ROS Package

- ROS Packages according to ROS Wiki

  Software in ROS is organized in *packages*. A package might contain ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module. The goal of these packages it to provide this useful functionality in an easy-to-consume manner so that software can be easily reused. In general, ROS packages follow a "Goldilocks" principle: enough functionality to be useful, but not too much that the package is heavyweight and difficult to use from other software.

Reference

1. Packages

Create a ROS Package

- This tutorial will demonstrate how to use the `catkin_create_pkg` script to create a new catkin package, and what you can do with it after it has been created.

1. First, navigate to the source space directory of the catkin workspace you've created.

   ```
   cd ~/catkin_ws/src
   ```

2. Now, use the `catkin_create_pkg` script to create a new package called `pkg_ros_basics` which depends on std_msgs, roscpp, and rospy:

   ```
   catkin_create_pkg pkg_ros_basics std_msgs rospy roscpp
   ```

   - This will create a beginner_tutorials folder which contains a `package.xml` and a `CMakeLists.txt`, which have been partially filled out with the information you gave `catkin_create_pkg`.
   - `catkin_create_pkg` requires that you give it a `package_name` and optionally a list of dependencies on which that package depends:

   ```
   catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
   ```

3. Now, you need to build the packages in the catkin workspace:

   ```
   cd ~/catkin_ws
   ```

   ```
   catkin build
   ```

- Inside the package, there are `src` folder, `package.xml`, `CMakeLists.txt`, and the `include` folders.
    - **CMakeLists.txt:** This file has all the commands to build the ROS source code inside the package and create the executable. For more information about CMakeLists visit here.
    - **package.xml:** This is an XML file. It mainly contains the package dependencies, information, and so forth.
    - **src:** The source code of ROS packages are kept in this folder.

ROS Nodes

- A ROS Node is a piece of software/executable that uses ROS to communicate with other ROS Nodes.
- ROS Nodes are the building block of any ROS Application.
- For example, if you have a wall-following robot then one ROS Node could get distance sensor values and another node can control the motors of the robot. So, these two nodes will communicate with each other in order to move the robot.
- You can write your entire ROS Application in a single node but having multiple nodes ensures that if a node crashes it won't crash your entire ROS application.
- In this eYRC Theme your job will be to write suitable ROS Nodes for your ROS Application.
- A ROS package can have multiple ROS Nodes.
- Python and C++ are majorly used to write ROS Nodes.
- In this eYRC Theme we will use Python to write ROS Nodes.

Reference

1. ROS WIki - Understanding Nodes

Create a ROS Node

In this section we will learn how to create a ROS Node inside `pkg_ros_basics` ROS Package which we created in the previous section.

Create a ROS Node

In this section we will learn how to create a ROS Node inside `pkg_ros_basics` ROS Package which we created in the previous section.

1. Navigate to `pkg_ros_basics`.

   ```
   cd ~/catkin_ws/src/pkg_ros_basics
   ```

   OR

   ```
   roscd pkg_ros_basics
   ```

   NOTE: `roscd` will work only if you have sourced `setup.bash` of your catkin workspace.

2. Create a `scripts` folder for your Python scripts and navigate into the folder.

   ```
   mkdir scripts
   ```

   ```
   cd scripts
   ```

3. Create a Python script called `node_hello_ros.py`.

   ```
   touch node_hello_ros.py
   ```

4. Open the script in any text-editor and start editing.

```
gedit node_hello_ros.py
```

5. First line of all your Python ROS scripts should be the following shebang

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

6. Now write a ROS Node to print `Hello World!` on the console.

```
#!/usr/bin/env python
import rospy
def main():
    # 1. Make the script a ROS Node.
    rospy.init_node('node_hello_ros', anonymous=True)
    # 2. Print info on console.
    rospy.loginfo("Hello World!")
    # 3. Keep the node alive till it is killed by the user.
    rospy.spin()
if __name__ == '__main__':
    try:
        main()
    except rospy.ROSInterruptException:
  pass
```

7. Now you have to make this script an executable.

```
sudo chmod +x node_hello_ros.py
```

8. Now in order to run your ROS Node,
    1. Open up a terminal and run ROS Master.

        ```
        roscore
        ```

    2. Once the roscore is up running, open a new terminal and run the ROS Node.

        ```
        rosrun pkg_ros_basics node_hello_ros.py
        ```

        NOTE: This command will work only if you have sourced `setup.bash` of your catkin workspace either manually or using `.bashrc`.

9. You should get some output like this,

    ```
    [INFO] [1601277063.968749]: Hello World!
    ```

Command: rosrun

`rosrun` allows you to run an executable in an arbitrary package from anywhere without having to give its full path or cd/roscd there first.

Usage:

```
rosrun <package> <executable>
```

`<package>` is nothing but the package name which you have created using `catkin_create_pkg` command or used any other package.

`<executable>` is the python or cpp file.

To create an executable python file

After creating a package, create a folder in the package names as `scripts` folder to store all the python files in that folder.

```
cd ~/catkin_ws/src/<package>
```

```
mkdir scripts
```

Here we can create python scripts by running this command by going into the `scripts` directory,

```
cd scripts
```

```
touch filename.py
```

Now you can edit your python file and before running you have to make it executable by running this command once,

```
cd ~/catkin_ws/src/<package>/scripts

chmod +x filename.py
```

To create an executable cpp file

After creating a package, create a folder in the package names as `src` folder to store all the cpp files in that folder.

```
cd ~/catkin_ws/src/<package>

mkdir src
```

Here we can create cpp files by running this command by going into the `src` directory,

```
cd src

touch filename.cpp
```

Now you can edit your cpp file , but for making it executable we have to edit the `CMakeLists.txt` file which is present in the package.

Add these few lines at the bottom of `CMakeLists.txt` file,

```
add_executable(filename src/filename.cpp)

target_link_libraries(filename ${catkin_LIBRARIES})
```

Then run this command,

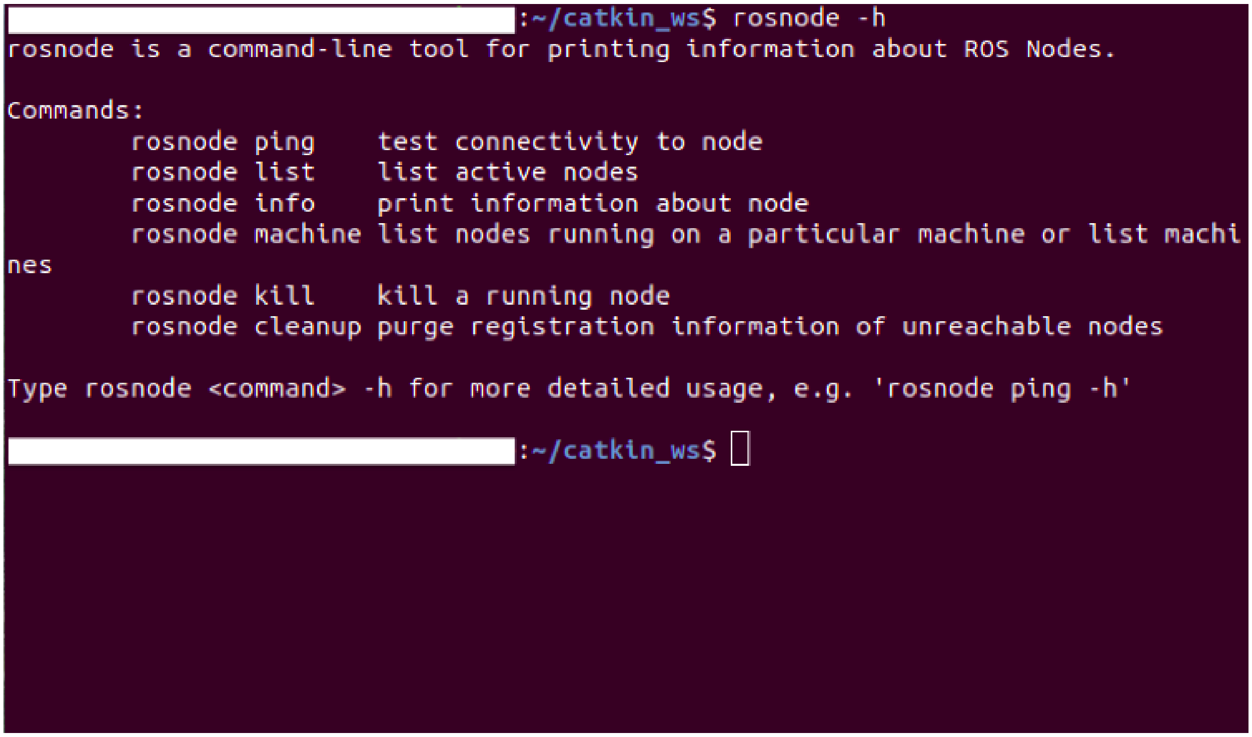
```
cd ~/catkin_ws

catkin build
```

Command: rosnode

`rosnode` contains the rosnode command-line tool for displaying debug information about ROS Nodes.

Note: For quick information about any command, be that outside of ROS, simply type the command along with suffix `--h` or `-help`. This is a widely used concept among other Linux commands for quick references. Here's an example for `rosnode --h` command



```
[redacted]:~/catkin_ws$ rosnode -h
rosnode is a command-line tool for printing information about ROS Nodes.

Commands:
        rosnode ping    test connectivity to node
        rosnode list    list active nodes
        rosnode info    print information about node
        rosnode machine list nodes running on a particular machine or list machi
nes
        rosnode kill    kill a running node
        rosnode cleanup purge registration information of unreachable nodes

Type rosnode <command> -h for more detailed usage, e.g. 'rosnode ping -h'

[redacted]:~/catkin_ws$ 
```

list

`rosnode list` displays a list of all current nodes.

Let's figure out what argument the `list` sub-command needs. In a new terminal run start the rosmaster:

```
roscore
```

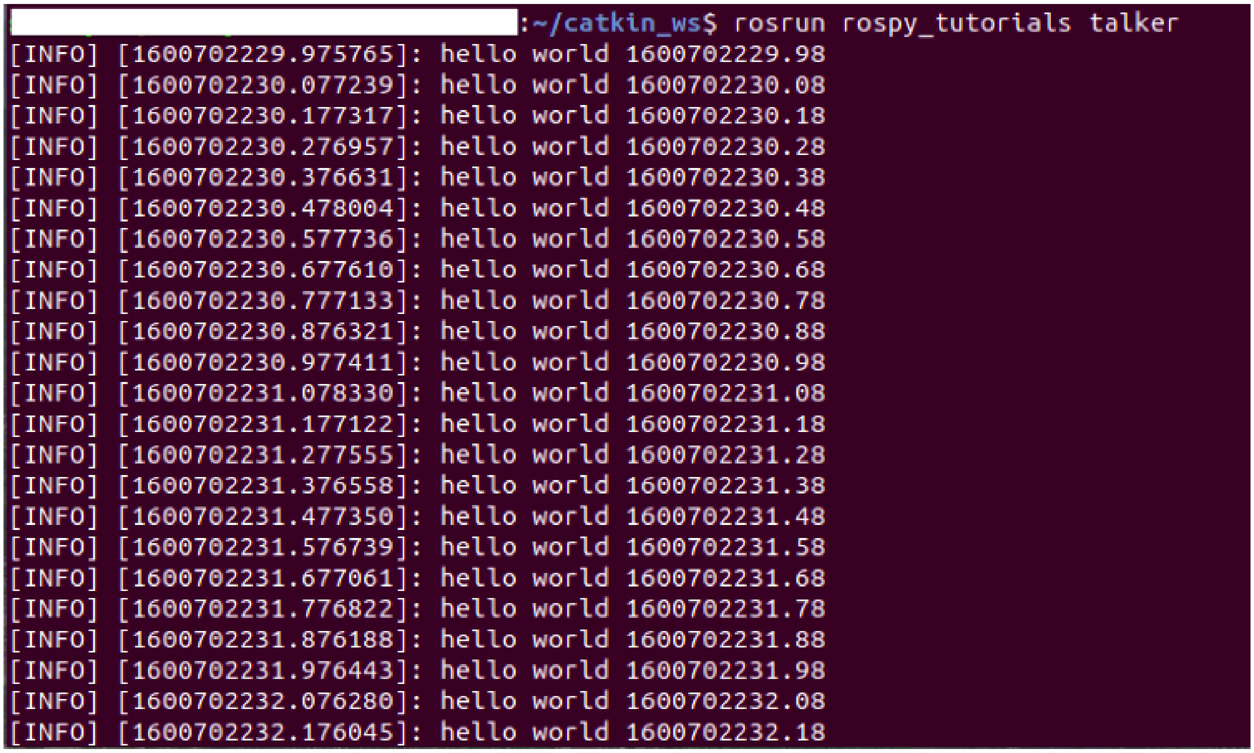
And in another terminal, run:

```
rosrun rospy_tutorials talker
```

And in another terminal, run:

```
rosnode list
```

Now the node named `talker`(node with word talker in it) will be printed on the terminal.



```
[redacted]:~/catkin_ws$ rosrun rospy_tutorials talker
[INFO] [1600702229.975765]: hello world 1600702229.98
[INFO] [1600702230.077239]: hello world 1600702230.08
[INFO] [1600702230.177317]: hello world 1600702230.18
[INFO] [1600702230.276957]: hello world 1600702230.28
[INFO] [1600702230.376631]: hello world 1600702230.38
[INFO] [1600702230.478004]: hello world 1600702230.48
[INFO] [1600702230.577736]: hello world 1600702230.58
[INFO] [1600702230.677610]: hello world 1600702230.68
[INFO] [1600702230.777133]: hello world 1600702230.78
[INFO] [1600702230.876321]: hello world 1600702230.88
[INFO] [1600702230.977411]: hello world 1600702230.98
[INFO] [1600702231.078330]: hello world 1600702231.08
[INFO] [1600702231.177122]: hello world 1600702231.18
[INFO] [1600702231.277555]: hello world 1600702231.28
[INFO] [1600702231.376558]: hello world 1600702231.38
[INFO] [1600702231.477350]: hello world 1600702231.48
[INFO] [1600702231.576739]: hello world 1600702231.58
[INFO] [1600702231.677061]: hello world 1600702231.68
[INFO] [1600702231.776822]: hello world 1600702231.78
[INFO] [1600702231.876188]: hello world 1600702231.88
[INFO] [1600702231.976443]: hello world 1600702231.98
[INFO] [1600702232.076280]: hello world 1600702232.08
[INFO] [1600702232.176045]: hello world 1600702232.18
```

info

`rosnode info /node_name` displays information about a node, including publications and subscriptions.

Let's figure out what argument the `info` sub-command needs. In a new terminal run start the rosmaster:

```
roscore
```

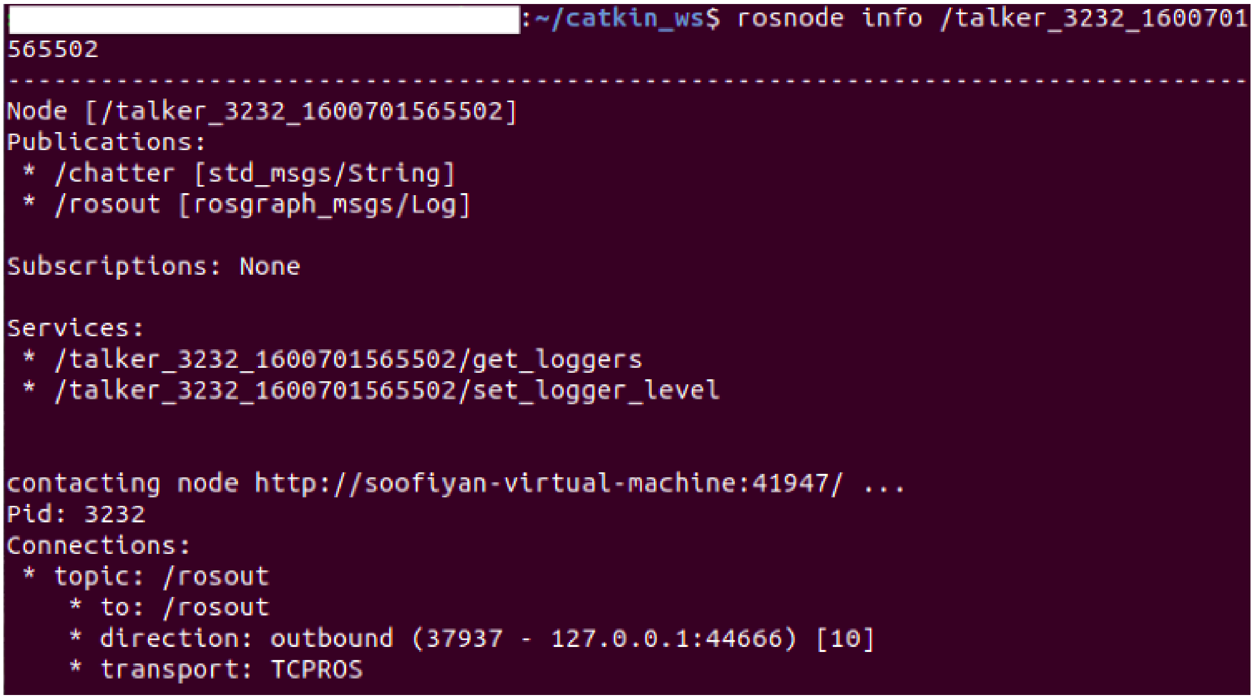
And in another terminal, run:

```
rosrun rospy_tutorials talker
```

And in another terminal, run:

```
rosnode info <talker_node>
```

This should give details related to the particular node as shown below:



```
[redacted]:~/catkin_ws$ rosnode info /talker_3232_1600701
565502
--------------------------------------------------------------------------------
Node [/talker_3232_1600701565502]
Publications:
 * /chatter [std_msgs/String]
 * /rosout [rosgraph_msgs/Log]

Subscriptions: None

Services:
 * /talker_3232_1600701565502/get_loggers
 * /talker_3232_1600701565502/set_logger_level


contacting node http://soofiyan-virtual-machine:41947/ ...
Pid: 3232
Connections:
 * topic: /rosout
    * to: /rosout
    * direction: outbound (37937 - 127.0.0.1:44666) [10]
    * transport: TCPROS
```

kill

IMPORTANT: rosnode kill is not guaranteed to succeed. Let's figure out what argument the `kill` sub-command needs. In a new terminal run start the rosmaster:

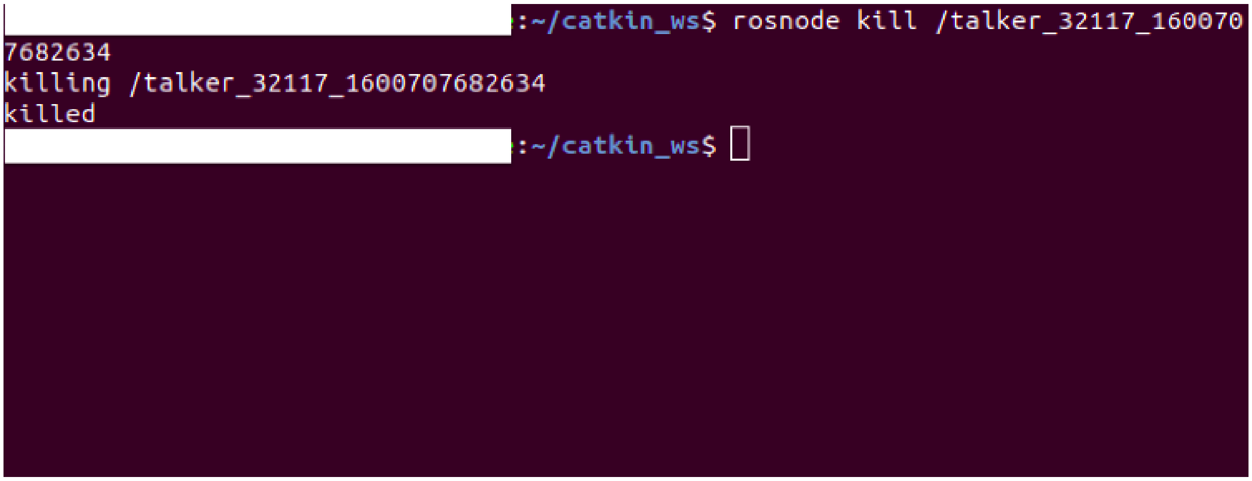
```
roscore
```

And in another terminal, run:

```
rosrun rospy_tutorials talker
```

And in another terminal, run:

```
rosnode kill rosout <talker_node>
```



```
[redacted]:~/catkin_ws$ rosnode kill /talker_32117_160070
7682634
killing /talker_32117_1600707682634
killed
[redacted]:~/catkin_ws$ 
```

Interactive mode. This enables you to select which node to kill from a numbered list, which is useful for killing anonymous nodes.

```
rosnode kill
```

```
1. /rosout
```

```
Please enter the number of the node you wish to kill.
```

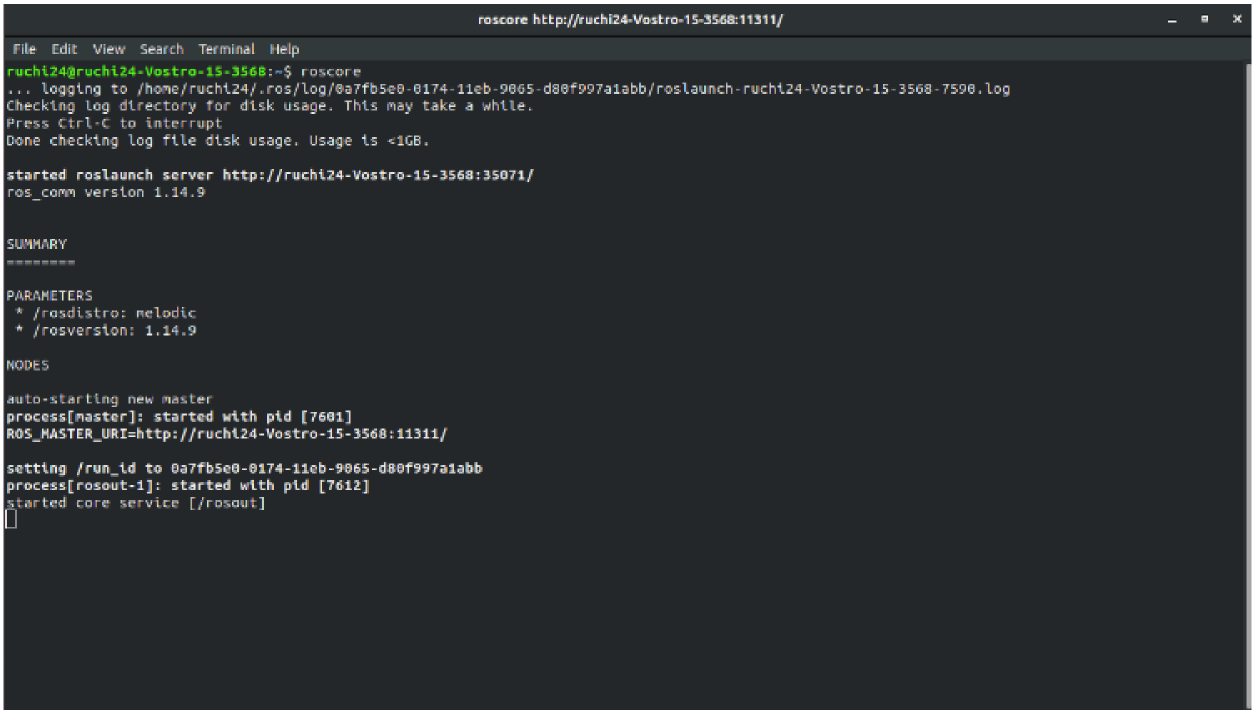
ROS Master

- As you know ROS Nodes are building blocks of any ROS Application. A single ROS Application may have multiple ROS Nodes which communicate with each other.
- The role of the ROS Master is to enable individual ROS nodes to locate one another.
- Once these nodes have located each other they communicate with each other peer-to-peer.
- The ROS Master provides naming and registration services to the rest of the nodes in the ROS system.
- You can say, communication is established between nodes by the ROS Master. So, without ROS Master running ROS Nodes can not communicate with each other.

Start ROS Master

To start ROS Master you just have to enter the following command in the terminal.

```
roscore
```


roscore http://ruchi24-Vostro-15-3568:11311/
File Edit View Search Terminal Help

```
ruchi24@ruchi24-Vostro-15-3568:~$ roscore
... logging to /home/ruchi24/.ros/log/0a7fb5e0-0174-11eb-9065-d80f997a1abb/roslaunch-ruchi24-Vostro-15-3568-7590.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://ruchi24-Vostro-15-3568:35071/
ros_comm version 1.14.9


SUMMARY
========

PARAMETERS
 * /rosdistro: melodic
 * /rosversion: 1.14.9

NODES

auto-starting new master
process[master]: started with pid [7601]
ROS_MASTER_URI=http://ruchi24-Vostro-15-3568:11311/

setting /run_id to 0a7fb5e0-0174-11eb-9065-d80f997a1abb
process[rosout-1]: started with pid [7612]
started core service [/rosout]
```

So roscore will start the following:

1. ROS Master
2. ROS Parameter Server
3. `rosout` Logging Node

In the preceding output, you can see information about the computer, parameter which list the name (Melodic) and version number of ROS distribution, and some other information.

Reading Assignment

1. ROS Wiki - Master
2. ROS Wiki - roscore

ROS Parameter Server

- You can think Parameter Server as a space where all the necessary data that needs to be shared among various ROS Nodes is stored.
- Parameter Server runs inside ROS Master.
- ROS Nodes can view and even modify data stored in the Parameter Server.
- Typically Parameter Server is used to store configuration parameters.

Reading Assignment

1. ROS Wiki - Parameter Server

Load Parameters using YAML file

In this section we will learn how to load your own parameters in ROS Parameter Server using a YAML File.

Steps

1. Navigate to `pkg_ros_basics`.

   ```
   cd ~/catkin_ws/src/pkg_ros_basics
   ```

   OR

   ```
   roscd pkg_ros_basics
   ```

NOTE: `roscd` will work only if you have sourced `setup.bash` of your catkin workspace.

2. Create a `config` folder for your Python scripts and navigate into the folder.

   ```
   mkdir config
   ```

   ```
   cd config
   ```

3. Create a configuration YAML file called `config_my.yaml`.

   ```
   touch config_my.yaml
   ```

4. Open the script in any text-editor and start editing.

   ```
   gedit config_my.yaml
   ```

5. Now fill your config file.

   ```
   # Comment: my_config.yaml Configuration

   details:

     name:

        first: "Hisenberg"  # First Name

        last: "White" # Last Name

     contact:

        address: "XYZ Street, XYZ"  # Address

        phone: 77777    # Contact
   ```

   - ROS Build system will create a Python Dictionary called `details`.
   - This dictionary will have two keys,
     1. Dictionary `name`
     2. Dictionary `contact`

- In your ROS Node you can use `rospy` to get parameters stored in this `config_my` dictionary.

  ```
  param_config_my = rospy.get_param('details')

  first_name = param_config_my['name']['first']

  phone = param_config_my['contact']['phone']
  ```

6. Now if you want to load the parameters defined in the YAML file you have to first start the ROS Parameter Server.
   Open up a new terminal and enter the following.

   ```
   roscore
   ```

7. Now load your parameters.

```
rosparam load config_my.yaml
```

8. Now get the list of parameters loaded in your ROS Parameter Server.

```
rosparam list
```

Output:

```
/details/contact/address

/details/contact/phone

/details/last

/details/name/first

/rosdistro

/roslaunch/uris/host_pc__37763

/rosversion

/run_id
```

Here you can see the first four parameters are loaded from our `config_my.yaml` file.

9. Now to view the content of any parameter do the following.

   ```
   rosparam get /details/contact/phone
   ```

   Output:

   ```
   77777
   ```

   This is the value which we defined in the `config_my.yaml` file.

Example #1: ROS Node to Get and Set Parameters

Aim

To write a ROS Node to read `config_my.yaml` file loaded in ROS Parameter Server , print it on the console and modify the phone number.

Code

node_param_get_set.py

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

import rospy


def main():

    # 1. Make the script a ROS Node.
    rospy.init_node('node_param_get_set', anonymous=True)

    # 2. Read from Parameter Server
    rospy.loginfo("Reading from Parameter Server.")


    param_config_my = rospy.get_param('details')    # Get all the parameters
inside 'details'

    # Store the parameters in variables
    first_name = param_config_my['name']['first']
    last_name = param_config_my['name']['last']
    address = param_config_my['contact']['address']
    phone = param_config_my['contact']['phone']

    # Print the parameters
    rospy.loginfo(">> First Name: {}".format(first_name))
    rospy.loginfo(">> Last Name: {}".format(last_name))
    rospy.loginfo(">> Address: {}".format(address))
    rospy.loginfo(">> Phone: {}".format(phone))

    # 3. Modify the Phone Number

    rospy.set_param('/details/contact/phone', 55555)        # Modify only Phone
Number in Parameter Server
    new_phone = rospy.get_param('/details/contact/phone')   # Get only Phone
Number from Parameter Server
    rospy.loginfo(">> New Phone: {}".format(new_phone))     # Print the new
Phone Number



if __name__ == '__main__':
    try:
        main()
    except rospy.ROSInterruptException:
        pass
```

Output

```
[INFO] [1601389248.001963]: Reading from Parameter Server.
[INFO] [1601389248.007928]: >> First Name: Hisenberg
[INFO] [1601389248.010338]: >> Last Name: White
[INFO] [1601389248.012679]: >> Address: XYZ Street, XYZ
[INFO] [1601389248.014838]: >> Phone: 77777
[INFO] [1601389248.020719]: >> New Phone: 55555
```

- The code is self-explanatory.

ROS Launch Files

- In the previous sections you must have noticed that we need to use `roscore` command to start ROS Master and Parameter Server, `rosrun` command to run a ROS Node, `rosparam load` command to load parameters etc.
- This is a tedious process to manually run nodes and load parameters.
- Launch files provides the capability to do all these stuff using a single command.
- The idea is to mention all the nodes that you want to run, all the config file that you want to load etc in a single file which you can run using `roslaunch` command.

Reading Assignment

1. ROS Wiki - roslaunch

Create a ROS Launch File

roslaunch Command

- `roslaunch` is a tool for easily launching multiple ROS nodes locally and remotely via SSH.
- It includes options to automatically respawn processes that have already died. `roslaunch` takes in one or more XML configuration files (with the .launch extension) that specify the parameters to set and nodes to launch.
- Usage:

  ```
  roslaunch <package> file.launch
  ```

-
- `<package>` is nothing but the package name which you have created using `catkin_create_pkg` command or used any other package.

Steps to create a launch file

1. After creating a package, create a folder in the package names as a `launch` folder to store all the launch files in that folder.

   ```
   cd ~/catkin_ws/src/<package>
   ```

   ```
   mkdir launch
   ```

2. Here we can create launch files by running this command by going into the `launch` directory, we can keep any name for the launch file,

```
cd launch
```

```
touch filename.launch
```

Now you can edit your launch file by adding different nodes that you have to run simultaneously.

Steps to add a ROS node in the launch file

1. Launch files always starts with

   ```
   <launch>
   ```

   and end with

   ```
   </launch>
   ```

2. Now to add any executable file which we have seen in the rosrun_command section, we have to add this line,

   ```
   <node pkg="name_of_package" type="name_of_executable.py"
   name="name_of_executable" output="screen"/>
   ```

   - `pkg` is the package name which you have created
   - `type` is the name of executable file
   - `name` is the name of the node which is created in that executable
   - `output` means it will print the data given to the roslog command

Steps to load Config YAML file in ROS Parameter Server

- You can use `rosparam` tag to load the YAML file.

  ```
  <rosparam file ="$(find name_of_package)/config/config.yaml"/>
  ```

  - `name_of_package` is the name of your ROS package.
  - `config.yaml` is the name of your configuration file.

Steps to add a Shell Script in the launch file

- You can use `node` tag to run any shell script using launch file

  ```
  <node pkg="name_of_package" type="shell_script.sh"
  name="shell_script" output="screen">

  <param name="cmd" value="$(find
  name_of_package)/launch/shell_script.sh"/>

  </node>
  ```

  - `name_of_package` is the name of your ROS package.
  - `shell_script.sh` is the name of your configuration file.
  - `/launch/shell_script.sh` is the location of the shell script inside your ROS Package folder.

Example #1: Launch two ROS Nodes

Aim

- To launch `talker` and `listener` node present in `rospy_tutorials` package.
- For this create a `chatter.launch` file and save it in the `launch` folder inside `pkg_ros_basics` package.

NOTE: To install `rospy_tutorials` package in your system you can run `sudo apt-get install ros-melodic-ros-tutorials` this command.

Once installed, you can use `listener` python script and talker executable written in C++ present in `rospy_tutorials` package.

Code

`chatter.launch`

```
<launch>
  <node name="talker" pkg="rospy_tutorials" type="talker" output="screen"/>
  <node name="listener" pkg="rospy_tutorials" type="listener.py"
output="screen"/>
</launch>
```

- Here first `talker.cpp` file (for cpp file we dont need to add .cpp extension) has been included with the node name as talker and also set output as screen so you can see the output from talker node.
- Next we have added `listener.py` which has node name as listener and here also we have set output as screen.

Run Command

Now run these command to run the launch file,

```
roslaunch pkg_ros_basics chatter.launch
```

Example #2: Launch Turtle in Forest

Aim

- To write a launch file to run `turtlesim_node` node and `turtle_teleop_key` node present in `turtlesim` package.
- While launching the `turtlesim_node` make sure to change the background colour of the simulator from blue to forest green.
- Name the launch file `turtlesim.launch` and save it in `launch` folder inside `pkg_ros_basics` package.

Code

`turtlesim.launch`

```
<launch>
```

```
    <node pkg="turtlesim" type="turtlesim_node" name="node_turtlesim_node">
        <param name="/turtlesim_node/background_r" value="34" />
        <param name="/turtlesim_node/background_g" value="139" />
        <param name="/turtlesim_node/background_b" value="34" />
        <param name="/background_r" value="34" />
        <param name="/background_g" value="139" />
        <param name="/background_b" value="34" />
    </node>
    <node pkg="turtlesim" type="turtle_teleop_key"
name="node_turtle_teleop_key" />

</launch>
```

Run Command

```
roslaunch pkg_ros_basics turtlesim.launch
```

- The code is self-explanatory.
- If you are not able to understand the code feel free to seek help from us.

Example #3: Load YAML

Aim

- To write a launch file to load `config_my.yaml` present in `pkg_ros_basics` package.
- Also launch the `node_param_get_set.py` ROS node after loading the YAML file.

Code

`load_yaml.launch`

```
<launch>

    <rosparam file ="$(find pkg_ros_basics)/config/config_my.yaml"/>

    <node pkg="pkg_ros_basics" type="node_param_get_set.py"
name="node_param_get_set" output="screen"/>

</launch>
```

Run Command

```
roslaunch pkg_ros_basics load_yaml.launch
```

Output

```
... logging to
/home/user/.ros/log/e4944c60-025e-11eb-9079-40ec993efb48/roslaunch-pc-16736.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://ey-pc:39407/

SUMMARY
========

PARAMETERS
 * /details/contact/address: XYZ Street, XYZ
 * /details/contact/phone: 77777
 * /details/name/first: Hisenberg
 * /details/name/last: White
 * /rosdistro: melodic
 * /rosversion: 1.14.7

NODES
  /
    node_param_get_set (pkg_ros_basics/node_param_get_set.py)

ROS_MASTER_URI=http://localhost:11311

process[node_param_get_set-1]: started with pid [16751]
[INFO] [1601393750.973627]: Reading from Parameter Server.
[INFO] [1601393750.977933]: >> First Name: Hisenberg
[INFO] [1601393750.980172]: >> Last Name: White
[INFO] [1601393750.982522]: >> Address: XYZ Street, XYZ
[INFO] [1601393750.985037]: >> Phone: 77777
[INFO] [1601393750.990911]: >> New Phone: 55555
[node_param_get_set-1] process has finished cleanly
log file:
/home/ey-pc/.ros/log/e4944c60-025e-11eb-9079-40ec993efb48/node_param_get_set-1*
.log
all processes on machine have died, roslaunch will exit
shutting down processing monitor...
... shutting down processing monitor complete
done
```

- The code is self-explanatory.
- If you are not able to understand the code feel free to seek help from us.

Example #4: Launch Shell Script and ROS Node

Aim

- To write a launch file called `web_node.launch` to open `e-yantra.org` in firefox and run `node_hello_ros.py` of `pkg_ros_basics`.
- You need to write a shell script called `webpage_launch.sh` to open `e-yantra.org` in firefox and save it in `launch` folder of `pkg_ros_bascis`.

Codes

`webpage_launch.sh`

```bash
#!/bin/bash

# Store URL in a variable
URL1="https://www.e-yantra.org/"

# Print some message
echo "** Opening $URL1 in Firefox **"

# Use firefox to open the URL in a new window
firefox -new-window $URL1
```

NOTE: You need to make this shell script an executable using `chmod` before using it in a launch file.

```
web_node.launch
```

```
<launch>

    <node pkg="pkg_ros_basics" type="webpage_launch.sh" name="webpage-launch"
output="screen">
        <param name="cmd" value="$(find
pkg_ros_basics)/launch/webpage-launch.sh"/>
    </node>

    <node pkg="pkg_ros_basics" type="node_hello_ros.py" name="node_hello_ros"
output="screen"/>

</launch>
```

Run Command

```
roslaunch pkg_ros_basics web_node.launch
```

Output

```
... logging to
/home/ey-pc/.ros/log/e4944c60-025e-11eb-9079-40ec993efb48/roslaunch-ey-pc-23774
.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://ey-pc:37667/

SUMMARY
========

PARAMETERS
 * /rosdistro: melodic
 * /rosversion: 1.14.7
 * /webpage_launch/cmd: /home/ey-pc/eyrc...

NODES
  /
    node_hello_ros (pkg_ros_basics/node_hello_ros.py)
    webpage_launch (pkg_ros_basics/webpage_launch.sh)

ROS_MASTER_URI=http://localhost:11311
```

```
process[webpage_launch-1]: started with pid [23790]
** Opening https://www.e-yantra.org/ in Firefox **
process[node_hello_ros-2]: started with pid [23793]
[webpage_launch-1] process has finished cleanly
log file:
/home/ey-pc/.ros/log/e4944c60-025e-11eb-9079-40ec993efb48/webpage_launch-1*.log
[INFO] [1601399716.259020]: Hello World!
```

- The code is self-explanatory.

ROS Communication

- In ROS there are essentially three ways in which two nodes can communicate with each other. These are,
    1. ROS Topics
    2. ROS Services
    3. ROS Actions
- We will learn about all these three in this section.

ROS Topics

- ROS Topics allow **unidirectional** communication between ROS Nodes.
- When using ROS Topics a ROS Node can be a **publisher**, **subscriber** or both.
- A ROS Node acting as a publisher can publish data on a ROS Topic and a subscriber ROS Node can subscribe to a ROS Topic.
- Publisher and Subscriber Nodes will exchange ROS Messages over a ROS Topic.
- A ROS Message is a simple data structure, comprising typed fields (integer, floating point, boolean, etc.). So a ROS Message can hold data of various data-types.
- Consider this analogy,
    - Let's say you are subscribed to a newspaper called *The Melodic* published by a publishing house called *OSRF*.
    - Every morning your paperboy *Jon Doe* will deliver this newspaper to you.
    - You like *The Melodic* because it has dedicated section on *sports* and *robotics* news.

- In this analogy you can think,
    - `OSRF <--> ROS Publisher Node`

      OSRF which is publishing the newspaper as a Publisher Node.
    - `You <--> ROS Subscriber Node`

      You along with your neighbours who are subscribed to this newspaper as Subscriber Nodes.
    - `Jon Doe <--> ROS Topic`

      Your paperboy who is taking the newspaper from the publisher and delivering it to its subscribers as a ROS Topic.
    - `The Melodic Newspaper <--> ROS Message`

      The physical newspaper is your ROS Message.
    - `Sports and Robotics Sections of The Melodic <--> Data Fields defined in ROS Message`

      The sections of the newspaper is the Data Fields defined in the ROS Message.

Reading Assignment

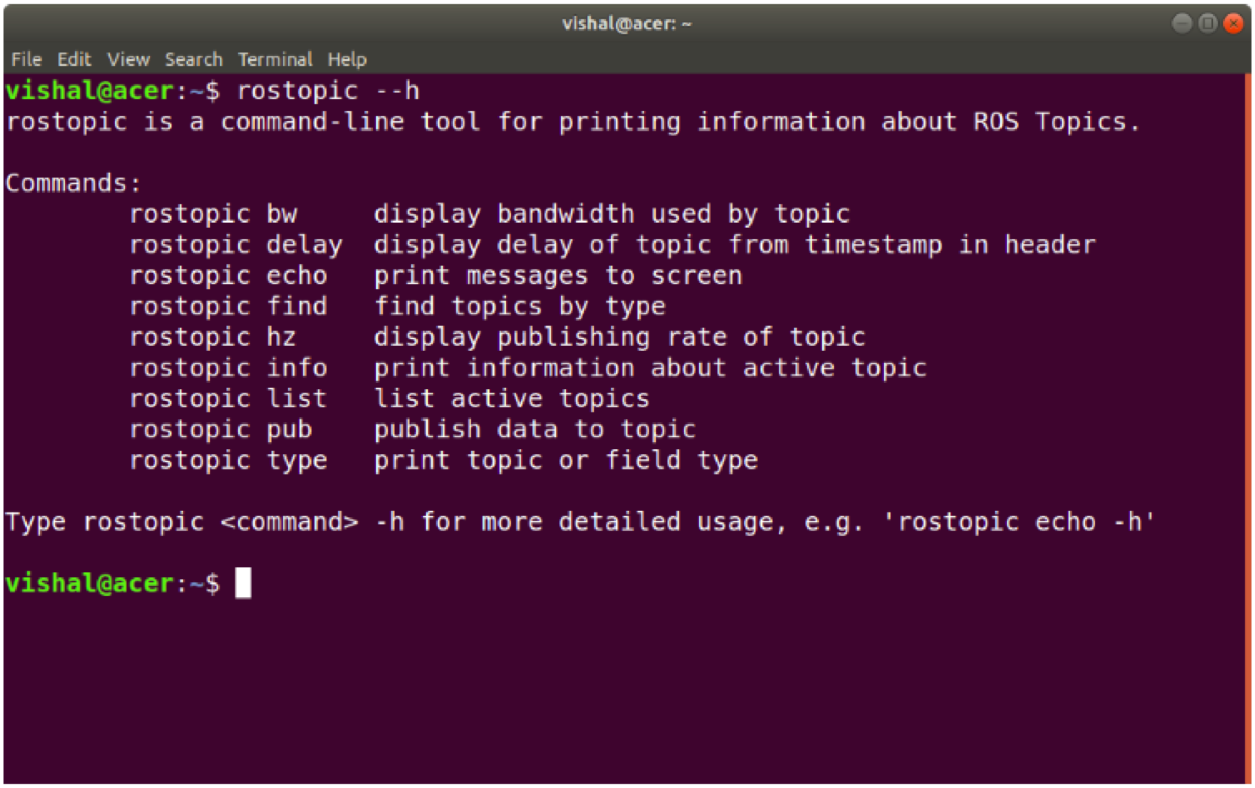
1. ROS Wiki - Topic
2. ROS Wiki - Messages

Command: rostopic

`rostopic` contains the rostopic command-line tool for displaying debug information about ROS Topics, including publishers, subscribers, publishing rate, and ROS Messages."_

Reference: http://wiki.ros.org/rostopic

Note: For quick information about any command, be that outside of ROS, simply type the command along with suffix `--h` or `-help`. This is a widely used concept among other Linux commands for quick referencing. Here's an example for `rostopic --h` command



```
vishal@acer:~$ rostopic --h
rostopic is a command-line tool for printing information about ROS Topics.

Commands:
        rostopic bw     display bandwidth used by topic
        rostopic delay  display delay of topic from timestamp in header
        rostopic echo   print messages to screen
        rostopic find   find topics by type
        rostopic hz     display publishing rate of topic
        rostopic info   print information about active topic
        rostopic list   list active topics
        rostopic pub    publish data to topic
        rostopic type   print topic or field type

Type rostopic <command> -h for more detailed usage, e.g. 'rostopic echo -h'

vishal@acer:~$ 
```

As you can see in the above image, there are multiple suffixes associated with `rostopic`, these suffixes are the commands to analyze any existing or developing system. We'll look into this sub-command or suffixes one by one.

Optional read, http://ros.informatik.uni-freiburg.de/roswiki/rostopic.html

ROS Services

- The publish/subscribe model is a very flexible communication paradigm, but its many-to-many one-way transport is not appropriate for 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.
- Like topics, services have an associated service type that is the package resource name of the `.srv` file.

Reading Assignment

1. ROS Wiki - Services
2. AGITR - Services

ROS Actions

- As you know ROS Services provide Client-Server request-response type architecture.
- So, ROS Services are Synchronous i.e the Client would wait for the Server for its response on the request sent by the client.
- This kind of behaviour is useful if you want to do something quickly and don't want to wait for the server to complete processing.
    - For eg. You can have a Server which can activate and deactivate vacuum grippers attached to a robotic arm. You can then have a client which would send activation or deactivation request using ROS Services to the Server and once request has been processed the server will send back the response.
- Now, let's say there is a case where the Client,
    - Does not want to wait for the server to complete the request.
    - Wants to get periodic feedback on progress of the request as it is being processed.
    - Wants to cancel the request in-between.
- In this case, ROS Actions are more appropriate than ROS Services.
- In ROS Actions,
    - Client can send multiple **goals** to the Server. (Like Requests in ROS Services)
    - Client can **cancel** any Goal or all the Goals anytime.
    - Client can get **feedback** and **status** of the Goal while it is being processed.

  - Client won't have to wait for the result from the server as processing will happen **asynchronously** at server. So, client can work on other things while the server is processing the goal.

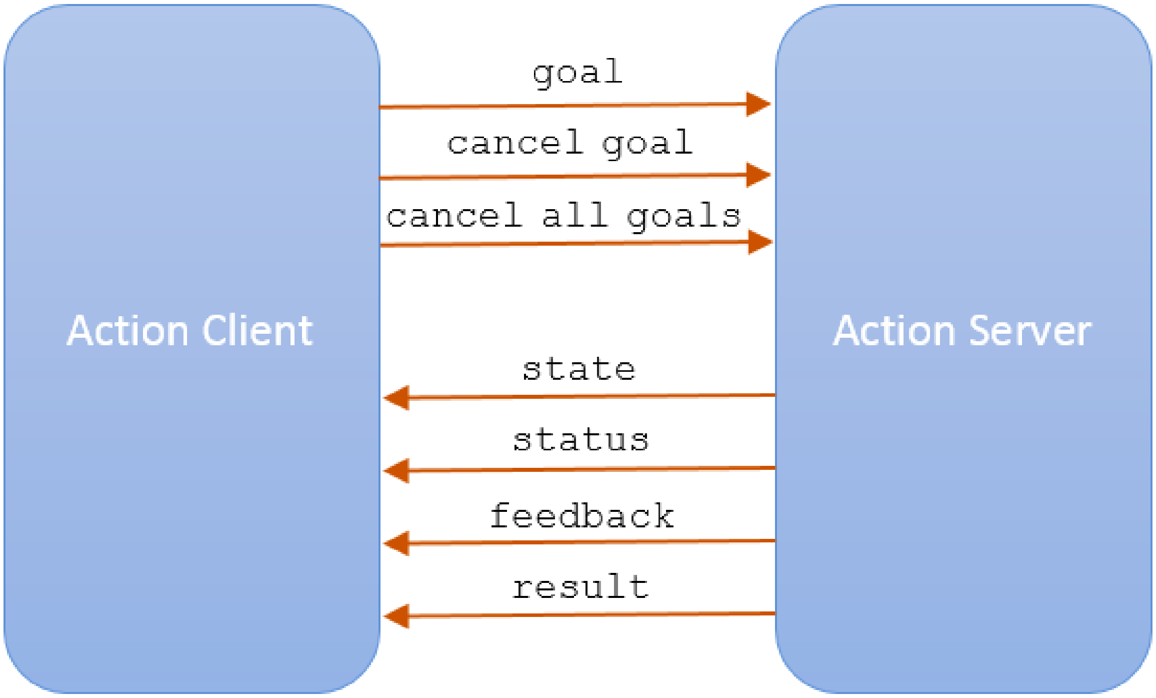


Image by Mathwork

Usage

- To use ROS Actions you would have to use `actionlib` library provided by ROS in your ROS Nodes.
- The `actionlib` library provides following classes,
  - `ActionServer` and `ActionClient` : These two classes are used to make your ROS Nodes Action Server and Action Client respectively.

    - `SimpleActionServer` and `SimpleActionClient`: These two classes provides simple interface for the users to use ROS Actions. Some of the features of `ActionServer` and `ActionClient`are missing in this.

Reading Assignment

1. ROS Robot Programming Book (available in Books Section) - Page: 172 - Section 7.4 Writing and Running the Action Server and Client Node
2. ROS Wiki - actionlib

Robotics Simulation Overview

This section is just to quick overview of the simulation and visualization tools in ROS.

NOTE: This is only to give **quick overview** of what these terms are. There is a lot to explore and learn in each of the following sub titles, and we strongly recommend you to explore these further as you do the tasks.

Rviz

- Visualizing sensor information is an important part in developing and debugging controllers.
- Rviz is a powerful 3D visualization tool in ROS that will hep you do exactly that.
- It allows the user to view the simulated robot model, log sensor information from the robot's sensors, and replay the logged sensor information.

Reference

1. ROS Wiki: Rviz
2. Gazebo: Visualization and logging

Gazebo

- Robot simulation is an essential tool in every roboticist's toolbox.
- A robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces, makes Gazebo a top Choise for 3D Simulator.

.World File: The file used to describe a collection of objects (such as buildings, tables, and lights), and global parameters including the sky, ambient light, and physics properties.

Reference

1. Gazebo tutorials

URDF

- The Unified Robot Description Format (URDF) contains a number of XML specifications for robot models, sensors, scenes, etc.
- It describes the position of all the joints, sensors, type of joints, structure of the robot base, arm etc.

Reference

1. ROS Wiki: URDF overview
2. ROS Wiki: URDF Tutorials

XACRO

- Xacro (XML Macros) Xacro is an XML macro language.
- With xacro, you can construct shorter and more readable XML files by using macros that expand to larger XML expressions.
- Xacro is useful when the structure of the robot is complex so instead of describing the whole structure in an urdf we can divide the structure in small parts and call those macro files in the main xacro file.
- Xacros also make it easier to define common structures. For example, let's say the robot has 2 wheels, we just need to make macros of a cylindrical structure(wheels), call it in the main xacro file and then define 2 different joints using the same structure but giving different joint location.

Reference

1. ROS Wiki: Using Xacro to Clean Up a URDF File
2. ROS Wiki: Xacro overview

ROS & Gazebo

- ROS and Gazebo together are a great combination to simulate how your algoirthm would work in real time scenarios.

Transmission Tags

- Transmission tags are used to link actuators to joints.

- If the transmission tags the joints won't move in Gazebo and they will be considered as stationary objects.
- We need to define transmission for every dynamic(moving) joint.

Gazebo Plugins

- In addition to the transmission tags, a Gazebo plugin needs to be added to your URDF that actually parses the transmission tags and loads the appropriate hardware interfaces and controller manager.
- Plugins basically replicate exact architecture of the sensors in use or the control system used to control the movement of the robot.

Reference

1. Gazebo tutorials: ROS Control

Example of call backs

```
def laser_callback(msg):
    global regions
    regions = {
        'bright':    ,
        'fright':    ,
        'front':     ,
        'fleft':     ,
        'bleft':     ,
    }
```

```
def odom_callback(data):
    global pose
    x  = data.pose.pose.orientation.x;
    y  = data.pose.pose.orientation.y;
    z = data.pose.pose.orientation.z;
    w = data.pose.pose.orientation.w;
    pose = [data.pose.pose.position.x, data.pose.pose.position.y,
euler_from_quaternion([x,y,z,w])[2]]
```

http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29