовувати три згаданих вище інструменти і реалізувати їх за допомогою прикладів коду.

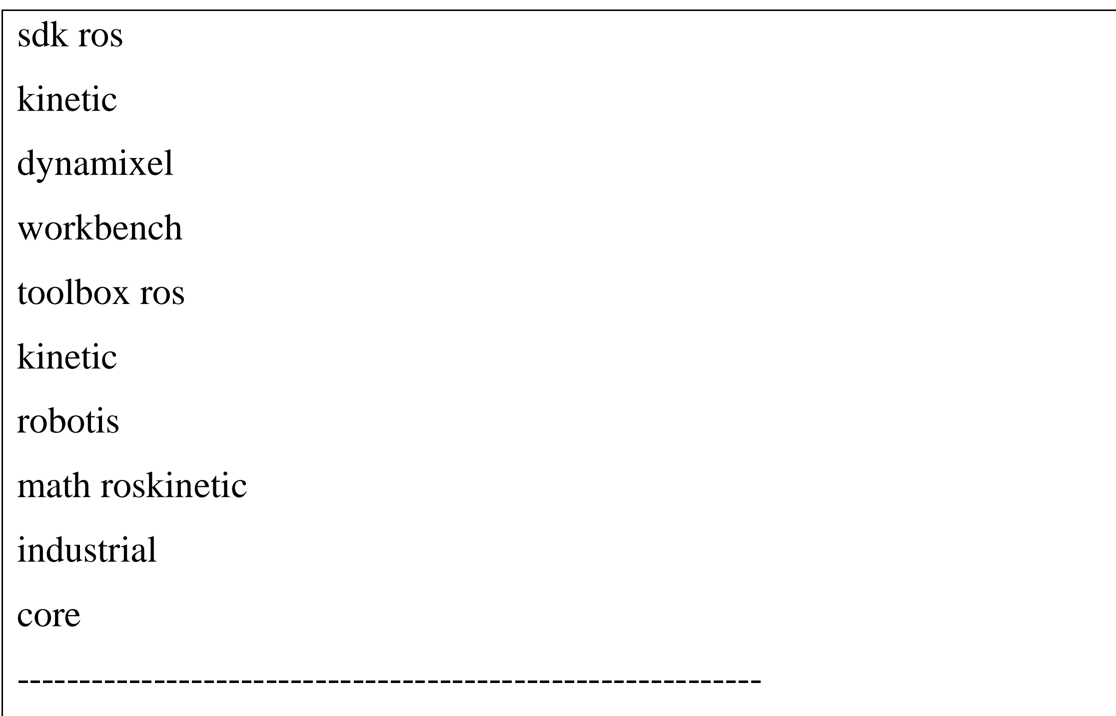
OpenManipulator - це програмне забезпечення з відкритим кодом та апаратне забезпечення з відкритим вихідним кодом, розроблене компанією ROBOTICS. OpenManipulator підтримує серію Dynamixel X, і ви можете створювати роботів, вибираючи приводи необхідних вам специфікацій. Крім того, оскільки він складається з базової рами і 3D-друкованої рами, можна виготовити новий тип маніпулятора в відповідно до вашого середовища або призначення. З цими характеристиками ми будемо надавати маніпулятори з різними формами і функціями, такими як SCARA, Planar і Delta на додаток до маніпулятора з чотирма суглобами. OpenManipulator підтримує ROS, Opencv, Arduino IDE і Processing.

OpenManipulator Chain, що використовується в цій главі, має саму базову форму маніпулятора, а кінцевий ефектор має лінійну

форму захоплення, виконану з 3D-друкованої рами. OpenManipulator та всі файли проектування ланцюжків доступні на Onshape, а вихідний код можна завантажити з сайту ROBOTIS GitHub. Вихідний код підтримує як ROS, Arduino, так і обробку, пакет Gazebo для OpenManipulator Chain і пакет MoveIt!. Крім того, OpenManipulator Chain механічно сумісний з TurtleBot3 Waffle і Waffle Pi і має потенціал до розширення функціоналу, щоб компенсувати недолік свободи.

Ми подивимося на відкритий вихідний код OpenManipulator Chain, URDF, Gazebo і MoveIt!. Нижче наведені пакети ROS, необхідні для використання перерахованих вище трьох інструментів. Давайте встановимо ці пакети.

```
-----------------------------------------------------------
$ sudo apt
get install ros
kinetic
ros
controllers ros
kinetic
gazebo* ros
kinetic
moveit* roskinetic
dynamixel
```



```
sdk ros
kinetic
dynamixel
workbench
toolbox ros
kinetic
robotis
math roskinetic
industrial
core
----------------------------------------------------------
```

Давайте розглянемо, як зробити модель кожного компонента для імітації реального маніпулятора у віртуальному просторі. Перш ніж розглядати URDF в ланцюзі OpenManipulator, давайте створимо простий URDF для маніпулятора, що складається з трьох з'єднань і чотирьох ланок.

Спочатку створіть пакет *'testbot_description'* таким чином, а потім створіть папку urdf.

Потім за допомогою редактора створіть файл *testbot.urdf* і введіть наступний приклад URDF.

```
--------------------------------------------------
$ cd ~/catkin_ws/src
$ catkin_create_pkg testbot_description urdf
```

```
$ cd testbot_description
$ mkdir urdf
$ cd urdf
$ gedit testbot.urdf
--------------------------------------------------


---------------------------------------------------------------
testbot_description/urdf/testbot.urdf
<?xml version="1.0" ?>
<robot name="testbot">
 <material name="black">
 <color rgba="0.0 0.0 0.0 1.0"/>
 </material>
 <material name="orange">
 <color rgba="1.0 0.4 0.0 1.0"/>
 </material>
 <link name="base"/>
 <joint name="fixed" type="fixed">
 <parent link="base"/>
 <child link="link1"/>
 </joint>
 <link name="link1">
 <collision>
```

```
<origin xyz="0 0 0.25" rpy="0 0 0"/>
<geometry>
<box size="0.1 0.1 0.5"/>
</geometry>
</collision>
<visual>
<origin xyz="0 0 0.25" rpy="0 0 0"/>
<geometry>
<box size="0.1 0.1 0.5"/>
</geometry>
<material name="black"/>
</visual>
<inertial>
<origin xyz="0 0 0.25" rpy="0 0 0"/>
<mass value="1"/>
<inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
</inertial>
</link>
<joint name="joint1" type="revolute">
<parent link="link1"/>
<child link="link2"/>
<origin xyz="0 0 0.5" rpy="0 0 0"/>
<axis xyz="0 0 1"/>
```

```
<limit effort="30" lower="-2.617" upper="2.617" velocity="1.571"/>
</joint>
<link name="link2">
<collision>
<origin xyz="0 0 0.25" rpy="0 0 0"/>
<geometry>
<box size="0.1 0.1 0.5"/>
</geometry>
</collision>
<visual>
<origin xyz="0 0 0.25" rpy="0 0 0"/>
<geometry>
<box size="0.1 0.1 0.5"/>
</geometry>
<material name="orange"/>
</visual>
<inertial>
<origin xyz="0 0 0.25" rpy="0 0 0"/>
<mass value="1"/>
<inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
</inertial>
</link>
<joint name="joint2" type="revolute">
```

```
<parent link="link2"/>
<child link="link3"/>
<origin xyz="0 0 0.5" rpy="0 0 0"/>
<axis xyz="0 1 0"/>
<limit effort="30" lower="-2.617" upper="2.617" velocity="1.571"/>
</joint>
<link name="link3">
<collision>
<origin xyz="0 0 0.5" rpy="0 0 0"/>
<geometry>
<box size="0.1 0.1 1"/>
</geometry>
</collision>
<visual>
<origin xyz="0 0 0.5" rpy="0 0 0"/>
<geometry>
<box size="0.1 0.1 1"/>
</geometry>
<material name="black"/>
</visual>
<inertial>
<origin xyz="0 0 0.5" rpy="0 0 0"/>
<mass value="1"/>
```

```
<inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
</inertial>
</link>
<joint name="joint3" type="revolute">
<parent link="link3"/>
<child link="link4"/>
<origin xyz="0 0 1.0" rpy="0 0 0"/>
<axis xyz="0 1 0"/>
<limit effort="30" lower="-2.617" upper="2.617" velocity="1.571"/>
</joint>
<link name="link4">
<collision>
<origin xyz="0 0 0.25" rpy="0 0 0"/>
<geometry>
<box size="0.1 0.1 0.5"/>
</geometry>
</collision>
<visual>
<origin xyz="0 0 0.25" rpy="0 0 0"/>
<geometry>
<box size="0.1 0.1 0.5"/>
</geometry>
<material name="orange"/>
```

```
 </visual>
 <inertial>
 <origin xyz="0 0 0.25" rpy="0 0 0"/>
 <mass value="1"/>
 <inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
 </inertial>
 </link>
</robot>
---------------------------------------------------------------------------------
---
```

URDF описує кожен компонент робота за допомогою XML-тегів. У форматі URDF спочатку опишіть ім'я робота, ім'я та тип бази (URDF припускає, що база є фіксованою ланкою), а також опис ланки, пов'язаної з базою, а потім опишіть кожне з'єднання і ланку.

Посилання описує ім'я, розмір, вагу, інерцію посилання.

З'єднання описують ім'я, тип і зв'язок, з'єднану з кожним з'єднанням.

Динамічні параметри робота, візуалізація і модель зіткнення можуть бути легко задані.

URDF ініціюється тегом <robot>, а в як правило, тег <link> і тег <joint> з'являються поперемінно для визначення зв'язків і з'єднань, які є компонентами робота.

Тег <transmission> також часто включається для взаємодії з ROS-управлінням, щоб встановити зв'язок між з'єднанням і приводом. Давайте докладніше розглянемо створений нами *testbot.urdf.*

Тег матеріалу описує таку інформацію, як колір і текстура посилання. В наступному прикладі ми визначили два матеріали, чорний і помаранчевий, щоб розрізняти кожну ланку. Колір використовує колірну мітку, яку можна встановити після опції rgba, ввівши число між 0.0 і 1.0, відповідні червоному, зеленому і синьому. Останнє число позначає значення прозорості (альфа) від 0,0 до 1,0, а значення 1,0 означає, що деталь непрозора.

```
-----------------------------------------------------
<material name="black">
 <color rgba="0.0 0.0 0.0 1.0"/>
</material>
<material name="orange">
 <color rgba="1.0 0.4 0.0 1.0"/>
</material>
-----------------------------------------------------
```

Першим компонентом маніпулятора є база, яка може бути представлена у вигляді посилання в URDF . Основа з'єднана з першою ланкою і з'єднанням, і це з'єднання не рухається і знаходиться на

початку координат (0, 0, 0). Давайте подивимося на перший тег посилання для більш докладного опису тега *<link>*.

```
--------------------------------------------------------
<link name="base"/>
<joint name="fixed" type="fixed">
 <parent link="base"/>
 <child link="link1"/>
</joint>
<link name="link1">
 <collision>
 <origin xyz="0 0 0.25" rpy="0 0 0"/>
 <geometry>
 <box size="0.1 0.1 0.5"/>
 </geometry>
 </collision>
 <visual>
 <origin xyz="0 0 0.25" rpy="0 0 0"/>
 <geometry>
 <box size="0.1 0.1 0.5"/>
 </geometry>
 <material name="black"/>
 </visual>
 <inertial>
```

```
 <origin xyz="0 0 0.25" rpy="0 0 0"/>
 <mass value="1"/>
 <inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
 </inertial>
</link>
---------------------------------------------------
```

Тег URDF <link> складається з колізійних, візуальних та інерційних тегів (див. рис. 204), як і в наведеному вище прикладі link1. Мітка зіткнення дозволяє ввести геометричну інформацію, що вказує діапазон інтерференції ланки. Початок координат вказує координати центру інтерференційного діапазону. А геометрія записує форму і розмір інтерференційного діапазону, центрованого за вихідною координатою. Наприклад, інтерференційний діапазон шестигранника- це величина ширини, довжини і висота. На додаток до шестигранного типу існують циліндричний тип і сферичний тип, і кожна з цих форм має різні параметри для введення. Напишіть фактичну форму у візуальному тезі. Походження і геометрія ідентичні тегам зіткнення. Ви також можете ввести тут CAD-файли, такі як STL і DAE. Ви можете використовувати CAD-модель в тезі collision, але її можна використовувати тільки з деякими фізичними движками, такими як ODE або Bullet. DART і Simbody не підтримують файли САПР. Інерційна мітка визначає вага ланки (кг) і момент інерції·кг * м2). Ця

інерційна інформація може бути отримана за допомогою проектного програмного забезпечення або фактичних вимірювань і розрахунків і використовується для моделювання динаміки.

Описані посилання 1, посилання 2, посилання і 4 в прикладі testbot.urdf зсуваються від початку координат верхній суглоб (fixed, joint1, і joint3 відповідно) зміщення 0,25 м, де площа колонки 0,1 м шириною і 0,1 м в довжину і на 0,5 м (0,25 м в сторону плюс і 0.25 м в мінус напрямок) в довжину, що проходить в напрямку осі Z від переміщеного центру. Аналогічно для ланки 3 початок координат зміщено на 0,5 м від початку з'єднання (joint2), де розташований шестигранник довжиною 1 м і шириною 0,1 м і довжиною 0,1 м в напрямку осі Z.

Розуміння відносного перетворення координат в URDF може бути досить складним при першому знайомстві. Можливо, буде легше зрозуміти кожне значення конфігурації, побачивши його, але корисно зрозуміти, як виражається відносне перетворення координат кожної осі. RViz, як показано на рис. 199. Я порадив би тобі спробувати.

Таблиця 30

Властивості тега посилання

<link>:	Візуалізація зв'язків, Налаштування інформації про зіткнення та інерції

<collision>:	Встановіть інформацію для розрахунку колізії посилань
<visual>:	Налаштування інформації візуалізації для посилань
<inertial>:	Встановіть інерційну інформацію для посилання
<mass>:	Установка вагу ланки (кілограм)
<inertia>:	Установка тензора інерції
<origin>:	Встановіть перетворення і поворот відносно системи координат ланки
<geometry>:	Введіть форму моделі: коробка, циліндр і сфера.
<collision>	Тег, в якому ви можете скоротити час розрахунку, вказавши його в простій формі
<material>:	Налаштування кольору і текстури посилань

Файли дизайну форматів COLLADA (.dae), STL (.stl) також можуть бути імпортовані.

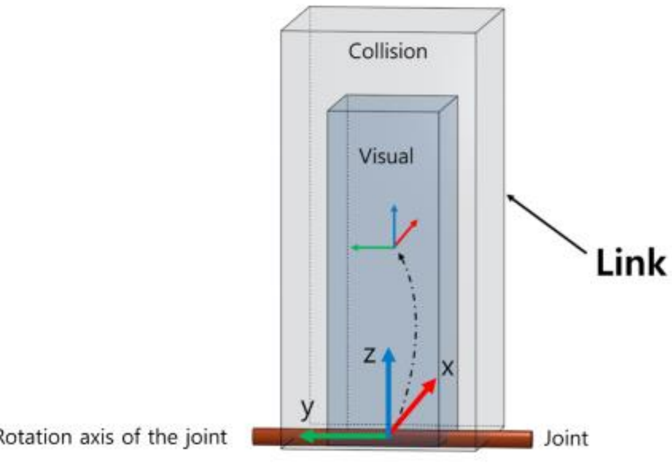


Рис. 204 Елементи моделювання посилань

Далі давайте розглянемо спільний тег, який з'єднує посилання з посиланнями. Мітка з'єднання описує характеристики з'єднання, як показано на рис. 204. Зокрема, назва і тип з'єднання, такі як обертається, призматична, безперервна, нерухома, плаваюча і плоска форма. Мітка з'єднання також описує імена двох ланок, які з'єднані, розташування з'єднань і обмеження руху осі з точки зору обертання і поступального руху. Підключеним посиланням присвоюються імена батьківських посилань і дочірніх посилань. Батьківське посилання зазвичай є найближчою до базового посилання.

У наступному прикладі показано Налаштування з'єднання для з'єднання2.

```
-----------------------------------------------------------------
<joint name="joint2" type="revolute">
 <parent link="link2"/>
 <child link="link3"/>
 <origin xyz="0 0 0.5" rpy="0 0 0"/>
 <axis xyz="0 1 0"/>
 <limit effort="30" lower="-2.617" upper="2.617" velocity="1.571"/>
</joint>
-----------------------------------------------------------------
```

Таблиця 31

Властивості спільної мітки

<joint>:	Зв'язок з налаштуванням типу з'єднання і з'єднання
<parent>:	батьківська ланка суглоба
<child>:	дочірня ланка суглоба
<origin>:	Перетворення системи координат батьківського посилання в систему координат дочірнього посилання
<axis>:	Встановити вісь обертання
<limit>:	Встановіть швидкість, силу і радіус з'єднання (застосовується тільки для обертових або призматичних з'єднань)

Давайте подивимося ближче, щоб зрозуміти. Тип joint2 був встановлений на револьверне з'єднання. Батьківське посилання встановлено на link2, а ДОЧІРНЄ-на link3. Крім того, початок координат визначає відносну позу (положення + орієнтацію) системи координат joint2, причому система координат суглоба 1 є початком координат в якості початку координат. Наприклад, початок координат joint2 знаходиться на відстані 0,5 м від joint1 в напрямку осі z системи координат joint1 . Наступний крок-Налаштування осі. В Налаштуваннях осі запишіть напрямок осі обертання якщо це шарнір обертального типу, то і напрямок руху, якщо це шарнір перекладного типу. У разі joint2 воно встановлюється як з'єднання, що обертається в напрямку осі y. межа встановлює обмеження для спільного рух. Властивості включають силу (зусилля, одиниця виміру В Н),

мінімальний, максимальний кут (нижній, Верхній, в радіанах) і швидкість (в рад/с), задану суглобу.

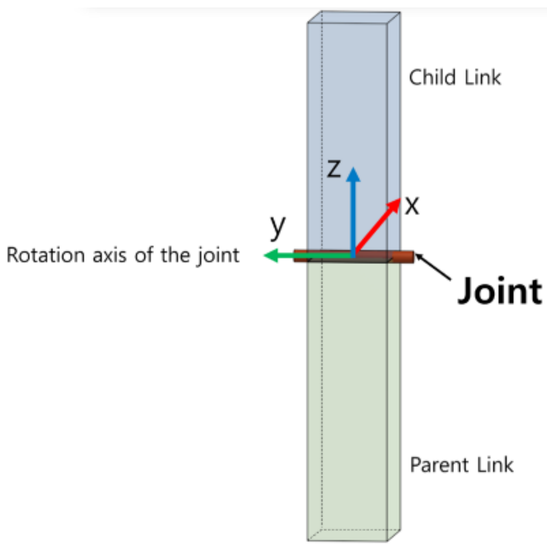


Рис. 205 Моделюючі фактори суглобів

Коли ви закінчите створювати модель, давайте розглянемо кожну ланку і з'єднання, щоб побачити, чи є вони логічно правильними. У ROS ви можете перевірити граматичну помилку URDF, створену командою 'check_urdf', і відношення з'єднання кожного посилання в наступному прикладі. Якщо файл граматично і логічно коректний, ми можемо підтвердити, що посилання 1, 2, 3 і 4 пов'язані наступним чином образ.

```
----------------------------------------
$ check_urdf testbot.urdf
robot name is: testbot
---------- Successfully Parsed XML ---------------
```

```
root Link: base has 1 child(ren)
 child(1): link1
 child(1): link2
 child(1): link3
 child(1): link4
-----------------------------------------
```

Далі давайте побудуємо графік моделі,яку ми створили за допомогою програми "urdf_to_graphiz'. Якщо ви запустите ' urdf_ to_graphiz', як у наступному прикладі, будуть створені файли '*.gv' і'*.pdf’. Якщо ви подивитеся на програму перегляду PDF-файлів, то побачите взаємозв'язок між зв'язком і з'єднанням, а також відносне перетворення координат між кожним з'єднанням, як показано на рис. 206.

```
-----------------------------------------
$ urdf_to_graphiz testbot.urdf
Created file testbot.gv
Created file testbot.pdf
-----------------------------------------
```

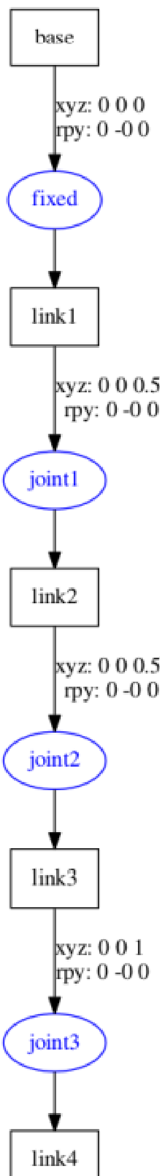


Рис. 206 ВІдношення між URDF та посиланням

Це найшвидший спосіб перевірити зв'язок моделі з 'check_urdf' і ‘urdf_to_ graphiz’. Нарешті, давайте перевіримо модель робота за

допомогою RViz. Для цього перейдіть в розділ "testbot_description". Упакуйте папку та створіть ' testbot.запустіть файл', як показано в наступному прикладі.

```
-----------------------------------------
$ cd ~/catkin_ws/src/testbot_description
$ mkdir launch
$ cd launch
$ gedit testbot.launch
-----------------------------------------


-----------------------------------------------------
testbot_description/launch/testbot.launch
<launch>
 <arg name="model" default="$(find testbot_description)/urdf/testbot.urdf" />
 <arg name="gui" default="True" />
 <param name="robot_description" textfile="$(arg model)" />
 <param name="use_gui" value="$(arg gui)"/>
 <node pkg="joint_state_publisher" type="joint_state_publisher" name="joint_state_publisher"/>
 <node pkg="robot_state_publisher" type="state_publisher" name="robot_state_publisher"/>
</launch>
```

Файл запуску складається з параметрів, що містять URDF, вузол" joint_state_publisher " і вузол "robot_state_publisher". Вузол 'joint_state_publisher' публікує спільний стан роботи, створеної URDF, через повідомлення 'sensor_msgs / JointState' і надає графічний інструмент для передачі команд з'єднанням. Вузол "robot_state_publisher' публікує результат прямої кінематики, обчисленої за допомогою інформації про роботу, та інформацію про тему " sensor_msgs / JointState" , задану в URDF у вигляді повідомлення tf (див.207).

Рис. 207 Теми у вузлі joint_state_publisher та вузлі robot_state_publisher

Як тільки все буде готово, запустіть testbot.launch і RViz наступним чином:

```
$ roslaunch testbot_description testbot.launch
$ rviz
------------------------------------------
```

При виконанні файлу запуску виконується графічний інтерфейс вузла joint_state_publisher, як показано на рис. 208. Тут ви можете контролювати значення суглобів 1, 2 і 3. Якщо ви запускаєте RViz, виберіть "база" в Налаштуваннях фіксованого кадру і натисніть кнопку [Додати] в лівому нижньому кутку, щоб додати "RobotModel", щоб побачити форму кожного з'єднання і посилання в RViz, як показано на рис. 209. Якщо ви додасте відображення "TF' і зміна значення" Альфа " моделі робота приблизно до 0,3, ви можете перевірити форму кожної ланки і співвідношення між суглобами, як показано на нижньому малюнку рис. 209.

Якщо ви налаштуєте панель GUI вузла joint_state_publisher, то побачите, що віртуальний робот на RViz управляється так, як показано на рис. 210. Відповідний вихідний код можна знайти в репозиторії GitHub:

- https://github.com/ROBOTIS-GIT/ros_tutorials/tree/master/testbot_description

Рис.208 Спільний державний видавець GUI

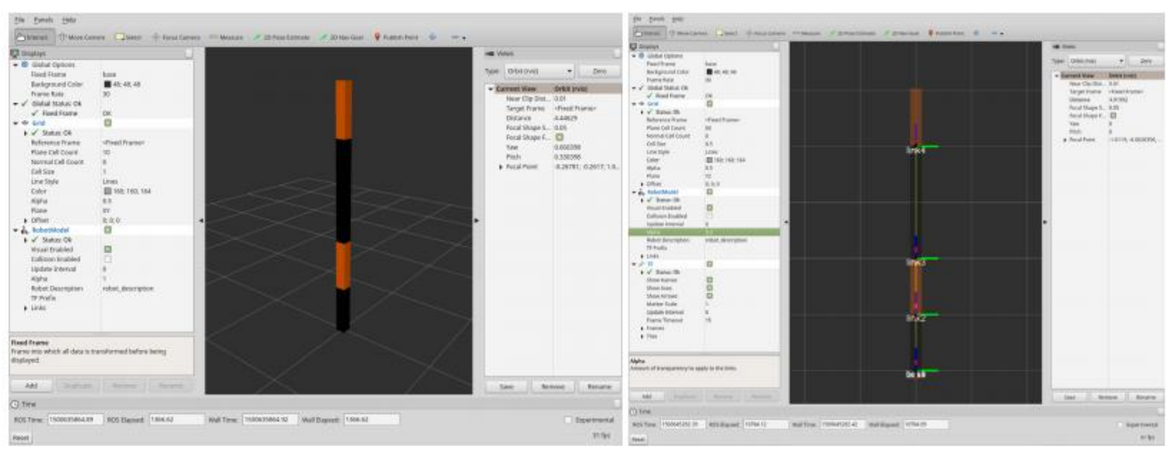


Рис. 209 Перегляд RViz кожного з'єднання та посилання

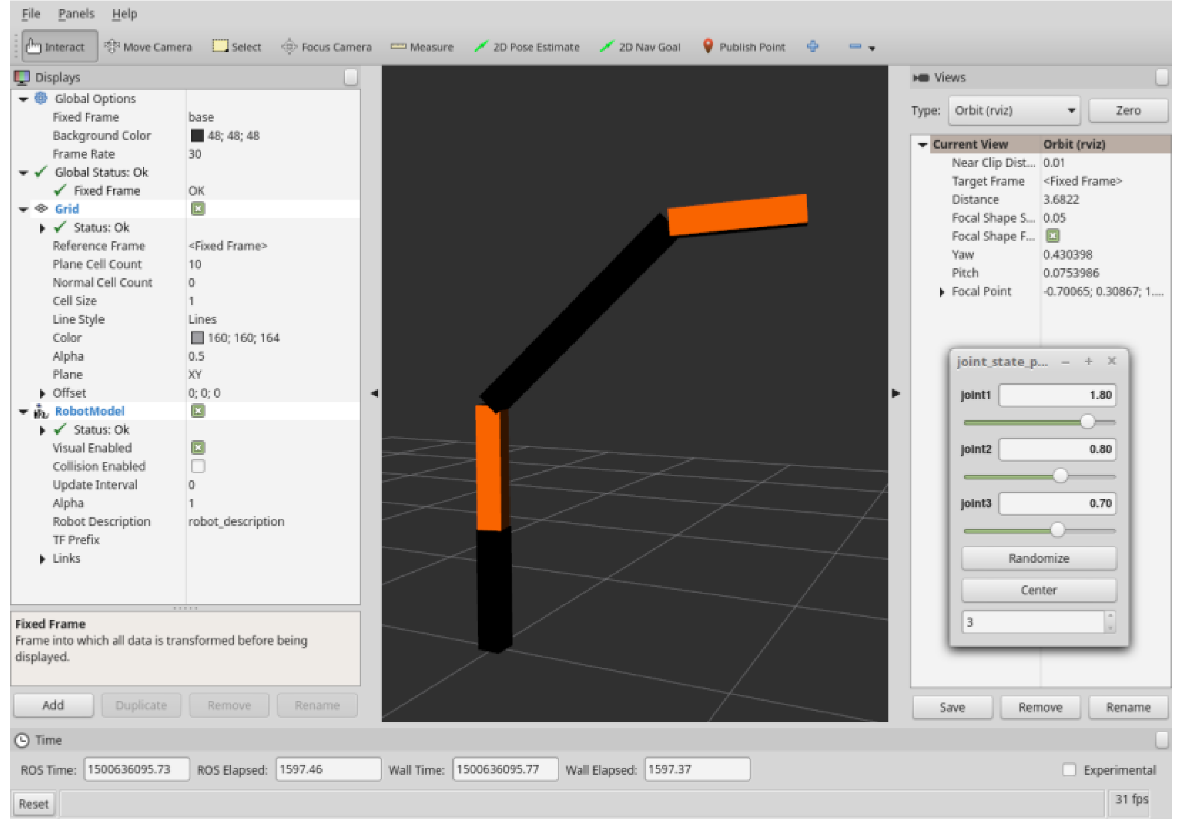
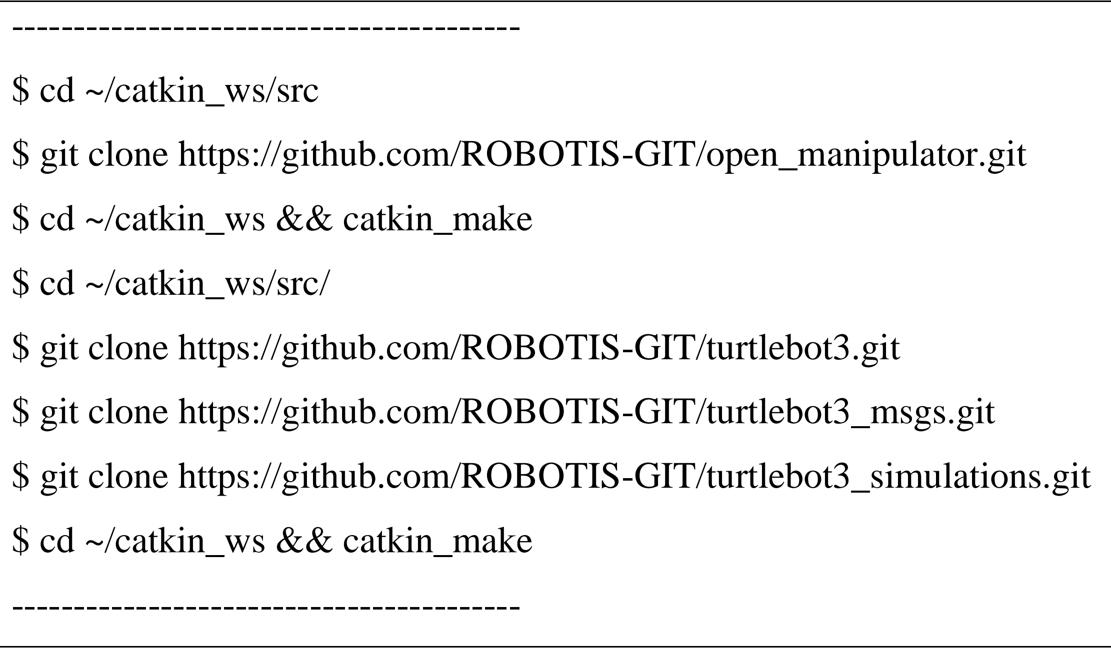


Рис.10 Результати маніпулювання кожним стиком у Спільному державному видавництві GUI

Якщо ви налаштуєте панель GUI вузла joint_state_publisher, то побачите, що віртуальний робот на RViz управляється так, як показано на рис. 13-13. Відповідний вихідний код можна знайти в репозиторії GitHub: ми створили 3-осьовий маніпулятор відповідно до формату URDF і підтвердили його за допомогою RViz. Виходячи з цього, давайте розглянемо URDF ланцюга OpenManipulator, яка складається з 4-осьового шарніра і лінійного захоплення. По-перше, завантажте вихідний код з GitHub на Open Manipulator і TurtleBot3



```
-----------------------------------------
$ cd ~/catkin_ws/src
$ git clone https://github.com/ROBOTIS-GIT/open_manipulator.git
$ cd ~/catkin_ws && catkin_make
$ cd ~/catkin_ws/src/
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3.git
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git
$ cd ~/catkin_ws && catkin_make
-----------------------------------------
```

Ось конфігурація в папці OpenManipulator, скопійована з Github.

```
----------------------------------------
$ cd ~/catkin_ws/src/open_manipulator
$ ls
Arduino                              → Library for Arduino
open_manipulator                     → MetaPackage
open_manipulator_description         → Modeling package
open_manipulator_dynamixel_ctrl      → Dynamixel control package
open_manipulator_gazebo              → Gazebo package
open_manipulator_moveit              → MoveIt! package
open_manipulator_msgs                → Message package
open_manipulator_position_ctrl       → Position control package
```

```
open_manipulator_with_tb3            →    OpenManipulator    and
TurtleBot3 package
---------------------------------------
```

Пакет моделювання (open_manipulator_description) складається з папки запуску, що містить виконувані файли, папки meshes, що містить файли проектування, папки src, що містить вузол Publisher, і папки urdf. Відкрийте папку urdf і папку запуску і подивіться на конфігурація.

```
---------------------------------------
$ roscd open_manipulator_description/urdf
$ ls
materials.xacro                  → Material info
open_manipulator_chain.xacro          → Manipulator modeling
open_manipulator_chain.gazebo.xacro       →    Manipulator    Gazebo
modeling

$ roscd open_manipulator_description/launch
$ ls
open_manipulator.rviz                 → RViz configuration file
open_manipulator_chain_ctrl.launch  →  File  to  execute  manipulator
state info Publisher node
```

```
open_manipulator_chain_rviz.launch → File to execute manipulator modeling info visualization node
---------------------------------------
```

Після перегляду файлів відкрийте файл materials.xacro.

```
---------------------------------------
$ roscd open_manipulator_description/urdf
$ gedit materials.xacro
---------------------------------------


---------------------------------------------------
open_manipulator_description/urdf/materials.xacro
 <?xml version="1.0"?>
<robot>
 <material name="black">
 <color rgba="0.0 0.0 0.0 1.0"/>
 </material>
 <material name="white">
 <color rgba="1.0 1.0 1.0 1.0"/>
 </material>
 <material name="red">
 <color rgba="0.8 0.0 0.0 1.0"/>
 </material>
 <material name="blue">
```

```
 <color rgba="0.0 0.0 0.8 1.0"/>
 </material>
 <material name="green">
 <color rgba="0.0 0.8 0.0 1.0"/>
 </material>
 <material name="grey">
 <color rgba="0.5 0.5 0.5 1.0"/>
 </material>
 <material name="orange">
 <color rgba="${255/255} ${108/255} ${10/255} 1.0"/>
 </material>
 <material name="brown">
 <color rgba="${222/255} ${207/255} ${195/255} 1.0"/>
 </material>
</robot>
-------------------------------------------------------------------------
```

Макрос XML-це макрос, який дозволяє викликати код, скорочено називається xacro. Я рекомендував вам створити макрос для багаторазово використовуваних кодів. Файл "material.xacro" визначає кольори, необхідні для візуалізації майбутнього маніпулятора.

Далі розглянемо файл URDF, необхідний для моделювання та візуалізації OpenManipulatorChain.

```
-------------------------------------------------------
```

```
$ roscd open_manipulator_description/urdf
$ gedit open_manipulator_chain.xacro
-------------------------------------------------------


------------------------------
open_manipulator_description/urdf/open_manipulator_chain.xacro
<!-- some parameters -->
<xacro:property name="pi" value="3.141592654" />
<!-- Import all Gazebo-customization elements, including Gazebo colors
-->
<xacro:include                                   filename="$(find
open_manipulator_description)/urdf/open_manipulator_chain.gazebo
.xacro" />
<!-- Import RViz colors -->
<xacro:include                                   filename="$(find
open_manipulator_description)/urdf/materials.xacro" />
---------------------------------------------------------------------------
```

URDF має багато повторюваних фраз для того, щоб представляти сполучну структуру ланок і з'єднань, і тому має недолік в тому, що займає багато часу на модифікацію. Однак використання xacro може значно скоротити ці завдання. Наприклад, можна ефективно керувати кодом, задаючи змінну кола, як зазначено вище, або окремо керуючи файлом інформації про матеріал і файлом

конфігурації Альтанки, створеним вище, і включаючи його в фактично використовуваний файл.

Раніше ми вже розповідали про теги <link> і <joint> при створенні URDF для 3-осьовий маніпулятор. OpenManipulator chain також використовує тег <transmission> для роботи з ROS-Control. Давайте поглянемо на мітку передачі.

```
-----------------------------
open_manipulator_description/urdf/open_manipulator_chain.xacro
<!-- Transmission 1 -->
<transmission name="tran1">
 <type>transmission_interface/SimpleTransmission</type>
 <joint name="joint1">
 <hardwareInterface>PositionJointInterface</hardwareInterface>
 </joint>
 <actuator name="motor1">
 <hardwareInterface>PositionJointInterface</hardwareInterface>
 <mechanicalReduction>1</mechanicalReduction>
 </actuator>
</transmission>
-----------------------------------------------------------------------------
```

<transmission> - обов'язковий тег для взаємодії з ROS-Control. Він вводить командний інтерфейс між з'єднанням і приводом.

Командні інтерфейси-це зусилля, швидкість і положення, і користувачі можуть вибрати потрібний керуючий вхід.

Таблиця 32

<transmission> tag

<transmission>	Змінна настройка між з'єднаннями і виконавчими механізмами
<type>	Тип установки для способу передачі крутного моменту
<joint>	Інформація про конфігурацію з'єднання
<hardwareInterface>	Налаштування апаратного інтерфейсу
<actuator>	Настройка інформації про привід
<mechanicalReduction>	Установка передавального числа між приводом і шарніром

OpenManipulator Chain складається з чотирьох шарнірів (двигунів) і п'яти ланок, лінійний захват складається з двох ланок і одного шарніра (двигуна). За винятком того, що лінійне захоплення є призматичним, решта опису збігаються з попереднім описом, тому, будь ласка, перевірте файл URDF.

Запустіть файл запуску, щоб візуалізувати завершений файл URDF в RViz, і перемістіть з'єднання за допомогою графічного інтерфейсу joint_state_publisher, як показано на рис. 211.

```
$ roslaunch open_manipulator_description open_manipulator_chain_rviz.launch
```

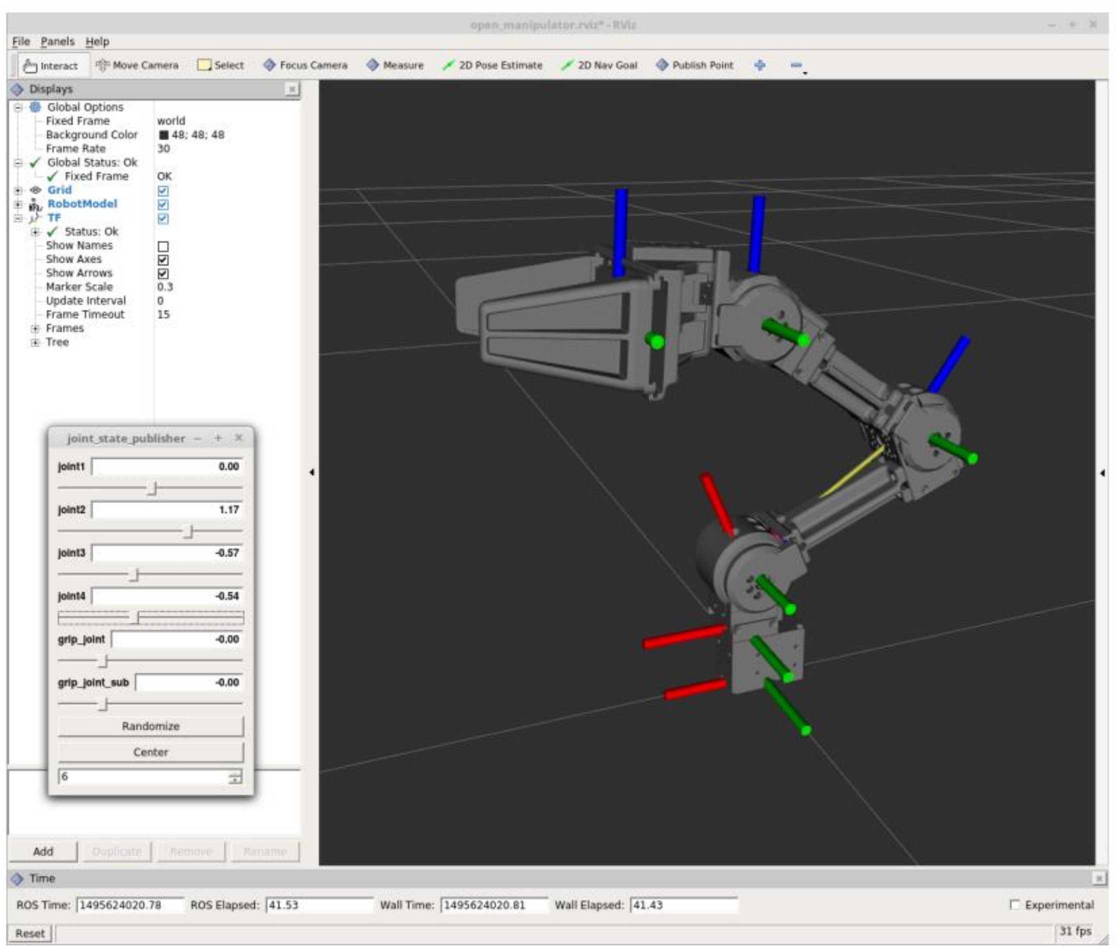
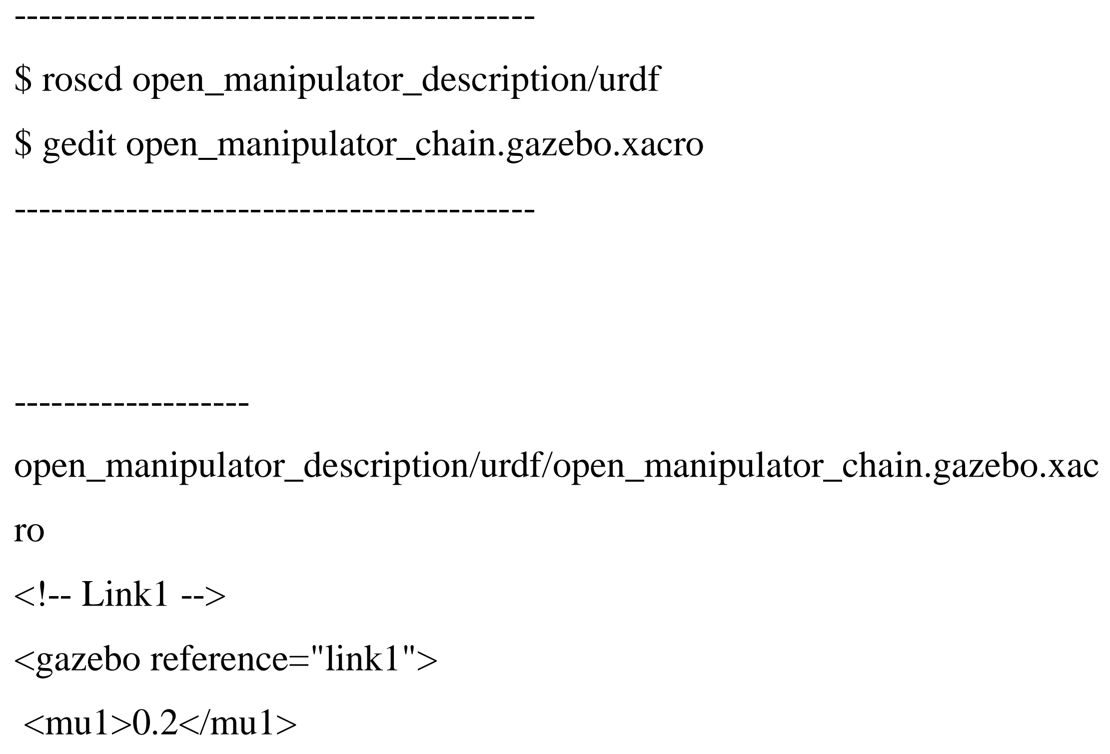


Рис. 211 Ланцюжок OpenManipulator зі спільними значеннями змінюється за допомогою GUI

Gazebo - це 3D-симулятор робота, який є незалежним програмним забезпеченням, що підтримує ROS. Це дозволяє проектувати роботів, тестувати алгоритми, проводити регресійний аналіз, навчати штучного інтелекту і т. д., а також підтримує різні роботи, і багато користувачів ROS використовують його для моделювання роботів. Оскільки RViz є інструментом візуалізації, він

не може отримати фізичні зміни (інерцію, крутний момент, зіткнення і т. д.) робота або навколишнього середовища в режимі реального часу. З іншого боку, Gazebo має перевагу моніторингу таких даних в режимі реального часу. Це дозволяє запобігти несправності робота і жертви під час експерименту.

URDF, створений у попередньому розділі, був призначений для візуалізації за допомогою RViz. Давати додамо кілька тегів, щоб використовувати його в середовищі моделювання Альтанки. Теги для моделювання Gazebo зберігаються у файлі open_manipulator_chain.gazebo.xacro. Давайте подивимося.



```
----------------------------------------
$ roscd open_manipulator_description/urdf
$ gedit open_manipulator_chain.gazebo.xacro
----------------------------------------



------------------
open_manipulator_description/urdf/open_manipulator_chain.gazebo.xac
ro
<!-- Link1 -->
<gazebo reference="link1">
 <mu1>0.2</mu1>
```

```
 <mu2>0.2</mu2>
 <material>Gazebo/Grey</material>
</gazebo>
-----------------------------------------------------------------------
```

Інформація про колір та інерцію має важливе значення для налаштування посилання, яка буде використовуватися в Gazebo. Оскільки інформація про інерцію включена в створений раніше файл URDF, необхідно налаштувати тільки колір. Крім того, для відкритої динаміки можуть бути встановлені сили тяжіння, демпфірування і тертя

Движок (ODE), фізичний движок, підтримуваний Gazebo. У наведеному вище файлі як приклад заданий тільки коефіцієнт тертя. Існує також параметр для спільної інформації, який не згадувати.

Таблиця 33

<gazebo> tag

<gazebo>:	Налаштування параметрів для моделювання Gazebo
<mu1>, <mu2>:	Установка коефіцієнта тертя
<material>:	Настройка кольору посилання

```
-------------------------------------------------------------------------------
<!-- ros_control plugin -->
```

```
<gazebo>
 <plugin                                    name="gazebo_ros_control"
filename="libgazebo_ros_control.so">
 <robotNamespace>/open_manipulator_chain</robotNamespace>

<robotSimType>gazebo_ros_control/DefaultRobotHWSim</robotSimT
ype>
 </plugin>
</gazebo>
--------------------------------------------------------------------------------
```

Gazebo plugin42 - це інструмент для підтримки стану і управління датчиком і двигуном робота, створений URDF або SDF за допомогою ROS-повідомлення та сервісного зв'язку. Плагін підтримує різні датчики, такі як камера, лазер, інерціальний навігаційний датчик, управління мобільною платформою, таке як диференціал, привід рульового управління заносом, паралельний рух і ROS-управління. Відкрита ланцюг маніпулятора використовує інтерфейс управління розташування з'єднання і включає бібліотеку модулів, що підключаються за замовчуванням.

Таблиця 34

<gazebo> tag

<gazebo>:	Налаштування параметрів для моделювання Gazebo

<plugin>:	Інструмент для контролю стану датчиків і роботів
<robotNamespace>:	Встановіть ім'я робота для використання в Gazebo
<robotSimType>:	Установка імені плагіна для інтерфейсу моделювання роботів.

Наведений вище код буде повторений, тому ознайомтеся з наступним вихідним кодом.

```
----------------
open_manipulator_description/urdf/open_manipulator_chain.gazebo.xacro
<?xml version="1.0"?>
<robot>
 <!-- World -->
 <gazebo reference="world">
 </gazebo>
 <!-- Link1 -->
 <gazebo reference="link1">
 <mu1>0.2</mu1>
 <mu2>0.2</mu2>
 <material>Gazebo/Grey</material>
 </gazebo>
```

```
<!-- Link2 -->
<gazebo reference="link2">
<mu1>0.2</mu1>
<mu2>0.2</mu2>
<material>Gazebo/Grey</material>
</gazebo>
<!-- Link3 -->
<gazebo reference="link3">
<mu1>0.2</mu1>
<mu2>0.2</mu2>
<material>Gazebo/Grey</material>
</gazebo>
<!-- Link4 -->
<gazebo reference="link4">
<mu1>0.2</mu1>
<mu2>0.2</mu2>
<material>Gazebo/Grey</material>
</gazebo>
<!-- Link5 -->
<gazebo reference="link5">
<mu1>0.2</mu1>
<mu2>0.2</mu2>
<material>Gazebo/Grey</material>
```

```
</gazebo>
<!-- grip_link -->
<gazebo reference="grip_link">
<mu1>0.2</mu1>
<mu2>0.2</mu2>
<material>Gazebo/Grey</material>
</gazebo>
<!-- grip_link_sub -->
<gazebo reference="grip_link_sub">
<mu1>0.2</mu1>
<mu2>0.2</mu2>
<material>Gazebo/Grey</material>
</gazebo>
<!-- ros_control plugin -->
<gazebo>
<plugin name="gazebo_ros_control"
filename="libgazebo_ros_control.so">
<robotNamespace>/open_manipulator_chain</robotNamespace>

<robotSimType>gazebo_ros_control/DefaultRobotHWSim</robotSimT
ype>
</plugin>
</gazebo>
```

```
</robot>
--------------------------------------------------------------------------------------
```

Файл ' open_manipulator_chain.gazebo.xacro ' використовується для налаштування параметрів моделювання Gazebo і включений у файл ' open_manipulator_chain.xacro’. Тепер використовуйте завершений URDF для відображення OpenManipulator chain в Gazebo.

```
--------------------------------------------------------
$ roscd open_manipulator_gazebo/launch
$ ls
open_manipulator_gazebo.launch       → File to launch Gazebo
position_controller.launch           → File to launch ROS-CONTROL
--------------------------------------------------------
```

Папка запуску, яка є підпапкою папки 'open_manipulator_gazebo', містить файл запуску для запуску Gazebo і виконання ROS-Control. Відкрийте ці файли запуску Gazebo і подивіться, які вузли включені.

```
--------------------------------------------------------
$ roscd open_manipulator_gazebo/launch
$ gedit open_manipulator_gazebo.launch
--------------------------------------------------------
```

```
------------------
open_manipulator_gazebo/launch/open_manipulator_gazebo.launch
<?xml version="1.0" ?>
<launch>
 <!-- These are the arguments you can pass this launch file, for example
paused:=true -->
 <arg name="paused" default="false"/>
 <arg name="use_sim_time" default="true"/>
 <arg name="gui" default="true"/>
 <arg name="headless" default="false"/>
 <arg name="debug" default="false"/>
 <!-- We resume the logic in empty_world.launch, changing only the
name of the world to be
launched -->
 <include file="$(find gazebo_ros)/launch/empty_world.launch">
 <arg name="world_name" value="$(find
open_manipulator_gazebo)/world/empty.world"/>
 <arg name="debug" value="$(arg debug)" />
 <arg name="gui" value="$(arg gui)" />
 <arg name="paused" value="$(arg paused)"/>
 <arg name="use_sim_time" value="$(arg use_sim_time)"/>
 <arg name="headless" value="$(arg headless)"/>
```

```
</include>
<!-- Load the URDF into the ROS Parameter Server -->
<param name="robot_description"
command="$(find xacro)/xacro.py '$(find
open_manipulator_description)/urdf/open_manipulator_ch
ain.xacro'"/>
<!-- Run a python script to the send a service call to gazebo_ros to spawn
a URDF robot -->
<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
respawn="false" output="screen"
args="-urdf -model open_manipulator_chain -z 0.0 -param
robot_description"/>
<!-- ros_control robotis manipulator launch file -->
<include file="$(find
open_manipulator_gazebo)/launch/position_controller.launch"/>
</launch>
-----------------------------------------------------------------
```

Наведений вище файл запуску включає в себе 'empty_world.файл запуску, вузол spawn_model і файл position_controller.launch. ‘empty_world.launch’ містить вузли, які виконуються Gazebo, так що ви можете налаштувати середовище моделювання, Графічний інтерфейс і час. Середовище моделювання

Gazebo підтримує файли, створені у форматі SDF. Вузол 'spawn_model' відповідає за виклик робота на основі URDF, а 'position_controller.launch ' відповідає за налаштування і виконання ROS-Control.

Тепер, якщо ви запустите Gazebo, ввівши наступну команду в термінал, ви побачите OpenManipulator Chain в просторі моделювання Gazebo, як показано на рис. 212.

```
$ roslaunch open_manipulator_gazebo open_manipulator_gazebo.launch
```

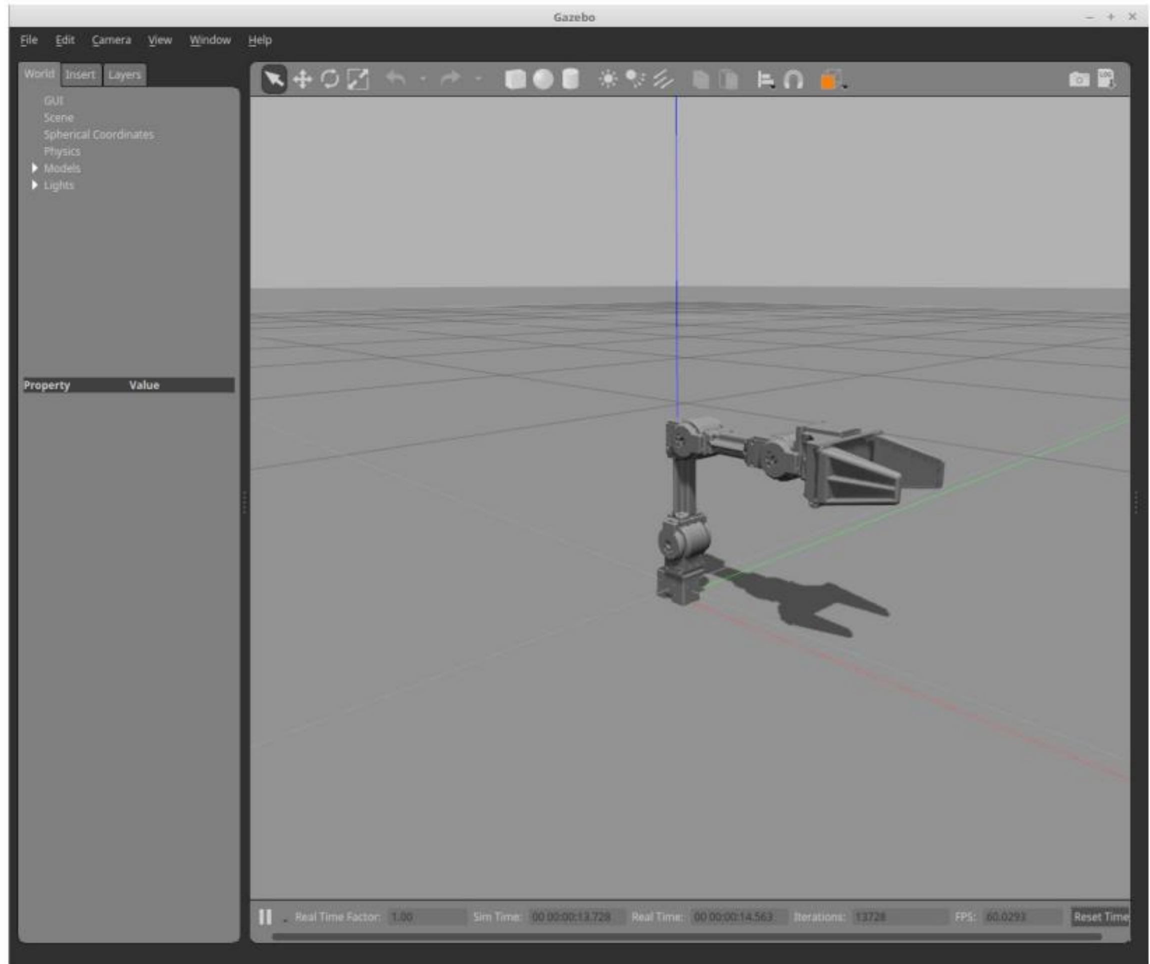


Рис. 212 Ланцюжок OpenManipulator простору моделювання Gazebo

Потім відкрийте нове вікно терміналу та перевірте список тем.

```
----------------------------------------------------------
$ rostopic list
/clock
/gazebo/link_states
/gazebo/model_states
```

```
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/joint_states
/open_manipulator_chain/grip_joint_position/command
/open_manipulator_chain/grip_joint_sub_position/command
/open_manipulator_chain/joint1_position/command
/open_manipulator_chain/joint2_position/command
/open_manipulator_chain/joint3_position/command
/open_manipulator_chain/joint4_position/command
/open_manipulator_chain/joint_states
/rosout
/rosout_agg
----------------------------------------------------------
```

Якщо подивитися на список тем, тобто теми з простором імен '/gazebo' і теми з простором імен ‘/open_manipulator_chain'. Ми можемо використовувати Ros-Control для перевірки та управління станом робота на Gazebo, використовуючи теми з простором імен '/open_manipulator_chain'. Давайте перемістимо робота, використовуючи наступну команду.

```

----------------------------------------------------------
```

```
$ rostopic pub /open_manipulator_chain/joint2_position/command std_msgs/Float64 "data: 1.0" --once
-----------------------------------------------------------
```

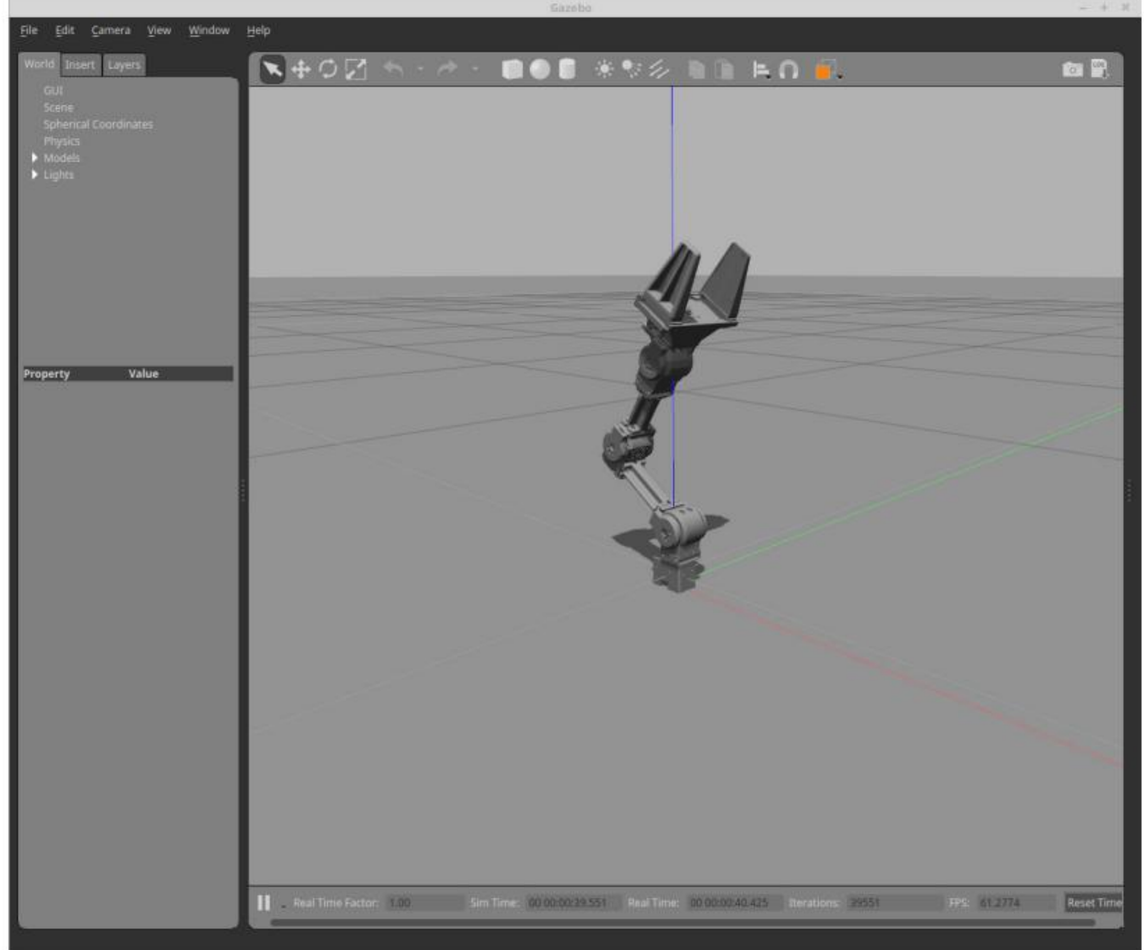


Рис. 213 Ланцюжок OpenManipulator, керований зв'язком з ROS-CONTROL

За допомогою простої передачі повідомлень ви можете бачити, що друге з'єднання OpenManipulator chain рухається, як показано на рис. 213.

13.3. MoveIt!

Move It! - це інтегрована бібліотека для маніпуляторів, яка надає безліч функцій, включаючи швидкий зворотний кінематичний аналіз для планування руху, передові алгоритми маніпуляції, ручне управління роботом, динаміку, контролери і планування руху. Він також простий у використанні без попереднього знання маніпулятора, тому що графічний інтерфейс пропонується для допомоги з різними настройками, необхідними для використання Move It!. Це інструмент, який багато користувачів ROS люблять, тому що він дозволяє візуальну зворотний зв'язок з допомогою RViz. Давайте коротко розглянемо структуру MOVEit!, а потім створимо MOVEit! посилка до управляйте OpenManipulator chain.

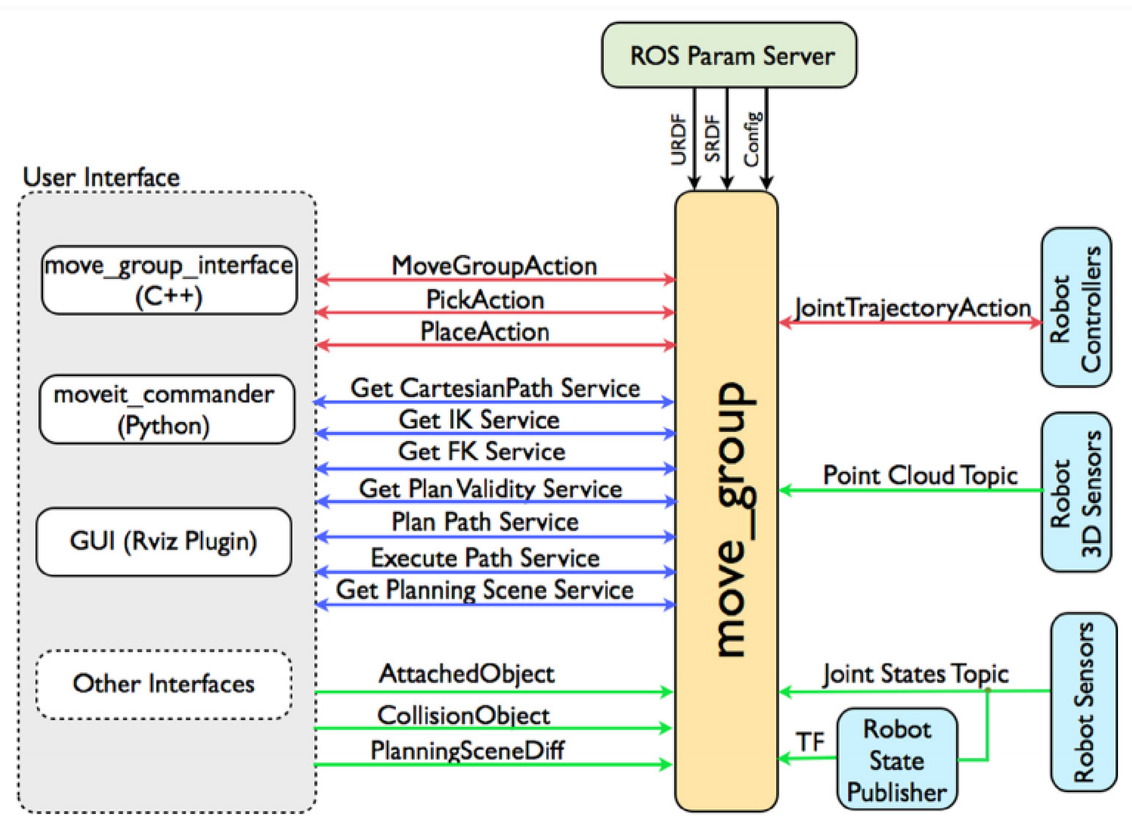


Рис. 214 Схема зв'язку з вузлом move_group

Як показано на рис. 214, вузол "move_group" може обмінюватися командами з користувачем з допомогою дій і служб ROS. Ворушилися! надає різні користувальницькі інтерфейси і побудував сервіс, який дозволяє більшій кількості користувачів спілкуватися з вузлом "move_group" за допомогою інтерфейсу "move_group" в Мова C++ або за допомогою 'move_commander' на мові Python і 'Motion Planning plugin to RViz'.

Вузол "move_group' отримує інформацію про роботу від URDF, семантичного робота Формат опису (RDF) 44, і перемістіть його!

Конфігурація. URDF, який ви створили, буде використовуватися в той час як SRDF і перемістити його! Конфігурація буде створена за допомогою помічника з налаштування 45, наданого Move It!.

Вузол "move_group' забезпечує стан і контроль робота і його оточення з допомогою тем і дій ROS. Спільний стан використовує повідомлення "sensor_msg / JointStates", інформація про перетворення використовує бібліотеку tf, а контролер використовує інтерфейс "FollowTrajectoryAction" для інформування користувача про стан робота. Крім того, користувачеві надається інформація про навколишнє середовище, в якому працює робот, і про стан робота через "сцену планування".

move_group надає функцію плагіна для своєї розширюваності і надає можливість застосовувати різні функції (управління, генерація шляху, динаміка і т. д.) до робота користувача через бібліотеку з відкритим вихідним кодом. Плагін, вбудований в Move It! це відмінна бібліотека, яка вже була доведено багатьом людям, і ряд недавно розроблених бібліотек з відкритим вихідним кодом код також скоро буде доступний. Open Motion Planning Library (OMPL), кінематична та динамічна бібліотека

(KDL) і гнучка бібліотека зіткнень (FCL) відносяться до категорії цих бібліотек.

Щоб створити MoveIt! пакет для маніпуляторів, вам знадобиться URDF, SDF, і MoveIt! Файли конфігурації. Setup Assistant,

наданий MoveIt! створює SRDF на основі URDF і переміщує його! Конфігураційні файли для MoveIt! пакет. Давайте дізнаємося, як створити MoveIt! пакет за допомогою MoveIt! Setup Assistant для OpenManipulator Chain.

Введіть наступну команду в термінал і запустіть MoveIt! Setup Assistant.

$ roslaunch moveit_setup_assistant setup_assistant.launch

Рис. 215 Стартова сторінка помічника з налаштування MoveIt!

Рис. 215 - це перша сторінка, яку ви побачите при запуску програми " MOVEit! Помічник з налаштування". На цьому екрані ви можете побачити репрезентативний символ ROS, Черепаху, праворуч, і ви можете вибрати, чи створювати новий пакет або змінювати існуючий пакет на лівій сторінці. Оскільки нам потрібно створити

новий пакет, давайте натиснемо на кнопку [Create New MOVEit Configuration Package] кнопка.

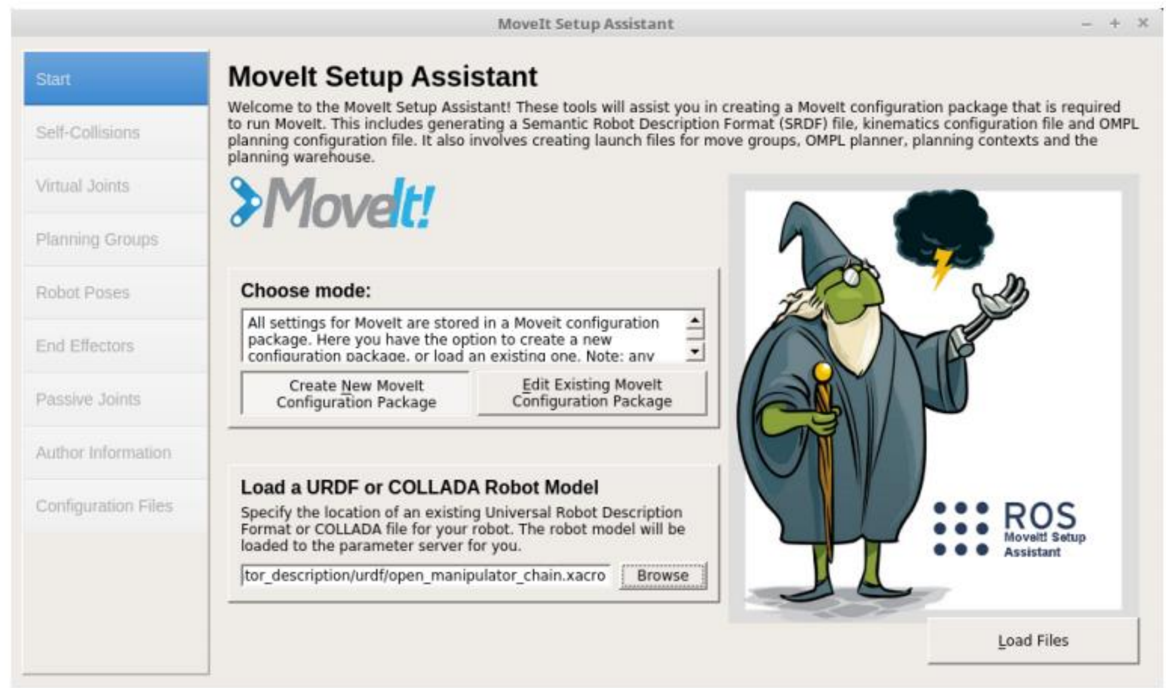


Рис. 216 Стартова сторінка помічника з налаштування MoveIt!

MoveIt! Setup Assistant створює файл RDF з додатковими настройками на основі моделі робота, зберігається у файлі URDF або файлі COLLADA. Натисніть кнопку [Огляд], показану на рис. 216, відкрийте файл open_manipulator_chain.xacro, створений раніше, і натисніть кнопку [Завантажити файли].

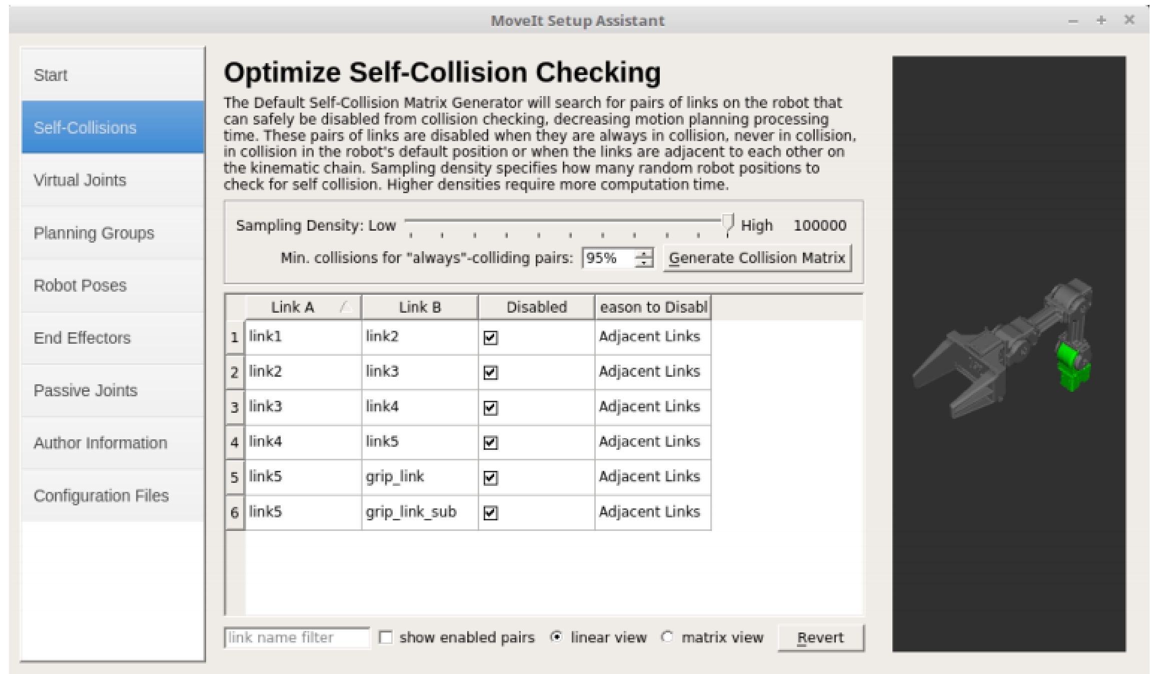


Рис. 217 Сторінка Self-Collisions помічника з налаштувань MoveIt!

Якщо ви успішно завантажили файл, перейдіть на сторінку 'Self-Collision'. Ця сторінка дозволяє визначити щільність вибірки, необхідну для побудови "матриці самоколлізій", і, при необхідності, Користувач може визначити діапазон колізій між ланками, складовими робота, як показано на рис. 217 вище. Чим вище щільність вибірки, тим більше обчислень потрібно для запобігання зіткнення ланок в різних позах робота. Встановіть бажану щільність вибірки і натисніть кнопку [Створити матрицю зіткнень]. Значення за замовчуванням дорівнює "10 000'.

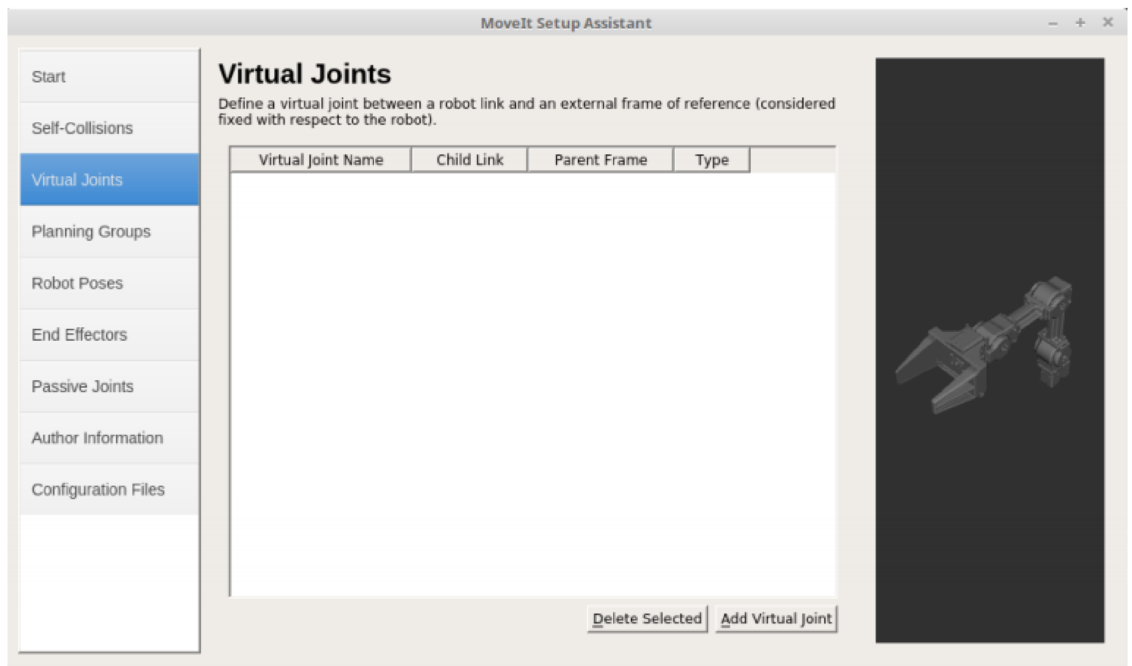


Рис. 218 Сторінка віртуальних з'єднань помічника з налаштувань MoveIt!

Сторінка ‘Virtual Joints’ надає віртуальне з'єднання між основою маніпулятора і опорною системою координат. Наприклад, якщо до TurtleBot3 Waffle або Waffle Pi прикріплена ‘OpenManipulator Chain’, то ступінь свободи її може бути забезпечена OpenManipulator Chain through через Virtual Joint. Оскільки база є нерухомою, вам не потрібно налаштовувати віртуальне з'єднання, як показано на рис. 218.

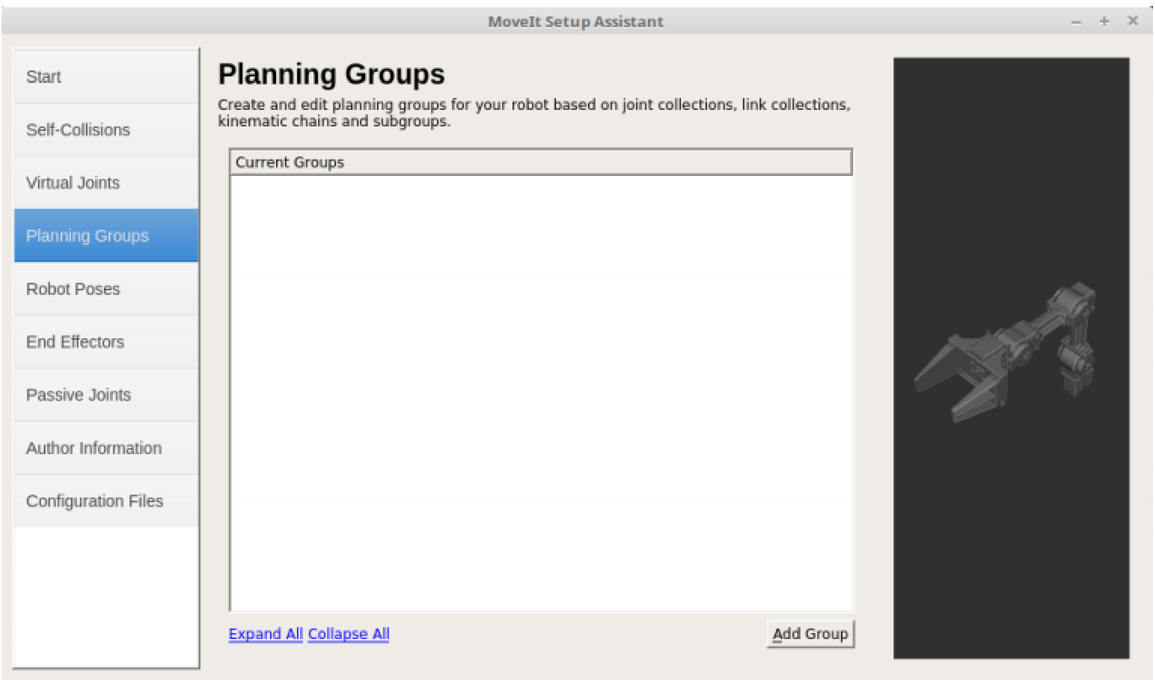


Рис. 219 Сторінка Planning Groups помічника з налаштувань MoveIt!

MoveIt! забезпечує планування руху для кожного з маніпуляторів, розділених на групи, які ви можете встановити на сторінці "групи планування". OpenManipulator Chain складається з чотирьох шарнірів і одного захоплення, тому встановлюється група важелів і група захоплень. Натискання кнопки [Додати групу] у правому нижньому кутку рис. 219 змінить вікно, як показано на рис. 220.

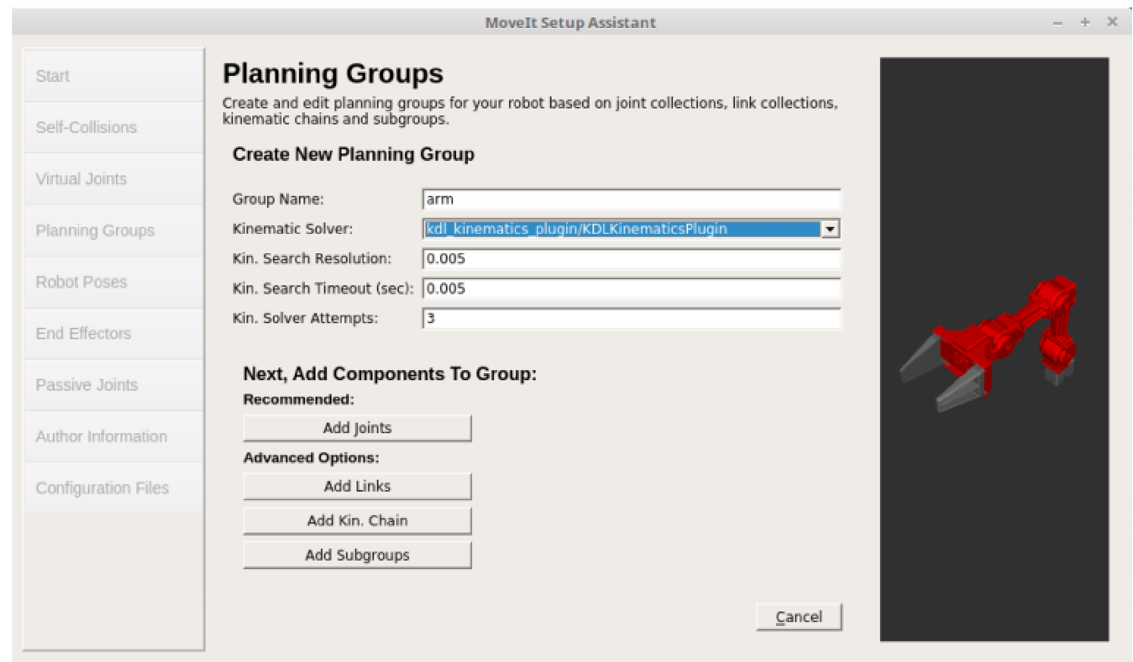


Рис. 220 Створіть нову сторінку групи на сторінці Planning Groups page

На цьому кроці ви можете вказати ім'я групи і вибрати потрібний плагін кінематичного аналізу (Кінематика нижче). Встановіть ім'я групи як arm і виберіть потрібний плагін інтерпретації кінематики. Оскільки маніпулятор буде розділений на групи суглобів, натисніть кнопку [Додати суглоб]. Якщо вікно зміниться, як показано на рис. 221, виберіть з'єднання 1~4 і натисніть кнопку [Зберегти], щоб створити групу, як показано на рис. 222.

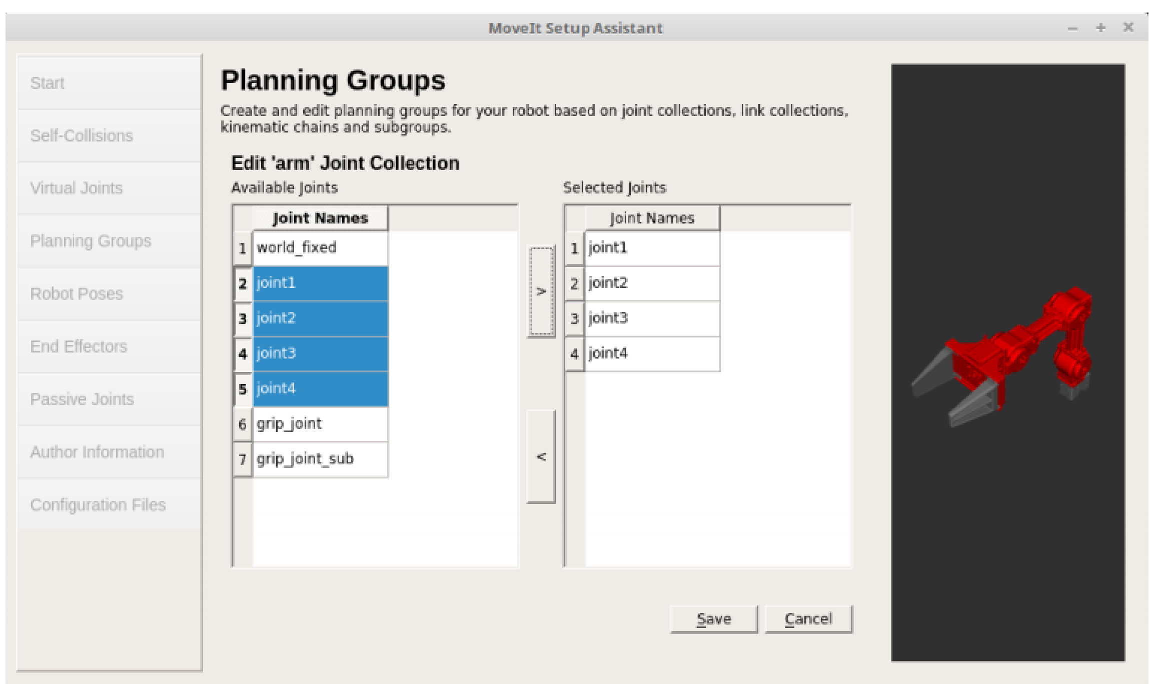


Рис. 221 Створіть нове групове вікно на Planning Groups page

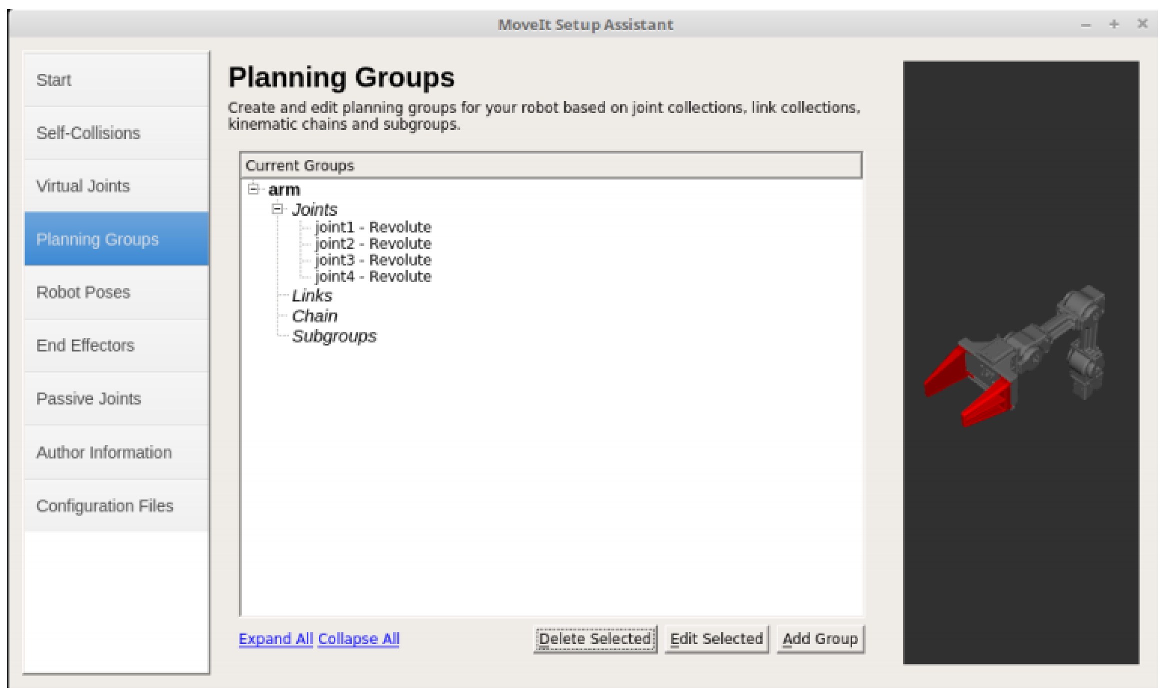


Рис. 222 Arm group створена на Planning Groups page

Якщо ви створили групу ‘arm’, давайте створимо групу ‘gripper’ точно так само, як це. Лінійний захоплювач, керований одним приводом, не підтримується модулем кінематичного аналізу. Тож при створенні групи захоплень встановіть модуль аналізу кінематики в положення "ні'. І, як і група arm, виберіть суглоби 'grip_joint' і 'grip_joint_sub' на сторінці 'Додати суглоби'. Як тільки група захоплення буде завершена, ви побачите групу ‘arm’ і групу ‘gripper’ , як показано на рис. 223

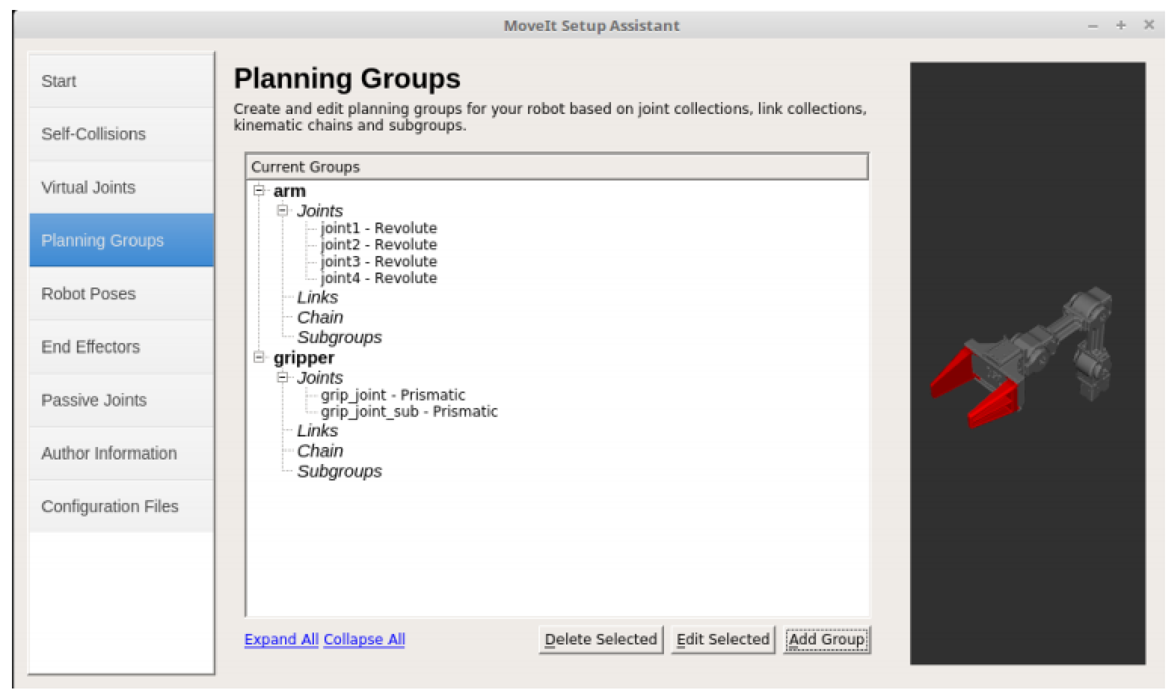


Рис. 223 Arm group та gripper groups створені на Planning Groups page

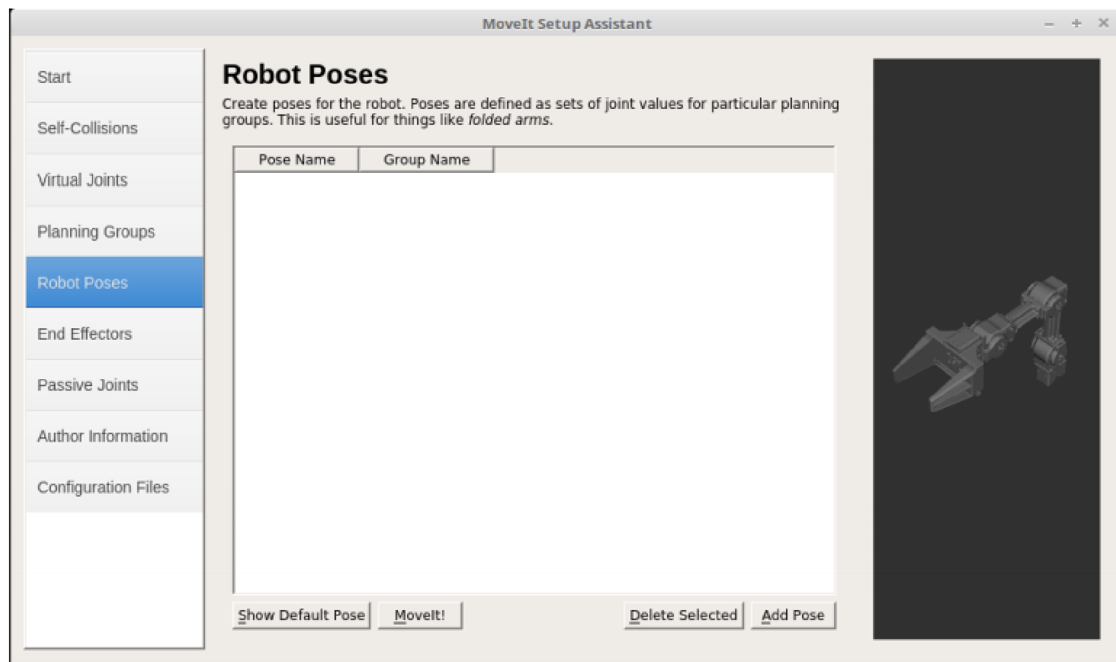


Рис. 224 Сторінка Robot Poses помічника з налаштувань MoveIt!

Сторінка "пози робота" дозволяє створювати і реєструвати конкретні пози робота. Натисніть кнопку [Додати паузу] в правому нижньому кутку рис. 224, щоб зареєструвати положення всіх двигунів під кутом нуль градусів. Натиснувши на кнопку, ви потрапите у вікно створення поста , як показано на рис. 225, де ви встановите всі спільні значення рівними 0.0 і запишете ім'я пози.

Рис. 225 Вікно генерації поз на сторінці Robot Poses

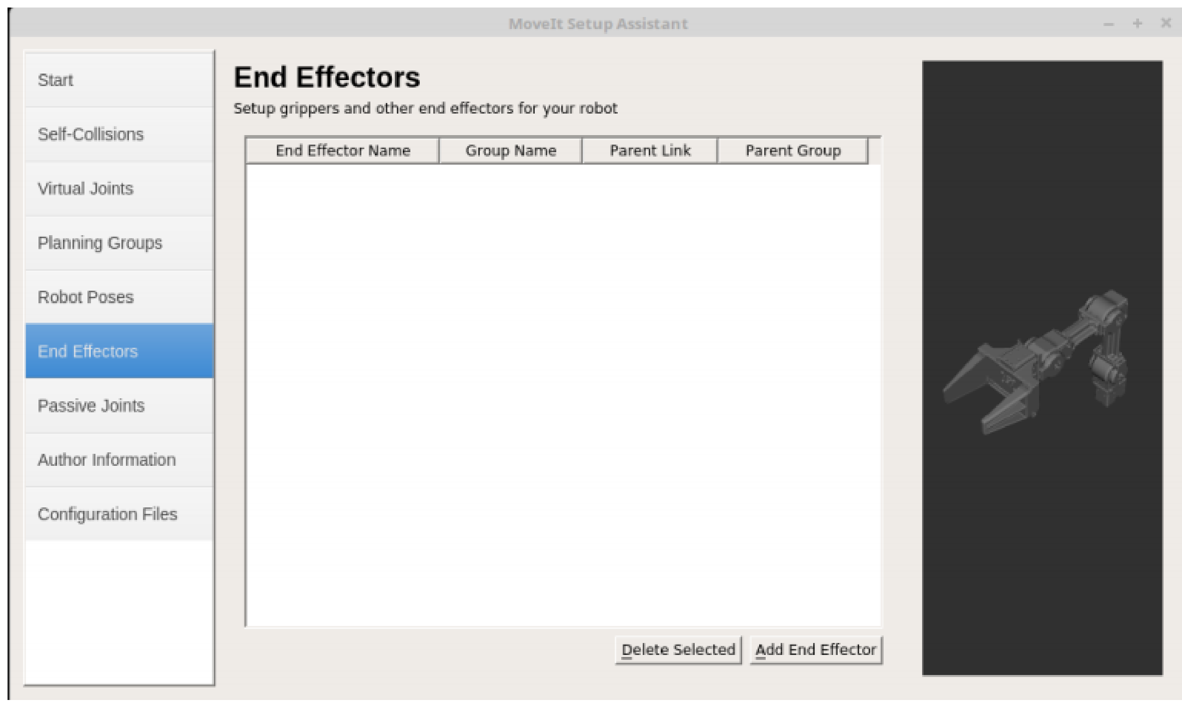


Рис. 226 Сторінка кінцевих ефектів помічника з налаштувань MoveIt!

Сторінка "кінцеві ефектори' може зареєструвати кінцевий ефектор маніпулятора. Натисніть кнопку [Додати кінцевий ефектор] в правому нижньому кутку, як показано на рис. 226, щоб зареєструвати лінійний захоплення, який має OpenManipulator Chain.

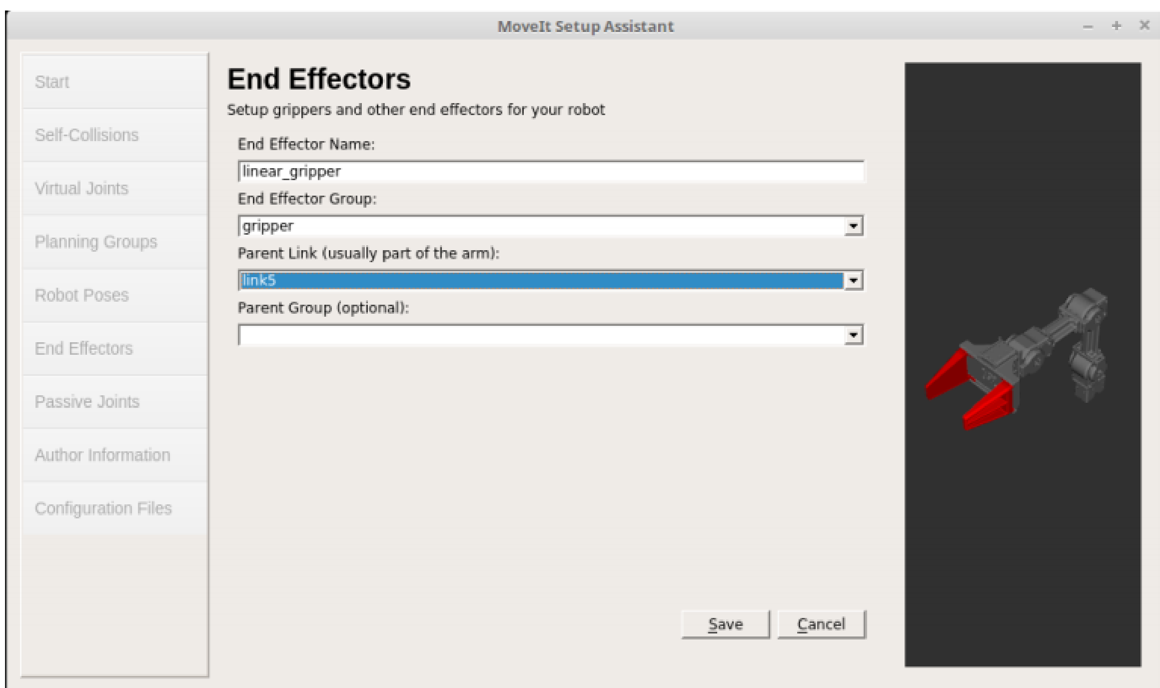


Рис. 227 Вікно кінцевих ефектів на сторінці End Effectors

Запишіть "ім'я кінцевого ефектора", як показано на рис. 227, і виберіть "захоплення" в групі, створеної на сторінці "групи планування". Коли ви перевіряєте URDF, захоплення має п'яту ланку в якості батьківської ланки.

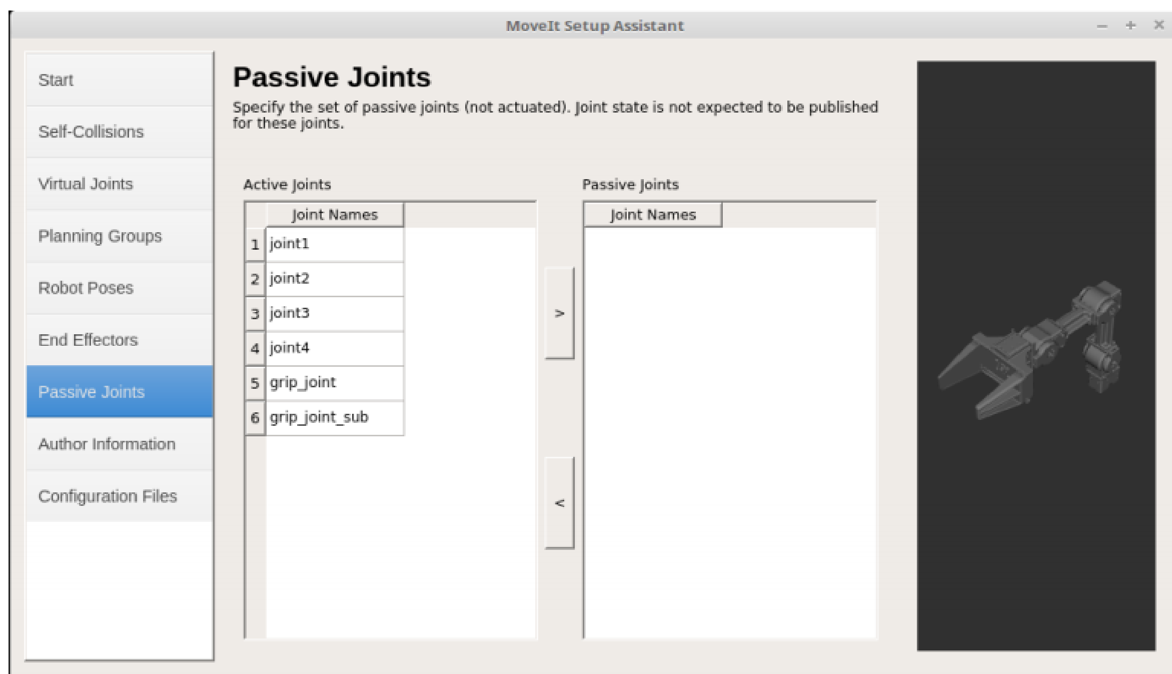


Рис. 228 Сторінка пасивних з'єднань помічника з налаштувань MoveIt!

Сторінка "Пасивні з'єднання" дозволяє вказати з'єднання, виключені з планування руху.

У ланцюзі OpenManipulator немає пасивного з'єднання, тому давайте рухатися далі, не вносячи ніяких змін, як показано на рис. 228.

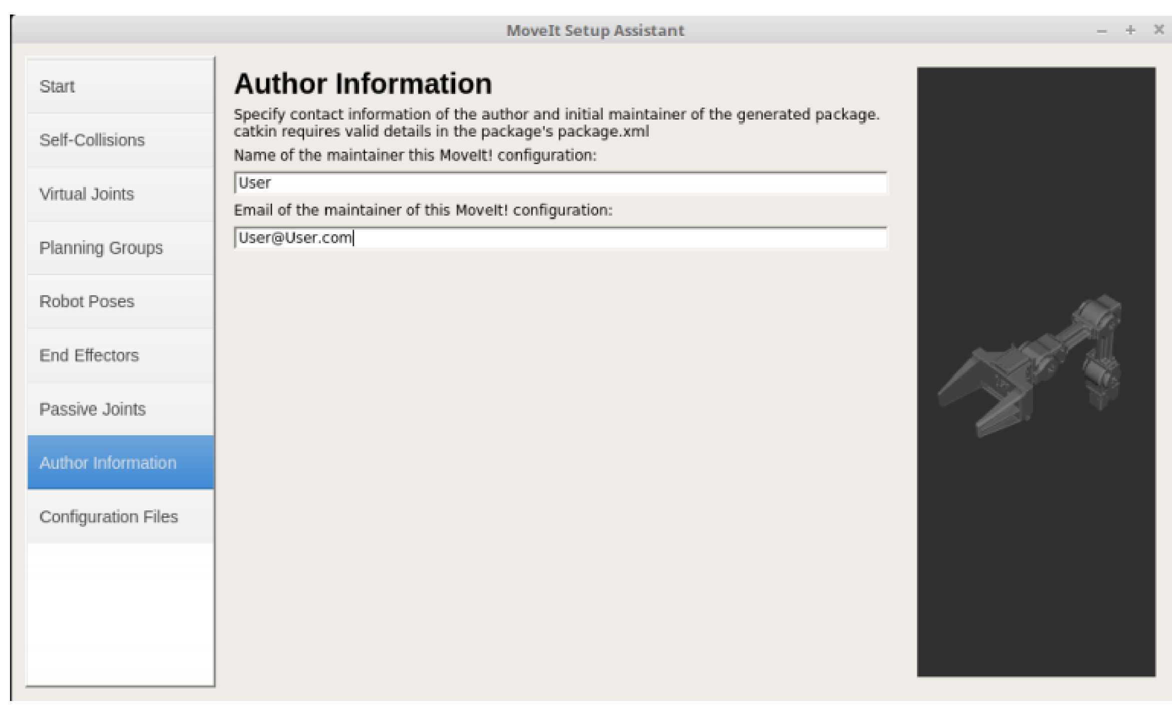
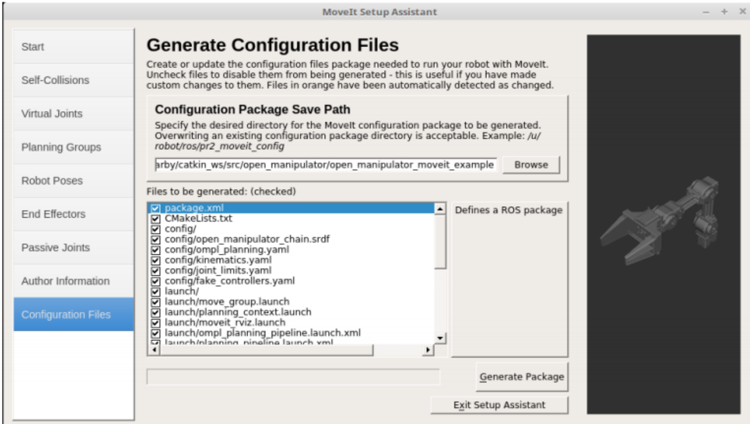


Рис. 229 Сторінка інформації про автора помічника з налаштувань MoveIt!

На сторінці "Інформація про автора" введіть ім'я та адресу електронної пошти творця пакета. Заповніть прогалини, як показано на рис. 229.



230 Сторінка конфігурації файлів помічника з налаштувань MoveIt!

Як тільки всі налаштування будуть завершені, ви можете закінчити сторінку "файли конфігурації". Натисніть кнопку [Огляд] у верхній частині рис. 230, щоб створити папку "open_manipulator_moveit_example" в папці "open_manipulator", і натисніть кнопку [Generate Package] в правому нижньому кутку, щоб створити папку конфігурації і папку запуску, що містить виконувані файли конфігурації для MoveIt!.

Перевірте згенерований пакет і запустіть демо-версію.

```
--------------------------------------------
$ cd ~/catkin_ws/src/open_manipulator/open_manipulator_moveit_example
$ ls
Config              → YAML and SRDF files for MoveIt! Configuration
launch              → Launch file
.setup_assistant    → Package information generated by the setup assistant
CMakeLists.txt      → CMake build system input file
package.xml         → Defining package properties

$ cd ~/catkin_ws && catkin_make
$ roslaunch open_manipulator_moveit_example demo.launch
```

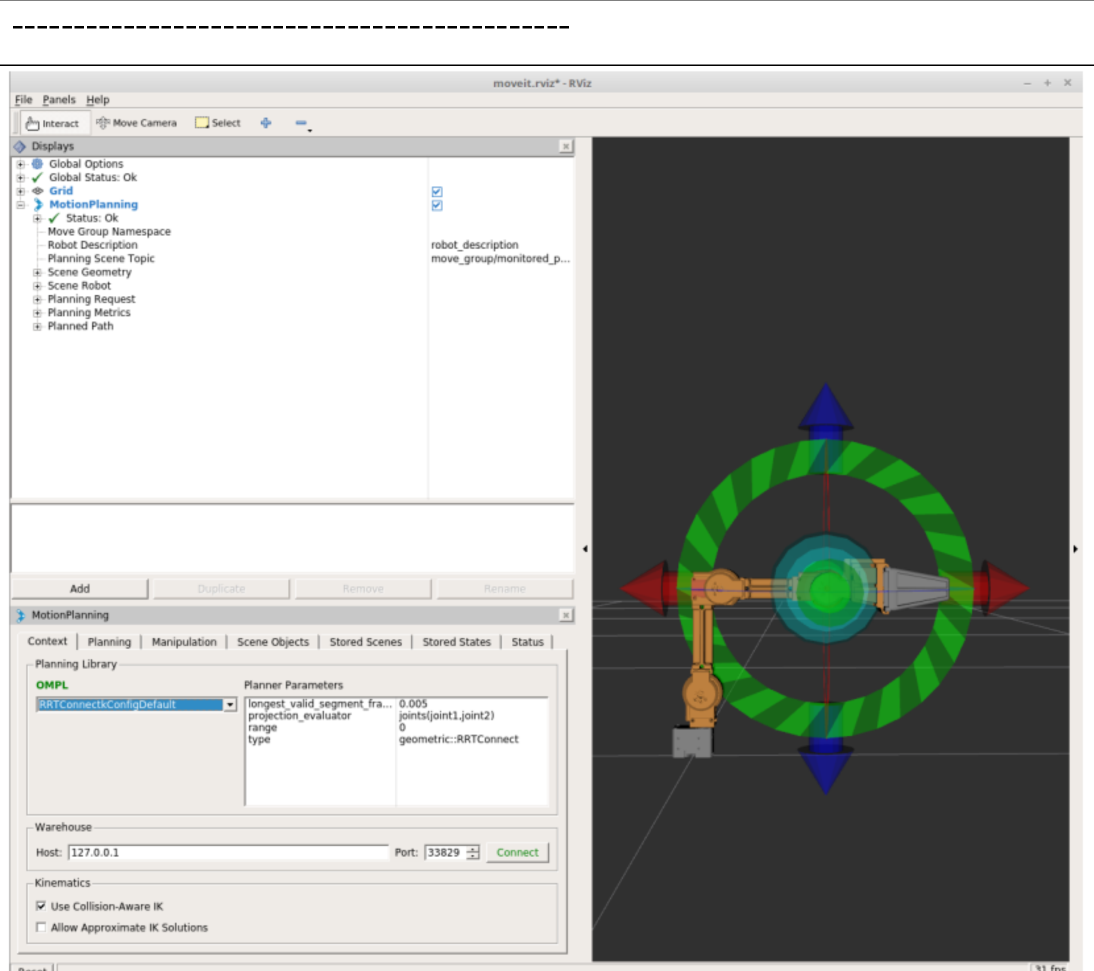


Рис. 231 Демо MoveIt! Rviz

При запуску демо-версії ви можете побачити Відкритий ланцюжок маніпуляторів через вікно RViz, як показано на рис. 231. На контекстній сторінці командного вікна планування руху, розташованого у лівому нижньому кутку, можна вибрати бібліотеку планування руху, підтримувану OMPL. Вибрати 'RRTConnectkConfigDefault' і перейдіть на сторінку планування.

Спочатку координати кінцевої точки маніпулятора представляються у вигляді X, Y, Z, а значення обертання - у вигляді θ(Roll), ϕ(Pitch), і ψ(Yaw). Однак, так як OpenManipulator Chain має тільки чотири з'єднання, кінцеві точки будуть мати тільки ступеня свободи X, Z і Осі тангажа (залишилася одна ступінь свободи-це обертання осі нишпорення першого шарніра). Запам'ятайте це і перемістіть кінцеву точку в потрібні координати.

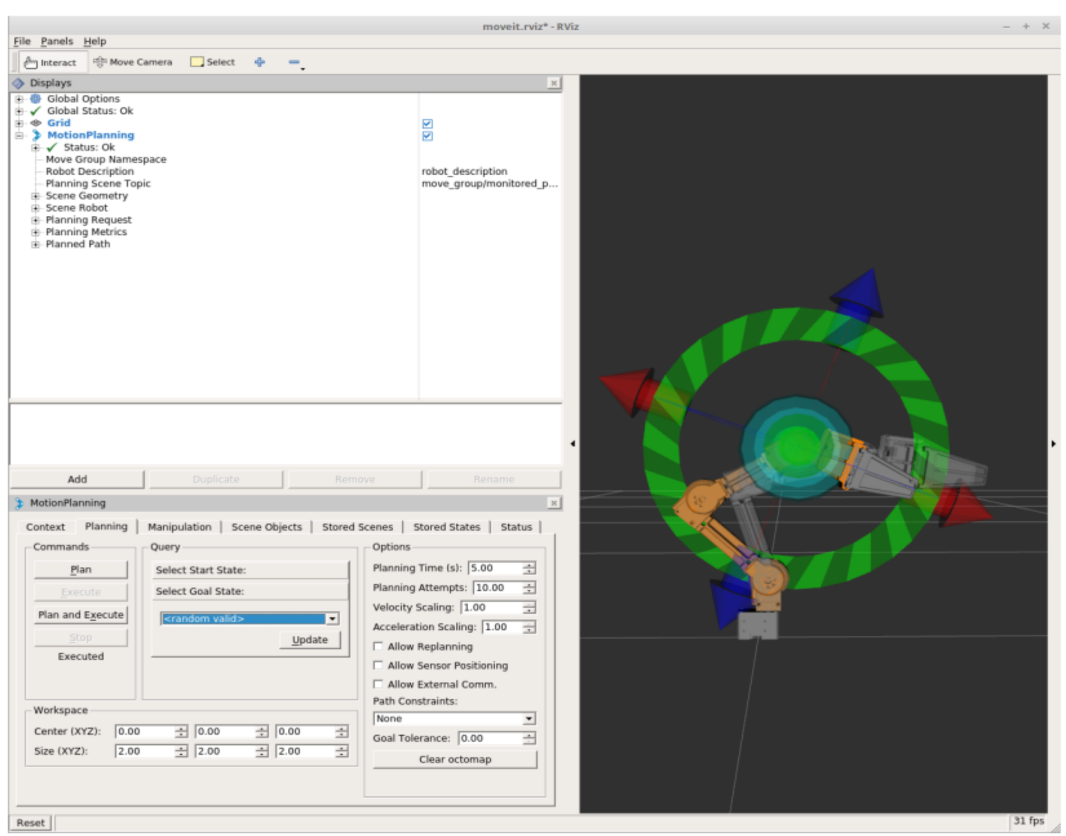


Рис. 232 Планування руху маніпулятора за допомогою демо-версії MoveIt! RViz

Якщо ви перемістилися в потрібне місце, натисніть кнопку [Plan & Execute] на сторінці планування, щоб перевірити переміщення.

Позначення цільової пози (положення + орієнтація) маніпулятора

У цьому прикладі ви можете вказати цільову позу маніпулятора, ввівши положення і орієнтацію на RViz. На додаток до цього, ви також можете використовувати метод вказівки тільки позиції, переміщаючи маркер зеленої сфери (інтерактивний маркер). Для цього в нижній частині кінематики напишіть " position_only_ik: true" .файл yaml в папці config.

```
$ roscd open_manipulator_moveit/config/
$ gedit kinematics.yaml

arm:

kinematics_solver: kdl_kinematics_plugin/KDLKinematicsPlugin
 kinematics_solver_search_resolution: 0.005
 kinematics_solver_timeout: 0.005
 kinematics_solver_attempts: 3
 position_only_ik: true
----------------------------------------------------------------------------------
```

OpenManipulator надає приклад пакету MoveIt!. Перевірте папку пакета 'open_manipulator_moveit' в папці 'open_manipulator'.

```
--------------------------------------------------------------------------------
$ roscd open_manipulator_moveit
$ ls
Config → YAML and SRDF files for the move_group setup
include → Trajectory filter header file
launch → Launch file
src → Trajectory filter source code file
.setup_assistant → Package information produced by the setup assistant
CMakeLists.txt → CMake build system input file
package.xml → Defining package properties
planning_request_adapters_plugin_description.xml → Plugin set-up file
--------------------------------------------------------------------------------
```

Ви можете побачити більше файлів, ніж ті, які були створені за допомогою помічника установки вище. Rapidly-exploring Random Tree (RRT), зазвичай відоме як алгоритм генерації шляху, випадковим чином вибирає шлях, щоб знайти правильний шлях для переміщення з поточного місця розташування в цільове місце розташування, і повертає результат. Кожне повертається спільне значення являє собою необхідні дані пози для переміщення в цільову позицію. Однак для того, щоб перейти від n-й пози до (n+1)-й позі, необхідні спільні значення, отримані за менший час вибірки. Таким чином, "industrial_trajectory_filters' підтримується Для цієї мети використовуються ROS-INDUSTRIAL.

Як застосовувати industrial_trajectory_filters

1. Завантажте пакет ROS-INDUSTRIAL CORE package

$ git clone https://github.com/ros-industrial/industrial_core.git

2. Перевірте 'industrial_trajectory_filters' у завантаженому пакеті ros-industrial

```
$ cd ~/catkin_ws/src/industrial_core/industrial_trajectory_filters/
```

3. Копії ' planning_request_adapters_plugin_description.xml папки', 'src' і 'include' знаходяться в папці 'industrial_trajectory_filters' і вставляються в створений раніше пакет moveit

```
$ cp
r planning_request_adapters_plugin_description.xml src include
~/catkin_ws/src/ open_manipulator/open_manipulator_moveit_example/
```

4. Створіть ' smoothing_filter_params.yaml' в папці config і вкажіть коефіцієнт

```
$ cd
~/catkin_ws/src/open_manipulator/open_manipulator_moveit_example/
config
$ gedit smoothing_filter_params.yaml
smoothing_filter_name: /move_group/smoothing_5_coef
smoothing_5_coef:
            - 0.25
```

```
- 0.50
- 1.00
- 0.50
- 0.25
```

5. Відкрийте ' ompl_planning_pipeline.launch.xml' в папці запуску і додайте наступний фільтр в planning_adapters:

```
$ cd ~/catkin_ws/src/open_manipulator/open_manipulator_moveit_example/launch
$ gedit ompl_planning_pipeline.launch.xml
industrial_trajectory_filters/UniformSampleFilter
industrial_trajectory_filters/AddSmoothingFilter
```

6. Додайте наступні параметри в той же файл.

```
<param name="sample_duration" value="0.04" />
<rosparam command="load" file="$(find open_manipulator_moveit)/config/smoothing_filter_
params.yaml"/>
```

7. Вміст файлу запуску при вставці вмісту 5 і 6 показано в наслідувати приклад.

```
<launch>
 <!-- OMPL Plugin for MoveIt! -->
 <arg name="planning_plugin" value="ompl_interface/OMPLPlanner" />
```

```
<!-- The request adapters (plugins) used when planning with OMPL.
ORDER MATTERS -->
<arg name="planning_adapters" value="
        industrial_trajectory_filters/UniformSampleFilter
        industrial_trajectory_filters/AddSmoothingFilter

        default_planner_request_adapters/AddTimeParameterization
                default_planner_request_adapters/FixWorkspaceBounds
                default_planner_request_adapters/FixStartStateBounds
                default_planner_request_adapters/FixStartStateCollision

        default_planner_request_adapters/FixStartStatePathConstraints"
/>
<arg name="start_state_max_bounds_error" value="0.1" />
<param name="planning_plugin" value="$(arg planning_plugin)" />
<param name="request_adapters" value="$(arg planning_adapters)" />
<param name="start_state_max_bounds_error" value="$(arg
start_state_max_bounds_error)" />
<param name="sample_duration" value="0.04" />
<rosparam command="load" file="$(find
open_manipulator_moveit)/config/ompl_planning.yaml"/>
<rosparam command="load" file="$(find
open_manipulator_moveit)/config/smoothing_
```

```
filter_params.yaml"/>
</launch>
```

8. Як запустити

$ roslaunch open_manipulator_moveit_example demo.launch

Використовуйте наступну команду, щоб побачити, чим це відрізняється від попередньої демонстрації.

```
-----------
$ roslaunch open_manipulator_moveit open_manipulator_demo.launch
-----------
```

Раніше симулятор Gazebo був здатний тестувати поведінку робота в реальних умовах. У цьому розділі ми перевіримо рух робота, використовуючи комунікацію повідомлень між "move_group' і ланцюгом OpenManipulator в симуляторі Gazebo. Давайте спочатку запустимо Gazebo.

```
$ roslaunch open_manipulator_gazebo open_manipulator_gazebo.launch
```

У попередньому розділі ми спробували керувати роботом симулятора Gazebo за допомогою повідомлення зв'язки. Давайте подивимося на пакет open_manipulator_position_ctrl, який є вихідним кодом для цього.

```
$ roscd open_manipulator_position_ctrl/src
$ gedit position_controller.cpp
```

```
----------------------------------
open_manipulator_position_ctrl/src/position_controller.cpp
bool PositionController::initStatePublisher(bool using_gazebo)
{
 // ROS Publisher
 if (using_gazebo)
 {
 ROS_WARN("SET Gazebo Simulation Mode");
 for (std::map<std::string, uint8_t>::iterator state_iter = joint_id_.begin();
 state_iter != joint_id_.end(); state_iter++)
 {
 std::string joint_name = state_iter->first;
 gazebo_goal_joint_position_pub_[joint_id_[joint_name]-1] =
nh_.advertise<std_msgs::Float64>("/" + robot_name_ + "/" + joint_name
+ "_position/command", 10);
 }
 gazebo_gripper_position_pub_[LEFT_GRIP]                    =
nh_.advertise<std_msgs::Float64>("/" + robot_name_
+ "/grip_joint_position/command", 10);
 gazebo_gripper_position_pub_[RIGHT_GRIP]                   =
nh_.advertise<std_msgs::Float64>("/" +
robot_name_ + "/grip_joint_sub_position/command", 10);
 }
```

```
 else
 {
 goal_joint_position_pub_ =
nh_.advertise<sensor_msgs::JointState>("/robotis/
open_manipulator/goal_joint_states", 10);
 }
}
bool PositionController::initStateSubscriber(bool using_gazebo)
{
 // ROS Subscriber
 if (using_gazebo)
 {
 gazebo_present_joint_position_sub_ = nh_.subscribe("/" + robot_name_
+ "/joint_states", 10,
&PositionController::gazeboPresentJointPositionMsgCallback, this);
 }
 else
 {
 present_joint_position_sub_ =
nh_.subscribe("/robotis/open_manipulator/
present_joint_states", 10,
&PositionController::presentJointPositionMsgCallback, this);
 }
```

```
 move_group_feedback_sub_ = nh_.subscribe("/move_group/feedback", 10,
&PositionController::moveGroupActionFeedbackMsgCallback, this);
 display_planned_path_sub_ =
nh_.subscribe("/move_group/display_planned_path", 10,
&PositionController::displayPlannedPathMsgCallback, this);
 gripper_position_sub_ =
nh_.subscribe("/robotis/open_manipulator/gripper", 10,
&PositionController::gripperPositionMsgCallback, this);
}
-----------------------------------------------------------------------------------------------
```

Вузол position_controller пакета open_manipulator_position_ctrl отримує результат планування руху через повідомлення зв'язку з вузлом "move_group" і відповідає за створення кінцевого вхідного значення, яке може управляти маніпулятором за допомогою обчислення зворотної кінематики для проходження Згенерованого плану руху. У наведеному вище коді ми перевіряємо, чи слід використовувати Gazebo, і якщо маніпулятор управляється в середовищі Gazebo, ми реєструємо відповідний йому topicpublisher і підписуємося на стан кожного спільного значення поточного маніпулятора.

Вузол "position_controller" включений у файл " open_manipulator_demo.launch’. Введіть значення параметра для зв'язку з Gazebo після команди для виконання цієї демонстрації.

```
$ roslaunch open_manipulator_moveit open_manipulator_demo.launch use_gazebo:=true
```

Якщо ви виберете потрібну бібліотеку планування руху у вікні RViz, визначте координати кінцевої точки і натиснете кнопку Plan and Execute, як показано на рис. 232, то побачите, що маніпулятор в середовищі симулятора Gazebo і маніпулятор в вікні RViz, як показано на рис. 233, рухаються разом.

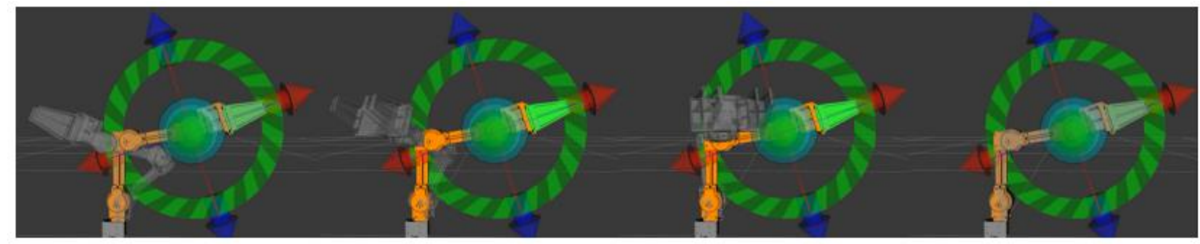


Рис. 233 Планування руху маніпулятора за допомогою демо-версії MoveIt! RViz

Рис. 234 Поведінка маніпулятора в середовищі Gazebo використовуючи MoveIt!

Якщо ви виберете потрібну бібліотеку планування руху у вікні RViz, визначте координати кінцевої точки і натиснете кнопку Plan and Execute, як показано на рис. 232, то побачите, що маніпулятор в

середовищі симулятора альтанки і маніпулятор в вікні RViz, як показано на рис. 232, рухаються разом. Вузол "position_controller" відповідає за управління лінійним захопленням з повідомленням зв'язку з "move_group".

```
-----------------------------------
open_manipulator_position_ctrl/src/position_controller.cpp
void PositionController::gripperPositionMsgCallback(const std_msgs::String::ConstPtr &msg)
{
 if (msg->data == "grip_on")
 {
 gripOn();
 }
 else if (msg->data == "grip_off")
 {
 gripOff();
 }
 else
 {
 ROS_ERROR("If you want to grip or release something, publish 'grip_on' or 'grip_off'");
 }
}
```

```
-----------------------------------------------------------------------------------------
---------------------------------
```

Щоб перемістити лінійне захоплення, просто опублікуйте тему наступним чином. Ви можете бачити, як рухається захоплення, як показано на рис. 235. Щоб діяти навпаки, використовуйте "grip_off' в команда.

```
$ rostopic pub /robotis/open_manipulator/gripper std_msgs/String "data:
'grip_on'"
once
```

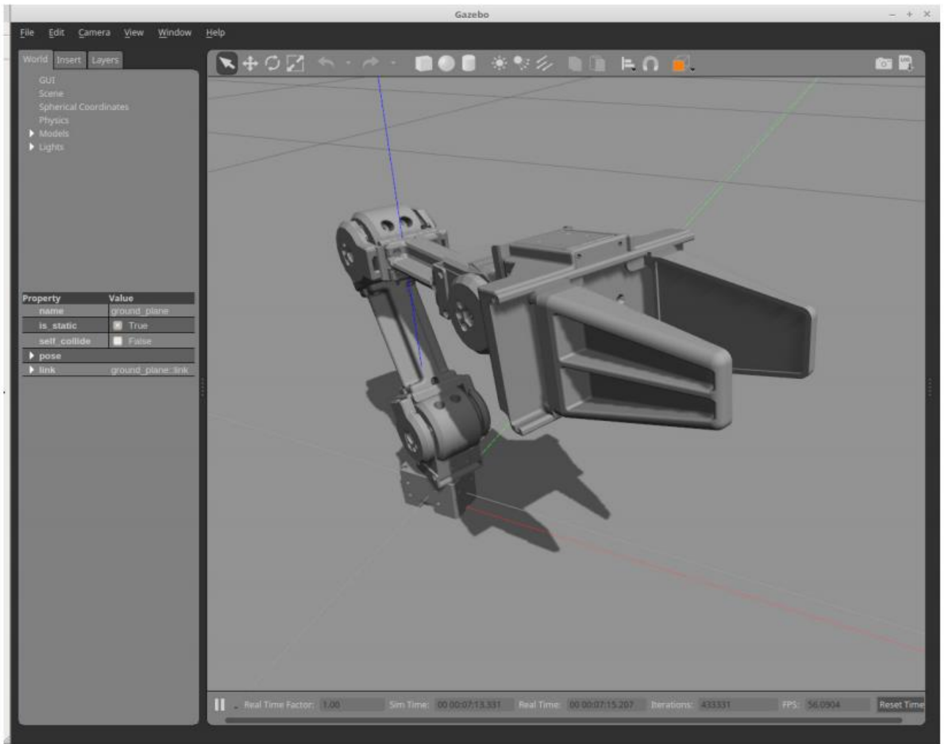


Рис. 235 Операція захоплення в середовищі Gazebo використовуючи MoveIt!

13.4. Застосування до фактичної платформи

До цих пір ми використовували Move It! щоб управляти маніпулятором на тренажері Gazebo. В цьому розділі давайте з'ясуємо, що нам потрібно для управління фактичним OpenManipulator Chain і управління ним.

OpenManipulator Chain складається в цілому з п'яти сумісних рам серії Dynamixel X actuator і деталей для 3D-друку. Користувачі можуть придбати диски Dynamixel X і сумісні рами, а також завантажити файли дизайну для 3D-друку з Onshape .

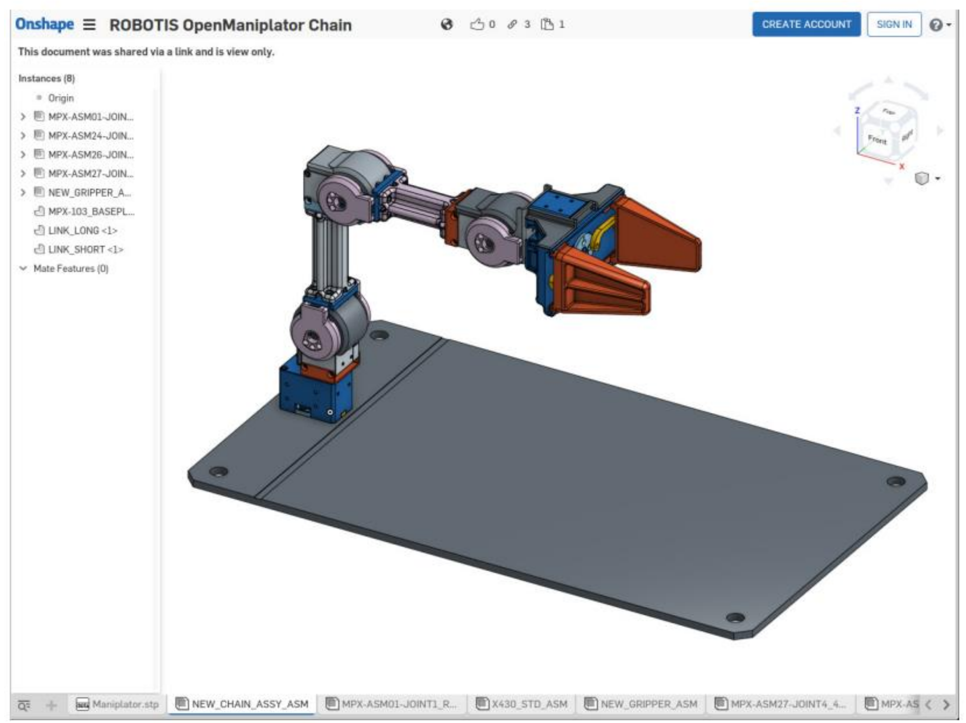


Рис. 236 Файл дизайну OpenManipulator Chain завантажений в Onshape

Для запуску Dynamixel на ROS вам знадобиться пакет dynamixel_sdk. ‘dynamixel_sdk " - це пакет ROS, включений в пакет

DYNAMIXEL SDK, що надається компанією ROBOTICS, який допомагає керувати Dynamixel простіше, надаючи функцію управління пакетним зв'язком.

Рис. 237 DYNAMIXEL SDK надається компанією ROBOTIS

Метапакет " dynamixel_workbench ", також наданий ROBOTIS, полегшує зміну параметрів керуючої таблиці Dynamixel за допомогою пакета "single_manager", а також забезпечує зручний графічний інтерфейс. Він також дає приклад того, як управляти Dynamixel в ROS за допомогою бібліотеки toolbox і пакета controller.

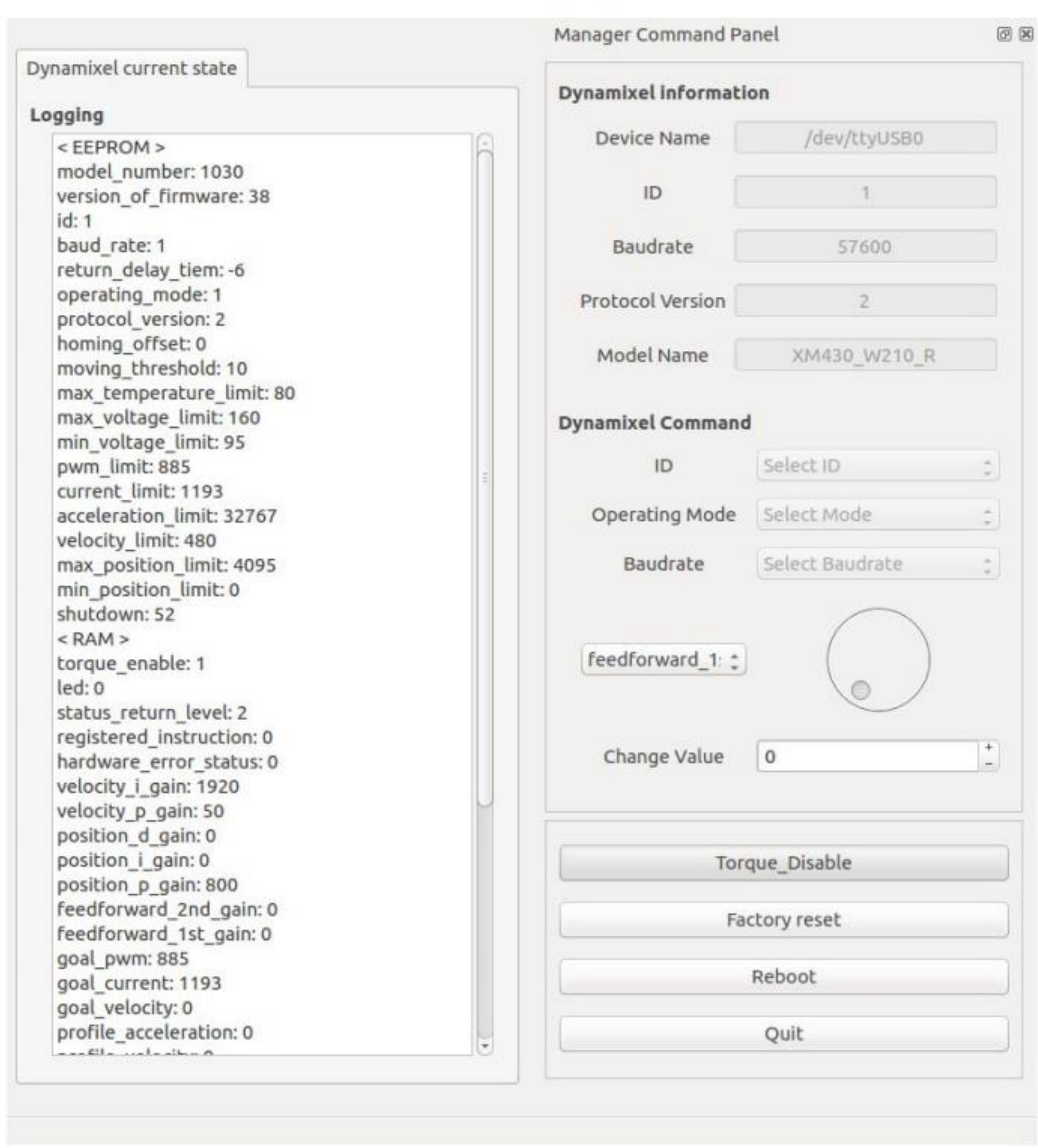


Рис. 238 DYNAMIXEL -робочий стіл, один з пакетів ROS, що надається компанією ROBOTIS

Встановіть всі види динаміків на 1 Мбіт / с (1 000 000 біт / с) і встановіть режим роботи в режим управління положення. Призначте ідентифікатори від 1 до 5 для кожного Dynamixel і почніть збірку,

звернувшись до OpenManipulator Wiki і опублікованої інформації про апаратне забезпечення (про форму). Після завершення збірки використовуйте U2 D2 для зв'язку з OpenManipulator Chain, Перетворіть метод підключення TTL або RS-485 в USB і підключіть його до головного комп'ютера. Задля живлення використовуйте SMPS з виходом 12 В, 5 А і подавайте живлення на Dynamixels за допомогою SMPS2DYNAMIXEL.

U2D2 and USB2DYNAMIXEL

U2D2 - це остання версія USB2DYNAMIXEL і має роз'єми, сумісні з серією Dynamixel X. Крім того, на відміну від існуючого USB2DYNAMIXEL, він підтримує роз'єм micro USB, і його розмір був значно зменшений. U2D2 підтримує RS-485, TTL і додатковий UART.

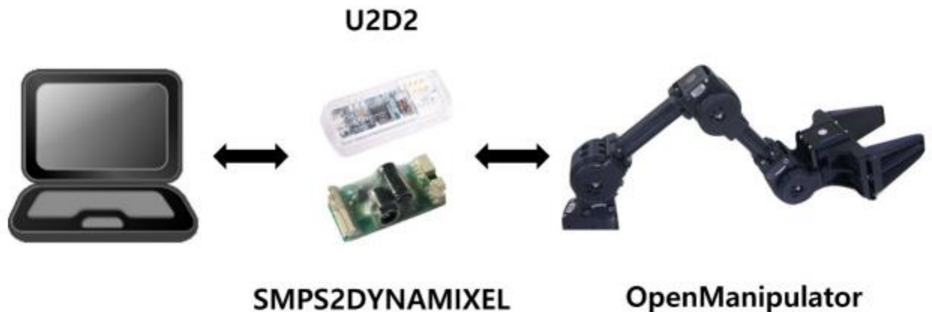


Рис.239 Налаштування та метод підключення для запуску OpenManipulator

Коли з'єднання буде завершено, введіть наступну команду у вікні терміналу.

Тут' chmod ' встановлює дозвіл на використання пристрою. Наступний приклад є прикладом того, коли U2D2 розпізнається як ttyUSB0.

$ sudo chmod a+rw /dev/ttyUSB0

$ roslaunch open_manipulator_dynamixel_ctrl dynamixel_controller.launch

Коли виконання завершено, до кожного Dynamixel додається крутний момент. Давайте перевіримо список тем.

```
--------------------------------------
$ rostopic list
/robotis/dynamixel/goal_states
/robotis/dynamixel/present_states
/rosout
/rosout_agg
--------------------------------------
```

Як уже згадувалося вище, поточне положення кожного динаміка може контролюватися через повідомлення теми зв'язку, і динамік може працювати при бажаному значенні кута.

Тепер давайте керувати фактичним маніпулятором через MOVEit!

$ roslaunch open_manipulator_moveit open_manipulator_demo.launch

У вікні *rviz* виберіть потрібну бібліотеку планування руху, введіть координати кінцевої точки і натисніть кнопку [Plan & Execute], щоб переконатися, що робот дійсно рухається.

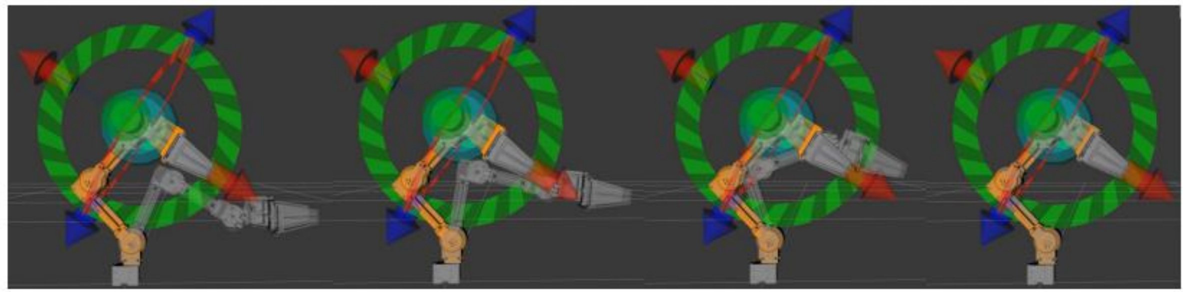


Рис.240 Планування руху ланцюга OpenManipulator за допомогою MoveIt!

Рис. 241 Робота ланцюга OpenManipulator

У наступному прикладі показано приклад використання єдиної позиції цільової пози шляхом установки елемента "position_only_ik" в значення "true". Подробиці див. у розділі [Посилання]. ([Посилання] 'Позначення цільової пози (положення + орієнтація) маніпулятора", яка була описана в розділі " 13.3.2 MoveIt! Setup Assistant') .

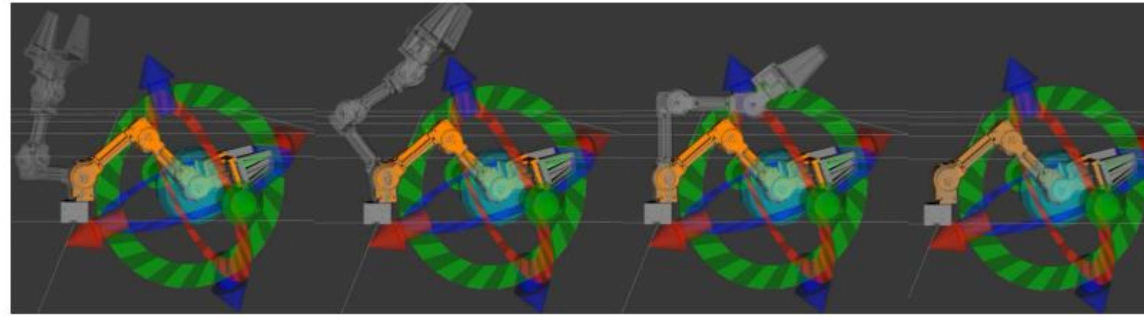


Рис. 242 Робота з плануванням руху ланцюга OpenManipulator за допомогою MoveIt! (Лише позиція IK)

Рис. 243 Робота ланцюга OpenManipulator

OpenManipulator Chain має ту перевагу, що вона сумісна з TurtleBot3 Waffle і Waffle Pi. Завдяки цій сумісності можна компенсувати відсутність свободи і мати велику повноту в якості сервісного робота з тими можливостями шолома і навігації, якими володіє TurtleBot3.

У цьому розділі ми розглянемо RViz, додавши TurtleBot3 Waffle URDF у файл ' open_manipulator_chain.xacro', створений вище.

Ви можете побачити папку URDF, в якій ви зберегли файл URDF, в папці turtlebot3_description. Коли я створював файл URDF OpenManipulator Chain, я згадав, що було б легше використовувати

його пізніше, якщо б він був збережений у форматі файлу xacro. Давайте розглянемо пакет *open_manipulator_with_tb3*.

```
$ roscd open_manipulator_with_tb3/urdf
$ gedit open_manipulator_chain_with_tb3.xacro



-----------------
open_manipulator_with_tb3/urdf/open_manipulator_chain_with_tb3.xa
cro
<!-- Include TurtleBot3 Waffle URDF -->
 <xacro:include                                            filename="$(find
turtlebot3_description)/urdf/turtlebot3_waffle_naked.urdf.xacro" />
 <!-- Base fixed joint -->
 <joint name="base_fixed" type="fixed">
 <origin xyz="-0.005 0.0 0.091" rpy="0 0 0"/>
 <parent link="base_link"/>
 <child link="link1"/>
 </joint>
-----------------------------------------------------------------------------------
----------------
```

Якщо ви відкриєте файл URDF пакета open_manipulator_with_tb3, то побачите код, щоб увімкнути файл turtlebot3_waffle_naked.urdf.xacro у верхній частині. За допомогою

цього коду TurtleBot 3 Waffle може бути завантажена, і маніпулятор створює нерухоме з'єднання на ланці, де вона повинна бути розташовувати.

TurtleBot3 Waffle і Waffle Pi

TurtleBot 3 Waffle і Waffle Pi в основному оснащені датчиком LIDAR, але для зручності користувача ROBOTIS також пропонує URDF без датчика LIDAR.

Тепер давайте запустимо RViz з наступною командою.

```
$ roslaunch open_manipulator_with_tb3 open_manipulator_chain_with_tb3_rviz.launch
```

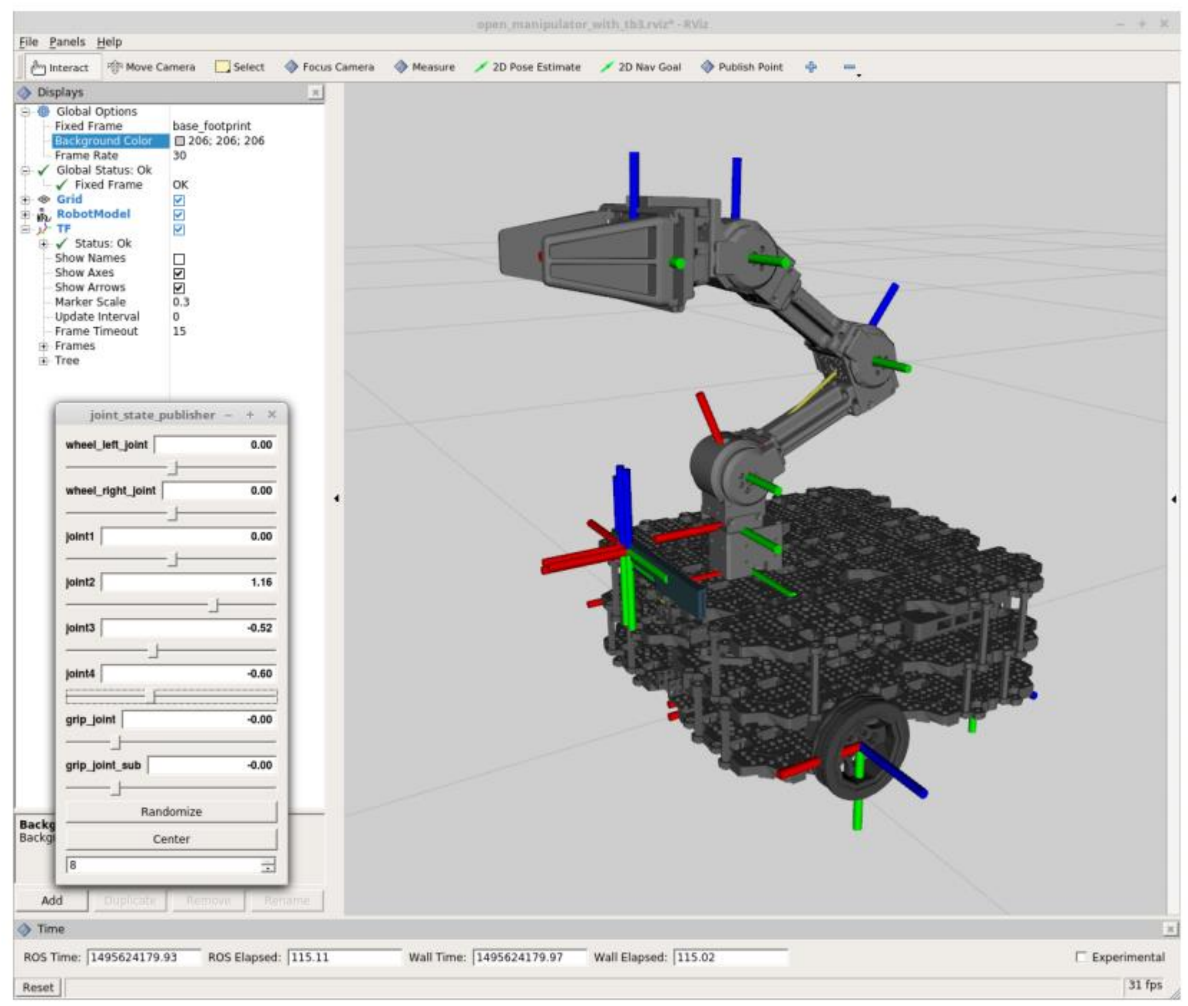


Рис. 244 Ланцюг OpenManipulator, встановлений на urtleBot3 Waffle

OpenManipulator Chain використовує модульний привід під назвою Dynamixel, який може бути встановлений на різних роботах. Рекомендується використовувати можливість повторного використання xacro на замовному роботі.