Name : Prem Sharan S
Regno : 19BEE0003

Assignment-4

a) ROS Topics:

ROS Topics are named buses over which nodes exchange messages. Topics have anonymous publish/subscribe semantics, which decouples the production of information from its consumption. In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data *subscribe* to the relevant topic; nodes that generate data *publish* to the relevant topic. There can be multiple publishers and subscribers to a topic.

Topics are intended for unidirectional, streaming communication. Nodes that need to perform remote procedure calls, i.e. receive a response to a request, should use services instead. There is also the Parameter Server for maintaining small amounts of state.

b) ROS Services:

The publish / subscribe model is a very flexible communication paradigm, but its many-to-many one-way transport is not appropriate for RPC request / reply interactions, which are often required in a distributed system. Request / reply is done via a *Service,* which is defined by a pair of messages: one for the request and one for the reply. A providing ROS node offers a service under a string name, and a client calls the service by sending the request message and awaiting the reply. Client libraries usually present this interaction to the programmer as if it were a remote procedure call.

Services are defined using srv files, which are compiled into source code by a ROS client library.

A client can make a *persistent connection* to a service, which enables higher performance at the cost of less robustness to service provider changes.

Let us take an example to understand ROS Topics and ROS Services Let us consider Turtlesim Node that provides a simple simulator to control a 'Turtle' character on screen. Various related topics:

1) turtleX/cmd_vel(x - 1/2/3...) : The linear and angular command velocity for turtleX. The turtle will execute a velocity command for 1 second then time out. Twist.linear.x is the forward velocity, Twist.linear.y is the strafe velocity, and Twist.angular.z is the angular velocity. We can publish to this topic to move the bot

2) turtleX/pose(x - 1/2/3...) : We can subscribe to this topic, to get relevant data such as the x, y, theta, linear velocity, and angular velocity of turtleX.

Various related services:

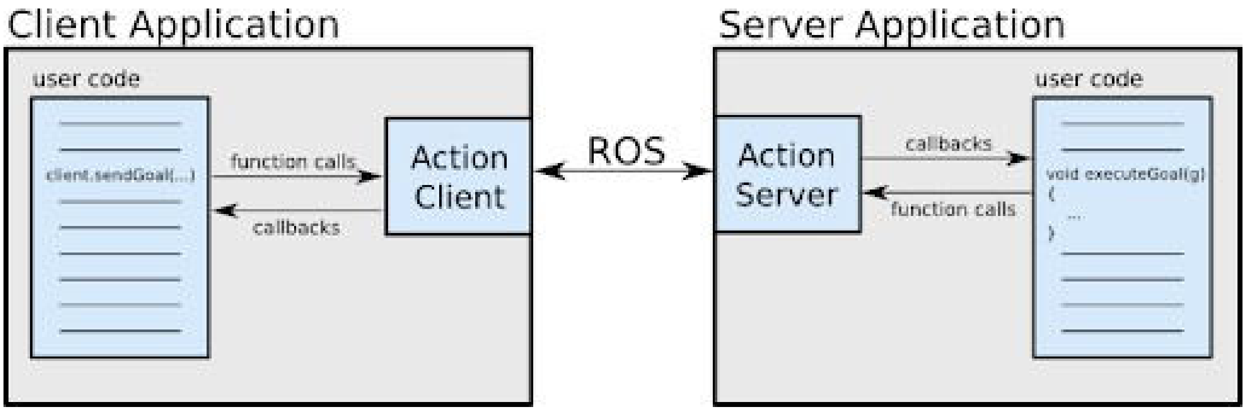
1) clear: Clears the turtlesim background and sets the color to the value of the background parameters.

2) reset: Resets the turtlesim to the start configuration and sets the background color to the value of the background.

3) kill: Kills a turtle by name.

C) **<u>ROS actions:</u>**

An action is an asynchronous call to another node's functionality. “Asynchronous” means we don't have to wait for the result. We can do other things pending the result. The node that provides the functionality has to implement an Action Server.



Goal To accomplish tasks using actions, we introduce the notion of a goal that can be sent to an ActionServer by an ActionClient. In the case of moving the base, the goal would be a PoseStamped message that contains information about where the robot should move to in the world. For controlling the tilting laser scanner, the goal would contain the scan parameters (min angle, max angle, speed, etc). Feedback Feedback provides server implementers a way to tell an ActionClient about the incremental progress of a goal. For moving the base, this might be the robot's current pose along the path. For controlling the tilting laser scanner, this might be the time left until the scan completes. Result A result is sent from the ActionServer to the ActionClient upon completion of the goal. This is different from feedback, since it is sent exactly once. This is extremely useful when the purpose of the action is to provide some sort of information. For move base, the result isn't very important, but it might contain the final pose of the robot. For controlling the tilting laser scanner, the result might contain a point cloud generated from the requested scan.

d) <u>The three messages:</u>

● Navigation Messages -

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

● Geometry Messages - Also known as 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 Messages - Also known as sensor_msgs this package defines messages for commonly used sensors, including cameras and scanning laser rangefinders.