ROS

ITMO more than a UNIVERSITY

Операционная система ROS

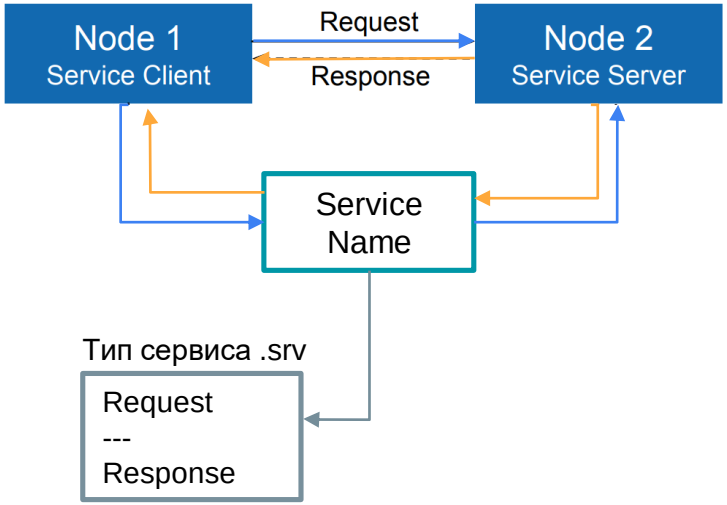
Разработка программных решений для роботов

План занятия

- Опрос на15 мин
- Создание ROS Service
  - Service Server
  - Service Client
- ROS Action (actionlib)
- ROS time
- ROS bags
- 10 мин на вопросы по лабораторным

ROS Service

ROS реализует общение в формате Запрос/ответ между узлами при помощи **сервисов.**
По структуре схожи с сообщениями, типы сервисов задаются в файлах *.srv



rosservice list - Вывод информации об активных сервисах
rosservice type /service_name - вывод типа сервиса
rosservice info /service_name – вывод информации о сервисе
rosservice call /service_name args – вызвать сервис с запросом
rosservice find /service_type – найти сервисы по его типу

Ссылка на wiki: https://wiki.ros.org/Services

Примеры сервисов

std_srvs/Trigger.srv

```
---
bool success
string message
```

Запрос: пустой
Ответ: 2 поля

nav_msgs/GetPlan.srv

```
geometry_msgs/PoseStamped start
geometry_msgs/PoseStamped goal
float32 tolerance
---
nav_msgs/Path plan
```

Запрос: 3 поля
Ответ: 1 поле

В примере справа гораздо больше полей, т.к. используются вложенные типы…

rossrv show /service_type – вывод определения типа (содержимое .srv)

Service Server

В rospy создание сервиса выполняется путем создания экземпляра класса **Service** со своей функцией callback, которая срабатывает при получении новых запросов.
Важно: Каждый callback запускается в отдельном потоке.
Требуется spin() чтобы заблокировать поток запуска до завершения работы.

```
def add_two_ints(req):
    return rospy_tutorials.srv.AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', rospy_tutorials.srv.AddTwoInts, add_two_ints)
    rospy.spin()
```

Заголовки соединений headers являются особенность топиков и сервисов в ROS, которая позволяет отправлять дополнительные метаданные когда выполняется первое соединение между узлами.

Ссылка на вики: https://wiki.ros.org/rospy/Overview/Services
http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28python%29

Service Client

К серверу можно обращаться из ноды или из консоли через
rosservice call /server_service “”

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

from __future__ import print_function

import sys
import rospy
from beginner_tutorials.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp1 = add_two_ints(x, y)
        return resp1.sum
    except rospy.ServiceException as e:
        print("Service call failed: %s"%e)

def usage():
    return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:
        print(usage())
        sys.exit(1)
    print("Requesting %s+%s"%(x, y))
    print("%s + %s = %s"%(x, y, add_two_ints_client(x, y)))
```

Ссылка на вики: http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28python%29

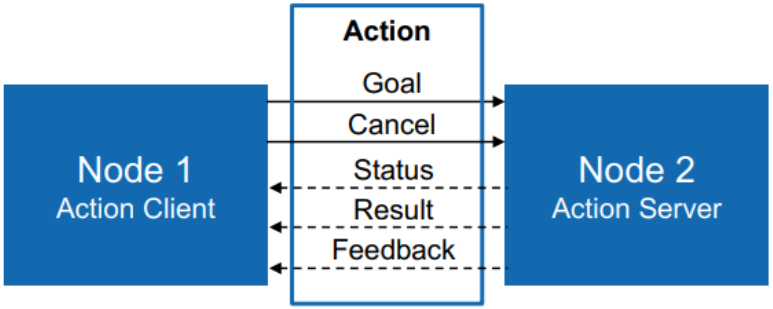
ROS Actions (actionlib)

Action очень похож на сервисы, только в отличии от сервиса он позволяет получать промежуточную обратную связь (текущий прогресс, или даже отменить), тем самым позволяет гибко взаимодействовать с процессами.
Action - лучший способ реализации взаимодействия с процессами, которые занимают много времени или поведением ориентированным на цель.

- По структуре очень похожи на сервисы и заданы в файлах с расширением .action
- Внутри реализованы набором топиков

В файлах .action 3 раздела, разделенные «---»

```
Request
---
Result
---
Feedback
```



Ссылка на вики: http://wiki.ros.org/actionlib_tutorials

ROS Actions Server

Примеры

Timer.action

```
duration time_to_wait
---
duration time_elapsed
uint32 updates_sent
---
duration time_elapsed
duration time_remaining
```

```python
import roslib
roslib.load_manifest('my_action_demo_controller')
import rospy
import actionlib
import time
from my_action_demo_controller.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback

def do_action(goal): # action function
    start_time = time.time()
    update_count = 0
    if goal.time_to_wait.to_sec() > 60.0: # check req duration
        result = TimerResult()
        result.time_elapsed = rospy.Duration.from_sec( time.time() - start_time)
        result.updates_sent = update_count
        server.set_aborted(result, "Aborted: too long to wait")
        return # too long of a requested wait
    while (time.time()-start_time) < goal.time_to_wait.to_sec(): # waiting to meet goal duration
        if server.is_preempt_requested(): # check preemption
            result = TimerResult()
            result.time_elapsed = rospy.Duration.from_sec( time.time() - start_time)
            result.updates_sent = update_count
            server.set_preempted(result, "Timer preempted (the goal updated)")
            return
        feedback = TimerFeedback()
        feedback.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
        feedback.time_remaining  =  goal.time_to_wait - feedback.time_elapsed
        server.publish_feedback(feedback)
        update_count += 1
        time.sleep(1.0)
    result = TimerResult()
    result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
    result.updates_sent = update_count
    server.set_succeeded(result, "Timer completed successfully")

rospy.init_node('timer_action_server') # initialize node
rospy.loginfo('action server started!!')
server = actionlib.SimpleActionServer('timer_server', TimerAction, do_action, False)
server.start()
rospy.spin()
```

Ссылка на вики: http://wiki.ros.org/actionlib_tutorials

ROS Actions Client

Примеры

```
import rospy
import time # for regular Python timing
import actionlib    # for actions!
from my_action_demo_controller.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback
def the_feedback_cb(feedback): # feedback callback function
    print('[Feedback] Time elapsed: %f' % (feedback.time_elapsed.to_sec()))
    print('[Feedback] Time remaining: %f' % (feedback.time_remaining.to_sec()))

rospy.init_node('timer_action_client') # initialize node

client = actionlib.SimpleActionClient( # register client
    'timer_action_server',  # action server name
    TimerAction # action Action message
)
client.wait_for_server()    # wait for action server
goal = TimerGoal()  # create goal object
goal.time_to_wait = rospy.Duration.from_sec(5.0) # set field
# Uncomment this line to test server-side abort:
# goal.time_to_wait = rospy.Duration.from_sec(500.0)

client.send_goal(goal, feedback_cb = the_feedback_cb) # send goal
# # Uncomment these lines to test goal
# time.sleep(3.0)
# client.cancel_goal()
# client.get_state()


client.wait_for_result() # wait for action server to finish
# print results: print('[Result) State: %d' % (client.get_state()))
# print(.[Result] Status: % (client.get_goal_status_text()))

if client.get_result():
    print('[Result] Time elapsed: %f' % (client.get_result().time_elapsed.to_sec()))
    print('[Result) Updates sent: %d' % (client.get_result().updates_sent))
```

ROS time

Обычно ROS использует системное время.

ROS имеет встроенные примитивные типы **времени (конкретный момент времени)** и **продолжительности (период времени)**, которые rospy предоставляет в виде классов.

```
int32 secs
int32 nsecs
```

Следующие два способа являются альтернативными:

- rospy.Time.now() # Возвращает класс с полями **secs** и **nsecs**
- rospy.get_rostime()

Но от них отличается:

seconds = rospy.get_time() #возвращает время float в секундах

d = rospy.Duration.from_sec(60.1) # возвращает класс Duration с полями **secs** и **nsecs**

Вместо использования **системного времени** (модуля time.time, wall time) для доступа к текущему времени следует использовать **симуляционное время**, если работы с симулятором или bag, для этого время надо запускать **Clock Server**

Ссылка на вики: http://wiki.ros.org/rospy/Overview/Time

Clock Server

это любой узел, который публикует информацию в топике /clock, и в одной сети ROS никогда не должно быть более одного.

В большинстве случаев Clock Server представляет собой либо **симулятор**, либо инструмент воспроизведения журналов.

Нужно чтобы для параметра /**use_sim_time** было установлено значение true во всех файлах запуска, использующих Clock Server.

Если вы воспроизводите файл **Bag** с помощью rosbag play, использование опции --lock запустит сервер синхронизации во время воспроизведения файла Bag.

В симуляторе Gazebo по умолчанию уже включен Clock Server.

Ссылка на вики: http://wiki.ros.org/Bags

Sleeping and Rates

Как и с другими примитивными типами, вы можете выполнять арифметические операции со временем и продолжительностью.

1 hour + 1 hour = 2 hours (*duration* + *duration* = *duration*)
2 hours - 1 hour = 1 hour (*duration* - *duration* = *duration*)
Today + 1 day = tomorrow (*time* + *duration* = *time*)
Today - tomorrow = -1 day (*time* - *time* = *duration*)
Today + tomorrow = *error* (*time* + *time is undefined*)

Продолжительность может быть rospy.Duration или количество секунд (float).
ROS будет спать заданное время. rospy.sleep(20.0)
ROS предоставляет удобный класс rospy.Rate, который делает все возможное для поддержания определенной скорости в цикле.

Ссылка на вики:

Timer

Ros предоставляет удобный класс rospy.Timer, который вызывает callback функцию с заданной периодичностью.

Аргументы конструктора:

- Period – Время между последовательными вызовами (например: rospy.Duration(0.1))
- Callback – вызываемая функция
- Oneshot – одноразовый или нет?

```
def my_callback(event):
    print 'Timer called at ' + str(event.current_real)

rospy.Timer(rospy.Duration(2), my_callback)
```

Позволяет запустить несколько паблишеров/сабскрайберов с разной частотой.

```
rospy.Timer(rospy.Duration(1.0/10.0), ts.publish_temperature)

rospy.Timer(rospy.Duration(1.0/10.0), ts.read_temperature_sensor_data)
```

ROS bags

Bag - это формат файла в ROS для хранения данных сообщений ROS.
Данные типа .bag играют важную роль в ROS, и было написано множество инструментов, позволяющих записывать, хранить, обрабатывать, анализировать и визуализировать их.

rosbag record –all
rosbag record /topic_name
Остановить запись можно через **Ctrl + C**
Записи сохраняются в текущую папку, а файл называется датой и временем запуска записи.
rosbag info file_name.bag - Показать информацию
rosbag play file_name.bag – Считывать и публиковать содержимое
Также можно специфичные параметры менять
rosbag play --rate=0.5 file_name.bag – задать частоту и т.д.

rqt_bag – 3D визуализация

Ссылка на вики: http://wiki.ros.org/Bags ,http://wiki.ros.org/rosbag/Commandline

Параметры

rosparam позволяет хранить данные и манипулировать ими на сервере параметров ROS
rosparam использует язык разметки YAML и может содержать данные типа:

- 1 — целое число,
- 1.0 — число с плавающей запятой,
- one — строка,
- true — логическое значение,
- [1, 2, 3] — список целых чисел
- {a: b, c: d} — словарь.

В rosparam есть множество команд, которые можно использовать с параметрами, как показано ниже:

```
rosparam set            set parameter
rosparam get            get parameter
rosparam load           load parameters from file
rosparam dump           dump parameters to file
rosparam delete         delete parameter
rosparam list           list parameter names
```

Ссылка на вики: http://wiki.ros.org/rosparam

Параметры в launch

Тег <param> задает значения параметров, которые будут отправлены в Сервер параметров.
Доступны следующие атрибуты:

- name="namespace/name" – название параметра
- value="value"*(optional) – значение параметра*
- type="str|int|double|bool|yaml"*(optional) – тип параметра*
- textfile="$(find pkg-name)/path/file.txt"*(optional) – считывание текстового файла*
- binfile="$(find pkg-name)/path/file"*(optional) – считывание бинарного файла*

Пример: <param name="publish_frequency" type="double" value="10.0" />

Параметры также могут храниться в файле и подгружаться при запуске:
<rosparam command="load" file="FILENAME" />
$(find pkg-name) - замена аргументов также поддерживается в launch файлах, поэтому тут будет возвращен путь относительно пакета. ($(dirname) - абсолютный путь к папке с launch файлом).

Ссылка на вики: http://wiki.ros.org/rosparam

Пример

```
  <node pkg="robot_state_publisher" type="robot_state_publisher" name="robot_state_publisher" output="screen" >
    <param name="publish_frequency" type="double" value="40.0" />
  </node>


  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find profi2022_test_scene)/worlds/simple_scene.world"/>
    <arg name="paused" value="false"/>
    <arg name="use_sim_time" value="true"/>
    <arg name="gui" value="true"/>
    <arg name="headless" value="false"/>
    <arg name="debug" value="false"/>
  </include>


  <node name="diff_drive_robot_spawn" pkg="gazebo_ros" type="spawn_model" output="screen"
   args="-urdf -param robot_description -model diff_drive_robot" />

</launch>
```

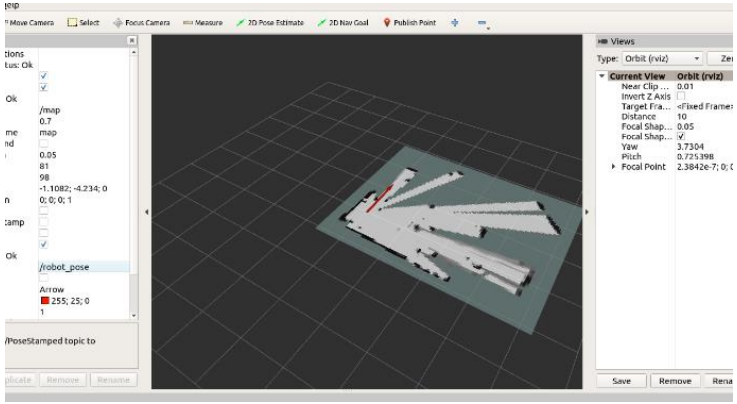
Полезности прочие

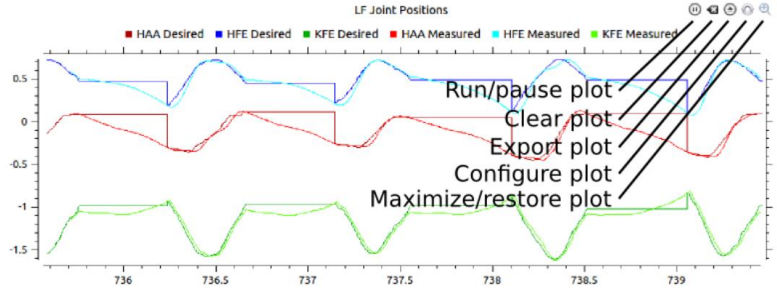
- rospy.get_name() - Получить полное имя этого узла.
- rospy.get_namespace() – Получить текущее пространство имен

roswtf просматривает переменные среды, конфигурации пакетов, конфигурации стека и многое другое. Он также может получить файл roslaunch и попытаться найти в нем любые потенциальные проблемы с конфигурацией, например пакеты, которые не были собраны должным образом и т.д.

rqt_multiplot – плагин для визуализации 2D данных

Rviz – инструмент для 3D визуализации





ITMO re than a UNIVERSITY

Спасибо за внимание!