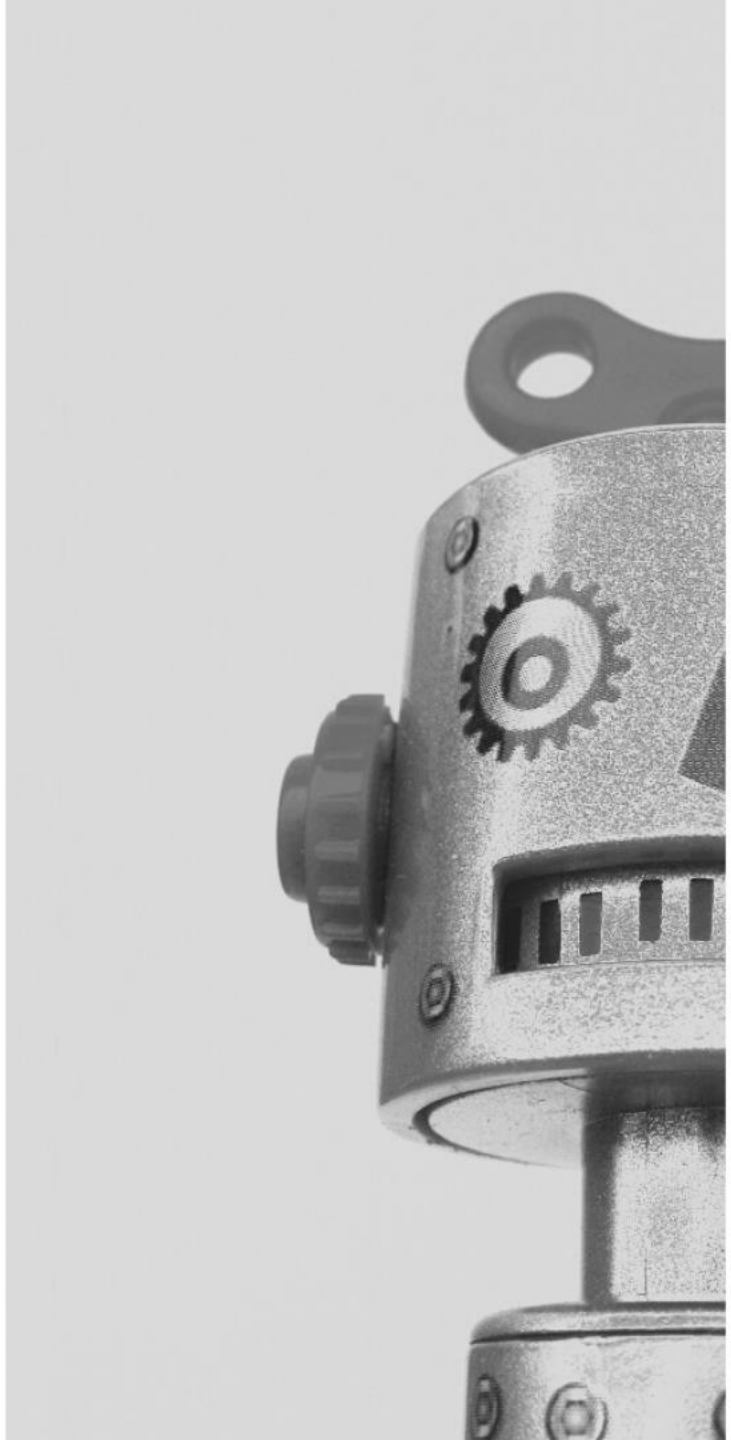
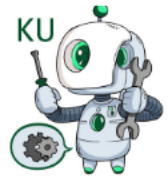


ROS

OVERVIEW

ROS

- Environment setup
- ROS Concept
- ROS Package / File system
- ROS Computation Graph level
  - Node
  - Topic
  - Service
- ROS Demo – Turtlesim
- Create workspace and package
- Writing a simple ROS node
  - Publisher / Subscriber
  - Service / Client
- Recording / Logging
  - Record and Playback data
  - Logging data
- Launch File

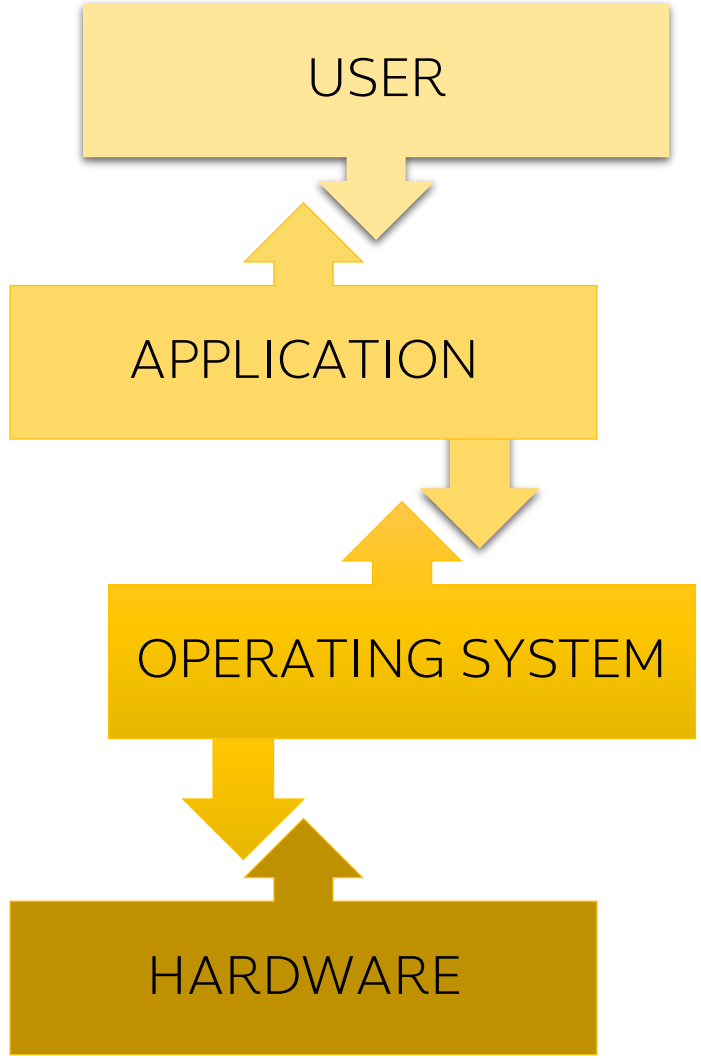


KU
KASETSART
UNIVERSITY

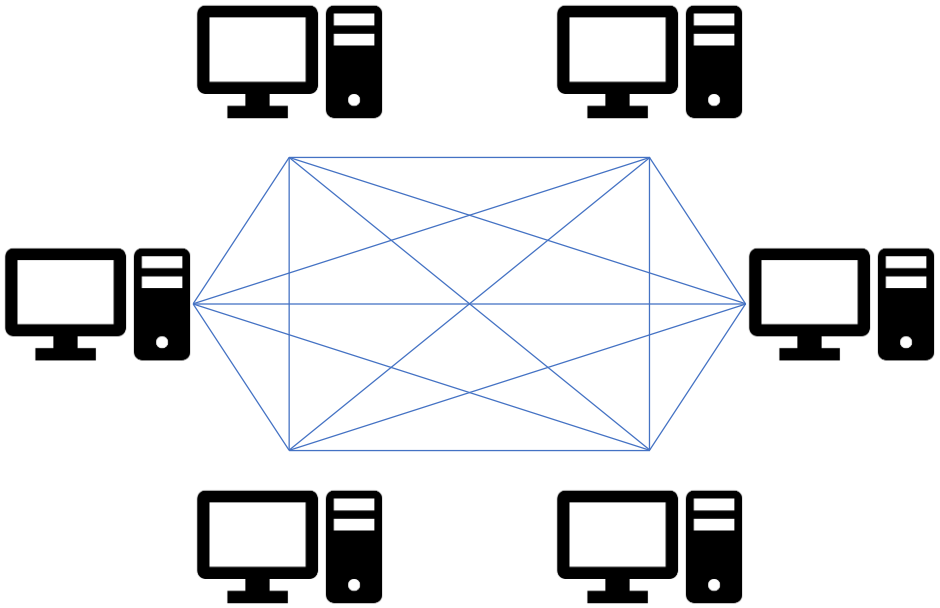
What is ROS

ROS

ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. ROS is similar in some respects to robot frameworks.



What is ROS



- **Peer to Peer**
  ROS systems consist of numerous small computer programs which connect to each other and continuously exchange messages
- **Tools-based**
  There are many small, generic programs that perform tasks such as visualization, logging, plotting data streams, etc.

What is ROS

- **Multi-Lingual**
  ROS software modules can be written in any language for which a client library has been written. Currently client libraries exist for C++, Python, LISP, Java, JavaScript, MATLAB, Ruby, and more.
- **Thin**
  The ROS conventions encourage contributors to create stand-alone libraries and then wrap those libraries, so they send and receive messages to/from other ROS modules.
- **Free and open source**

ROS Installation

- **Setup soures.list**

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

- **Setup Keys**

```
$ sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

- **Setup Keys**

```
$ sudo apt-get update
```

ROS Installation

- **Desktop-Full Install: (Recommended) : ROS, rqt, rviz, robot-generic libraries, 2D/3D simulators, navigation and 2D/3D perception**

```
$ sudo apt-get install ros-kinetic-desktop-full
```

- **To setup the ROS environment using .bashrc to automatically added to your bash session every time a new shell is launched**

```
$ echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

ROS Installation

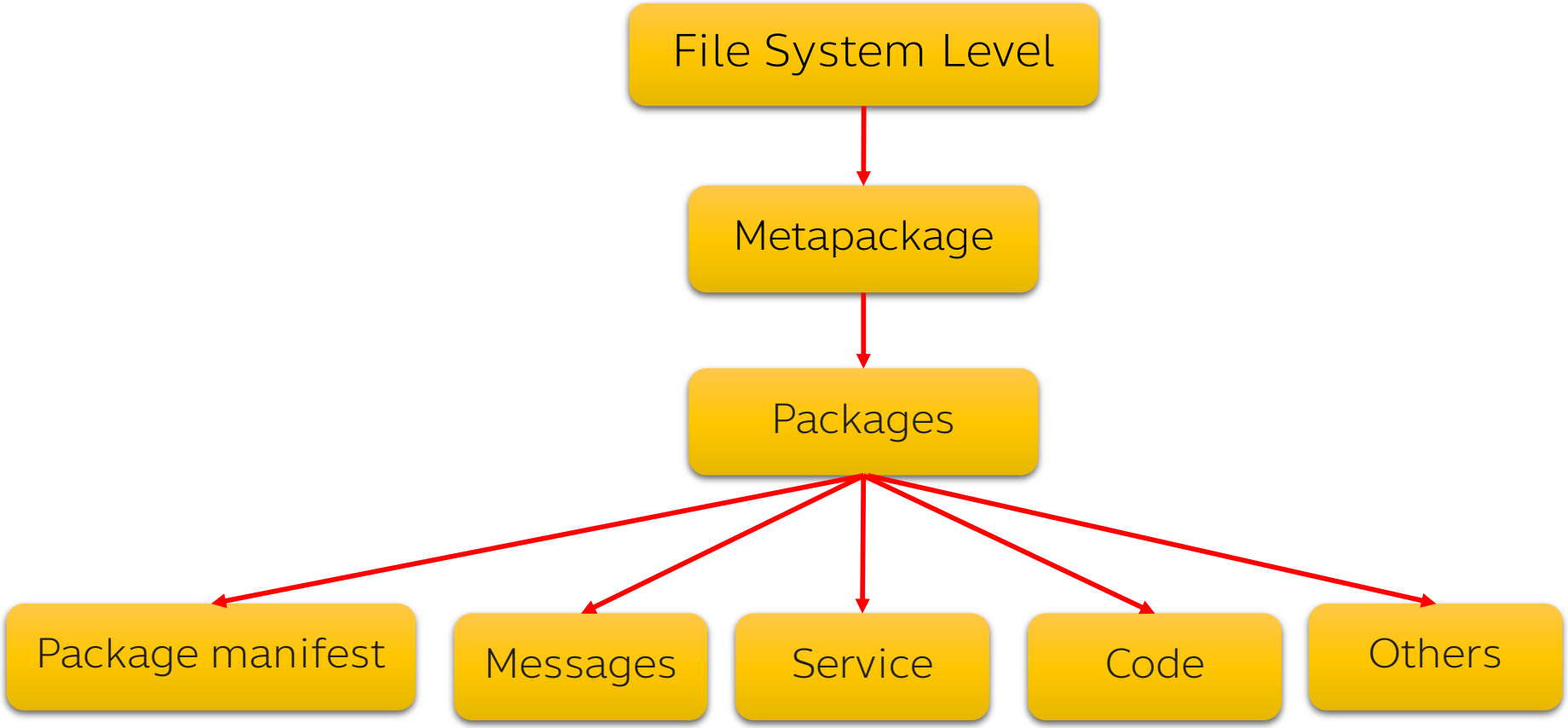
- **Install this tool and other dependencies for building ROS packages, run:**

```
$ sudo apt install python-rosdep python-rosinstall python-rosinstall-generator
python-wstool build-essential
```

- **Before using ROS tools, rosdep need to be initialized. It enables accessibility to dependencies installation to run some core components in ROS.**

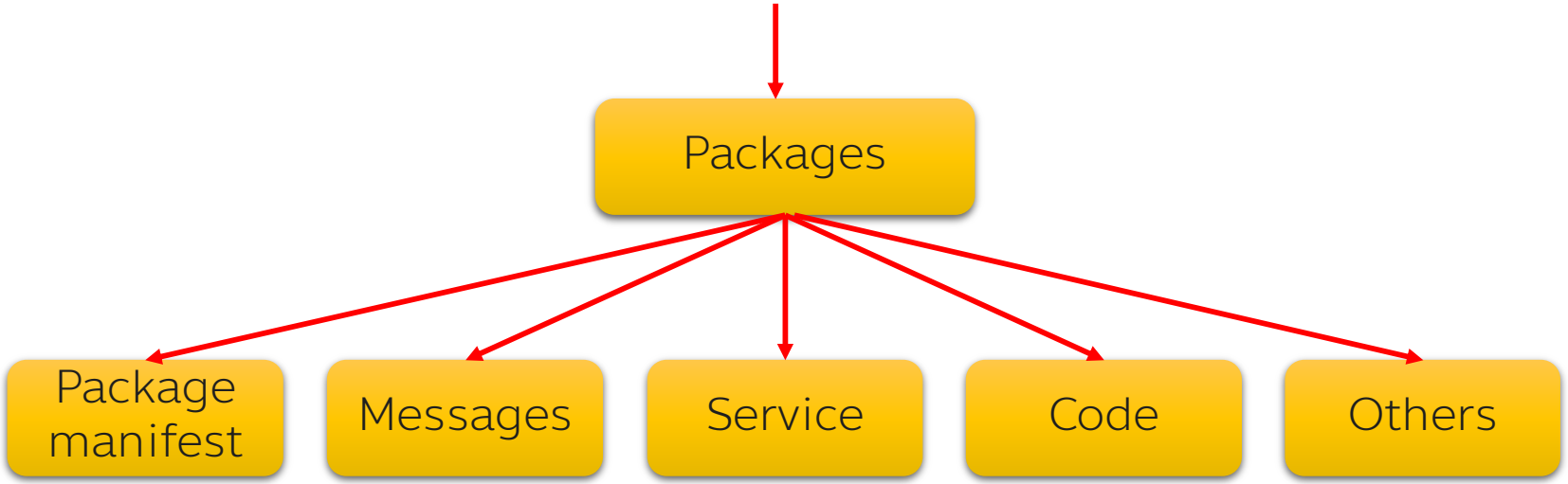
```
$ sudo rosdep init
$ rosdep update
```

ROS Package/Filesystem level



Like an operating system, an ROS program is divided into folders, and these
folders have files that describe their functionalities

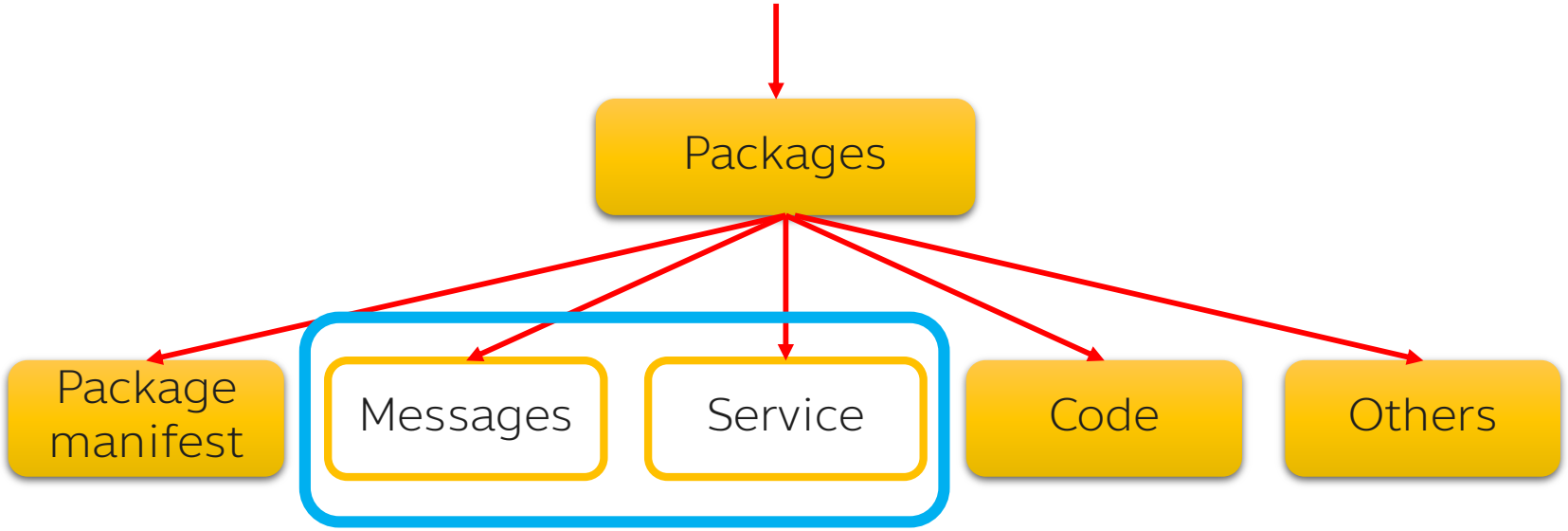
ROS Package/Filesystem level



• **Packages**: Packages from the atomic level of ROS filesystem.A package has the minimum structure and content to create a program within ROS.It may have ROS runtime processes (nodes), configuration files, and so on.

• **Package manifests**: Package manifests provide information about a package, licenses, dependencies, compilation flags, and so on. A package manifest is managed with a file called package.xml.

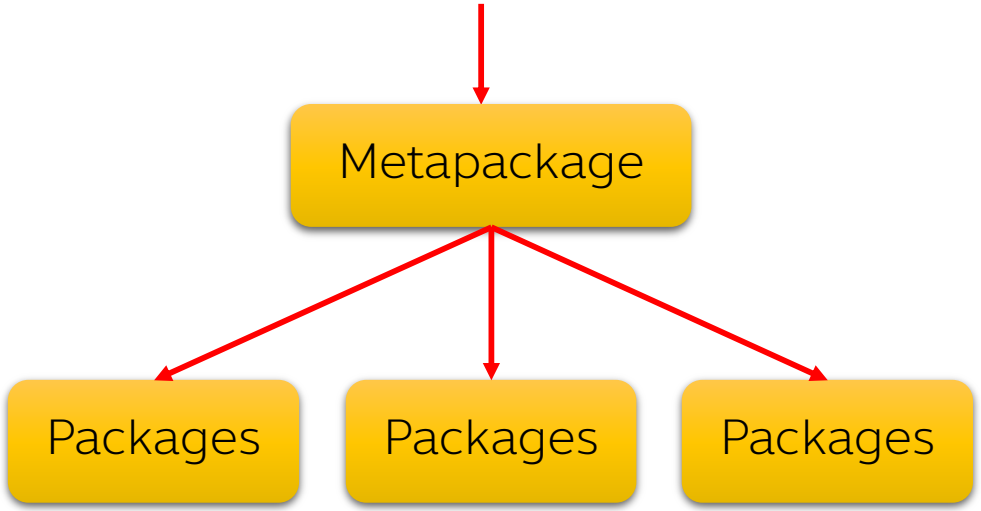
ROS Package/Filesystem level



• **Message (msg) types**: A message is the information that a process sends to other processes via Topic protocol. ROS has a lot of standard types of messages. Message descriptions are stored in msg folder.

• **Service (srv) types**: Service descriptions, stored in srv folder , define the request and response data structures for services provided by each process in ROS.

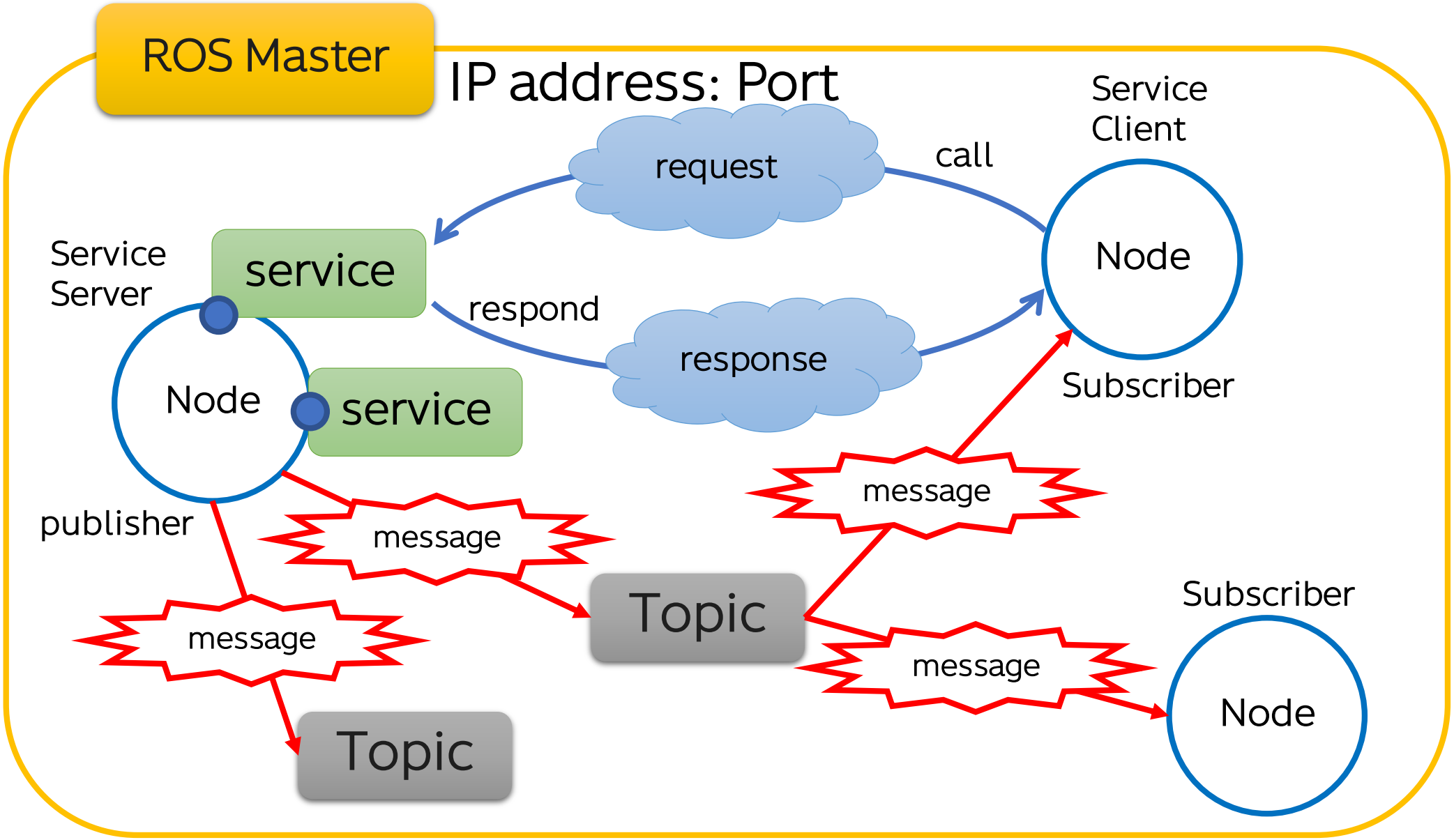
ROS Package/Filesystem level



- **Metapackages**: Metapackages are used in assembling packages into a group. In ROS, there exist a lot of these metapackages; one of them is the navigation stack.

- **Metapackage manifests**: Metapackage manifests (package.xml) are similar to a normal package but with an export tag in XML. It also has certain restrictions in its structure.

ROS Computation Level



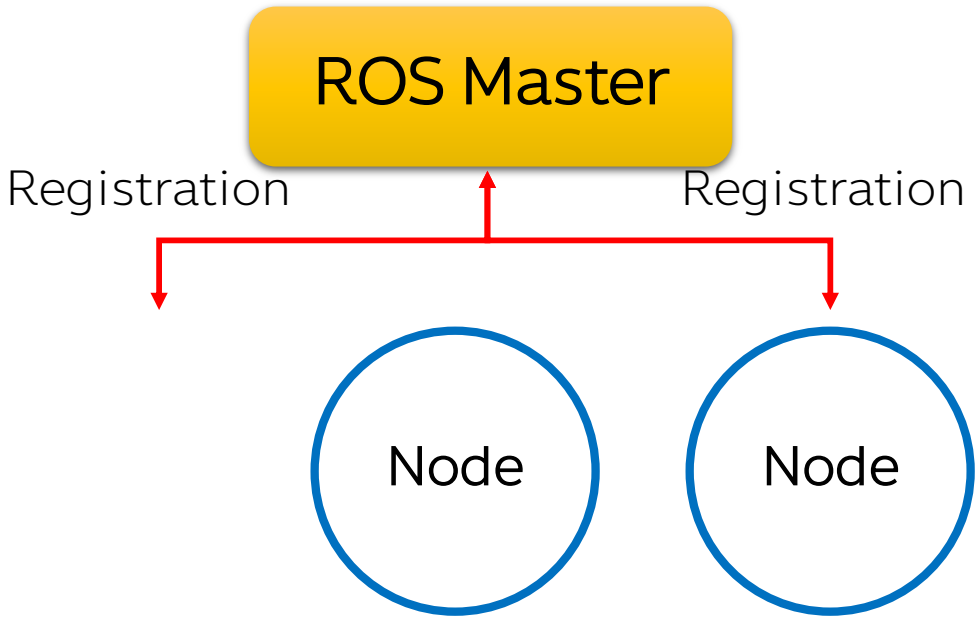
ROS Master

IP address: Port

ROS Master

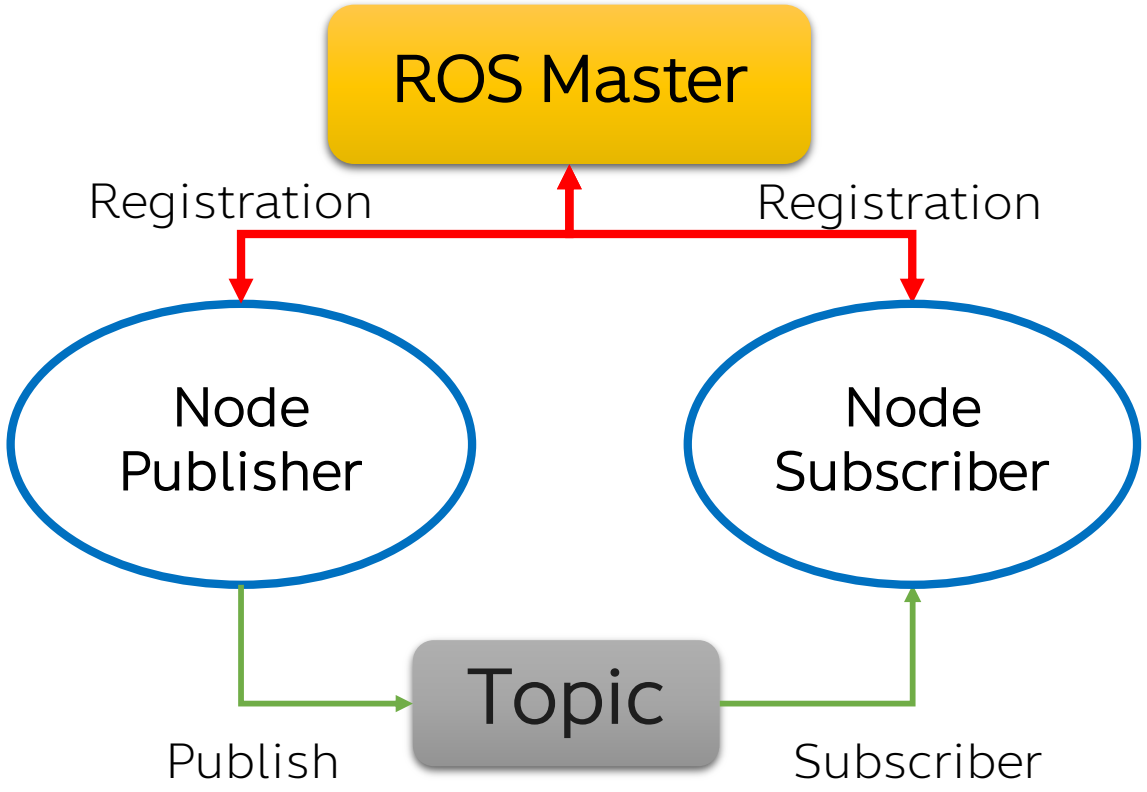
- The ROS Master provides naming and registration services to the rest of the nodes. The Master enables individual ROS node to locate one another which will communicate with each other peer-to-peer.
- The Master also provides the Parameter Server.
- The Master is commonly run using the roscore command, which loads the ROS Master along with other essential components.

ROS Node



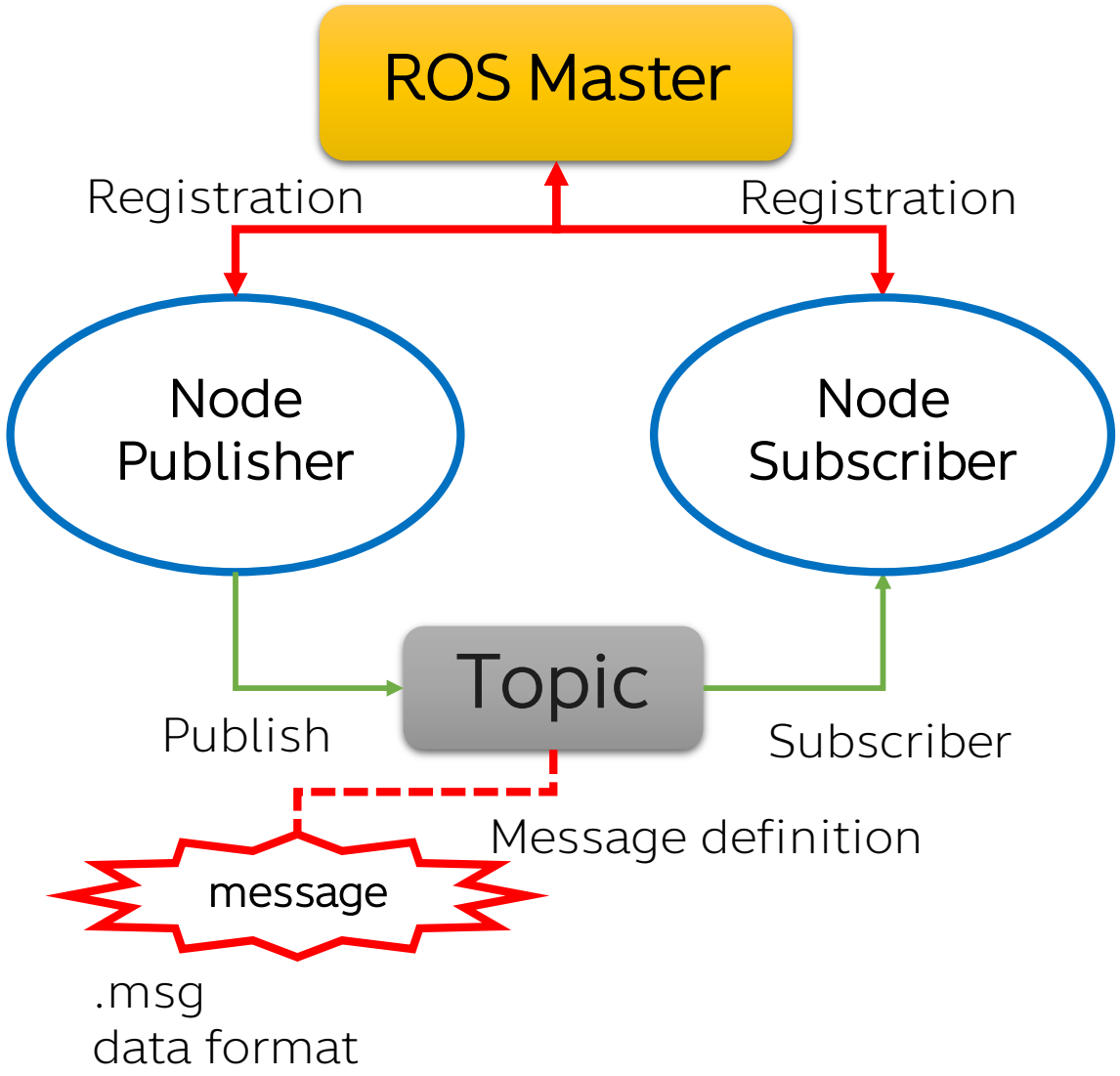
- Single-purposed executable programs
  e.g. sensor driver(s), actuator driver(s), mapper, planner, UI, etc.
- Individually compiled, executed, and managed
- Nodes are written using a ROS client library
  roscpp – C++ client library
  rospy – python client library
- Nodes can publish or subscribe to a Topic
- Nodes can also provide (server) or use (client) a Service

ROS Topic



- A topic is a name for a stream of messages with a defined type
  e.g., data from a laser range-finder might be sent on a topic called scan, with a message type of LaserScan
- Nodes communicate with each other by publishing messages to topics and subscribe to topic.
- Publish/Subscribe model: 1-to-N broadcasting

ROS Message



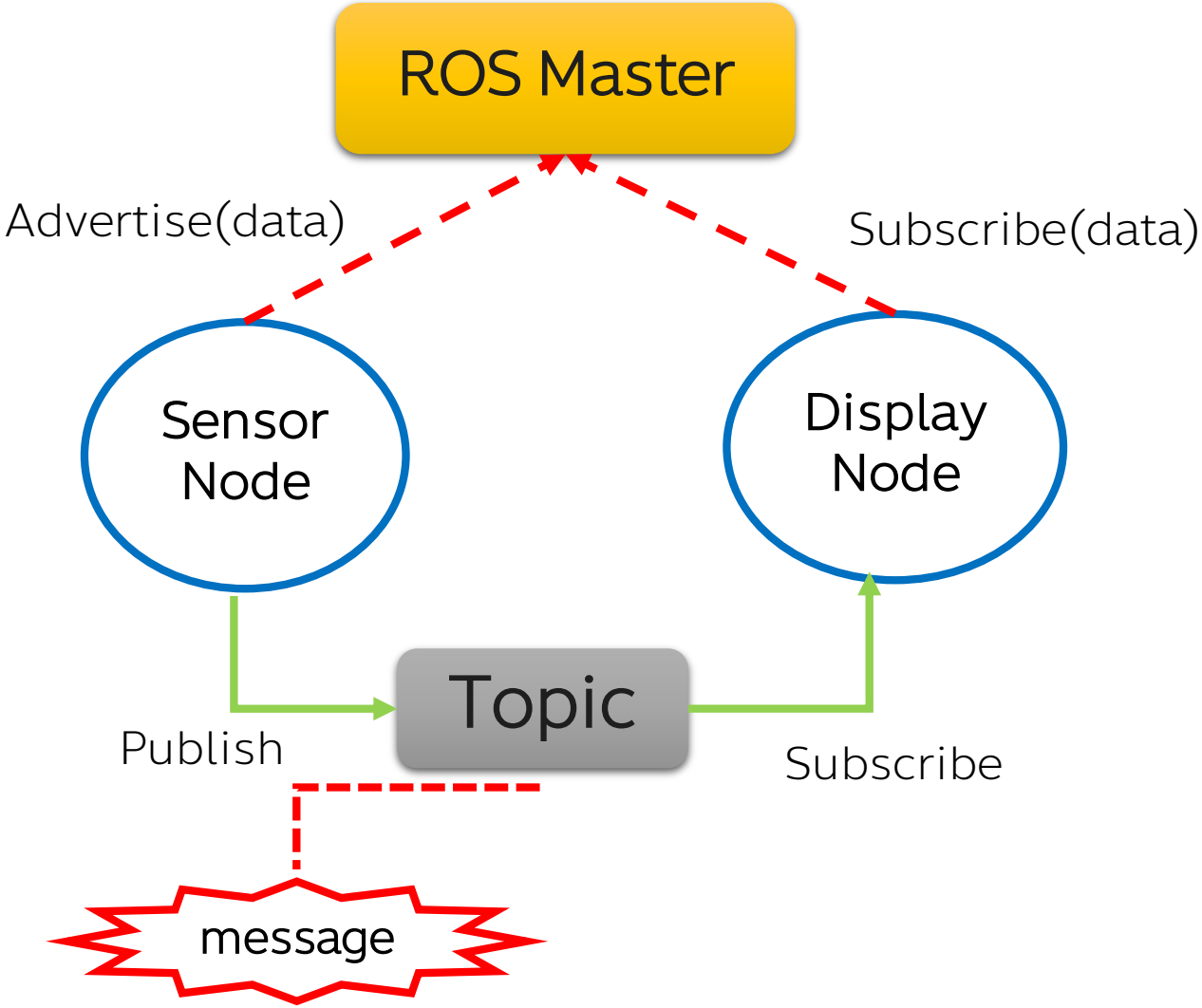
- A node sends information to another node using messages that are published by topics. The message has a simple structure that uses standard types or types developed by the user.
- For example, geometry_msgs/Twist is used to express velocity commands:

```
Vector3 linear
Vector3 angular
```

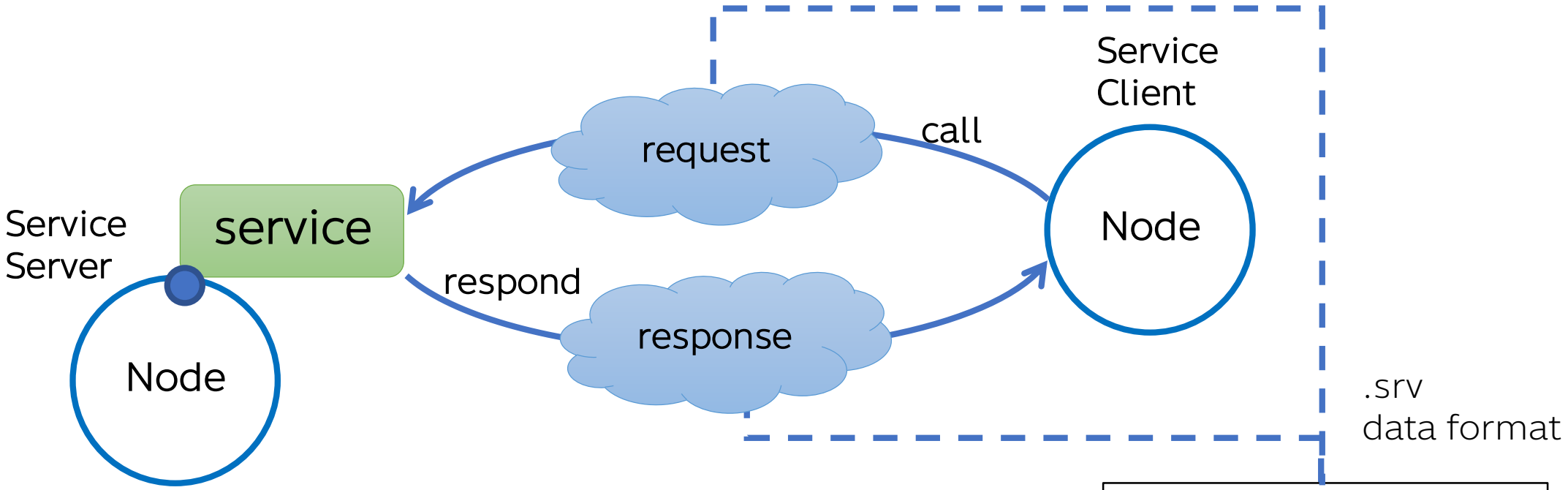
- Vector3 is another message type composed of:

```
float64 x
float64 y
float64 z
```

ROS Master Process

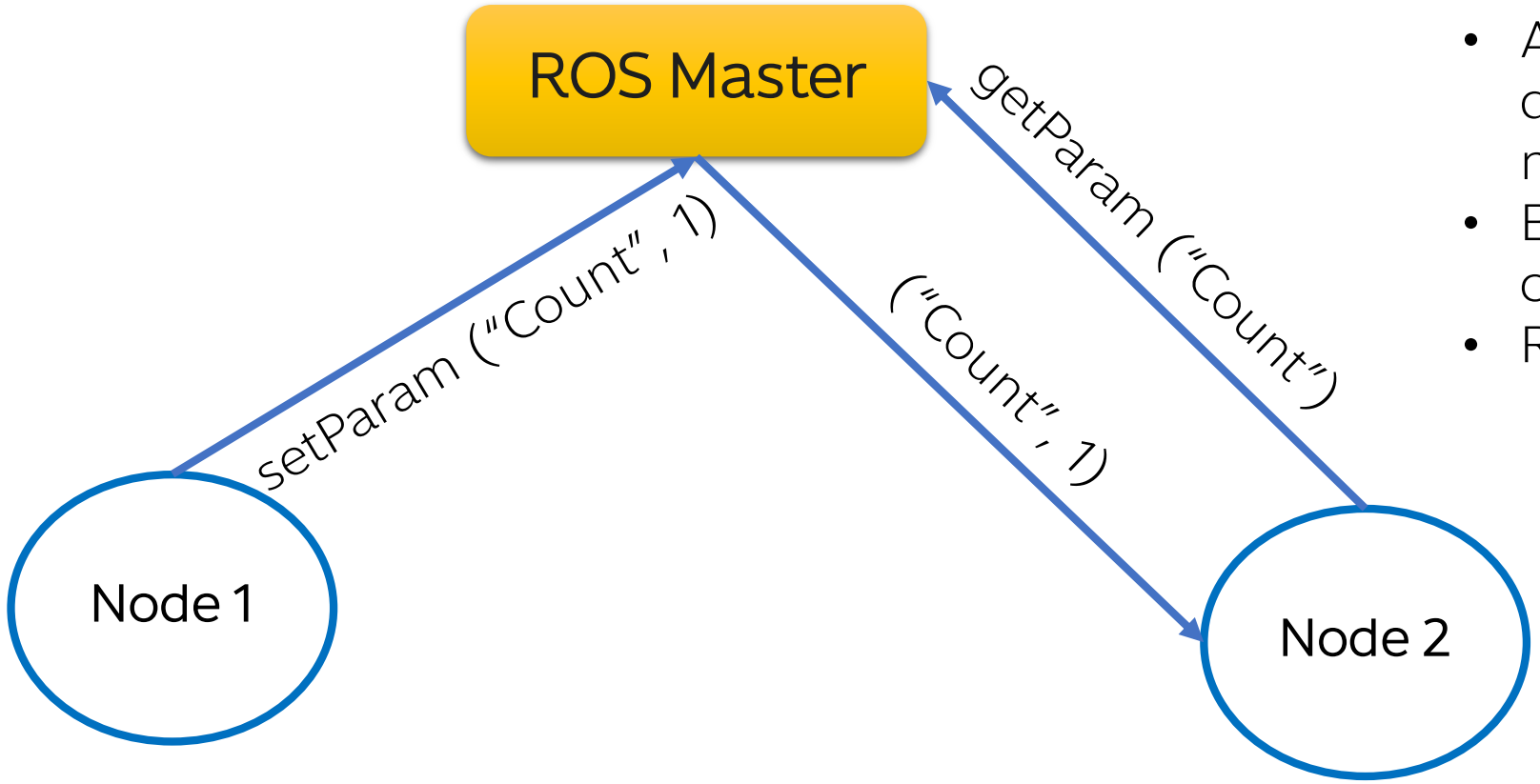


ROS Service



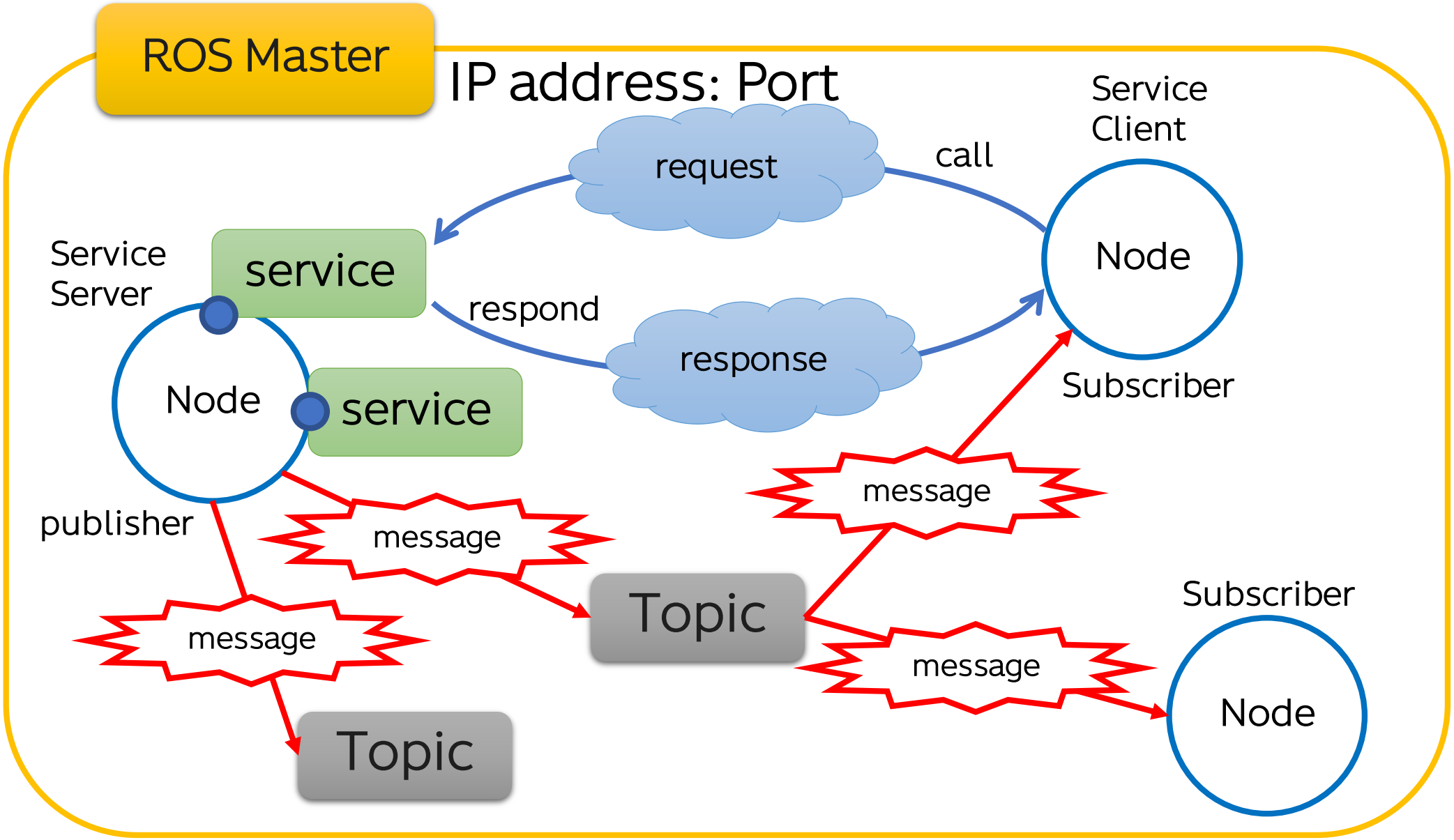
- Synchronous inter-node transactions / RPC
- Service/Client model: 1-to-1 request-response
- Service roles:
  carry out remote computation
  trigger functionality / behavior

ROS Parameter Server

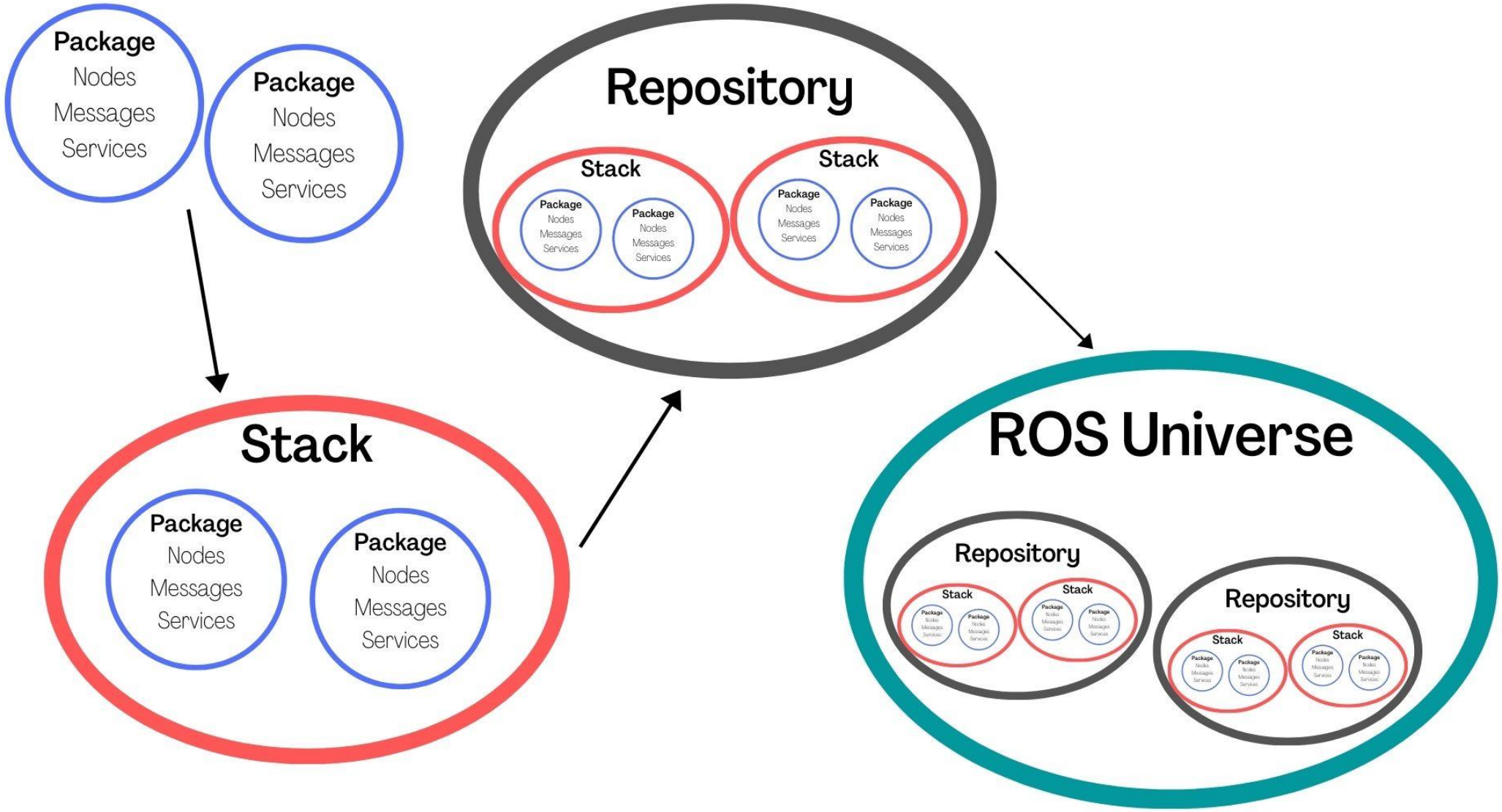


- A shared, multi-variate dictionary that is accessible via network APIs
- Best used for static data such as configuration parameters
- Runs inside the ROS master

ROS Computation Level



ROS Universe



ROS Bash command

ROS package bash command

- **rospack**: This command is used to get information or find packages in the system.
- **catkin_create_pkg**: This command is used when you want to create a new package.
- **catkin_make**: This command is used to compile a workspace.
- **rosdep**: This command installs the system dependencies of a package.
- **roscd**: This command helps us change the directory. This is similar to the cd command in Linux.

ROS Bash command

ROS node bash command

- **rosnode info *NODE*** : This prints information about a node
- **rosnode kill *NODE*** : This kills a running node or sends a given signal
- **rosnode list**: This lists the active nodes
- **rosnode *machine hostname*** : This lists the nodes running on a particular machine or lists machines
- **rosnode ping *NODE*** : This tests the connectivity to the node.
- **rosnode cleanup**: This purges the registration information from unreachable nodes

ROS Bash command

ROS topic bash command

- **rostopic bw *<u>/topic</u>*** : This displays the bandwidth used by the topic.
- **rostopic echo *<u>/topic</u>*** : This prints messages to the screen.
- **rostopic find *<u>message_type</u>*** : This finds topics by their type.
- **rostopic hz *<u>/topic</u>*** : This displays the publishing rate of the topic.
- **rostopic info *<u>/topic</u>*** : This prints information about the active topic, topics published, ones it is subscribed to, and services.

ROS Bash command

ROS topic bash command

- **rostopic list** : This prints information about active topics.

- **rostopic pub */topic type args*** : This publishes data to the topic. It allows us to create and publish data in whatever topic we want, directly from the command line.

- **rostopic type */topic*** : This prints the topic type, that is, the type of the message it publishes.

ROS Bash command

ROS message bash command

- **rosmsg show** : This displays the fields of a message
- **rosmsg list** : This lists all messages
- **rosmsg package** : This lists all of the messages in a package
- **rosmsg packages** : This lists all of the packages that have the message
- **rosmsg users** : This searches for code files that use the message type
- **rosmsg md5** : This displays the MD5 sum of a message

ROS Bash command

ROS service bash command

- **rosservice call** ***/service args*** : This calls the service with the arguments provided
- **rosservice find** ***msg-type*** : This finds services by service type
- **rosservice info** ***/service*** : This prints information about the service
- **rosservice list** : This lists the active services
- **rosservice type** ***/service*** : This prints the service type
- **rosservice uri** ***/service*** : This prints the ROSRPC URI service

ROS Bash command

ROS param bash command

- **rosparam list** : This lists all the parameters in the server
- **rosparam get *parameter*** : This gets the value of a parameter
- **rosparam set *parameter value*** : This sets the value of a parameter
- **rosparam delete *parameter*** : This deletes a parameter
- **rosparam dump *file*** : This saves Parameter Server to a file
- **rosparam load *file*** : This loads a file (with parameters) on Parameter Server

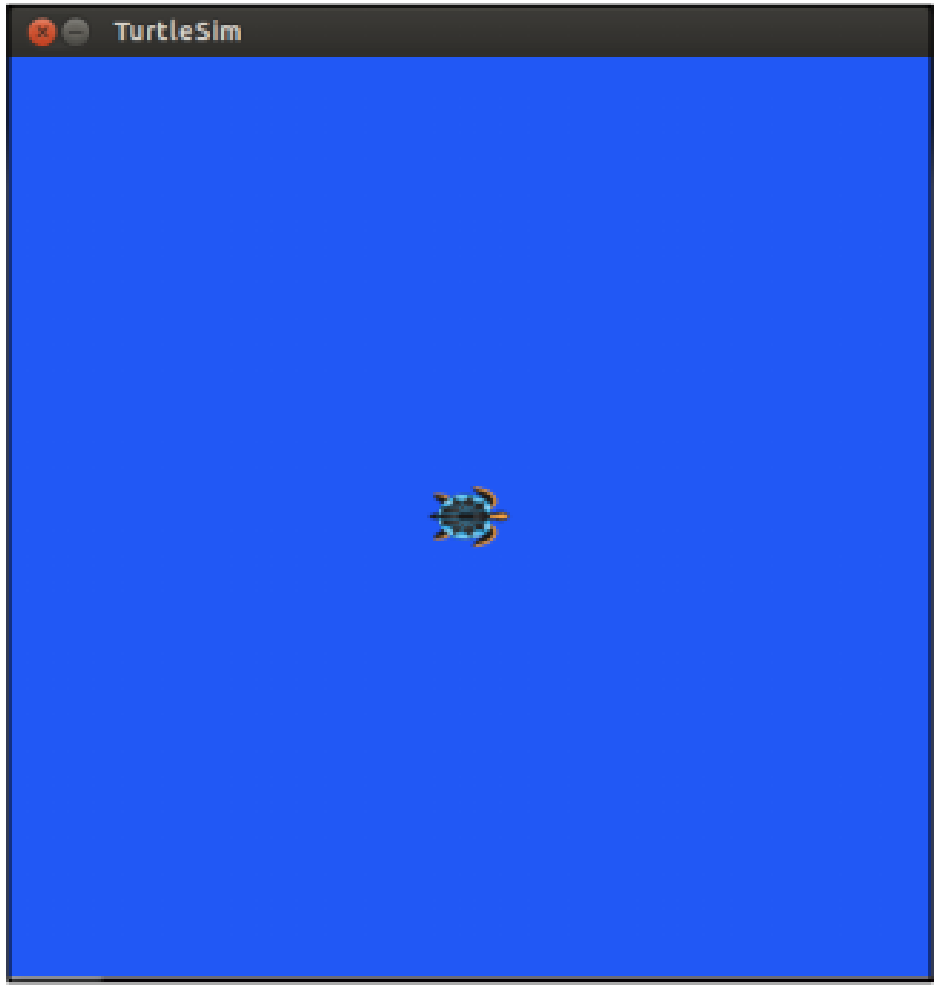
ROS Demo - Turtlesim

In separate terminal windows run:

```
$ roscore
```

```
$ rosrun turtlesim turtlesim_node
```

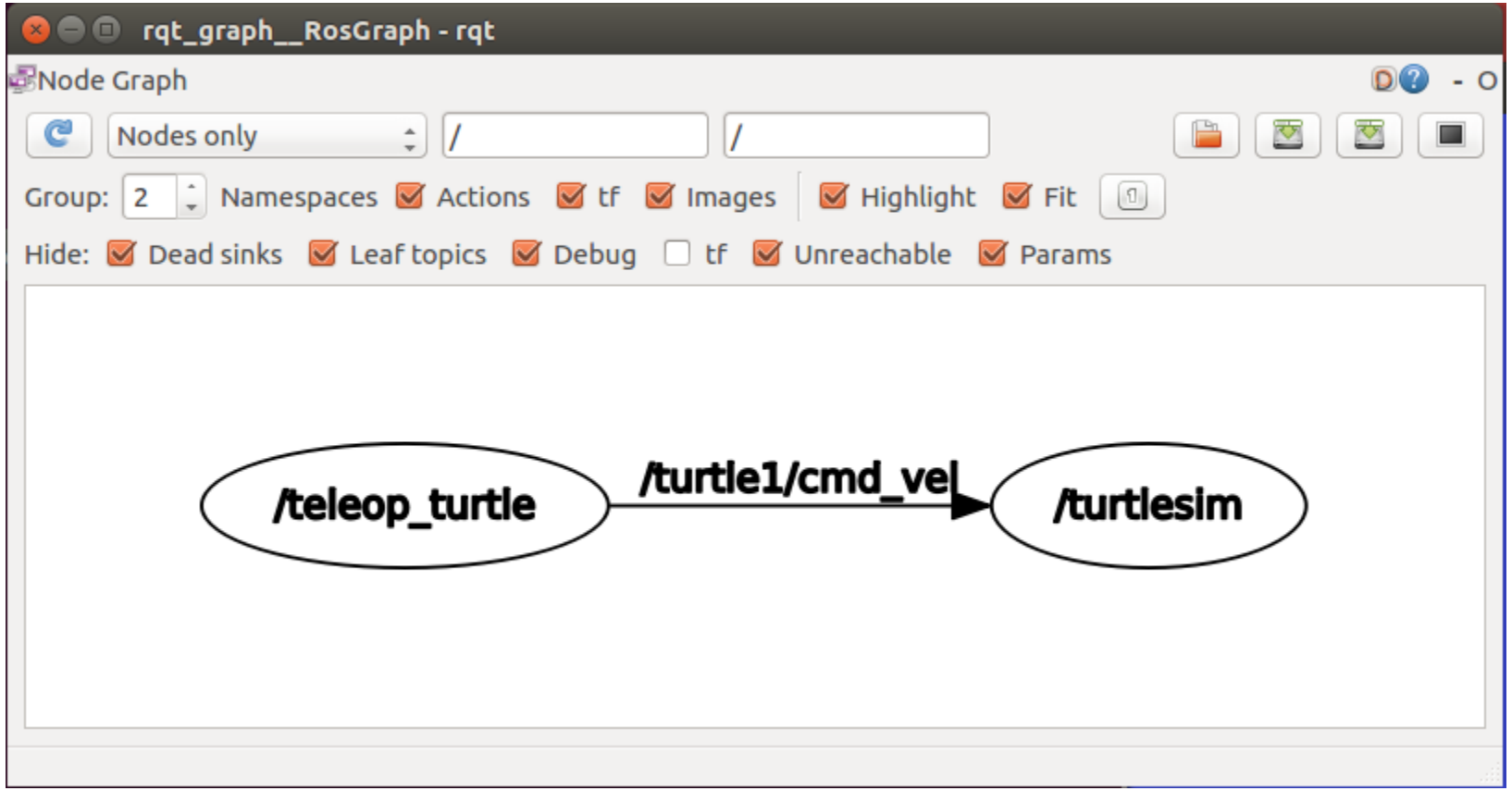
```
$ rosrun turtlesim turtle_teleop_key
```



ROS Demo - Turtlesim

ROS computation graph visualization

```
$ rosrun rqt_graph rqt_graph
```



ROS Demo - Turtlesim

- Use the rostopic pub command to publish messages to a topic
- For example, to make the turtle move forward at a 0.2m/s speed, you can publish a cmd_vel message to the topic /turtle1/cmd_vel:

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist '{linear: {x: 0.2, y: 0, z: 0}, angular: {x: 0, y: 0, z: 0}}'
```

- To specify only the linear x velocity:

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist 'linear: x: 0.2'
```

ROS Demo - Turtlesim

- Some of the messages like cmd_vel have a predefined timeout
- If you want to publish a message continuously use the argument -r with the loop rate in Hz
- For example, to make the turtle turn in circles continuously, type:

```
$ rostopic pub /turtle1/cmd_vel -r 10 geometry_msgs/Twist ‘angular: z: 0.5'
```

ROS Demo - Turtlesim

- The /spawn service. This service will create another turtle in another location with a different orientation. To start with, we are going to see the following type of message:

```
$ rosservice type /spawn | rossrv show
```

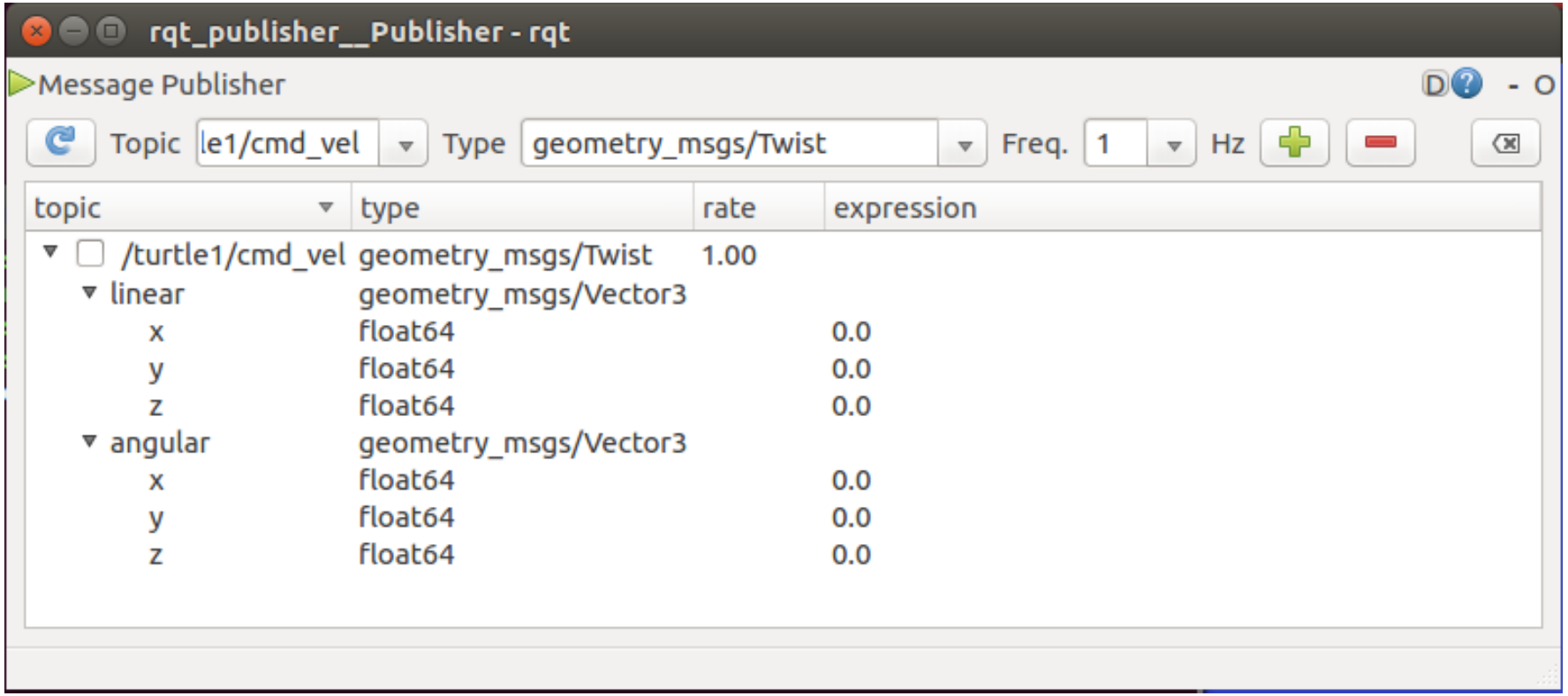
- To invoke the service and spawn the new turtlesim with new position of x and y, the orientation (theta), and the name of the new turtle

```
$ rosservice call /spawn 3 3 0.2 "new_turtle"
```

ROS Demo - Turtlesim

ROS publish using GUI

```
$ rosrun rqt_publisher rqt_publisher
```



Creating our own workspace

- To show the workspace that ROS is using, use the following command:

```
$ echo $ROS_PACKAGE_PATH
```

- The folder that we are going to create is in ~/ros_workshop_ws/src/. To add this folder, we use the following commands:

```
$ mkdir –p ~/ros_workshop_ws/src
$ cd ~/ros_workshop_ws/src
$ catkin_init_workspace
```

Creating our own workspace

- There are no packages inside—only CMakeList.txt. The next step is building the workspace. To do this, we use the following commands:

```
$ cd ~/ros_workshop_ws
$ catkin_make
```

- To finish the configuration, use the following command

```
$ source devel/setup.bash
$ echo "source ~/ros_workshop_ws/devel/setup.bash" >> ~/.bashrc
```

Creating our own package

- **cakin_create_pkg** command is using to create custom package The format of this command includes the name of the package and the dependencies

```
$ catkin_create_pkg [package_name] [depend1] [depend2] [depend3] ...
```

- We will create the new package in the workspace created previously using the following commands:

```
$ cd ~/ros_workshop_ws/src
$ catkin_create_pkg my_package std_msgs roscpp rospy
```

```
user@ros-workshop:~/ros_workshop_ws/src$ catkin_create_pkg my_package std_msgs r
oscpp rospy
Created file my_package/package.xml
Created file my_package/CMakeLists.txt
Created folder my_package/include/my_package
Created folder my_package/src
Successfully created files in /home/user/ros_workshop_ws/src/my_package. Please
adjust the values in package.xml.
```

Creating our own package

Once you have your package created and you have some code especially c++ code, msg ,srv it is necessary to build the package

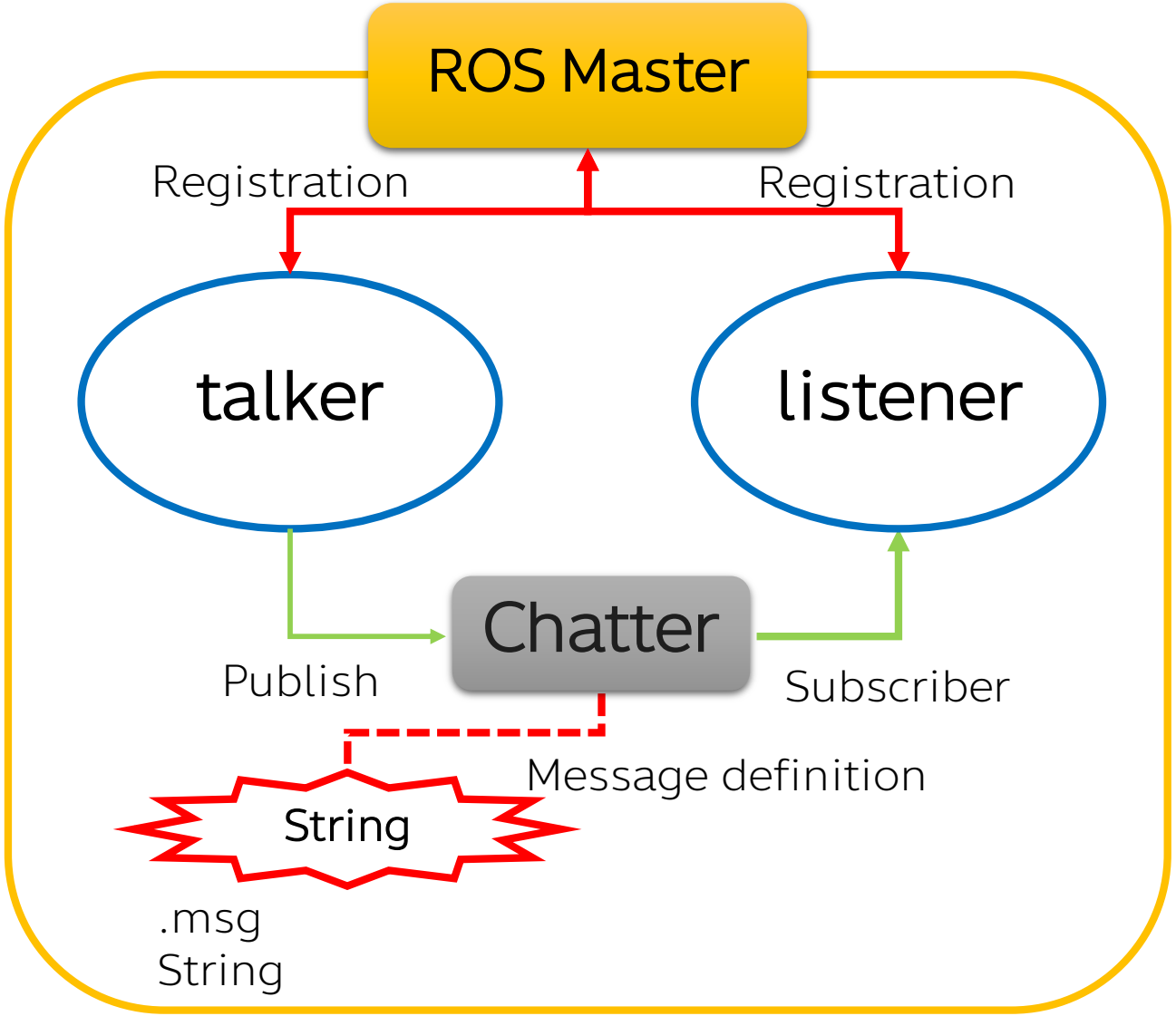
```
$ cd ~/ros_workshop_ws/
$ catkin_make
```

```
-- ~~  traversing 1 packages in topological order:
-- ~~  - my_package
-- +++ processing catkin package: 'my_package'
-- ==> add_subdirectory(my_package)
-- Configuring done
-- Generating done
-- Build files have been written to: /home/user/ros_workshop_ws/build
```

```
$ roscd my_package
```

Writing the Publisher/Subscriber Node

Our Goal



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

Writing the Publisher Node(talker)

- Change directory into the my_package package, you created in the earlier

```
$ roscd my_package
```

- First let’s create a 'scripts' folder to store our Python scripts in

```
$ mkdir scripts
$ cd scripts
```

- Create and edit python file using some editor

```
$ gedit talker.py
```

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

Writing the Publisher Node(talker)

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world "
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()
if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

Writing the Publisher Node(talker)

The first line makes sure your script is executed as a Python script. In case of many python version is installed in system, this command ensures that the interpreter will use the first installed version on your environment's $PATH

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

Import rospy if you are writing a ROS Node. The std_msgs.msg import is so that we can reuse the std_msgs/String message type.

```
import rospy
from std_msgs.msg import String
```

See: http://docs.ros.org/melodic/api/std_msgs/html/msg/String.html

Writing the Publisher Node(talker)

- Declares that your node is publishing to the chatter topic using the message type String. The queue_size argument limits the amount of queued messages if any subscriber is not receiving them fast enough (Buffer)

- rospy.init_node(NAME, ...) tells rospy the name of your node and communicate with ROS MASTER

```
rospy.init_node('talker', anonymous=True)
pub = rospy.Publisher('chatter', String, queue_size=10)
```

- To offers a convenient way for looping at the desired rate. With its argument of 10, we should expect to go through the loop 10 times per second

```
rate = rospy.Rate(10) # 10hz
```

Writing the Publisher Node(talker)

while not **rospy.is_shutdown()** : This loop is a standard rospy construct: checking the rospy.is_shutdown() flag and then doing work

- **pub.publish(hello_str)** : publishes a string to our chatter topic
- **rate.sleep()** : sleeps just long enough to maintain the desired rate through the loop
- **rospy.loginfo(str)** : performs triple-duty; the messages get printed to screen, it gets written to the Node's log file, and it gets written to rosout. rosout is a handy tool for debugging

```
while not rospy.is_shutdown():
    hello_str = "hello world“
    rospy.loginfo(hello_str)
    pub.publish(hello_str)
    rate.sleep()
```

Writing the Publisher Node(talker)

The other ways to create the message object

- **msg = String()** : Create String msg object
- **msg.data** : assigned data value with str

```
msg = String()
msg.data = hello_str
```

Or using create object with initialize parameter

```
msg = String(data=str)
```

The new code become

```
while not rospy.is_shutdown():
        hello_str = "hello world"
        msg = String()
        msg.data = hello_str
        pub.publish(msg)
        rate.sleep()
```

Writing the Publisher Node(talker)

In addition to the standard Python __main__ check, this catches a rospy.ROSInterruptException exception, which can be thrown by rospy.sleep() and rospy.Rate.sleep() methods when Ctrl-C is pressed, or your Node is otherwise shutdown. The reason this exception is raised is so that you don't accidentally continue executing code after the sleep().

```
try:
   talker()
except rospy.ROSInterruptException:
   pass
```

Writing the Subscriber Node (listener)

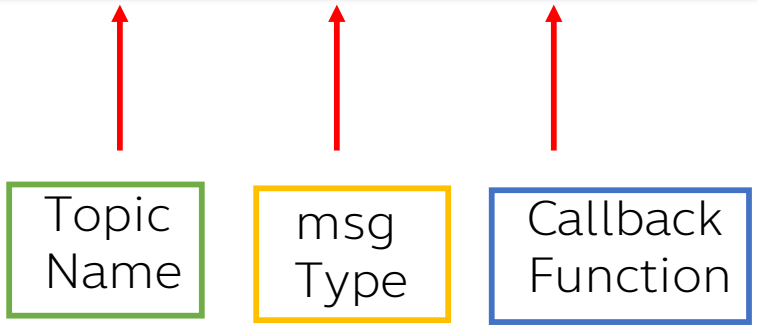

```
$ gedit listener.py
```

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "Receive : %s", data.data)

def listener():
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("chatter", String, callback)
    rospy.spin()

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

Writing the Subscriber Node (listener)

This declares that your node subscribes to the chatter topic which is of type std_msgs.msgs.String.

```
rospy.init_node('listener', anonymous=True)
rospy.Subscriber("chatter", String, callback)
```



rospy.spin() simply keeps your node from exiting until the node has been shutdown does not affect the subscriber callback functions, as those have their own threads.

```
rospy.spin()
```

Writing the Subscriber Node (listener)

When new messages are received, callback is invoked with the message as the first argument.

```
def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "Receive : %s", data.data)
```

std_msgs/String Message

File: std_msgs/String.msg

Raw Message Definition

```
string data
```

Compact Message Definition

```
string data
```

Create Custom msg

Now to create a msg
First, going to my_create package and create msg folder

```
$ roscd my_package
$ mkdir msg
$ cd msg
```

Create UserRecord.msg file and write

```
string first_name
string last_name
uint8 age
uint32 score
```

Create Custom msg

Add the message_generation dependency to generate messages in CMakeLists.txt:

```
find_package(catkin REQUIRED COMPONENTS
        roscpp
        rospy
        std_msgs
        message_generation
)
```

Uncomment and add msg file name

```
add_message_files(
        FILES
         UserRecord.msg
)
```

Create Custom msg

Uncomment generate_message

```
generate_message(
   DEPENDENCIES
   std_msgs
   my_package
)
```

Add message runtime into catkin_package

```
catkin_package(
...
CATKIN_DEPENDS ... message_runtime
....
)
```

Create Custom msg

Open package.xml and add two lines below

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

Create Custom msg

Then make the package to generate msg executable file.

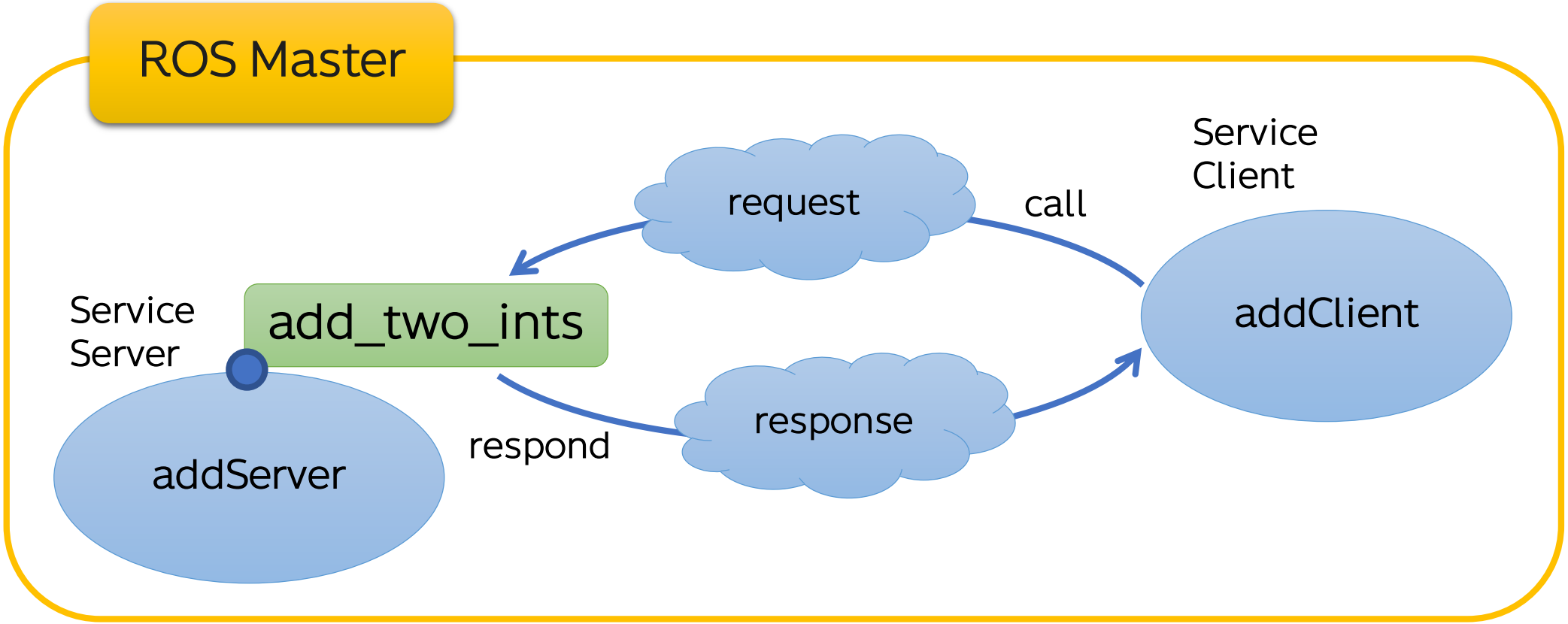
```
$ cd ~/ros_workshop_ws
$ catkin_make
```

Check the new msg is already created

```
user@ros-workshop:~/ros_workshop_ws$ rosmsg show UserRecord.msg
[my_package/UserRecord]:
string first_name
string last_name
uint8 age
uint32 score
```

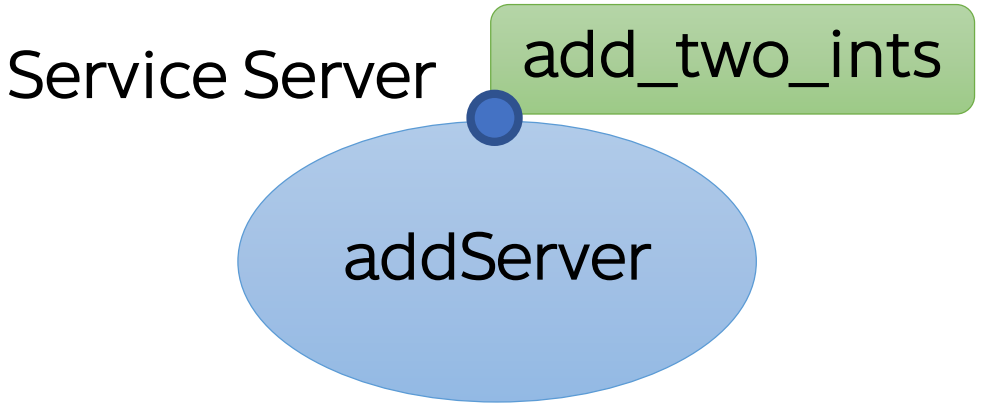
Writing the Service/Client Node

Our Goal



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

Writing The Service Server Node (addServer)



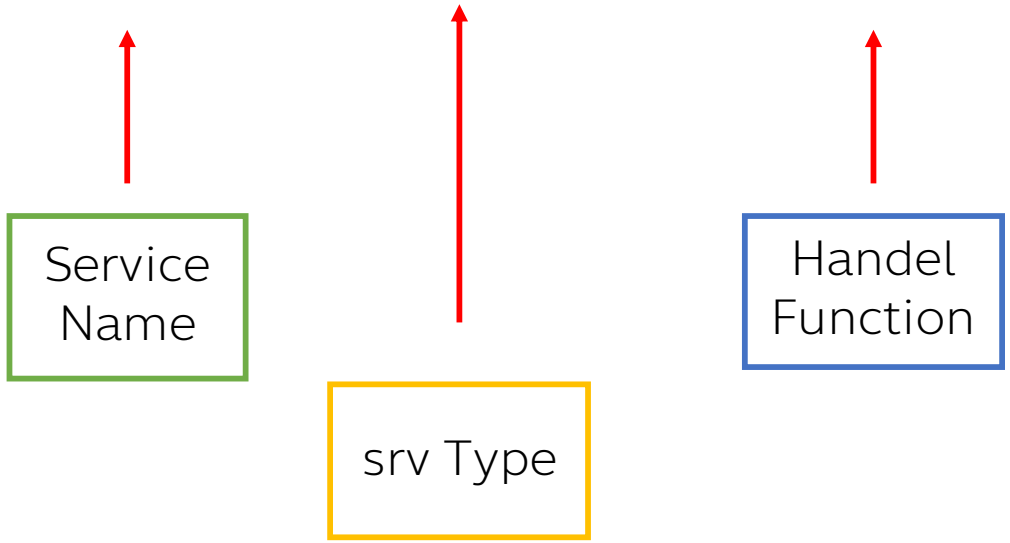
Writing The Service Server Node (addServer)

```
#!/usr/bin/env python
from my_package.srv import AddTwoInts,AddTwoIntsResponse
import rospy

def handle_add_two_ints(req):
    print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b))
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
    print "Ready to add two ints."
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```

Writing The Service Server Node (addServer)

This declares a new service named add_two_ints with the AddTwoInts service type. All requests are passed to handle_add_two_ints function.

```
s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
```



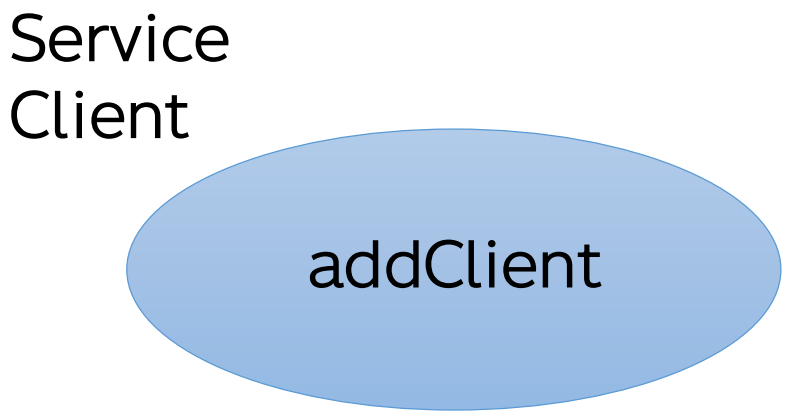
Writing The Service Server Node (addServer)

handle_add_two_ints is called with instances of AddTwoIntsRequest and returns instances of AddTwoIntsResponse.

```
def handle_add_two_ints(req):
    print ("Returning [%s + %s = %s]" % (req.a, req.b, (req.a + req.b)))
    response = AddTwoIntsResponse()
    response.sum = req.a + req.b
    return response
```

```
#Request
int64 a
int64 b
---
# Response
int64 sum
```

Writing The Service Client Node (addClient)



Writing The Service Client Node (addClient)

```
#!/usr/bin/env python
import sys
import rospy
from my_package.srv import AddTwoInts, AddTwoIntsResponse
def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp1 = add_two_ints(x, y)
        return resp1.sum
    except rospy.ServiceException, e:
        print "Service call failed: %s"%e
```

Writing The Service Client Node (addClient)

```
def usage():
    return "%s [x y]"%sys.argv[0]
if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:
        print usage()
        sys.exit(1)
    print "Requesting %s+%s"%(x, y)
    print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

Writing The Service Client Node (addClient)

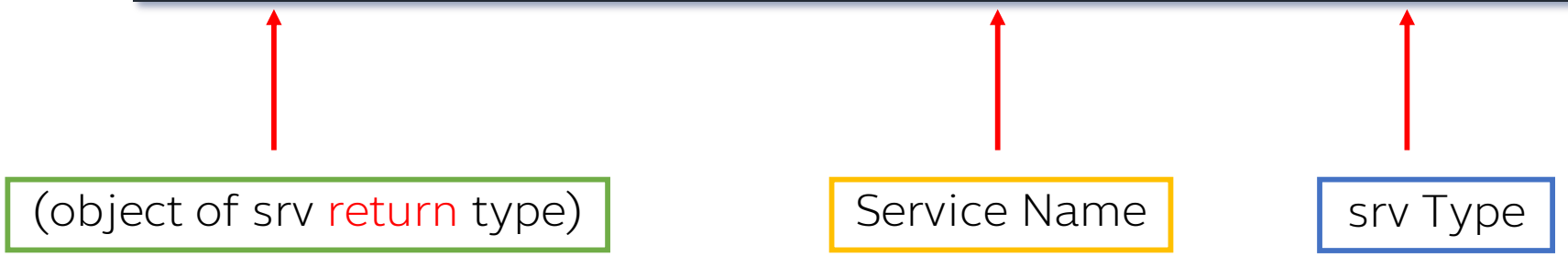
The client code for calling services is also simple. For clients you don't have to call init_node(). We first call:

This is a convenience method that blocks until the service named add_two_ints is available.

```
rospy.wait_for_service('add_two_ints')
```

Next we create a handle for calling the service:

```
add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
```



Writing The Service Client Node (addClient)

We can use this handle just like a normal function and call it:

```
resp1 = add_two_ints(x, y)
return resp1.sum
```

The return value is an AddTwoIntsResponse object

If the call fails, a rospy.ServiceException may be thrown, so you should setup the appropriate try/except block.

Create Custom srv

Now to create a srv
First, going to my_create package and create srv folder

```
$ roscd my_package
$ mkdir srv
$ cd srv
```

Create AddTwoInts.srv file and write

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

Create Custom srv

Add the message_generation dependency to generate messages in CMakeLists.txt:

```
find_package(catkin REQUIRED COMPONENTS
        roscpp
        rospy
        std_msgs
        message_generation
)
```

Uncomment and add srv file name

```
add_service_files(
        FILES
        AddTwoInts.srv
)
```

Create Custom srv

Uncomment generate_message

```
generate_message(
   DEPENDENCIES
   std_msgs
   my_package
)
```

Add message runtime into catkin_package

```
catkin_package(
...
CATKIN_DEPENDS ... message_runtime
....
)
```

Create Custom srv

Open package.xml and add two lines below

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

Create Custom srv

Then make the package to generate srv executable file.

```
$ cd ~/ros_workshop_ws
$ catkin_make
```

Check the new srv is already created

```
user@ros-workshop:~/ros_workshop_ws$ rossrv show my_package/AddTwoInts
int64 a
int64 b
---
int64 sum
```

Record and Playback data

To record the published (from Topic) data. You can use the following commands

```
rosbag record Topic1 [Topic2 Topic3 ...]
```

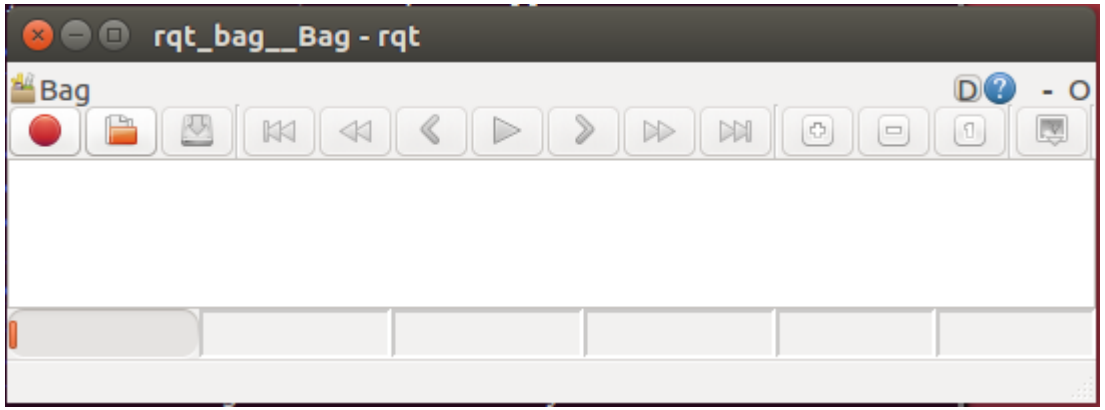
after press ctrl-C You should see something like this, with only the indicated topics:

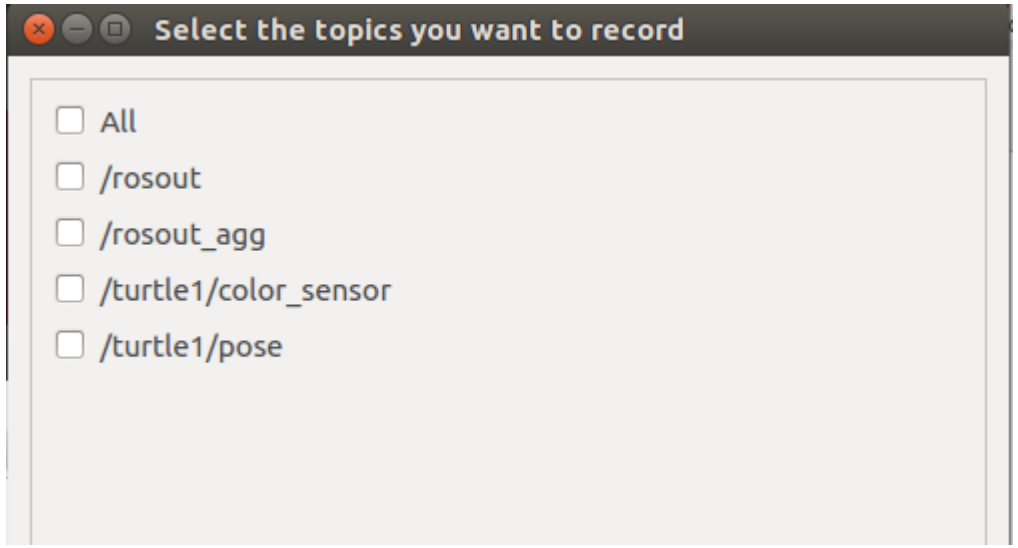
```
path:        <your bagfile>.bag
version:     2.0
duration:    12.6s
start:       Dec 10 2014 20:20:49.45 (1418271649.45)
end:         Dec 10 2014 20:21:02.07 (1418271662.07)
size:        68.3 KB
messages:    813
compression: none [1/1 chunks]
types:       geometry_msgs/Twist [9f195f881246fdfa2798d1d3eebca84a]
             turtlesim/Pose      [863b248d5016ca62ea2e895ae5265cf9]
topics:      /turtle1/cmd_vel    23 msgs    : geometry_msgs/Twist
             /turtle1/pose       790 msgs   : turtlesim/Pose
```

Record and Playback data

Or you can record the published data with GUI using following command

```
$ rosrun rqt_bag rqt_bag
```





Record and Playback data

To replay the bag file to reproduce behavior in the running system. In a terminal window run the following command in the directory where you took the original bag file:

```
$ rosbag play <your bagfile>
```

In this window you should immediately see something like:

```
[ INFO] [1418271315.162885976]: Opening 2014-12-10-20-08-34.bag

Waiting 0.2 seconds after advertising topics... done.

Hit space to toggle paused, or 's' to step.
```

Logging and Diagnostics

rospy has several methods for writing log messages, all starting with "log":

```
rospy.logdebug(msg, *args, **kwargs)
rospy.loginfo(msg, *args, **kwargs)
rospy.logwarn(msg, *args, **kwargs)
rospy.logerr(msg, *args, **kwargs)
rospy.logfatal(msg, *args, **kwargs)
```

All your node's log file will be in ROS_ROOT/log or ~/.ros/log, unless you override it with the $ROS_LOG_DIR environment variable. If you are using roslaunch, you can use the roslaunch-logs command to tell you the location of the log directory

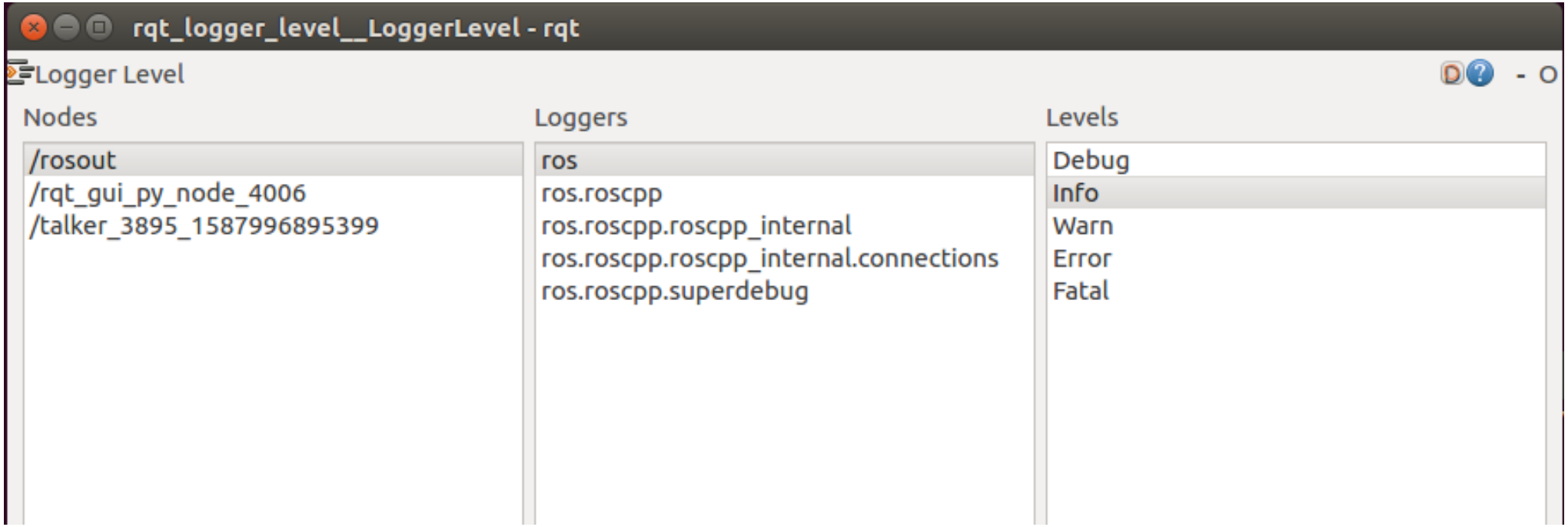
Logging and Diagnostics

- **DEBUG**
  Information that you never need to see if the system is working properly.
- **INFO**
  Small amounts of information that may be useful to a user.
- **WARN**
  Information that the user may find alarming, and may affect the output of the application, but is part of the expected working of the system
- **ERROR**
  Something serious (but recoverable) has gone wrong
- **FATAL**
  Something unrecoverable has happened. Examples: "Motors have caught fire!"

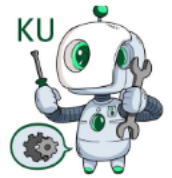
Logging and Diagnostics

You also using GUI to see or change rospy's logger level for individual loggers during runtime.

```
$ rosrun rqt_logger_level rqt_logger_level
```



LAUNCH FILE



KU
KASETSART
UNIVERSITY

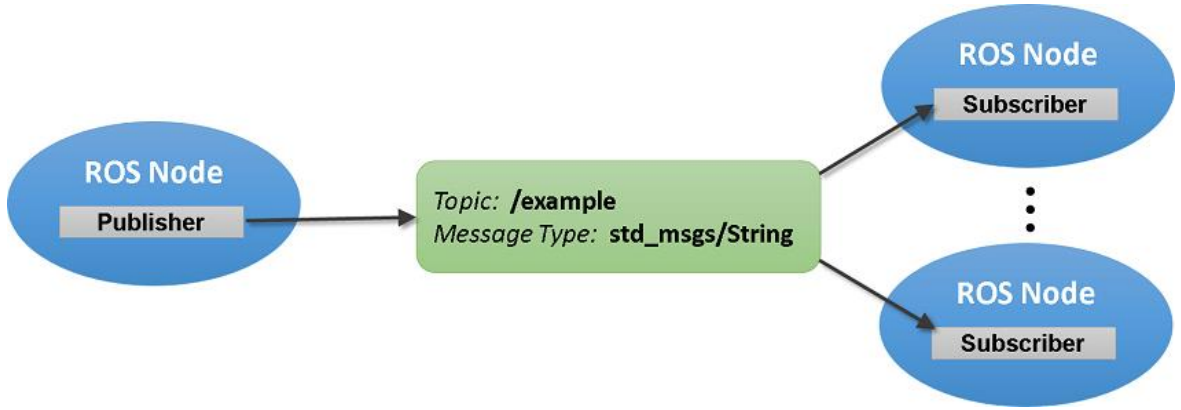
Architecture

Local process

- Using popen to launch and using POSIX to kill them
- In ros architecture, they can not know when node is initialized
- Can not confirm any particular order to the startup of node

```
<group_state name="vertical" group="arm">
    <joint name="crane_plus_elbow_joint" value="0" />
    <joint name="crane_plus_gripper_joint" value="0" />
    <joint name="crane_plus_shoulder_flex_joint" value="0" />
    <joint name="crane_plus_shoulder_revolute_joint" value="0" />
    <joint name="crane_plus_wrist_joint" value="0" />
</group_state>
<group_state name="resting" group="arm">
    <joint name="crane_plus_elbow_joint" value="1.7787" />
    <joint name="crane_plus_gripper_joint" value="0" />
    <joint name="crane_plus_shoulder_flex_joint" value="0.884" />
    <joint name="crane_plus_shoulder_revolute_joint" value="0" />
    <joint name="crane_plus_wrist_joint" value="1.3373" />
</group_state>
```

Tag



<launch>

- Root element of any launch file.

<node>

- Use for specifies node that you wish to launch.
- Attribute
  - **name**
    - Set node name.
  - **pkg**
    - Node package name.
  - **type**
    - Node type. There must be a corresponding executable with the same name. (Node file name)
  - **respawn (true, False)**
    - Restart the node automatically if it quits.

Tag

<include>

- Use for import another XML format files into current file.
- Attribute
  - **file**
    - Full part to file that you are want to include.

<param>

- Use for defined a parameter to set on the parameter server.
- Attribute
  - **name**
    - Set parameter name.
  - **type**
    - Set parameter type. (str,double,int,bool,yaml)
  - **value**
    - Set parameter value.

Tag

<group>

- Use for group an elements sharing a namespace and remap.
- Attribute
  - **ns**
    - Set namespace for the group of nodes.

<arg>

- Use for create re-usable and configurable launch files
- Args are not global
- Attribute
  - **Name**
    - Set argument name
  - **Default**
    - Set argument default value
  - **Value**
    - Set argument value

Substitution Args

$(find pkg)

- Use for specifies a package relative path.

$(arg foo)

- Use for evaluates to the value specified by an <arg> tag.

$(env ENVIRONMENT_VARIABLE)

- Use for substitute the value of a variable from the current environment.

If and unless attributes

All tags support if and unless attributes.

If=value

- If value evaluates to true, include tag and its contents.

unless=value

- Unless value evaluates to true, include tag and its contents

Example (node)

- Go to your package

```
$ roscd my_package
```

- Make new directory name launch

```
$ mkdir launch
```

- Go to launch directory

```
$ cd launch
```

Example (node)

Create talker.launch

```
$ gedit talker.launch
```

Launch talker node code

```
<launch>
   <node name="talker" pkg="my_package" type="talker.py" respawn="true" output="screen" />
</launch>
```

Set node name is talker , package name my_package , node type talker.py , respawn , and output to screen

Launch command

```
$ roslaunch my_package talker.launch
```

Example (node)

Create listener.launch

```
$ gedit listener.launch
```

- Launch listener node code

```
<launch>
   <node name="listener" pkg="my_package" type="listener.py" respawn="true" output="screen" />
</launch>
```

Set node name is listener , package name my_package , node type listener.py , respawn , and output to screen

Launch command

```
$ roslaunch my_package listener.launch
```

Example (param)

Go to scripts directory and copy talker.py to dynamics_talker.py

```
$ roscd my_package/scripts
$ cp talker.py dynamics_talker.py
```

Edit dynamics_talker.py

```
$ gedit dynamics_talker.py
```

Replace hello_str="hello world" by hello_str=rospy.get_param("talker_str")

Example (param)

Run

```
$ roscd my_package/launch
$ gedit dynamics_talker.launch
```

Write launch file

```
<launch>
  <param name="talker_str" type="str" value="hello world"/>
   <node name="dynamics_talker" pkg="my_package" type="dynamics_talker.py
" respawn="true" output="screen" />
</launch>
```

Example (param)

Launch file

```
$ roslaunch my_package dynamics_talker.launch
```

You will see “hello world” on your screen

To change “hello world” to other word open another screen and run

```
$ rosparam set /talker_str Goodbye
```

You will see string that show on your screen has been changed.

Example (include)

Create combine.launch

```
$ gedit combine.launch
```

Combine launch code

```
<launch>
  <include file="$(find my_package)/launch/dynamics_talker.launch" />
   <include file="$(find my_package)/launch/listener.launch"/>
</launch>
```

Explain

Use $(find my_package) to get relative path of my_package and concatenate with launch file path and use tag include for launch that file

- **Launch command**

```
$ roslaunch my_package combine.launch
```

Example (group)

- **Edit combine.launch**

```
<launch>
      <arg name="hello" value="True" />
      <group If="$(arg hello)">
            <include file="$(find my_package)/launch/dynamics_talker.launch" />
            <include file="$(find my_package)/launch/listener.launch"/>
            <param name="talker_str" type="str" value="hello" />
       </group>

       <group  unless ="$(arg hello)">
            <include file="$(find my_package)/launch/dynamics_talker.launch" />
            <include file="$(find my_package)/launch/listener.launch"/>
            <param name="talker_str" type="str" value="goodbye" />
       </group>

</launch>
```

THANK YOU