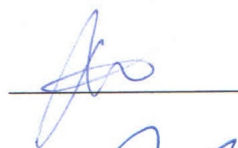



МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по педагогической практике
аспиранта

Аспирант

направление 09.06.01
специальность 05.13.11

Филатов Антон Юрьевич

Руководитель д.т.н., проф

Экало Александр Владимирович

Санкт-Петербург
2018

ЗАДАНИЕ НА ПЕДАГОГИЧЕСКУЮ ПРАКТИКУ АСПИРАНТА

Аспирант Филатов Антон Юрьевич
направление 09.06.01
специальность 05.13.11
Тема практики: технологическая практика аспиранта

Задание на практику:

Разработать методические указания для курса «Системы реального времени на основе Linux». Методические указания должны теоретические вставки для помощи студентам в выполнении лабораторных работ.

Сроки прохождения практики: 01.09.2017 – 10.01.2018

Дата сдачи отчета: 11.01.2018

Дата защиты отчета: 11.01.2018

Аспирант

Руководитель



Филатов А. Ю.

Экало А. В.

АННОТАЦИЯ

Робототехника – одна из самых активно развивающихся областей. Особую красоту результата и, одновременно с этим, сложность реализации обеспечивает слияние схемотехники, электротехники и программирования. Для успешного программирования роботов необходимо уметь пользоваться операционными системами, работающими в реальном времени. Отличительной особенностью таких систем является потребность обрабатывать входные сигналы за установленный заранее промежуток времени, и превышение этого промежутка недопустимо. Среди фреймворков, позволяющих программировать поведение роботов в реальном времени можно выделить Robot Operating System (ROS).

SUMMARY

Robotics is one of the most rapidly developing areas. The special beauty of the result and, at the same time, the complexity of implementation is ensured by the merger of circuitry, electrical engineering and programming. For successful robot programming, you must be able to use real-time operating systems. A distinctive feature of such systems is the need to process input signals for a pre-set period of time, and exceeding this period is unacceptable. Among the frameworks that allow you to program the behavior of robots in real time, one can distinguish Robot Operating System (ROS).

ВВЕДЕНИЕ

В ROS все базируется на концепции node-ов. Каждый исполнитель – это одна node-a. Что бы ни делал модуль, чем бы он ни занимался – его можно представить в виде node-ы – экземпляра рабочего класса.

Node-ы должны обмениваться друг с другом информацией. Всего существует два глобальных способа обмена информацией между узлами: общая память и передача сообщений. В силу того, что выбрана абстракция независимых нод, то обеспечить им всем общую память оказывается малоэффективно. Поэтому в ROS был выбран механизм передачи сообщений. Сообщение – это представление структуры данных, которое генерируется и наполняется информацией на одной ноде и передается в “эфир”, где подхватывается другой нодой, и записанная информация начинает использоваться. Абстракцию “эфира” в ROS выполняют топики. Топик – это очередь сообщений определенного типа. Таким образом, когда одна нода генерирует сообщение, то она помещает его в такую очередь, а когда другая нода нуждается в информации из вне (например, нода-построитель графика ждет новые координаты точки), то из этой очереди сообщение считывается. Но остается в этом случае вопрос, что делать, когда нод очень много, а, допустим, нода-источник сообщений всего одна? Так, например, это очень распространенная ситуация в робототехнике, когда одна нода робота считывает показания со всех датчиков робота и передает их на дальнейшую обработку. Обработкой могут заниматься несколько нод: одна может выполнять коррекцию, другая отображать “сырые данные”, третья – логгировать сообщения. В этом случае делать для каждой связи между нодами свой топик оказывается неэффективным. Поэтому по факту создается только одна очередь на один тип сообщений. Причем не важно, сколько нод генерируют сообщения такого типа, а сколько читают. Топик всего один. Благодаря этому возможно сделать так, чтобы нода, подключенная позднее остальных могла принять и обработать ранее сгенерированные сообщения. Таким образом, ноды получают независимы от всей окружающей среды. Каждая из них не знает о существовании других. Один узел знает только какие сообщения он публикует (publish), а на какие подписан (subscribe), и его не интересует кто прочитает его сообщение, кто послал принятое им сообщение. Однако механизм передачи сообщений – не единственный способ общения. Есть также и способ сообщить информацию от одной конкретной ноды к другой. Этот механизм называется “служебный вызов” (service call). Отличия service call от message:

1. Служебный вызов двунаправлен. То есть сначала вызов инициируется и передается другой нодой, потом ожидается подтверждение. В то время как сообщения не нуждаются в подтверждении и транслируются без заботы быть кем-то принятыми.

2. Служебный вызов предназначен для связи один-к-одному. В то время как сообщения – это способ связи многие-со-многими, где не важно сколько нод публикуют сообщения, а сколько подписаны, механизм служебных вызовов строго конкретизирует какая нода будет связана с какой.

Для передачи системного вызова происходит реализация общения между сервером и клиентом. Одна нода называется клиентом (client) и генерирует запрос (request). Другая нода называется сервером (server) получает этот запрос, проводит его обработку и на основании произведенных действий генерирует ответ (response). Вид этих запросов или ответов не отличается от сообщений: это также некоторая структура данных с единственной разницей – она разделена на две части: часть запроса и часть ответа. Причем очереди сообщений у системных вызовов нет. От нее можно отказаться, так как каждый клиент блокирует сервер на прием вызовов других клиентов, поэтому каждый клиент хранит свое сообщение сам до тех пор, пока сервер не будет доступен для связи.

Теперь поговорим о разрешимости имен. В ROS передача сообщений и служебных вызовов происходит по имени топика, если речь идет о сообщениях, или по имени ноды, если речь идет о системных вызовах. И что будет, если создать топики с одинаковыми именами, но разным содержимым? Или создать ноды с одинаковыми именами? Здесь следует сказать о строении имени в ROS. Оно состоит из трех составляющих:

1. имя пакета;
2. имя исполняемого файла;
3. имя ноды.

Пакет (package) – это абстрактное объединение нод, структур сообщений и структур служебных вызовов.

Имя исполняемого файла (executable) – что за исполняемый файл используется при создании ноды.

Имя ноды – непосредственное имя ноды, запускаемое в ROS.

При этом при запуске ноды указывается из какого она пакета, а также, что за исполняемый файл используется. При этом как таковое имя можно не указывать – оно присвоится автоматически. Однако, при попытке создать еще

одну ноду без указания имени, создастся нода с тем же именем, которая сынициирует уничтожение первой ноды.

При обращении к топику используется имя пакета, в котором этот топик определен, а также непосредственно имя топики. Таким образом можно создать два топики с одинаковым названием но в рамках двух пакетов. В случае, если два топики с одним именем будут описаны в рамках одного пакета, при попытке компиляции будет сформирована ошибка схожая с ошибкой, получаемой при определении класса C++ с тем же именем.

2. ОБ ОКРУЖЕНИИ

Для того чтобы начать работать с ROS, его сначала необходимо скачать и установить.

ROS имеет несколько версий. Но, в отличии от классического числового представления новой версии, новые версии ROS выпускаются под новой буквой английского алфавита. Так, второй версией ROS был не ROS 2.0 а ROS box turtle, за ним был ROS C Turtle. Последними на данный момент являются версии ROS indigo, ROS jade и ROS kinetic. Работа будет производиться на ROS kinetic, и именно его предполагается установить. На вики по ROS есть подробная инструкция, как это сделать. Давайте прокомментируем каждое действие, чтобы понять, что и зачем делается

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

Эта команда предназначена для того, чтобы установить пути, где искать пакеты ros.

```
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key 0xB01FA116
```

После выполнения этой команды операционная система обменивается ключами шифрования с удаленным репозиторием, откуда в дальнейшем будет скачан пакет ROS.

```
$ sudo apt-get update
```

Проверка на то, что система обновлена. Эта команда не связана с репозиторием ROS. Ее выполнение важно только для того, чтобы все вспомогательные компоненты были в актуальном состоянии.

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

Непосредственно скачивание пакета ROS. Рекомендуется скачивать именно версию “full”, поскольку в этом случае не придется задумываться, присутствуют ли в системе те или иные вспомогательные пакеты ROS. Если система, на

которую устанавливается ROS, критична к количеству свободной памяти на жестком диске, можно скачать desktop или base версии. В этом случае разработчик должен точно знать, чем он пользуется, и скачивать необходимые ему пакеты самостоятельно.

Можно скачивать только отдельные пакеты с помощью команды

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

Если была выполнена команда `sudo apt-get install ros-kinetic-desktop-full`, то отдельно качать пакеты придется только в случае появления новых.

```
$ sudo rosdep init  
$ rosdep update
```

Перед использованием ROS необходимо инициализировать `rosdep`. Эта утилита позволяет автоматически устанавливать зависимости в разрабатываемых проектах; кроме того, некоторые компоненты ROS без `rosdep` работать не будут.

```
$ echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc  
$ source ~/.bashrc
```

Первая команда добавляет строчку в файл `bashrc`, вторая – применяет изменения. Остановимся на этом подробнее. Для того, чтобы выполнялись особые команды ROS (как, например `roscd`), необходимо «объяснить» системе, где искать эти команды. Это делается при помощи команды `source`. Конечно, можно каждый раз при запуске системы вводить команду

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

Однако есть способ автоматизировать вызов команды `source` – записать вызов этой команды в `bashrc`. Важно знать, что запись команды в `bashrc` необходимо сделать лишь один раз. Далее при каждом новом запуске системы будут выполнены все команды из `bashrc`, в том числе и требуемый `source`.

```
$ sudo apt-get install python-rosinstall
```

Этой командой устанавливается утилита `rosinstall`, которая поставляется отдельно от ROS. Эта утилита позволяет упростить скачивание пакетов ROS.

После выполнения вышеперечисленных действий, ROS будет установлен на вашей системе.

Прежде чем идти дальше, необходимо рассказать о представлении компонентов ROS в файловой системе.

По умолчанию ROS устанавливается в директорию `/opt/ros/kinetic/`. Компоненты ROS называются *пакетами*. Пакет – это условно независимый проект. Независимый – потому что его компоненты могут быть запущены и

выполнены, как самостоятельные программы. Условно – потому что зависимости в нем все-таки есть. Зависимости от стандартных пакетов ROS или других пакетов. Для упрощения навигации между пакетами существует команда `roscd`. Ее преимущество над `cd` в том, что можно перемещаться между пакетами ROS не по полному пути, а только по названию пакета. То есть вместо

```
$ cd /opt/ros/kinetik/share/my_package
```

можно использовать

```
$ roscd my_package
```

Теперь попробуем создать свой пакет. В ROS существует два способа компиляции и сборки пакетов. Первый способ можно нестрого называть “*catkin*”, а второй “*roscd*”. В дальнейшем всегда будем пользоваться первым способом, поскольку он требует меньше настроек и больше делает автоматически.

При использовании *catkin* необходимо запомнить три команды: `catkin_init_workspace`, `catkin_create_package` и `catkin_make`.

Создать новую директорию, где будет располагаться новый проект, и инициализировать новое рабочее пространство можно с помощью следующих команд.

```
$ mkdir -p ~/<put_your_path_here>/workspace/src  
$ cd ~/<put_your_path_here>/workspace/src  
$ catkin_init_workspace
```

В ответ в терминале появится сообщение со следующим текстом:

```
Creating symlink "/<put_your_path_here>/src/CMakeLists.txt" pointing to  
"/opt/ros/kinetic/share/catkin/cmake/toplevel.cmake"
```

Далее следует выполнить следующие команды:

```
$ cd ..  
$ catkin_make
```

Теперь следует более подробно остановиться на том, что происходило в моменты выполнения команд выше. Первой командой была создана папка `workspace/` и в ней подпапка `src/`. Второй командой текущая директория была перенесена в папку `workspace/src/`, где вызывается `catkin_init_workspace`. Единственное, что делает эта команда – создает ссылку на CMake файл, находящийся где-то в недрах ROS.

Последняя команда создала две новые директории `workspace/build` и `workspace/devel`, наполнив их содержимым, необходимым для сборки проекта. Следует заметить, что `catkin_make` должен вызываться из корневой директории

рабочей области (а не из src/).

Следует отметить, что при выполнении команды `catkin_make` в системе происходит что-то похожее на указанную ниже последовательность команд:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
$ make install
```

Но это все скрыто от пользователя и выполняется автоматически. Далее необходимо пояснить, почему был выбран такой способ представления и создания программных компонентов. Для этого, для начала, следует разобраться в структуре пакетов ROS.

Существует соглашение, согласно которому необходимо строить иерархию папок следующим образом:

```
workspace/
  src/
    CMakeLists.txt
  devel/
  ...
  build/
  ...
```

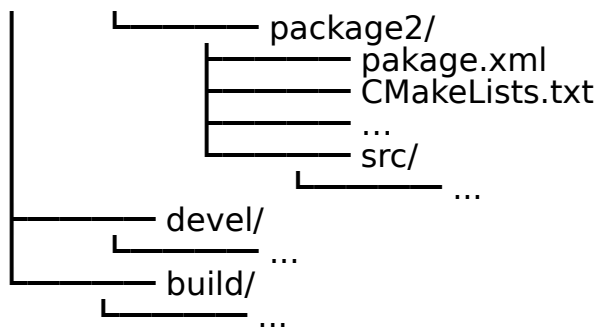
Название “workspace” было выбрано произвольно — вместо него можно использовать любое допустимое название папки. Папка “src” обязательно должна называться именно так. Если она будет называться по-другому или вообще отсутствовать, `catkin_make` будет выдавать различные ошибки.

Особо важным сгенерированным файлом является `devel/setup.bash`. Для того, чтобы написанная программа запускалась в контексте ROS, необходимо сначала вызвать

```
$ source devel/setup.bash
```

После вызова этой команды, описанные пакеты будут индексированы для ROS. Под “описанными пакетами” понимаются будущие пакеты, которые будут описаны в инициализированной рабочей области. После создания нескольких пакетов иерархия папок будет выглядеть следующим образом.

```
workspace/
├── src/
│   ├── CMakeLists.txt
│   └── package/
│       ├── package.xml
│       ├── CMakeLists.txt
│       ├── ...
│       └── src/
│           └── ...
```



Необходимо заметить на будущее, что сборка каждого пакета, то есть вызов `catkin_make`, должна происходить в корневой папке рабочей области, а не в папке с пакетом.

Для того чтобы создать пакет, нет необходимости вручную создавать папку, а также все внутренние файлы, названия которых, в придачу, должны соответствовать соглашению. Достаточно вызвать команду

```
$ catkin_create_pkg <имя_пакета> [зависимость]
```

Проще всего создать проект без зависимостей, например

```
$ catkin_create_pkg <имя_пакета>
```

Эту команду необходимо выполнять находясь в папке `workspace/src/`. Тогда создастся папка с именем `<имя_пакета>`, в которой будут сгенерированы файлы `package.xml` и `CMakeLists.txt`

Далее необходимо остановиться подробнее на структуре `package.xml`. Если убрать все комментарии, то останется такая “начинка”:

```

<?xml version="1.0"?>
<package>
  <name>my_awesome_pkg</name>
  <version>0.0.0</version>
  <description>The my_awesome_pkg package</description>
  <license>TODO</license>
  <buildtool_depend>catkin</buildtool_depend>
  <export>
</export>
</package>

```

Значения всех тегов понятны из названия. Можно совершенно безболезненно менять значение `version` и `description`. Значение `name` должно совпадать с именем пакета в файле `CMakeLists.txt`. Значение `Buildtool_depend` всегда должно быть установлено `catkin`, чтобы была возможность собирать проект при помощи `catkin_make`.

В дальнейшем все комплексные примеры будут показываться на примере пакета `tiny_slam`. Подробнее про этот пакет можно узнать в соответствующей главе.

Ниже представлен немного более сложный файл package.xml, который используется в пакете tiny_slam.

```
<?xml version="1.0"?>
<package>
  <name>tiny_slam</name>
  <version>0.1.2</version>
  <description>TinySLAM ROS implementation</description>
  <maintainer email="ros@osll.ru">OSLL</maintainer>
  <license>MIT</license>
  <url type="website">http://wiki.ros.org/tiny_slam</url>
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>sensor_msgs</build_depend>
  <build_depend>tf</build_depend>
  <build_depend>message_filters</build_depend>
  <build_depend>rosbag_storage</build_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>std_msgs</run_depend>
  <run_depend>tf</run_depend>
  <run_depend>message_filters</run_depend>
  <run_depend>sensor_msgs</run_depend>
  <run_depend>rosbag_storage</run_depend>
  <export>
</export>
</package>
```

В этом примере добавляются теги <build_depend> и <run_depend>. В build depend указываются пакеты, которые нужны для сборки пакета, а в run depend – для запуска. Если пакет указан в build depend, то он всегда потребуется и в run depend. Однако, возможна ситуация, когда пакет не нужен в build depend, но требуется в run depend. С этой ситуацией мы познакомимся позже.

Зависимости можно писать “руками” в файле “package.xml”, а можно указывать при запуске catkin_create_pkg. Зависимости, указанные при вызове этой команды записываются и в run и в build depend.

3. ОПИСАНИЕ NODES, TOPICS, MESSAGES

Во всех учебниках по ROS знакомство с понятиями Node, Topic, Message начинают с примера, который называется “turtle_sim”. Работа с этим пакетом очень наглядна и помогает разобраться в том, как происходит взаимодействие между нодами.

В первую очередь необходимо помнить, что ROS – это *Robotics Operation System*, ее основное предназначение в упрощении работы с роботами. Робот – это совокупность узлов, каждый из которых принимает какие-то данные,

обрабатывает их и передает другому узлу. В ROS *нода* – это абстракция, которую можно сравнить с таким узлом робота. Физически нода – это поток, выполняющийся в системе и обменивающийся сообщениями с другими нодами.

Запуск нод в ROS осуществляется при помощи команды *roslaunch*. Синтаксис употребления этой команды выглядит так:

```
$ roslaunch <пакет> <нода> [<имя параметра>:=<значение>]
```

Причем в графе <нода> указывается не произвольное имя новой ноды, а класс-тип нод, экземпляр которого нужно запустить. Непосредственно имя ноды устанавливается через параметры.

Прежде, чем запускать эту команду, необходимо запустить ядро ROS. Поскольку ноды не существуют сами по себе, а взаимодействуют друг с другом, им необходимо окружение. В случае с настоящим роботом роль этого окружения играет операционная система этого робота. В нашем же случае в первую очередь необходимо вызвать команду

```
$ roscore
```

Эта команда будет выполняться в терминале, из которого она была вызвана. Поэтому для дальнейшей работы необходимо открыть еще одно окно терминала.

Как только запущено окружение, можно посмотреть какие ноды запустились по умолчанию. Это можно сделать, выполнив команду

```
$ rosnode list
```

И на данном этапе в консоли появится следующий вывод:

```
/rosout
```

Это означает, что по умолчанию вместе с roscore запускается одна нода с именем rosout. Причем знак слеш “/” означает, что эта нода находится в глобальном поле имен и используется без префикса-имени пакета. Нода /rosout отвечает за вывод сообщений на экран. Безусловно в компьютерной симуляции роль /rosout может выполнить std::cout, но в реальных условиях у робота не будет iostream.h или stdio.h, поэтому в ROS для вывода сообщений создана настраиваемая нода /rosout. По любой ноде можно узнать, какие сообщения она посылает и на какие подписана. Для того, чтобы узнать информацию о ноде наберите

```
$ rosnode info <имя ноды>
```

То есть в данном случае

```
$ rosnode info /rosout
```

В ответ будет получено

```
Node [/rosout]
Publications:
* /rosout_agg [rosgraph_msgs/Log]
Subscriptions:
* /rosout [unknown type]
Services:
* /rosout/set_logger_level
* /rosout/get_loggers
```

Разберемся по порядку, что представляет информация, которая была получена в ответе. Итак, первая строка представляет имя ноды, информацию о которой запрашивалась. Далее идет список топиков, в которых эта нода публикует сообщения. Топик – абстракция однонаправленной коммуникации. /rosout посылает сообщения в топик /rosout_agg типа rosgraph_msgs/Log. Сперва это может показаться странным, что rosout – нода, предназначенная только для вывода сообщений на экран, еще публикует какую-то информацию еще куда-то. Но если прикинуть ситуацию, в которой появляется необходимость сохранять все, что пишется на экран еще и в файл, становится понятно, зачем это было сделано. Ведь, допустим, если есть несколько нод, которые что-то пишут на экран, то для осуществления логирования этой всей информации на экран, необходимо вручную для каждой ноды указывать дополнительный топик, куда они будут клонировать информацию. Для того, чтобы избежать ручной работы, было принято решение создать дополнительный топик, куда будут передаваться все сообщения, полученный нодой /rosout.

Далее в списке информации о ноде следует список нод, на которых она подписана. В данном случае /rosout подписан на /rosout. Пусть вас не смущают одинаковые имена, ведь в одном случае речь идет о ноде, а во втором – о топике. Тут сразу возникает следующий вопрос: если /rosout подписан на /rosout и получает всю информацию через этот топик, то зачем нужен топик /rosout_agg, который был рассмотрен выше? Ответ прост: в топик /rosout_agg информация поступает уже отформатированная с указанием, кто, когда и откуда присылал сообщение для вывода на экран. Следующий вопрос: почему тип сообщения, на которые подписан /rosout, помечен как “unknown type”? Дело в том, что топик фактически создается тогда, когда создается нода-publisher в этот топик. Нода-subscriber не создает топик, а лишь упоминает о нем, но не представляет какого типа сообщения в этот топик будут поступать. Этот тип будет конкретизирован как только будет создана хотя бы одна нода управляющая сообщениями в этот топик.

Графически описанные топики и ноды представлены на рисунке ниже.



Рис. 3.1. Связь ноды /rosout и ее топиков

При отображении топиков и нод принято соглашение, что эллипсами отображаются ноды, а прямоугольниками – топики. Стрелочками показывается поток сообщений из топика к ноде или наоборот. Таким образом, на рис. 3.1 показано, что нода /rosout подписана на сообщения из топика /rosout и публикует свои сообщения в топик /rosout_agg.

И последнее о чем следует сказать про информацию о ноде – это список сервисов, которые эта нода представляет. Сервис – это аналог топика, только в отличие от последнего, сообщение сервиса двунаправлено и ожидает ответа на запрос. Сервис можно рассматривать как команду с некоторым откликом, которую одна нода просит выполнить у другой ноды. Подробнее о сервисах и их использовании будет рассказано позднее.

Теперь вернемся к turtle_sim и в новом окне запустим команду

```
$ rosrun turtlesim turtlesim_node
```

Что означает, что будет запущена нода типа turtlesim_node из пакета turtlesim. Имя ноды в данном случае не указывалось и оно присвоится автоматически. Теперь, если в новом терминале ввести команду

```
$ rosnodet list
```

в ответ будет получено следующее:

```
/rosout
/turtlesim
```

Таким образом создалась еще одна нода с именем turtlesim и определенная в глобальном поле имен, о чем говорит предшествующий символ слеш «/». Если в новом терминале вновь будет выполнена команда

```
$ rosrun turtlesim turtlesim_node
```

То создастся новая нода с тем же именем /turtlesim. Это вызовет аварийную остановку уже созданной ноды:

```
[ WARN] [1471851936.261860979]: Shutdown request received.
[ WARN] [1471851936.261924457]: Reason given for shutdown:      [new
node registered with same name]
```

Для того чтобы указать имя ноды, при ее создании необходимо присвоить значение переменной __name (с двумя символами нижнего подчеркивания). Таким образом, чтобы создать ноду с именем “turtle_node” необходимо выполнить команду

```
$ rosrun turtlesim turtlesim_node __name:=turtle_node
```

При запуске ноды `turtlesim_node` появится окно синего цвета с черепашкой в центре. Причем от запуска к запуску внешний вид черепашки может меняться. Ее вид выбирается произвольно. Теперь, если запросить информацию о новой ноде

```
$ rosnode info /turtle_node
```

то будет получен следующий ответ:

```
Node [/turtle_node]
Publications:
* /turtle1/color_sensor [turtlesim/Color]
* /rosout [roscpp_msgs/Log]
* /turtle1/pose [turtlesim/Pose]
Subscriptions:
* /turtle1/cmd_vel [unknown type]
Services:
* /turtle1/teleport_absolute
* /turtle_node/set_logger_level
* /turtle_node/get_loggers
* /reset
* /spawn
* /clear
* /turtle1/set_pen
* /turtle1/teleport_relative
* /kill
```

Здесь видно, что `turtlesim_node` представляет большое количество сервисов. Но нам сейчас более интересными являются топики, на которые эта нода подписывается и в какие публикуется. Подписывается она на топик `/turtle1/cmd_vel`. По сути это – топик команд перемещения в него будут посылаться команды перемещения черепашки. Поместить сообщение в топик можно с клавиатуры командой:

```
$ rostopic pub <имя топики> <тип сообщения> <сообщение>
```

В `/turtle1/cmd_vel` посылаются сообщения типа `geometry_msgs/Twist`. В этом можно убедиться, прочитав документацию по `turtlesim`-у.

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist "[2, 0, 0]" "[0, 0, 1]"
```

Это передвинет черепашку на двойку в местных координатах вперед и повернет на один радиан против часовой стрелки. Остальные числа в квадратных скобках могут быть произвольными. Нода `/turtle_node` игнорирует их и использует только первое число, как величина пути вперед и последнее, как величина поворота в радианах. Чтобы не запоминать формат сообщения, можно после типа сообщения дважды нажать клавишу `tab`. Это выпишет имена

полей и рядом начальные значения, которыми они инициализируются. Далее их можно изменить:

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist "linear:  
  x: 0.0  
  y: 0.0  
  z: 0.0  
angular:  
  x: 0.0  
  y: 0.0  
  z: 0.0"
```

Данный формат говорит о том, что сообщение типа `geometry_msgs/Twist` представляет собой две структуры названные “linear” и “angular”, каждая из которых состоит из трех вещественных переменных. Как уже было указано ранее, `/turtle_node` использует только координаты `linear.x` и `angular.z`. Для прерывания исполнения команды используйте `ctrl+c`.

Теперь, если мы посмотрим на граф нод и топиков, он будет выглядеть, как на рисунке ниже.

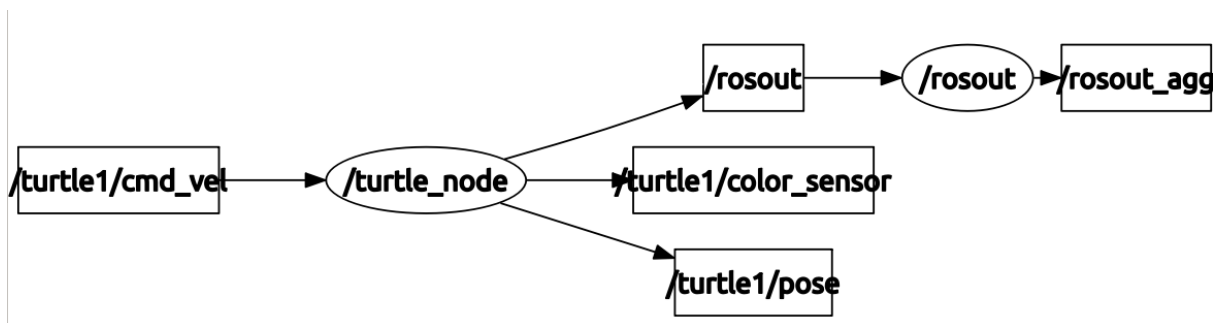


Рис. 3.2. Две ноды `/turtle_node` и `/rosout` и топики между ними

Теперь создадим ноду-publisher-а в топик `/turtle1/cmd_vel`. Например, ноду из пакета `turtle_sim` называемую `turtle_teleop_key`. Запустим ее в новом терминале так как, как вы успели заметить, каждая нода — это отдельный процесс:

```
$ rosrn turtlesim turtle_teleop_key
```

Теперь, если оставить активным терминал, в котором была запущена эта нода, и нажимать клавиши `←↑↓→` это приведет к движению черепашки в окне `turtle_node`.

Теперь, связь нод и топиков будет выглядеть так, как показано на рисунке ниже.

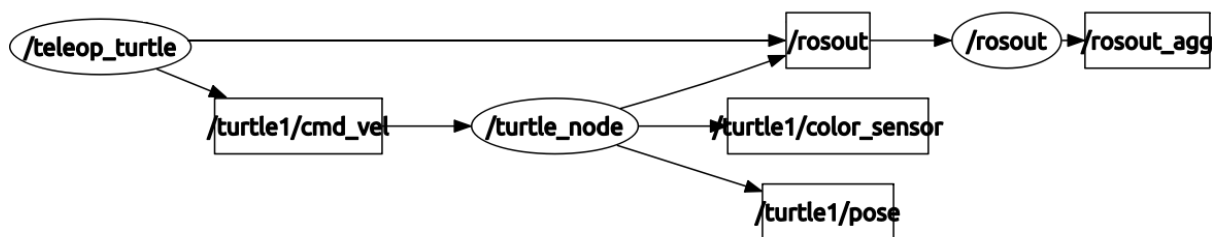


Рис. 3.3. Ноды /teleop_turtle, /turtle_node и /rosout и топики между ними

Наконец, чтобы отобразить связи существующих нод и топиков можно запустить еще графовую ноду rqt_graph из пакета rqt_graph командой

```
$ rosrn rqt_graph rqt_graph
```

Эта нода собирает всю информацию о запущенных нодах и созданных топиках и визуализирует их связи. При первичном запуске появиться окно вида, представленного на рисунке ниже.

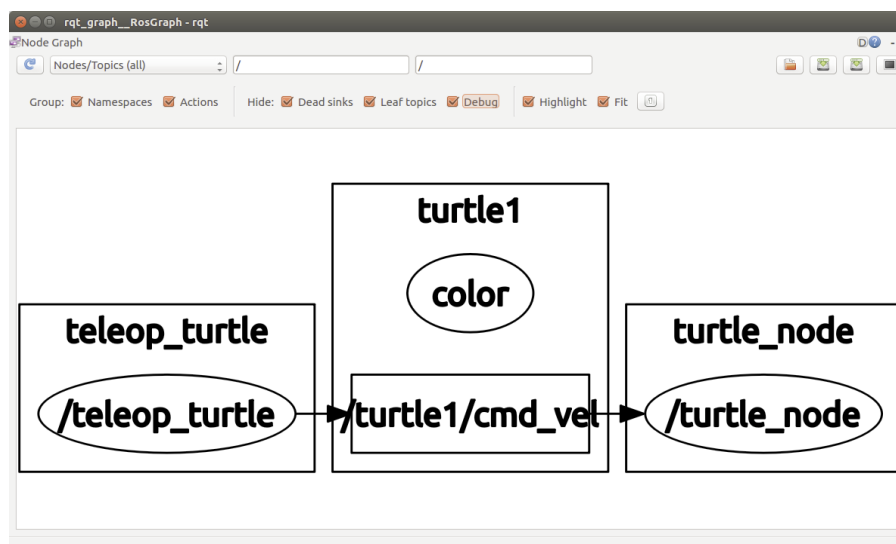


Рис. 3.4. Иллюстрация работы ноды типа rqt_graph

Если убрать галочки Namespaces (которая отображает пространство имен нод и топиков), Dead skins, Leaf topics и Debug, то можно увидеть все существующие ноды и топики. В этом случае также будет показана нода созданного графового строителя.

На рисунке 3.4 можно видеть набор запущенных важных нод и топиков между ними. В этом случае не указывается ни нода /rosout, ни нода графа /rqt_gui_py_node, ни их топики, а также не указываются “висячие” топики – в которых никто не пишет и из которых никто не читает. На рисунке 3.5 показаны все запущенные ноды и топики. Этот граф иллюстрирует все состояние робота, но является нагруженным для выделения основных компонентов.

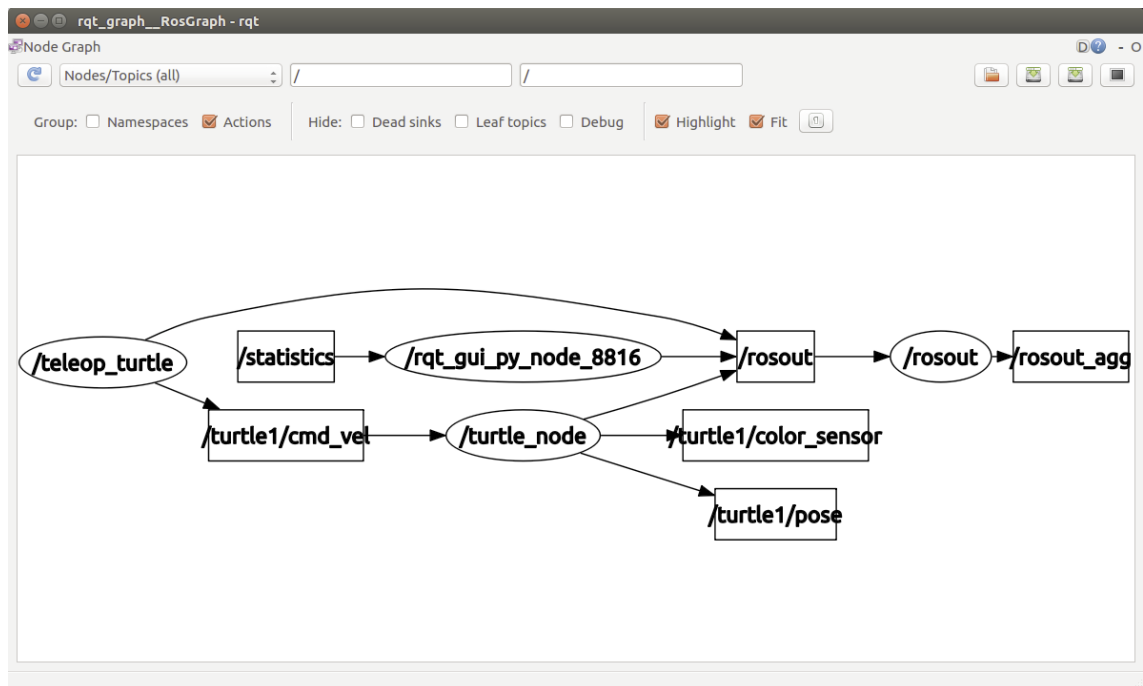


Рис. 3.5. Иллюстрация работы ноды типа rqt_graph с полным представлением

4. ROS HELLO WORLD И ПРИНЦИПЫ НАПИСАНИЯ СОБСТВЕННЫХ ПАКЕТОВ

Используя ROS, условимся, что основным языком программирования будет C++. Для того, чтобы написать простой Hello World потребуется некоторое количество усилий, но в будущем станет понятно, что такая тщательная настройка позволяет сделать проект очень гибким, кроссплатформенным и грамотно распределить логику всего приложения.

Приведем здесь один раз полную последовательность шагов, которые необходимо проделать, чтобы создать пакет `hello_world`, создать в нем исполняемую ноду `hello_node`, описать необходимые поля в `package.xml` и в `CMakeLists.txt`.

Во-первых, необходимо перейти в директорию рабочей области. Слово `workspace` предполагает, что такая директория есть только одна и все рабочие проекты создаются в подпапках этой директории. Такой подход вполне оправдан, однако его недостаток в том, что во время выполнения `catkin_make` происходит сборка всех проектов из рабочей области. Поэтому иногда разумнее создать отдельную рабочую область.

Для инициализации `workspace` необходимо создать папку, где будет содержаться рабочая область и в нее создать папку `src/`. Существует соглашение, следуя которому нельзя в названии рабочей области или пакетов использовать заглавные буквы. Можно использовать прописные буквы, цифры или знак нижнего подчеркивания. Но начинаться имя должно с прописной буквы. Итак, в папке `workspace/src/` вызовем команду

```
$ catkin_init_workspace
```

Далее запустим

```
$ catkin_create_pkg hello_world
```

Сразу изменим файл `package.xml`, поскольку мы уже четко представляем себе, что будет должно делать приложение и какие зависимости оно будет иметь. Иногда заранее это может быть неизвестно, и тогда приходится изменять `package.xml` во время разработки исходного кода проекта.

Итак, изменим содержимое `package.xml` так, как показано в листинге

```
<?xml version="1.0"?>
<package>
  <name>hello_world</name>
  <version>0.0.1</version>
  <description>The hello_world package. It contains one node
    that prints a message on a screen</description>
  <maintainer email="ant.filatov@mail.ru">anton</maintainer>
  <license>TODO</license>
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <run_depend>roscpp</run_depend>
</package>
```

В сущности содержимое этого файла понятно, о необходимости `roscpp` будет сказано позднее.

Теперь необходимо создать в папке `hello_world/` папку `src/`, где будет располагаться исходный код программы. Строго говоря это действие необязательное, и исходный код может содержаться в папке `hello_world` вперемешку со всеми остальными файлами. Однако практика выносить файлы исходного кода в отдельную папку – это хороший тон, который очень помогает в разработке крупных проектов.

Исходный код программы “Hello World!” необходимо поместить в файл `hello.cpp`. Он выглядит следующим образом

```
// This is a ROS version of the standard "hello , world"
// program.
// This header defines the standard ROS classes .
#include <ros/ros.h>

int main (int argc, char **argv) {
  // Initialize the ROS system.
  ros::init(argc, argv, "hello_world");
  // Establish this program as a ROS node.
  ros::NodeHandle nh;
  // Send some output as a log message.
  ROS_INFO_STREAM("Hello, ROS!");
```

```
}
```

Остановимся подробнее на каждой строчке этой программы

```
#include <ros/ros.h>
```

Необходимо подключать этот заголовочный файл во все проекты, использующие ROS. В нем описано пространство имен `ros`.

```
ros::init(argc, argv, "hello_world");
```

Эту команду также необходимо выполнять во всех программах, использующих `ros`. Это команда должна быть выполнена до любых команд, использующих ROS. Последний параметр функции `init` – это имя ноды по умолчанию.

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

Здесь создается объект, управляющий поведением ноды. Используя этот объект можно подписывать ноду на передачу или прием сообщений и прочее. В данном случае этот объект нужен для того, чтобы выполнялась следующая строка.

```
ROS_INFO_STREAM("Hello, ROS!");
```

На самом деле это не функция, а макрос. В теле этого макроса осуществляется передача текстового сообщения в топик `/rosout` вместе с некоторой служебной информацией, такой как, например, время передачи сообщения. Сообщение, переданное в `/rosout`, будет выведено на экран.

Теперь, когда стало понятно, что делает написанная программа, модифицируем файл `CMakeLists.txt` так, чтобы написанная программа компилировалась и выполнялась.

Исходный (сгенерированный по умолчанию) `CMakeLists.txt` выглядит следующим образом:

```
cmake_minimum_required(VERSION 2.8.3)
project(hello_world)
find_package(catkin REQUIRED)
catkin_package()
```

Изменим его так, чтобы он выглядел следующим образом (жирным выделены внесенные изменения):

```
cmake_minimum_required(VERSION 2.8.3)
project(hello_world)
find_package(catkin REQUIRED COMPONENTS roscpp)
catkin_package()
include_directories(include ${catkin_INCLUDE_DIRS})
add_executable(hello src/hello.cpp)
target_link_libraries(hello ${catkin_LIBRARIES})
```

Снова разберем последовательно все строчки кода.

```
find_package(catkin REQUIRED COMPONENTS roscpp)
```

Команда выполняет поиск catkin (который содержится внутри библиотек ROS). После удачного выполнения будут проинициализированы переменные окружения, что позволит выполнить последующую компиляцию и линковку.

REQUIRED означает, что если catkin найден не будет, прекратить выполнения инструкций CMake.

COMPONENTS означает, что необходимо подключить описанные далее компоненты. В данном случае необходимо подключить компонент roscpp, поскольку он требуется для всех программ, написанных на C++ и использующих ROS.

```
include_directories(include ${catkin_INCLUDE_DIRS})
```

Поскольку в тексте программы использовались функции, описанные в библиотеке ROS, необходимо определить пути до заголовочных файлов. Переменная catkin_INCLUDE_DIRS была установлена после успешного выполнения find_package.

```
add_executable(hello src/hello.cpp)
```

Здесь устанавливается имя исполняемого файла и файлы исходного кода, из которых будет собран этот исполняемый файл. Следует отметить, что имя “hello” это будет тот самый тип ноды, который будет использоваться при вызове команды roslaunch. Обратите внимание, что в тексте программы в функции ros::init говорилось о том, что имя ноды будет “hello_world”, а не “hello”, как указано здесь. На самом деле в этом нет противоречия. Имя hello используется как тип ноды (см. главу 3), в то время как имя ноды “hello_world” свойственно одной ноде и может быть изменено в момент запуска.

```
target_link_libraries(hello ${catkin_LIBRARIES})
```

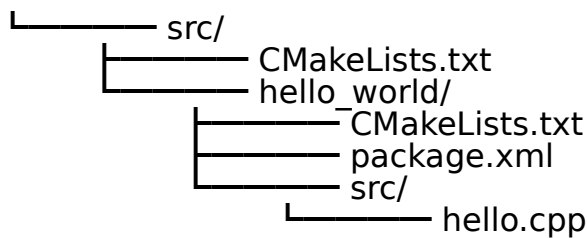
Позволяет линковщику собрать исполняемый файл hello вместе с требуемыми библиотеками. Переменная catkin_LIBRARIES также была установлена после выполнения функции find_package.

Теперь можно перейти в основную рабочую область и вызвать там catkin_make.

Если после выполнения catkin_make появились ошибки, следует проверить, что вызов происходил из нужной директории.

Приблизительно дерево директорий до вызова catkin_make должно выглядеть следующим образом:

```
workspace/
```



При этом вызов `catkin_make` должен осуществляться из директории `workspace/`.

После удачной сборки в папе `workspace` появятся две новые папки `devel/` и `build/`. Эти папки генерируются автоматически. Нас будет интересовать файл `devel/setup.bash`. Перед тем, как запускать программу, необходимо выполнить

```
$ source devel/setup.bash
```

После выполнения этой команды в переменные окружения ROS будет записана информация, где искать созданный новый пакет `hello_world`. Эту команду придется выполнять в каждом новом терминале до тех пор, пока этот пакет находится отдельно от остальных пакетов ROS.

Теперь запустим в отдельном терминале

```
$ roscore
```

После этого в первом терминале можно запускать написанную программу.

```
$ roslaunch hello_world hello
```

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Jason M. O'Kane, A Gentle Introduction to ROS, CreateSpace Independent Publishing Platform, Oct 2013., 162 P.
2. W. Burgard, C. Stachniss, G. Grisetti, B. Steder, R. Kummerle, C. Dorn-hege, M. Ruhnke, A. Kleiner, and J. D. Tardes. A comparison of slam algorithms based on a graph of relations. In 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems , pages 2089–2095, Oct 2009.
3. C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. J. Leonard. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. IEEE Transactions on Robotics, 32(6):1309–1332, Dec 2016.
4. M. Fallon, H. Johannsson, M. Kaess, and J. J. Leonard. The mit stata center dataset. The International Journal of Robotics Research, 32(14):1695–1699,

- 2013.
5. J. Funke and T. Pietzsch. A framework for evaluating visual slam. In Proceedings of the British Machine Vision Conference (BMVC), volume 6, 2009.
 6. B. Gerkey. Ros slam gmapping. http://wiki.ros.org/slam_gmapping, 2010. [Accessed 31-July-2017].
 7. Google. 2d cartographer backpack deutsches museum, 2016. [Accessed 31-July-2017].
 8. C. Harris and M. Stephens. A combined corner and edge detector. In Alvey vision conference, volume 15, pages 10–5244. Manchester, UK, 1988.
 9. W. Hess, D. Kohler, H. Rapp, and D. Andor. Real-time loop closure in 2d lidar slam. In Robotics and Automation (ICRA), 2016 IEEE International Conference on, pages 1271–1278. IEEE, 2016.