

F1/10

Autonomous Racing

**ROS services
roslaunch**

Madhur Behl

CS 4501/SYS 4582
Spring 2018
Rice Hall 120

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
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

Experimental

Client API Commonly Used Features

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

Build and run

- To build your code do:

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

- To run it execute these three lines in three different terminals:

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

- To see the publis/subscriber connection do:

```
$ rosrun rqt_graph rqt_graph
```

ROS Computational Graph / Communication: ROS Services

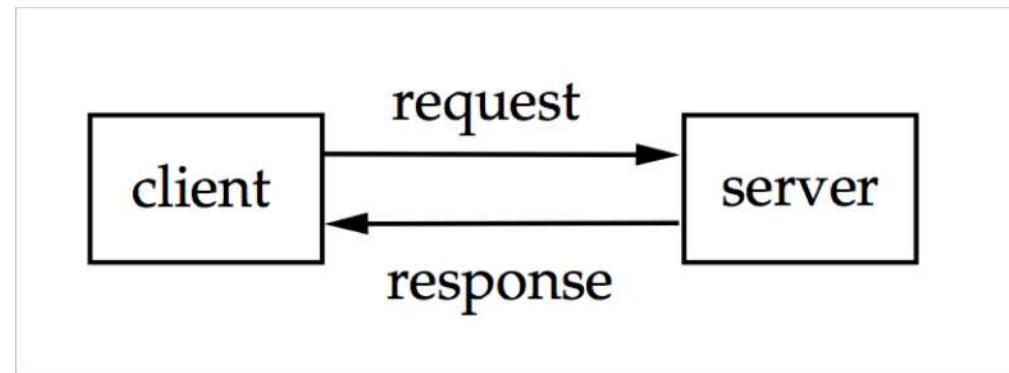
- Publish / subscribe, many-to-many, unidirectional topic model works well for most use cases
- Request / reply model is better for some use cases
 - Equivalent to a remote procedure call (RPC)
 - Request: client node sends a providing node a command and/or data
 - Reply: providing node sends data back to the client node
 - Example – set a camera's calibration matrix, receive confirmation of success
- Services: ROS implementation of the request / reply model
 - Services are named
 - Providing node advertises the service by name
 - Client node calls the service by name
 - Master performs name registration
 - Type defined in .srv files
 - Similar to .msg files, but include both the request and response type
 - Should reply quickly / do not block (service call can not be preempted)
 - Limitations: not logged, difficult to inspect
 - Tend to be used infrequently

ROS Services: enable request/response communication between nodes.

- Service descriptions are stored in `.srv` files in the `srv/` subdirectory of a package.
- When you publish topics, you are sending data in a many-to-many fashion, but when you need a request or an answer from a node, you can't do it with topics.

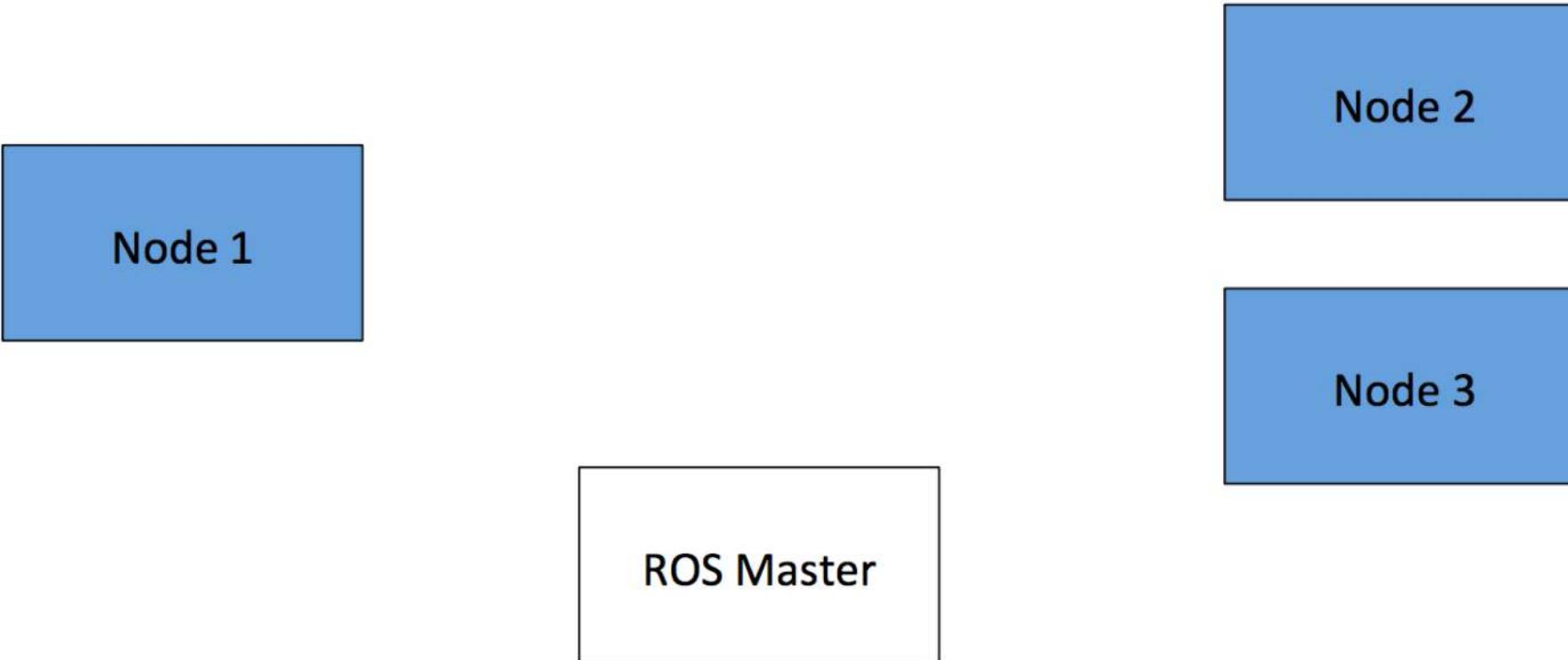
Service calls are **bi-directional**.

One node sends information to another node and waits for a response.



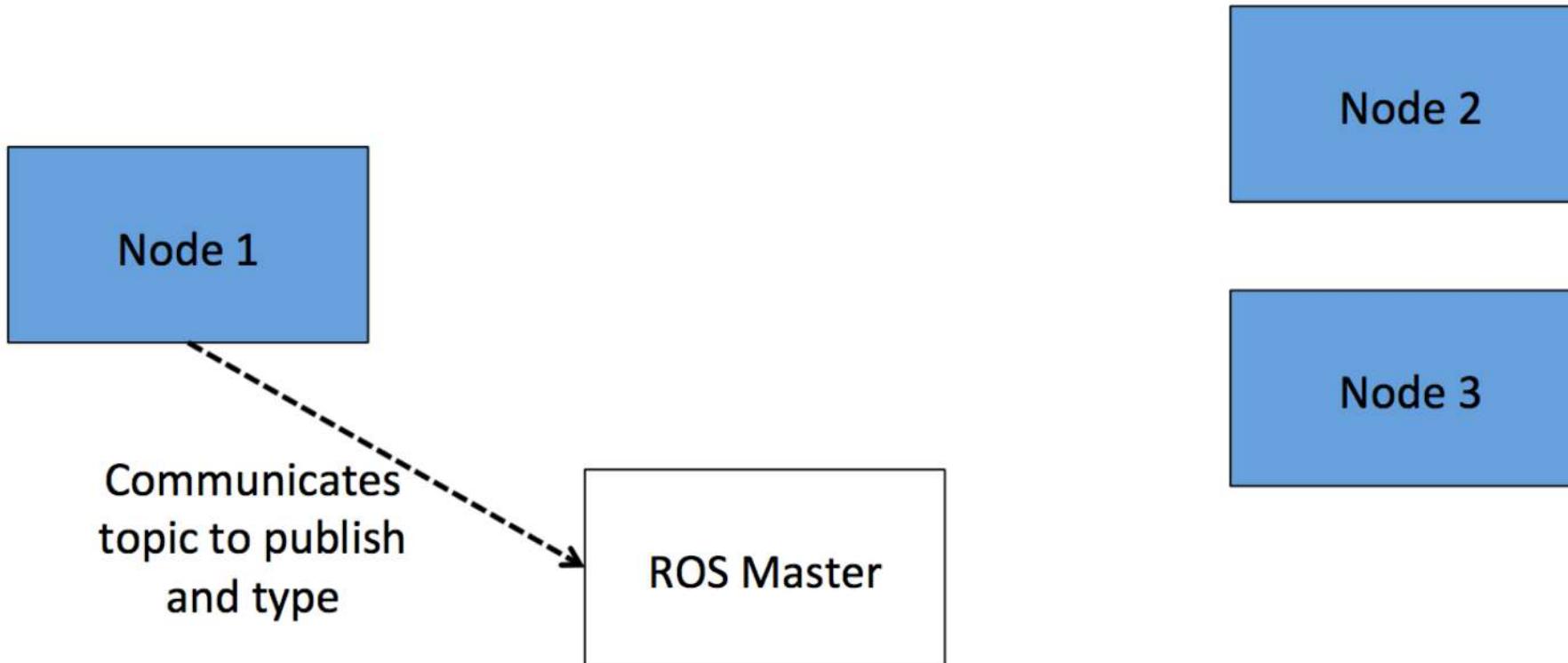
In contrast, when a message is published, there is no concept of a response, and not even any guarantee that anyone is subscribing to those messages.

Communications: Publish/Subscribe



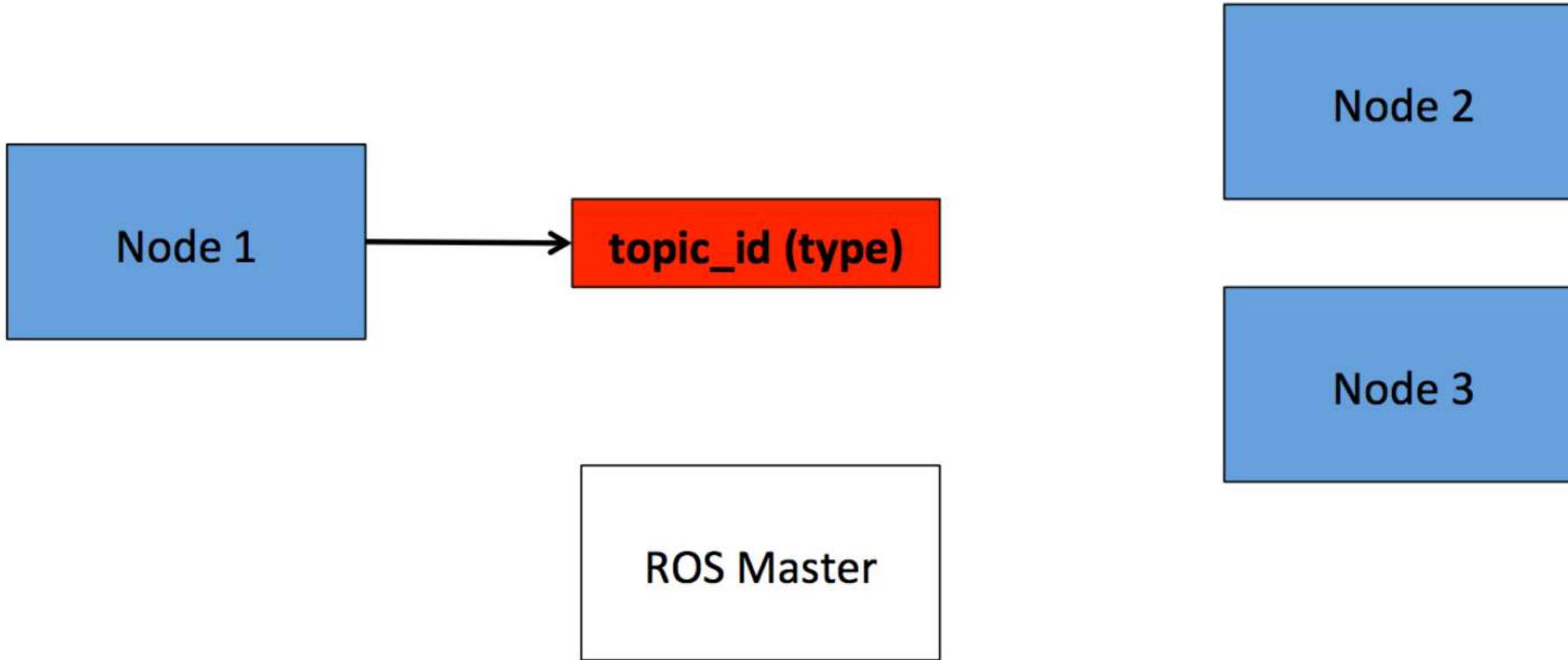
Asynchronous, many-to-many and one-way transport.

Communications: Publish/Subscribe



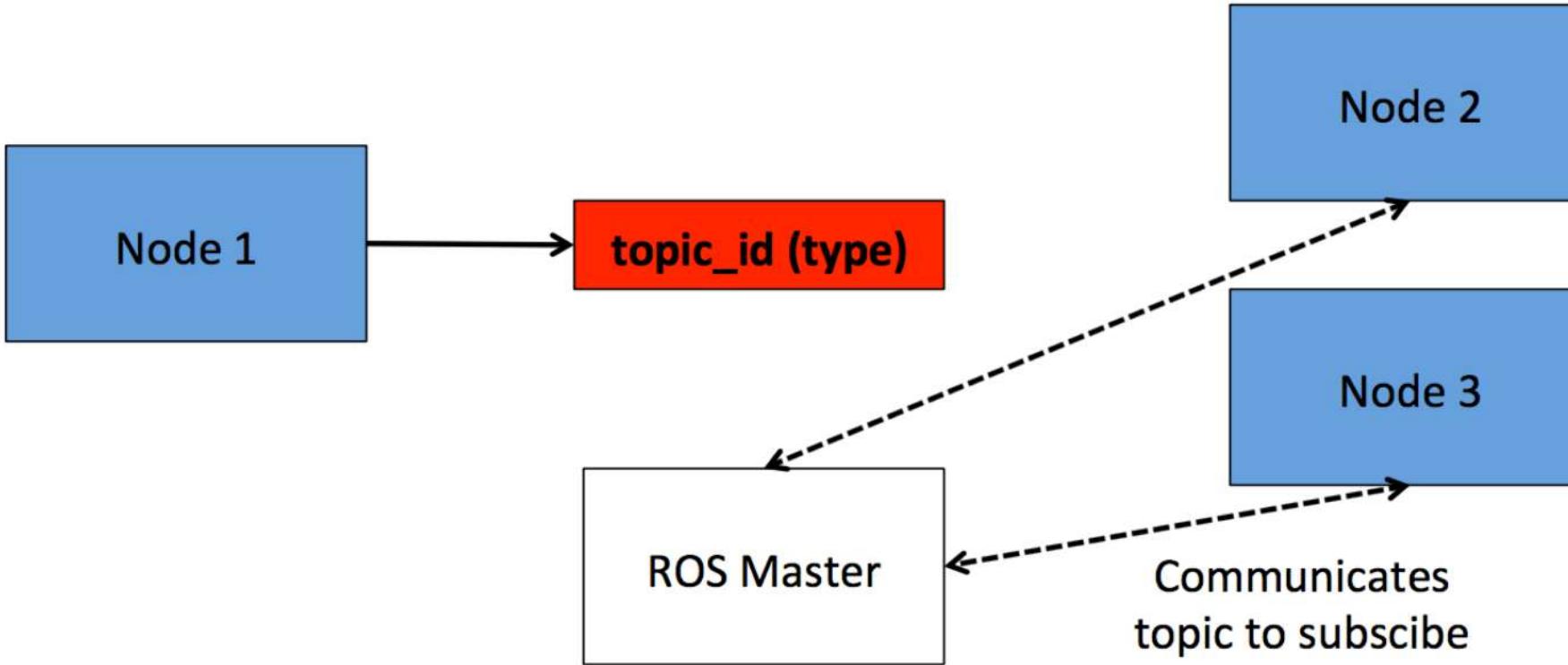
Asynchronous, many-to-many and one-way transport.

Communications: Publish/Subscribe



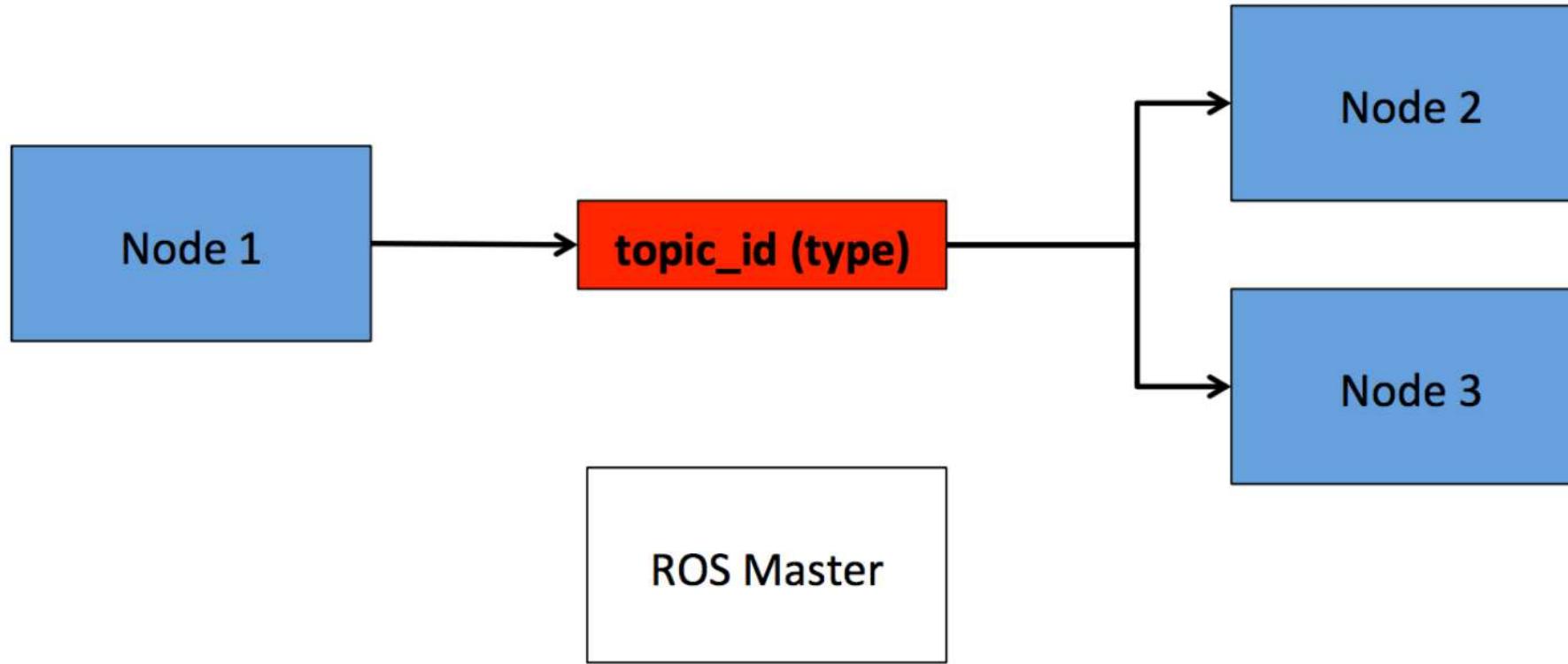
Asynchronous, many-to-many and one-way transport.

Communications: Publish/Subscribe



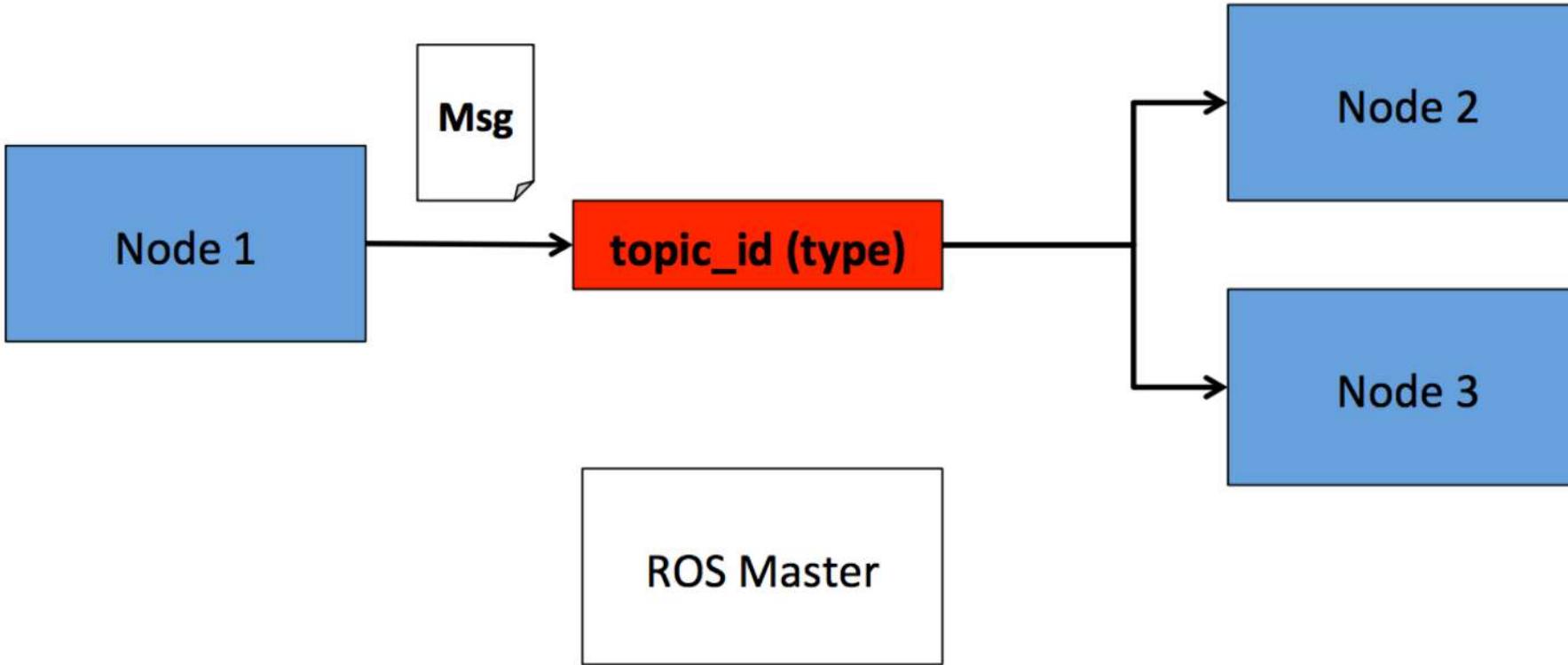
Asynchronous, many-to-many and one-way transport.

Communications: Publish/Subscribe



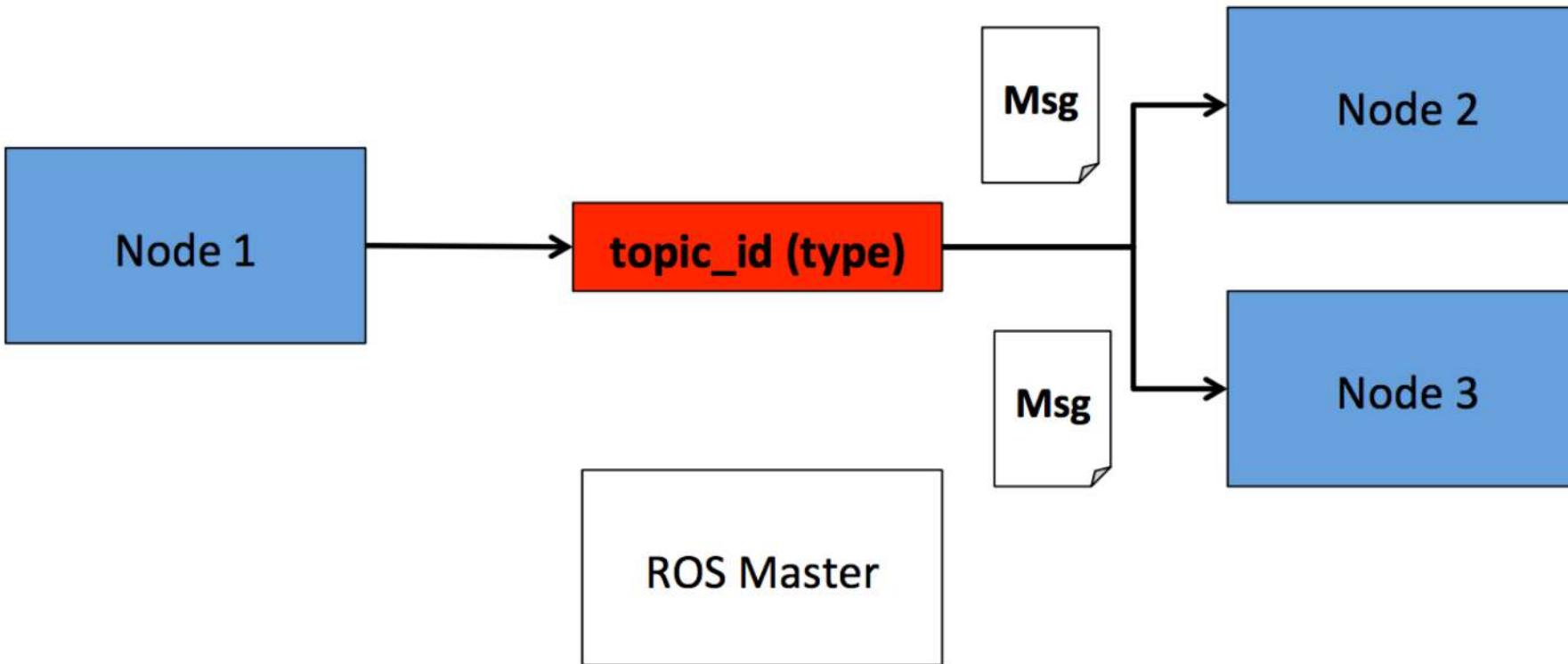
Asynchronous, many-to-many and one-way transport.

Communications: Publish/Subscribe



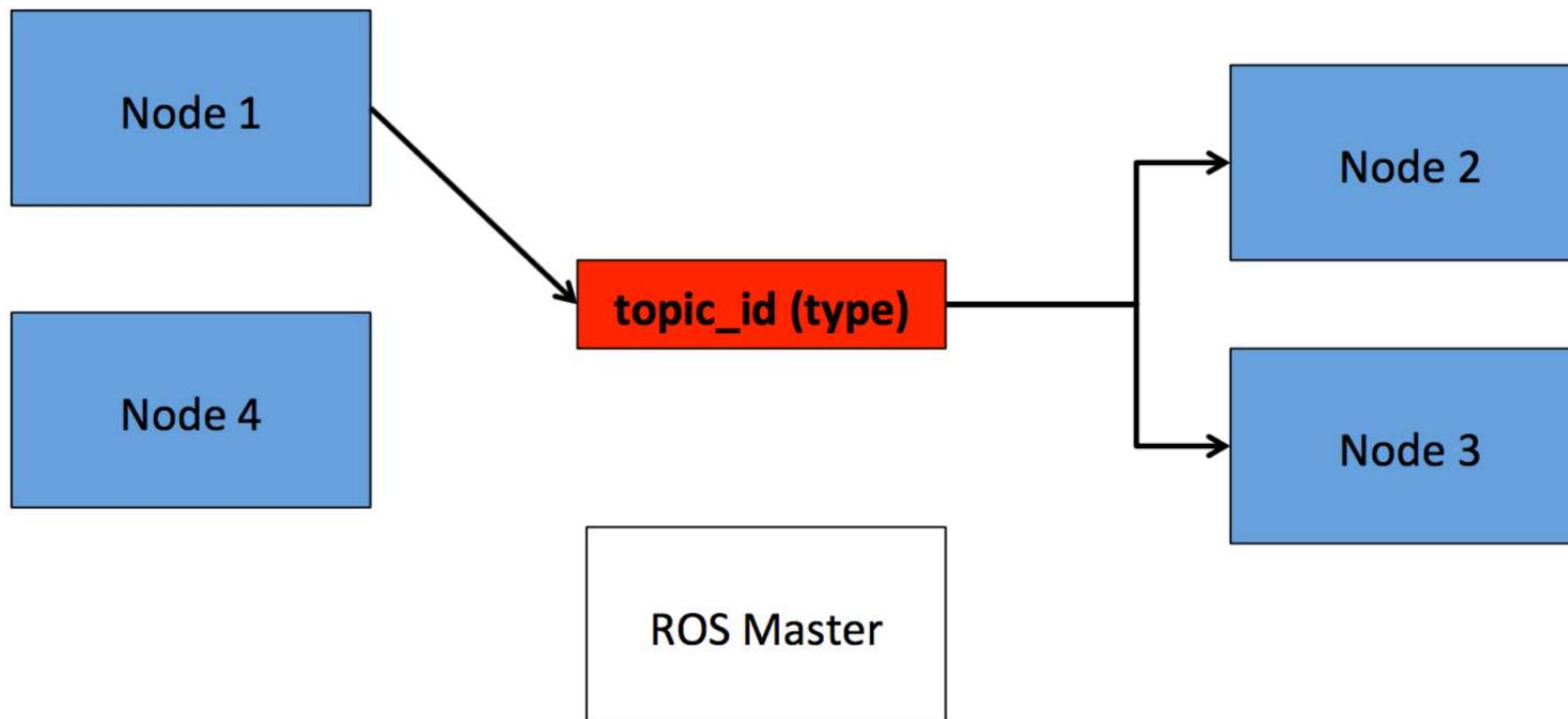
Asynchronous, many-to-many and one-way transport.

Communications: Publish/Subscribe



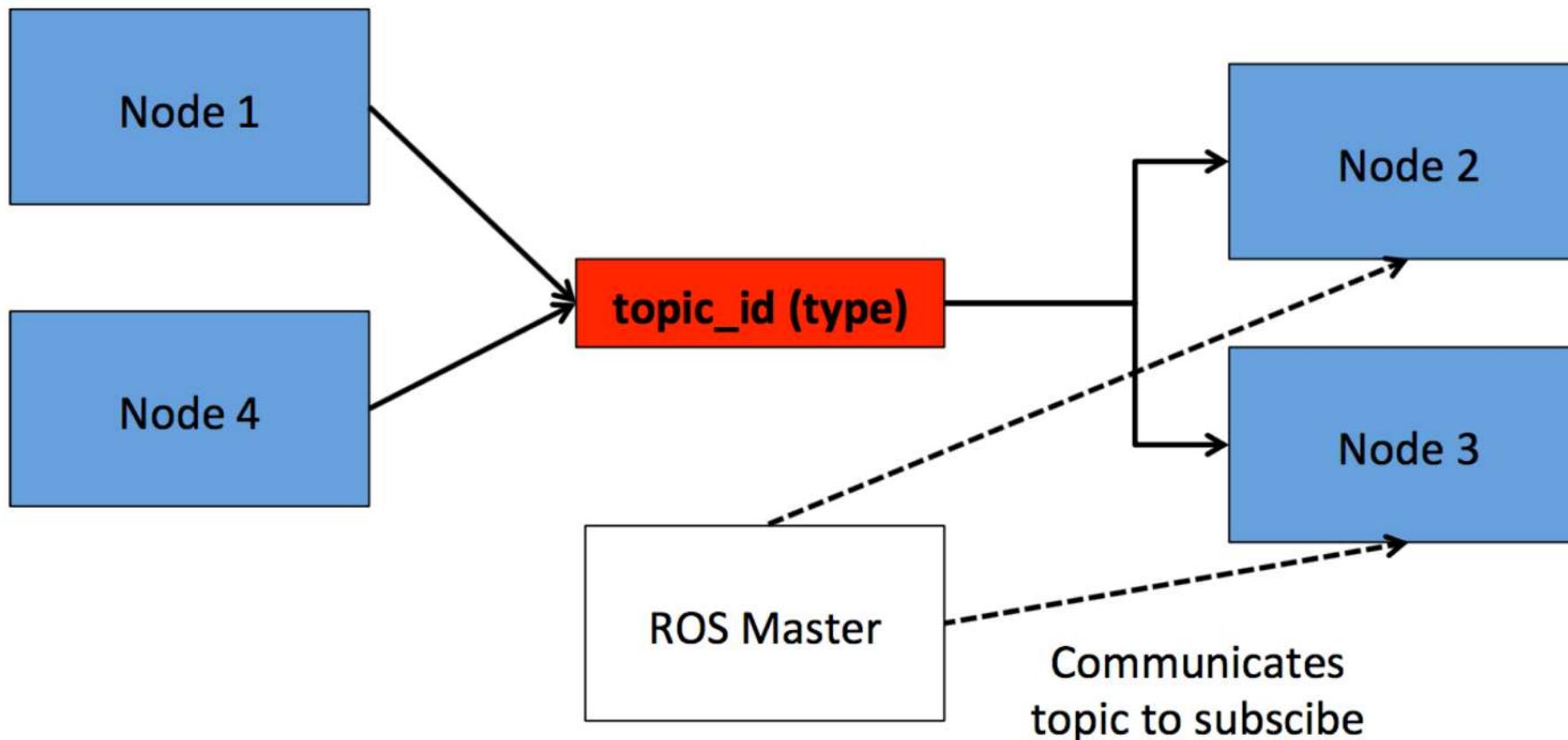
Asynchronous, many-to-many and one-way transport.

Communications: Publish/Subscribe



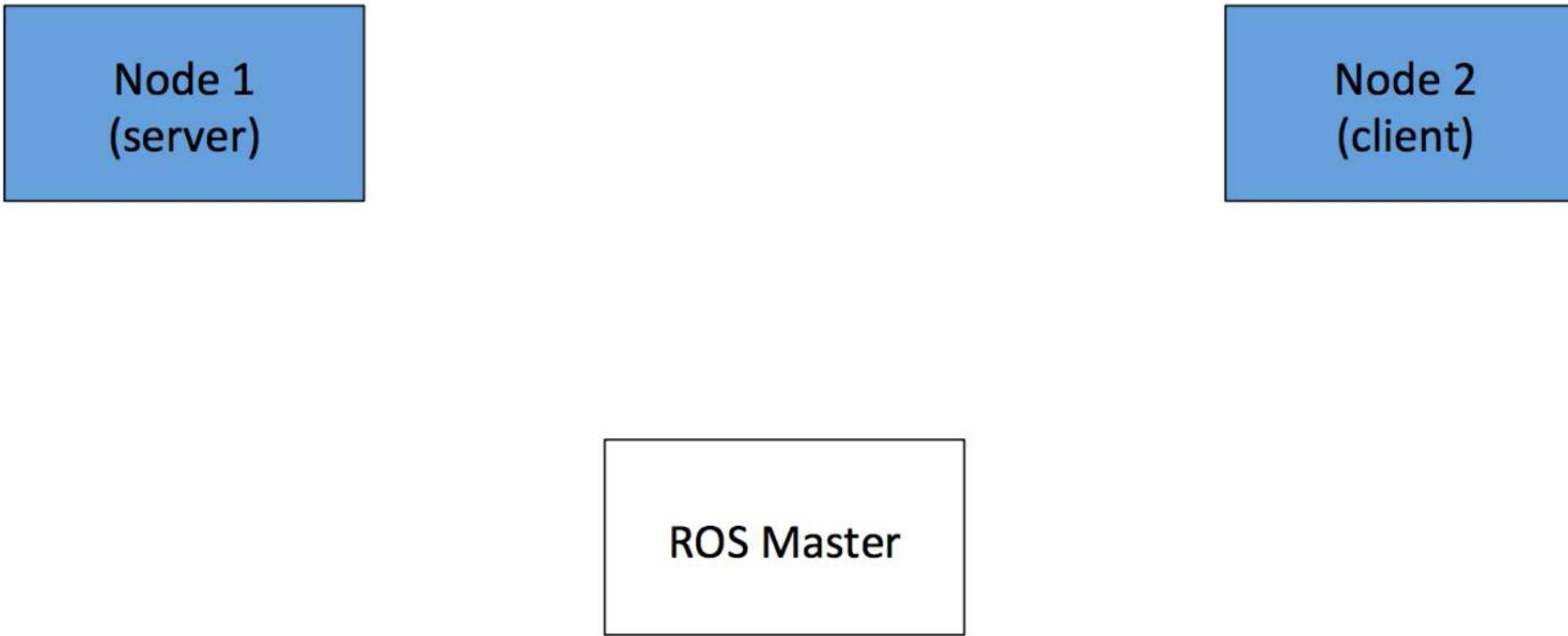
Asynchronous, many-to-many and one-way transport.

Communications: Publish/Subscribe



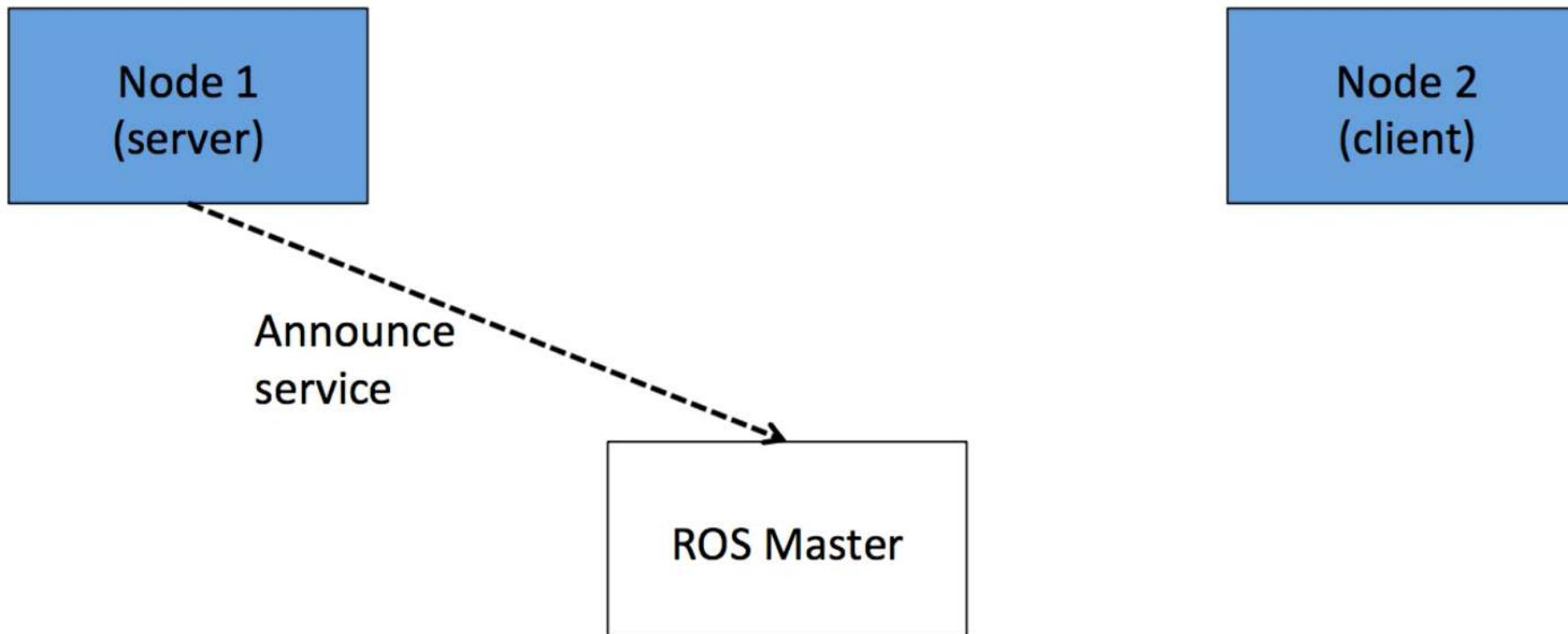
Asynchronous, many-to-many and one-way transport.

Communications: Client/Service



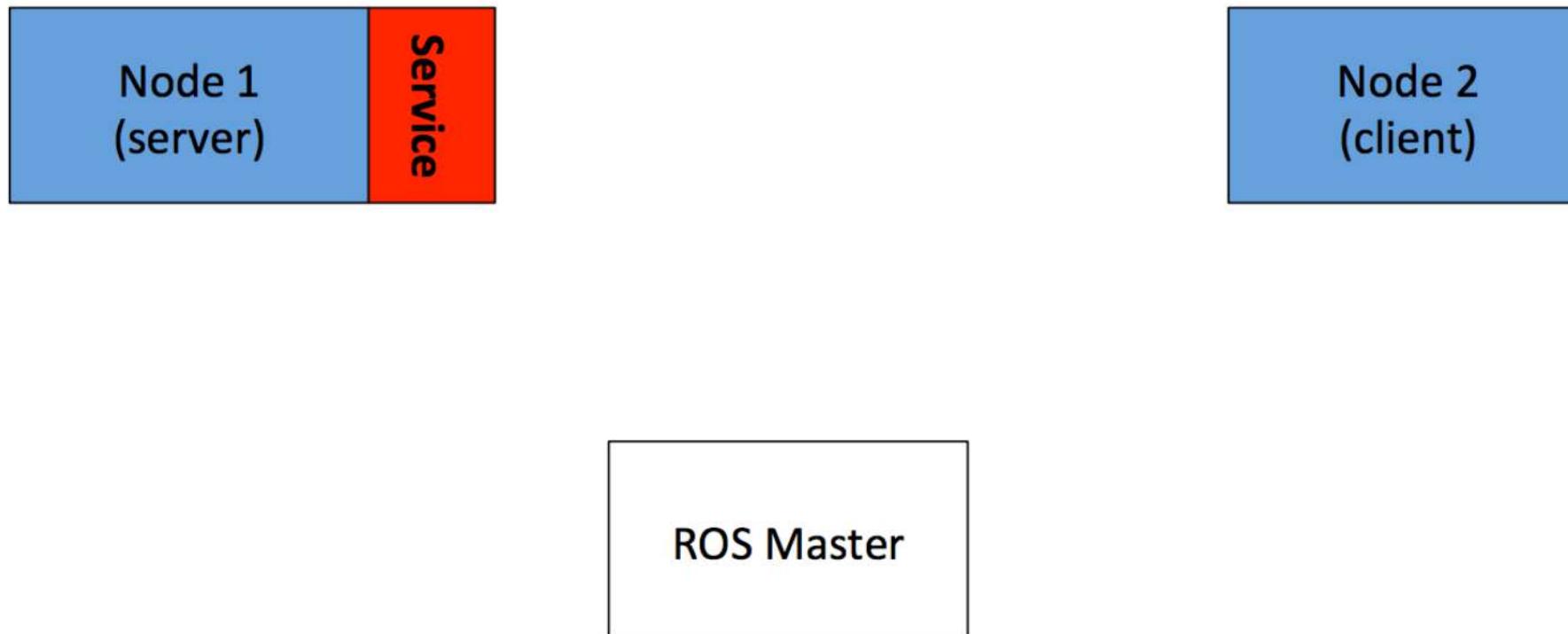
Synchronous, one-to-many

Communications: Client/Service



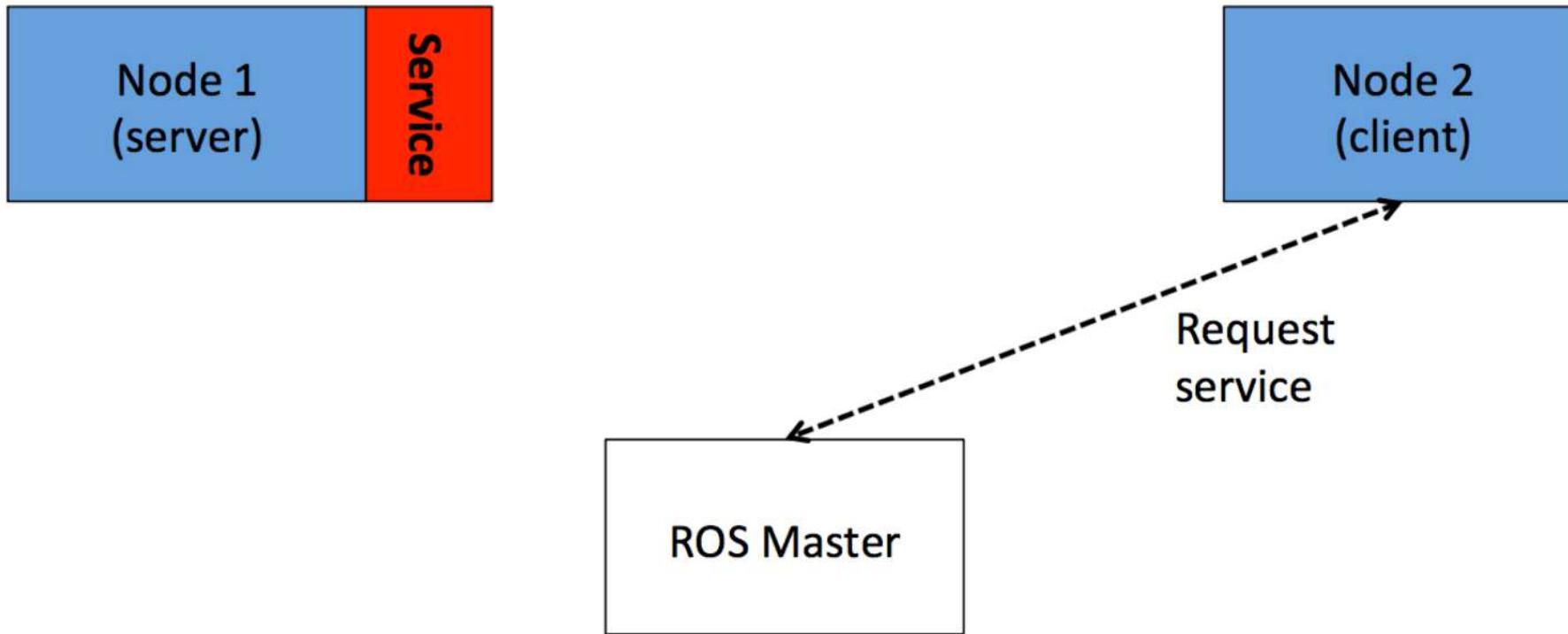
Synchronous, one-to-many

Communications: Client/Service



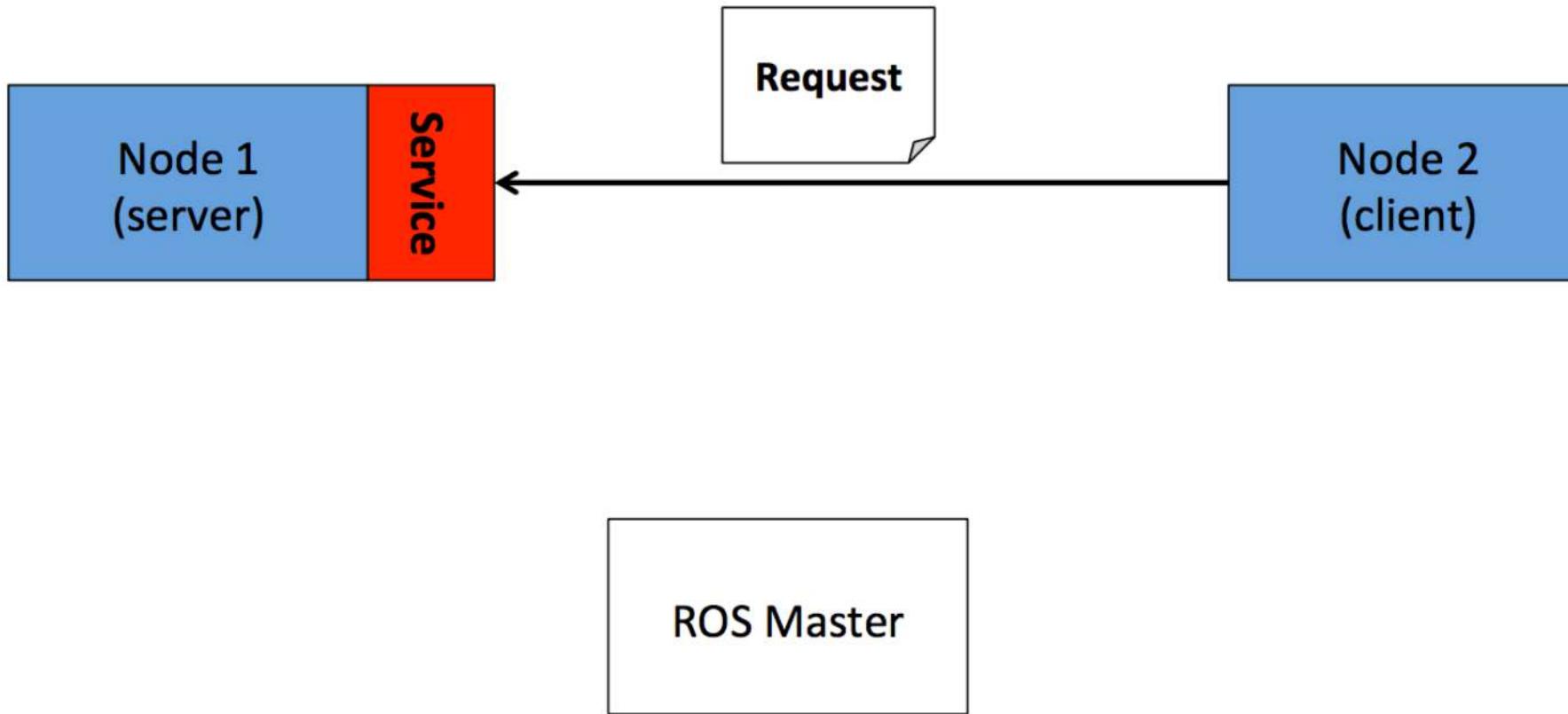
Synchronous, one-to-many

Communications: Client/Service



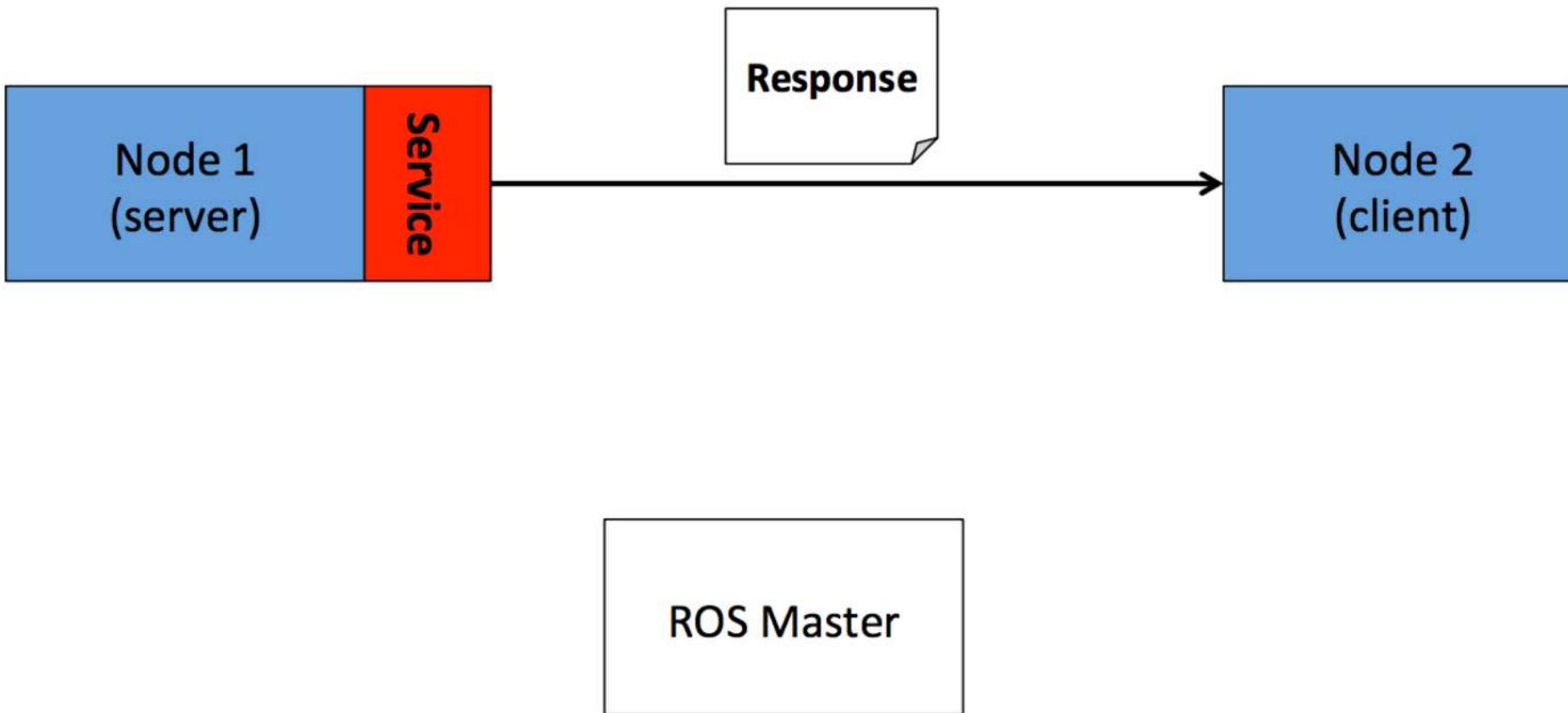
Synchronous, one-to-many

Communications: Client/Service



Synchronous, one-to-many

Communications: Client/Service



Synchronous, one-to-many

rospy client library: Services

ROS Services are defined by `srv` files, which contains a *request* message and a *response* message.

rospy converts these `srv` files into Python source code and creates three classes that you need to be familiar with: **service definitions, request messages, and response messages**.

The names of these classes come directly from the `srv` filename:

`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`

Branch: master ▾

[f1tenths-course-labs](#) / [beginner_tutorials](#) / [srv](#) / [AddTwoInts.srv](#)



madhurbehl name

1 contributor

5 lines (4 sloc) | 30 Bytes

```
1 int64 a
2 int64 b
3 ---
4 int64 sum
```

rospy client library: Service proxies

Create a callable proxy to a service.

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

Wait until a service becomes available.

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

rospy client library: Calling Service proxies

1. You call a service by creating a `rospy.ServiceProxy` with the name of the service you wish to call.
2. You often will want to call `rospy.wait_for_service()` to block until a service is available.
3. If a service returns an error for the request, a `rospy.ServiceException` will be raised. The exception will contain any error messages that the service sent.

rospy client library: Service call example

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

rospy client library: Calling services

Explicit style:

The explicit style is simple: you create your own `*Request` instance and pass it to publish, e.g.:

```
req = rospy_tutorials.srv.AddTwoIntsRequest(1, 2)
resp = add_two_ints(req)
```

Implicit style with in-order arguments:

In the *in-order* style, a new `Message` instance will be created with the arguments provided, in order. The argument order is the same as the order of the fields in the `Message`, and you must provide a value for all of the fields. For example, `rospy_tutorials.srv.AddTwoIntsRequest` has two integer fields:

```
resp = add_two_ints(1, 2)
```

Implicit style with keyword arguments

In the *keyword* style, you only initialize the fields that you wish to provide values for. The rest receive default values (e.g. 0, empty string, etc...). For example, `rospy_tutorials.srv.AddTwoIntsRequest` has two fields: `a` and `b`, which are both integers. You could call:

```
resp = add_two_ints(a=1)
```

which will set `a` to 1 and give `b` its default value of 0

rospy client library: Services

ROS Services are defined by `srv` files, which contains a *request* message and a *response* message.

rospy converts these `srv` files into Python source code and creates three classes that you need to be familiar with: **service definitions, request messages, and response messages**.

The names of these classes come directly from the `srv` filename:

`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`

Creating a ROS srv

- An **.srv** file describes a service. It is composed of two parts: a request and a response. Service files are stored in the *package_name/srv* folder.
- The field types you can use are:
 - int8, int16, int32, int64 (plus uint*)
 - float32, float64
 - string
 - time, duration
 - Header
 - other msg files
 - variable-length array[] and fixed-length array[C]

Creating a ROS srv

- Create a **srv** folder inside the *beginner_tutorials* folder

```
$ rosdep init  
$ rosdep update  
$ roscd beginner_tutorials  
$ mkdir srv  
$ cd srv
```

- Create a file named **AddTwoInts.srv** with the following content:

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

Creating a ROS srv

- Open the file **package.xml** and add these two lines:

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

- Open the file **CMakeList.txt** and add/uncomment:

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

Writing add_two_ints_server.py

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

from beginner_tutorials.srv import *
import rospy

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

def add_two_ints_server(): We declare our node using init_node() and then declare our service
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints',
                      AddTwoInts, handle_add_two_ints)
    print "Ready to add two ints." All requests are passed to handle_add_two_ints function
    rospy.spin() rospy.spin() keeps your code from exiting until the service is shutdown.

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

Writing add_two_ints_client.py

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

```
import sys
import rospy
from beginner_tutorials.srv import *

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
```

```
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))
```

For clients you don't have to call init_node()

Convenient method that blocks until the service named add_two_ints is available

Create a handle for calling the service

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

Build and run

- Make the scripts executable:

```
$ chmod +x add_two_ints_client.py  
$ chmod +x add_two_ints_server.py
```

- Build the service message as usual:

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

- Run it as before:

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

	Topics	Services
active things	rostopic	rosservice
data types	rosmsg	rossrv

```
$ rosservice list
```

You will obtain the following output:

```
/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
```

If you want to see the type of any service, for example, the /clear service, use the following command:

```
$ rosservice type /clear
```

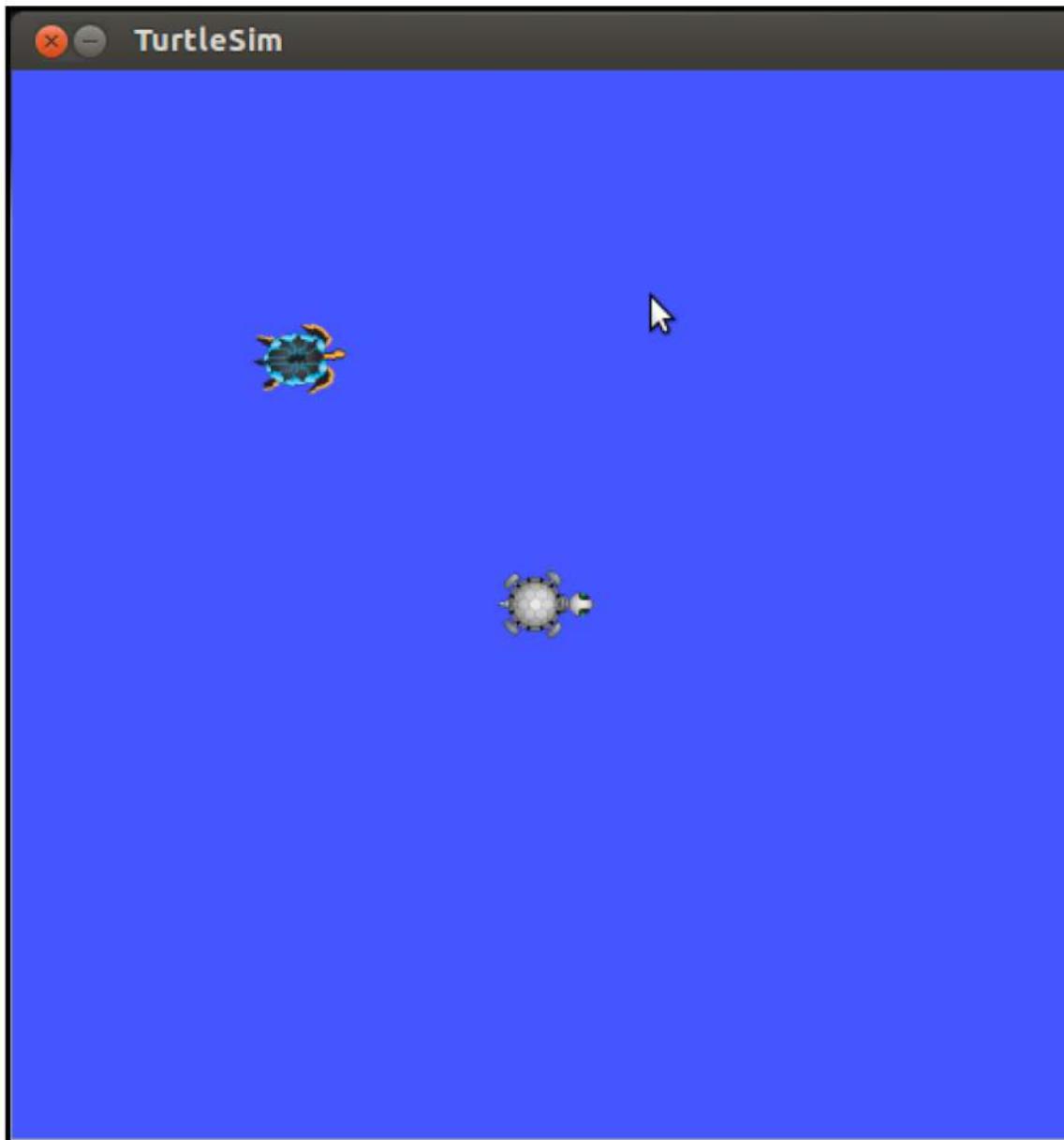
```
$ rosservice type /spawn | rossrv show
```

You will see something similar to the following output:

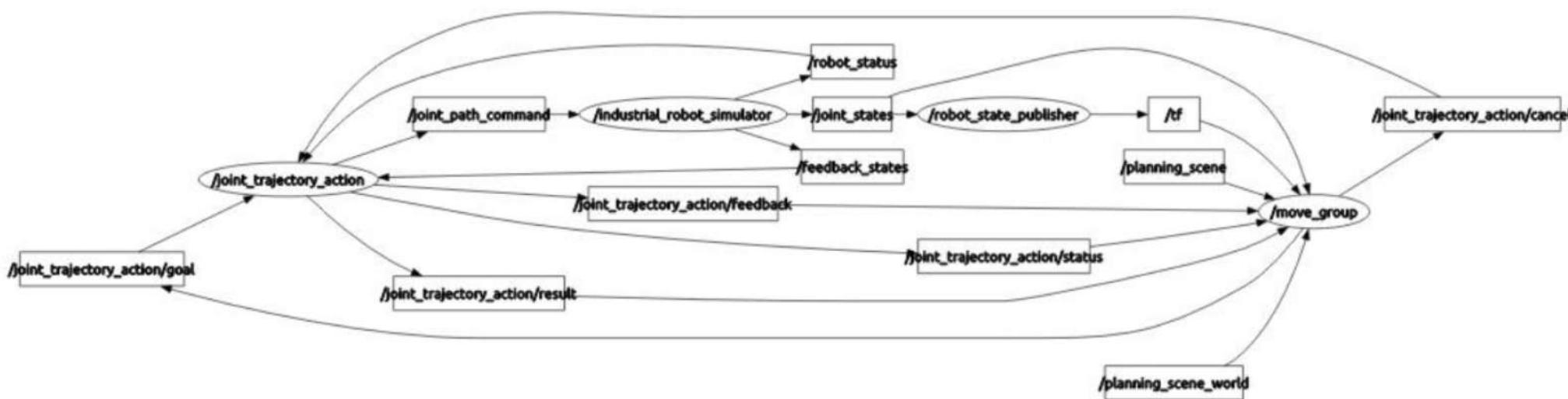
```
float32 x
float32 y
float32 theta
string name
---
string name
```

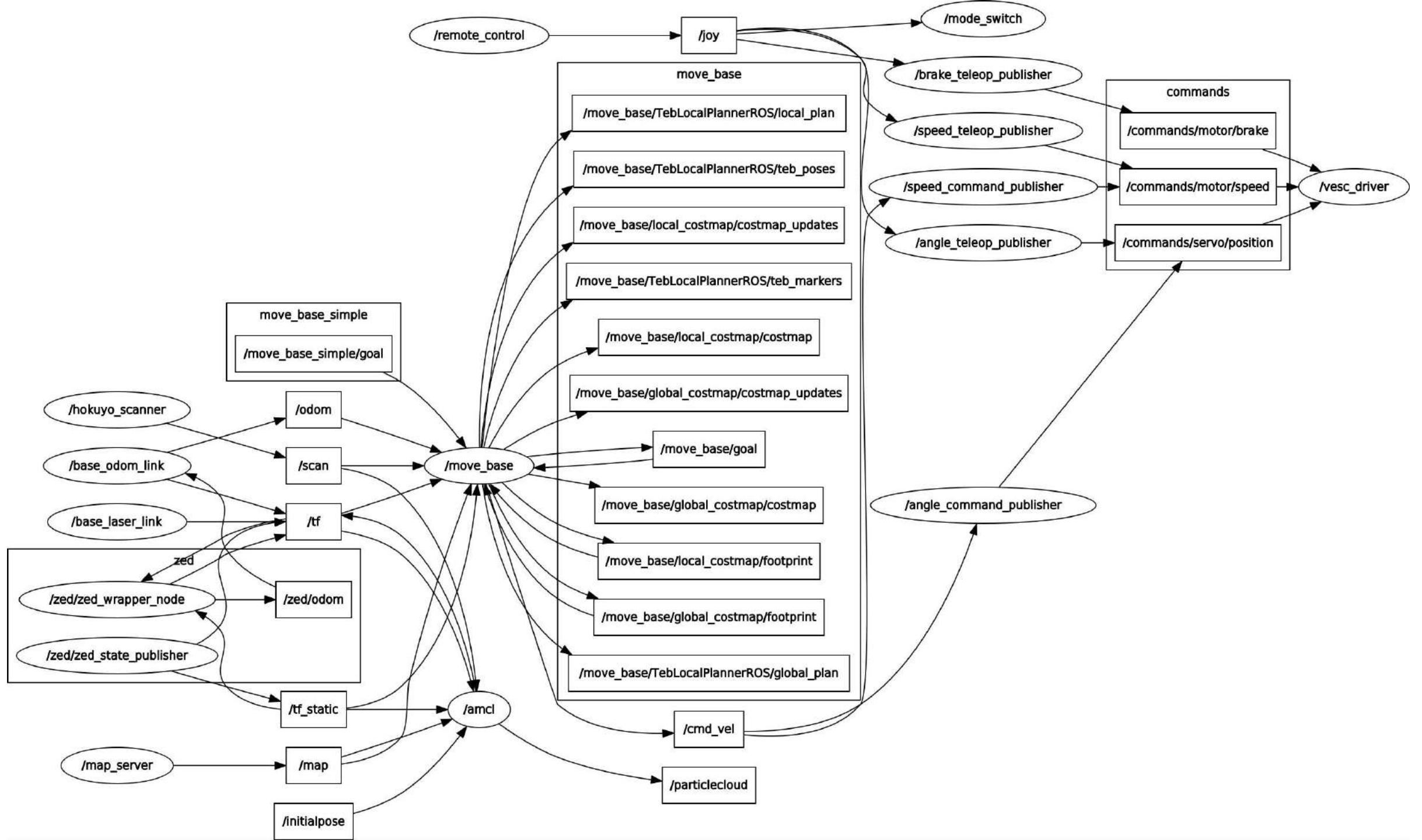
```
$ rosservice call /spawn 3 3 0.2 "new_turtle"
```

We then obtain the following result:



- ROS is a Distributed System
 - often 10s of nodes, plus configuration data
 - painful to start each node “manually”





Launch files

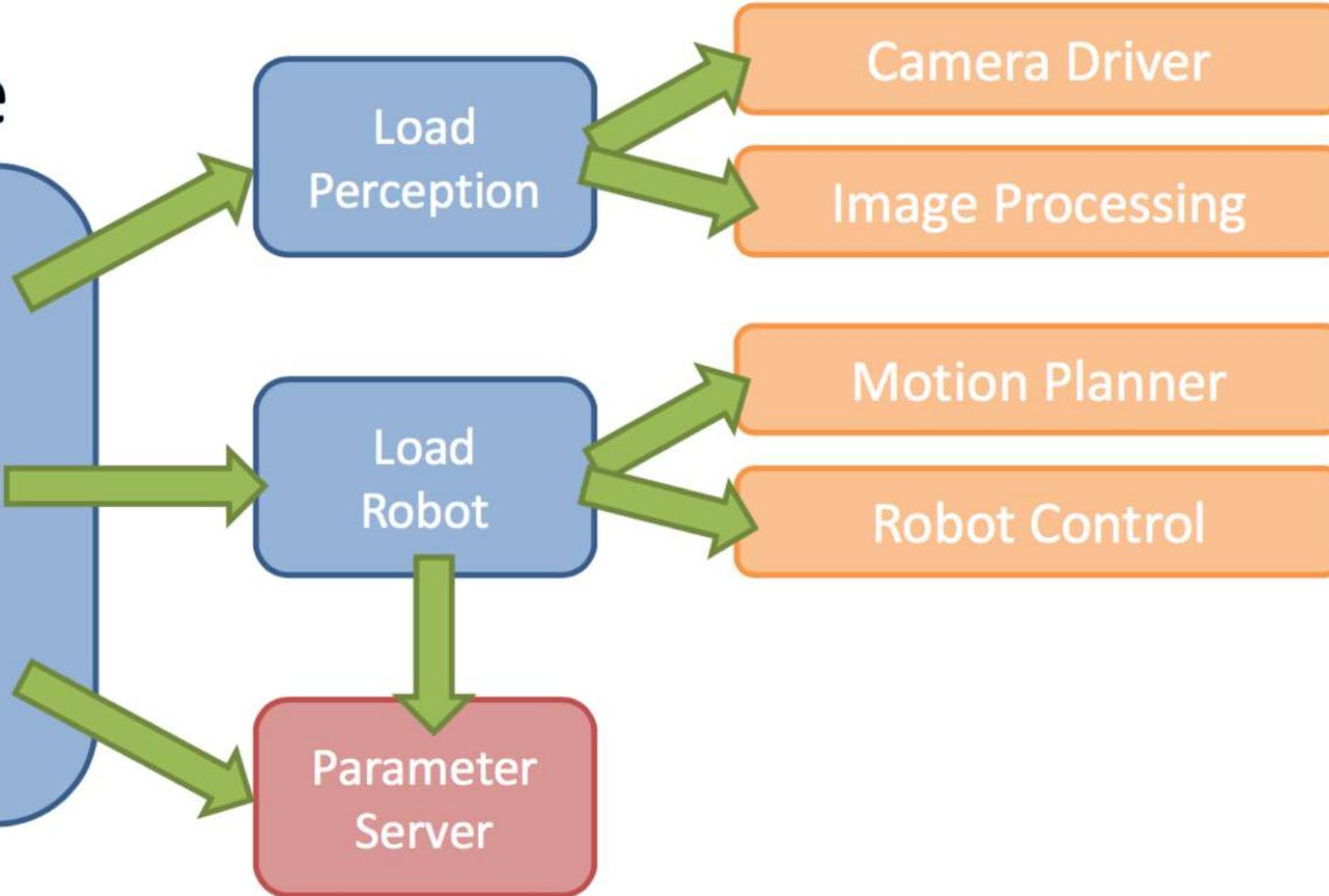


`roslaunch package_name file.launch`

Launch Files are like **Startup Scripts**

Launch File

Load
Robot
System



Launch Files: Notes

- Can launch *other* launch files
- Executed in order, without pause or wait*
 - * *Parameters set to parameter server before nodes are launched*
- Can accept arguments
- Can perform simple IF-THEN operations
- Supported parameter types:
 - Bool, string, int, double, text file, binary file

Example: Launch files

- Open an editor and save the following as `~/turtlesim_test.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>
```

start the launch file with the launch tag, so that the file is identified as a launch file

```
<group ns="turtlesim1">  
  <node pkg="turtlesim" name="sim" type="turtlesim_node"/>  
</group>  
  
<group ns="turtlesim2">  
  <node pkg="turtlesim" name="sim" type="turtlesim_node"/>  
</group>
```

start two groups with a namespace tag of turtlesim1 and turtlesim2 with a turtlesim node with a name of sim.
This allows us to start two simulators without having name conflicts.

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

start the mimic node with the topics input and output renamed to turtlesim1 and turtlesim2.
This renaming will cause turtlesim2 to mimic turtlesim1.

```
</launch>
```

This closes the xml tag for the 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
```



Example launch file for the F1/10 car

```
1 <launch>
2
3   <arg name="debug" default="false" />
4   <arg name="bag_path" default="$(find autonomous_driving)/bags/calibrated_140.bag"/>
5
6   <node pkg="rosbag" type="play" name="player" output="screen" args="-l $(arg bag_path)" if="$(arg debug)" ns="autonomous_driving"/>
7
8   <include file="$(find autonomous_driving)/launch/camera.launch">
9     <arg name="debug" value="$(arg debug)" />
10  </include>
11  <include file="$(find autonomous_driving)/launch/lane_detection.launch">
12    <arg name="debug" value="$(arg debug)" />
13  </include>
14  <include file="$(find autonomous_driving)/launch/object_detection.launch">
15    <arg name="debug" value="$(arg debug)" />
16  </include>
17  <include file="$(find autonomous_driving)/launch/rovercontrol.launch">
18    <arg name="debug" value="$(arg debug)" />
19  </include>
20 </launch>
```

rosbags: Recording and playing back data

all published topics should be accumulated in a bag file.

```
mkdir ~/bagfiles  
cd ~/bagfiles  
rosbag record -a
```

ROS Bags

- A *bag* is a format for storing message data
- Binary format with file extension *.bag
- Suited for logging and recording datasets for later visualization and analysis

Record all topics in a bag

```
> rosbag record --all
```

Record given topics

```
> rosbag record topic_1 topic_2 topic_3
```

Stop recording with Ctrl + C

Bags are saved with start date and time as file name in the current folder (e.g. 2017-02-07-01-27-13.bag)

rosbags: Examining and playing the bag file

```
rosbag info <your bagfile>
```

```
path:          2014-12-10-20-08-34.bag           year, date, and time and the suffix .bag
version:       2.0
duration:     1:38s (98s)
start:        Dec 10 2014 20:08:35.83 (1418270915.83)
end:          Dec 10 2014 20:10:14.38 (1418271014.38)
size:         865.0 KB
messages:     12471
compression:  none [1/1 chunks]
types:        geometry_msgs/Twist [9f195f881246fd...]
              rosgraph_msgs/Log  [acffd30cd6b6de30f120938c17c593fb]
              turtlesim/Color   [353891e354491c51aabe32df673fb446]
              turtlesim/Pose    [863b248d5016ca62ea2e895ae5265cf9]
topics:       /rosout                 4 msgs   : rosgraph_msgs/Log   (2 connections)
              /turtle1/cmd_vel   169 msgs  : geometry_msgs/Twist
              /turtle1/color_sensor 6149 msgs : turtlesim/Color
              /turtle1/pose      6149 msgs : turtlesim/Pose
```

rosbags: Playing back data

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

```
[ 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.
```

rosbags: Playing back data

```
rosbag play -r 2 <your bagfile>
```

allows you to change the rate of publishing by a specified factor.

rosbags: Recording specific topics

```
rosbag record -O subset /turtle1/cmd_vel /turtle1/pose
```

-O argument tells **rosbag record** to log to a file named subset.bag

rosbags: Recording specific topics

```
path:          subset.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
```

Sensors Udacity Lincoln MKZ

Camera 3x Blackfly GigE Camera, 20 Hz

Lidar Velodyne HDL-32E, 9.5 Hz

IMU Xsens, 400 Hz

GPS 2x fixed, 1 Hz

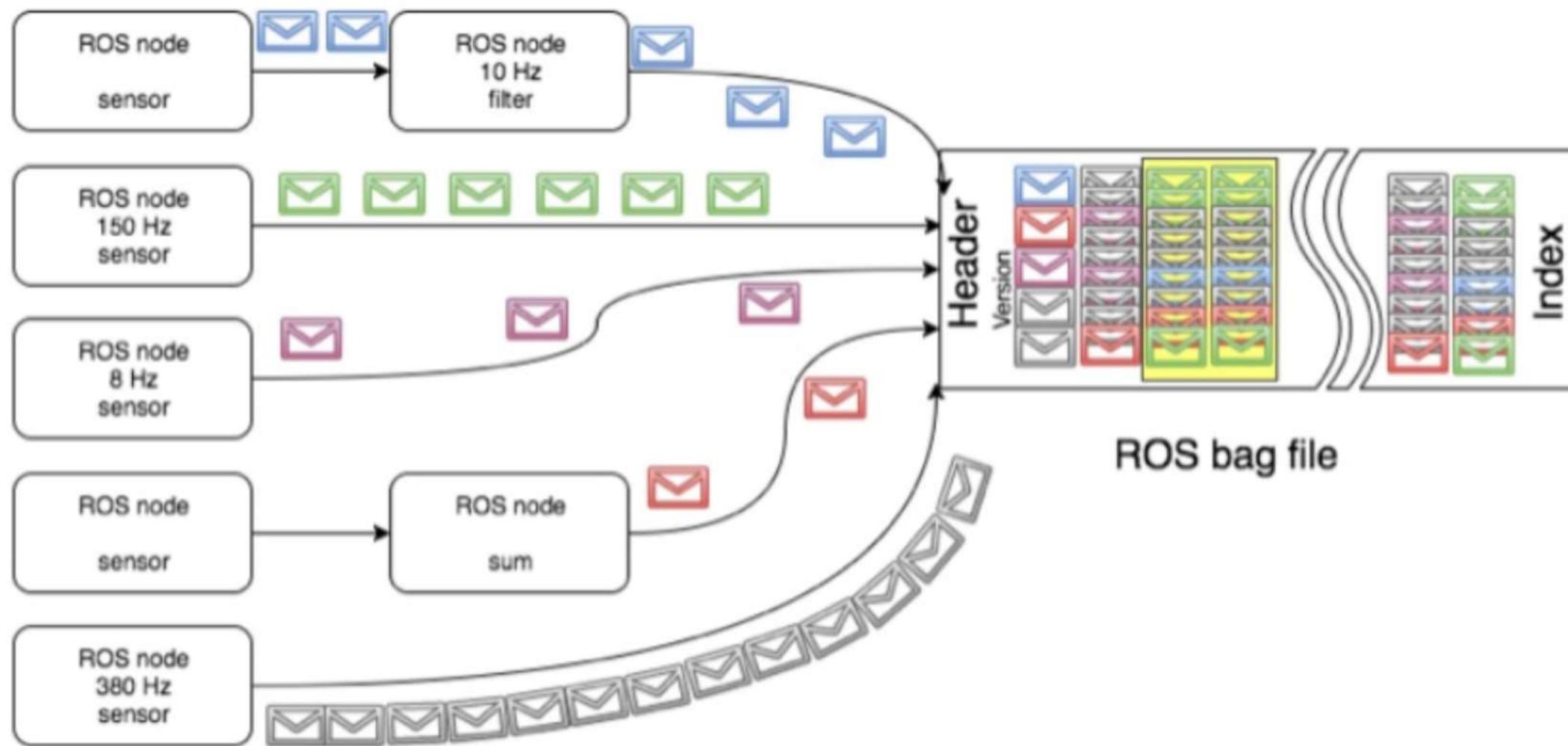
CAN bus, 1,1 kHz

Robot Operating System

Data 3 GB per minute

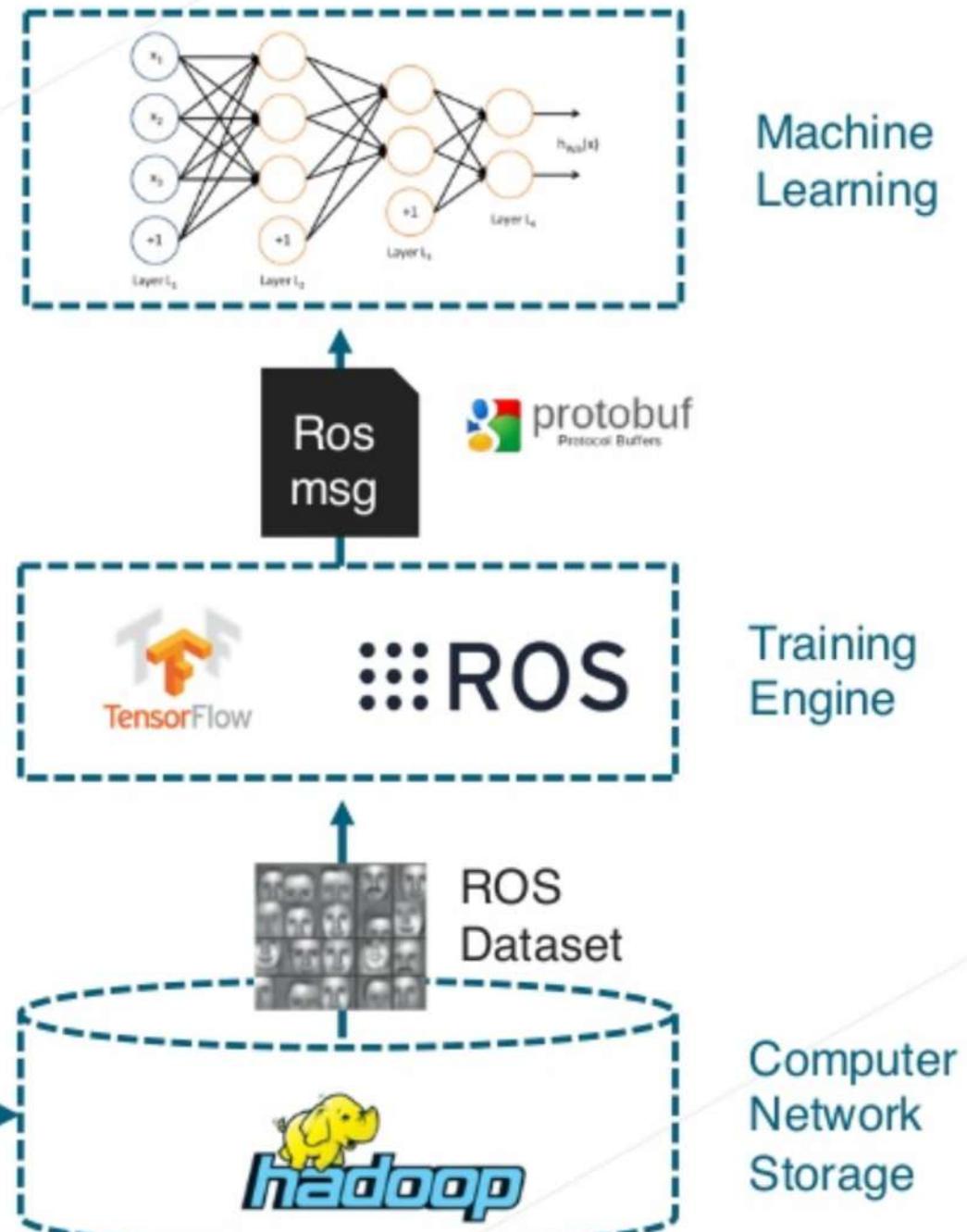
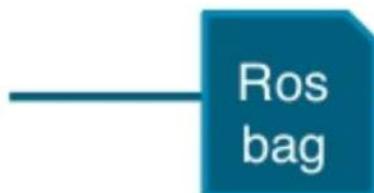


ROS bag data structure



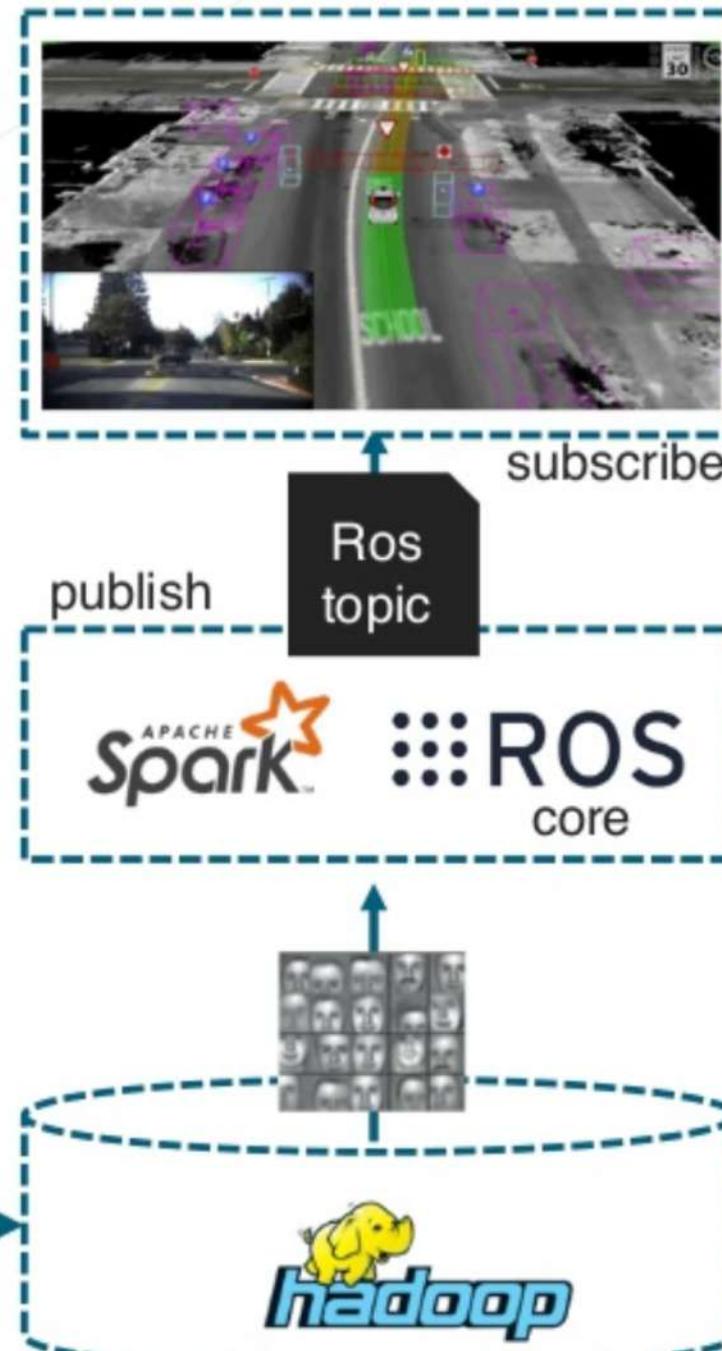
Training & Evaluation

- + Tensorflow ROSRecordDataset
- + Protocol Buffers to serialize records
- + Save time because data conversion not needed
- + Save storage because data duplication not needed



Re-Simulation & Testing

- + Use Spark for preprocessing, transformation, cleansing, aggregation, time window selection before publish to ROS topics
- + Use Re-Simulation framework of choice to subscribe to the ROS topics



Re-Simulation
with framework
of choice

Engine

Computer
Network
Storage

Runtime Manager

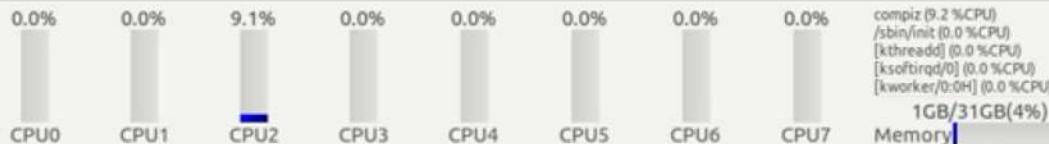
[Quick Start](#) [Setup](#) [Map](#) [Sensing](#) [Computing](#) [Interface](#) [Database](#) [Simulation](#) [Status](#) [Topics](#)

/media/ando/ssd/log/rosbag/rosbag.bag

Ref

Rate: Start Time (s): Repeat[Play](#) [Stop](#) [Pause](#)

path: /media/ando/ssd/log/rosbag/rosbag.bag
version: 2.0
duration: 11:35s (695s)
start: Jun 23 2015 10:57:55.84 (1435024675.84)
end: Jun 23 2015 11:09:31.80 (1435025371.80)
size: 26.1 GB
messages: 17393
compression: none [17393/17393 chunks]
types: sensor_msgs/Image [060021388200f6f0f447d0fc9c64743]
sensor_msgs/PointCloud2 [1158d486dd51d683ce2f1be655c3c181]
topics: /image_raw 10439 msgs :sensor_msgs/Image
/points_raw 6954 msgs :sensor_msgs/PointCloud2

[ROSBAG](#) [RViz](#) [RQT](#)

AutoWare

