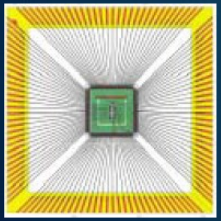
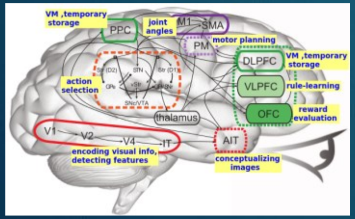
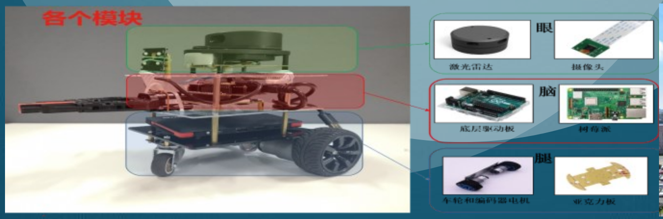
ROS Service, ROS Launch and ROS Parameter

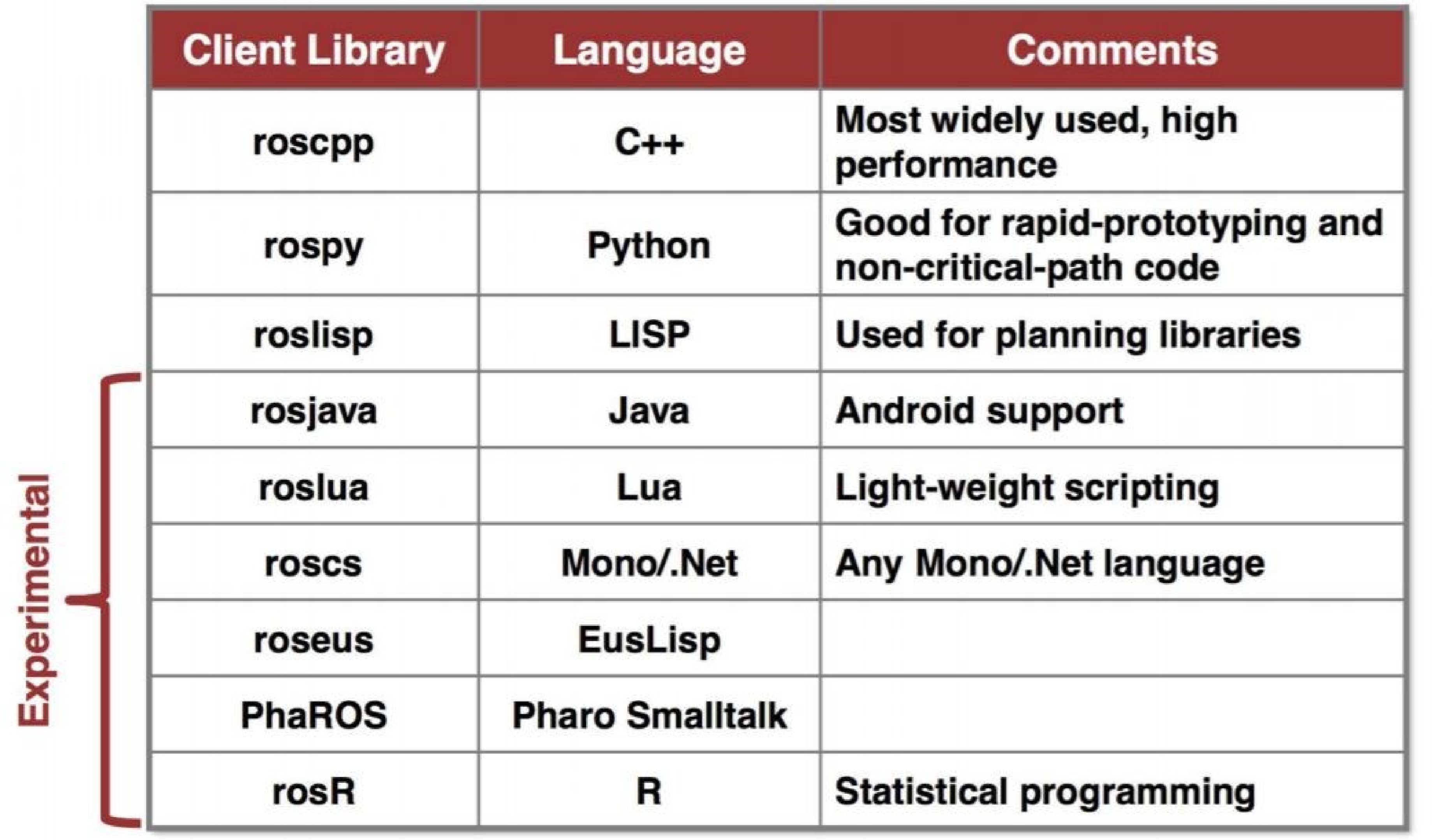
杰克科技 中央研究院





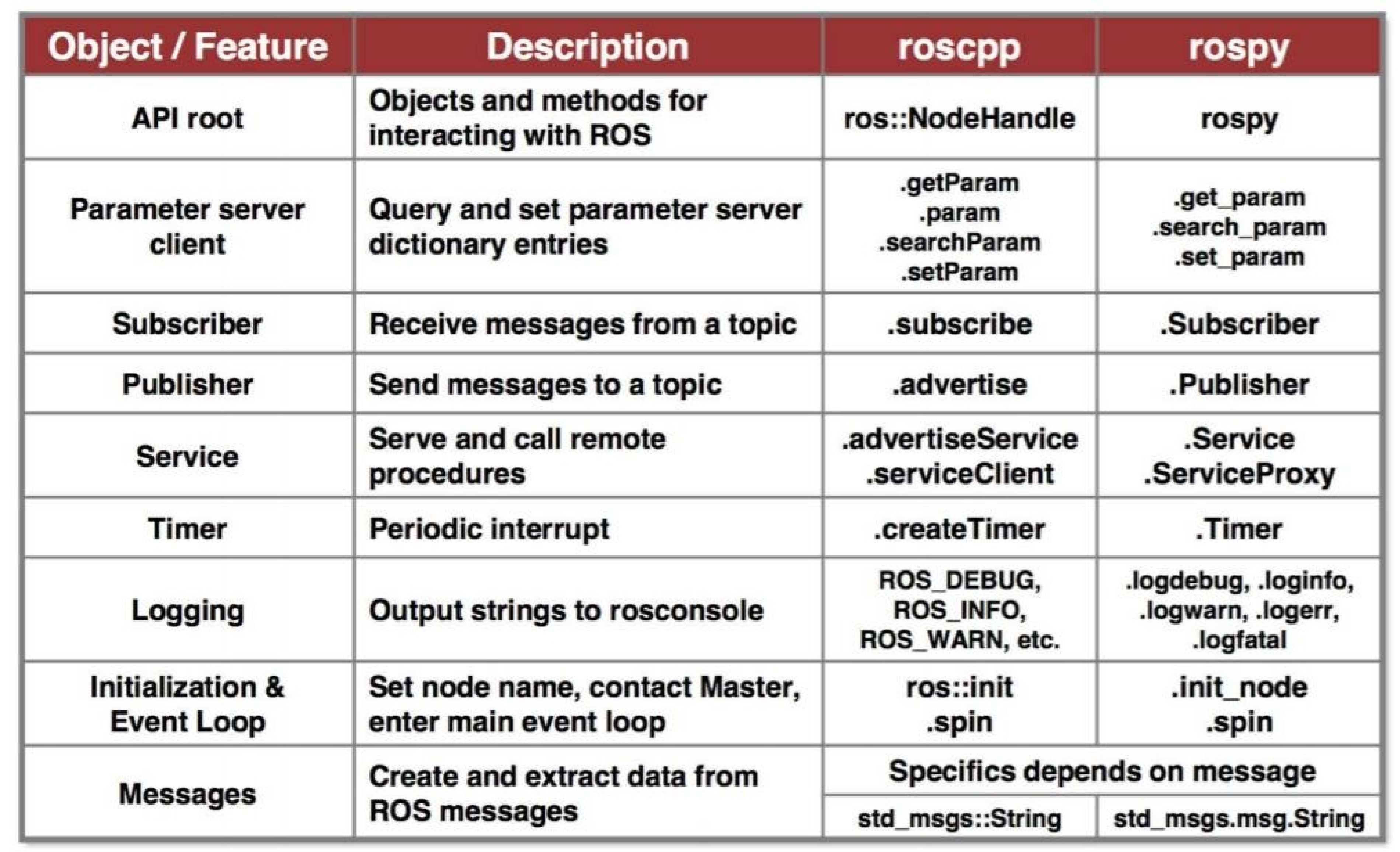


ROS client libraries



	Client Library	Language	Comments
	roscpp	C++	Most widely used, high performance
	rospy	Python	Good for rapid-prototyping and non-critical-path code
	roslisp	LISP	Used for planning libraries
Experimental	rosjava	Java	Android support
	roslua	Lua	Light-weight scripting
	roscs	Mono/.Net	Any Mono/.Net language
	roseus	EusLisp	
	PhaROS	Pharo Smalltalk	
	rosR	R	Statistical programming

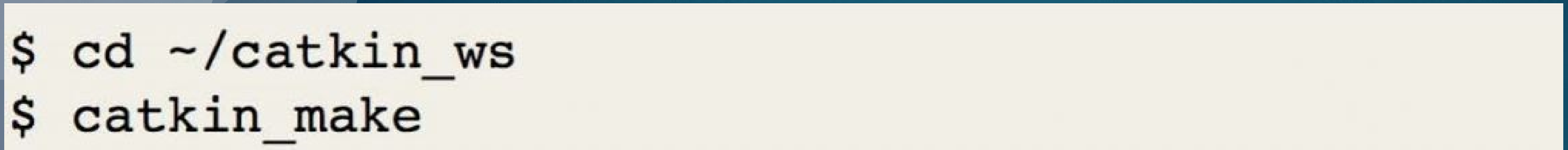
Client API Commonly Used Features



Object / Feature	Description	roscpp	rospy
API root	Objects and methods for interacting with ROS	ros::NodeHandle	rospy
Parameter server client	Query and set parameter server dictionary entries	.getParam .param .searchParam .setParam	.get_param .search_param .set_param
Subscriber	Receive messages from a topic	.subscribe	.Subscriber
Publisher	Send messages to a topic	.advertise	.Publisher
Service	Serve and call remote procedures	.advertiseService .serviceClient	.Service .ServiceProxy
Timer	Periodic interrupt	.createTimer	.Timer
Logging	Output strings to rosconsole	ROS_DEBUG, ROS_INFO, ROS_WARN, etc.	.logdebug, .loginfo, .logwarn, .logerr, .logfatal
Initialization & Event Loop	Set node name, contact Master, enter main event loop	ros::init .spin	.init_node .spin
Messages	Create and extract data from ROS messages	Specifics depends on message	
		std_msgs::String	std_msgs.msg.String

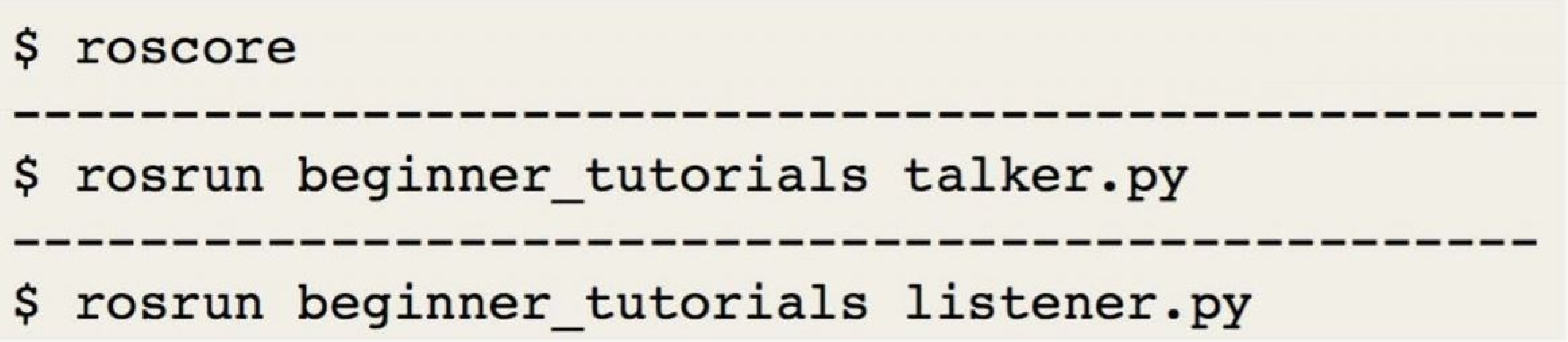
Recap: Run ROS Projects

- To compile ROS catkin workspace



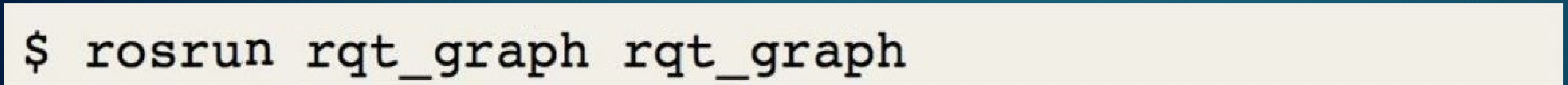
```
$ cd ~/catkin_ws
$ catkin_make
```

- Run ROS packages on Different Shells



```
$ roscore
----------------------------------------
$ rosrun beginner_tutorials talker.py
----------------------------------------
$ rosrun beginner_tutorials listener.py
```

- View ROS nodes and topics



```
$ rosrun rqt_graph rqt_graph
```

ROS Services

- ◆ ROS Topic Model（Publish/Subscribe， Multiple to Multiple nodes， Uni-directional transmission）Can deal with most applications

- ◆ However,， there are some applications which require request and response mechanism
  - * Like remote procedure call
  - * Request： Client node send topics or data to target node
  - * Response： target node responds to the client node
  - * Ex： Configure camera focal parameters， and receive focal parameters adjustment success

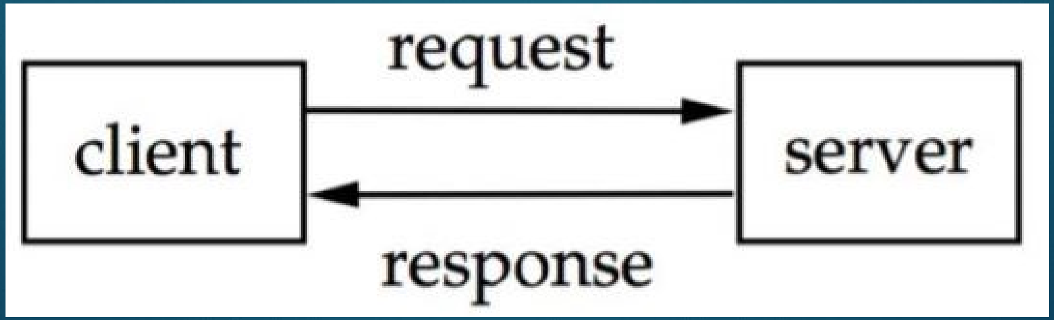
- ◆ Service： ROS way to implement request/response pair
  - * Service has its own name： Client and target nodes work with their names， ROS master registers the service names
  - * Service is defined in the .srv file（Similar to define .msg file， with the separated define of the request and response message types）
  - * Service should not be interrupted, response should be in time
  - * Cons： No recordings, so hard to trace
  - * Not as common as topics publishing / subscribing

ROS Service： Establish request/response communication between nodes

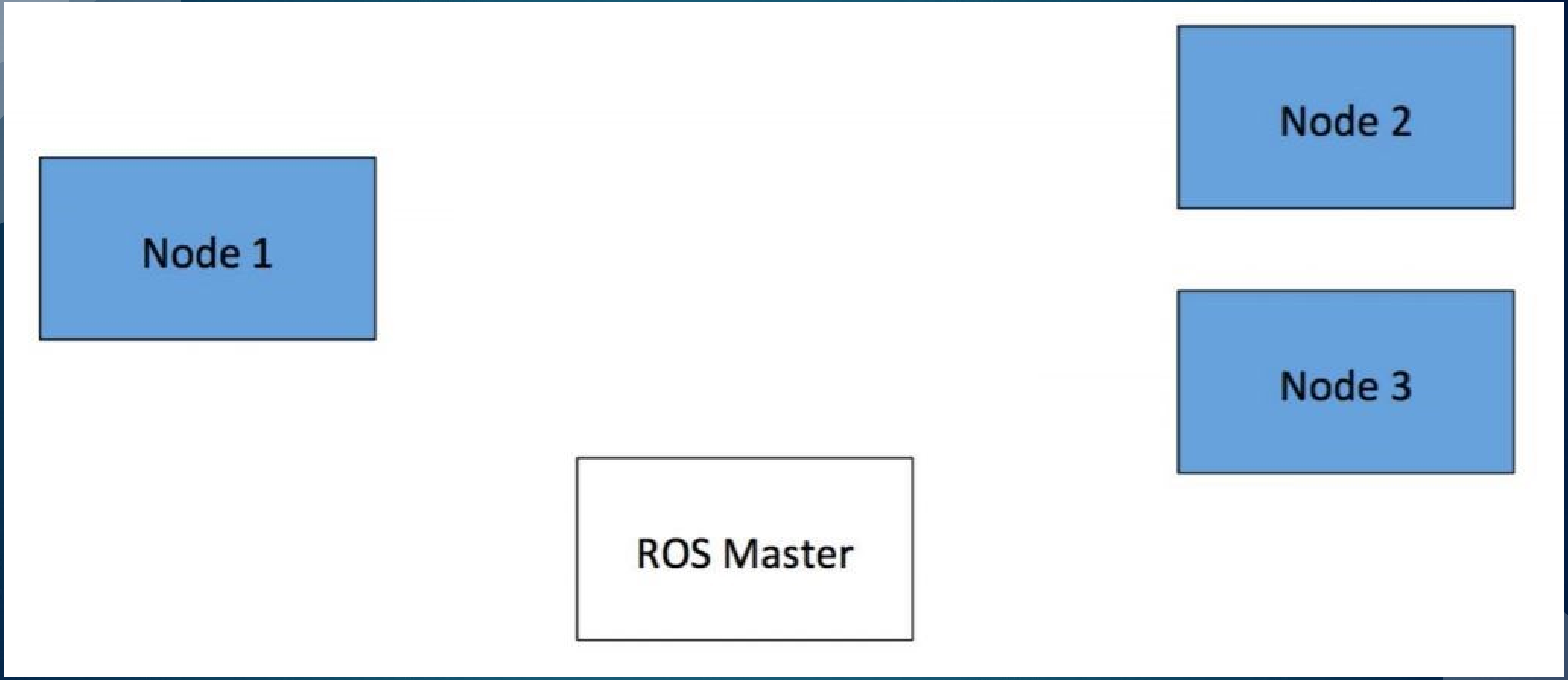
- **Service definition is saved in srv subdir of related packages as the .srv file**

- **When user publishes a topic, topics are uni-directional and can be received by multiple nodes, no feedback, and even no guarantee if there is any node will receive a topic; However, service has different transmission method which is different from topic broadcasting.**

- **Servie is bi-directional data flow, response is provided by receiver node when it get a request.**

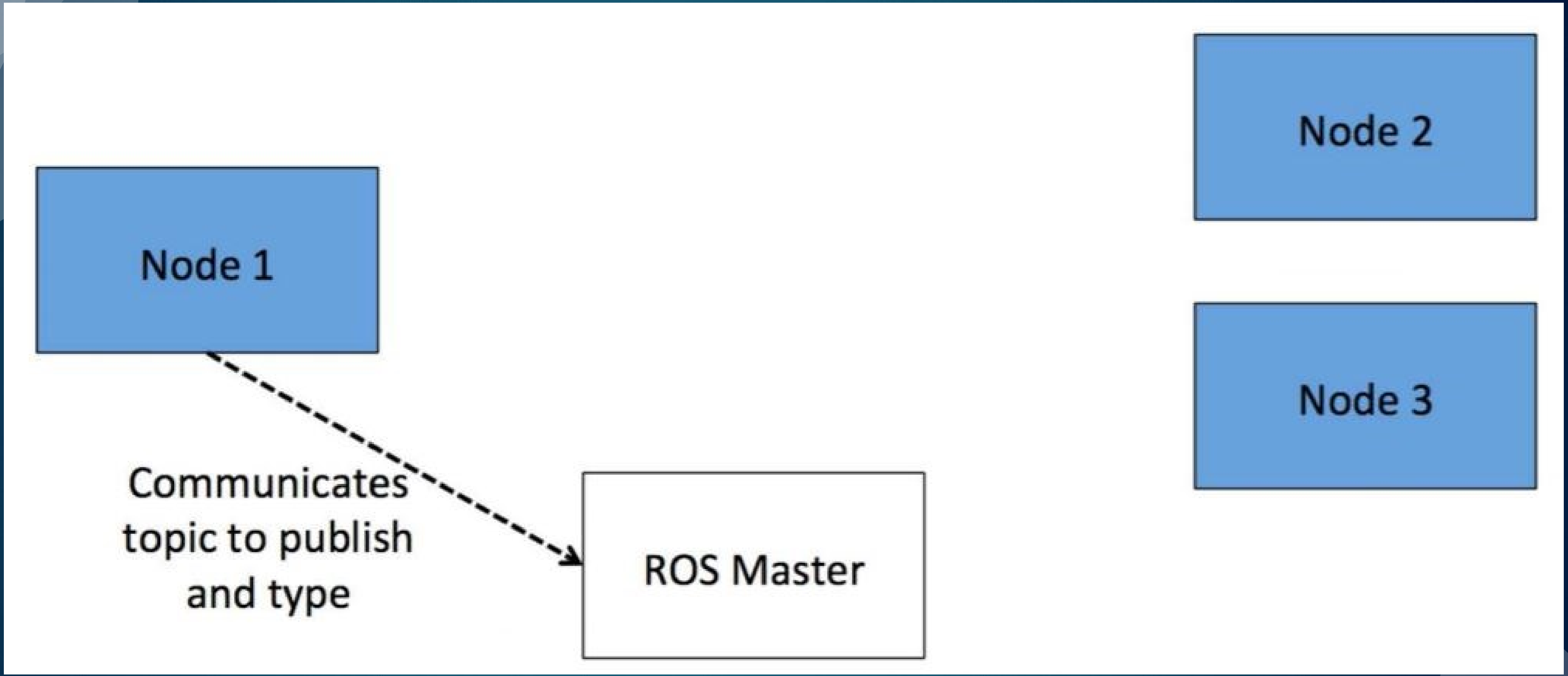


ROS Communication： Publish/Subscribe



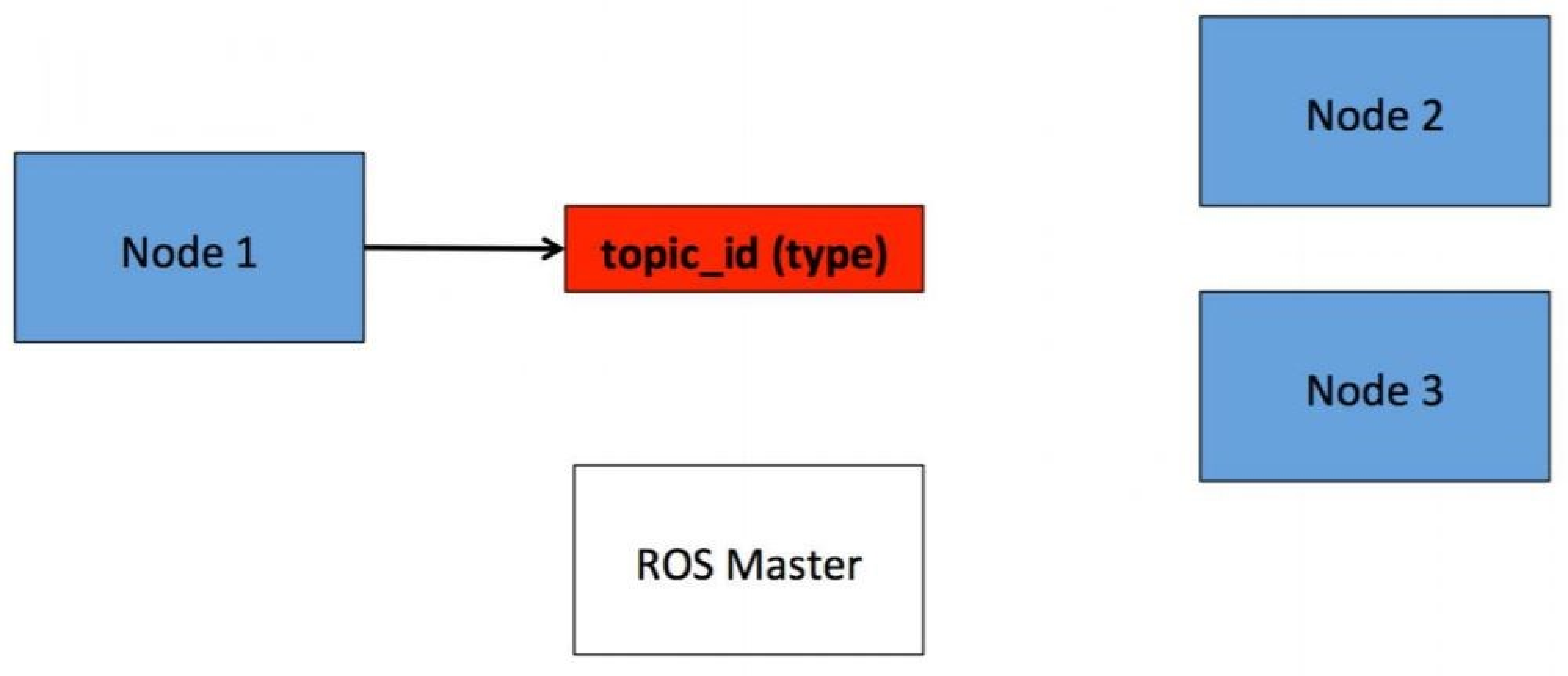
asynchronous、 multiple-multiple nodes、 uni-directional transmission

ROS Communication： Publish/Subscribe



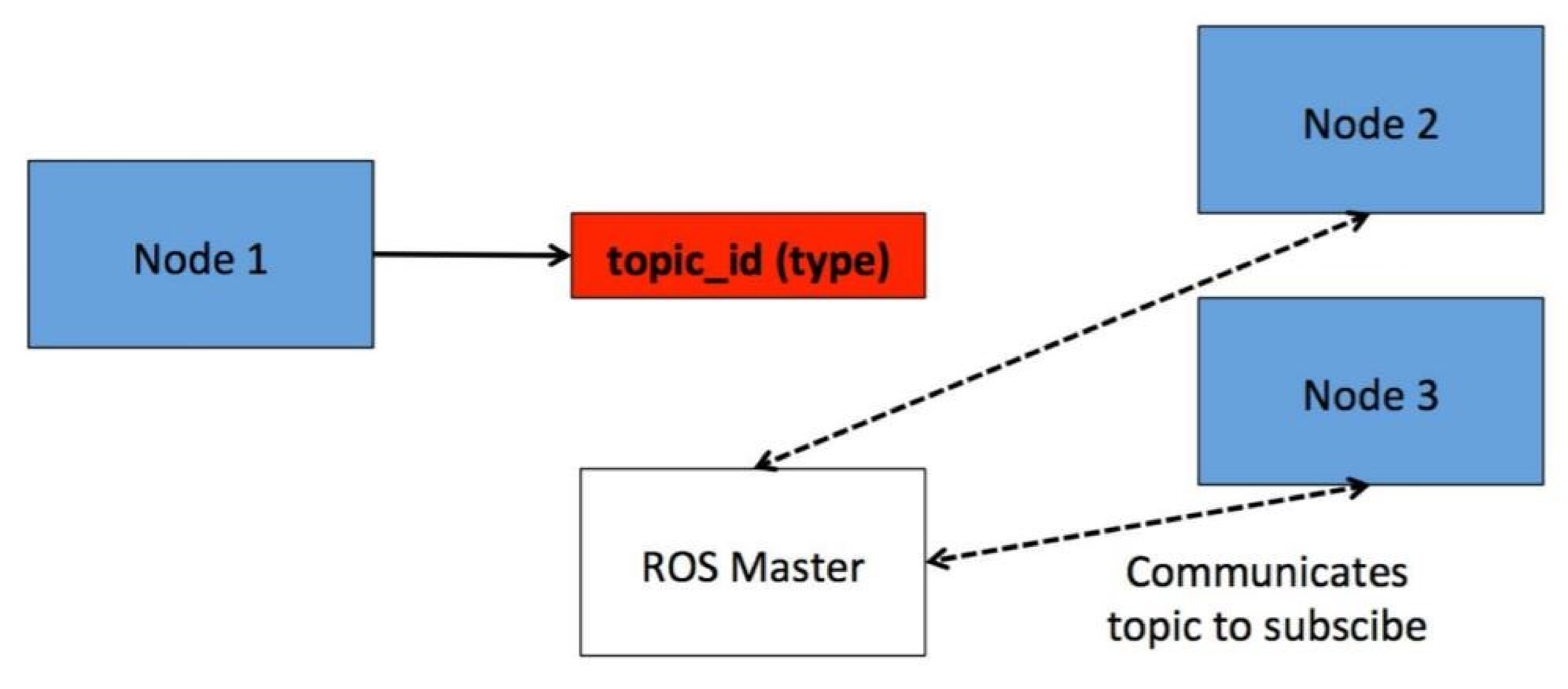
asynchronous、 multiple-multiple nodes、 uni-directional transmission

ROS Communication： Publish/Subscribe



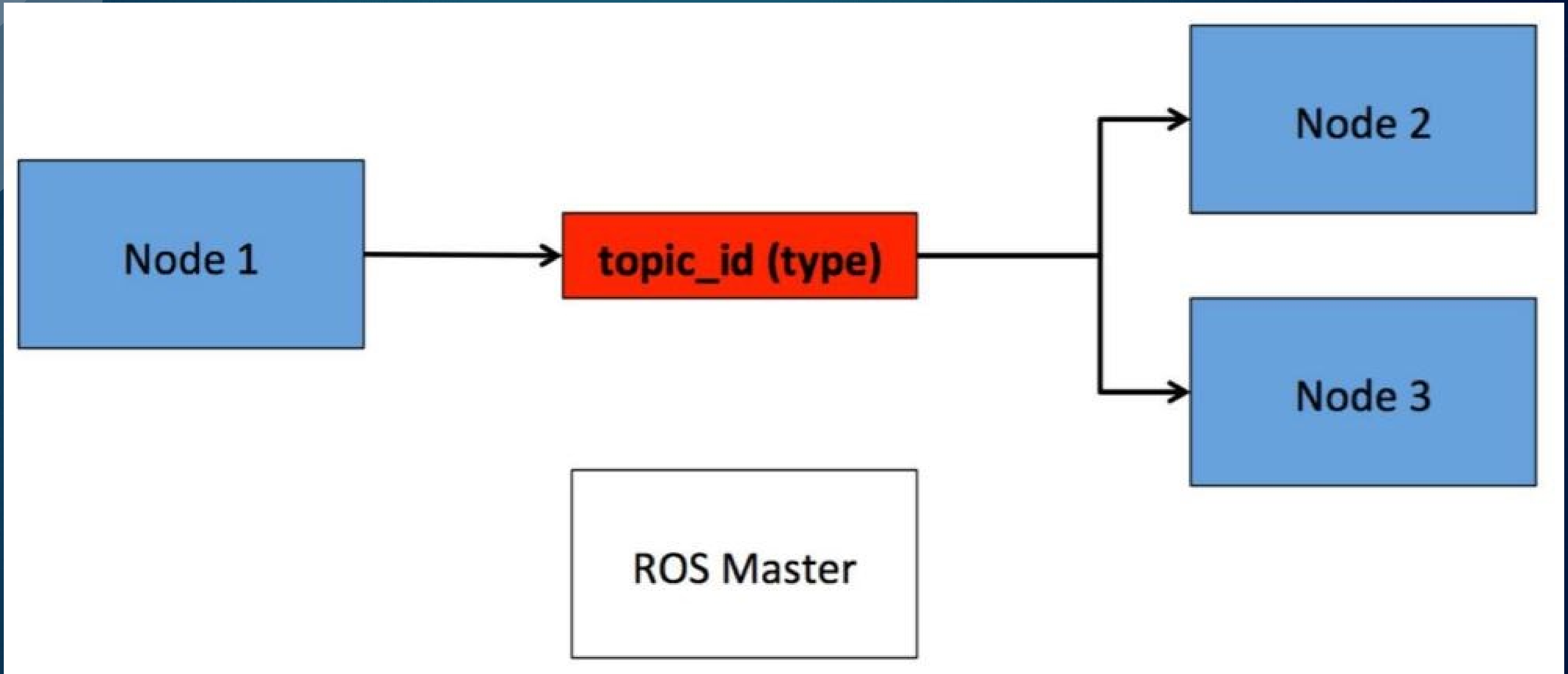
asynchronous、multiple-multiple nodes、uni-directional transmission

ROS Communication： Publish/Subscribe



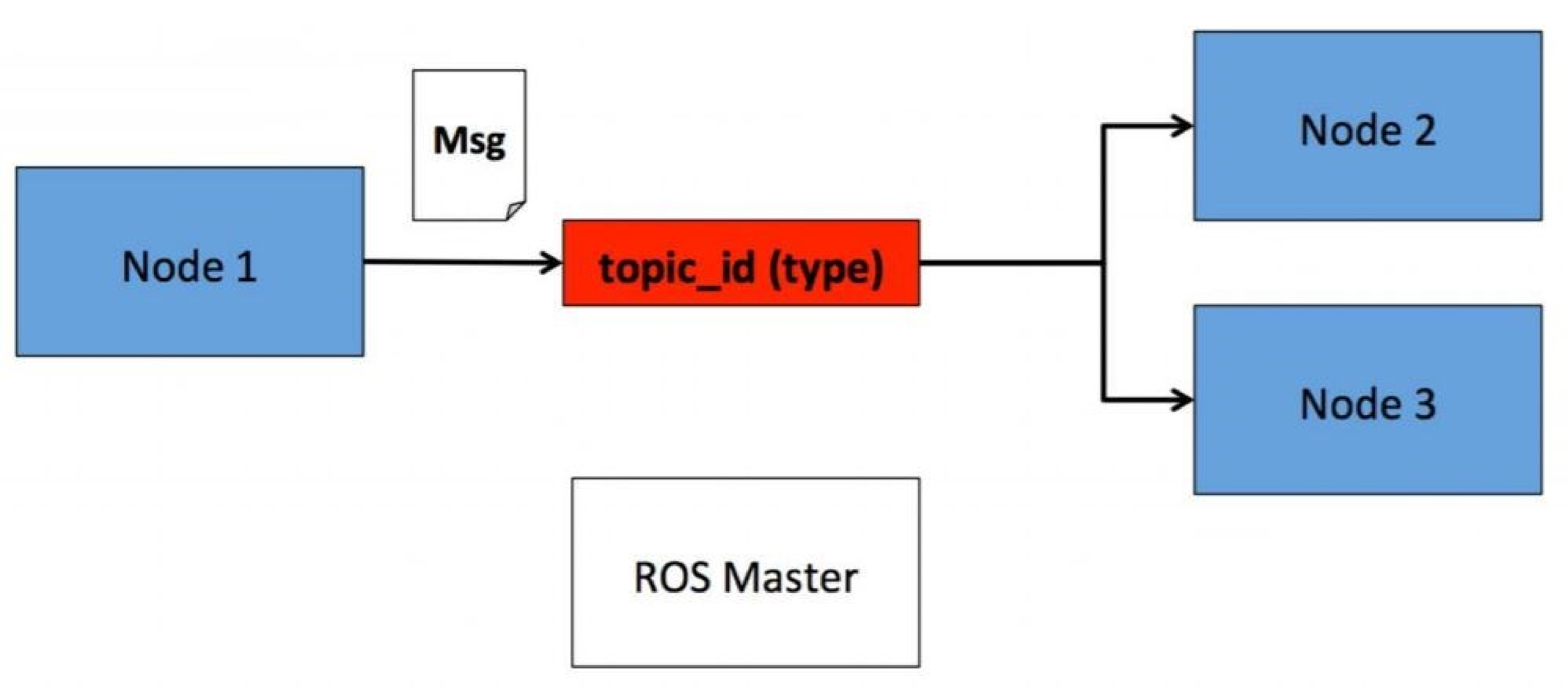
asynchronous、multiple-multiple nodes、uni-directional transmission

ROS Communication： Publish/Subscribe



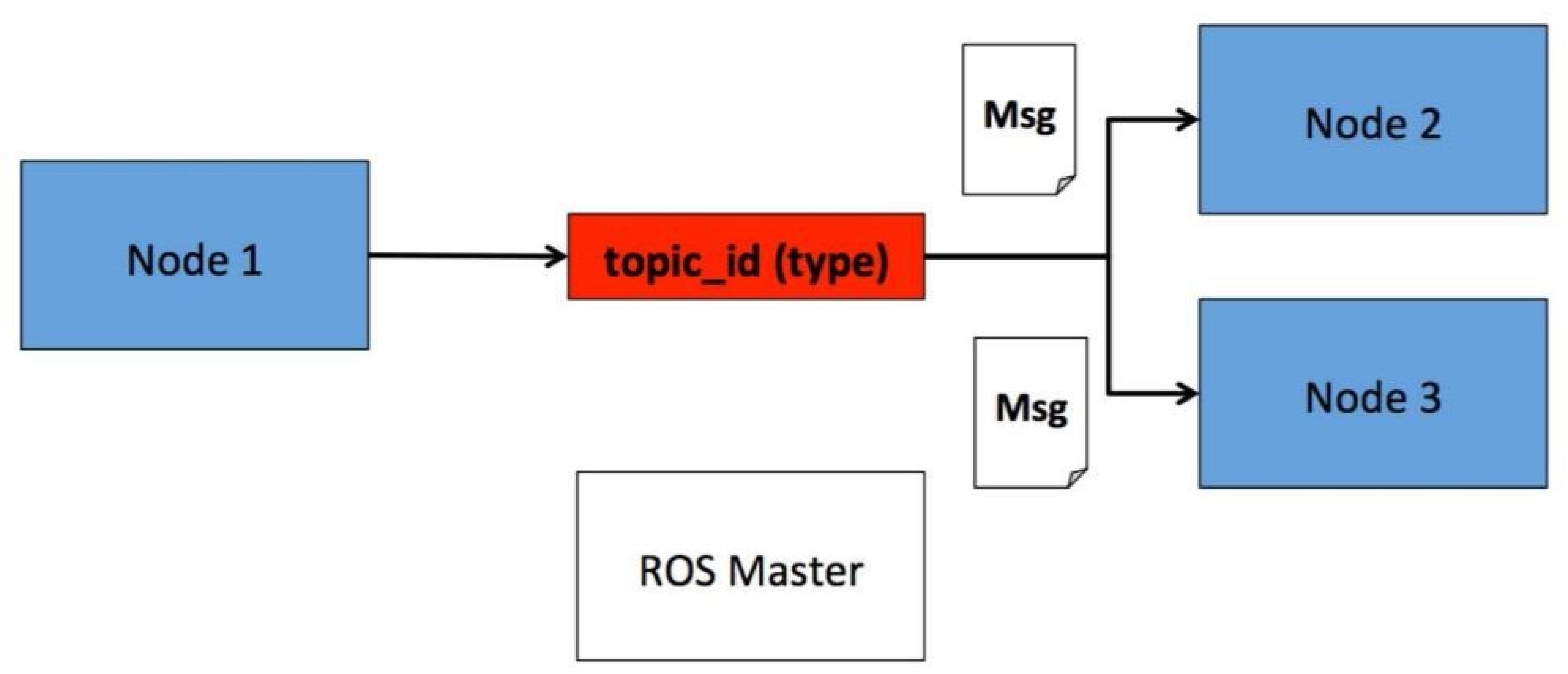
asynchronous、 multiple-multiple nodes、 uni-directional transmission

ROS Communication： Publish/Subscribe



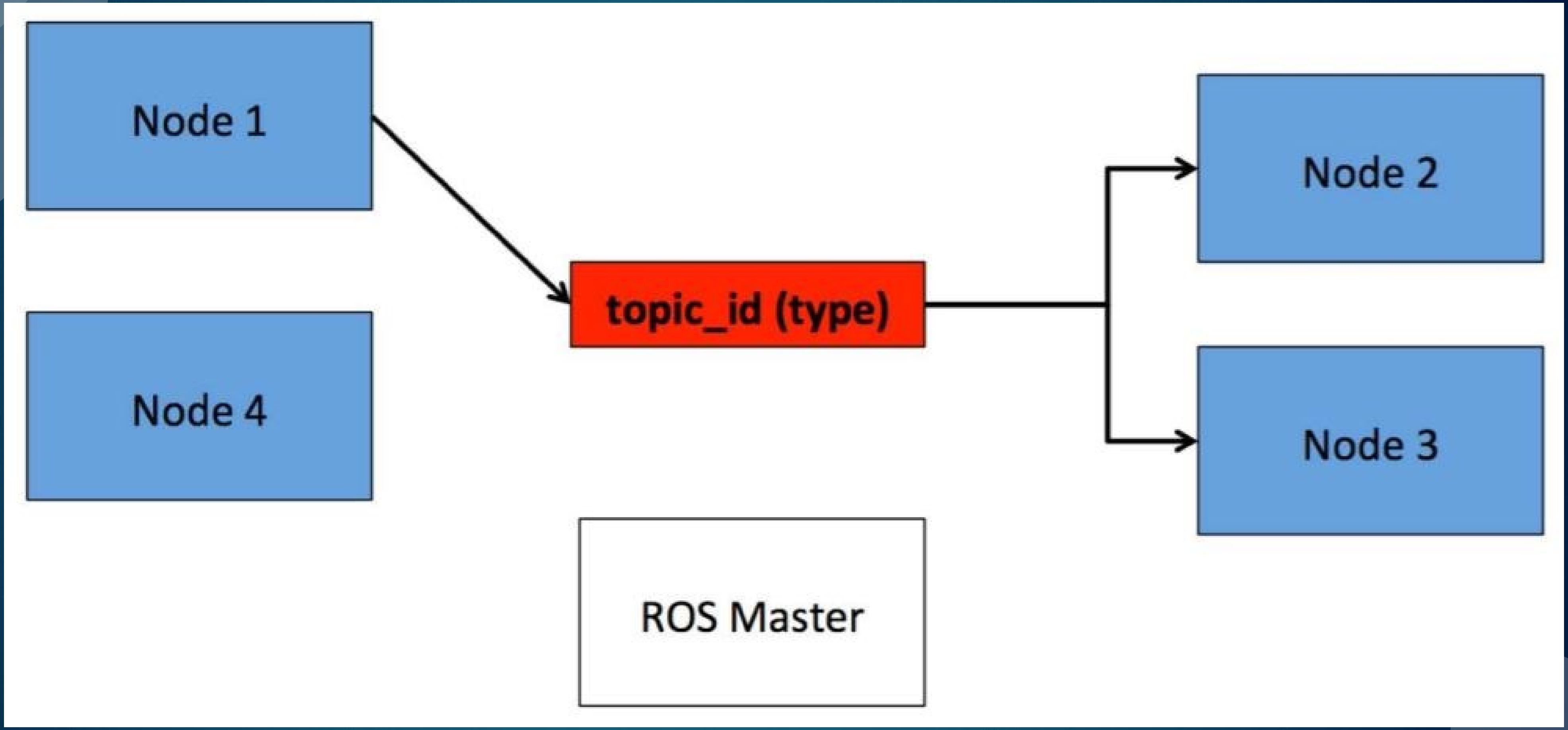
asynchronous、 multiple-multiple nodes、 uni-directional transmission

ROS Communication： Publish/Subscribe



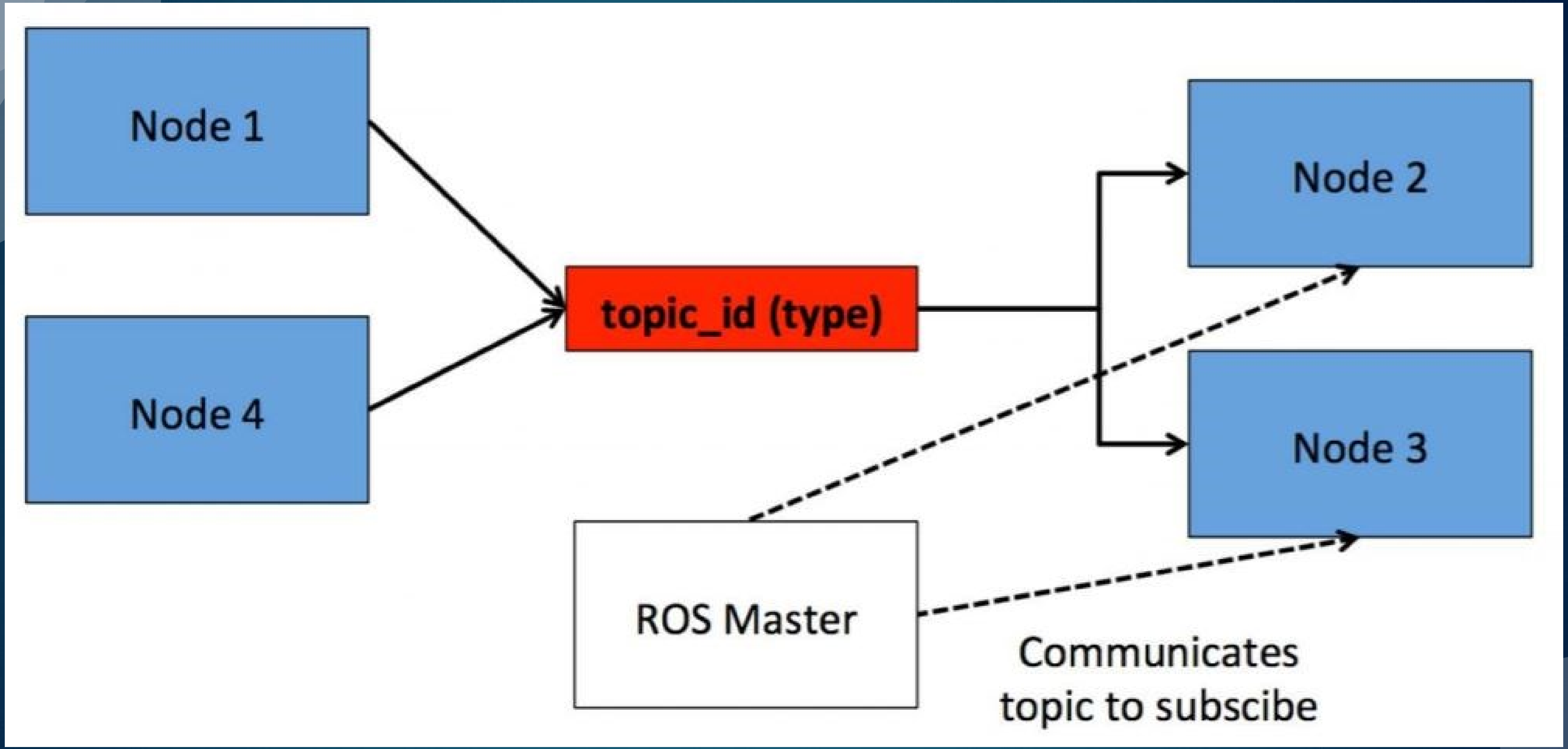
asynchronous、 multiple-multiple nodes、 uni-directional transmission

ROS Communication：Publish/Subscribe



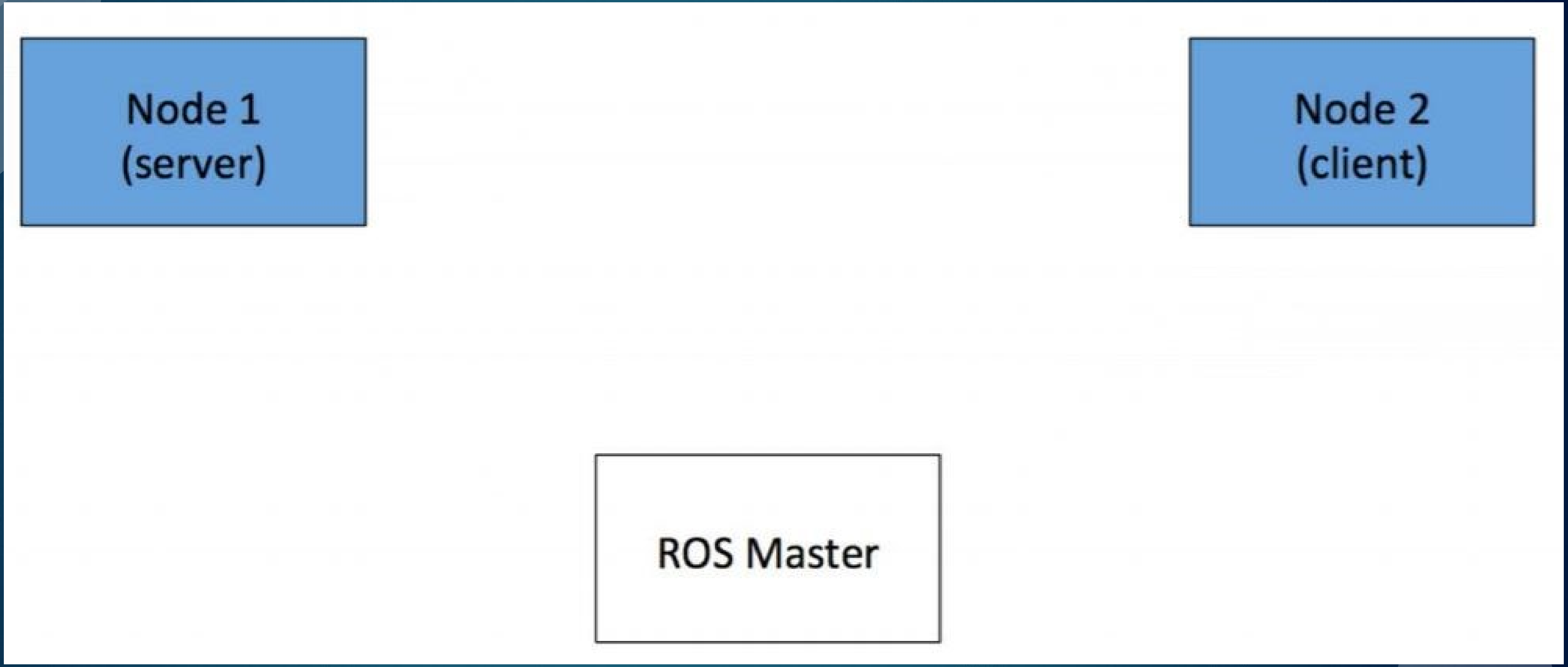
asynchronous、multiple-multiple nodes、uni-directional transmission

ROS Communication： Publish/Subscribe



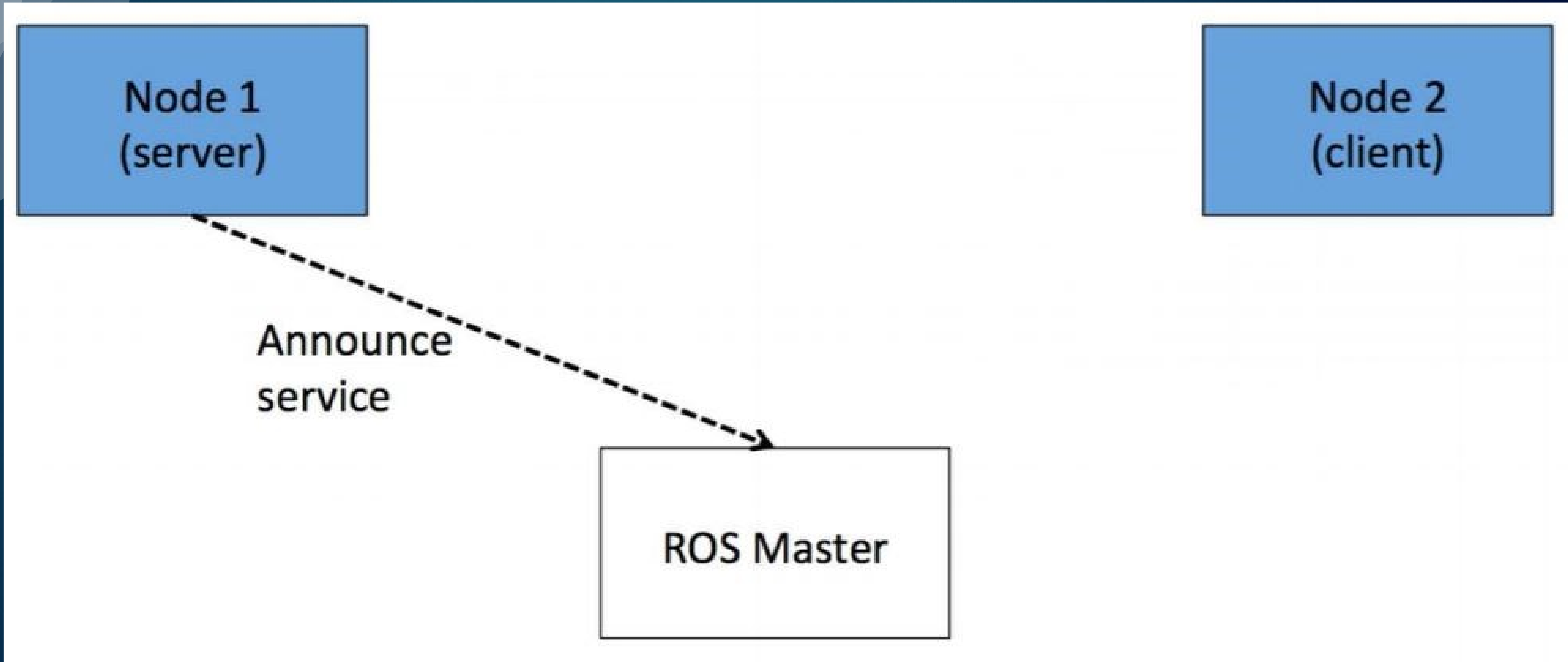
asynchronous、multiple-multiple nodes、uni-directional transmission

ROS Communication：Client/Service



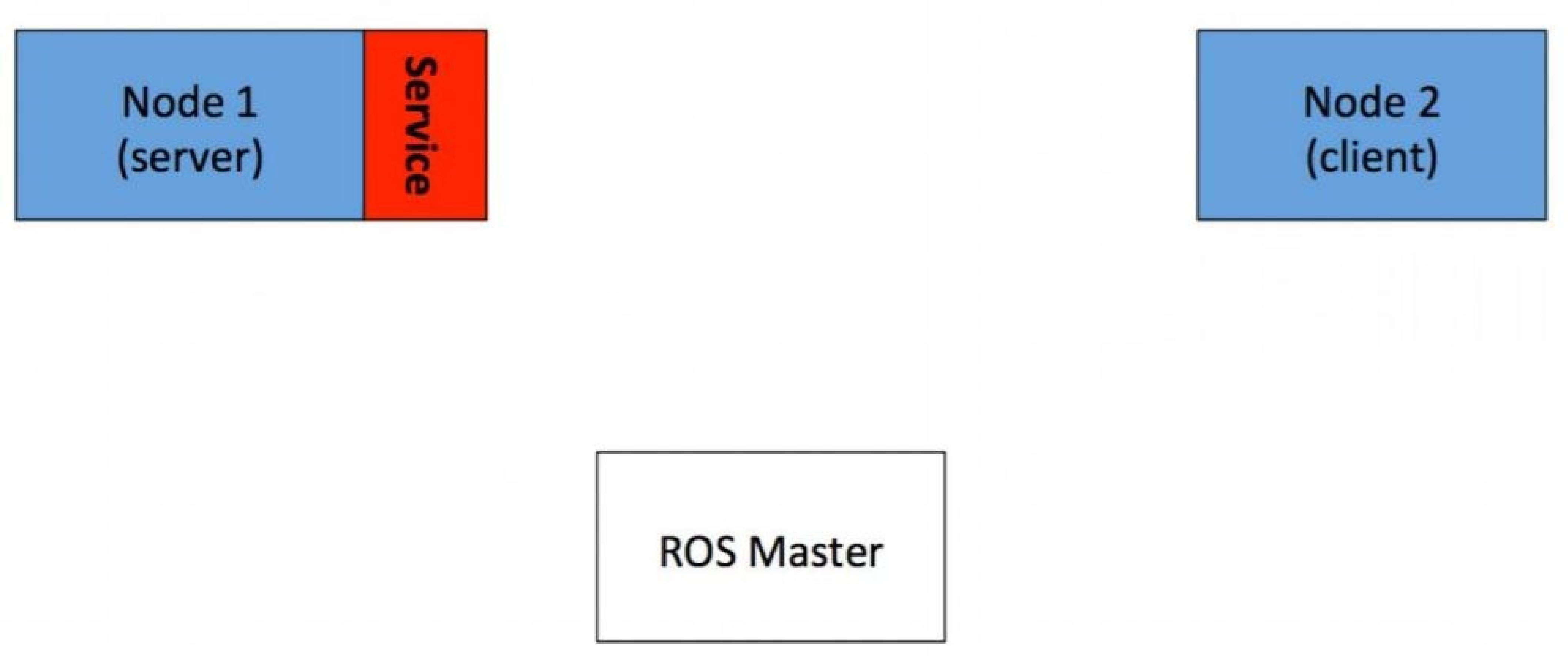
Synchronous、One-Multiple、Bi-directional transmission

ROS Communication： Client/Service



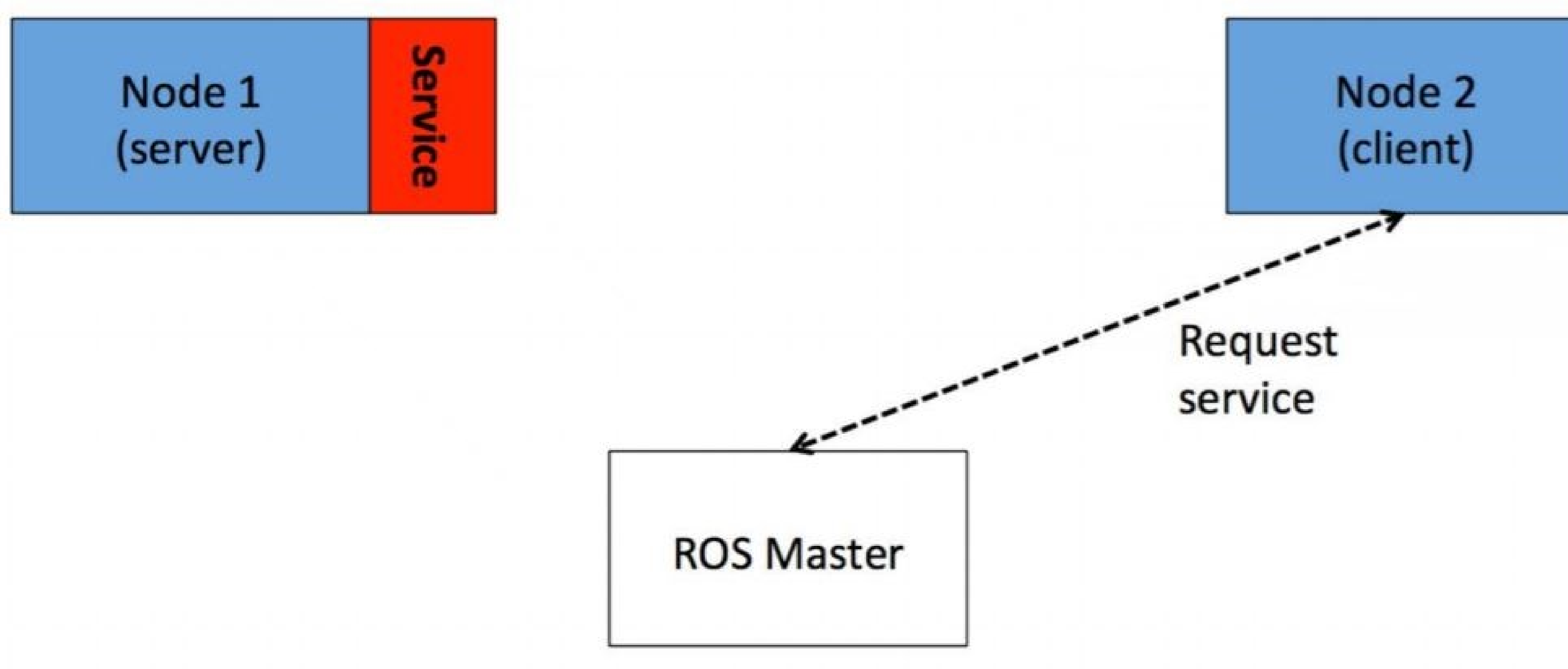
Synchronous、 One-Multiple、 Bi-directional transmission

ROS Communication： Client/Service



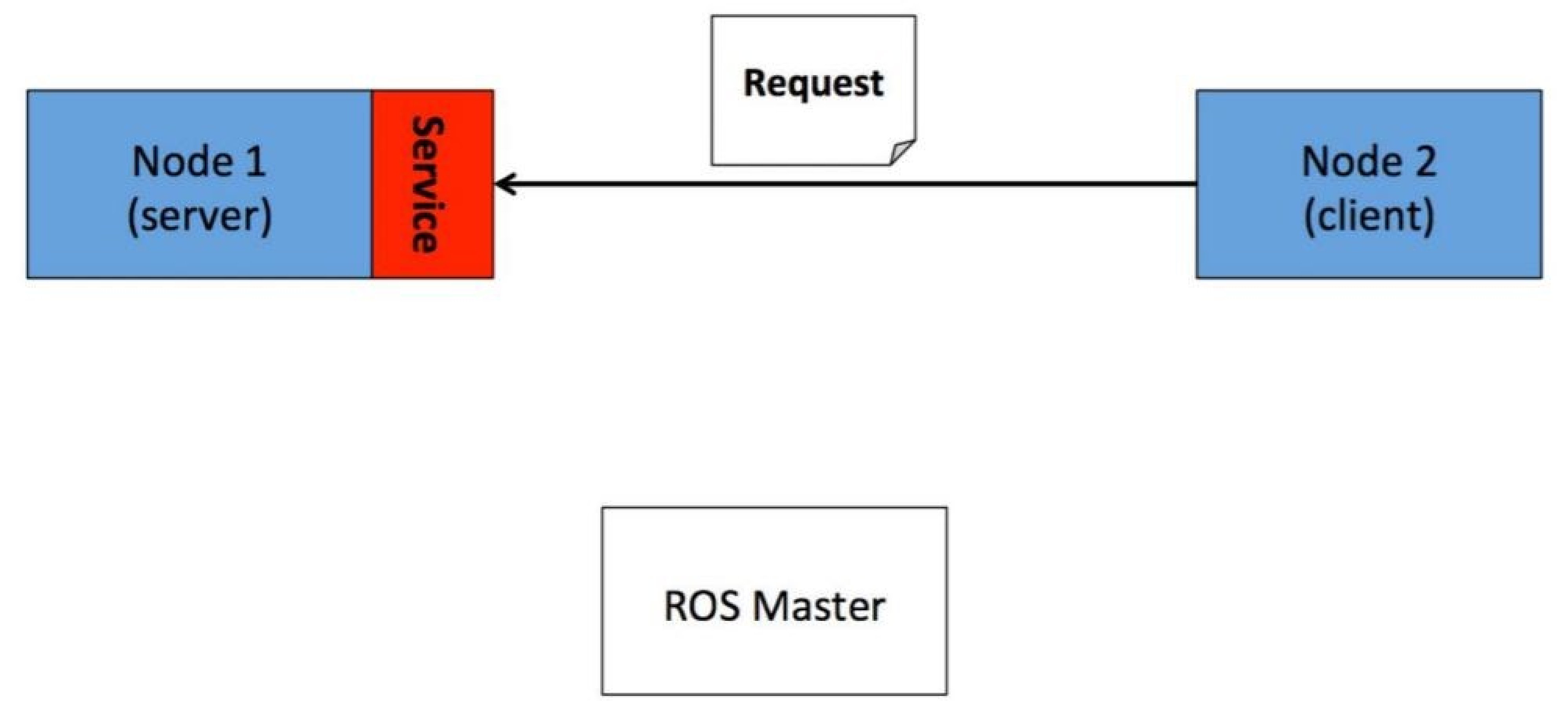
Synchronous、 One-Multiple、 Bi-directional transmission

ROS Communication： Client/Service



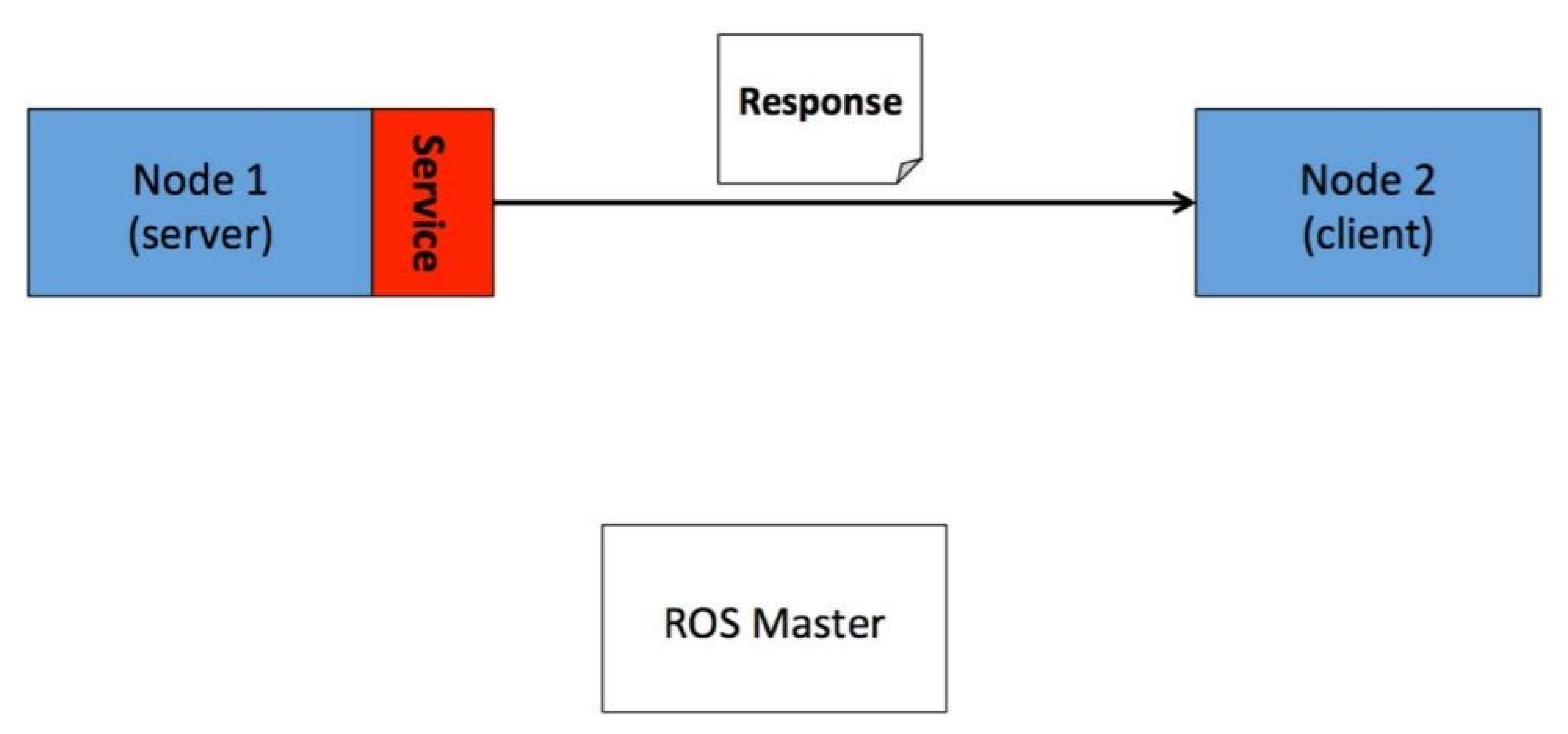
Synchronous、 One-Multiple、 Bi-directional transmission

ROS Communication： Client/Service



Synchronous、 One-Multiple、 Bi-directional transmission

ROS Communication：Client/Service



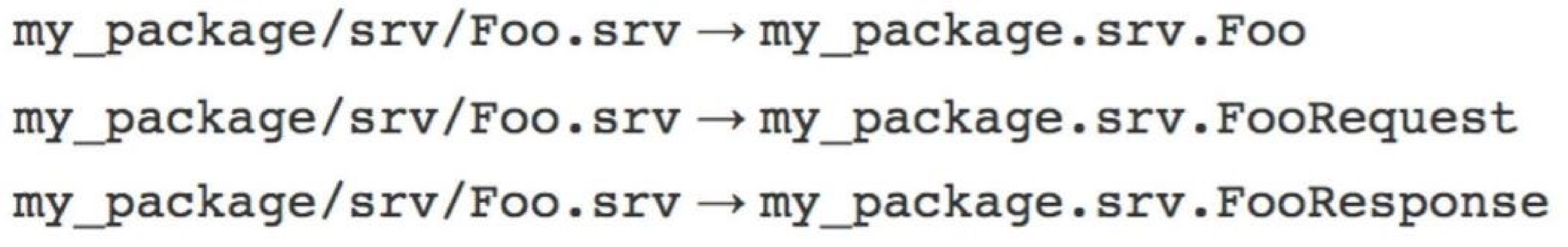
Synchronous、 One-Multiple、 Bi-directional transmission

rospy client library: Services

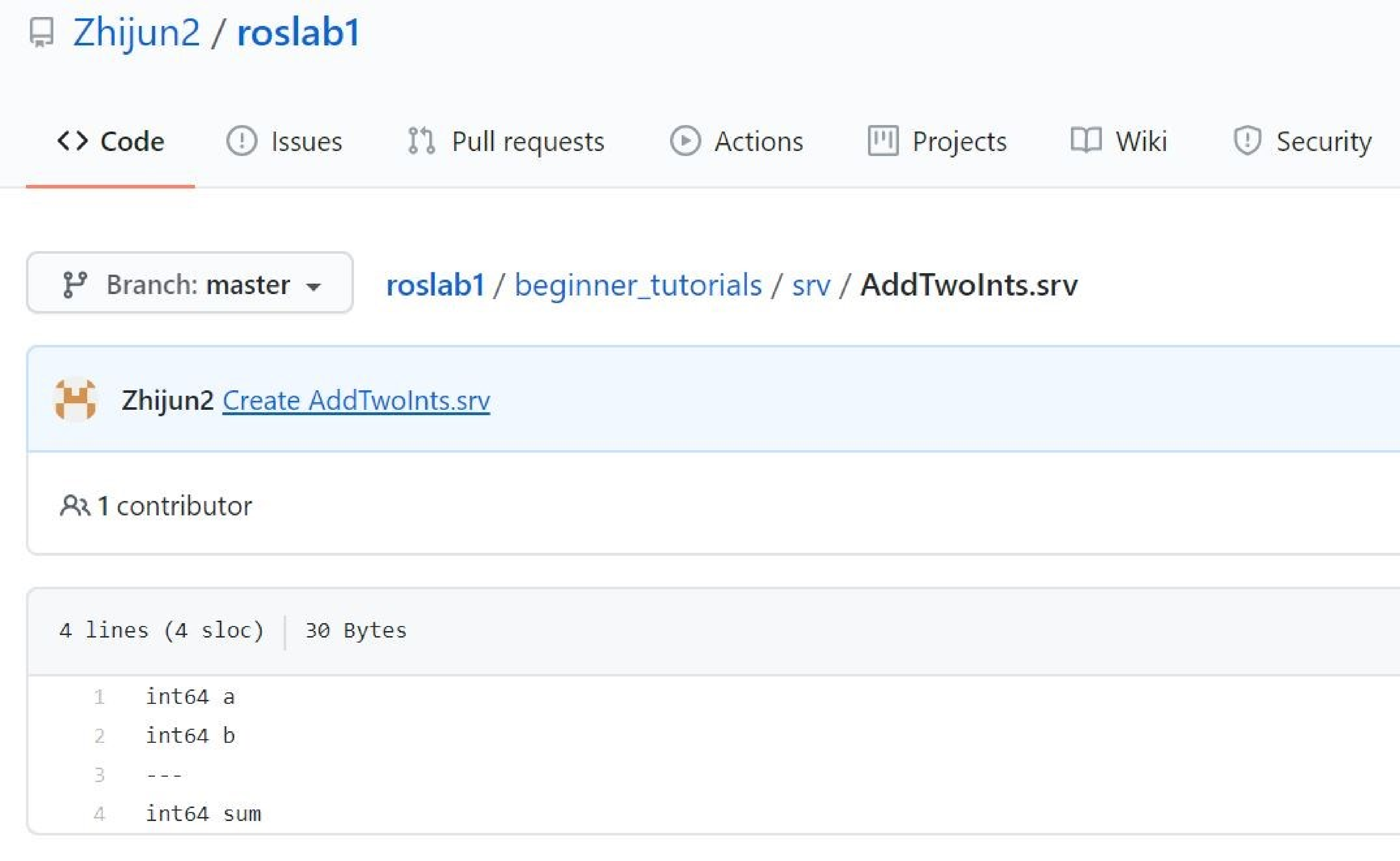
- ROS service is defined by srv file， which has one request message and one response message

- rospy converts this srv file to the Python code， 3 classes are automatically generated： Service definition class, Request message class and response message class

- The names of these classes are from the srv file， ex：



```
my_package/srv/Foo.srv → my_package.srv.Foo
my_package/srv/Foo.srv → my_package.srv.FooRequest
my_package/srv/Foo.srv → my_package.srv.FooResponse
```



Zhijun2 / roslab1

roslab1 / beginner_tutorials / srv / AddTwoInts.srv

Zhijun2 Create AddTwoInts.srv

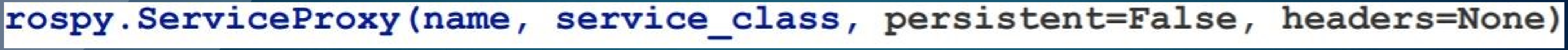
1 contributor

4 lines (4 sloc) | 30 Bytes

```
int64 a
int64 b
---
int64 sum
```

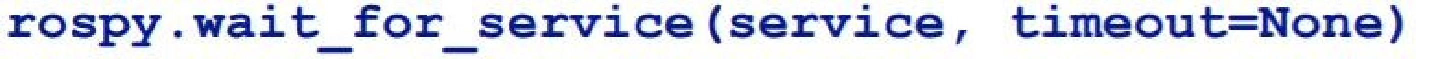
rospy client library: Service proxies

- Create a callable proxy to a service



```
rospy.ServiceProxy(name, service_class, persistent=False, headers=None)
```

- Wait for the service ready



```
rospy.wait_for_service(service, timeout=None)
```

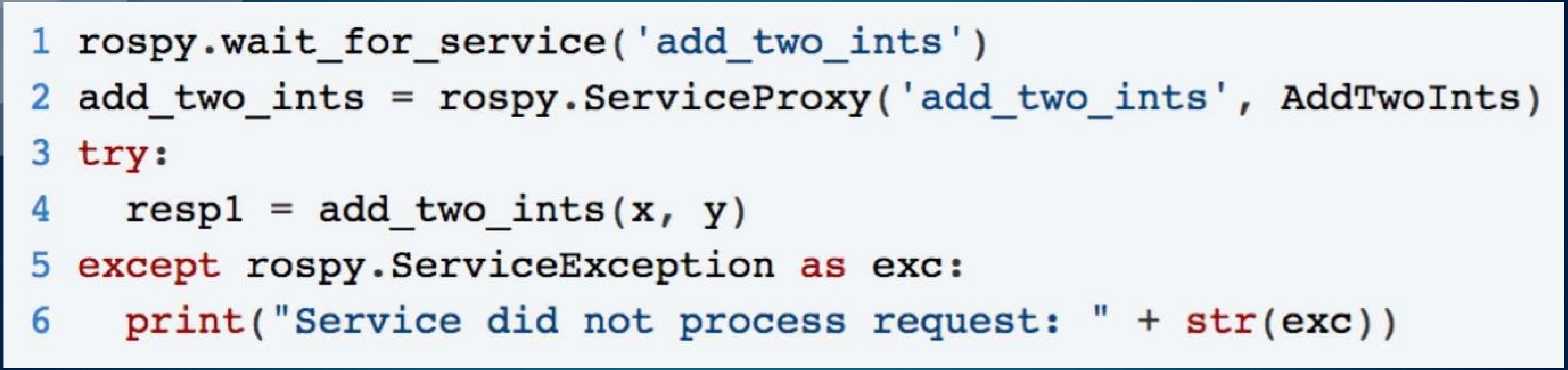
rospy client library: Calling Service proxies

- User call a service through generation of an instruction rospy.ServiceProxy with the service name

- User usually needs to call rospy.wait_for_service() to wait for a service becoming ready

- If any error occurs when calling a service, rospy.ServiceException returns error messages for managing and debugging errors.

rospy client library: Service call example



```python
rospy.wait_for_service('add_two_ints')
add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
try:
  resp1 = add_two_ints(x, y)
except rospy.ServiceException as exc:
  print("Service did not process request: " + str(exc))
```

Create a ROS service

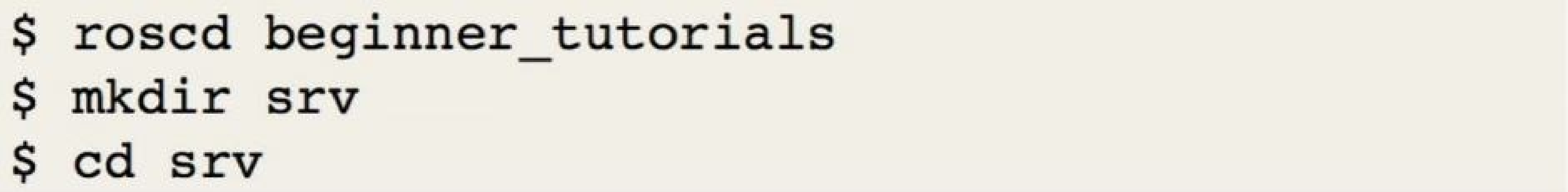
- The .srv file defines a service. A service is composed of two parts： Request and response. Service file is saved in the subdir of a package /srv

- The data types in a service are：

  - int8, int16, int32, int64
  - float32, float64
  - string
  - time, duration
  - Header
  - other msg files
  - variable-length array[] and fixed-length array[C]

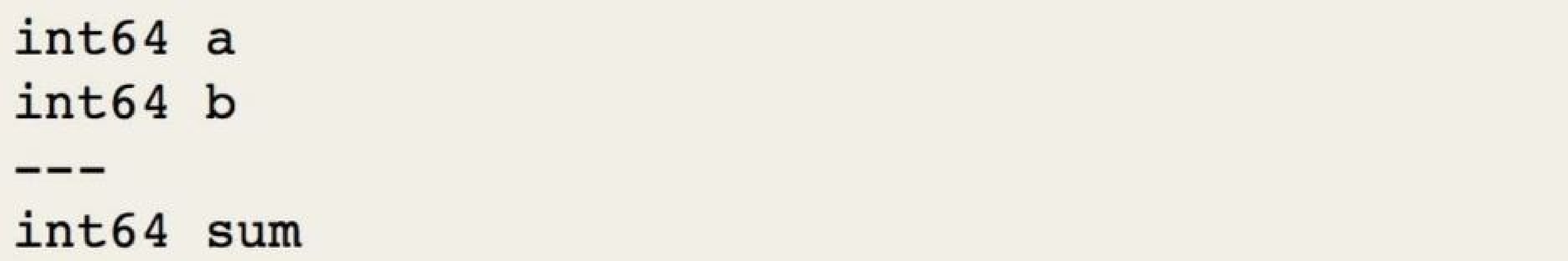
Create a ROS service (Example)

- Create a service subdir in the package of beginner_tutorials, with the name of srv



```
$ roscd beginner_tutorials
$ mkdir srv
$ cd srv
```

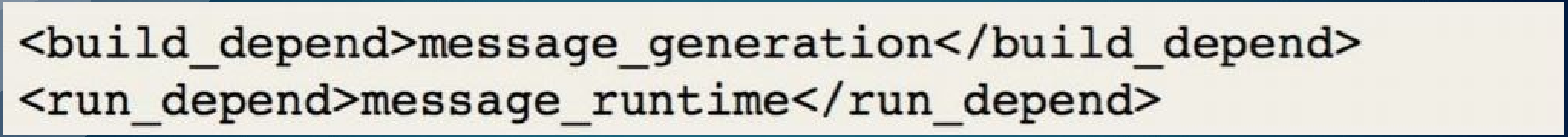
- Create a new file in srv subdir, naming it AddTwoInts.srv. Its content shows below



```
int64 a
int64 b
---
int64 sum
```

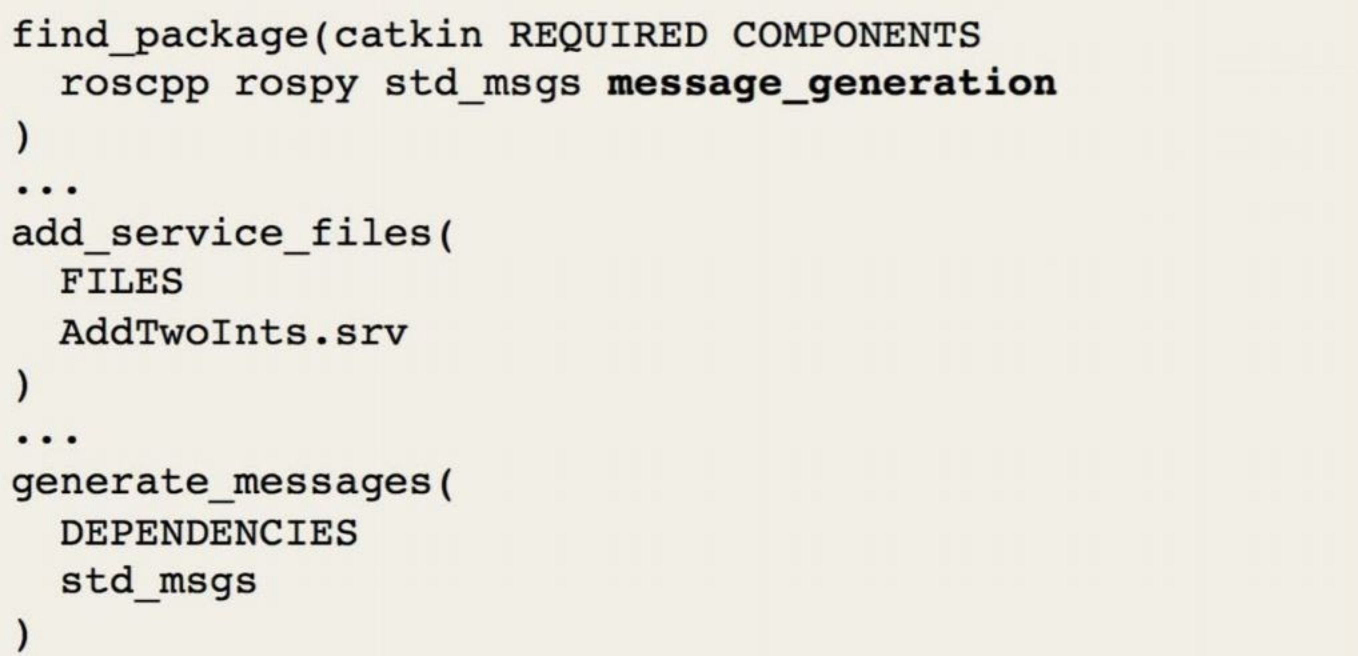
Create a ROS service (Example)

- Open package.xml， add the following lines：



```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

- Open CMakeList.txt and add/uncomment the following lines：



```
find_package(catkin REQUIRED COMPONENTS
  roscpp rospy std_msgs message_generation
)
...
add_service_files(
  FILES
  AddTwoInts.srv
)
...
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

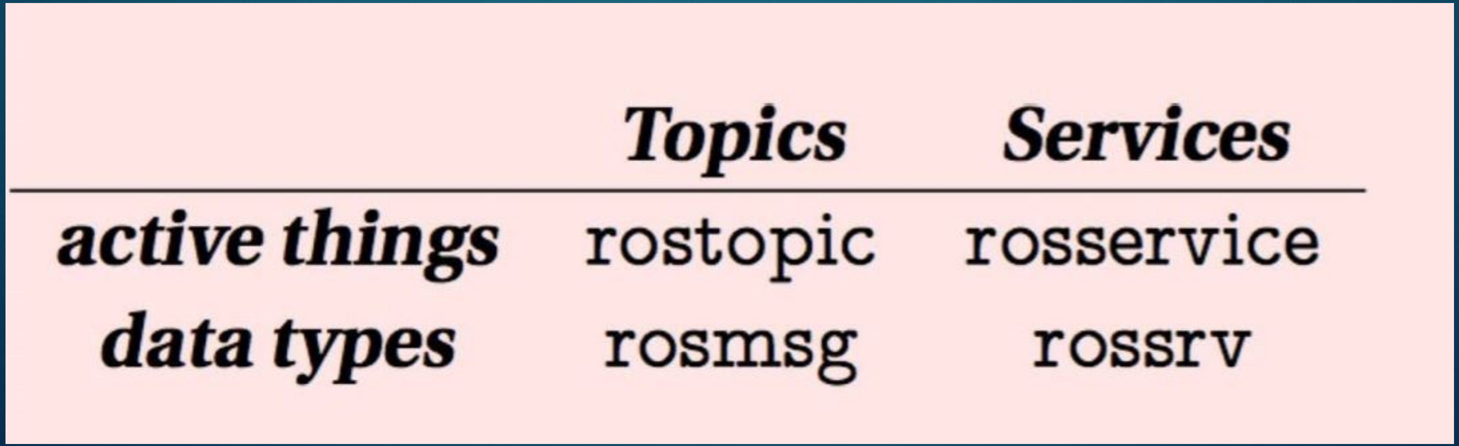
Create a ROS service (Example)

Check the online example below： add_two_ints_server.py and add_two_ints_client.py

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

After you view the example, compile and run

http://wiki.ros.org/ROS/Tutorials/ExaminingServiceClient

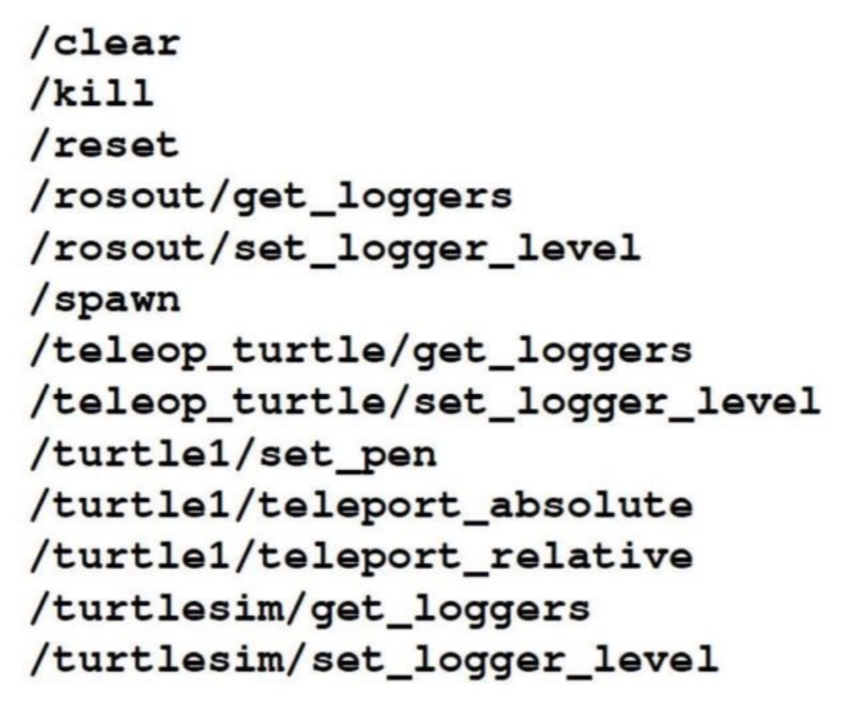


	Topics	*Services*
active things	rostopic	rosservice
data types	rosmsg	rossrv

rosservice and rossrv

% rosservice list

User will get something on the right



```
/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/spawn
/teleop_turtle/get_loggers
/teleop_turtle/set_logger_level
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level
```

% rosservice type /spawn

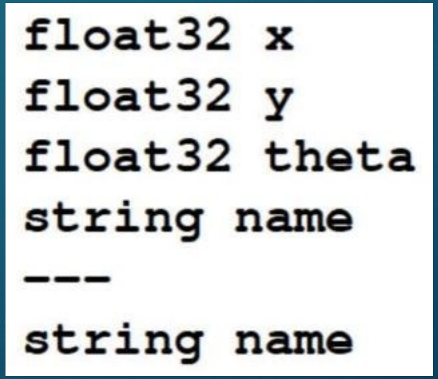
User will see the service types below

turtlesim/Spawn

Ex: show service /spawn data structure：

% rosservice type /spawn | rossrv info

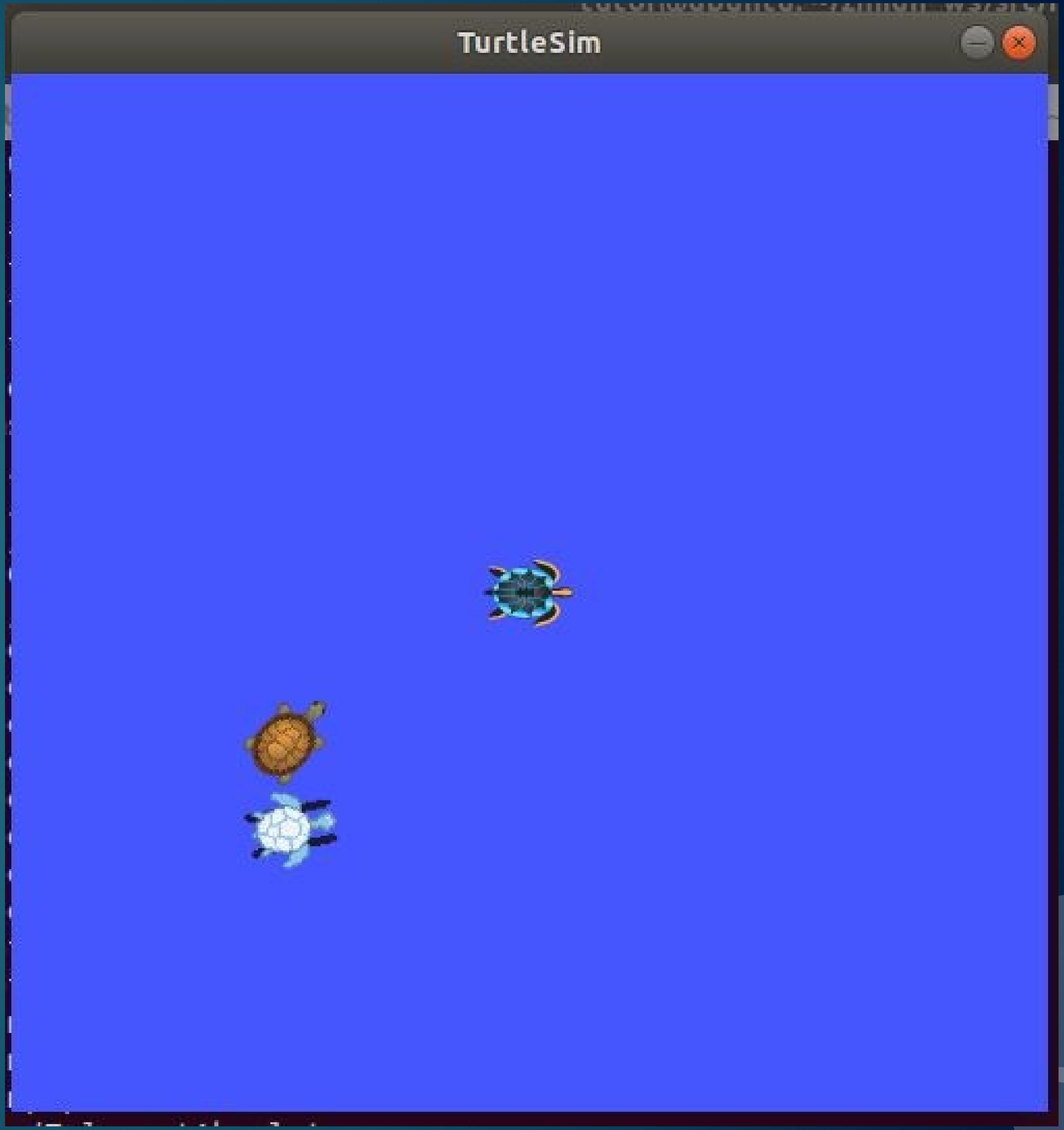
Results are shown right：



```
float32 x
float32 y
float32 theta
string name
---
string name
```

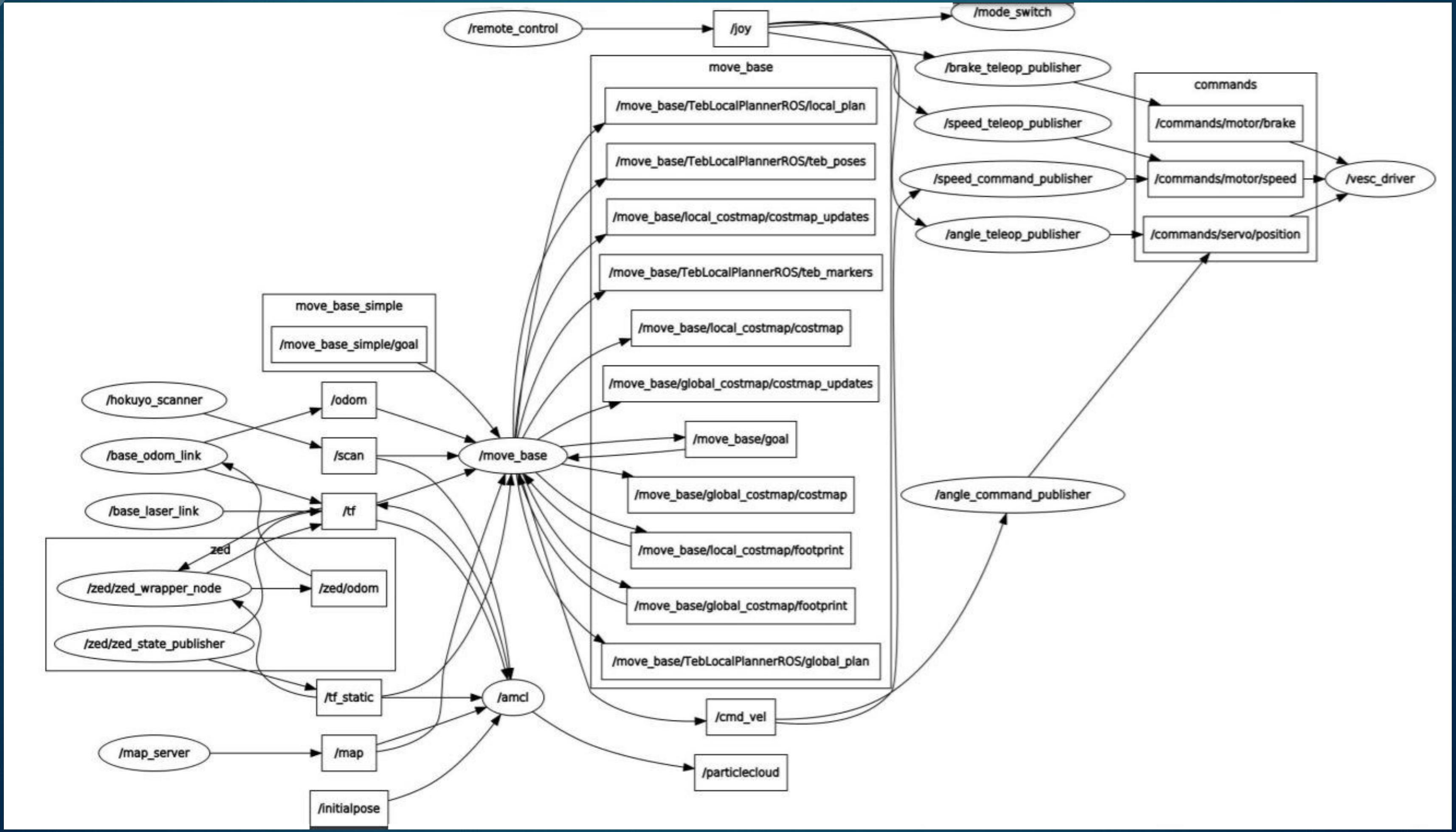
rosservice

```
% roscore
% rosrun turtlesim turtlesim_node
% rosservice call /spawn 5 4 0.8 ‘new_turtle’
```



ROS is a distribution system

- **It is common to have many nodes, and data configurations**
- **It is very hard if all system configurations are done manually**



Launch File

```
roslaunch  package_name  file.launch
```

What are Launch files

◆ launch file is actually an XML file, it has：

- Some nodes which need to be executed simultaneously
- Set up node parameters
- Can embed any other launch files

◆ roslaunch is a ROS document which is able to launch many nodes

◆ launch has an extension name .launch

◆ roslaunch automatically launch ROS master

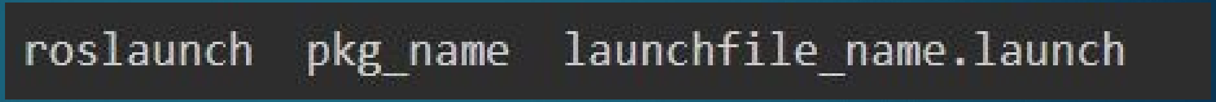
◆ Similar to script files， launch file is executed in order, no wait or pause（Parameters need to be set in parameter server before hand）

What are Launch files

- ROS robot is a system composed of many nodes and topics（plus service, parameters etc）, a network also called ROS graph， roslaunch can launch many nodes by only one click, and possible to set parameters in launch file

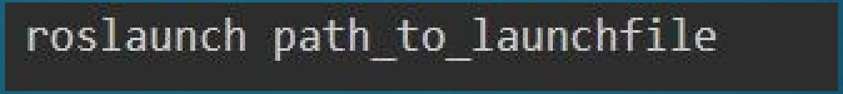
- Two ways to launch a script using roslaunch

1. Launch with package path：



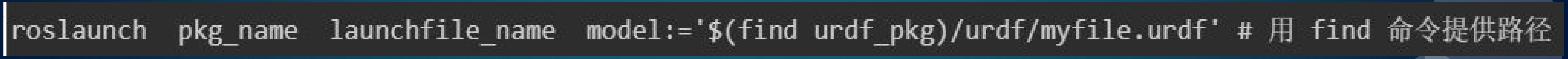
```
roslaunch  pkg_name  launchfile_name.launch
```

2. Directly launch if path is known



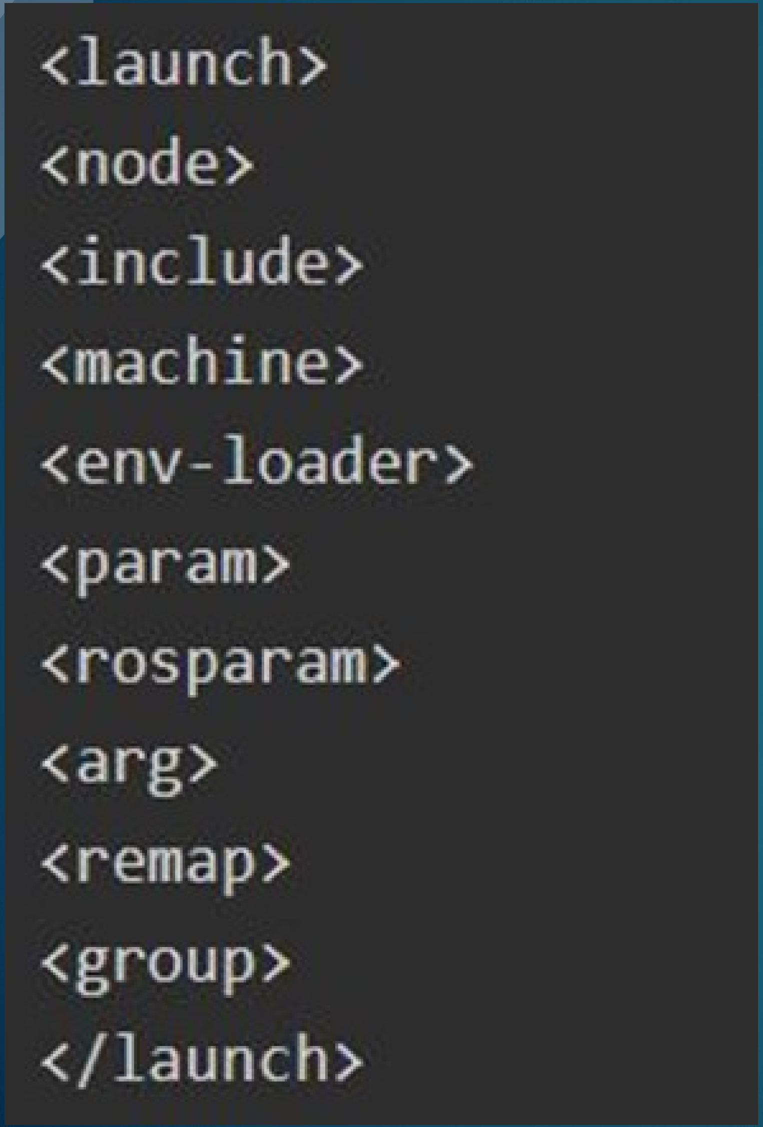
```
roslaunch path_to_launchfile
```

- Both ways can have command line parameters, ex.:



```
roslaunch  pkg_name  launchfile_name  model:='$(find urdf_pkg)/urdf/myfile.urdf' # 用 find 命令提供路径
```

Launch tag



```
<launch>
<node>
<include>
<machine>
<env-loader>
<param>
<rosparam>
<arg>
<remap>
<group>
</launch>
```

<!-- Launch file tag →
<!-- node launched and parameters -->

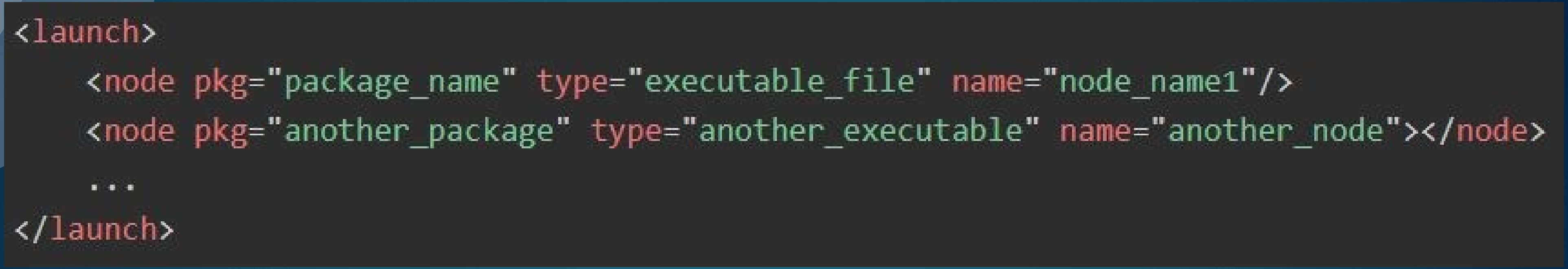
<!--include other launch files -->
<!--specify running machine -->
<!-- set environment variable -->

<!-- define parameters in param server -->
<!-- load params in yaml file to param server -->
<!--define variable -->

<!-- set topic mapping -->
<!-- set group -->
<!-- end of launch file -->

Launch tags

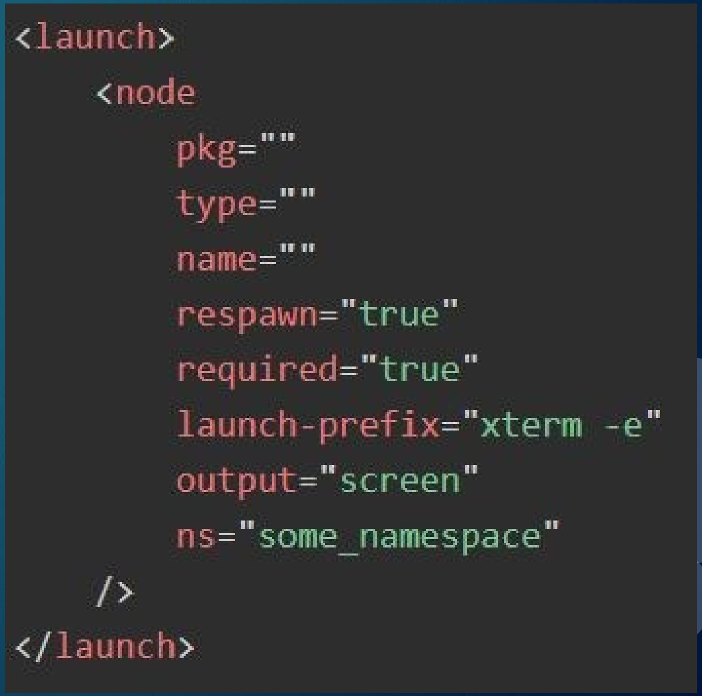
- <launch> 、<node>



```
<launch>
    <node pkg="package_name" type="executable_file" name="node_name1"/>
    <node pkg="another_package" type="another_executable" name="another_node"></node>
    ...
</launch>
```

- <node> has more tags apart from pkg、type、name

respawn: auto restart upon closure
required: close all other nodes upon this node closure
launch-prefix: if open a new window to run
output: the output of the node
ns: namespace of a node， i.e., add ns name before a nodec

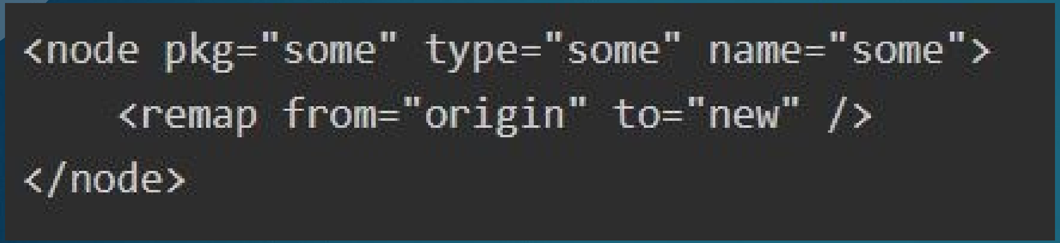


```
<launch>
    <node
        pkg=""
        type=""
        name=""
        respawn="true"
        required="true"
        launch-prefix="xterm -e"
        output="screen"
        ns="some_namespace"
    />
</launch>
```

Launch tags

- <remap>:

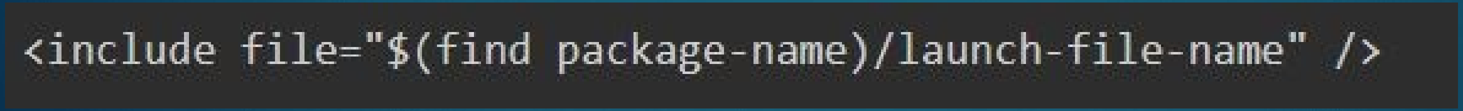
It is often used with node tag, to modify topic， so the same node can be used in different environment



```
<node pkg="some" type="some" name="some">
    <remap from="origin" to="new" />
</node>
```

- <include>

Inlcude another launch file, like embedding the launch files.



```
<include file="$(find package-name)/launch-file-name" />
```

Launch tags

- ◆ <arg>:

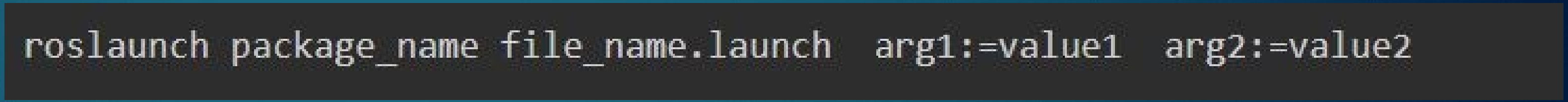
Reuse parameters by using <arg> tag， also facilitating modifications. Three common uses：

<arg name = “foo” > declare an arg, without setting its value， which can be set via command line, or <include> tag

<arg name = “foo” default = “1” > set default value, which can be overwritten by <include> tag

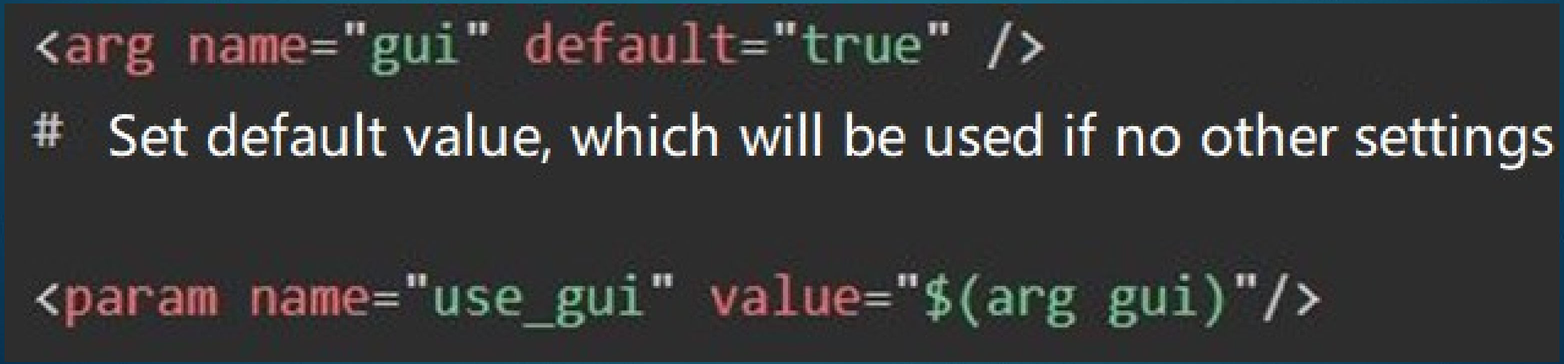
<arg name = “foo” value = “1” > set fixed value, which cannot be changed

Ex， set value via command line



```
roslaunch package_name file_name.launch  arg1:=value1  arg2:=value2
```

- ◆ $(arg arg_name) : this place is repaced with arg value specified by <arg> tag

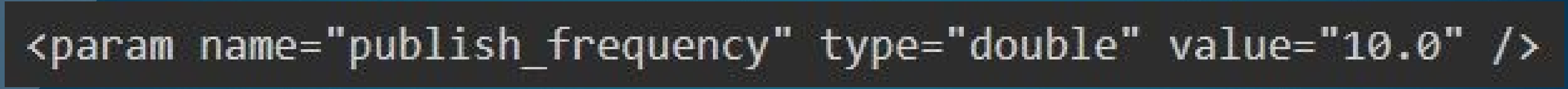


```
<arg name="gui" default="true" />
#  Set default value, which will be used if no other settings

<param name="use_gui" value="$(arg gui)"/>
```

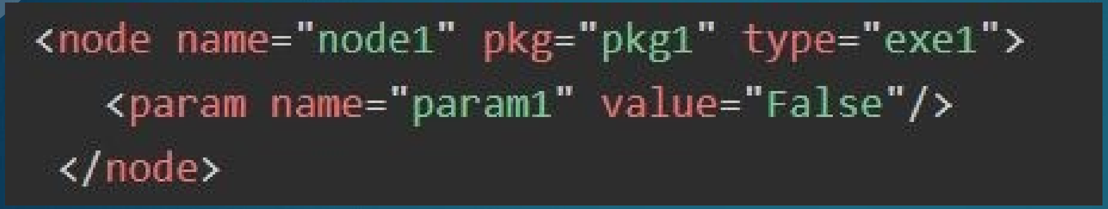
Launch tags

◆ <param>:

Different from arg ， param is sharable， it can set the parameters in parameter server：

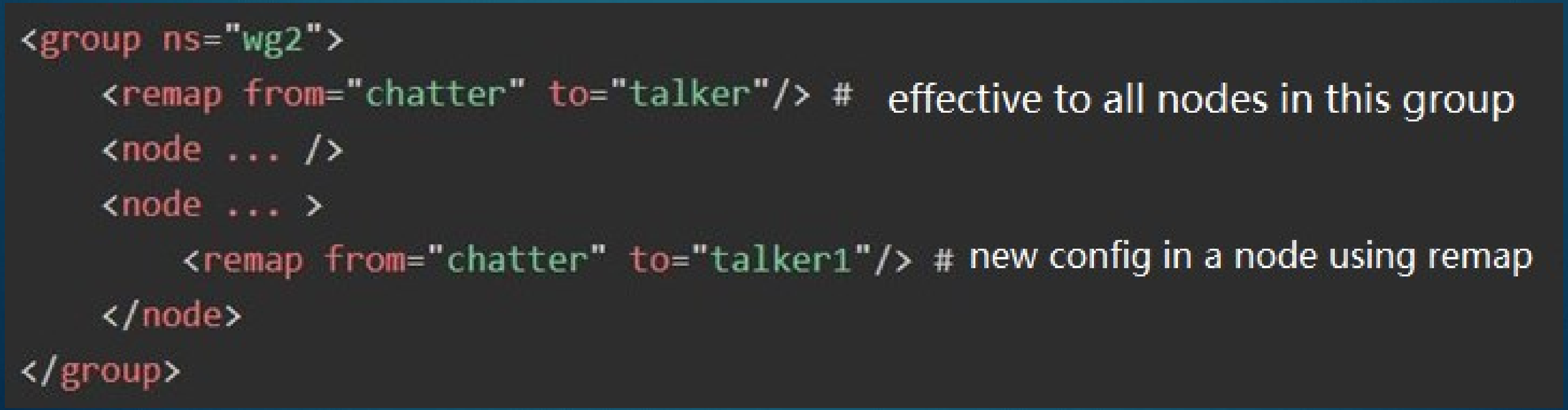


```
<param name="publish_frequency" type="double" value="10.0" />
```



```
<node name="node1" pkg="pkg1" type="exe1">
   <param name="param1" value="False"/>
 </node>
```

◆ <group> can set multiple nodes with same, or different configurations

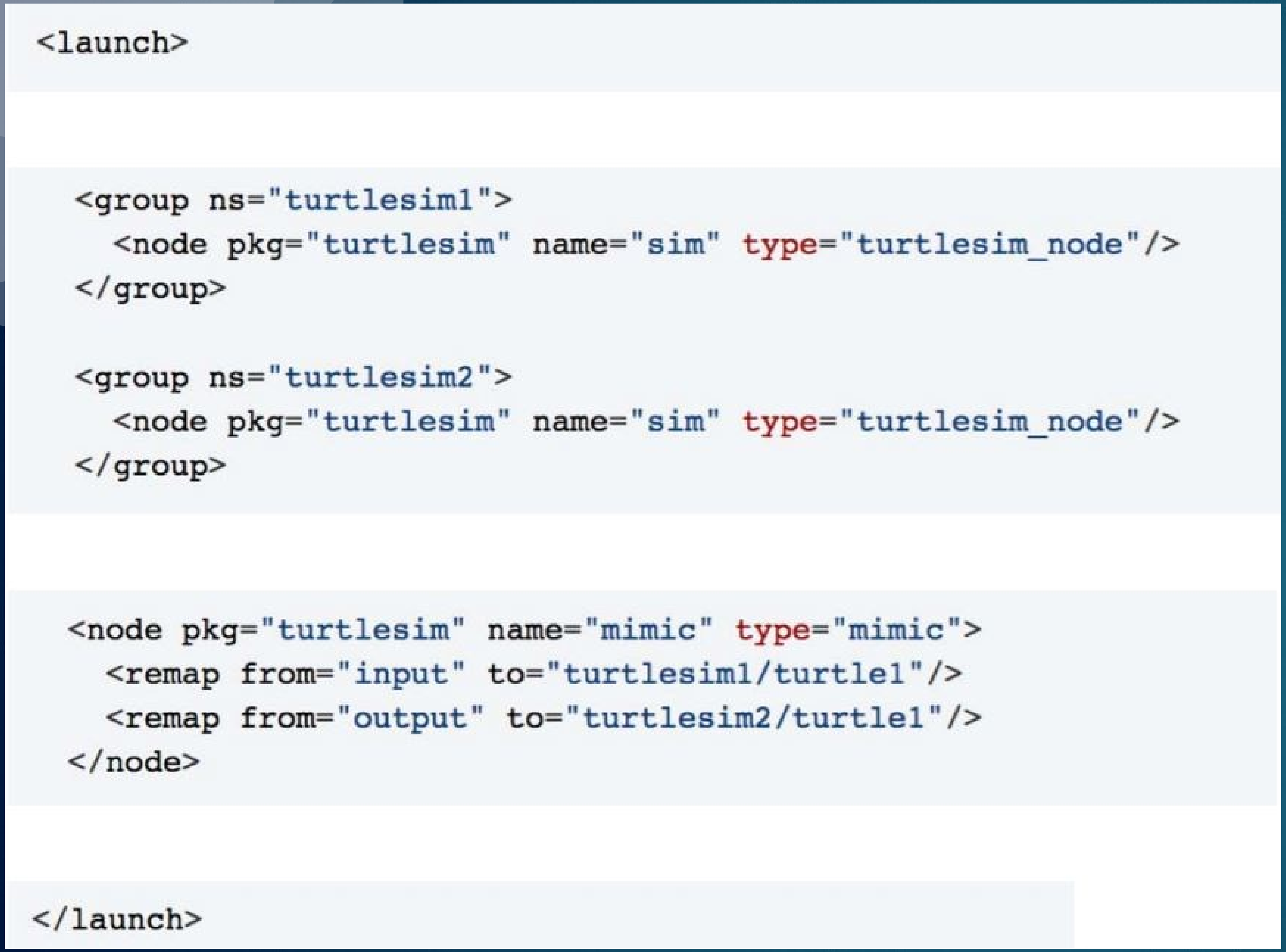


```
<group ns="wg2">
    <remap from="chatter" to="talker"/> #  effective to all nodes in this group
    <node ... />
    <node ... >
        <remap from="chatter" to="talker1"/> # new config in a node using remap
    </node>
</group>
```

Launch file references（examples）

http://wiki.ros.org/roslaunch/XML

Launch file examples （name： turtlemimic.launch）



```
<launch>


  <group ns="turtlesim1">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>

  <group ns="turtlesim2">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>


  <node pkg="turtlesim" name="mimic" type="mimic">
    <remap from="input" to="turtlesim1/turtle1"/>
    <remap from="output" to="turtlesim2/turtle1"/>
  </node>


</launch>
```

Launch file starts with XML tag

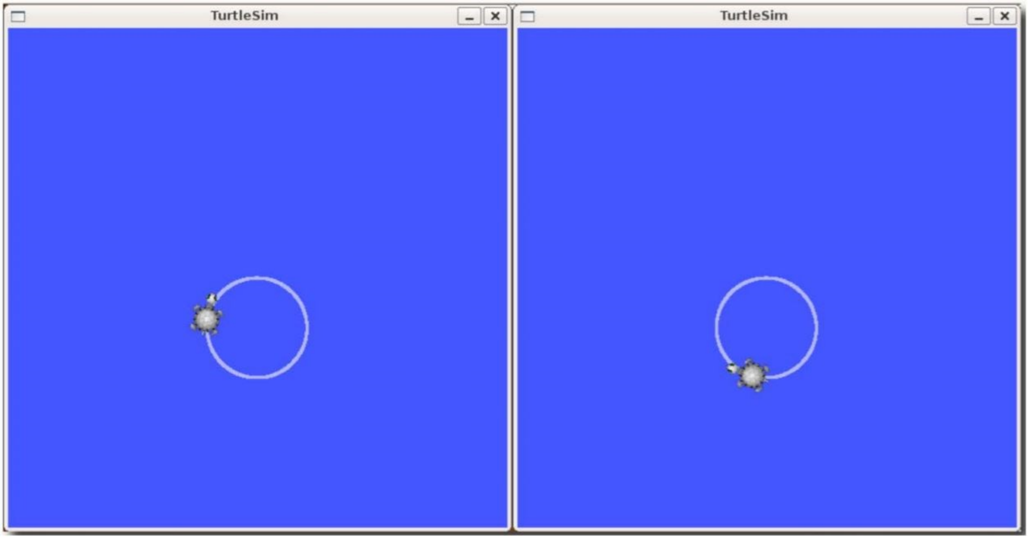
Start two simulators， with the names of
turtlesim1 and turtlesim2, respectively.
The NS of two simulators different, the nodes
In them have the same name. In this way, the
use of the same name is avoid.

Start node mimic， rename input and output
Topics as turtlesim1 and turtlesim2,
turtlesim2 traces turtlesim1

Launch file ends with XML tag

Run Launch File

```
$ roslaunch beginner_tutorials turtlemimic.launch
$ rostopic pub /turtlesim1/turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'
$ rqt_graph
```

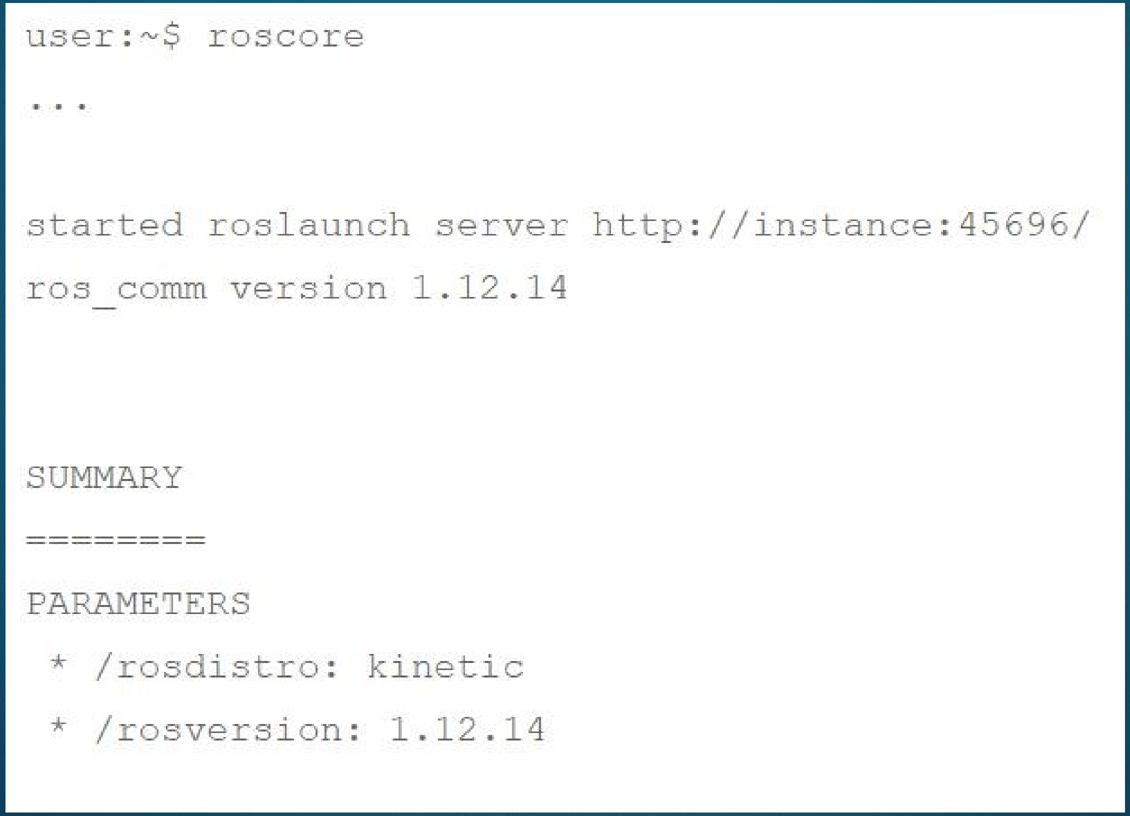


exercise： please modify turtlemimic.launch on the previous page， allow two turtles move in the same namespace

ROS Parameter Server

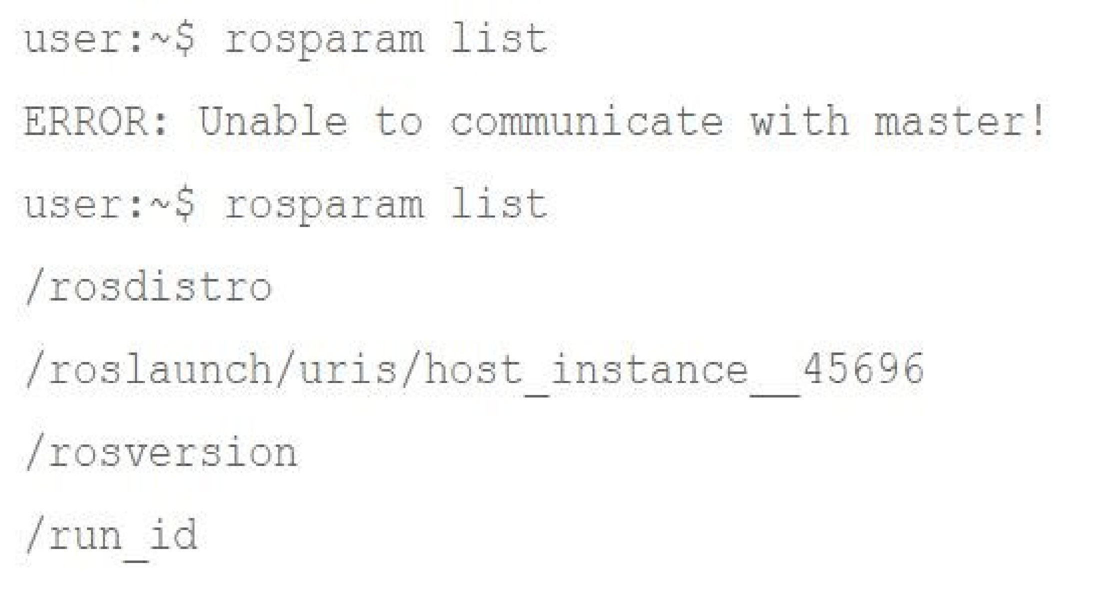
- ROS parameter server aims to save String, Integer, Floats, Binary, Dictionary and Lists

- ROS parameter server is a part of ROS Master， user need to launch ROS Master in order to use it

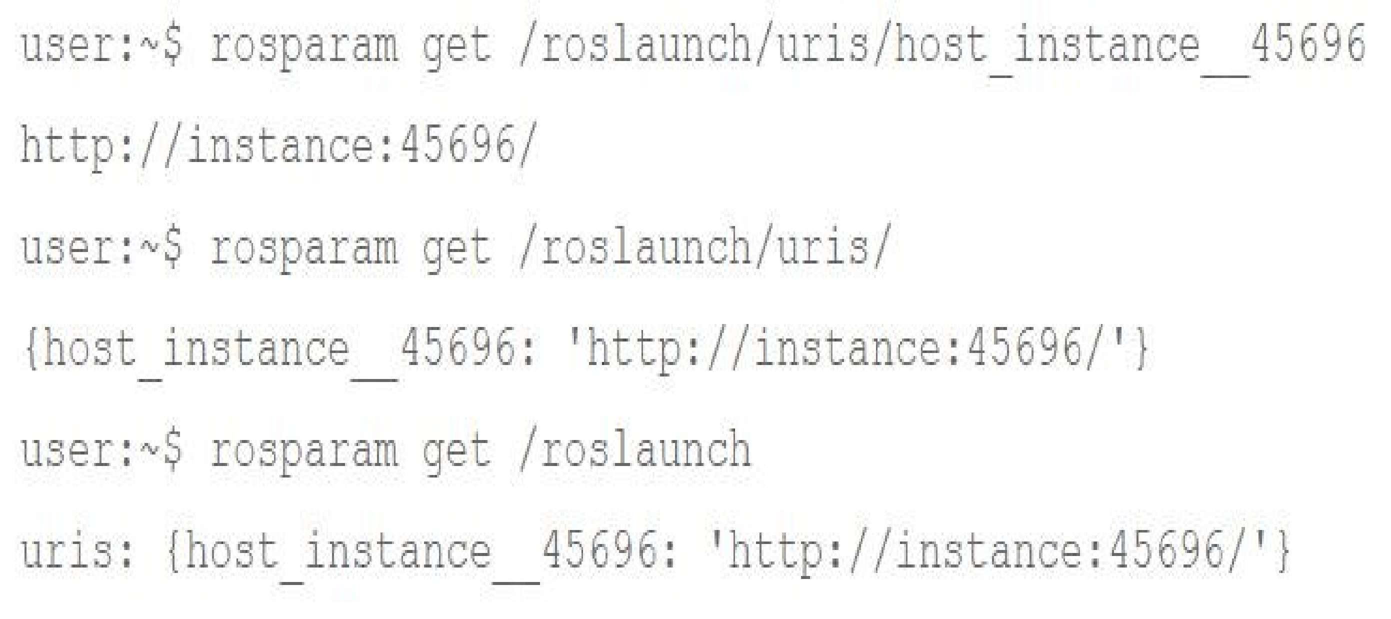


```
user:~$ roscore
...

started roslaunch server http://instance:45696/
ros_comm version 1.12.14


SUMMARY
========
PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.14
```

Manipulate parameters in ROS parameter server from command line



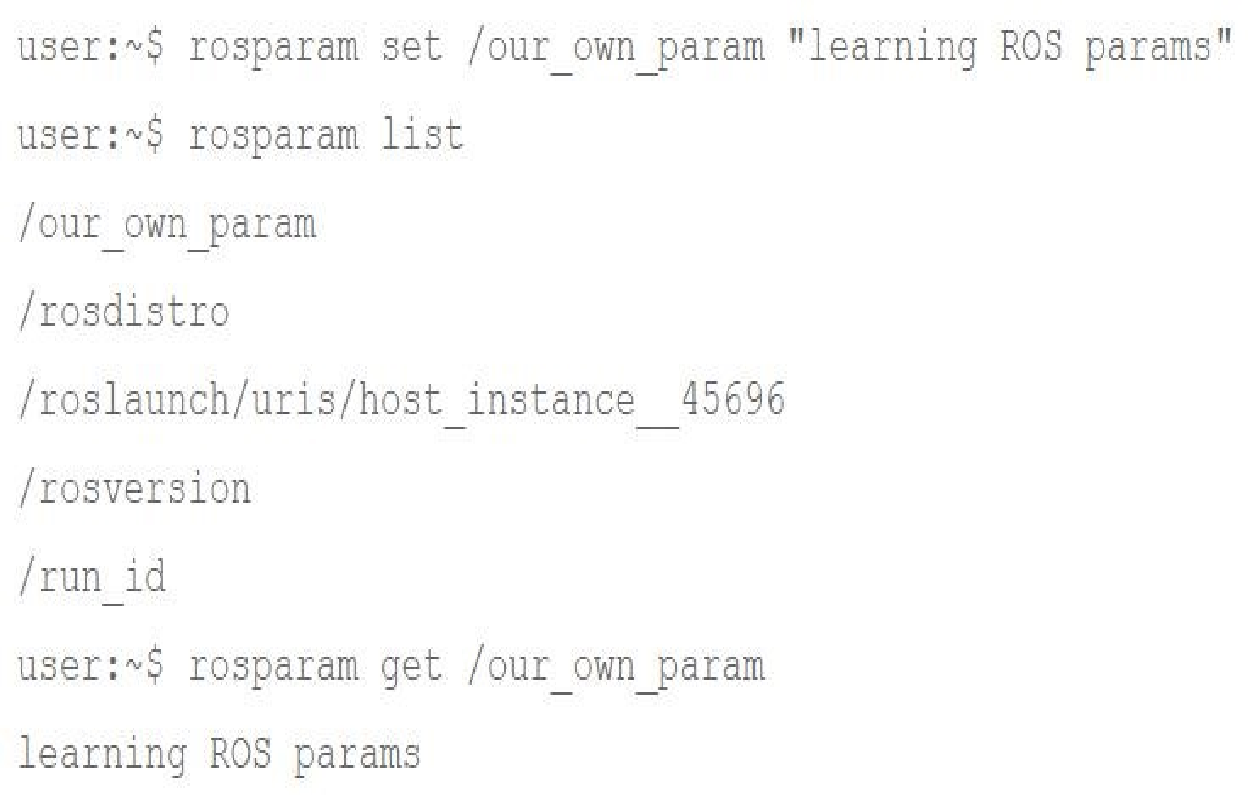
```
user:~$ rosparam list
ERROR: Unable to communicate with master!
user:~$ rosparam list
/rosdistro
/roslaunch/uris/host_instance__45696
/rosversion
/run_id
```



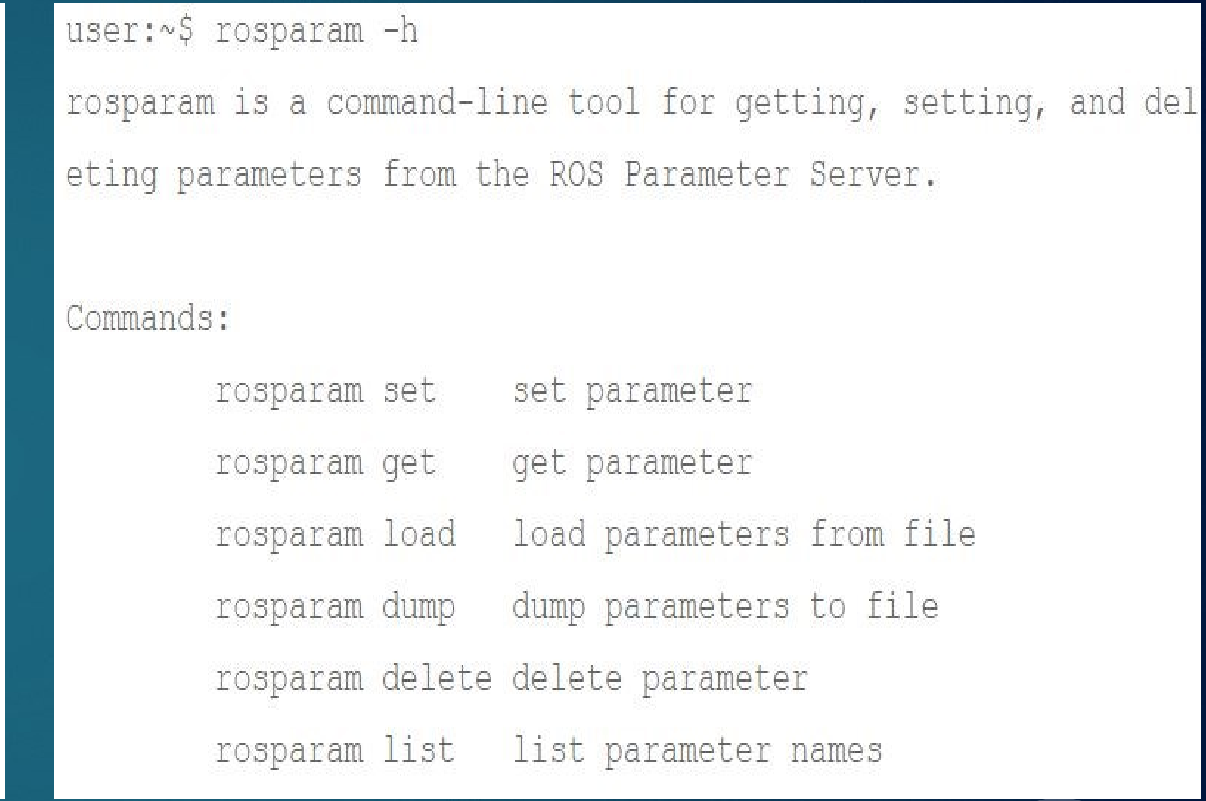
```
user:~$ rosparam get /roslaunch/uris/host_instance__45696
http://instance:45696/
user:~$ rosparam get /roslaunch/uris/
{host_instance__45696: 'http://instance:45696/'}
user:~$ rosparam get /roslaunch
uris: {host_instance__45696: 'http://instance:45696/'}
```

Like nodes, parameter has namespace to avoid collision of the same names.

Manipulate parameters in ROS parameter server from command line



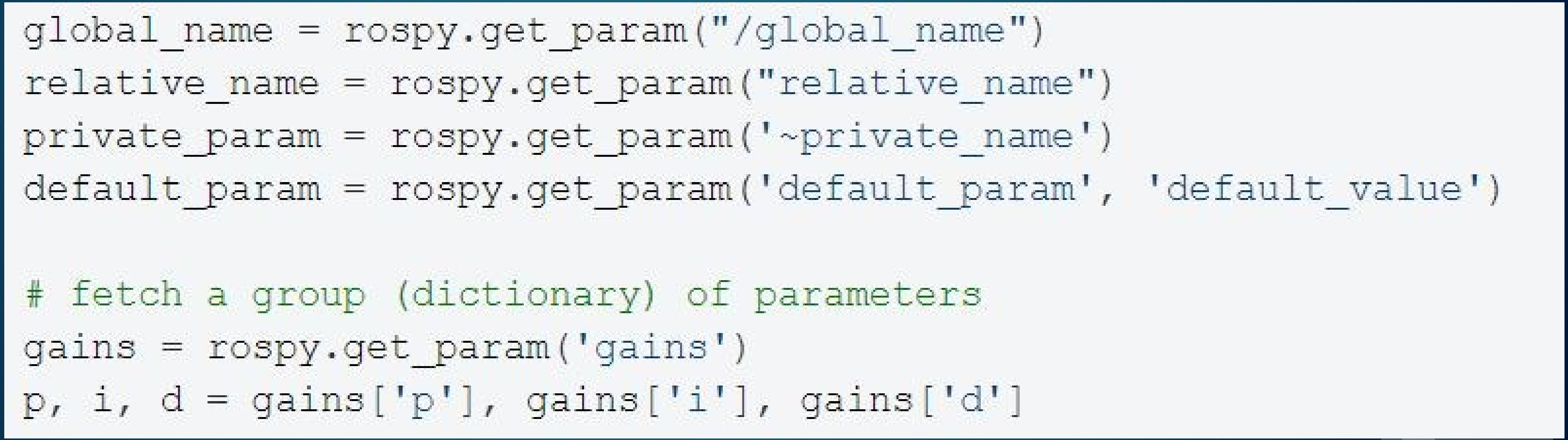
```
user:~$ rosparam set /our_own_param "learning ROS params"
user:~$ rosparam list
/our_own_param
/rosdistro
/roslaunch/uris/host_instance__45696
/rosversion
/run_id
user:~$ rosparam get /our_own_param
learning ROS params
```



```
user:~$ rosparam -h
rosparam is a command-line tool for getting, setting, and del
eting parameters from the ROS Parameter Server.

Commands:
        rosparam set    set parameter
        rosparam get    get parameter
        rosparam load   load parameters from file
        rosparam dump   dump parameters to file
        rosparam delete delete parameter
        rosparam list   list parameter names
```

ROS Parameter Server http://wiki.ros.org/rospy/Overview/Parameter%20Server

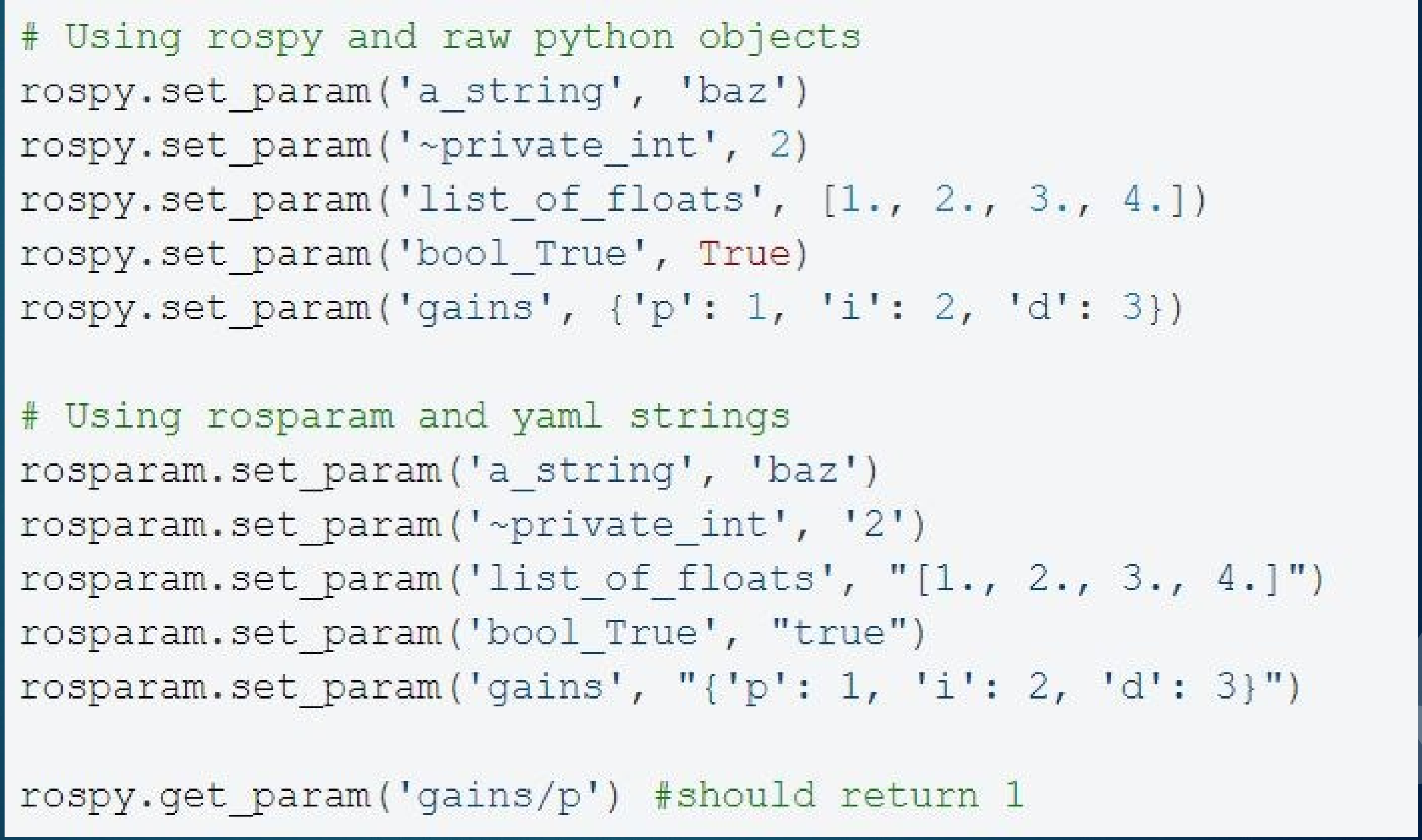
- ROS parameter server is used to save string, integer, float, binary, dictionary and lists.
- rospy API is used to access to the data in parameter server in codes（like using Python programming）
    - Obtain parameters： rospy.get_param(param_name)



```
global_name = rospy.get_param("/global_name")
relative_name = rospy.get_param("relative_name")
private_param = rospy.get_param('~private_name')
default_param = rospy.get_param('default_param', 'default_value')

# fetch a group (dictionary) of parameters
gains = rospy.get_param('gains')
p, i, d = gains['p'], gains['i'], gains['d']
```

ROS Parameter Server

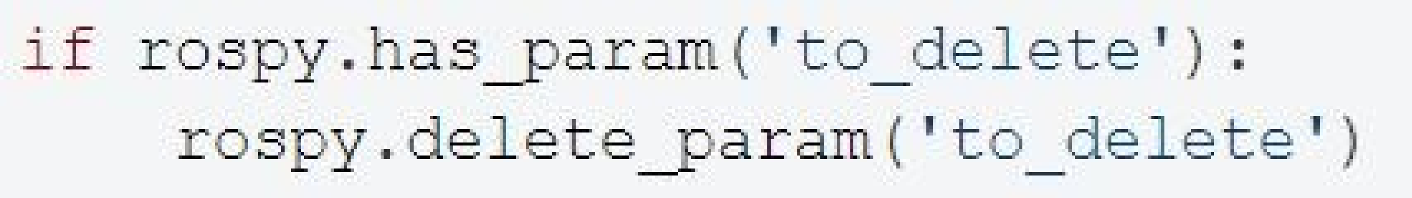
- Set parameters：rospy.set_param(param_name, param_value)



```
# Using rospy and raw python objects
rospy.set_param('a_string', 'baz')
rospy.set_param('~private_int', 2)
rospy.set_param('list_of_floats', [1., 2., 3., 4.])
rospy.set_param('bool_True', True)
rospy.set_param('gains', {'p': 1, 'i': 2, 'd': 3})

# Using rosparam and yaml strings
rosparam.set_param('a_string', 'baz')
rosparam.set_param('~private_int', '2')
rosparam.set_param('list_of_floats', "[1., 2., 3., 4.]")
rosparam.set_param('bool_True', "true")
rosparam.set_param('gains', "{'p': 1, 'i': 2, 'd': 3}")

rospy.get_param('gains/p') #should return 1
```

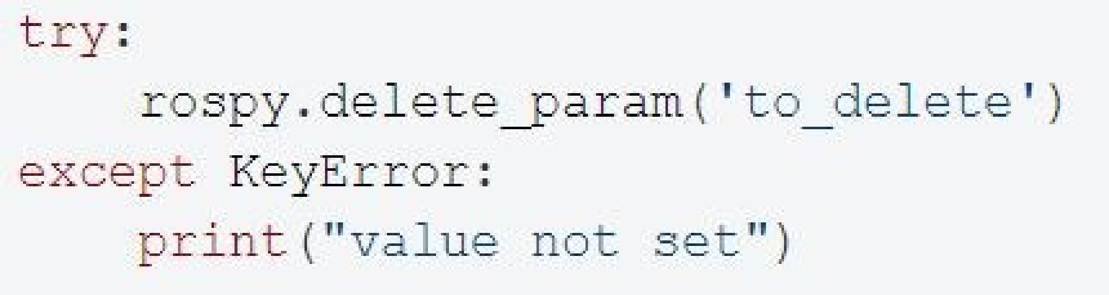
ROS Parameter Server

- Inquire if parameters existing in parameter server： rospy.has_param(param_name)



```
if rospy.has_param('to_delete'):
    rospy.delete_param('to_delete')
```

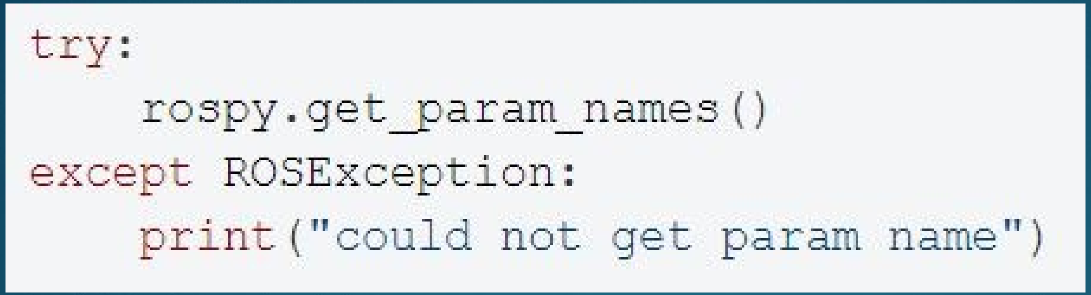
- Delete parameters in server： rospy.delete_param(param_name)



```
try:
    rospy.delete_param('to_delete')
except KeyError:
    print("value not set")
```

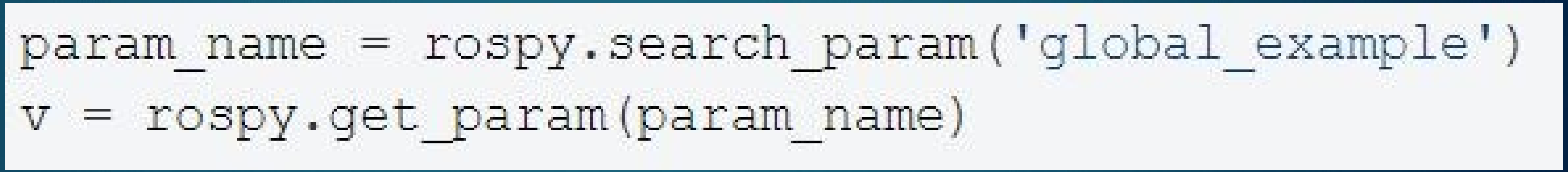
ROS Parameter Server

- **Obtain list of all parameters in a parameter server（display list of string names）**



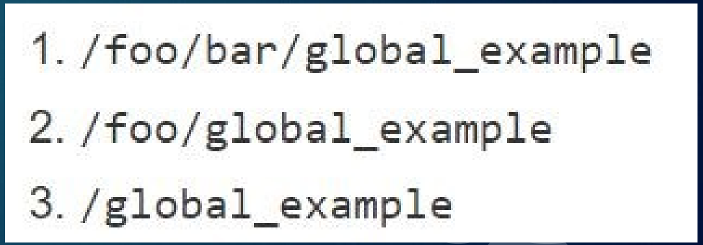
```
try:
    rospy.get_param_names()
except ROSException:
    print("could not get param name")
```

- **Search for parameters：rospy.search_param(param_name)**



```
param_name = rospy.search_param('global_example')
v = rospy.get_param(param_name)
```

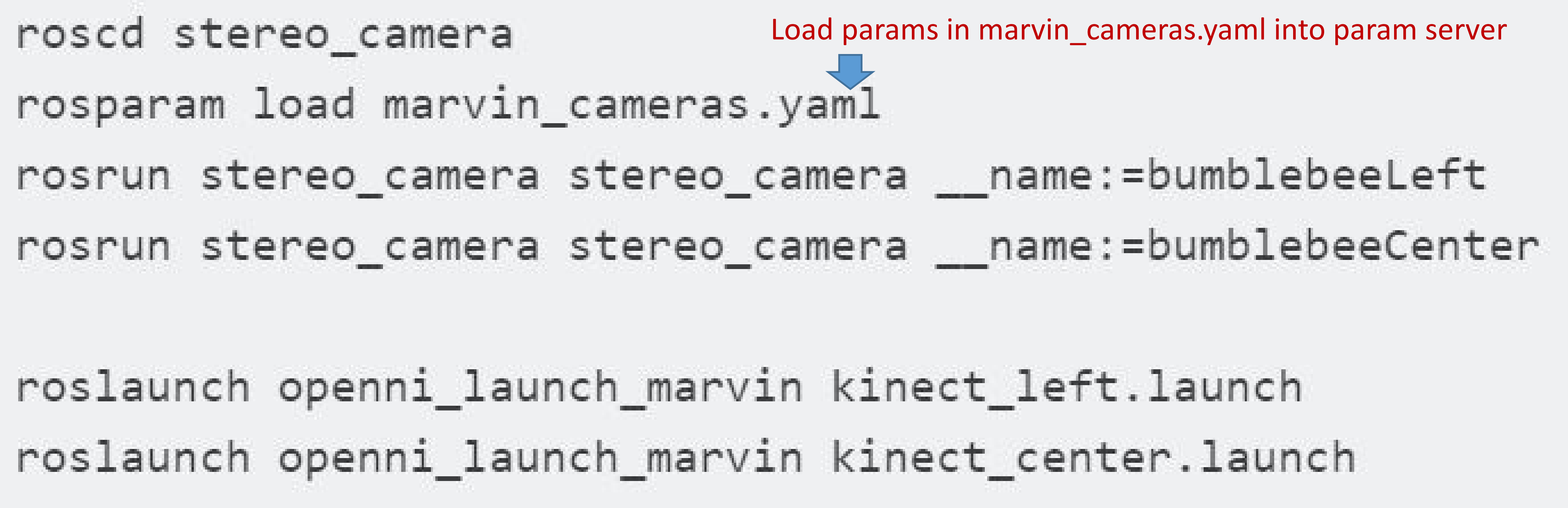
- If this code is in node /foo/bar，the order of search is private ns first, then global ns.



1. /foo/bar/global_example
2. /foo/global_example
3. /global_example

Roslaunch recap

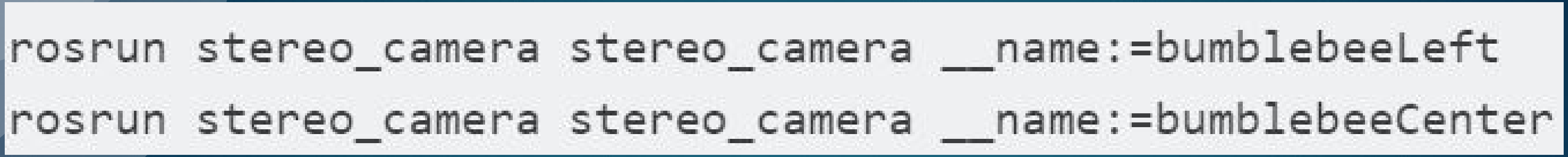
◆ If user wants to automatically run the following ROS command in order for a project, how to?



Load params in marvin_cameras.yaml into param server

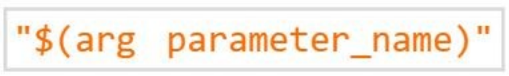
```
roscd stereo_camera
rosparam load marvin_cameras.yaml
rosrun stereo_camera stereo_camera __name:=bumblebeeLeft
rosrun stereo_camera stereo_camera __name:=bumblebeeCenter

roslaunch openni_launch_marvin kinect_left.launch
roslaunch openni_launch_marvin kinect_center.launch
```

roslaunch recap



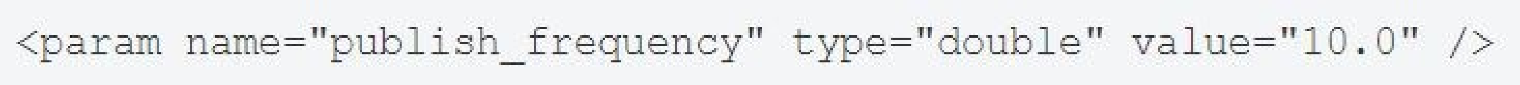
```
rosrun stereo_camera stereo_camera __name:=bumblebeeLeft
rosrun stereo_camera stereo_camera __name:=bumblebeeCenter
```

```
<launch>
   <node name="$(arg name)" pkg="stereo_camera" type="stereo_camera" output="screen">
        <param name="name" value="bumblebeeLeft" />
   </node>

   <node name="$(arg name)" pkg="stereo_camera" type="stereo_camera" output="screen">
        <param name="name" value="bumblebeeCenter" />
   </node>
</launch>
```



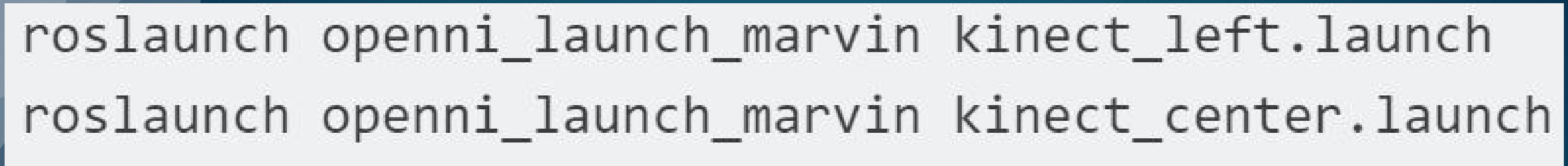
```
"$(arg parameter_name)"
```

Do not fix node (or param) name for flexibility



```
<param name="publish_frequency" type="double" value="10.0" />
```

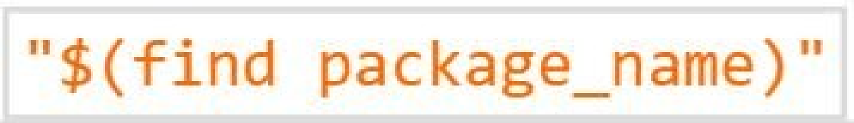
roslaunch recap



```
roslaunch openni_launch_marvin kinect_left.launch
roslaunch openni_launch_marvin kinect_center.launch
```

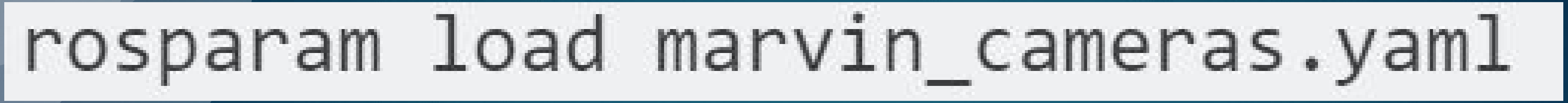
```
<include file="$(find openni_launch_marvin)/launch/kinect_left.launch" />
<include file="$(find openni_launch_marvin)/launch/kinect_center.launch" />
```

roscd →



```
"$(find package_name)"
```

roslaunch recap



```
rosparam load marvin_cameras.yaml
```

```
<rosparam command="load" file="$(find marvin_cameras)/config/marvin_cameras.yaml" />
```

```
<launch>

    <rosparam command="load" file="$(find marvin_cameras)/config/marvin_cameras.yaml" />

    <node name="$(arg name)" pkg="stereo_camera" type="stereo_camera" output="screen">
        <param name="name" value="bumblebeeLeft" />
    </node>

    <node name="$(arg name)" pkg="stereo_camera" type="stereo_camera" output="screen">
        <param name="name" value="bumblebeeCenter" />
    </node>

    <include file="$(find openni_launch_marvin)/launch/kinect_left.launch" />
    <include file="$(find openni_launch_marvin)/launch/kinect_center.launch" />

</launch>
```

ROS parameters recap

When you create a ROS master, a ROS parameter server is created. It contains a dictionary, accessible globally on the ROS environment.

A ROS parameter is basically just one of the shared variable stored in the parameter server.

A ROS parameter has a name, and a data type. Among the most common types, you can use:

- Boolean
- Integer number
- Double number
- String
- List of previous data types

Why do we need ROS parameters

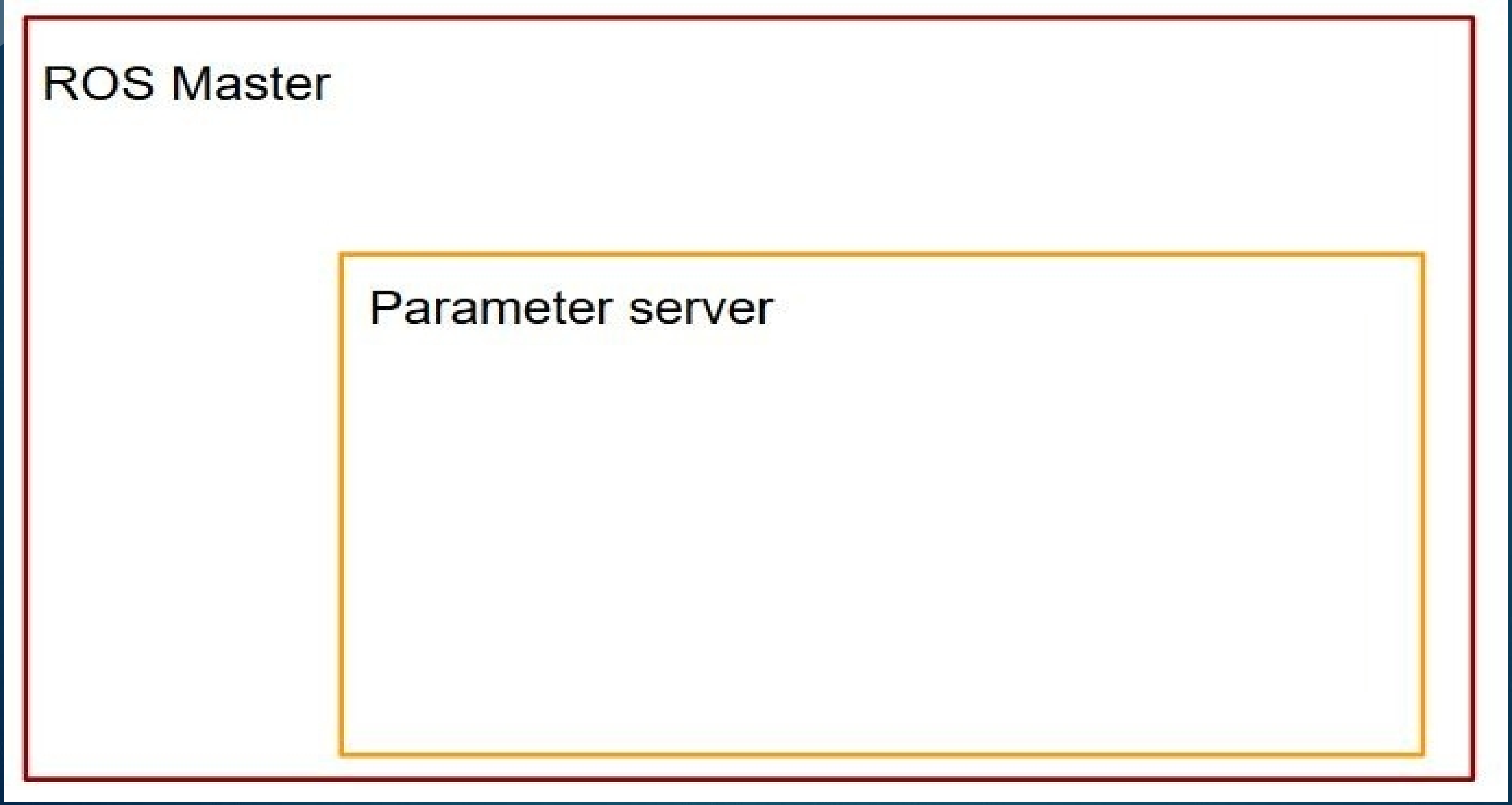
Suppose we have a nice robot application with many packages and nodes inside these packages. Now we want to create some global settings, e.g.,

-- The name of your robot
-- The frequency at which you read some sensors
-- A simulation flag in all your nodes informing the robot in real or simulaiton mode

We need a sort of global dictionary for shared parameters in our application, so that parameters can be retrieved at runtime, when we launch our nodes.

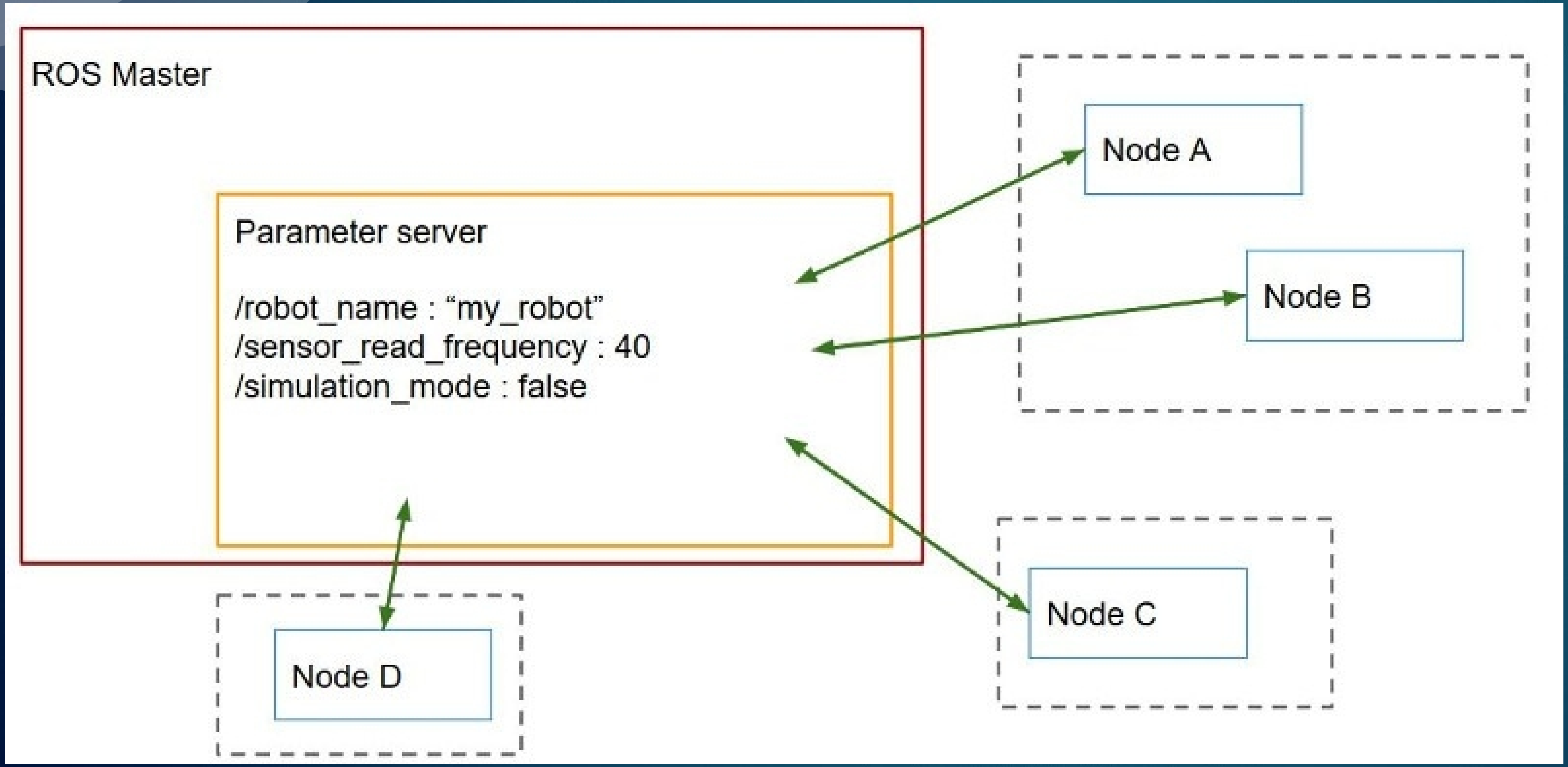
Why do we need ROS parameters

- **After you launch the ROS master, the parameter server is automatically created inside the ROS master**



Why do we need ROS parameters

- The parameter server is basically a dictionary containing global variable which are accessible from anywhere in the current ROS environment
- The global variables are called ROS parameters



Here I have set 3 parameters:

Robot name (string type)
Sensor read frequency (integer type)
Simulation mode flag (boolean type)

At any time, a node can read a parameter, modify a parameter, and can create new ones. A parameter created/modified by a node can be accessed by all other nodes.

Get and Set ROS Params with rospy

- **Set parameters**

```
1. Command line tool (useful for debugging)
$ rosparam set my_integer 7
$ rosparam set my_float 3.14
$ rosparam set my_string "hello"
$ rosparam list
/my_float
/my_integer
/my_string
...

2. launch file
<launch>
    <param name="my_integer" type="int" value="7" />
    <param name="my_float" type="double" value="3.14" />
    <param name="my_string" type="str" value="hello" />

    <node name="node_name" pkg="your_package" type="test_params.py" output="screen"/>
</launch>

3. rospy
rospy.set_param('/another_integer', 12)

4. In yaml file
See https://roboticsbackend.com/ros-param-yaml-format/
```

Get and Set ROS Params with rospy

◆ Get parameters with rospy

```
int_var = rospy.get_param("/my_integer")
float_var = rospy.get_param("/my_float")
string_var = rospy.get_param("/my_string")
rospy.loginfo("Int: %s, Float: %s, String: %s", int_var, float_var, string_var)
```

¡The rospy.get_param() function will return the corresponding value from the Parameter Server, that you can directly use or assign to a variable.

You can check if a parameter exists before accessing it:

```
if rospy.has_param('/my_integer'):
    rospy.get_param('/my_integer')
```

You can also use a default value if the parameter doesn’t exist:

```
str_var = rospy.get_param('/my_string', 'this is a default value')
```