UNIVERSITÀ DEGLI STUDI DI TRENTO
Dipartimento di Ingegneria
e Scienza dell'Informazione

Fundamentals of Robotics
ROS programming – Part 3

Luigi Palopoli & Nicu Sebe & Michele Focchi

Creation of messages and services

- Let us try to see how to create our own services and messages
- Specifically
  - **msg:** msg files are simple text fiels that describe the fields of a ROS message. They can be used to generate messages in different languages (stored in the msg directory of a package)
  - **srv**: describe a service and are composed of a request and a response (stored in the srv directory of a package)
- Messages are text files with a filed type and a filed name per line
  - int8, int16, int32, int64 (plus uint*)
  - float32, float64
  - string
  - time, duration
  - other msg files
  - variable-length array[] and fixed-length array[C]
  - in addition there is a Header type that contains a timestamp and a coordinate frame information (commonly used in ROS)

Example of messages and services

- An example of a message using header and string primitives is the following

```
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

- A service is very similar, excpet that it contains request and response.

```
int64 A
int64 B
---
int64 Sum
```

Creation of a message

- Let us create a message in our beginner_tutorials package.

```
> roscd beginner_tutorials
> mkdir msg
> echo "int64 num" >
msg/Num.msg
```

- Our message will have to be translated into C++, Python, etc,
- First edit your package.xml uncommenting a couple of lines (the first used at build time and the second used at runtime

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

- Now, modify CMakeLists.txt to add a dependence from message generation

```
# Do not just add this to your CMakeLists.txt, modify the existing text to add
message_generation before the closing parenthesis
find_package(catkin REQUIRED COMPONENTS
   roscpp
   rospy
   std_msgs
   message_generation)
```

Creation of a message

Let us make sure to export the message run-time dependency

```
catkin_package(
  ...
  CATKIN_DEPENDS message_runtime ...
  ...)
```

We can now uncomment the line of code to add message files

```
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )
```

→

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

One last line to force the message generation

```
# generate_messages(
#   DEPENDENCIES
#   std_msgs
# )
```

→

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

Creation of a message

- After compiling, We can verify if the message has been created by

```
> rosmsg show
beginner_tutorials/Num
```

- If you do not remember the package you can as well use

```
> rosmsg show Num
```

Creating a service

- The creation of a service goes through similar steps

```
> roscd beginner_tutorials
> mkdir srv
```

- we can copy a service from another package

```
> roscp rospy_tutorials AddTwoInts.srv srv/AddTwoInts.srv
```

- We follow the same steps as we did for message generation to create dependencies (the steps are common)

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

```
# Do not just add this to your CMakeLists.txt, modify the existing text to add message_generation
before the closing parenthesis
find_package(catkin REQUIRED COMPONENTS
   roscpp
   rospy
   std_msgs
   message_generation
)
```

Creating a service

- And then force the creation of the service

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

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

- We can check that the service has been created by:

```
> rossrv show beginner_tutorials/AddTwoInts
```

Service: writing the Server node (C++)

- Create the src/add_two_ints_server.cpp file within the beginner_tutorials package and paste the following inside it:
- here we will create the service ("add_two_ints_server") node which will receive two ints and return the

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

bool add(beginner_tutorials::AddTwoInts::Request  &req,
         beginner_tutorials::AddTwoInts::Response &res)
{
   res.sum = req.a + req.b;
   ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
   ROS_INFO("sending back response: [%ld]", (long int)res.sum);
   return true;
}

int main(int argc, char **argv)
{
   ros::init(argc, argv, "add_two_ints_server");
   ros::NodeHandle n;

   ros::ServiceServer service = n.advertiseService("add_two_ints", add);
   ros::spin();

   return 0;
}
```

The header file generated from the srv file that we created earlier.

function that provides the service for adding two ints

The service is created and advertised over ROS.

Everytime a client ask for service the function add(req, res) is called

Service:writing the Client Node (C++)

- Create the src/add_two_ints_client.cpp file within the beginner_tutorials package and paste the following

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"
#include <cstdlib>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_client");
  if (argc != 3)
  {
    ROS_INFO("usage: add_two_ints_client X Y");
    return 1;
  }
  ros::NodeHandle n;
  ros::ServiceClient service_client = n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_ints");
  beginner_tutorials::AddTwoInts srv;
  srv.request.a = atoll(argv[1]);
  srv.request.b = atoll(argv[2]);
  if (service_client.call(srv))
  {
    ROS_INFO("Sum: %ld", (long int)srv.response.sum);
  }
  else
  {
    ROS_ERROR("Failed to call service add_two_ints");
    return 1;
  }
  return 0;
}
```

The header file generated from the srv file that we created earlier.

init ROS node

Creates a client for the add_two_ints service

Instantiate the service class

Fill in the service request member

calls the service
(returns true if succeded and the value in srv.response)

unpacks the response message

Building the nodes

- Again edit the beginner_tutorials CMakeLists.txt located at ~/catkin_ws/src/beginner_tutorials and add the following lines:

```
add_executable(add_two_ints_server src/add_two_ints_server.cpp)
target_link_libraries(add_two_ints_server ${catkin_LIBRARIES})
add_dependencies(add_two_ints_server beginner_tutorials_gencpp)

add_executable(add_two_ints_client src/add_two_ints_client.cpp)
target_link_libraries(add_two_ints_client ${catkin_LIBRARIES})
add_dependencies(add_two_ints_client beginner_tutorials_gencpp)
```

- Now run catkin_make. This will create two executables: **add_two_ints_server** and **add_two_ints_client**, which by default will go into package directory of your devel space.

Running the nodes

- Run the server opening a new shell and typing:

```
>rosrun beginner_tutorials add_two_ints_server
```

- You should see something similar to:

```
Ready to add two ints.
```

- Now let's run the client with the necessary arguments, in another shell:

```
>rosrun beginner_tutorials add_two_ints_client 1 3
```

- In the client's shell, you should see something similar to:

```
Sum: 4
```

- We could have obtained the same result, calling directly the service from terminal

```
>rosservice call /add_two_ints 1 3
```

and now an application

A simple publisher subscriber

- Let us switch to the src directory and let us write our first application

```
> roscd beginner_tutorials
> mkdir -p src
```

- We now activate our preferred editor and we start writing our application (talker.cpp and listener.cpp)

Talker.cpp

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;

  ros::Publisher chatter_pub =
n.advertise<std_msgs::String>("chatter", 1000);
  ros::Rate loop_rate(10);

  int count = 0;
  while (ros::ok())
  {
    std_msgs::String msg;

    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());

    chatter_pub.publish(msg);

    ros::spinOnce();

    loop_rate.sleep();
    ++count;
  }


  return 0;
}
```

Initialise ROS. We specify the name of our node. Node names must be unique in a running system.

Create a handle to this process' node. Initialises the node with the master (cleaned up upon destructuion)

Declare to publish on a topic and get a publisher in return

specify the frequency that the node loops at (in conjuction with sleep)

Creation of a message

Print on the screen

Publishing

Used for callbacks (not really needed here)

listener.cpp

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

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

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

    ros::NodeHandle n;

   ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);

  ros::spin();

  return 0;
}
```

Callback code

Initialise ROS node

Register

Subscribe to chatter topic

Keep waiting for callbacks

To build your package

- Just append the following to your CMakeLists.txt

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

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

- This will create the two executable. Now we need to create a dependency between the executable and the message generation

```
add_dependencies(talker   beginner_tutorials_generate_messages_cpp)
add_dependencies(listener   beginner_tutorials_generate_messages_cpp)
```

- You may have dependencies also from other packages. You can sort them out by

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

Build

- We can now build by

```
# In your catkin workspace
> cd ~/catkin_ws
> catkin_make
> source devel/setup.bash
```

- To run the newly created node just type

```
rosrun beginner_tutorials talker
```

```
rosrun beginner_tutorials listener
```

Assignment

- It is time to start developing your task planning node
- The task planning node exchanges services / topics with the **motion planning** node and the **vision node**
- We want to send target positions from the vision node to the motion planning node
- Create your own custom **target pose** message containing
  - vector3 msg for the grasping position
  - quaternion msg for the grasping orientation (expressed in quaternions)
  - vector3 msg for the grasping orientation (expressed in euler angles)
- Create a service called /setTarget that sends the target pose message