Robot Operating System

Using ROS parameter Server

Lecture No. 10

Dr. Eng. Essa Alghannam

1 Introduction:

1.1 rosparam:

rosparam allows to store and manipulate data on the ROS Parameter Server. The Parameter Server can store integers, floats, boolean, dictionaries, and lists. rosparam uses the YAML markup language for syntax.

In simple cases, YAML looks very natural: 1 is an integer, 1.0 is a float, one is a string, true is a boolean, [1, 2, 3] is a list of integers, and {a: b, c: d} is a dictionary.

1.2 ROS Parameter Server:

The ROS Parameter Server is a fundamental component of the Robot Operating System (ROS) that provides a centralized, shared, and dynamic storage system for parameters. It is a global dictionary or a shared blackboard where all ROS nodes can store and retrieve configuration data. The Parameter Server gives us the possibility to store data in a central location.

Key Characteristics and Concepts:

1- Centralized and Shared: All ROS nodes connected to the same ROS Master can access the Parameter Server. This means a parameter set by one node or a launch file can be read by any other node.
2- Dynamic: Parameters can be set, retrieved, and modified at runtime (while nodes are running) without needing to recompile or restart nodes. This allows for flexible configuration and dynamic adjustment of robot behavior or algorithm tuning.

3- Hierarchical (Namespaces): Parameters are organized in a tree-like structure, similar to a file system. They have namespaced paths (e.g., /robot_description/wheel_radius, /camera/image_topic, /planner/costmap/resolution). This helps prevent naming conflicts and organizes related parameters.
4- Key-Value Store: Each parameter consists of a unique key (its name/path) and an associated value. Values can be different data types: integers, floats, Booleans, strings, lists, or dictionaries (YAML structures). The Parameter Server uses XMLRPC data types for parameter values, which include the following:
   - 32-bit integers
   - Booleans
   - Strings
   - Doubles
   - Lists
   - Base 64-encoded binary data
   - ISO 8601 dates
5- **Persistence** (during roscore uptime): Once a parameter is set, its value persists on the server until: 1- It's explicitly deleted. 2- Its value is explicitly changed. 3- The roscore process is stopped.

The Parameter Server is crucial for:

1- **Configuration Management:** Instead of hardcoding values (like robot dimensions, sensor offsets, algorithm constants) directly into the C++ or Python code, we can store them on the Parameter Server. This makes the code more **generic** and **reusable**.
2- **Flexibility and Tunability:** we can easily modify how the robot behaves or how an algorithm operates simply by changing parameter values, without needing to recompile code. This is invaluable during development, testing, and even for adaptive behavior in deployed systems.
3- **Sharing Information:** Common values that multiple nodes need (e.g., the base frame name, a global speed limit) can be stored once and accessed by all relevant nodes.
4- **Debugging and Experimentation:** we can inspect current parameter values using command-line tools (rosparam get). We can temporarily change a parameter value (rosparam set) to test its effect on a running system.
5- **Simplified Launching:** ROS launch files can load an entire set of parameters from a YAML file onto the server, making it easy to set up complex configurations at startup.

1.3 Interact with the Parameter Server:

1. **Command Line (rosparam):** rosparam has many commands that can be used on parameters, as shown below:
   - rosparam list: Lists all currently set parameters in the server.
   - rosparam get <param_name>: Retrieves or gets the value of a specific parameter.
   - rosparam set <param_name> <value>: Sets or updates a parameter's value.

- rosparam delete <param_name>: Removes a parameter.
- rosparam dump <file.yaml> [namespace]: Saves (dumps) current parameters to a YAML file.
- rosparam load <file.yaml> [namespace]: Loads parameters from a YAML file on the Parameter Server.

2. **ROS Launch Files (<param> and <rosparam> tags):**
   - This is the most common way to load parameters when starting the ROS system.
   - **<param name="my_param" value="123" />:** Sets a single parameter.
   - **<rosparam command="load" file="path/to/my_params.yaml" />:** Loads multiple parameters from a YAML file.
   - Parameters can be loaded into the **global namespace** (default, /my_param) or a **node's private namespace** (/my_node/my_param by placing <rosparam> inside a <node> tag).
3. **ROS Client Libraries (e.g., rospy for Python, roscpp for C++):**

Nodes can programmatically get and set parameters. For example, in **Python (rospy):**

- rospy.get_param('param_name', default_value): Reads a parameter.
  Use ~param_name for private node parameters.
- rospy.set_param('param_name', new_value): Sets a parameter.
- rospy.has_param('param_name'): Checks if a parameter exists.
- rospy.delete_param('param_name'): Deletes a parameter.

1.4 rosparam list:

In terminal 1, try **roscore** command and then in terminal 2 try **rosparam list**, the output will be:

- /rosdistro
- /roslaunch/uris/host_essa__38837
- /rosversion
- /run_id

In terminal 2, try **rosrun turtlesim turtlesim_node**; then, in terminal 3, try **rosparam list** again. The output will be:

- /rosdistro
- /roslaunch/uris/host_essa__38837
- /rosversion
- /run_id
- /turtlesim/background_b
- /turtlesim/background_g
- /turtlesim/background_r

In a terminal 3: **essa@essa:~$ rosparam list /turtlesim.** The output will be only the parameters with the **namespace** called **/turtlesim**:

- /turtlesim/background_b
- /turtlesim/background_g
- /turtlesim/background_r

1.5 rosparam get and rosparam set commands:

```
essa@essa:~$ rosparam get /turtlesim/background_b
```

output: 255

```
essa@essa:~$ rosparam get /turtlesim/background_r
```

output: 69

```
essa@essa:~$ rosparam get /turtlesim/background_g
```

output: 86

After that, try:

```
essa@essa:~$ rosparam set /turtlesim/background_g 255
```

This changes the parameter value.

```
essa@essa:~$ rosparam get /turtlesim/background_g
```

output: 255

Now we have to call the clear service for the parameter change to take effect:

```
essa@essa:~$ rosservice call /clear
```

We can also use **rosparam get /** to show us the contents of the entire Parameter Server.

```
essa@essa:~$ rosparam get /
```

```
rosdistro: 'noetic

 '
roslaunch:
  uris:
    host_essa__38837: http://essa:38837/
rosversion: '1.17.0

 '
run_id: b1cdb880-4380-11f0-9238-4338e491ddf1
turtlesim:
  background_b: 255
  background_g: 255
  background_r: 69
```

1.6 Delete All Parameters (Clearing the Entire Server):

```
essa@essa:~$ rosparam delete /
```

CAUTION: Executing **rosparam delete /** will remove ALL parameters from the server, including any default ROS parameters (like rosversion, rosdistro, run_id) that are automatically set by roscore or roslaunch. While these core parameters are usually recreated if roscore is still running and new nodes are launched. Only use this if we truly want a completely blank slate for parameters.

Verification after deleting: we can use **rosparam list** or **rosparam get /** to see the current state of the parameter server and confirm if parameters have been removed.

1- # Example: Delete a parameter named '/my_global_param'

```
rosparam delete /my_global_param
```

2- # Example: Delete a parameter in a node's private namespace (e.g., if we had /my_node/param_a)

```
rosparam delete /my_node/param_a
```

3- # Example: Delete all parameters under '/my_node'

```
rosparam delete /my_node
```

4- # Example: Delete all parameters under '/sensors/camera'

```
rosparam delete /sensors/camera
```

1.7 rosparam dump:

When we run **rosparam dump params.yaml**, it connects to the currently running ROS Master (which hosts the Parameter Server) and retrieves the values of **all parameters** currently stored on the server. It then writes these parameters, along with their hierarchical structure, into the specified YAML file (params.yaml in this case).

1. **Snapshotting/Backing Up Parameters:** we can capture the current state of our system's parameters at any given moment. This is invaluable for debugging, especially if we've manually tweaked parameters using rosparam set during development. It creates a backup of the configuration.
2. **Inspection and Documentation:** It's often easier to review the full set of active parameters in a structured YAML file than using rosparam get / repeatedly or scrolling through the output of rosparam list. The dumped file serves as an implicit form of documentation for the system's configuration.
3. **Portability and Reusability:** If we've finely tuned parameters manually in a running system, we can dump them to a file. This file can then be used later with rosparam load or directly included in a roslaunch file to quickly restore that specific configuration. This is crucial for replicating experimental setups.

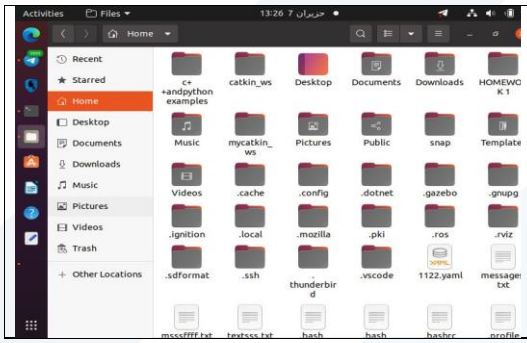
4. **Debugging Parameter Loading Issues:** If we're having trouble with a roslaunch file not loading parameters as expected, dumping the parameters after launching can show us exactly what values ended up on the server, helping us pinpoint errors in the launch file or YAML syntax.

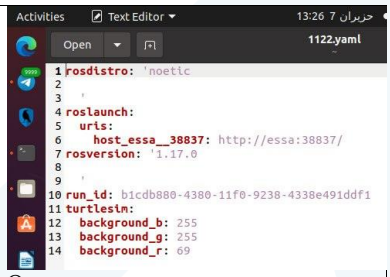
In terminal 1: essa@essa:~$ roscore

In terminal 2: essa@essa:~$ rosrun turtlesim turtlesim_node

In terminal 3: essa@essa:~$ rosparam set /turtlesim/background_g 255

In terminal 4: essa@essa:~$ rosparam dump 1122.yaml

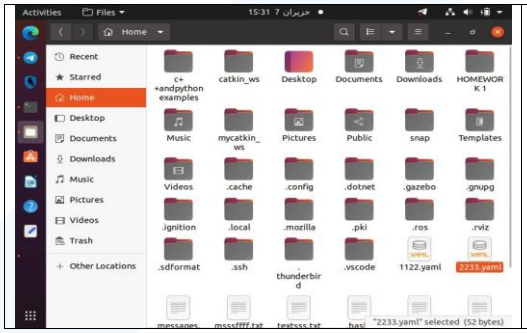


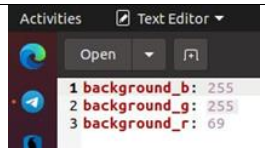


Or use
cat 1122.yaml

We can also specify a namespace to dump only parameters within that scope, for example:

In terminal 5: essa@essa:~$ **rosparam dump 2233.yaml /turtlesim**





Or use
cat 2233.yaml

2 Example:

Suppose that we have all_params.yaml file and it contains:

```
/publisher_node:
  message_text: "Hello from param file!"
```

```
  publish_rate: 2.0
/rosdistro: "noetic"
/roslaunch:
  uris:
    host_example_param_pkg__publisher_node__63768: "http://the_machine_ip:the_port/"
    host_example_param_pkg__subscriber_node__63768:
"http://the_machine_ip:the_port/"
/rosversion: "1.15.14" # ROS version
/run_id: "run_id_string"
```

Notice that publish_rate and message_text are under /publisher_node. they should be loaded into that **node's private namespace** using ~ in **rospy.get_param('~publish_rate')** and the **rosparam command="load"** being **inside** the **<node> tag** in the **launch file**.

2.1 rosparam load to load parameters from a YAML file onto the ROS Parameter Server, and use them:

There are two primary ways to load parameters from a YAML file onto the ROS Parameter Server:

Suppose that we have yaml file and it looks like below:

~/catkin_ws/src/example_param_pkg/params/config.yaml
/parameter_name_1: 10 /parameter_name_2: 'D'

2.1.1 Method A: Using rosparam load and get from the command line:

This is useful for quick tests, setting up parameters manually, or loading global configurations before launching specific nodes.

Syntax: rosparam load <path_to_yaml_file> [namespace]

- <path_to_yaml_file>: The full path to the YAML file (e.g., ~/catkin_ws/src/example_param_pkg/params/config.yaml).
- [namespace]: **(Optional)** If provided, all parameters from the YAML file will be loaded under this namespace on the Parameter Server. If omitted, they are loaded into the global (/) namespace.

2.1.1.1 Example 1:

First, ensure roscore is running

```
roscore
```

Load config.yaml into the global namespace

```
rosparam load ~/catkin_ws/src/example_param_pkg/params/config.yaml

# Verify parameters are loaded (they'll appear under /)

rosparam list

# Expected output might include:

# /parameter_name_1

# /parameter_name_2
```

2.1.1.2 Example 2:

```
# Alternatively, load into a specific namespace (e.g., /my_settings)

rosparam load ~/catkin_ws/src/example_param_pkg/params/config.yaml   /my_settings

# Verify

rosparam list

# Expected output might include:

# /my_settings/parameter_name_1

# /my_settings/parameter_name_2
```

To get the parameter value from

```
$ rosparam get /my_settings/parameter_name_1
```

2.1.1.3 Example 3:

Suppose we have the file 1122.yaml

```
essa@essa:~$ cat 1122.yaml
```

```
rosdistro: 'noetic

  '
roslaunch:
  uris:
    host_essa__38837: http://essa:38837/
rosversion: '1.17.0

  '
run_id: b1cdb880-4380-11f0-9238-4338e491ddf1
turtlesim:
  background_b: 255
  background_g: 255
  background_r: 69
rosdistro: 'noetic
```

```
  '
roslaunch:
  uris:
    host_essa__38837: http://essa:38837/
rosversion: '1.17.0

  '
run_id: b1cdb880-4380-11f0-9238-4338e491ddf1
turtlesim:
  background_b: 255
  background_g: 255
  background_r: 69
```

In terminal 1: essa@essa:~$ roscore

In terminal 2: essa@essa:~$ rosparam load 1122.yaml copy

In terminal 2: essa@essa:~$ rosparam get /

Commented [EA1]: Load this YAML file into new namespace, e.g., copy in the Parameter Server

Commented [EA2]: show us the contents of the entire Parameter Server

```
copy:
  rosdistro: 'noetic

    '
  roslaunch:
    uris:
      host_essa__38837: http://essa:38837/
  rosversion: '1.17.0

    '
  run_id: b1cdb880-4380-11f0-9238-4338e491ddf1
  turtlesim:
    background_b: 255
    background_g: 255
    background_r: 69
rosdistro: 'noetic

  '
roslaunch:
  uris:
    host_essa__37705: http://essa:37705/
rosversion: '1.17.0

  '
run_id: b1093710-444a-11f0-940b-83af93e8830b
```

In terminal 2: essa@essa:~$ rosparam get /copy/turtlesim/background_b

Output: 255

In terminal 2: essa@essa:~$ MY_VARIABLE=$(rosparam get /copy/turtlesim/background_r)

In terminal 2: essa@essa:~$ echo $MY_VARIABLE

Output: 69

In terminal 2: essa@essa:~$ MY_VARIABLE=$(rosparam get /copy/turtlesim/)

In terminal 2: essa@essa:~$ echo $MY_VARIABLE

Output: background_b: 255 background_g: 255 background_r: 69

2.1.2 Method B: Using <rosparam> tag in a ROS Launch File and rospy.get_param() in python (Recommended)

This is the standard and most robust way to manage parameters in a ROS system, as it ties parameter loading directly to the system's startup.

2.1.2.1 Loading into the Global Namespace:

If we want the parameters to be accessible globally by any node (e.g., /publish_rate), we can place the <rosparam> tag directly under the <launch> tag:

```
<?xml version="1.0" encoding="UTF-8"?>


<!-- File: ~/catkin_ws/src/example_param_pkg/launch/global_params.launch -->
<launch>
  <rosparam command="load" file="$(find example_param_pkg)/params/config.yaml" />


  <!-- Now, if we launch a node, it can access /publish_rate and /message_text -->
  <node pkg="example_param_pkg" type="publisher_node.py" name="publisher_node"
output="screen" />
  <node pkg="example_param_pkg" type="subscriber_node.py" name="subscriber_node"
output="screen" />


</launch>
```

2.1.2.2 Loading into a Node's Private Namespace (Most Common for Node-Specific Params):

This is typically preferred for parameters that are specific to a single node. The parameters will be loaded under the node's name (e.g., /publisher_node/publish_rate). This prevents naming conflicts if multiple nodes use similar parameter names.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- File: ~/catkin_ws/src/example_param_pkg/launch/node_private_params.launch -->
<launch>
  <!--
  The <rosparam> tag inside the <node> tag loads parameters
  into that specific node's private namespace.
  If the node name is "publisher_node", then parameters like "publish_rate"
  from config.yaml will be loaded into the parameter server under "publisher_node"
namespace, and it will be accessible as /publisher_node/publish_rate.
  -->
  <node pkg="example_param_pkg" type="publisher_node.py" name="publisher_node"
output="screen">
    <rosparam command="load" file="$(find example_param_pkg)/params/config.yaml" />
  </node>


  <node pkg="example_param_pkg" type="subscriber_node.py" name="subscriber_node"
output="screen" />


</launch>
```

Key point about $(find package_name): This is a roslaunch substitution argument that automatically resolves to the path of the specified ROS package.

2.1.2.3 Read Specific Parameters from the Parameter Server using python:

Once parameters are loaded onto the ROS Parameter Server (either via rosparam load or roslaunch), we can read them within the ROS nodes using the client library (e.g., rospy for Python, roscpp for C++). To do it in **Python (rospy)** we need to use a primary tool or function **rospy.get_param()**.

a. Reading a Global Parameter:

If `publish_rate` was loaded into the global namespace (e.g., `/publish_rate`):

```
import rospy


# Initialize the node (essential before using parameters)
rospy.init_node('my_reader_node')
```

```
# Read the global parameter
# Full path starting with '/'
global_rate = rospy.get_param('/publish_rate')
rospy.loginfo(f"Global publish rate: {global_rate}")
```

b. Reading a Private Parameter (Recommended for Node-Specific Params):

If `publish_rate` was loaded into the current node's private namespace (e.g., `/publisher_node/publish_rate` for a node named `publisher_node`):

```
import rospy

# Initialize the node
rospy.init_node('publisher_node') # Node name should match the one in the launch file

# Read the private parameter using '~' (tilde) prefix
# The '~' refers to the node's private namespace
private_rate = rospy.get_param('~publish_rate')
rospy.loginfo(f"Private publish rate: {private_rate}")
```

c. Reading with a Default Value:

It's highly recommended to provide a default value. If the parameter is not found on the server, `rospy.get_param()` will return the default value instead of raising a `KeyError`.

```
import rospy

rospy.init_node('my_node')

# Try to get 'non_existent_param', if not found, use 5.0
param_with_default = rospy.get_param('~non_existent_param', 5.0)
rospy.loginfo(f"Parameter with default: {param_with_default}")

# Example from the config:
publish_rate = rospy.get_param('~publish_rate', 1.0) # Will use 2.0 if found, else 1.0
message_text = rospy.get_param('~message_text', "Default fallback message") # Will use "Hello..." if found
```

d. Checking if a Parameter Exists:

We can use `rospy.has_param()` to check for a parameter's existence before trying to retrieve it.

```
import rospy

rospy.init_node('my_node')

if rospy.has_param('/some_global_param'):
    value = rospy.get_param('/some_global_param')
    rospy.loginfo(f"Global param exists: {value}")
else:
    rospy.logwarn("Global param does not exist.")

if rospy.has_param('~some_private_param'):
    value = rospy.get_param('~some_private_param')
    rospy.loginfo(f"Private param exists: {value}")
else:
    rospy.logwarn("Private param does not exist.")
```

3 Example 0:

In example_param_pkg, the launch/param_example.launch file uses:

```
<?xml version="1.0" encoding="UTF-8"?>
<node pkg="example_param_pkg" type="publisher_node.py" name="publisher_node" output="screen">
    <rosparam command="load" file="$(find example_param_pkg)/params/config.yaml" />
  </node>
```

This means publish_rate and message_text from config.yaml are loaded into the **private namespace of the publisher_node**. So, they will be accessible on the parameter server as:

- /publisher_node/publish_rate
- /publisher_node/message_text

And in src/publisher_node.py, we correctly read them using the tilde (~):

```
    publish_rate = rospy.get_param('~publish_rate', 1.0) # Reads from /publisher_node/publish_rate
    message_text = rospy.get_param('~message_text', "Default message from node if param not found!") # Reads from /publisher_node/message_text
```

4 Example 1:

4.1 Package and files structure:

example_param_pkg/

- launch/
  - ✓ example.launch
- params/
  - ✓ config.param
- src/
  - ✓ publisher_node.py
  - ✓ subscriber_node.py
- CMakeLists.txt
- package.xml

- mkdir -p ~/ mycatkin_ws /src
- cd ~/ mycatkin_ws /src
- catkin_create_pkg example_param_pkg rospy roscpp std_msgs
- cd ..
- catkin_make

4.2 Create the .param file with parameters:

Use gedit to create the config.param in the path: ~/mycatkin_ws/src/example_param_pkg/params/config.param (YAML format)

```
publish_rate: 2.0
message_text: "Hello from param file!"
```

4.3 Create the publisher node:

Use gedit to create publisher_node.py in the path ~/mycatkin_ws/src/example_param_pkg/src/publisher_node.py

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

import rospy
from std_msgs.msg import String

def publisher():
    rospy.init_node('param_publisher_node')
    pub = rospy.Publisher('chatter', String, queue_size=10)
```

```
    # Load parameters from parameter server, provide default values just in case
    publish_rate = rospy.get_param('~publish_rate', 1.0)
"""
used to access parameters stored in the ROS parameter server.
the parameter server is a central repository for configuration settings in ROS.
This means the function will look for a parameter named publish_rate within the node's private namespace. If the node is named 'param_publisher_node', the actual parameter looked up on the server is /param_publisher_node/publish_rate. Using the private namespace is generally the best practice to avoid naming collisions with parameters from other nodes. If we omit the ~, it looks for a parameter globally.
1.0: This is the second argument to rospy.get_param(). It's the default value. If a parameter named ~publish_rate is not found in the node's private namespace on the parameter server, the function will return this default value (1.0 in this case).
"""

    message_text = rospy.get_param('~message_text', 'Hello Default')
    rate = rospy.Rate(publish_rate)
    rospy.loginfo(f"[Publisher] Publishing at {publish_rate} Hz: '{message_text}'")

    while not rospy.is_shutdown():
        pub.publish(message_text)
        rate.sleep()

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

```
chmod +x src/publisher_node.py
```

4.4 Create the subscriber node:

Use gedit to create subscriber_node.py in the path ~/mycatkin_ws/src/example_param_pkg/src/subscriber_node.py

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

import rospy
from std_msgs.msg import String

def callback(msg):
    rospy.loginfo(f"[Subscriber] Received: {msg.data}")

def subscriber():
    rospy.init_node('param_subscriber_node')
    rospy.Subscriber('chatter', String, callback)
    rospy.spin()
```

```
if __name__ == '__main__':
    subscriber()
```

```
chmod +x src/subscriber_node.py
```

4.5 Create a launch file:

Use gedit to create example.launch in the path ~/mycatkin_ws/src/example_param_pkg/launch/**example.launch**

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>

    <node pkg="example_param_pkg" type="publisher_node.py"
name="param_publisher_node" output="screen">
    <rosparam file="$(find example_param_pkg)/params/config.param" command="load"/>
    </node>

<!--
  The <rosparam> tag inside the <node> tag loads parameters
  into that specific node's private namespace.
  If the node name is "param_publisher_node", then parameters like "publish_rate" from
config.yaml will be loaded into the parameter server under "publisher_node" namespace,
and it will be accessible as /param_publisher_node/publish_rate.
  -->

    <!-- Launch the subscriber node -->
    <node pkg="example_param_pkg" type="subscriber_node.py"
name="param_subscriber" output="screen" />

</launch>
```

Commented [EA3]: This is a ROS launch file macro. find is a command that searches for ROS packages. It looks for a package named example_param_pkg in the ROS workspace.

4.6 Update CMakeLists.txt and package.xml:

In package.xml: (not required)

<build_depend>message_generation</build_depend>

<exec_depend>message_runtime</exec_depend>

<exec_depend>rospy</exec_depend>

<exec_depend>std_msgs</exec_depend>

In CMakeLists.txt:

a) Ensure scripts are installed (optional but good practice):
catkin_install_python(PROGRAMS

src/publisher_node.py

src/subscriber_node.py

DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}

)

4.7 Build the workspace & source & run:

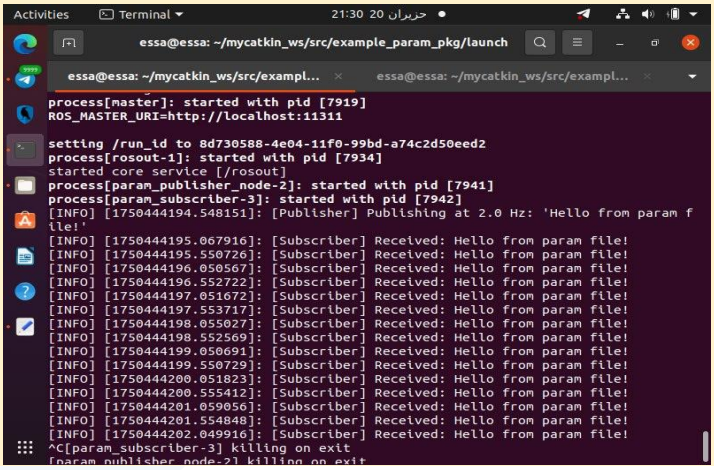
cd ~/mycatkin_ws

catkin_make

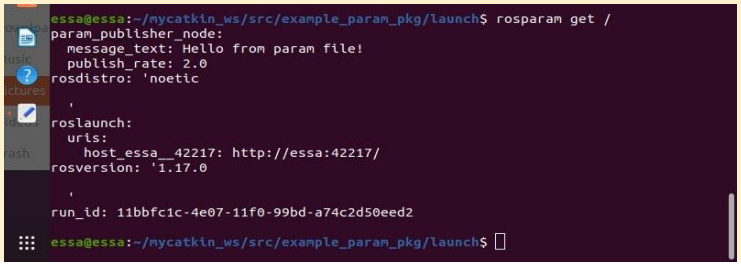
source ~/mycatkin_ws/devel/setup.bash

roslaunch example_param_pkg **example.launch**

Terminal Output:

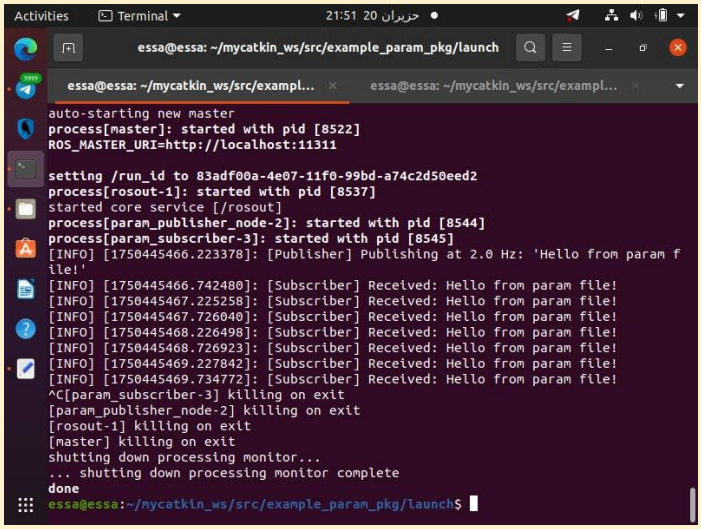


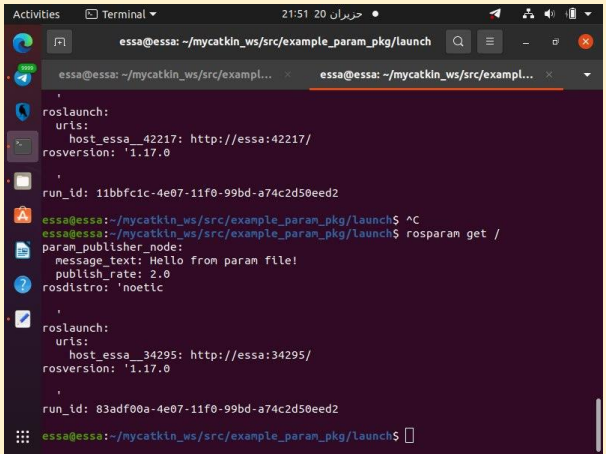
rosparam get /



roslaunch example_param_pkg example.launch publish_rate:=1 message_text:="88888888888hh"

It has no any effect. publish_rate remains 2.0khz and message_text remains “Hello from param file!". Same as param file.



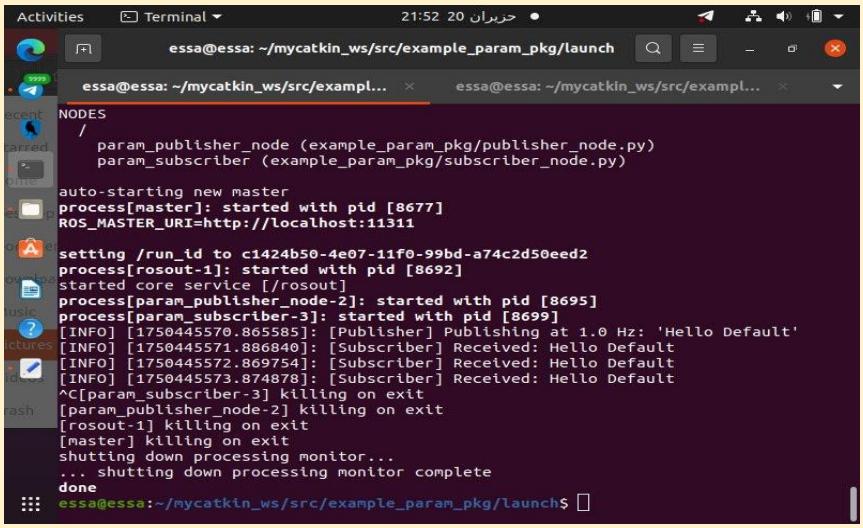


4.8 Modify Launch file:

1. Use gedit to create example1.launch in the path ~/mycatkin_ws/src/example_param_pkg/launch/**example4.launch**

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
  <arg name="publish_rate" default="55"/>
  <arg name="message_text" default="hhhhhhhhhh"/>
  <!-- Remap parameters into the global namespace as needed -->
   <param name="publish_rate" value="$(arg publish_rate)"/>
   <param name="message_text" value="$(arg message_text)"/>


    <node pkg="example_param_pkg" type="publisher_node.py"
name="param_publisher_node" output="screen">
   </node>
 <!-- Launch the subscriber node -->
  <node pkg="example_param_pkg" type="subscriber_node.py"
name="param_subscriber" output="screen" />
</launch>
```

roslaunch example_param_pkg **example4.launch**



Commented [EA4]: These lines of XML code define arguments for a ROS (Robot Operating System) node or launch file. When the launch file is executed, these arguments become variables that can be used within the launched node(s).

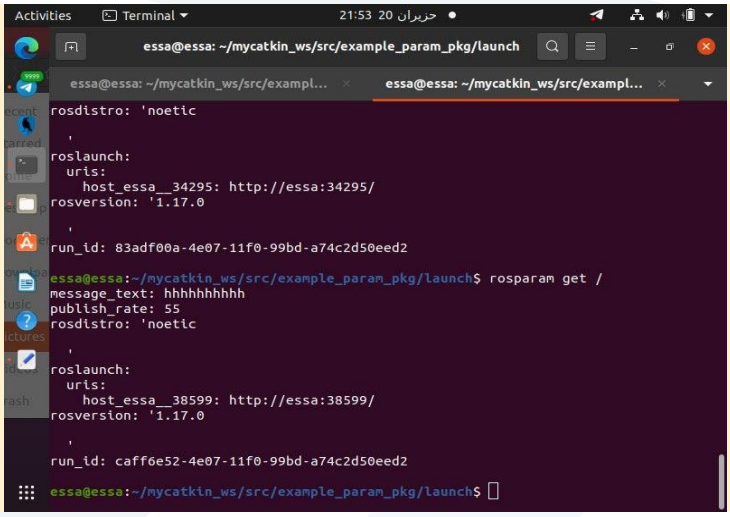
Commented [EA5]: Remap parameters into the **global namespace**

Commented [EA6]: publish_rate = rospy.get_param('~publish_rate', 1.0)
message_text = rospy.get_param('~message_text', 'Hello Default')

it gives the default values (the second args) to the parameters because the function cannot find them in a local namespace due to the use of ~.

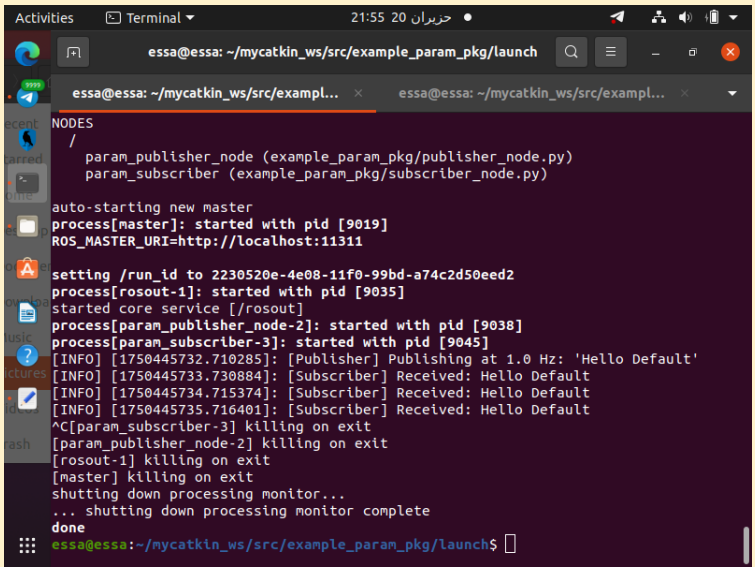
The parameters in the global namespace getsm its value from the launch file or command lines

rosparam get /



roslaunch example_param_pkg **example4.launch** publish_rate:=1 message_text:="88888888888hh"

Terminal Output:

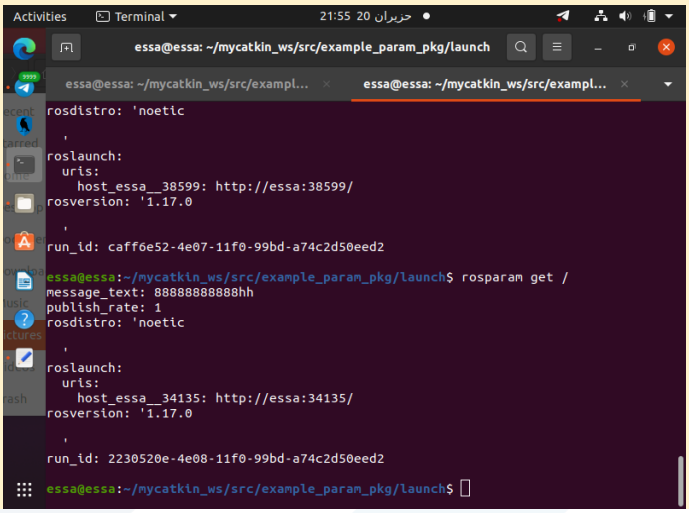


Commented [EA7]: publish_rate = rospy.get_param('~publish_rate', 1.0)
message_text = rospy.get_param('~message_text', 'Hello Default')

it gives the default values (the second args) to the parameters because the function cannot find them in a local namespace due to the use of ~.

The parameters in the global namespace getsm its value from the launch file or command lines

rosparam get /



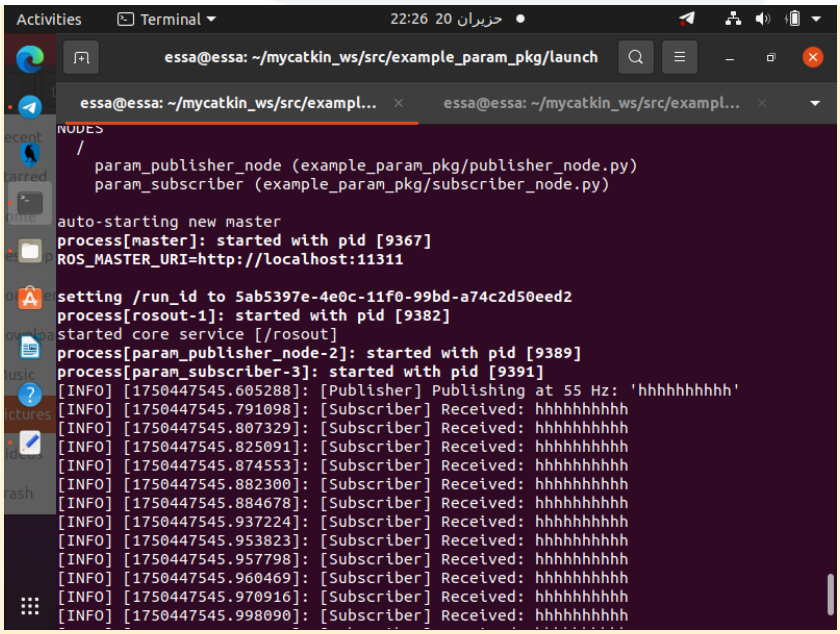
2. Use gedit to create example1.launch in the path ~/mycatkin_ws/src/example_param_pkg/launch/**example3.launch**

```
<?xml version="1.0" encoding="UTF-8"?>

<launch>
  <arg name="publish_rate" default="55"/>
  <arg name="message_text" default="hhhhhhhhhh"/>


  <node pkg="example_param_pkg" type="publisher_node.py" name="param_publisher_node" output="screen">
  <!-- Remap parameters into the private namespace as needed -->
    <param name="publish_rate" value="$(arg publish_rate)"/>
    <param name="message_text" value="$(arg message_text)"/>
  </node>


  <!-- Launch the subscriber node -->
  <node pkg="example_param_pkg" type="subscriber_node.py" name="param_subscriber" output="screen" />

</launch>
```

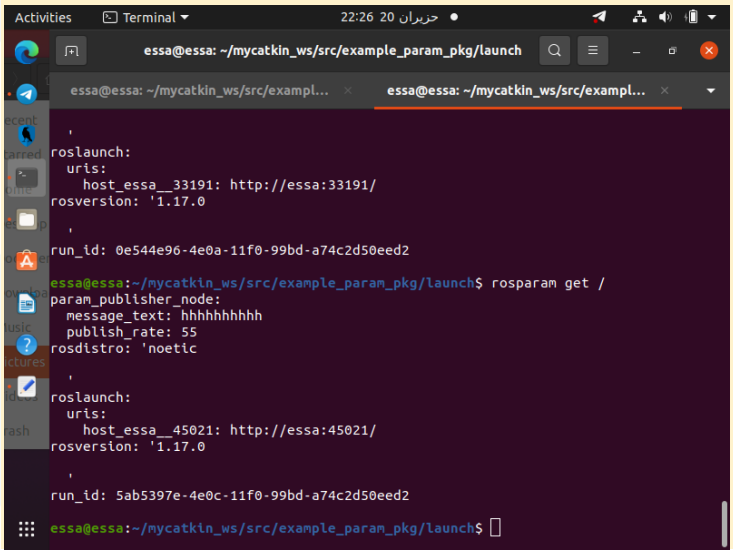
Commented [EA8]: These lines of XML code define arguments for a ROS (Robot Operating System) node or launch file.

Commented [EA9]: Remap parameters into the private namespace

roslaunch example_param_pkg **example3.launch**

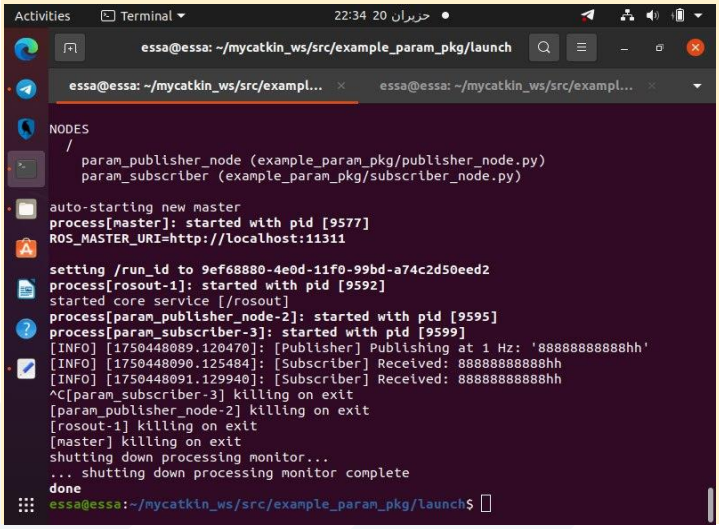


rosparam get /

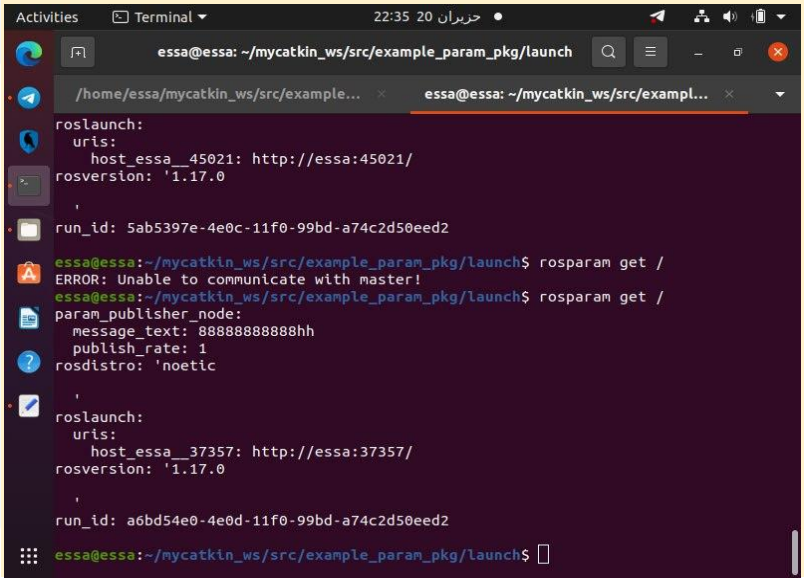


roslaunch example_param_pkg **example3.launch** publish_rate:=1 message_text:="88888888888hh"

Terminal Output:



rosparam get /



3. Use gedit to create example1.launch in the path ~/mycatkin_ws/src/example_param_pkg/launch/example2.launch

```
<?xml version="1.0" encoding="UTF-8"?>

<launch>
  <rosparam file="$(find example_param_pkg)/params/config.param" command="load"/>
    <!-- Load parameters globally from the .param file -->


  <arg name="publish_rate" default="55"/>
  <arg name="message_text" default="hhhhhhhhhh"/>



  <!-- Launch the publisher node -->
  <node pkg="example_param_pkg" type="publisher_node.py" name="param_publisher" output="screen">
    <!-- Remap parameters into the private namespace as needed -->
    <param name="publish_rate" value="$(arg publish_rate)"/>
    <param name="message_text" value="$(arg message_text)"/>
  </node>

  <!-- Launch the subscriber node -->
  <node pkg="example_param_pkg" type="subscriber_node.py" name="param_subscriber" output="screen" />
</launch>
```

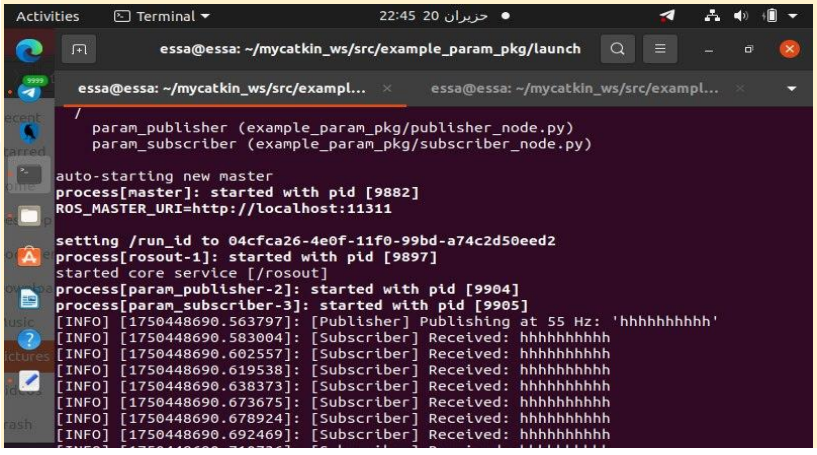
Commented [EA10]: Load globally to the parameter server

Commented [EA11]: Define parameters

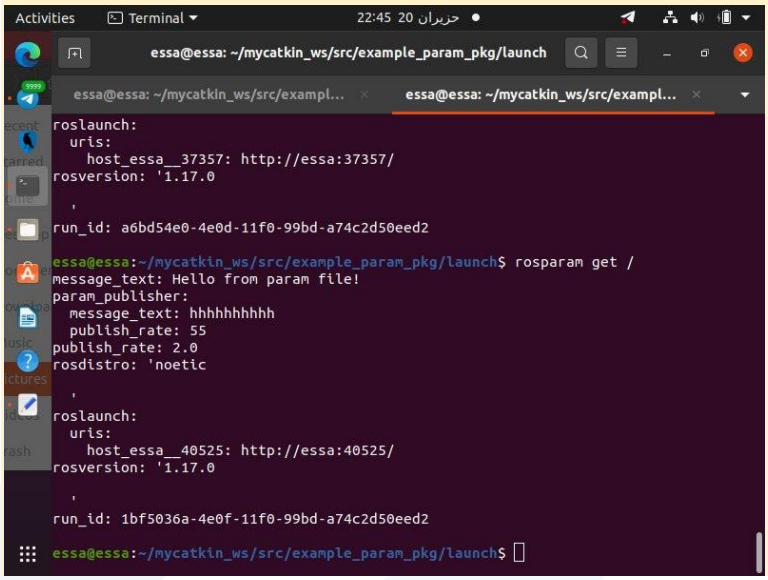
Commented [EA12]: Remap the defined parameters into the private namespace

```
roslaunch example_param_pkg example2.launch
```

Terminal Output:

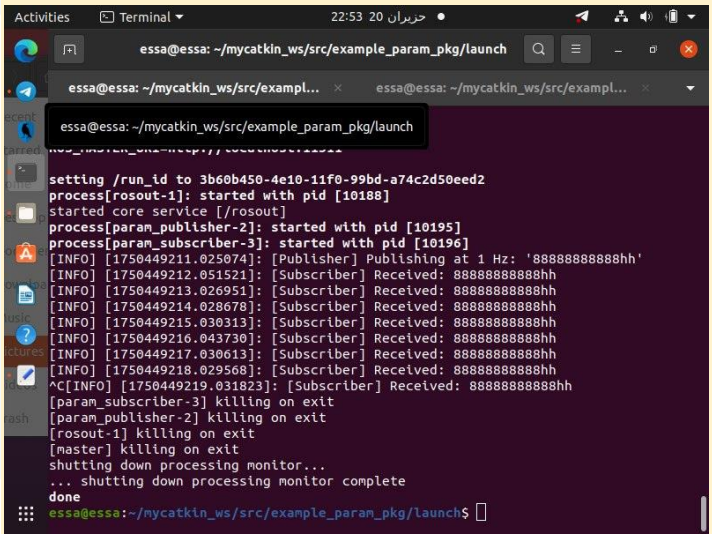


rosparam get /

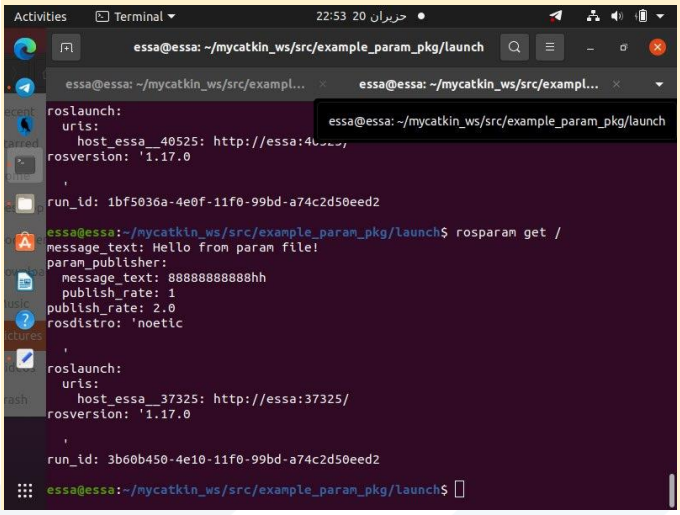


roslaunch example_param_pkg **example2.launch** publish_rate:=1 message_text:="88888888888hh"

Terminal Output:



rosparam get /



4. Use gedit to create example1.launch in the path ~/mycatkin_ws/src/example_param_pkg/launch/**example1.launch**

```
<?xml version="1.0" encoding="UTF-8"?>

<launch>
  <arg name="publish_rate" default="55"/>
  <arg name="message_text" default="hhhhhhhhhh"/>


  <node pkg="example_param_pkg" type="publisher_node.py" name="param_publisher_node" output="screen">
  <rosparam file="$(find example_param_pkg)/params/config.param" command="load"/>
  <!-- Remap parameters into the private namespace as needed -->
   <param name="publish_rate" value="$(arg publish_rate)"/>
   <param name="message_text" value="$(arg message_text)"/>
  </node>


  <!-- Launch the subscriber node -->
  <node pkg="example_param_pkg" type="subscriber_node.py" name="param_subscriber" output="screen" />

5. </launch>
```

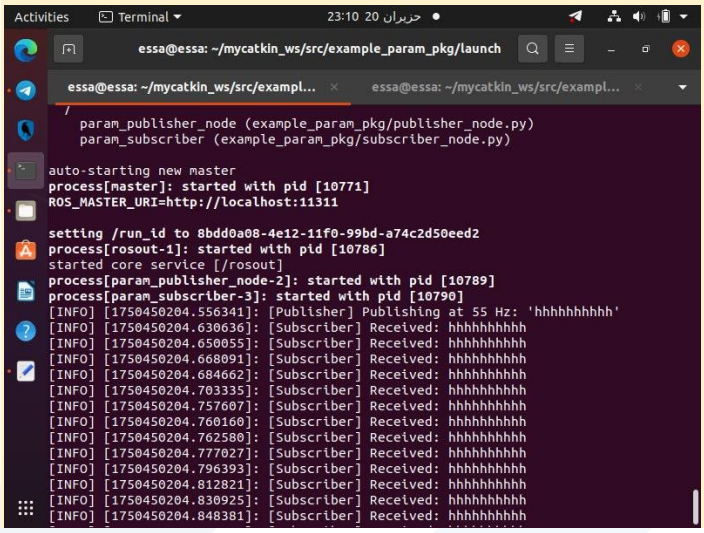
Commented [EA13]: Define parameters

Commented [EA14]: Load privatly to the parameter server

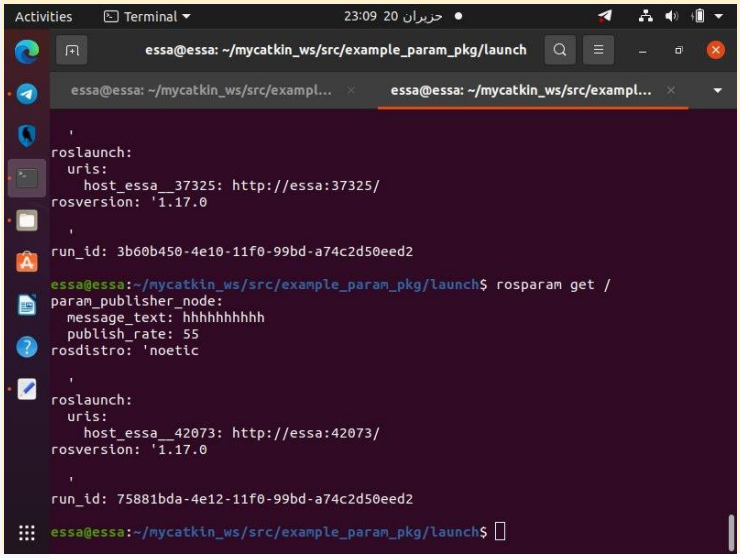
Commented [EA15]: Remap the defined parameters into the private namespace

roslaunch example_param_pkg example1.launch

Terminal Output:

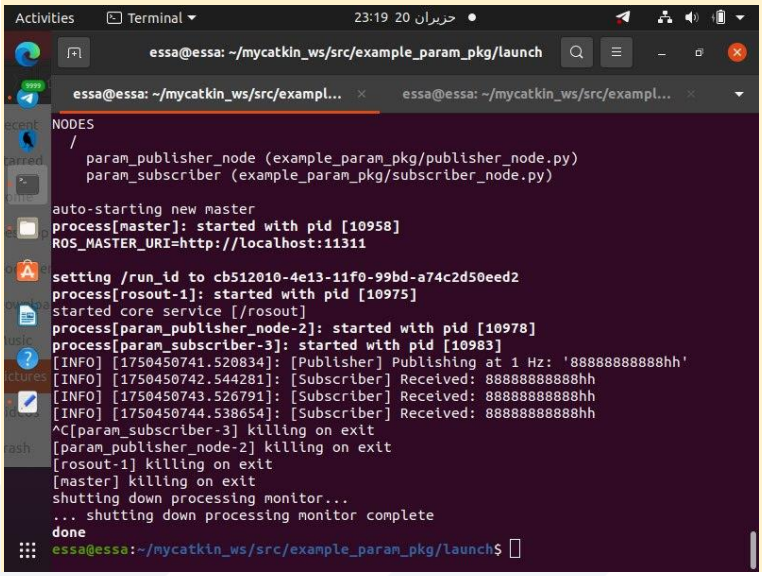


rosparam get /

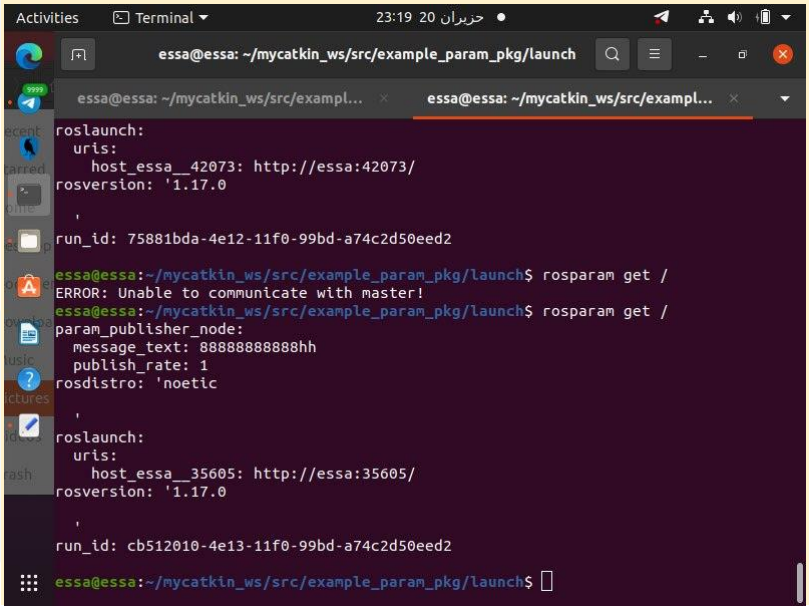


roslaunch example_param_pkg example1.launch publish_rate:=1 message_text:="88888888888hh"

Terminal Output:



rosparam get /



5 Changing parameter after node launch:

If we change the parameter value while the Python script and the launch file are running, the currently running script will not update its parameter value automatically.

rospy.get_param() reads the parameter only once on startup in previous examples.

To make it dynamic, we'd need to re-read the parameter inside the handle function, or in a loop, or use a dynamic reconfigure server (more advanced). For this example, it reads it only on node startup.

In a dynamic reconfigure server, it is possible to configure nodes while they're running or to change the working of the nodes.