Практика ROS Services and Actions

Создание сервисв

1. Начнем с оздания типа сервиса.

`mkdir srv`

`cd srv`

`touch AddTwoInts.srv`

Этот файл должен содержать две части: Часть запроса и часть ответа. Откроем этот файл в редакторе VS Code и запишем туда следующее:

```
int64 a
int64 b
---
int64 sum
```

Теперь необходимо внести исправления в **CMakeLists.txt** чтобы при следующей сборке пакета выполнилась сборка нового типа сообщений. Для этого нужно найти фрагмент

```
# add_service_files(
#   FILES
#   Service1.srv
#   Service2.srv
# )
```

и змeнить на

```
add_service_files(
  FILES
  AddTwoInts.srv
)
```

Также необходимо убедиться, что раскомментированы следующие фрагменты

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

```
<build_depend>message_generation</build_depend>
  <exec_depend>message_runtime</exec_depend>
```

Если все сделано правильно, то после сборки пакета вы можете выполнить команду

`rossrv show my_robot_controller/AddTwoInts`

и увидеть содержимоей нового типа сообщений.

> При создании новых типов сообщений и сервисов бывет удобнее скопировать уже имеющийся и отредактировать уже его. Для этого удобнно пользоваться командой
> `roscp` [package_name] [file_to_copy_path] [copy_path]

2. Чтобы сервис работал, необхходимо 2 компонента: Сервер и Клиент. Начнем с создания сервера сервиса:

   `touch scripts/service_server.py`

   И в редакторе укажем содержимое файла следующее:

```
#!/usr/bin/env python3

from __future__ import print_function

from my_robot_controller.srv import AddTwoInts,AddTwoIntsResponse
import rospy

def handle_add_two_ints(req):
    # Здесь будут обрабатываться запросы к серверу
    print("Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b)))
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
    print("Ready to add two ints.")
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```

Не смотря на то, что мы пишем на питоне, catkin все равно нужно указывать список исполняемых файлов в **CMakeLists.txt**, чтобы потом наши скрипты можно было экспортировать в другие пакеты. Для этого находим и редактируем в **CMakeLists.txt** :

```
# catkin_install_python(PROGRAMS
#   scripts/my_python_script
#   DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
# )
```

Заменить на следующий фрагмент

```
catkin_install_python(PROGRAMS
  scripts/service_server.py scripts/service_client.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

Чтобы проверить работу сервиса можно сначала запустить сервер

`rosrun my_robot_controller scripts/service_server.py`

А потом запустить ноду Клиента и передать 2 числа

`rosrun my_robot_controller scripts/service_client.py 2 4`

Также можно вызвать сервис из терминала:

```
rosservice call /add_two_ints "a: 5
b: 3"
```

Создать файл `scripts/service_client.py` со следующим содержимым:

```
#!/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)))
```

ROS Actions

1. Для данной части работы мы создадим новый пакет

`catkin_create_pkg my_action_demo_controller roscpp rospy std_msgs actionlib_msgs`

Вы также можете уже в существующий файл дописать зависимости и не создавать новый пакет. Но в любом случае вам необходимо убедиться, что в файле CMakeLists.txt имеются следующие зависимости.

```
find_package(catkin REQUIRED genmsg actionlib_msgs)
add_action_files(
  FILES
  Timer.action
)
generate_messages(DEPENDENCIES actionlib_msgs)

catkin_package(
CATKIN_DEPENDS actionlib_msgs roscpp rospy std_msgs
)
```

Также в package.xml необходимо проверить:

```
<build_depend>actionlib_msgs</build_depend>
<exec_depend>actionlib_msgs</exec_depend>
```

2. Создадим новый тип action под названием Timer

   `mkdir action`

   `touch action/Timer.action`

   И в этот файл добавим следующее определение типа нашего Action

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

```
duration time_elapsed
duration time_remaining
```

3. Создаем сервер

```
#!/usr/bin/env python3

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()
```

4. Создаем клиента

```
#!/usr/bin/env python3
```

```
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))
```

Задание

Написать экшн, который выполняет задачу одометрии для turlesim

Ссылки на ROS Wiki:

http://wiki.ros.org/ROS/Tutorials/WritingServiceClient(python)

http://wiki.ros.org/actionlib