Robotics- ROS Assignment 4

Name: Shrirang Alias Samarth Patil
University: Vellore Institute of Technology, Bhopal
Contact No.: 8999171921
Email ID: shrirang.alias2019@vitbhopal.ac.in

Identify any of the following things and Make a document about their functionality and importance

- ROS Topics

Topics implement a publish/subscribe communication mechanism, one of the more common ways to exchange data in a distributed system. Before nodes start to transmit data over topics, they must first announce, or advertise, both the topic name and the types of messages that are going to be sent. Then they can start to send, or publish, the actual data on the topic. Nodes that want to receive messages on a topic can subscribe to that topic by making a request to roscore. After subscribing, all messages on the topic are delivered to the node that made the request. One of the main advantages to using ROS is that all the messy details of setting up the necessary connections when nodes advertise or subscribe to topics is handled for you by the underlying communication mechanism so that you don't have to worry about it yourself In ROS, all messages on the same topic must be of the same data type. Although ROS does not enforce it, topic names often describe the messages that are sent over them. For example, on the PR2 robot, the topic /wide_stereo/right/image_color is used for color images from the rightmost camera of the wide-angle stereo pair.

Publishing to a Topic To send data on a topic, you have to be a publisher. It involves "announcing" that you will publish on a topic and mention the Data type that will be published. Also, remember since topic names are unique, if you announce a topic who's name already exists, it get overwritten. You can use rostopic to see the status. rostopic list shows all channels currently online. rostopic echo topicname will show the messages on the topic called topicname. Adding the -n <num_msgs> flag will limit the number of messages to <num_msgs>

Subscribing to a topic For listening to a topic, ROS provides an asynchronous way to listen to the messages. Example here,

- ROS Services

Services are another way to pass data between nodes in ROS. Services are just synchronous remote procedure calls; they allow one node to call a function that executes in another node. Service calls are well suited to things that you only need to do occasionally and that take a bounded amount of time to complete.

The first step in creating a new service is to define the service call inputs and outputs. This is done in a service-definition file , which has a similar structure to the message-definition. However, since a service call has both inputs and outputs, it's a bit more complicated than a message.

The inputs to the service call come first. In this case, we're just going to use the ROS built-in string type. Three dashes (—) mark the end of the inputs and the start of the output definition. We're going to use a 32-bit unsigned integer (uint32) for our output. The file holding this definition is called WordCount.srv and is traditionally in a directory called srv in the main package directory (although this is not strictly required). After saving this, run catkin_make to create the code and definitions. The find_package() call in CMakeLists.txt needs to include message_generation. Also another update required is the addition of add_service_files() section. Also the package.xml needs to be updated catkin_make will generate the classes - WordCount, WordCountRequest and WordCountResponse .

- ROS Actions

ROS actions are the best way to implement interfaces to time-extended, goal-oriented behaviors like goto_position While services are synchronous, actions are asynchronous. Similar to the request and response of a service, an action uses a goal to initiate a behavior and sends a result when the behavior is complete. But the action further uses feedback to provide updates on the behavior's progress toward the goal and also allows for goals to be canceled.

Actions are themselves implemented internally using topics. It is essentially a higher-level protocol that specifies how a set of topics (goal, result, feedback, etc.) should be used in combination.

For example - Using an action interface to goto_position , you send a goal, then move on to other tasks while the robot is driving. Along the way, you receive periodic progress updates (distance traveled, estimated time to goal, etc.), culminating in a result message (did the robot make it to the goal or was it forced to give up?). And if something more important comes up, you can at any time cancel the goal and send the robot somewhere else. Actions require only a little more effort to define and use than do services, and they provide a lot more power and flexibility.

The first step in creating a new action is to define the goal, result, and feedback message formats in an action definition file, which by convention has the suffix .action. The .action file format is similar to the .srv format used to define services, just with an additional field. And, as with services, each field within an .action file will become its own message. Just like with service-definition files, we use three dashes (—) as the separator between the parts of the definition. While service definitions have two parts (request and response), action definitions have three parts (goal, result, and feedback). This is an action definition file for simple Timer , which has three parts: the goal, the result, and the feedback.

- Navigation messages, Geometry Messages, and sensor messages.

nav_msgs defines the common messages used to interact with the navigation stack.

geometry_msgs provides messages for common geometric primitives such as points, vectors, and poses. These primitives are designed to provide a common data type and facilitate interoperability throughout the system.

sensor_msgs package defines messages for commonly used sensors, including cameras and scanning laser rangefinders.