Оглавление

Введение

Программная платформа Robot Operating System (ПП ROS) является наиболее широко используемым в настоящее время программным продуктом с открытым исходным кодом, предназначенным для разработки программного обеспечения (ПО) для робототехнических систем (РТС). ROS предоставляет возможность использования последних достижений в области планирования движения, кинематики, обработки показаний датчиков, навигации и управления.

Архитектура разрабатываемого на базе ROS ПО представляет из себя модульную систему, состоящую из отдельных узлов. Являясь программной платформой, ROS организует исполнение своих прикладных программных модулей, предоставляя механизмы передачи информации между ними в виде тем, сервисов и действий.

Создание программного обеспечения на базе ROS заключается в реализации узлов на различных языках программирования и связано с использованием систем сборки программ из исходного кода, которые автоматизируют вызов компиляторов из скриптов сборки, определяют порядок их запуска, этапы компиляции, подключение зависимостей, а также действия до и после компиляции.

Программные модули ROS организуются в пакеты, куда также могут входить библиотеки, конфигурационные файлы, базы данных. Для создания пакетов в ПП ROS применяется система сборки catkin, включающая в себя макросы широко используемой кроссплатформенной системы генерации файлов управления сборкой Cmake, а также скрипты на языке Python. Система catkin позволяет сделать процесс сборки более удобным и управляемым, одновременно производить сборку нескольких пакетов ROS, решая проблемы зависимостей между ними. В пособии содержится руководство к выполнению лабораторной работы, целью которой является развитие навыков создания и построения единиц организации ПО на базе ROS – пакетов, изучение

механизмов передачи информации между программными модулями ПП ROS – узлами, создание их программной реализации.

Выполнение упражнений заключается в создании пакетов, решающих простые математические задачи. Выбор простых задач позволяет сконцентрировать внимание студентов на программной реализации связи между узлами, без углубления в тонкости работы РТС. В следующих лабораторных работах рассмотренные механизмы передачи информации будут использоваться для решения задач построения систем управления РТС.

Учебное пособие предназначено для студентов по направлению подготовки 15.03.06 «Мехатроника и робототехника», а также представляет интерес для научных сотрудников, инженеров, занимающихся созданием и применением робототехнических средств и интересующихся проблемами робототехники.

Список условных сокращений

ПО – программное обеспечение,

ПП – программная платформа,

РТС – робототехническая система,

IDL (interface definition language) – язык определения интерфейсов,

ROS (Robot Operating System) – робототехническая операционная система.

1. Создание пакета и его структура

Пакет – это набор файлов и директорий, содержащий исполняемые и вспомогательные файлы. Пакеты являются единицей организации ПО, выполняющей конкретную задачу.

Для создания пакета с именем my_pkg, который зависит от клиентских библиотек, содержащихся в пакетах pkg_1 и pkg_2, в командной строке необходимо перейти в директорию ~/catkin_ws/src:

$ cd ~/catkin_ws/src

и выполнить команду на создание пакета:

$ catkin_create_pkg my_pkg pkg_1 pkg_2

Здесь и далее наличие символа «$» в начале строки означает, что команду выполняют в окне терминала. При этом сам символ «$» в команду не входит.

Для сборки пакетов, находящихся в рабочем пространстве catkin, в командной строке необходимо перейти в директорию ~/catkin_ws:

$ cd

$ cd ~/catkin_ws

и выполнить команду:

$ catkin_make

Пакет с именем my_pkg может содержать в себе следующие директории и файлы:

1) *include/my_pkg/*

– директория, содержащая заголовочные файлы C++ с расширением .h;

2) *src/*

– директория, содержащая файлы C++ исходного кода с расширением .cpp;

3) *msg/*

– директория, содержащая файлы с расширением .msg, описывающие типы сообщений для тем;

4) *srv/*

– директория, содержащая файлы с расширением .srv, описывающие типы сервисных запросов;

5) *action/*

– директория, содержащая файлы с расширением . action, описывающие типы запросов действий;

6) *CMakeLists.txt*

– файл сборки пакета;

7) *package.xml*

– манифест, конфигурационный файл пакета.

Рассмотрим структуру файлов сообщений, сервисных запросов, запросов действий, файла сборки и манифеста пакета более подробно.

1.1. Файлы сообщений, сервисных запросов и запросов действий

Спецификой ROS является обмен информацией между программными модулями (узлами) на уровне сообщений. Поскольку обмениваться сообщениями могут узлы, реализованные на разных языках, разработка ПО на базе ROS происходит без привязки к конкретному языку программирования. Конечным результатом является не зависящая от языка обработки сообщений схема, где для реализации отдельных программных модулей по желанию разработчика могут быть использованы различные языки программирования.

Для описания сообщений в ROS применяется простой нейтральный язык определения интерфейсов IDL (interface definition language). Сообщение, сформирование на языке IDL, представляет собой короткий текстовый файл, в строках которого содержатся поля сообщения с указанием их типов и имен. Этот файл используются для генерации заголовочных файлов, содержащих исходный код, который описывает поля сообщения.

Типы полей, которые можно использовать при создании ***файлов сообщений*** на языке IDL:

– int8, int16, int32, int64 (целочисленные типы);

– float32, float64 (типы чисел с плавающей запятой);

– string (строка);

– time, duration (временные метки);

– другие файлы сообщений;

– массивы переменной и фиксированной длины перечисленных выше типов,

– Header (содержит временные метки и координационную информацию, которая обычно используется в ROS).

Файл сообщения темы – это текстовый файл на языке IDL, описывающий поля сообщения, которое используют тема. Файлы сообщений тем имеют расширение .msg и хранятся в директории msg/ каталога пакета.

Пример файла сообщения темы, типы полей которого представляют собой Header, строчный примитив и другое сообщение:

Header header

string child_frame_id

geometry_msgs/Point position

Входящее в состав рассмотренного выше сообщения поле с именем position описано в пакете geometry_msgs:

float64 x

float64 y

float64 z

Это сообщение, в свою очередь, включает в себя набор полей, представленных числами с плавающей запятой.

Файл сервисного запроса – это текстовый файл на языке IDL, который описывает сообщение, используемое сервисом. Файлы сервисных запросов имеют расширение .srv и хранятся в директории srv/ каталога пакета.

Файлы сервисных сообщений состоят из двух частей – запроса и ответа. Две части разделяются набором символов «---». Пример файла сервисного сообщения, типы полей запроса которого представляют собой целое число, временную метку и строку, а ответа – массив значений с плавающей запятой:

int32 seq

time stamp

string frame_id

---

float32[] data

В приведенном выше примере поля seq, stamp и frame_id – это запрос, а поле data – ответ.

Количество полей запроса и ответа может быть любым, в том числе и равным нулю.

Файл запроса действия – это текстовый файл на языке IDL, который описывает сообщение, которое использует сервис действия. Файлы запросов действия имеют расширение .action и хранятся в директории action/ каталога пакета.

Файлы запросов действий состоят из трех частей – задачи, результата и обратной связи. Три части разделяются набором символов «---». Пример файла запроса действий, типы полей задачи, результата и обратной связи которого представляют собой число с плавающей точкой, массив чисел с плавающей точкой и целое число:

float32 order

---

float32[] sequence

---

int32 state

В приведенном выше примере поле order – это задача, поле sequence – результат, а поле state – обратная связь.

Количество полей задачи, результата и обратной связи может быть любым, в том числе и равным нулю.

1.2. Структура конфигурационного файла

Каждый пакет содержит *манифест* – конфигурационный файл с именем package.xml. Каталог, содержащий package.xml, называется каталогом пакета. Скомпилированные программы хранятся в отдельной стандартной папке рабочего пространства. Чтобы найти каталог пакета с именем my_pkg, используйте команду rospack find:

$ rospack find my_pkg

Манифест содержит описание некоторых свойств пакета, включая его название, версию, сведения о разработчике, а также указание на зависимости от других пакетов.

Каждый файл package.xml содержит основной элемент с тегом <package>. Остальные элементы записываются внутри него. Минимальный набор тегов пакета с именем my_pkg включает имя пакета, его версию, описание, информацию о разработчике, лицензию:

<package>

<name> my_pkg </name>

<version> 0.0.0 </version>

<description> The my_pkg package </description>

<maintainer> user </maintainer>

<license> TODO </license>

</package>

В манифест пакета также следует добавить перечень зависимостей. Инструмент, который будет использоваться при сборке пакета, записывается под тегом <buildtool_depend>. В данной работе в качестве инструмента сборки используется catkin. Пакеты, необходимые во время сборки, записываются под тегом <build_depend>. Пакеты, используемые во время запуска исполняемых файлов, ставятся в тег <run_depend>. К примеру, зависимость от пакета roscpp библиотек языка C++ будет записана в виде:

<package>

...

< buildtool_depend> catkin </ buildtool_depend>

<build_depend> roscpp </build_depend>

<run_depend> roscpp </run_depend>

</package>

Можно описать несколько зависимостей, включив соответствующие строки в файл манифеста. Рассмотрим случай, когда в пакете, написанном на языке C++, создается, например, узел-издатель темы, тип которой входит в набор стандартной ROS-библиотеки std_msgs. Тогда в исполняемом файле, в котором описывается узел-издатель, необходимо подключить соответствующий заголовочный файл, описывающий тип сообщения. Расположение такого заголовочного файла задается добавлением следующих зависимостей:

<package>

...

< buildtool_depend> catkin </ buildtool_depend>

<build_depend> roscpp </build_depend>

<build_depend> std_msgs </build_depend>

<run_depend> roscpp </run_depend>

<run_depend> std_msgs </run_depend>

</package>

Рассмотрим случай, когда в пакете, написанном на языке C++, создается, например, узел-абонент темы, тип которой задается пользователем путем формирования файла с расширением .msg и полями, входящими в набор библиотеки стандартного ROS-пакета std_msgs. Тогда, в исполняемом файле, в котором описывается узел-издатель, необходимо подключить соответствующий заголовочный файл, описывающий тип сообщения. Для генерации такого заголовочного файла в манифест добавляются следующие зависимости:

<package>

...

< buildtool_depend> catkin </ buildtool_depend>

<build_depend> roscpp </build_depend>

<build_depend> std_msgs </build_depend>

<build_depend> message_generation </build_depend>

<run_depend> roscpp </run_depend>

<run_depend> std_msgs </run_depend>

<run_depend> message_runtime </run_depend>

</package>

В этом случае для построения используются библиотеки пакетов roscpp, std_msgs и библиотека пакета message_generation, обеспечивающая генерацию заголовочных файлов пользовательских сообщений. Для запуска пакета используются библиотеки пакетов roscpp, std_msgs и библиотека пакета message_runtime, обеспечивающая подключение заголовочных файлов и библиотек, необходимых исполняемым файлам пакета в процессе их выполнения.

При указании зависимостей абсолютные пути к существующим или генерируемым заголовочным файлам сообщений определяются сборщиком автоматически.

1.3. Структура файла сборки

Помимо файла манифеста пакет должен содержать *файл сборки* CMakeLists.txt – файл, содержащий перечень инструкций для сборки пакета, включая то, какие исполняемые файлы должны быть созданы, какие библиотеки для них использовать.

Файл сборки состоит из набора макросов – программных алгоритмов действий, записанных пользователем. Любой файл CMakeLists.txt должен соответствовать приведенной ниже структуре:

1) *cmake_minimum_required()*

– требуемая версия CMake. К примеру, записывается в виде:

cmake_minimum_required(VERSION 2.8.3)

2) *project()*

– имя пакета. К примеру, в CMakeLists.txt пакета с именем my_pkg будет записано:

project(my_pkg)

3) *find_package()*

– поиск сторонних пакетов, от которых зависит созданный пакет. Для программ на языке C++ данный шаг обеспечивает передачу компилятору соответствующих флагов для нахождения библиотек и заголовочных файлов, необходимых для сборки. Для задания зависимости от стороннего пакета с именем pkg, редактируется следующая стандартная строка файла CMakeLists.txt:

find_package(pkg REQUIRED)

У пакетов ROS всегда есть как минимум одна зависимость – от catkin:

find_package(catkin REQUIRED)

Зависимости от пакетов, созданные при помощи catkin, можно перечислить, добавив параметр COMPONENTS. К примеру, добавление пакета roscpp, содержащего библиотеку C++ для ROS, будет выглядеть следующим образом:

find_package(catkin REQUIRED COMPONENTS roscpp)

Пакеты, перечисленные в макросе find_package(catkin REQUIRED COMPONENTS), соответствуют указанным в тегах <build_depend> манифеста зависимостям от catkin пакетов.

В результате работы макроса find_package() системой сборки создаются различные переменные, которые предоставляют информацию о найденных пакетах, описывают, где находятся их заголовочные и исходные файлы, от каких библиотек зависят эти пакеты.

Эти переменные имеют вид ${pkg_PROPERTY}, где pkg – имя пакета, PROPERTY – свойство пакета pkg. Они могут быть использованы в других макросах. К примеру, в результате выполнения макроса

find_package(pkg REQUIRED)

определяются переменные ${pkg_INCLUDE_DIRS}, указывающая пути к заголовочным файлам пакета pkg, и ${pkg_LIBRARIES}, указывающая на библиотеки, экспортируемые пакетом pkg.

4) *add_message_files(), add_service_files(), add_action_files()*

– перечисление создаваемых типов сообщений, сервисных запросов и запросов действий. Имена файлов сообщений указываются после добавления параметра FILES.

К примеру, объявление типа сообщений Num, описанного в файле Num.msg, будет выглядеть следующим образом:

```
add_message_files(
    FILES
    Num.msg
)
```

Макросы add_service_files() и add_action_files() используются аналогично. Например, если узлы пакета используют сервисные сообщения типов Scale и Rotate, описание которых находится в файлах Scale.srv и Rotate.srv, в файле сборки пакета необходимо записать следующие строки:

```
add_service_files(
    FILES
    Scale.srv
    Rotate.srv
)
```

Если узлы пакета используют запросы действий типов Reconstruct и Reconfigure, описание которых находится в файлах Reconstruct.action и Reconfigure.action, в файле сборки пакета необходимо записать следующие строки:

```
add_action_files(
    FILES
    Reconstruct.action
    Reconfigure.action
```

)

По умолчанию при выполнении макросов add_message_files(), add_service_files() и add_action_files() поиск файлов будет осуществляться в каталогах msg, srv и action в директории собираемого пакета. Для указания другого пути, по которому следует искать файлы, перед их перечислением добавляют параметр DIRECTORY.

К примеру, объявление типа сообщений Num, описание которого содержится в файле Num.msg, находящемся в директории messages/ собираемого пакета, будет выглядеть следующим образом:

```
add_message_files(
    DIRECTORY messages
    FILES
    Num.msg
)
```

5) *generate_messages()*

– макрос, обеспечивающий генерацию новых типов сообщений, запросов и запросов действий из файлов сообщений, указанных в макросах add_message_files(), add_service_files() и add_action_files(). Поскольку генерируемые сообщения зависят от сообщений, содержащихся в библиотеках ROS, необходимо указать эту зависимость. Например, если создаваемое сообщение зависит от std_msgs, макрос примет вид:

```
generate_messages(DEPENDENCIES std_msgs)
```

Генерация заключается в создании заголовочного файла, описывающего сообщение. Имя создаваемого заголовочного файла задается автоматически. Если сообщение темы описано в файле с именем myMsg.msg, то имя заголовочного файла будет myMsg.h. Если сервисный запрос описан в файле с именем mySrv.srv, то имя заголовочного файла будет mySrv.h. Если запрос действия описан в файле с именем myAction.action, то имя заголовочного файла будет myActionAction.h.

Созданный заголовочный файл будет располагаться в директории пакета, в котором описано сообщение или запрос.

6) *catkin_package()*

– макрос, предоставляющий специфичную для catkin информацию системе сборки. Макрос имеет пять параметров, которые **не являются обязательными**:

INCLUDE_DIRS – указание директорий, содержащих подключаемые файлы пакета;

LIBRARIES – библиотеки, собираемые в данном проекте;

CATKIN_DEPENDS – указание других проектов catkin, от которых зависит собираемый пакет;

DEPENDS – другие (не catkin) проекты, от которых зависит собираемый пакет;

CFG_EXTRAS - дополнительные параметры, необходимые для сборки.

К примеру, файл сборки содержит следующую запись:

```
catkin_package(
    INCLUDE_DIRS include
    LIBRARIES math_lib
    CATKIN_DEPENDS nodelet
    DEPENDS eigen
)
```

Это означает, что заголовочные файлы содержатся в папке include/ в директории пакета. В пакете будет создана библиотека math_lib. Пакеты nodelet и eigen являются зависимостями собираемого пакета.

В том случае, если в пакете создаются заголовочные файлы пользовательских сообщений, следует записать:

```
catkin_package(CATKIN_DEPENDS message_runtime)
```

Пакеты, перечисленные в макросе catkin_package(CATKIN_DEPENDS) соответствуют зависимостям, перечисленным в тегах <run_depend> манифеста.

7) *include_directories()*

– указание директорий, содержащих подключаемые файлы. Например, указание на директории найденных в результате выполнения макроса find_package(catkin REQUIRED COMPONENTS) пакетов записывается в виде:

include_directories(${catkin_INCLUDE_DIRS})

8) *add_library()*

– объявление собираемых библиотек (указание целей сборки). Для создания библиотеки с именем my_lib, файл исходного кода которой my_lib_src.cpp находится в директории src/ пакета, в файле сборки нужно записать:

add_library(my_lib src/my_lib_src.cpp)

9) *add_executable()*

– объявление собираемых узлов (указание целей сборки). Для создания узла с именем my_node, файл исходного кода которой my_node_src.cpp находится в директории src/ пакета, в файле сборки нужно записать:

add_executable(my_node src/my_node_src.cpp)

Последние два макроса объявляют, что будет создан исполняемый файл (узел) либо библиотека с заданным именем с использованием исходного кода из указанного файла. Если файлов с исходным кодом несколько, их перечисляют, разделяя между собой пробелами.

10) *target_link_libraries()*

– подключение библиотек. Этот макрос сообщает системе автоматизации сборки о необходимости использовать соответствующие флаги, определенные в макросе find_package или add_library, при подключении библиотек. У узлов пакетов ROS всегда есть как минимум одна подключаемая библиотека – библиотека catkin. Для узла с именем my_node в файле сборки нужно записать:

target_link_libraries (my_node ${catkin_LIBRARIES})

Если подключаемых библиотек несколько, они записываются через пробел. Для подключения библиотеки с именем lib к узлу с именем my_node, в файле сборки нужно записать:

target_link_libraries (my_node ${catkin_LIBRARIES} lib)

11) *add_dependencies()*

– добавление зависимостей. Так, например, если в пакете создается узел с именем my_node, в файле исходного кода которого подключается заголовочный файл сообщения, генерируемый в процессе сборки этого же пакета, то необходимо явно указать зависимость цели сборки от создаваемого заголовочного файла. Это можно сделать следующим образом:

add_dependencies(my_node ${PROJECT_NAME}_generate_messages_cpp)

Иначе возможна ситуация, когда компиляция кода узла начнется до окончания генерации заголовочного файла, что приведет к ошибке сборки.

Порядок перечисления макросов в файле сборки имеет значение и является обязательным.

2. Механизмы передачи информации между узлами

ПП ROS основана на архитектуре графов. Обработка данных происходит в узлах (nodes), которые могут получать и передавать сообщения между собой. Узлы представляют собой процессы, в которых выполняются вычисления. Они могут обмениваться между собой информацией посредством передачи сообщений (messages).

Узлы отправляют и получают сообщения путем публикации или подписки на темы (topics), а так же с помощью предоставления или использования сервисов (services) или действий (actions).

Рассмотрим эти механизмы передачи информации между узлами, а также их программную реализацию более подробно.

2.1. Темы

При таком механизме передачи информации связь между узлами происходит путем опубликования и считывания *сообщения* из темы. Узел, публикующий сообщения, называется *издатель* (publisher), а узел,

принимающий сообщения, – *абонент* (subscriber). Тема определяется именем и типом передаваемого сообщения.

Соотношение между узлом-издателем, узлом-абонентом и передаваемыми сообщениями представлено на рисунке 1.

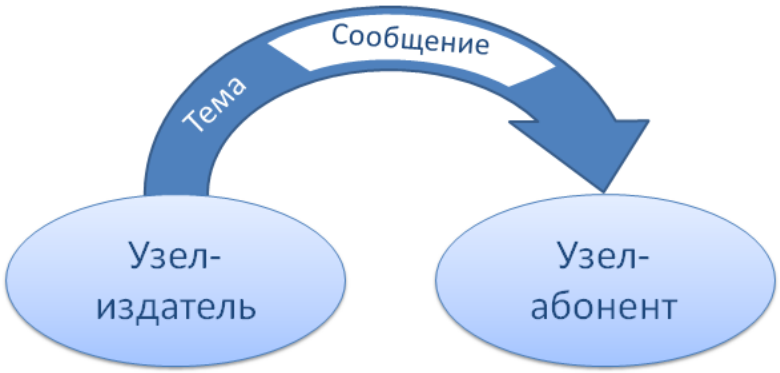

Рисунок 1 – Соотношение между узлом-издателем, узлом-абонентом и передаваемыми сообщениями

Одновременно могут существовать несколько издателей и абонентов по одной теме. Один узел может публиковать и/или подписываться на несколько тем. Как правило, издатели и абоненты не знают о существовании друг друга.

2.1.1. Программная реализация простого узла-издателя

Код простого узла-издателя имеет следующий вид:

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

int main(int argc, char **argv)
{
    ros::init(argc, argv, "my_pub_node");
    ros::NodeHandle n;
    ros::Publisher my_pub =
```

```
        n.advertise<msg_pkg::msg>("my_topic", QUEUE_SIZE);
    ros::Rate loop_rate(HZ);
    msg_pkg::msg my_msg;

    while (ros::ok())
    {
        my_msg. field_1 = ...;
        my_msg. field_2 = ...;
        ...
        my_pub.publish(my_msg);
        ros::spinOnce();
        loop_rate.sleep();
    }
    return 0;
}
```

Разберем код простого узла-издателя построчно:

#include "ros/ros.h"

– подключение заголовочного файла, который содержит в себе основные требуемые заголовочные файлы ROS.

#include "msg_pkg/msg.h "

– подключение заголовочного файла, содержащего описание типа сообщений, которые будут публиковаться в теме. В данном случае msg – имя заголовочного файла, msg_pkg – имя пакета, в котором лежит заголовочный файл.

*int main(int argc, char **argv)*

– главная функция исполняемого файла (узла).

ros::init(argc, argv, "my_pub_node");

– инициализация узла. В качестве аргументов функции init выступают argc и argv, через которые происходит преобразование или переопределение аргументов ROS, а также имя узла в кавычках (в данном случае my_pub_node).

Имя узла должно быть уникальным в работающей системе и базовым, то есть не содержащим символ косой черты "/".

ros::NodeHandle n;

– создание объекта класса ros::NodeHandle для получения доступа к свойствам узла внутри программы. Конструктор ros::NodeHandle производит старт узла, а деструктор ~ ros::NodeHandle завершит его работу.

ros::Publisher my_pub =

n.advertise<msg_pkg::msg>("my_topic", QUEUE_SIZE);

– создание объекта my_pub типа ros::Publisher посредством выполнения функции ros::NodeHandle::advertise(). При создании этого объекта ROS-мастеру поступает информация, что данный узел будет осуществлять публикацию сообщений типа msg_pkg/msg в тему с именем my_topic. Максимальное количество исходящих сообщений, которые будут поставлены в очередь для доставки подписчикам, задается с помощью параметра QUEUE_SIZE типа int.

ros::Rate loop_rate(HZ);

– создание объекта loop_rate типа.

msg_pkg::msg my_msg;

– создание объекта my_msg типа передаваемых в теме сообщений.

ros::ok()

– булева переменная, отражающая состояние узла. В момент старта ros::ok() тождественен true. ros::ok() вернёт false если: из терминала получена команда $ Ctrl+C; запущен другой узел с тем же именем; другим приложением был вызван метод ros::shutdown(). После того, как ros::ok() возвращает false, работа узла завершается.

my_msg. field_1 = ...;

my_msg. field_2 = ...;

...

– присвоение значений полям сообщения.

my_pub.publish(my_msg);

– публикация сообщения в тему.

ros::spinOnce();

– обновление информации обратных связей запущенных узлов. Так как в данном случае узел не имеет функций обратных связей, вызов ros::spinOnce() не является обязательным. Его добавление в данном коде является хорошим тоном программирования на базе ROS.

loop_rate.sleep();

– приостановка выполнения узла на время, равное разнице между периодом работы узла и интервалом времени с момента последнего вызова ros::Rate::sleep().

Таким образом, при создании простого узла-издателя нужно:

1) инициализировать и запустить узел;

2) сообщить ROS-мастеру, что созданный узел будет публиковать сообщения типа msg_pkg/msg в тему с именем my_topic;

3) осуществлять публикацию сообщений в тему с заданной частотой.

2.1.2. Программная реализация простого узла-абонента

Код простого узла-абонента имеет следующий вид:

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

void my_callback(const msg_pkg::msg::ConstPtr& msg)
{
    ...
}

int main(int argc, char **argv)
{
```

```
    ros::init(argc, argv, " my_sub_node ");
    ros::NodeHandle n;
    ros::Subscriber my_sub = n.subscribe("my_topic ", QUEUE_SIZE,
                                                    my_callback);
    ros::spin();
    return 0;
}
```

Разберем код простого узла-абонента построчно, пропуская некоторые части, которые уже были описаны выше:

void my_callback(const msg_pkg::msg::ConstPtr& msg)

– функция обратной связи, которая будет выполняться при получении нового сообщения из темы my_topic. В качестве параметра функция принимает указатель на полученное сообщение.

ros::Subscriber my_sub = n.subscribe("my_topic ", QUEUE_SIZE,
my_callback);

– создание объекта my_sub типа ros::Subscriber посредством выполнения функции ros::NodeHandle::subscribe(). При создании этого объекта ROS-мастеру поступает информация, что данный узел будет подписан на тему с именем my_topic. Количество входящих сообщений в очереди на обработку в функции обратной связи с именем my_callback задается с помощью параметра QUEUE_SIZE типа int (сообщения сверх этого количества будут отбрасываться);

ros::spin();

– функция, аналогичная выполнению цикла

```
while(ros::ok())
{
    ros::spinOnce()
}.
```

Таким образом, при создании простого узла-подписчика нужно:

1) инициализировать и запустить узел;

2) сообщить ROS-мастеру, что созданный узел будет подписан на тему с именем my_topic, и что при поступлении сообщения будет вызываться функция обратной связи my_callback;

3) ожидать прибытия сообщений в ros::spin().

2.2. Сервисы

Сервисы являются вторым механизмом передачи информации между узлами. Узел, предоставляющий сервис, называется *сервером* (server), а узел, использующий его, – *клиентом* (client). Сервисы позволяют клиентам послать *запрос* (request) сервису и получить от него *ответ* (response). Сообщение, формируемое из запроса и ответа, называют *сервисным запросом*. Сервер определяется именем и типом сервисного запроса (парой строго типизированных сообщений – один для запроса и второй для ответа).

Соотношение между узлом-сервером, узлом-клиентом и сервисным запросом представлено на рисунке 2.

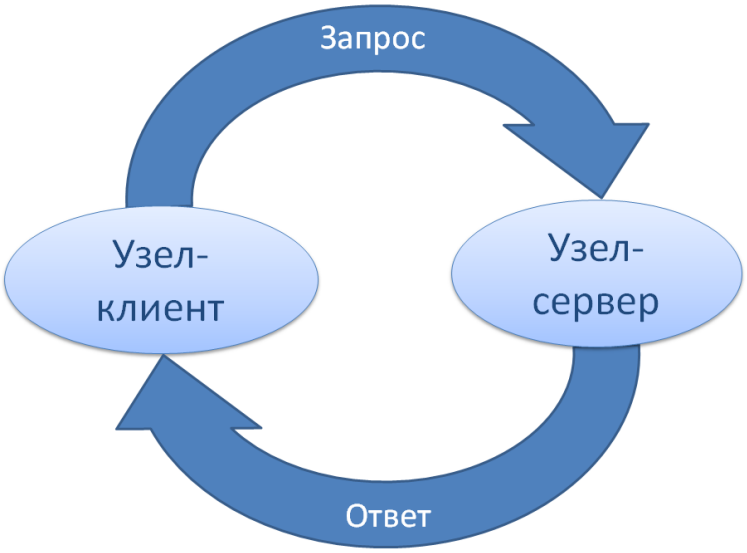

Рисунок 2 – Соотношение между узлом-сервером, узлом-клиентом и сервисным запросом

В отличие от темы, в которую несколько узлов могут осуществлять публикацию сообщений, сервис с конкретным именем может быть предоставлен только одним узлом. Второе отличие сервисов от тем заключается в том, что в тему сообщения публикуются непрерывно с заданной частотой, а в случае вызова сервиса запрос и ответ посылаются один раз.

2.2.1. Программная реализация простого узла-сервера

Код простого узла-сервера имеет следующий вид:

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

bool my_callback (msg_pkg::srv::Request &req, msg_pkg::srv::Response &res)
{
    res->field_1 = ...;
    res->field_2 = ...;
    ...
    return true;
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "my_server_node");
    ros::NodeHandle n;
    ros::ServiceServer my_server = n.advertiseService("my_service",
                                                      my_callback);
    ros::spin();
    return 0;
```

}

Разберем код простого узла-сервиса построчно, пропуская некоторые части, которые уже были описаны выше:

#include "msg_pkg/srv.h"

– подключение заголовочного файла, содержащего описание типа сервисных сообщений. В данном случае srv – имя заголовочного файла, msg_pkg – имя пакета, в котором лежит заголовочный файл.

bool my_callback (msg_pkg::srv::Request &req,
msg_pkg::srv::Response &res)

– функция обратной связи, которая будет выполняться при получении нового сервисного запроса. В качестве параметра функция принимает указатели на поля запроса (req) и ответа (res) сервисного сообщения. В функции обратной связи формируются значения полей ответа, которые будут доступны узлу-клиенту после вызова сервиса. Функция возвращает логический флаг типа bool.

ros::ServiceServer my_server = n.advertiseService("my_service",
my_callback);

– создание объекта my_server типа ros::ServiceServer посредством выполнения функции ros::NodeHandle::advertiseService(). При создании этого объекта ROS-мастеру поступает информация, что данный узел предоставляет сервис с именем my_service посредством выполнения функции my_callback.

Таким образом, при создании простого узла-сервера нужно:

1) Инициализировать и запустить узел;

2) Сообщить ROS-мастеру, что созданный узел будет предоставлять сервис с именем my_service посредством выполнения функции my_callback, в которой формируются поля ответа сервисного сообщения;

3) Ожидать поступления сервисного запроса в ros::spin().

2.2.2. Программная реализация простого узла-клиента

Код простого узла-клиента имеет следующий вид:

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

int main(int argc, char **argv)
{
	ros::init(argc, argv, "my_client_node");
	ros::NodeHandle n;
	ros::ServiceClient my_client =
				n.serviceClient<msg_pkg::srv>("my_service");
	msg_pkg::srv my_srv;
	my_srv.request.field_1  = ...;
	my_srv.request.field_2  = ...;
	...

	if (my_client.call(my_srv))
	{
		ROS_INFO("%...", my_srv.response.field_1);
		ROS_INFO("%...", my_srv.response.field_2);
		...
	}
	else
	{
		ROS_ERROR("Failed to call service");
		return 1;
	}
	 return 0;
```

}

Разберем код простого узла-клиента построчно, пропуская некоторые части, которые уже были описаны выше:

ros::ServiceClient my_client =

n.serviceClient<msg_pkg::srv>("my_service");

– создание объекта my_client типа ros::ServiceClient посредством выполнения функции ros::NodeHandle::serviceClient(). При создании этого объекта ROS-мастеру поступает информация, что данный узел является клиентом сервиса с именем my_service и типом сервисного запроса msg_pkg/srv.

msg_pkg::srv my_srv;

– создание объекта my_srv типа сервисного запроса.

my_srv.request.field_1 = ...;

my_srv.request.field_2 = ...;

– присвоение значений полям запроса сервисного сообщения.

my_client.call(my_srv)

– вызов сервиса с передачей сервисного запроса. Так как после вызова сервис блокируется, то возврат из функции происходит только после завершения вызова. Если вызов сервиса был успешным, то call() вернёт true и результирующее значение в полях ответа my_srv.response. В противном случае call() вернёт false и значения полей ответа будут пустыми.

ROS_INFO(...);

– функция форматного вывода в терминал, предоставляемая ROS. Замена стандартным функциям printf и cout.

Таким образом, при создании простого узла-клиента нужно:

1) инициализировать и запустить узел;

2) сообщить ROS-мастеру, что созданный узел будет клиентом сервиса с именем my_service и типом сервисного запроса msg_pkg/srv;

3) сформировать поля запроса сервисного сообщения;

4) вызвать сервис с передачей ему сервисного запроса.

2.3. Действия

Третьим механизмом передачи информации между узлами являются действия. Узел, предоставляющий действия, называется *сервером действия* (action server), а узел, использующий его, – *клиентом действия* (action client). Действия, по аналогии с сервисами, позволяют клиенту посылать *задачу* (goal) серверу действий и получить от него *результат* (result). Отличием действий от сервисов является наличие *обратной связи* (feedback), посредствам которой клиент действий получает информацию от сервера действий в процессе обработки в нем данных для формирования ответа. Сообщение, формируемое из задачи, результата и обратной связи называют *запросом действия*. Сервер действий определяется именем и типом запроса действий (тройкой строго типизированных сообщений – для задачи, результата и обратной связи).

Соотношение между узлом-сервером действия, узлом-клиентом действия и запросом действия представлено на рисунке 3.

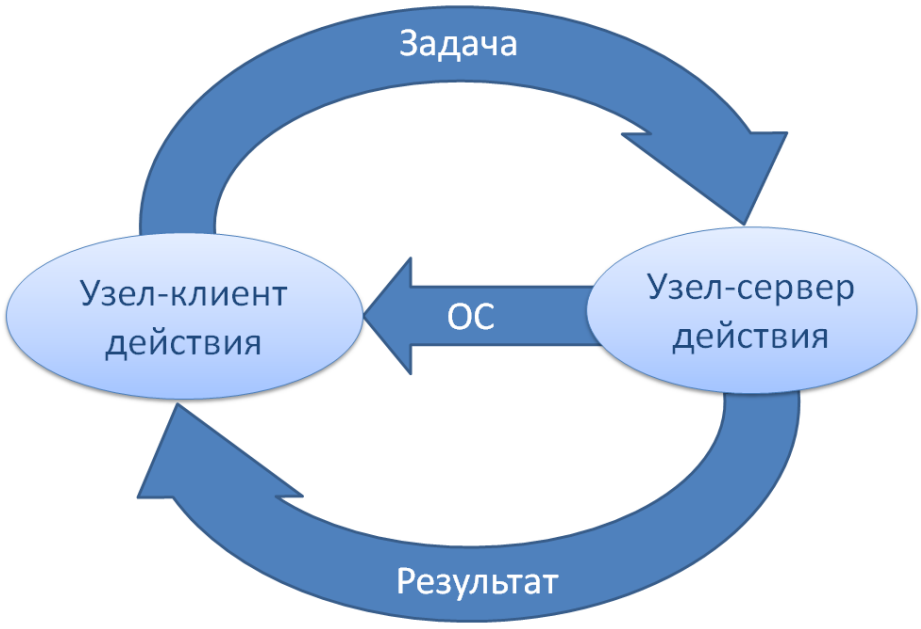

Рисунок 3 – Соотношение между узлом-сервером действия, узлом-клиентом действия и запросом действия (на рисунке ОС – обратная связь)

Так же, как и в случае сервисов, в отличие от темы, только один узел может предоставлять действие с конкретным именем, а в случае вызова действия задача и результат посылаются один раз. Отличие действий от сервисов заключается в наличии обратной связи, которая публикуется в промежуток времени между вызовом действия и формированием результата, а также в возможности прерывания выполнения действия до его завершения.

2.3.1. Программная реализация простого узла-сервера действия

Код простого узла-сервера действия имеет следующий вид:

```
#include "ros/ros.h"
#include "actionlib/server/simple_action_server.h"
#include "msg_pkg/actionAction.h"

void my_callback(const msg_pkg::actionGoalConstPtr& goal,
                 actionlib::SimpleActionServer<msg_pkg::actionAction>* as)
{
    ros::Rate loop_rate(HZ);
    bool success = true;
    msg_pkg::actionFeedback feedback;
    msg_pkg::actionResult result;

    for(...)
    {
        if (as->isPreemptRequested() || !ros::ok())
        {
            ROS_INFO("Preempted");
```

```
                as->setPreempted();
                success = false;
                break;
            }
        feedback.field_1 = ...;
        feedback.field_2 = ...;
        ...
        as->publishFeedback(feedback);
        loop_rate.sleep();
    }

    if(success)
    {
        result.field_1 = ...;
        result.field_2 = ...;
        ...
        ROS_INFO("Succeeded");
        as->setSucceeded(result);
    }
}

int main(int argc, char** argv)
{
    ros::init(argc, argv, "my_action_server_node");
    ros::NodeHandle n;
    actionlib::SimpleActionServer<msg_pkg::actionAction> as
        (n, "my_action_service", boost::bind(&my_callback, _1, &as), false);
    as.start();
    ros::spin();
    return 0;
```

}

Разберем код простого узла-сервера действия построчно, пропуская некоторые части, которые уже были описаны выше:

#include "actionlib/server/simple_action_server.h"

– подключение заголовочного файла, содержащего описание библиотеки для создания и работы простого сервера действия;

#include "msg_pkg/actionAction.h"

– подключение заголовочного файла, содержащего описание типа запроса действия. В данном случае actionAction – имя заголовочного файла, msg_pkg – имя пакета, в котором лежит заголовочный файл;

void my_callback(const msg_pkg::actionGoalConstPtr& goal,
actionlib::SimpleActionServer<msg_pkg::actionAction> as)*

– функция обратной связи, которая будет выполняться при получении нового запроса действия. В качестве параметра функция принимает указатели на поля цели (goal) и сервера действий as с указанием типа запроса действия msg_pkg/actionAction. В функции обратной связи формируются значения полей обратной связи, которые будут публиковаться в процессе выполнения запроса действия в теме с именем /my_action_server/feedback, где my_action_server – имя сервера действия. Также в функции обратной связи формируются значения полей результата после завершения выполнения действия, которые будут опубликованы в теме с именем /my_action_server/ result, где my_action_server – имя сервера действия.

bool success = true;

– булева переменная, отражающая успешность выполнения запроса действия. Если в процессе выполнения запроса поступит другой запрос действия, обработка первого запроса будет прервана и начнется обработка второго. Выполнение первого запроса неудачным (прерванным), и переменная success будет false;

msg_pkg::actionFeedback feedback;

– создание объекта с типом обратной связи запроса действия;

msg_pkg::actionResult result;

– создание объекта с типом результата запроса действия;

for(...)

– цикл выполнения запроса действия;

if (as->isPreemptRequested() || !ros::ok())

– проверка на прерывание (поступление другого запроса действия) и состояние узла;

as->setPreempted();

– назначение состояния «прервано» сервису действия;

feedback. field_1 = ...;

feedback. field_2 = ...;

– присвоение значений полям обратной связи запроса действия;

as->publishFeedback(feedback);

– опубликование обратной связи в теме с именем /my_action_server/feedback, где my_action_server – имя сервера действия;

if(success)

– проверка на отсутствие прерывания в процессе выполнения запроса действия;

result.field_1 = ...;

result.field_2 = ...;

– присвоение значений полям результата запроса действия;

as->setSucceeded(result);

– назначение состояния «выполнено успешно» сервису действия и передача ему полей результата запроса действия. Внутри этой функции происходит опубликование результата в теме с именем /my_action_server/result, где my_action_server – имя сервера действия;

actionlib::SimpleActionServer<msg_pkg::actionAction> as

(n, "my_action_ service", boost::bind(&my_callback, _1, &as), false);

– конструктор объекта сервера действия as типа actionlib::SimpleActionServer.

При создании этого объекта ROS-мастеру поступает информация, что данный узел предоставляет действие с именем my_action_service и типом запроса действия msg_pkg/actionAction по средствам выполнения функции my_callback в которую передается созданный сервер действия. Последний параметр типа bool, передаваемый конструктору, задет характер запуска сервера действия – если true, то сервер будет запущен сразу после его создания, если false, то запуск сервера нужно проводить отдельно;

as.start();

– запуска сервера действия.

Таким образом, при создании простого узла-сервера действия нужно:

1) Инициализировать и запустить узел;

2) Сообщить ROS-мастеру, что созданный узел будет предоставлять действие с именем my_action_service по средствам выполнения функции my_callback, в которой формируются поля результата и обратной связи запроса действия;

3) Ожидать поступления запроса действия в ros::spin().

2.3.2. Программная реализация простого узла-клиента действия

Код простого узла-клиента действия имеет следующий вид:

```
#include "ros/ros.h"
#include "actionlib/client/simple_action_client.h"
#include "actionlib/client/terminal_state.h"
#include "msg_pkg/actionAction.h"

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

```
    actionlib::SimpleActionClient<msg_pkg::actionAction> ac
                                                ("my_action_service", true);
    ac.waitForServer();
    msg_pkg::actionGoal goal;
    goal.field_1 = ...;
    goal.field_2 = ...;
    ...
    ac.sendGoal(goal);

    if (ac.waitForResult(ros::Duration(TIME)))
    {
        actionlib::SimpleClientGoalState state = ac.getState();
        ROS_INFO("Action finished: %s",state.toString().c_str()");
    }
    else
        ROS_INFO("Action did not finish before the time out");
    return 0;
}
```

Разберем код простого узла-сервера действия построчно, пропуская некоторые части, которые уже были описаны выше:

#include "actionlib/client/simple_action_client.h"

– подключение заголовочного файла, содержащего описание библиотеки для создания и работы простого клиента действия.

#include "actionlib/client/terminal_state.h"

– подключение заголовочного файла, содержащего описание библиотеки для определения возможных состояний запроса действия.

actionlib::SimpleActionClient<msg_pkg::actionAction> ac
("my_action_service", true);

– конструктор объекта клиента действия as типа actionlib:: SimpleActionClient.

При создании этого объекта ROS-мастеру поступает информация, что данный узел является клиентом действия с именем my_action_service и типом запроса действия msg_pkg/actionAction. Последний параметр типа bool, передаваемый конструктору, задет характер запуска клиента действия – true, если клиент должен быть запущен сразу после его создания, и false, если запуск клиента нужно проводить отдельно.

ac.waitForServer();

– ожидание начала работы сервера действия, предоставляющего требуемое действие.

msg_pkg::actionGoal goal;

– создание объекта с типом задачи запроса действия.

goal.field_1 = ...;

goal.field_2 = ...;

– присвоение значений полям задачи запроса действия.

ac.sendGoal(goal);

– вызов действия с передачей полей задачи запроса действия.

ac.waitForResult(ros::Duration(TIME))

– ожидание получения результата действия на протяжении времени TIME типа double. Если действие выполнено до назначенного времени, функция возвращает true, иначе – false.

actionlib::SimpleClientGoalState state = ac.getState();

– получение состояния действия. Возможные состояния: PENDING (ожидание на выполнение), ACTIVE (выполняется), RECALLED (повторный вызов), REJECTED (отброшено), PREEMPTED (прервано из-за поступления нового запроса), ABORTED (прервано из-за прекращения работы сервера), SUCCEEDED (успешно выполнено), LOST (потеряно).

Таким образом, при создании простого узла-клиента действия нужно:

1) инициализировать и запустить узел;

2) сообщить ROS-мастеру, что созданный узел будет клиентом сервиса действия с именем my_action_service и типом запроса действия msg_pkg/action;

3) сформировать поля задачи запроса действия;

4) вызвать сервис действия с передачей ему запроса действия.

3. Последовательность действий при выполнении работы

Лабораторная работа знакомит с механизмами передачи информации между узлами – темами, сервисами и действиями, а также с их программной реализацией на примере выполнения упражнений, заключающихся в создании пакетов, решающих простые математические задачи.

Цель выполнения описанных ниже упражнений:

– освоить основы создания и сборки пакета ROS , ознакомиться с его структурой;

– понять принципы написания узла-издателя, узла-абонента, файлов сообщений и реализовать их;

– изучить принцип взаимодействия узла-издателя и узла-абонента;

– понять принципы написания узла-сервера, узла-клиента, файлов сервисных запросов и реализовать их;

– изучить принцип взаимодействия узла-сервера и узла-клиента;

– понять принципы написания узла-сервера действия, узла-клиента действия, файлов запросов действий и реализовать их;

– изучить принцип взаимодействия узла-сервера действия и узла-клиента действия.

Упражнение №1

Требуется создать пакет, содержащий в себе два узла – узел-издатель и узел-абонент. Узел-издатель осуществляет публикацию сообщений,

содержащих два поля – строку «sin(α) = », где α – величина угла, для которой рассчитывается значение синуса, и величину угла. Угол должен изменять свое значение от -3 до 3 радиан и обратно с шагом 0,1. Узел-абонент принимает сообщение и выводит значения его полей в виде «sin(α) = β», где β – значение синуса угла α.

1. Включите ПК, загрузите ОС Ubuntu.

2. Откройте окно терминала, кликнув левой клавишей мыши на значок на панели управления (слева) или используя комбинацию клавиш Ctrl+Alt+T.

3. Создайте пакет с именем publisher_subscriber, который зависит от клиентских библиотек roscpp, std_msgs и message_generation.

4. Соберите пакеты, расположенные в рабочей области catkin_ws. Убедитесь, что сборка произошла без ошибок.

5. Откройте домашнюю папку пользователя. Для этого кликните левой клавишей мыши на значок на панели управления (слева). Перейдите в директорию источников рабочей области /catkin_ws/src. В ней должен располагаться пакет publisher_subscriber, который был создан в п. 3.

6. Зайдите в директорию пакета publisher_subscriber. Здесь создайте папку с именем msg. Для создания папки кликните правой кнопкой мыши по пустому полю директории /catkin_ws/src/publisher_subscriber/ и выберите «Создать новую папку». Внутри директории msg создайте файл SinValue.msg. Сформируйте файл сообщения согласно заданию, сохраните его и закройте. При возникновении затруднений обратитесь к Приложению 1.

7. Вернитесь в директорию пакета publisher_subscriber. Зайдите в директорию, содержащую файлы C++ исходного кода, и создайте два пустых документа – publisher.cpp и subscriber.cpp. Для создания пустого документа кликните правой кнопкой мыши по пустому полю директории /catkin_ws/src/publisher_subscriber/src/ и выберите «Создать новый документ» >> «Пустой документ». Напишите требуемый код в созданные файлы, опираясь на теоретическую часть пособия. В качестве имен узлов укажите talker для

издателя и listener для подписчика. Создаваемую тему назовите chatter. Сохраните и закройте файлы. При возникновении затруднений обратитесь к Приложению 1.

8. Вернитесь в директорию пакета publisher_subscriber. Откройте файл сборки CMakeLists.txt и измените его, опираясь на теоретическую часть пособия. В качестве имен собираемых узлов укажите publisher для издателя и subscriber для подписчика. Сохраните и закройте файл. При возникновении затруднений обратитесь к Приложению 1.

9. Вернитесь в директорию пакета publisher_subscriber. Откройте конфигурационный файл package.xml и измените его, опираясь на теоретическую часть пособия, сохраните его и закройте. При возникновении затруднений обратитесь к Приложению 1.

10. Соберите пакеты, расположенные в рабочей области catkin_ws. Убедитесь, что сборка произошла без ошибок.

11. В окне терминала запустите ROS-мастер. Для этого выполните команду выполните команду *$ roscore.* Сверните окно терминала. До окончания работы оно не понадобится.

12. Откройте два новых окна терминала – второе и третье (в первом запущен roscore). Для этого дважды проделаете действия – кликните левой клавишей мыши на значок на панели управления (слева) или используйте комбинацию клавиш Ctrl+Alt+T.

13. Во втором окне терминала запустите узел publisher из пакета publisher_subscriber. Для этого выполните команду *$ rosrun publisher_subscriber publisher*

Процесс работы узла представлен на рисунке 4.

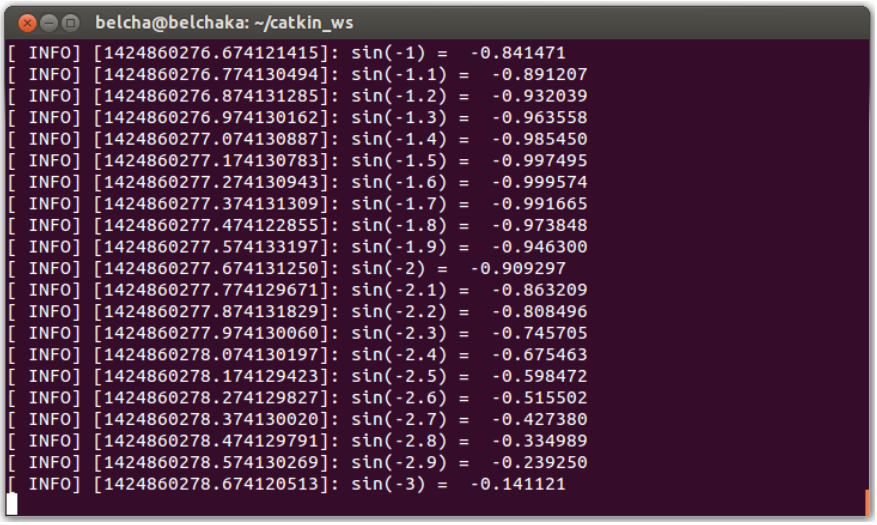

Рисунок 4 – Процесс работы узла publisher из пакета publisher_subscriber

14. В третьем окне терминала запустите узел subscriber из пакета publisher_subscriber. Для этого выполните команду *$ rosrun publisher_subscriber subscriber*

Процесс работы узла представлен на рисунке 5.

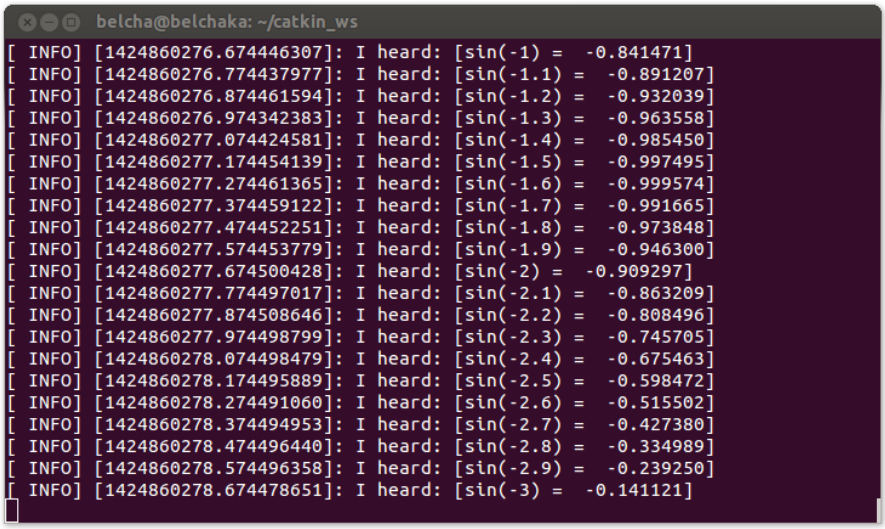

Рисунок 5 – Процесс работы узла subscriber из пакета publisher_subscriber

15. Откройте четвертое окно терминала, кликнув левой клавишей мыши на значок на панели управления (слева) или используя комбинацию клавиш Ctrl+Alt+T.

16. В четвертом окне терминала запустите утилиту rqt_graph. Для этого выполните команду *$ rqt_graph*. Результат работы утилиты представлен на рисунке 6. Объясните взаимодействие узлов.

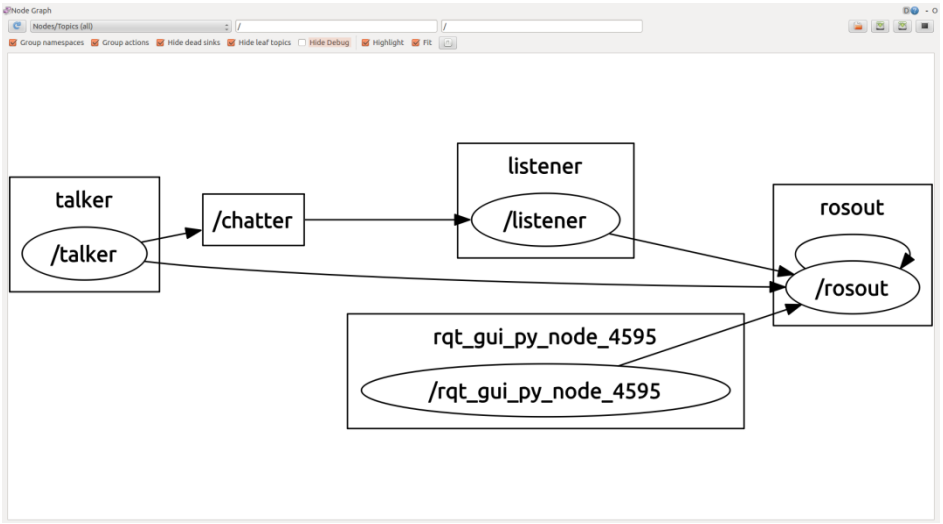

Рисунок 6 – Результат выполнения команды rqt_graph для работающих узлов publisher и subscriber пакета publisher_subscriber

Замечание: Обратите внимание, при запуске узла с помощью rosrun после имени пакета в качестве имени узла (publisher и subscriber) указывается имя собираемого узла, которое прописывается в файле сборки CMakeLists.txt. Но rqt_graph берет имена узлов и тем (talker, listener, chatter), указанные при их инициализации. Отметим, что хорошим тоном программирования для ROS считается указание одинаковых имен для собираемого узла в CMakeLists.txt и инициализируемого им узла в функции ros::init().

17. Остановите выполнение узла-издателя во втором окне терминала, выполнив в нем команду *$ Ctrl+C*. Зафиксируйте реакцию узла-подписчика, запущенного в третьем окне терминала.

18. Обновите rqt_graph. Для этого нажмите на голубую круговую стрелку в верхнем левом углу. Результат работы утилиты представлен на рисунке 7. Объясните взаимодействие узлов.

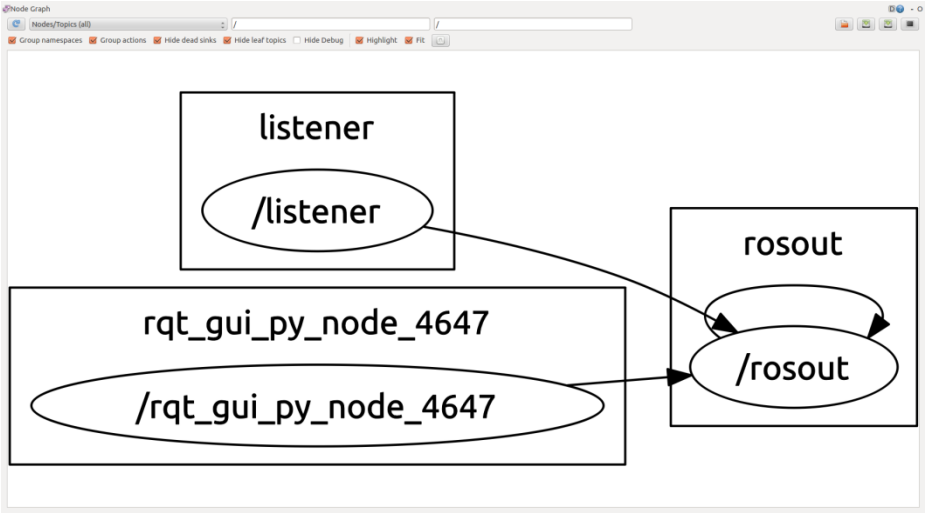

Рисунок 7 – Результат выполнения команды rqt_graph для неработающего узла publisher и работающего subscriber пакета publisher_subscriber

19. Возобновите выполнение узла publisher во втором окне терминала, выполнив в нем команду *$ rosrun publisher_subscriber publisher*. Зафиксируйте реакцию узла-подписчика, запущенного в третьем окне терминала.

20. Завершите выполнение всех узлов и утилит, выполнив в каждом окне терминала команду *$ Ctrl+C*, и закройте окна.

Упражнение №2

Требуется создать пакет, содержащий в себе два узла – узел-сервер и узел-клиент. Узел-сервер предоставляет сервис, принимающий в качестве запроса два числа с плавающей точкой, возвращающий в качестве ответа их сумму и выводит его в окно терминала. Узел-клиент формирует запрос сервисного сообщения из чисел, передаваемых в главную функцию узла в виде аргументов. После вызова сервиса, узел-клиент выводит ответ в окно терминала.

1. Включите ПК, загрузите ОС Ubuntu.

2. Откройте окно терминала, кликнув левой клавишей мыши на значок на панели управления (слева) или используя комбинацию клавиш Ctrl+Alt+T.

3. Создайте пакет с именем server_client, который зависит от клиентских библиотек roscpp, std_msgs и message_generation.

4. Соберите пакеты, расположенные в рабочей области catkin_ws. Убедитесь, что сборка произошла без ошибок.

5. Откройте домашнюю папку пользователя. Для этого кликните левой клавишей мыши на значок на панели управления (слева). Перейдите в директорию источников рабочей области /catkin_ws/src. В ней должен располагаться пакет server_client, который был создан в п. 3.

6. Зайдите в директорию пакета server_client. Здесь создайте папку с именем srv. Для создания папки кликните правой кнопкой мыши по пустому полю директории /catkin_ws/src/server_client/ и выберите «Создать новую папку». Внутри директории srv создайте файл Sum.srv. Сформируйте файл сервисного запроса согласно заданию, сохраните его и закройте. При возникновении затруднений обратитесь к Приложению 2.

7. Вернитесь в директорию пакета server_client. Зайдите в директорию, содержащую файлы C++ исходного кода, и создайте два пустых документа – server.cpp и client.cpp. Для создания пустого документа кликните правой кнопкой мыши по пустому полю директории /catkin_ws/src/server_client/src/ и выберите «Создать новый документ» >> «Пустой документ». Напишите требуемый код в созданные файлы, опираясь на теоретическую часть пособия. В качестве имен узлов укажите server для сервера и client для клиента. Создаваемый сервис назовите sum_two_floats. Сохраните и закройте файлы. При возникновении затруднений обратитесь к Приложению 2.

8. Вернитесь в директорию пакета server_client. Откройте файл сборки CMakeLists.txt и измените его, опираясь на теоретическую часть пособия. В качестве имен собираемых узлов укажите server для сервера и client для клиента. Сохраните и закройте файл. При возникновении затруднений обратитесь к Приложению 2.

9. Вернитесь в директорию пакета server_client. Откройте конфигурационный файл package.xml и измените его, опираясь на теоретическую часть пособия, сохраните его и закройте. При возникновении затруднений обратитесь к Приложению 2.

10. Соберите пакеты, расположенные в рабочей области catkin_ws. Убедитесь, что сборка произошла без ошибок.

11. В окне терминала запустите ROS-мастер. Для этого выполните команду выполните команду *$ roscore*. Сверните окно терминала. До окончания работы оно не понадобится.

12. Откройте два новых окна терминала – второе и третье (в первом запущен roscore). Для этого дважды выполните следующие действия – кликните левой клавишей мыши на значок на панели управления (слева) или используйте комбинацию клавиш Ctrl+Alt+T.

13. Во втором окне терминала запустите узел server из пакета server_client. Для этого выполните команду *$ rosrun server_client server*

Узел-сервер запущен и ожидает запроса (рисунок 8).

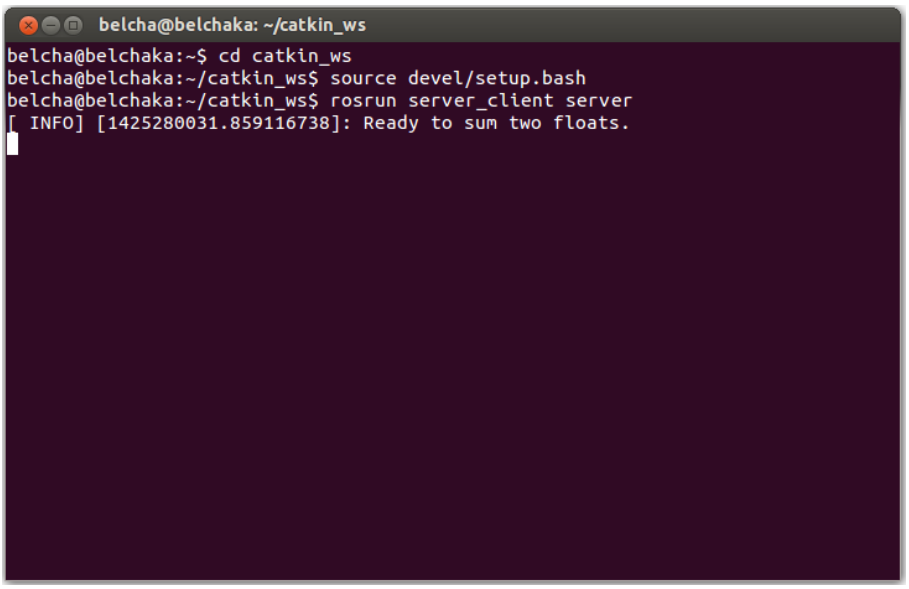

Рисунок 8 – Процесс работы узла server из пакета server_client (ожидание сервисного запроса)

14. В третьем окне терминала запустите узел client из пакета server_client и задайте значение запроса – два числа (например, 5 и 9), сумму которых должен вернуть сервис. Для этого выполните команду *$ rosrun server_client client 5 9* (рисунок 9).

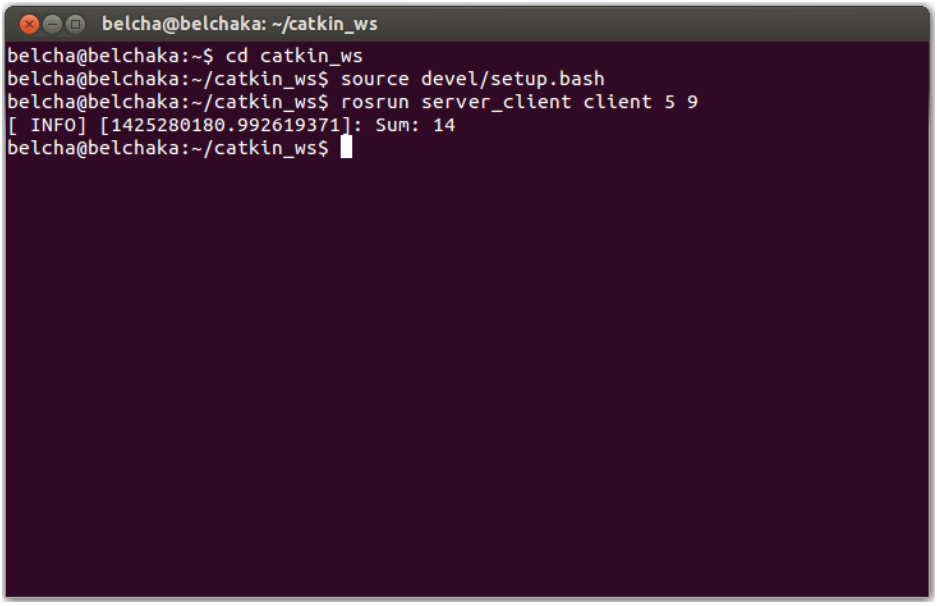

Рисунок 9 – Процесс работы узла client из пакета server_client

Реакция узла-сервера на запрос узла-клиента представлена на рисунке 10.

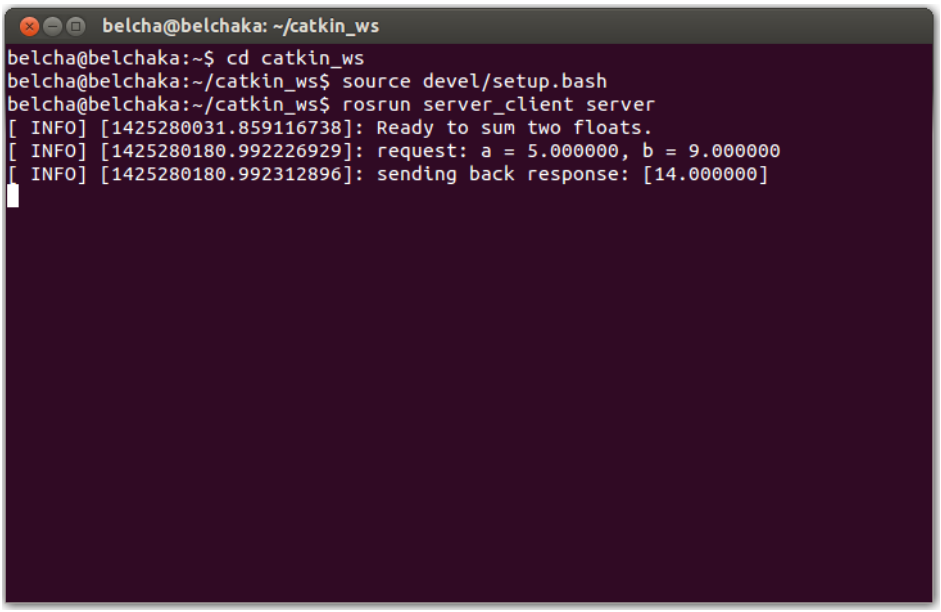

Рисунок 10 – Реакция узла-сервера на запрос узла-клиента

15. Завершите выполнение всех узлов, выполнив в каждом окне терминала команду *$ Ctrl+C*, и закройте окна.

Упражнение 3

Требуется создать пакет, содержащий в себе два узла – узел-сервер действия и узел-клиент действия. Узел-сервер действия предоставляет действие, формирующее последовательность Фибоначчи. В качестве задачи действие принимает количество элементов последовательности и в качестве результата возвращает сформированную последовательность. Результат формируется в цикле, на каждом шаге которого публикуется обратная связь, представляющая собой промежуточную последовательность. Узел-клиент действия формирует запрос действия, задача которого является числом, передаваемым в главную функцию узла в виде аргумента.

1. Включите ПК, загрузите ОС Ubuntu.

2. Откройте окно терминала, кликнув левой клавишей мыши на значок на панели управления (слева) или используя комбинацию клавиш Ctrl+Alt+T.

3. Создайте пакет с именем action_server_client, который зависит от клиентских библиотек roscpp, std_msgs, actionlib, actionlib_msgs и message_generation.

4. Соберите пакеты, расположенные в рабочей области catkin_ws. Убедитесь, что сборка произошла без ошибок.

5. Откройте домашнюю папку пользователя. Для этого кликните левой клавишей мыши на значок на панели управления (слева). Перейдите в директорию источников рабочей области /catkin_ws/src. В ней должен располагаться пакет action_server_client, который был создан в п. 3.

6. Зайдите в директорию пакета action_server_client. Здесь создайте папку с именем action. Для создания папки кликните правой кнопкой мыши по пустому полю директории /catkin_ws/src/action_server_client/ и выберите

«Создать новую папку». Внутри директории action создайте файл Fibonacci.action. Сформируйте файл сервисного запроса согласно заданию, сохраните его и закройте. При возникновении затруднений обратитесь к Приложению 3.

7. Вернитесь в директорию пакета action_server_client. Зайдите в директорию, содержащую файлы C++ исходного кода, и создайте два пустых документа – action_server.cpp и action_client.cpp. Для создания пустого документа кликните правой кнопкой мыши по пустому полю директории /catkin_ws/src/action_server_client/src/ и выберите «Создать новый документ» >> «Пустой документ». Напишите требуемый код в созданные файлы, опираясь на теоретическую часть пособия. В качестве имен узлов укажите action_server для сервера действия и action_client для клиента действия. Создаваемое действие назовите fibonacci. Сохраните и закройте файлы. При возникновении затруднений обратитесь к Приложению 3.

8. Вернитесь в директорию пакета action_server_client. Откройте файл сборки CMakeLists.txt и измените его, опираясь на теоретическую часть пособия. В качестве имен собираемых узлов укажите action_server для сервера действия и action_client для клиента действия. Сохраните и закройте файл. При возникновении затруднений обратитесь к Приложению 3.

9. Вернитесь в директорию пакета action_server_client. Откройте конфигурационный файл package.xml и измените его, опираясь на теоретическую часть пособия, сохраните его и закройте. При возникновении затруднений обратитесь к Приложению 3.

10. Соберите пакеты, расположенные в рабочей области catkin_ws. Убедитесь, что сборка произошла без ошибок.

11. В окне терминала запустите ROS-мастер. Для этого выполните команду выполните команду *$ roscore.* Сверните окно терминала. До окончания работы оно не понадобится.

12. Откройте четыре новых окна терминала – второе, третье, четвертое и пятое (в первом запущен roscore). Для этого четыре раза выполните следующие

действия – кликните левой клавишей мыши на значок на панели управления (слева) или используйте комбинацию клавиш Ctrl+Alt+T.

13. Во втором окне терминала запустите узел action_server из пакета action_server_client. Для этого выполните команду *$ rosrun action_server_client action_server*

Узел-сервера действия запущен и ожидает запроса действия (рисунок 11).

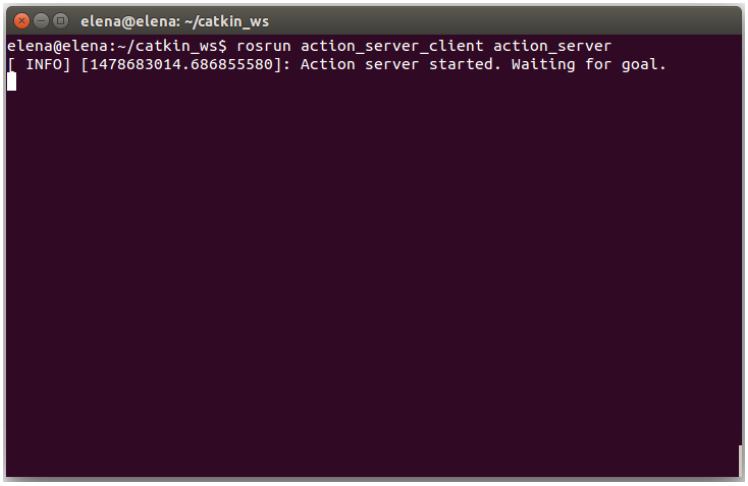

Рисунок 11 – Процесс работы узла action_server из пакета action_server_client (ожидание запроса действия)

14. В третьем окне терминала с помощью команды rostopic list выведите имена активных тем. Для этого выполните команду *$ rostopic list.*

Темы с именами /rosout и /rosout_agg создаются при запуске roscore. Темы с именами /fibonacci/cancel, /fibonacci/feedback, /fibonacci/goal, /fibonacci/result и /fibonacci/status создаются автоматически при запуске узла-сервера запроса.

В темы /fibonacci/goal и /fibonacci/cancel осуществляется публикация клиентом действия сообщений, содержащих запрос действия и запрос на прерывание соответственно.

В темы /fibonacci/status, /fibonacci/feedback и /fibonacci/result осуществляется публикация сервером действия сообщений, содержащих состояние сервиса, обратную связь и результат выполнения запроса действия соответственно.

Имена этих тем формируются следующим образом – /action_service_name/cancel, /action_service_name/feedback, /action_service_name/goal, /action_service_name/result и /action_service_name/status, где action_service_name – имя сервиса действия.

15. В третьем окне терминала с помощью команды rostopic echo выведите сообщения, опубликованные в теме с именем /fibonacci/feedback. Для этого выполните команду *$ rostopic echo /fibonacci/feedback*

Зафиксируйте значения сообщений темы и объясните их.

16. В четвертом окне терминала с помощью команды rostopic echo выведите сообщения, опубликованные в теме с именем /fibonacci/result. Для этого выполните команду *$ rostopic echo /fibonacci/result*

Зафиксируйте значения сообщений темы и объясните их.

17. В пятом окне терминала запустите узел action_client из пакета action_server_client и задайте значение цели – число элементов последовательности Фибоначчи (например, 20). Для этого выполните команду *$ rosrun action_server_client action_client 20* (рисунок 12).

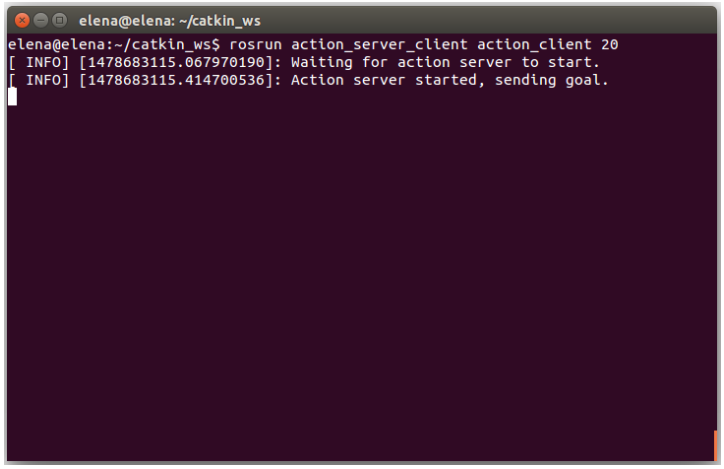

Рисунок 12 – Процесс работы узла action_client из пакета action_server_client

18. Зафиксируйте значения сообщений тем /fibonacci/feedback и /fibonacci/result в третьем и четвертом окнах терминала в процессе и после выполнения запроса действия. Объясните полученный результат.

19. Завершите выполнение всех узлов и инструментов, выполнив в каждом окне терминала команду *$ Ctrl+C*, и закройте окна.

Контрольные вопросы

1. Что такое пакет в ПП ROS? Опишите его структуру.

2. Опишите структуру сообщения темы на языке IDL. Приведите пример такого сообщения.

3. Опишите структуру сервисного запроса на языке IDL. Приведите пример такого сообщения.

4. Опишите структуру запроса действия на языке IDL. Приведите пример такого сообщения.

5. Опишите структуру конфигурационного файла.

6. Опишите структуру файла сборки.

7. В чем заключается механизм передачи информации между узлами с помощью темы? Чем тема отличается от сервиса и действия?

8. В чем заключается механизм передачи информации между узлами с помощью сервиса? Чем тема отличается от темы и действия?

9. В чем заключается механизм передачи информации между узлами с помощью действия? Чем тема отличается от темы и сервиса?

10. Перечислите основные действия, которые нужно совершить при создании узла-издателя.

11. Перечислите основные действия, которые нужно совершить при создании узла-подписчика.

12. Перечислите основные действия, которые нужно совершить при создании узла-сервера.

13. Перечислите основные действия, которые нужно совершить при создании узла-клиента.

14. Перечислите основные действия, которые нужно совершить при создании узла-сервера действия.

15. Перечислите основные действия, которые нужно совершить при создании узла-клиента действия.

Заключение

В пособии содержится руководство к выполнению лабораторной работы, при выполнении которой создаются навыки создания и построения пакетов, изучаются механизмы передачи информации между узлами с помощью темам, сервисов и действий. Осуществляется программная реализация перечисленных механизмов.

При выполнении лабораторной работы создаются пакеты, решающие простые математические задачи. При их создании происходит развитие навыков написания узла-издателя, узла-абонента, файлов сообщений для передачи информации с помощью темы, узла-сервера, узла-клиента, файлов сервисных запросов сообщений для передачи информации с помощью сервисов и узла-сервера действия, узла-клиента действия, файлов запросов действий для передачи информации с помощью действий.

На практике изучаются принципы взаимодействия между узлом-издателем и узлом-абонентом, узлом-сервером и узлом-клиентом, узлом-сервером действия и узлом-клиентом действия.

Приложение 1

SinValue.msg

```
string str
float64 value
```

publisher.cpp

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

int main(int argc, char **argv)
{
	ros::init(argc, argv, "talker");
	ros::NodeHandle n;
	ros::Publisher chatter_pub = n.advertise<publisher_subscriber::SinValue>
                                                        ("chatter", 1000);
	ros::Rate loop_rate(10);
	float angle = 0;
	bool increase = true;
	while (ros::ok())
	{
		std::stringstream ss;
		ss << "sin(" << angle << ") = ";
		publisher_subscriber::SinValue msg;
		msg.str = ss.str();
		msg.value = sin(angle);
		ROS_INFO("%s %f", msg.str.c_str(), msg.value);
		chatter_pub.publish(msg);
```

```
            ros::spinOnce();
            loop_rate.sleep();
            if(angle > 2.9)      increase = false;
            if(angle < -2.9)     increase = true;
            if(increase)         angle = angle + 0.1;
            if(!increase)        angle = angle - 0.1;
        }
        return 0;
}
```

subscriber.cpp

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

void chatterCallback(const publisher_subscriber::SinValue::ConstPtr& msg)
{
        ROS_INFO("I heard: [%s %f]", msg->str.c_str(), msg->value);
}

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;
}
```

CmakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
project(publisher_subscriber)
find_package(catkin REQUIRED COMPONENTS
        roscpp
        std_msgs
        message_generation
)
add_message_files(
        FILES
        SinValue.msg
)
generate_messages(
        DEPENDENCIES
        std_msgs
)
catkin_package(
        CATKIN_DEPENDS
        roscpp
        std_msgs
        message_runtime
)
include_directories(
        ${catkin_INCLUDE_DIRS}
)
add_executable(publisher src/publisher.cpp)
target_link_libraries(publisher ${catkin_LIBRARIES})
add_dependencies(publisher ${PROJECT_NAME}_generate_messages_cpp)
add_executable(subscriber src/subscriber.cpp)
target_link_libraries(subscriber ${catkin_LIBRARIES})
add_dependencies(subscriber ${PROJECT_NAME}_generate_messages_cpp)
```

package.xml

```
<?xml version="1.0"?>
<package>
	<name>publisher_subscriber</name>
	<version>0.0.0</version>
	<description>The publisher_subscriber package</description>
	<maintainer email="user@todo.todo">user</maintainer>
	<license>TODO</license>

	<buildtool_depend>catkin</buildtool_depend>

	<build_depend>roscpp</build_depend>
	<build_depend>std_msgs</build_depend>
	<build_depend>message_generation</build_depend>

	<run_depend>roscpp</run_depend>
	<run_depend>std_msgs</run_depend>
	<run_depend>message_runtime</run_depend>
</package>
```

Приложение 2

Sum.srv

```
float64 a
float64 b
---
float64 sum
```

server.cpp

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

bool sum(server_client::Sum::Request &req, server_client::Sum::Response &res)
{
	res.sum = req.a + req.b;
	ROS_INFO("request: a = %f, b = %f", req.a, req.b);
	ROS_INFO("sending back response: [%f]", res.sum);
	return true;
}

int main(int argc, char **argv)
{
	ros::init(argc, argv, "server");
	ros::NodeHandle n;
	ros::ServiceServer service = n.advertiseService("sum_two_floats", sum);
	ROS_INFO("Ready to sum two floats.");
	ros::spin();
	return 0;
```

```
}
```

client.cpp

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

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

CmakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
```

```
project(server_client)
find_package(catkin REQUIRED COMPONENTS
        roscpp
        std_msgs
        message_generation
)
add_service_files(
        FILES
        Sum.srv
)
generate_messages(
        DEPENDENCIES
        std_msgs
)
catkin_package(
        CATKIN_DEPENDS
        roscpp
        std_msgs
        message_runtime
)
include_directories(
        ${catkin_INCLUDE_DIRS}
)
add_executable(server src/server.cpp)
target_link_libraries(server ${catkin_LIBRARIES})
add_dependencies(server ${PROJECT_NAME}_generate_messages_cpp)
add_executable(client src/client.cpp)
target_link_libraries(client ${catkin_LIBRARIES})
add_dependencies(client ${PROJECT_NAME}_generate_messages_cpp)
```

package.xml

```
<?xml version="1.0"?>
<package>
	<name>server_client</name>
	<version>0.0.0</version>
	<description>The server_client package</description>
	<maintainer email="user@todo.todo">user</maintainer>
	<license>TODO</license>

	<buildtool_depend>catkin</buildtool_depend>

	<build_depend>roscpp</build_depend>
	<build_depend>std_msgs</build_depend>
	<build_depend>message_generation</build_depend>

	<run_depend>roscpp</run_depend>
	<run_depend>std_msgs</run_depend>
	<run_depend>message_runtime</run_depend>
</package>
```

Приложение 3

Fibonacci.action

```
int32 order
---
int32[] sequence
---
int32[] sequence
```

action_server.cpp

```
#include "ros/ros.h"
#include "actionlib/server/simple_action_server.h"
#include "action_server_client/FibonacciAction.h"

void callback(const action_server_client::FibonacciGoalConstPtr &goal,
actionlib::SimpleActionServer<action_server_client::FibonacciAction>* as)
{
	ros::Rate r(1);
	bool success = true;
	action_server_client::FibonacciFeedback feedback;
	action_server_client::FibonacciResult result;
	feedback.sequence.clear();
	feedback.sequence.push_back(0);
	feedback.sequence.push_back(1);
	ROS_INFO("Executing, creating fibonacci sequence of order %i with seeds
	%i, %i", goal->order, feedback.sequence[0], feedback.sequence[1]);
	for(int i=1; i<=goal->order; i++)
	{
		if (as->isPreemptRequested() || !ros::ok())
```

```
            {
                ROS_INFO("Preempted");
                as->setPreempted();
                success = false;
                break;
            }
            feedback.sequence.push_back(feedback.sequence[i] +
            feedback.sequence[i-1]);
            as->publishFeedback(feedback);
            r.sleep();
        }
        if(success)
        {
            result.sequence = feedback.sequence;
            ROS_INFO("Succeeded");
            as->setSucceeded(result);
        }
}

int main(int argc, char** argv)
{
        ros::init(argc, argv, "action_server");
        ros::NodeHandle n;
        actionlib::SimpleActionServer<action_server_client::FibonacciAction> as
                        (n, "fibonacci", boost::bind(&callback, _1, &as), false);
        as.start();
        ROS_INFO("Action server started. Waiting for goal.");
        ros::spin();
        return 0;
}
```

action_client.cpp

```
#include "ros/ros.h"
#include "actionlib/client/simple_action_client.h"
#include "actionlib/client/terminal_state.h"
#include "action_server_client/FibonacciAction.h"

int main (int argc, char **argv)
{
        ros::init(argc, argv, "action_client");
        actionlib::SimpleActionClient<action_server_client:: FibonacciAction> ac
                                                                ("fibonacci", true);
        ROS_INFO("Waiting for action server to start.");
        ac.waitForServer();
        ROS_INFO("Action server started, sending goal.");
        action_server_client::FibonacciGoal goal;
        goal.order = atoll(argv[1]);
        ac.sendGoal(goal);
        bool finished_before_timeout = ac.waitForResult(ros::Duration(30.0));
        if (finished_before_timeout)
        {
                actionlib::SimpleClientGoalState state = ac.getState();
                ROS_INFO("Action finished: %s",state.toString().c_str());
        }
        else
                ROS_INFO("Action did not finish before the time out.");
        return 0;
}
```

CmakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
project(action_server_client)
find_package(catkin REQUIRED COMPONENTS
        roscpp
        std_msgs
        actionlib
        actionlib_msgs
        message_generation
)
add_action_files(
        FILES
        Fibonacci.action
)
generate_messages(
        DEPENDENCIES
        actionlib_msgs
)
catkin_package(
        CATKIN_DEPENDS
        roscpp
        std_msgs
        actionlib
        actionlib_msgs
        message_runtime
)
include_directories(
        ${catkin_INCLUDE_DIRS}
)
add_executable(action_server src/action_server.cpp)
```

```
target_link_libraries(action_server ${catkin_LIBRARIES})
add_dependencies(action_server ${PROJECT_NAME}_generate_messages_cpp)
add_executable(action_client src/action_client.cpp)
target_link_libraries(action_client ${catkin_LIBRARIES})
add_dependencies(action_client ${PROJECT_NAME}_generate_messages_cpp)
```

package.xml

```
<?xml version="1.0"?>
<package>
  <name>action_server_client</name>
  <version>0.0.0</version>
  <description>The action_server_client package</description>
  <maintainer email="user@todo.todo">user</maintainer>
  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>roscpp</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>actionlib</build_depend>
  <build_depend>actionlib_msgs</build_depend>
  <build_depend>message_generation</build_depend>

  <run_depend>roscpp</run_depend>
  <run_depend>std_msgs</run_depend>
  <run_depend>actionlib</run_depend>
  <run_depend>actionlib_msgs</run_depend>
  <run_depend>message_runtime</run_depend>
</package>
```

Список рекомендованной литературы

1. Lentin J. Mastering ROS for Robotics Programming / Packt Publishing, 2015.
2. Fernández E,. Crespo L.S., Mahtani A., Martinez A. Learning ROS for Robotics Program-ming - second edition / Packt Publishing, 2015.
3. Goebel P. ROS By Example / Lulu, 2013.
4. Goebel P. ROS By Example. Volume 2 / Lulu, 2014.
5. Jason M. O'Kane. A Gentle Introduction to ROS / CreateSpace Independent Publishing Plat-form, 2013.
6. Martinez A., Fernández E. Learning ROS for Robotics Programming / Packt Publishing, 2013.
7. Robot Operating System http://wiki.ros.org