КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
ВЫСШАЯ ШКОЛА ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И
ИНТЕЛЛЕКТУАЛЬНЫХ СИСТЕМ
Кафедра интеллектуальной робототехники

Р.О. ЛАВРЕНОВ, Е.А. МАГИД

ПРОГРАММИРОВАНИЕ В РОБОТОТЕХНИЧЕСКОЙ ОПЕРАЦИОННОЙ СИСТЕМЕ (ROS)

Учебно-методическое пособие

КАЗАНЬ
2019

УДК 621.865.8(075.8)
ББК 32.816я7
Л13

Рецензент
кандидат технических наук, заведующий лабораторией
нейроморфных вычислений и нейросимуляции
Высшей школы ИТИС КФУ **М.О. Таланов**

Лавренов Р.О.

Л13 **Программирование в Робототехнической операционной системе (ROS):** учебно-методическое пособие / Р.О. Лавренов, Е.А. Магид. – Казань: Изд-во Казан. ун-та, 2019. – 40 с.

Учебно-методические пособие предназначено для студентов, обучающихся по направлению «Программная инженерия» и профилю подготовки «Интеллектуальная робототехника». Материалы, представленные в пособии, также могут быть полезны преподавателям робототехники. С помощью пособия читатель сможет самостоятельно освоить основы программирования роботов с Робототехнической операционной системой на языках программирования C++ и Python.

Пособие будет полезно для начинающих изучение Робототехнической операционной системы и может служить шпаргалкой для тех, кто уже знаком с ROS.

УДК 621.865.8(075.8)
ББК 32.816я7

Содержание

Введение

Начинающие изучать программирование с использованием ROS, должны обладать базовыми знаниями основных понятий ROS. То есть должен знать, что такое в ROS ноды (англ. node), топики (англ. topic), сообщения (англ. messages) и сервисы (англ. services). Читатель должен знать как запускать ноды и что такое .launch файлы. Кроме того, при программировании роботов будет необходимо использовать симуляторы. такие как RViz и Gazebo.

Программирование с ROS осуществляется в linux-системах (Ubuntu или Debian), поэтому читателю пригодятся умение работать с командной строкой (терминалом). Важное требование для работы с роботами — это знание нескольких языков программирования. Наиболее часто используемые языки программирования для создания роботизированных приложений, — это C++ и Python. В данном пособии будут приводиться примеры программирования для ROS на языке C++. поэтому читатели должны иметь базовые знания этого языка, и основные концепции объектно-ориентированного программирования (ООП).

Что означает программирование в ROS? Это означает, что ROS предоставляет некоторые встроенные функции для программирования робототехнических приложений. Например, если мы хотим реализовать новое сообщение ROS или сервис ROS, мы можем просто вызвать готовые функции для их создания. Нам не нужно реализовывать функции ROS с нуля. Программы, использующие ROS, называются нодами и используют API ROS. В этом пособии мы разработаем собственные ROS ноды на языках программирования C++ и Python.

Первый шаг в программировании — создать рабочее пространство ROS (англ. workspace).

1. Рабочее пространство ROS

1.1 Создание и описание рабочего пространства ROS

Первым шагом в разработке ROS является создание рабочего пространства, в котором хранятся пакеты ROS. В рабочем пространстве хранятся создаваемые пакеты, там же стоит располагать скаченные с различных репозиториев пакеты. В рабочем пространстве хранятся исходные файлы, промежуточные файлы сборки и финальные исполняемые файлы: ноды и .launch файлы.

Сначала вы должны создать папку рабочего пространства ROS. Обычно оно находится в домашней папке Ubuntu, в папке catkin_ws. Настоятельно рекомендуется сделать рабочее пространство там.

В новом окне терминала введите следующую команду. Она создаст папку с именем catkin_ws, внутри которой находится еще одна папка с именем src. Рабочее пространство ROS также называется рабочим пространством catkin (*catkin workspace*). Далее, будет подробно рассказано что это.

```
mkdir -p  /catkin_ws/src
```

После ввода команды перейдите в папку src с помощью linux-команды "cd":

```
cd catkin_ws/src
```

```
robot@robot-pc:~/catkin_ws/src$ catkin_init_workspace
Creating symlink "/home/robot/catkin_ws/src/CMakeLists.txt"
robot@robot-pc:~/catkin_ws/src$
robot@robot-pc:~/catkin_ws/src$
robot@robot-pc:~/catkin_ws/src$ ls
CMakeLists.txt
robot@robot-pc:~/catkin_ws/src$
```

Рис. 1.1: Инициализация рабочего пространства

Следующая команда инициализирует новое рабочее пространство ROS. Если вы не инициализируете рабочее пространство, вы не можете создавать и собирать исполняемые файлы из файлов с исходным кодом.

```
catkin_init_workspace
```

После этой команды вы должны увидеть сообщение на рис. 1.1 в вашем терминале.

Внутри папки src находится файл CMakeLists.txt. После инициализации рабочего пространства catkin вы можете создать рабочее пространство. Чтобы создать рабочее пространство, переключитесь из папки catkin_ws/src в папку catkin_ws. После этого создайте рабочее пространство командой

```
catkin_make
```

Данная команда создаст в папке catkin_ws директории build и devel. Далее, мы подробней разберем, что в них хранится. Но для начала требуется прописать наше рабочее пространство в пути поиска файлов для терминала. Чтобы система находила пути до файлов, которые мы будем создавать или скачивать с репозиториев. Для этого вам потребуется выполнить следующие шаги. Откройте файл .bashrc в директории home (для этого потребуются права суперпользователя — sudo):

```
sudo gedit .bashrc
```

При каждом запуске терминала выполняется скрипт .bashrc. Добавьте следующую строку в конец файла .bashrc (см рисунок):

```
source  /catkin_ws/devel/setup.bash
```

```
if ! shopt -oq posix; then
  if [ -f /usr/share/bash-completion/bash
    . /usr/share/bash-completion/bash_com
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi
source /opt/ros/kinetic/setup.bash

source ~/catkin_ws/devel/setup.bash
```

Рис. 1.2: Добавление рабочего пространства в .bashrc файл.

В файле setup.bash как раз и содержатся пути до файлов, которые потребуются при сборке и запуске ваших пакетов из терминала. Теперь, когда при использовании окон терминала, мы будем иметь доступ к пакетам внутри рабочего пространства.

Прежде чем обсуждать создание пакетов, нужно рассказать о системе сборки catkin в ROS.

Файлы с исходным кодом бесполезны, если из них нельзя собрать исполняемые файлы. Это могут быть запускаемые исполняемые файлы или файлы библиотек, статических или динамически подключаемых. Система сборки пакетов в ROS называется catkin (http://wiki.ros.org/catkin). Catkin – это комбинированная система сборки, объединяющая в себе систему сборки CMake и написанные на Python скрипты. Одной только системы CMake недостаточно, потому что сборка ROS-пакетов — сложный процесс. Сложность возрастает с увеличением количества пакетов и усложнения зависимостей пакетов. Система сборки catkin заботится обо всех этих вещах.

Вернемся к рабочему пространству ROS и объясним его структуру, зная теперь о системе сборке пакетов. Разберем по порядку предназначение директорий рабочего пространства:

1. Директория ***src***. Папка src внутри папки рабочей области catkin (папка catkin_ws) — это место, где вы можете создавать собственные пакеты и куда следует копировать пакеты из репозиториев. Сборка пакетов ROS и генерация исполняемых файлов происходит только тогда, когда они находятся в папке src. При выполнении команды catkin_make из папки catkin_ws, система сборки проверяет все папки внутри папки src и собирает каждый пакет.

2. Директория ***build***. Когда запускаем команду catkin_make из рабочего пространства ROS (catkin_ws), система сборки catkin создает различные промежуточные файлы сборки и промежуточные кэш-файлы CMake внутри папки build. Эти файлы кэша помогают предотвратить повторную сборку всех пакетов при запуске команды catkin_make. Например, если вы соберете пять пакетов, а затем добавите новый пакет в папку src, во время следующего вызова команды catkin_make будет собран только новый пакет. Это происходит из-за кэш-файлов внутри папки build. Если вы удалите папку build, все пакеты будут пересобраны.

3. Директория ***devel***. Когда мы запускаем команду catkin_make, происходит сборка каждого пакета, и если сборка проходит успешно, создается целевой исполняемый файл. Исполняемые файлы хранятся в папке devel. в этой же папке, в упомянутом выше файле setup.bash индексируются пути до появившихся исполняемых файлов. Поэтому данный файл и нужно добавить в .bashrc.

4. Директория ***install***. После сборки исполняемых файлов в директории devel, некоторые из файлов могут не запускаться, потому что им будет необходимо наличие дополнительных динамически подключаемых библиотек, расположенных в одной папке с ними. Часто это происходит когда несколько пакетов находятся в сложной взаимозависимости друг от друга. Для того чтобы провести

окончательную сборку всех необходимых файлов вместе может потребоваться выполнить команду:

catkin_make install

Данная команда создаст папку install и соберет в нем исполняемые файлы и библиотеки в необходимых для запуска комбинациях.

1.2 Создание пакета ROS

Разберем процесс создания собственного пакета ROS. Пакет ROS — это директория, где находится исходный код, скрипты, необходимые библиотеки для сборки одной или нескольких нод ROS. Можно создать пакет ROS с помощью следующей команды из системы сборки catkin:

catkin_create_pkg имя_ros_пакета зависимости_пакета

где, catkin_create_pkg — команда создания пакета; первый аргумент — имя создаваемого пакета; а далее через пробел указаны пакеты, от которых зависит создаваемый пакет. Разберем какие могут быть зависимости в следующей главе. А сейчас создадим пакет hello_world со всеми необходимыми зависимостями. Для этого в окне терминала нужно перейти в папку catkin_ws/src и выполнить следующую команду:

catkin_create_pkg hello_world roscpp rospy std_msgs

Рис. 1.3: Создание нового пакета ROS.

В результате вы создадите папку "hello_world"следующим содержимым (рис. 1.3):

1. Директория ***src***. В данной папке, как было сказано выше, хранятся файлы исходного кода. Если вам захочется сделать отдельную директорию под сприпты Python, то рядом с директорией src можно создать папку ***scripts***.

2. Директория ***include***. Данная папка создается автоматически и служит для хранения заголовочных файлов C++.

3. Файл ***package.xml***. Данный файл в формате XML должен быть включен в корневую папку любого совместимого с catkin пакета. Этот файл определяет свойства ROS-пакета, такие как имя пакета, номер версии, авторов пакета и зависимости от других пакетов. Если зависимости указаны некорректно, то пакет даже может собираться на вашем компьютере, но не будет гарантировано собираться и запускаться на других устройствах, где ваш пакет могут использовать.

4. Файл ***CMakeLists.txt*** является основным файлом настроек сборки пакетов программного обеспечения. В нем описано, как создавать код и куда его устанавливать. В данном файле перечислены основные настройки сборки, которые при необходимости следует раскомментировать и инициализировать вручную. Файл CMakeLists.txt ДОЛЖЕН соответствовать этому формату, иначе ваши пакеты не будут собраны правильно:

   (a) Требуемая версия CMake (cmake_minimum_required)

   (b) Название пакета (project())

   (c) Перечисление других CMake/Catkin пакетов, необходимых для сборки (find_package())

   (d) Поддержка скриптов Python (catkin_python_setup())

   (e) Генерация самописных сообщений/сервисов/действий (add_message_files(), add_service_files(), add_action_files())

   (f) Экспорт данных из пакета (catkin_package())

   (g) Собираемые библиотеки/исполняемые файлы (add_library()/add_executable()/target_link_libraries()

   (h) Собираемые тесты (catkin_add_gtest())

   (i) Правила установки (install())

2. Реализация функционала ROS в коде

2.1 Клиентские библиотеки ROS

Мы рассмотрели различные концепции ROS, такие как топики, сервисы, сообщения и так далее. Но как реализовать все это в коде? С помощью клиентских библиотек ROS. Клиентские библиотеки ROS представляют собой код с необходимыми классами и функциями для реализации концепций ROS. Мы можем просто включить эти библиотеки в нашу программу, чтобы сделать её нодой ROS. Клиентская библиотека экономит время разработки, поскольку она предоставляет готовые функции для приложений ROS. Мы можем написать ROS-ноды на любом языке программирования, для которого существуют клиентские библиотеки ROS. В противном случае нам может понадобиться реализовывать основные концепции ROS самостоятельно.

Ниже приведены основные клиентские библиотеки ROS:

- ***roscpp***: это клиентская библиотека ROS для C++. Широко используется для разработки приложений ROS из-за высокой производительности.
- ***rospy***: это клиентская библиотека ROS для Python. Преимущество — экономия времени на разработку. Мы можем создать узел ROS за меньшее время, чем с помощью roscpp. Идеально подходит для быстрого прототипирования приложений.
- ***roslisp***: это клиентская библиотека ROS для языка Lisp. В основном используется в библиотеках планирования движения на ROS, но не так популярна, как roscpp и rospy.

Существуют также экспериментальные клиентские библиотеки, включая ***rosjava***, ***rosnodejs*** и ***roslua***.

2.2 Подключение клиентских библиотек roscpp и rospy

Когда вы пишете код на C++, в первую очередь вы подключаете необходимые заголовочные файлы. Точно так же, когда вы пишете код

Python, в первую очередь вы импортируете модули Python. Чтобы создать ноду ROS в C++, мы должны подключить следующий заголовочный файл.

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

В файле ros.h есть подключения всех заголовочных файлов, необходимых для реализаций функций ROS. Мы не сможем создать в C++ ноду ROS без подключения этого заголовочного файла. Следующим типом заголовочных файлов, используемых в нодах ROS, является заголовочный файл сообщений ROS. В ROS имеется множество встроенных типов сообщений, однако, можно создать и свои собственные. В ROS есть встроенный пакет сообщений, называемый std_msgs, в котором есть определения стандартных типов данных, таких как int, float, string и т.д. Например, если мы хотим использовать в коде строковые сообщения, мы можем подключить готовую реализацию с помощью включения в код следующего заголовочного файла.

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

Здесь первая часть — это имя пакета, а следующая — имя типа сообщения. Если создан пользовательский тип сообщения, мы можем использовать его в коде с помощью следующего синтаксиса.

```
#include "msg_pkg_name/message_name.h"
```

Для создания ноды ROS в языке Python нам нужно импортировать модуль:

```
import rospy
```

В rospy имеются все важные функции ROS. Однако, чтобы импортировать типы сообщений, мы должны импортировать определенные модули, по аналогии с C++. Ниже приведен пример импорта строкового типа в Python:

```
from std_msgs.msg import String
```

То есть мы должны указать из какого пакета и какой тип мы импортируем.

2.3 Инициализация ROS ноды

При запуске любой ноды ROS первая вызванная функция должна инициализировать ноду. Это обязательный шаг в любой ноде ROS. В C++ нода инициализируется, при использовании следующей строчки кода.

```
int main(int argc, char **argv) {
ros::init(argc, argv, "name_of_node");
.....................
}
```

Внутри функции int main() мы должны вызвать функцию ros::init(), которая инициализирует ноду ROS. В функцию init () мы можем передать аргументы командной строки argc, argv и, собственно, имя создаваемой ноды.

После инициализации ноды мы должны создать экземпляр класса NodeHandle (дескриптор ноды), который запускает ноду ROS и другие функции, например, такие как публикация/прослушивание топиков. Для создания дескриптора в C++ используется класс ros::NodeHandle:

```
ros::NodeHandle nh;
```

Остальные функции в ноде будут использовать созданный экземпляр nh.

В Python мы используем следующую строку кода.

```
rospy.init_node('name_of_node', anonymous=True);
```

Первым аргументом является имя создаваемой ноды, а вторым аргументом является anonymous=True, что означает, что нода может работать в нескольких экземплярах.

В Python не нужно создавать дескриптор; модуль rospy обрабатывает функции ноды внутри себя.

2.4 Публикация и получение сообщений из топиков

Перед публикацией сообщений в топик мы должны создать эти сообщения. Например создадим строковое сообщение, воспользовавшись пакетом std_msgs. А после создания экземпляра сообщения ROS следует задать его содержимое.

```
std_msgs::String msg;
msg.data = "String data";
```

В языке Python делается аналогично, только без объявления типа переменной:

```
msg = String()
msg.data = "string data"
```

После формирования сообщения для отправки в топик, нужно создать экземпляр класса Publisher. Для этого в C++ используется следующий синтаксис:

```
ros::Publisher publisher_name =
            nh.advertise<ROS_message_type>("topic_name",1000);
```

где publisher_name — название переменной-экземпляра класса Publisher; topic_name — название топика, в который будут отправляться сообщения, а 1000 — размер очереди, которая будет промежуточным буфером при отправке сообщений. Функция advertise() вызывается из созданного ранее дескриптора ноды и нужна для инициализации экземпляра класса.

После создания и настройки класса-отправителя, можно уже воспользоваться его функцией publish(), чтобы отправить сообщение в топик:


```
publisher_name.publish(msg)
```

В языке Python систаксис будет аналогичным:

```
publisher_name =
        rospy.Publisher('topic_name', message_type, queue_size)
publisher_name.publish(msg)
```

Теперь разберем как подписываться на топик и прослушивать сообщения в нем. В языка C++ для прослушивания данных топика необходимо сначала создать экземпляр класса Subscriber, вызвав у дескриптора ноды функцию subscribe(). В качестве аргументов у этой функции будут: название топика (topic_name); размер очереди, которая будет промежуточным буфером при приемке сообщений; callback-функция обработки принятого сообщения:

```
ros::Subscriber subscriber_name =
            nh.subscribe("topic_name", 1000, callback_function);
```

Callback-функция — это пользовательская функция, которая выполняется после получения сообщения из топика. Внутри этой функции мы можем производить манипуляции с полученным сообщением или анализировать его и, в зависимости от содержимого совершать какие-либо действия.

В языке программирования Python подписаться на топик можно схожим образом, с тем исключением, что в Python вторым аргументом функции требуется указать тип получаемого сообщения:

```
rospy.Subscriber("topic_name", message_type, callback_funtion)
```

Когда мы подписываемся на топик ROS и в этот топик приходит сообщение, происходит вызов callback-функции, указанной при создании экземпляра класса Subscriber. Ниже приведен синтаксис и пример callback-функции в C++. Сообщение в функцию приходит в виде константного указателя.

```
void callback_name (const ros_message_const_pointer & pointer) {
pointer->data    // Access data
}
```

Ниже представлен пример вывода получаемых в callback-функции данных.

```
void chatterCallback(const std_msgs::String::ConstPtr& msg) {
ROS_INFO("I heard: [%s]", msg->data.c_str());
}
```

Callback-функция в Python выглядит как обычная функция. В примере ниже тоже происходит простой вывод полученного сообщения на экран:

```
def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.
        data)
```

2.5 Функции Spin, SpinOnce и Sleep

После начала публикации сообщений в топик программе может потребоваться время для обработки запроса на отправку. Для этого после отправки сообщения в языке C++ должна вызываться функция **spinOnce()**.

Если вы в программе C++ подписываетесь на прослушивание сообщений с топиков, то перед началом получения сообщений должны вызвать функцию **spin()**. Данная функция останавливает работу основного кода и передает управление callback-функциям.

Если в одной программе C++ вы и подписываетесь на топики и прослушиваете сообщения из какого-либо топика, то корректной обработки получаемой и отправляемой информации используйте функцию **spinOnce()**.

В Python нет функции **spinOnce()**, но при публикации сообщений вы можете использовать функцию **rospy.sleep()**. Если же вы только прослушиваете сообщения из топиков то пользуйтесь функцией **rospy.spin()**.

Если вы хотите чтобы какое-либо событие происходило с определенной периодичностью, например, отправка сообщений в топик, тогда нужно использовать функцию **Rate**, находящуюся внутри цикла.

Для этого сначала создается экземпляр класса **Rate** с указанием частоты в Гц. После этого, в цикле нужно вызывать функцию **sleep**, которая будет останавливать выполнение кода на заданное время.

Ниже приведен пример задания 10Гц в C++:

```
ros::Rate r(10); // 10hz
r.sleep();
```

И аналогичный пример в Python.

```
rate = rospy.Rate(10) # 10hz
rate.sleep()
```

Для того чтобы зациклить какое-либо действие, которое должно прекращаться по закрытию ноды, в C++ используется проверка значения **ros::ok()**. Если это значение станет ложным — значит произошло закрытие ноды. В языке Python для этих же целей проверяется значении функции **rospy.is_shutdown()**. Если оно истинно, то произошло закрытие ноды.

Ноду можно закрыть либо сочетанием клавиш **Ctrl+C** в окне, где она запущена, либо вызовом функции **Shutsown()** из кода.

2.6 Вывод на экран и обработка параметров

ROS предоставляет API для вывода сообщений. Это может быть полезным, когда в ходе работы ноды, в окне терминала необходимо получать какую либо справочную или служебную информацию. Функции

вывода получают шаблон строки в качестве первого аргумента, а в качестве последующих — вставляемые в строку значения. Виды функций вывода в C++:

- ROS_INFO(string_msg, args): Информационное сообщение
- ROS_WARN(string_msg, args): Предупреждение
- ROS_DEBUG(string_msg, args): Отладочная информация
- ROS_ERROR(string_msg, args): Сообщение об ошибке
- ROS_FATAL(string_msg, args): Критическая ошибка

Виды функций вывода в Python:

- rospy.loginfo(msg, *args): Информационное сообщение
- rospy.logwarn(msg, *args): Предупреждение
- rospy.logdebug(msg, *args): Отладочная информация
- rospy.logerr(msg, *args): Сообщение об ошибке
- rospy.logfatal(msg, *args): Критическая ошибка

Примеры использования вывода был упомянуты выше в демонстрации примеров использования callback-функции.

Для работы с различными параметрами нод в ROS задействован специальный сервер параметров. Из кода C++ мы может получать значения параметров, использовав функцию **getParam()** у дескриптора ноды. Первым аргументов требуется передать название параметра, а вторым — куда сохранять значение параметра:

```
std::string global_name;
if (nh.getParam("/global_name", global_name)) {
...
}
```

Задать значение параметра в C++ можно функцией **setParam()** дескриптора ноды, передав в неё название параметра и задаваемое ему значение:

```
nh.setParam("/global_param", 5);
```

В языке Python манипуляции с параметрами происходят с помощью функций **get_param**, **set_param** из клиентской библиотеки:

```
global_name = rospy.get_param("/global_name")
rospy.set_param('private_int', '2')
```

3. Программа «Hello world» в ROS

3.1 Создание пакета «Hello world»

Программы ROS организованы в виде пакетов. Поэтому мы должны создать пакет ROS перед написанием любой программы. Чтобы создать пакет ROS, мы должны задать имя пакета и зависимости создаваемого пакета. Например, если мы планируем писать программу на C++, то в зависимости следует добавить клиентскую библиотеку **roscpp**, если на Python, то следует добавить **rospy**. Перед созданием пакета сначала переключитесь в папку src. Создать пакет можно с использованием команды **catkin_create_pkg**. Выше когда мы изучали содержимое пакетов, мы уже создали пакет hello_world со всеми необходимыми зависимостями. Для этого мы воспользовались командой:

catkin_create_pkg hello_world roscpp rospy std_msgs

Проверьте, что в папке **catkin_ws/src** создана директория hello_world и в ней есть файлы package.xml и CMakeLists.txt.

3.2 Создание ROS нод в C++

После создания пакета следующим шагом является создание нод. Исходный код ноды, написанный на языке C++ хранится в папке src. Ниже приведен пример ноды. Это C++ нода для публикации строкового сообщения «Hello World» в топик **chatter**. Сохраните данный код как файл talker.cpp в папке src.

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
int main(int argc, char **argv) {
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("
      chatter", 1000);
  ros::Rate loop_rate(10);
  int count = 0;
  while (ros::ok()) {
```



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

Данный код создает строковые сообщения и отправляет их в топик **/chatter**. В данном коде вы можете увидеть все разобранные выше функции. В начале происходит инициализация ноды и создание дескриптора ноды, после чего создаем экземпляр класса Publisher. Далее, создаем цикл в котором с частотой 10 раз в секунду происходит отправка сообщений. Код выполняется до тех пор, пока вы не нажмете Ctrl+C.

Напишем код файла listener.cpp, в котором подпишемся на этот топик **/chatter**. При получении сообщений, в callback-функции выведем их на экран с помощью функции ROS_INFO().

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

В listener.cpp мы подписываемся на топик **/chatter** и, при получении сообщений будет вызываться callback-функция chatterCallback. Данная функция будет вызываться каждый раз при получении сообщения, внутри этой функции сообщения выводятся на экран с помощью функции ROS_INFO().

Заметьте, что в программе вызывается функция ros::spin(), которая заставляет ноду находится в режиме ожидания, при котором происходит выполнение только callback-функций. Нода не завершит работу, пока вы не нажмете Ctrl+C.

3.3 Сборка в C++ и запуск ROS нод

После создания двух файлов в папке hello_world/src ноды необходимо скомпилировать для создания исполняемого файла. Для этого нам нужно отредактировать файл CMakeLists.txt. Нужно добавить четыре стро-

ки кода в CMakeLists.txt. На рисунке 3.1 продемонстрированы те строки, которые нужно добавить в файл самостоятельно. Нужно добавить команды CMake **add_executable(...)** и **target_link_libraries(...)**.

Первый аргумент команды **add_executable()** — имя создаваемой ноды; второй и последующие аргументы — файлы исходного кода (только .cpp), которые необходимы для компиляции.

Первый аргумент команды **target_link_libraries()** — имя создаваемой ноды, второй и последующие аргументы — перечисление библиотек используемых при компоновке (линковке).

```
include_directories(
  ${catkin_INCLUDE_DIRS}
)

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

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

Рис. 3.1: Файл CMakeLists.txt

После сохранения CMakeLists.txt соберем наш пакет. Для этого перейдите в папку **catkin_ws/src** и выполните команду **catkin_make**. Если все выполнено верно, то вы увидите что сбока прошла успешно (см Рисунок 3.2).

```
Scanning dependencies of target listener
Scanning dependencies of target talker
[ 25%] Building CXX object hello_world/CMakeFiles/listener.dir/src/listener.cpp.o
[ 50%] Building CXX object hello_world/CMakeFiles/talker.dir/src/talker.cpp.o
[ 75%] Linking CXX executable /home/robot/catkin_ws/devel/lib/hello_world/talker
[100%] Linking CXX executable /home/robot/catkin_ws/devel/lib/hello_world/listener
[100%] Built target listener
[100%] Built target talker
robot@robot-pc:~/catkin_ws$
```

Рис. 3.2: Сборка написанных нод.

После того, как ноды собраны, можно проверить что в папке **catkin_ws/devel/lib** действительно находятся два исполняемых файла.

После создания исполняемых файлов мы можем запустить их в терминале Linux. Откройте три окна терминала и выполните каждую команду в отдельной вкладке. Сначала запустим ROS-Мастер:

roscore

Потом ноду **talker**:

rosrun hello_world talker

Затем выведем список активных топиков:

rostopic list

Вы увидите топики /chatter, /rosout, /rosout_agg, ... /chatter — это топик, в который публикует нода **talker**. Топики /rosout предназначены для хранения логов, они запускаются при команде **roscore**. Запустите ноду **listener** в новой вкладке терминала:

rosrun hello_world listener

Нода **listener** будет выводить на экран сообщения, которые получает из топика /chatter (см. рисунок 3.3.

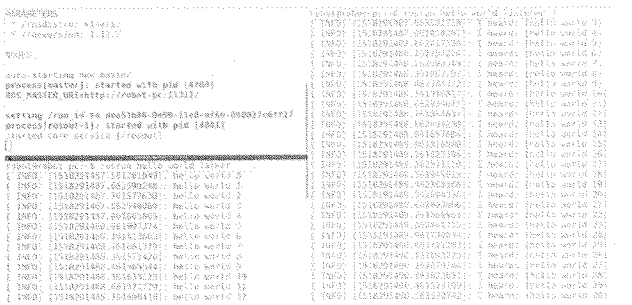

Рис. 3.3: Вывод C++ нод **talker** и **listener**.

3.4 Создание и запуск ROS нод в Python

В папке вашего пакета создайте директорию с названием script внутри пакета. Внутри этой папки создадим две программы. Первая из них talker.py. Вот её исходный код:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
def talker():
  rospy.init_node('talker', anonymous=True)
  pub = rospy.Publisher('chatter', String, queue_size=10)
  rate = rospy.Rate(10) # 10hz
  while not rospy.is_shutdown():
    hello_str = "hello world %s" % rospy.get_time()
    rospy.loginfo(hello_str)
    pub.publish(hello_str)
    rate.sleep()
if __name__ == "__main__":
  try:
```

```
    talker()
  except rospy.ROSInterruptException:
    pass
```

В коде talker.py в начале мы импортируем модуль rospy и модули сообщений ros. В функции talker() мы видим инициализацию ноды ROS, создание экземпляра класса-отправителя. После инициализации ноды мы используем цикл while для публикации строкового сообщения «Hello World» в топик /chatter.

Далее создадим ноду, прослушивающую топик /chatter. Создадим файл listener.py так же в папке script. Код этого Python файла:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
def listener():
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("chatter", String, callback)
    rospy.spin()
if __name__ == "__main__":
    listener()
```

У каждой ноды в ROS должно быть уникальное имя. Если запускается новая нода с именем уже запущенной ноды, то более старая нода закрывается. При инициализации ноды мы задали anonymous=True, значит ROS-мастер сам сгенерирует уникальное имя для запускаемой ноды и возможно будет запускать больше одного экземпляра этой ноды.

Запущенная нода также как и в C++ создает Subscriber и с помощью функции spin() уходит в режим ожидания, выполняя callback-функцию при получении сообщений.

Написанный на Python код не надо компилировать и проводить компоновку. Поэтому, написанные файлы можно сразу запускать в консоли как исполняемые. Не забудьте указать в настройках этих файлов, что их нужно запускать как программы. Запустите в разных вкладках терминала ROS-мастер и написанные ноды:

roscore
rosrun hello_world talker.py
rosrun hello_world listener.py

Вывод будет аналогичным тому, что мы увидели ранее (см рисунок 3.4).

3.5 Создание .launch файлов

Рассмотрим создание файлов запуска (.launch файлов). Как озвучивалось ранее, преимущество таких файлов в том, что появляется возможность запуска множества нод в одной вкладке терминала и не нужно отдельно запускать ROS-Мастер.

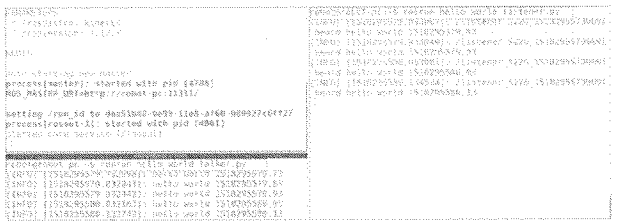

Рис. 3.4: Вывод Python нод **talker** и **listener**.

Создайте в папке вашего пакета директорию launch. В данной директории создайте файл запуска talker_listener.launch со следующим содержимым:

```
<launch>
<node name="listener1" pkg="hello_world" type="listener" output=
    "screen"/>
<node name="talker1" pkg="hello_world" type="talker" output="
    screen"/>
</launch>
```

Этот .launch файл запускает ноды talker и listener. Имя пакета задается в поле pkg, а имя исполняемого файла — в поле type. Ноде можно назначить любое имя (поле name), но, естественно, лучше чтобы оно было похоже на имя исполняемого файла.

Запустить написанный .launch файл вы можете в терминале с помощью команды roslaunch:

roslaunch hello_world talker_listener.launch

После запуска, в окне терминала вы можете увидеть, что сначала запускается ROS-Мастер, потом нода talker, затем нода listener. В терминал выводятся сообщения от обоих нод (см рисунок 3.5).

Для использования нод, написанных на языке Python, создадим другой .launch файл в директории launch. Назовите его talker_listener_python.launch. Его содержимое аналогично предыдущему файлу запуска:

```
<launch>
<node name="listener1" pkg="hello_world" type="listener.py"
    output="screen"/>
<node name="talker1" pkg="hello_world" type="talker.py" output="
    screen"/>
</launch>
```

Запускается этот файл уже знакомой вам командой roslaunch:

roslaunch hello_world talker_listener_python.launch

Для того чтобы визуально представить взаимосвязь нод, вы можете в отдельной вкладке терминала выполнить команду:

```
    listener_node (hello_world/listener)
    talker_node (hello_world/talker)

auto-starting new master
process[master]: started with pid [5635]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to 1736ac68-0ea5-11e8-af68-080027c6ff27
process[rosout-1]: started with pid [5648]
started core service [/rosout]
process[listener_node-2]: started with pid [5656]
process[talker_node-3]: started with pid [5670]
[ INFO] [1518296286.432156336]: hello world 0
[ INFO] [1518296286.532936503]: hello world 1
[ INFO] [1518296286.632730963]: hello world 2
[ INFO] [1518296286.733540586]: hello world 3
[ INFO] [1518296286.733994780]: I heard: [hello world 3]
[ INFO] [1518296286.833653567]: hello world 4
[ INFO] [1518296286.834120672]: I heard: [hello world 4]
[ INFO] [1518296286.932997644]: hello world 5
[ INFO] [1518296286.933576537]: I heard: [hello world 5]
[ INFO] [1518296287.032813044]: hello world 6
[ INFO] [1518296287.033458823]: I heard: [hello world 6]
[ INFO] [1518296287.133145129]: hello world 7
```

Рис. 3.5: Вывод при запуске файла talker_listener.launch.

rqt_graph

На построенном графе (рисунок 3.6) вы увидите talker_node — имя, данное ноде talker в .launch файле, и listener_node — имя ноды listener. Первая нода публикует в топик /chatter, вторая слушает его. Все отладочные сообщения от этих двух узлов собираются в топике /rosout. Это те сообщения, которые мы печатали с использованием функций отладки ROS_INFO. Нода /rqt_gui также отправляет сообщения в /rosout.

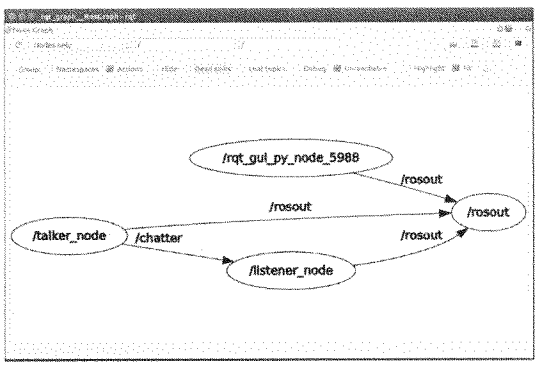

Рис. 3.6: Граф, построенный инструментом rqt_graph.

4. Программирование робота turtlesim

Turtlesim — примитивная модель робота в ROS. Представляет из себя двумерную модель всенаправленного (omnidirectional) робота. Этот робот устанавливается вместе с ROS. Давайте научимся управлять этим роботом сначала вручную, потом из кода программы. Будем использовать для этого Python, который больше подходит для быстрого прототипирования.

4.1 Перемещение робота turtlesim

Запустим робота turtlesim. Для этого в разных вкладках терминала введите команды:

roscore
rosrun turtlesim turtlesim_node

Первая запускает ROS-мастер, вторая запускает ноду **turtlesim_node** из пакета turtlesim.

Посмотрим, какие топики сейчас активны с помощью команды:

rostopic list

Получим список топиков:

- /rosout
- /rosout_agg
- /turtle1/cmd_vel
- /turtle1/color_sensor
- /turtle1/pose

Для того чтобы перемещать робота, нужно отправлять значения линейной и угловой скорости в топик /**cmd_vel**. Для того чтобы узнать какой тип данных передавать по этому топику, наберите команду:

rostopic type /turtle1/cmd_vel

Вы получите тип **geometry_msgs/Twist**. Именно переенные этого типа нужно отправлять в топик, чтобы робот начал перемещеаться. Узнать подробнее, что это за тип данных, можно с помощью команды **rosmsg show**:

rosmsg show geometry_msgs/Twist

```
robot@robot-pc:~$ rosmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Рис. 4.1: Вывод команды rosmsg show geometry_msgs/Twist.

В выводе этой команды (см рисунок 4.1) мы увидим, что этот тип данных содержит в себе два подраздела: линейную скорость и угловую скорость. Если мы зададим линейную составляющую скорости робота, он будет двигаться вперед или назад. В turtlesim мы можем установить только компонент linear.x, потому что он может двигаться только в направлении оси x; вдоль y и z движение невозможно. Также мы можем установить значение компонента angular.z для поворота робота вокруг своей оси. Сформируем сообщение этого типа и отправим его в топик с помощью консольной команды:

rostopic pub /turtle1/cmd_vel geometry_msgs/Twist
"linear: x:3.0 y:0 z:0 angular: x:0 y:0 z:0.5"

После ввода этой команды черепашка начнет движение по окружности.

Теперь напишем ноду, в коде которой будем формировать и отправлять сообщения в топик робота. Для этого создайте файл move_turtle.py. Его можно создать в папке script внутри уже имеющегося у вас пакета hello_world. Или создать новый пакет, и уже в нем создать папку script. Содержимое файла move_turtle.py:

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist
import sys

def move_turtle(lin_vel,ang_vel):
    rospy.init_node('move_turtle', anonymous=False)
    pub = rospy.Publisher('/turtle1/cmd_vel', Twist, queue_size
        =10)
    rate = rospy.Rate(10) # 10hz
    vel = Twist()
    while not rospy.is_shutdown():
```



```
        vel.linear.x = lin_vel
        vel.linear.y = 0
        vel.linear.z = 0
        vel.angular.x = 0
        vel.angular.y = 0
        vel.angular.z = ang_vel
        rospy.loginfo("LinearVel=%f: AngularVel=%f", lin_vel,
            ang_vel)
        pub.publish(vel)
        rate.sleep()
if __name__ == '__main__':
    try:
        move_turtle(float(sys.argv[1]), float(sys.argv[2]))
    except rospy.ROSInterruptException:
        pass
```

Этот код принимает линейную и угловую скорость в качестве аргументов командной строки. Внутри кода аргументы командной строки можно принимать с помощью модуля sys языка Python. Функция move_turtle() получает значения скоростей, формирует сообщение и отправляет его в топик /turtle1/cmd_vel.

Не забудьте сделать этот исполняемым. Запустите ноду с помощью следующих команд в разных вкладках терминала:

roscore
rosrun turtlesim turtlesim_node
rosrun hello_world move_turtle.py 0.2 0.1

Первой командой запускается ROS-Мастер, второй запускается симуляция робота-черепашки. Третьей запускается написанная нода с аргументами командой строки. На работа таким образом передается линейная скорость 0.2 м/с и угловая скорость 0.1 рад/с. При запуске вы увидите что робот поехал по окружности (рисунок 4.2).

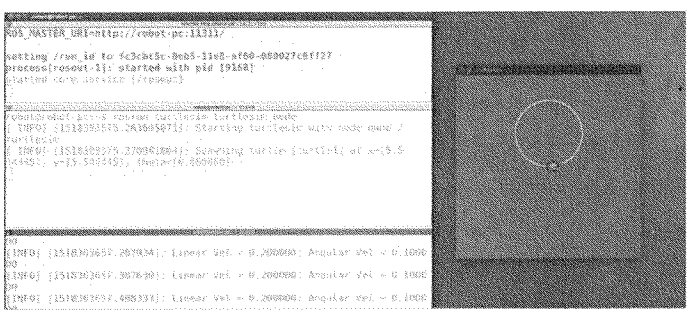

Рис. 4.2: Результат выполнения ноды move_turtle.py.

4.2 Получение положения робота

После того, как получилось управлять роботом по скорости, посмотрим как узнать его текущее положение. Информацию о положении можно взять из топика /turtle1/pose. Закройте ноду move_turtle.py и перезапустите ноду turtlesim_node. Просмотрим что публикуется в топик с помощью команды **rostopic echo**:

rostopic echo /turtle1/pose

Данная команда будет выводить в консоль текущее положение робота (координаты x, y и угол поворота theta) и угловую и линейную скорости (см рисунок 4.3).

```
robot@robot-pc:~$ rostopic echo /turtle1/pose
x: 5.544444561
y: 5.544444561
theta: 0.0
linear_velocity: 0.0
angular_velocity: 0.0
---
x: 5.544444561
y: 5.544444561
theta: 0.0
linear_velocity: 0.0
angular_velocity: 0.0
---
x: 5.544444561
y: 5.544444561
theta: 0.0
linear_velocity: 0.0
angular_velocity: 0.0
---
```

Рис. 4.3: Позиция робота из топика /turtle1/pose.

Для того чтобы получать положение робота в исходном коде, нужно подписаться на топик **turtle1/pose**. Посмотреть какой тип данных передается через этот топик можно с помощью команды:

rostopic type /turtle1/pose

В выводе будет тип **turtlsim/Pose**. А чтобы узнать из чего состоит этот тип, наберите в консоли:

rosmsg show turtlesim/Pose

В терминал будет выведено, что этот тип состоит из 5 компонент типа float32 (рисунок 4.4): координаты x, y, theta, линейная и угловая скорости.

Создадим новый файл с кодом на языке Python в папке script. Назовите его move_turtle_get_pose.py. По сути это будет прошлый файл, в который добавим функцию вывода текущего положения робота. Схема взаимодействия между нодами схематично изображена на рисунке 4.5.


```
robot@robot-pc:~$ rosmsg show turtlesim/Pose
float32 x
float32 y
float32 theta
float32 linear_velocity
float32 angular_velocity
```

Рис. 4.4: Описание типа turtlsim/Pose

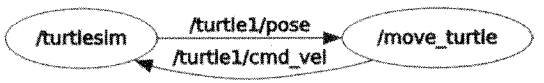

Рис. 4.5: Схема работы ноды move_turtle_get_pose.py

В исходном коде этого файла стоит отметить добавленный экземпляр класса Subscriber и добавленную, по сравнением с предыдущим файлом, callback-функцию в которой выводится на экран положение робота:

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
import sys
def pose_callback(pose):
  rospy.loginfo("Robot X=%f : Y=%f : Z=%f\n", pose.x, pose.y,
      pose.theta)

def move_turtle(lin_vel,ang_vel):
  rospy.init_node('move_turtle', anonymous=True)
  pub = rospy.Publisher('/turtle1/cmd_vel', Twist, queue_size
      =10)
  rospy.Subscriber('/turtle1/pose',Pose, pose_callback)
  rate = rospy.Rate(10) # 10hz
  vel = Twist()
    while not rospy.is_shutdown():
    vel.linear.x = lin_vel
    vel.linear.y = 0
    vel.linear.z = 0
    vel.angular.x = 0
    vel.angular.y = 0
    vel.angular.z = ang_vel
    rospy.loginfo("LinearVel=%f: AngularVel=%f", lin_vel,
        ang_vel)
    pub.publish(vel)
    rate.sleep()
if __name__ == '__main__':
  try:
    move_turtle(float(sys.argv[1]),float(sys.argv[2]))
  except rospy.ROSInterruptException:
    pass
```

Запустите написанный скрипт, выполнив в разных терминалах команды:

roscore
rosrun turtlesim turtlesim_node
rosrun hello_world move_turtle_get_pose.py 0.2 0.1

Вы получите робота, который едет с заданной ему скоростью и выводит в консоль своё положение (рисунок 4.6).

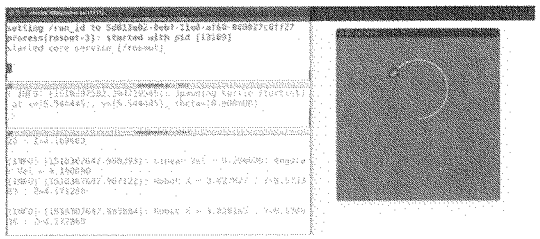

Рис. 4.6: Результат выполнения ноды move_turtle_get_pose.py.

Раз мы можем отправлять на робота команды скорости и получать его положение, то мы можем отправлять на него команды на какую дистанцию ему нужно проехать.

4.3 Перемещение робота с контролем дистанции

На основе предыдущего файла создадим новый скрипт, назовем его move_distance.py. Данная нода будет принимать три аргумента командной строки. Третий аргумент — длина траектории, которую нужно пройти роботу. Когда пройденный путь достигнет заданного значения, робот остановится. В коде это реализуется проверкой дистанции внутри цикла и в случае положительного результата — выхода из цикла командой break:

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
import sys
def pose_callback(pose):
    global robot_x
    rospy.loginfo("Robot X=%f " % pose.x)
    robot_x = pose.x
def move_turtle(lin_vel,ang_vel,distance):
    global robot_x
    rospy.init_node('move_turtle', anonymous=True)
    pub = rospy.Publisher('/turtle1/cmd_vel', Twist, queue_size
        =10)
    rospy.Subscriber('/turtle1/pose',Pose, pose_callback)
    rate = rospy.Rate(10) # 10hz
    vel = Twist()
```



```
    while not rospy.is_shutdown():
        vel.linear.x = lin_vel
        vel.linear.y = 0
        vel.linear.z = 0
        vel.angular.x = 0
        vel.angular.y = 0
        vel.angular.z = ang_vel
        if(robot_x >= distance):
            rospy.loginfo("Robot Reached destination")
            rospy.logwarn("Stopping robot")
            break
        pub.publish(vel)
        rate.sleep()
if __name__ == '__main__':
    try:
        move_turtle(float(sys.argv[1]), float(sys.argv[2],float(sys.
            argv[3]))
    except rospy.ROSInterruptException:
        pass
```

Запустите эту и остальные ноды введя в разных вкладках терминала команды:

roscore
rosrun turtlesim turtlesim_node
rosrun hello_world move_distance.py 0.2 0.0 8.0

Аргументами последней ноды являются линейная скорость вдоль оси x (0.2 м/с), скорость поворота (0.0 рад/с) и дистанция, которую надо проехать роботу (8.0 м). Результат запуска представлен на рисунке 4.7.

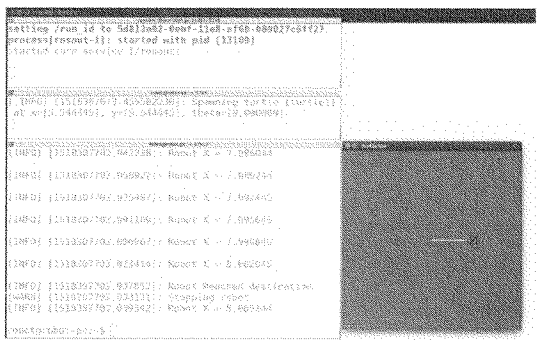

Рис. 4.7: Результат выполнения ноды move_distance.py.

4.4 Использование сервисов turtlesim

Разберем как использовать сервисы и параметры ROS из исходного кода. Для этого реализуем следующий пример. В коде будем произвоть

сброс положения робота случайным образом меять цвет фона. Сброс рабочей области выполняется с помощью сервиса ROS, а изменение цвета — с помощью параметра ROS. Когда рабочее пространство сбрасывается, робот возвращается в исходное положение, и модель черепашки меняется на другую.

Получить список активных сервисов можно с помощью команды:

rosservice list

```
robot@robot-pc:~$ rosservice list
/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/spawn
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level
```

Рис. 4.8: Список сервисов ноды turtlesim.

Вы увидите список существующих в системе сервисов, в том числе сервис textbf/reset. При вызове этого сервиса будет сбрасываться рабочее пространство робота. Узнать какой тип получает данный сервис можно с помощью следующей команды:

rosservice type /reset

Вы увидите что сервис на вход получает тип **std_srvs/Empty**. Узнать информацию об этом типе можно с помощью инструмента **rossrv** и его команды **show**:

rossrv show std_srvs/Empty

Будет выведена пустая строка, что означает, что тип **std_srvs/Empty** имеет нулевое содержимое, следовательно, для вызова сервиса /**reset** не требуется отправлять сообщения какого-либо типа.

Мы также можем получить список активных параметров ROS с помощью команды:

rosparam list

В появившемся списке (рисунок 4.9) можно увидеть три параметра, отвечающие за цвет фона: **background_b**, **background_g**, **background_r**. Если изменить эти параметры, то цвет фона изменится. После установки нового цвета, следует сбросить рабочее пространство (/**reset**), чтобы применялись новые настройки цвета.


```
robot@robot-pc:~$ rosparam list
/background_b
/background_g
/background_r
/rosdistro
/roslaunch/uris/host_robot_pc__44371
/rosversion
/run_id
```

Рис. 4.9: Список параметров ноды turtlesim.

Получить значения параметров можно с помощью инструмента **rosparam** и его команды **get**:

rosparam get /background_b

Кроме того, получить цвет фона под роботом можно и с помощью топика **color_sensor**. Для этого при запущенной ноде turtlesim наберите команду отображения содержимого топика:

rostopic echo /turtle1/color_sensor

Напишем ноду turtle_service_param.py, при вызове которой фон будет меняться на случайный и будет происходить сброс положения робота для применения настроек. Цвет будем менять случайно задавая значения для параметров **background_b**, **background_g**, **background_r** в интервале от 0 до 255.

```
#!/usr/bin/env python
import rospy
import random
from std_srvs.srv import Empty
def change_color():
    rospy.init_node('change_color', anonymous=True)
    rospy.set_param('/background_b',random.randint(0,255))
    rospy.set_param('/background_g',random.randint(0,255))
    rospy.set_param('/background_r',random.randint(0,255))
    rospy.wait_for_service('/reset')
    try:
        serv = rospy.ServiceProxy('/reset',Empty)
        resp = serv()
        rospy.loginfo("Executed service")
        except rospy.ServiceException, e:
            rospy.loginfo("Service call failed: %s" %e)
        rospy.spin()
if __name__ == '__main__':
    try:
        change_color()
    except rospy.ROSInterruptException:
        pass
```

Проверить правильность работы ноды можно запустив её, после того, как будет запущен ROS-Мастер и нода turtlesim:

roscore

rosrun turtlesim turtlesim_node
rosrun hello_world turtle_service_param.py

После запуска написанной ноды робот изменит свой внешний вид, переместится в центр поля и цвет фона изменится на случайный (рисунок 4.10).

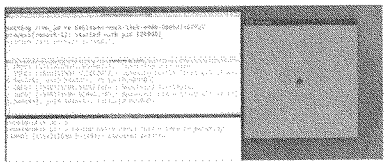

Рис. 4.10: Сброс поля и изменение цвета.

Таким образом зная как работать на роботе-черепашке, манипуляции с топиками, параметрами и сервисами на реальных роботах будут аналогичными. В следующем разделе объясняется, как сделать те же операции на модели реального робота.

5. Программирование робота Turtlebot

5.1 Установка и запуск робота Turtlebot

В ROS реализованы множество существующих сегодня роботов. В том числе робот Turtlebot 2. TurtleBot — это недорогие роботы, которые используются для обучения и исследований. Рассмотрим, как устанавливать пакеты для использования этого и прочих роботов и как запускать этого робота в симуляции.

Пакеты робота Turtlebot доступны в официальном ROS репозитории, поэтому установить их можно с помощью менеджера пакетов **apt**, введя команду:

sudo apt-get install ros-kinetic-turtlebot-gazebo
ros-kinetic-turtlebot-simulator ros-kinetic-turtlebot-description
ros-kinetic-turtlebot-teleop

После установки всех необходимых пакетов, запустить симуляцию робота Turtlebot можно с помощью команды:

roslaunch turtlebot_gazebo turtlebot_world.launch

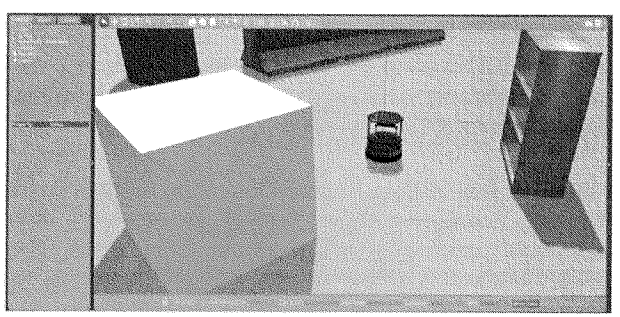

Рис. 5.1: Симуляция робота Turtlebot в Gazebo.

Среда симуляции Gazebo может не иметь в папке моделей необходимую модель окружающего мира. В таком случае первый запуск симулятора может занять много времени, потому что необходимые модели будут выкачиваться из интернета. После того как симулятор запуститься, вы увидите среду, состоящую из нескольких препятствий и робота Turtlebot, как на рисунке 5.1.

Для того, чтобы перемещать робота в ручном режиме, запустите в новой вкладке терминала ноду телеоперации:

roslaunch turtlebot_teleop keyboard_teleop.launch

После запуска этой ноды, вы сможете управлять роботом нажимая клавиши клавиатуры, отправляя тем самым сообщения на его топики скоростей (линейной и угловой). Кроме того, прочитая сообщения в терминале, вы можете увеличивать/уменьшать эти значения (см. рисунок 5.2).

```
Control Your Turtlebot!
---------------------------
Moving around:
   u    i    o
   j    k    l
   m    ,    .

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%
space key, k : force stop
anything else : stop smoothly

CTRL-C to quit

currently:      speed 0.2       turn 1
```

Рис. 5.2: Окно управления Turtlebot в режиме телеоперации.

Для того, чтобы остановить робота используйте клавишу "пробел". Чтобы закрыть выполнение ноды используйте сочетание Ctrl+C.

5.2 Перемещение робота Turtlebot

Чтобы переместить робота воспользуемся написанной ранее нодой move_distance.py. Если вывести все топики, которые используются при запуске симуляции робота Turtlebot, можно понять что команды скорости на робота подаются в топик **cmd_vel_mux/input/teleop**. Тип сообщений в этом топике уже вам знаком: **geometry_msgs/Twist**.

Робот передает свое положение в топике **/odom**, при этом используется тип **nav_msgs/Odometry**. Посмотреть подробную информацию об этом типе можно с помощью команды:

rosmsg show nav_msgs/Odometry

Для использования этих типов, в коде потребуется импортировать их модули. Логика движения робота такая же, как и у turtlesim. Начальное положение робота [0,0,0].

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
import sys
robot_x = 0
def pose_callback(msg):
  global robot_x
  robot_x = msg.pose.pose.position.x
  rospy.loginfo("Robot X = %f\n",robot_x)
def move_turtle(lin_vel,ang_vel,distance):
  global robot_x
  rospy.init_node('move_turtlebot', anonymous=False)
  pub = rospy.Publisher('/cmd_vel_mux/input/teleop', Twist,
      queue_size=10)
  rospy.Subscriber('/odom', Odometry, pose_callback)
  rate = rospy.Rate(10) # 10hz
  vel = Twist()
  while not rospy.is_shutdown():
    vel.linear.x = lin_vel
    vel.linear.y = 0
    vel.linear.z = 0
    vel.angular.x = 0
    vel.angular.y = 0
    vel.angular.z = ang_vel
    if(robot_x >= distance):
      rospy.loginfo("Robot Reached destination")
      rospy.logwarn("Stopping robot")
      break
    pub.publish(vel)
    rate.sleep()
if __name__ == '__main__':
  try:
    move_turtle(float(sys.argv[1]), float(sys.argv[2],float(sys.
        argv[3]))
  except rospy.ROSInterruptException:
    pass
```

Запустите этот код. Для этого сначала запустите симуляцию робота Turtlebot в Gazebo, после чего запустите ноду **move_turtlebot.py** с тремя параметрами:

roslaunch turtlebot_gazebo turtlebot_world.launch
rosrun hello_world move_turtlebot.py 0.2 0.0 3.0

В симуляции робот проедет 3 метра после чего остановится (см рисунок 5.3).

По аналогичной логике, вы можете находить препятствия вокруг робота, используя данные с лазерного сканера робота. Для этого воспользуйтесь из кода информацией из топика **/scan**. Тип сообщений в этом

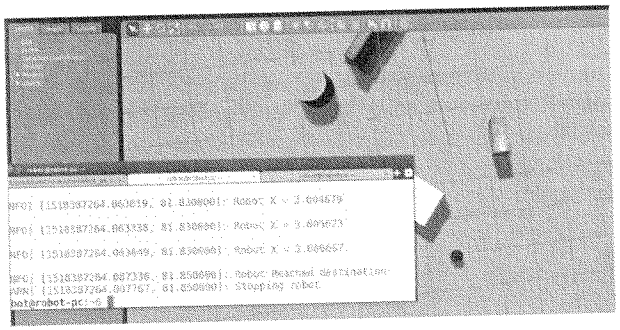

Рис. 5.3: Turtlebot проезжает 3 метра в симуляции.

топике: **/sensor_msgs/LaserScan**. Используя эту информацию попробуйте самостоятельно написать ноду, останавливающую робота при приближении к препятствиям.

Заключение

В данном пособии разобраны примеры программирования роботов с использованием ROS. Использованный в симуляции код практически без изменений может быть перенесен на реальных роботов. Поэтому на сегодняшний день все серьезные производители роботов обеспечивают наличие рабочих ROS-моделей для своих роботов. И специалисты программирования ROS становится в мире все более востребованными.

Продолжайте изучение Робототехнической Операционной Системы!

Все предложения и замечания просьба высылать по адресу *lavrenov@it.kfu.ru*.

Список литературы

1. *Quigley M.* Programming Robots with ROS: a practical introduction to the Robot Operating System / Quigley M., Gerkey B., Smart W. D. – "O'Reilly Media, Inc. 2015. — 448 с.
2. *Koubâa A.* Robot Operating System (ROS) / Quigley M. — Verlag : Springer, 2017. — 728 с.
3. *Martinez A.* Learning ROS for robotics programming/ Martinez A., Fernández E. — Packt Publishing Ltd, 2013. — 332 с.
4. ROS Tutorials. — URL:http://wiki.ros.org/ROS/Tutorials (дата обращения 15.05.2019)

Учебное издание

Лавренов Роман Олегович
Магид Евгений Аркадьевич

ПРОГРАММИРОВАНИЕ В РОБОТОТЕХНИЧЕСКОЙ ОПЕРАЦИОННОЙ СИСТЕМЕ (ROS)

Учебно-методическое пособие

Подписано в печать 24.05.2019.
Бумага офсетная. Печать цифровая.
Формат 60x84 1/16. Гарнитура «Book Antiqua». Усл. печ. л. 2,33.
Уч.-изд. л. 1,15. Тираж 50 экз. Заказ 216/5

Отпечатано с готового оригинал-макета
в типографии Издательства Казанского университета

420008, г. Казань, ул. Профессора Нужина, 1/37
тел. (843) 233-73-59, 233-73-28