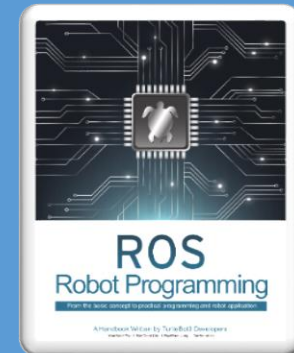


Basic ROS Programming

ROBOTIS

KAIST



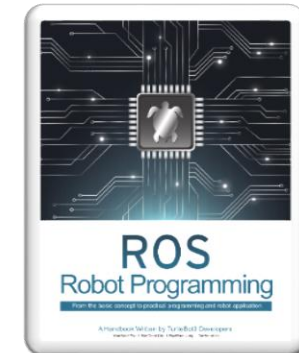
You Tube

Subscribe

Textbook
P. 148~193

Contents

- I. Things to know before programming ROS
- II. Create and run publishers and subscriber nodes
- III. Create and run service servers and client nodes
- IV. Create and run action servers and client nodes
- V. How to use parameters
- VI. How to use roslaunch



You Tube

 **Subscribe**

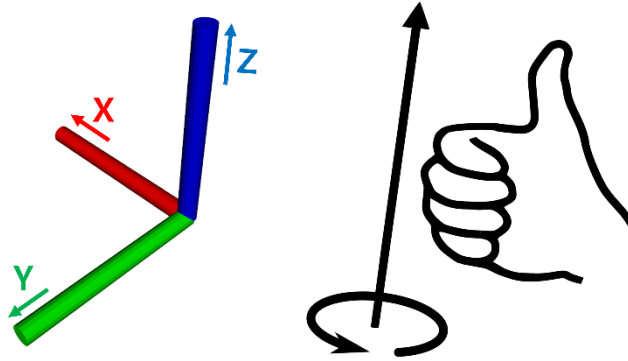
Textbook
P. 148~193

Things to know before programming ROS

Things to know before programming ROS

- Standard unit

- SI unit



Quantity	Unit	Quantity	Unit
angle	radian	length	meter
frequency	hertz	mass	kilogram
force	newton	time	second
power	watt	current	ampere
voltage	volt	temperature	celsius

- Coordinate representation

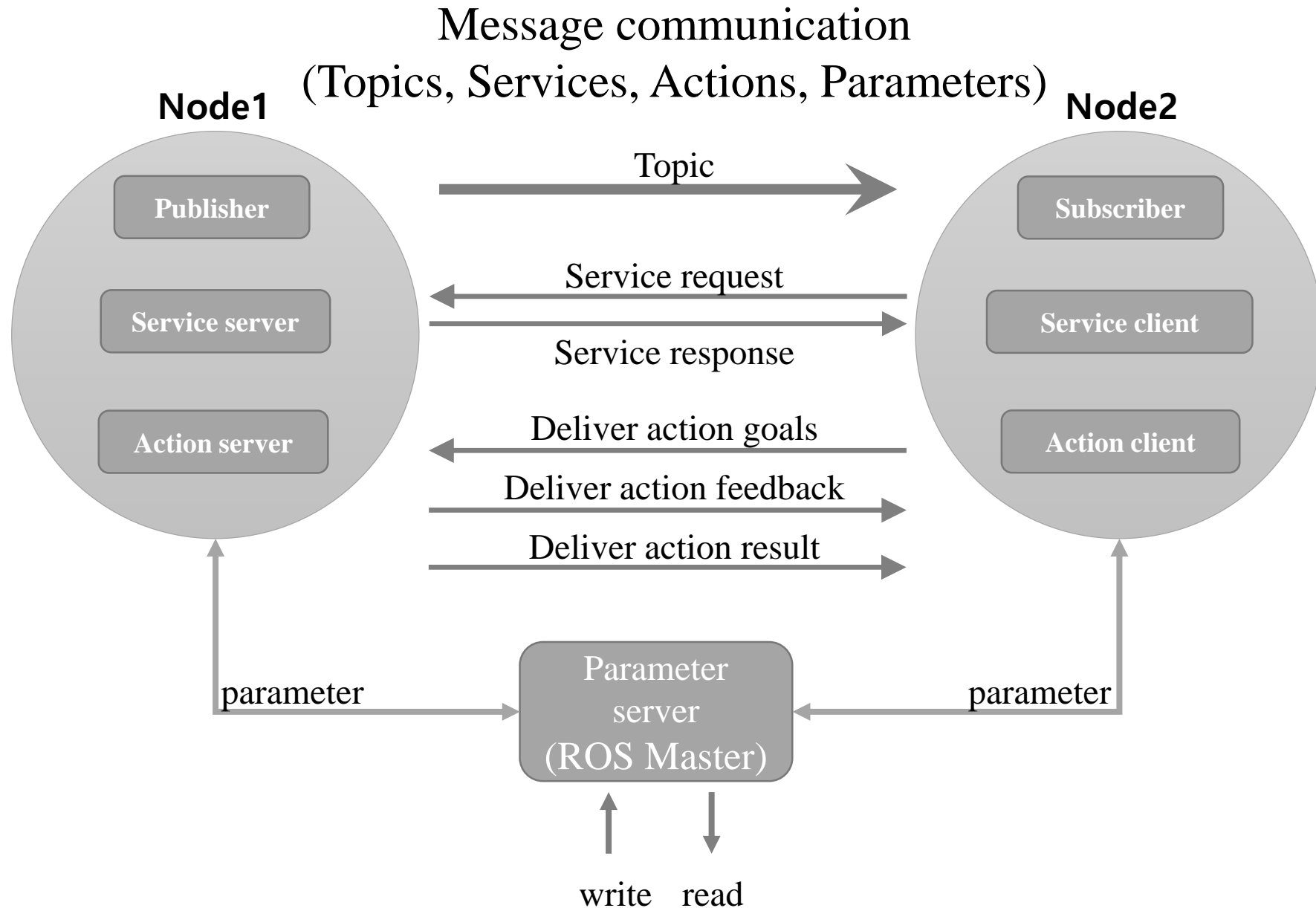
- x: forward, y: left, z: up
- Right-hand rule

- Programing rules

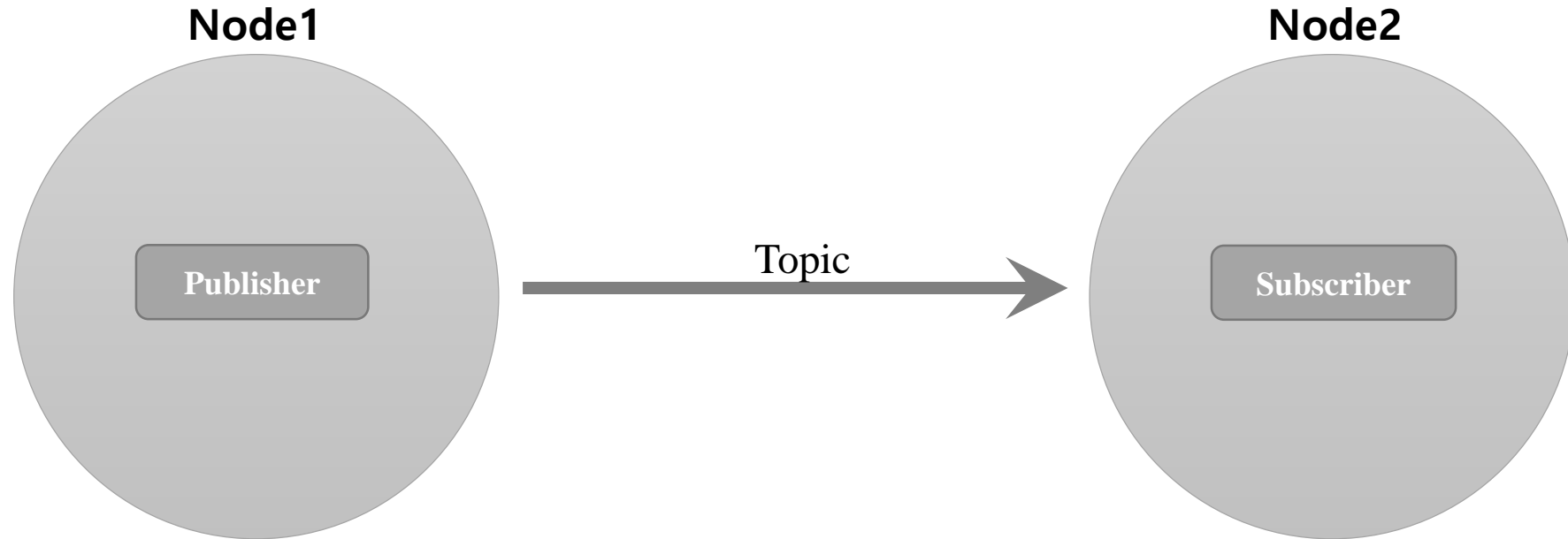
Object	Naming rule	example
Package	under_scored	Ex) first_ros_package
Topic, service	under_scored	Ex) raw_image
File	under_scored	Ex) turtlebot3_fake.cpp
Namespace	under_scored	Ex) ros_awesome_package
Variable	under_scored	Ex) string table_name;
type	CamelCased	Ex) typedef int32_t PropertiesNumber;
Class	CamelCased	Ex) class UrlTable
Structure	CamelCased	Ex) struct UrlTableProperties
Enumeration type	CamelCased	Ex) enum ChoiceNumber
Function	camelCased	Ex) addTableEntry();
Method	camelCased	Ex) void setNumEntries(int32_t num_entries)
Constant	ALL_CAPITALS	Ex) const uint8_t DAYS_IN_A_WEEK = 7;

Types of message communication

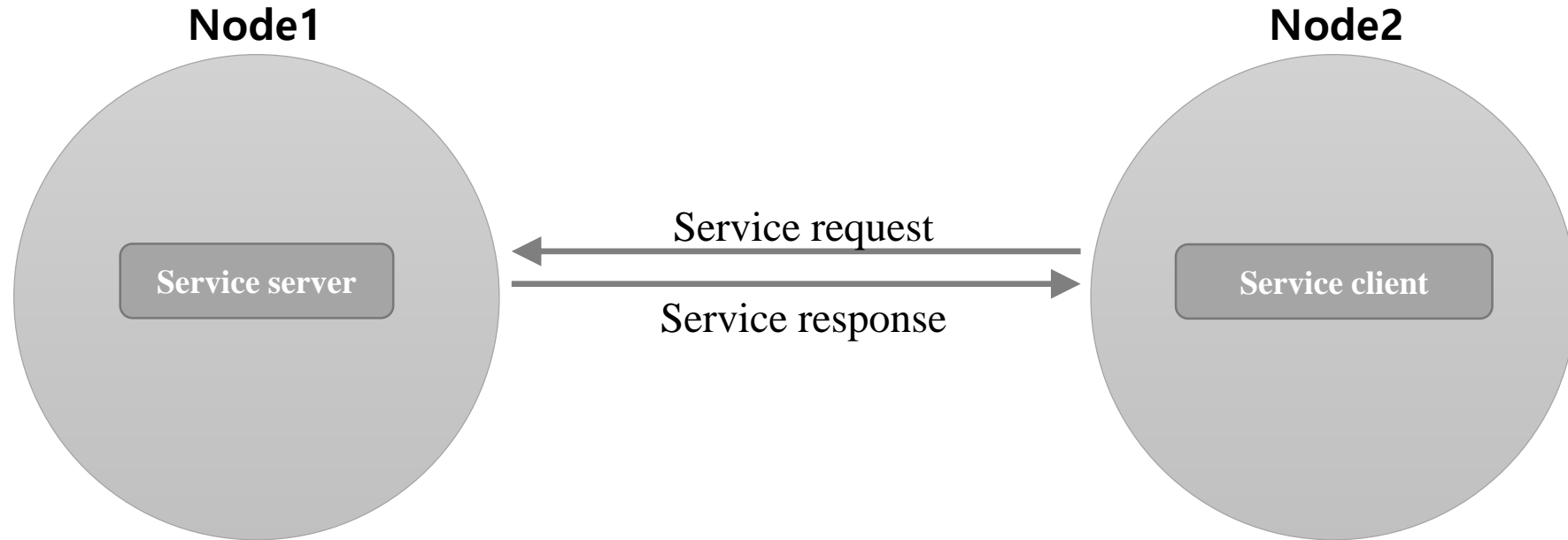
ROS Message communication



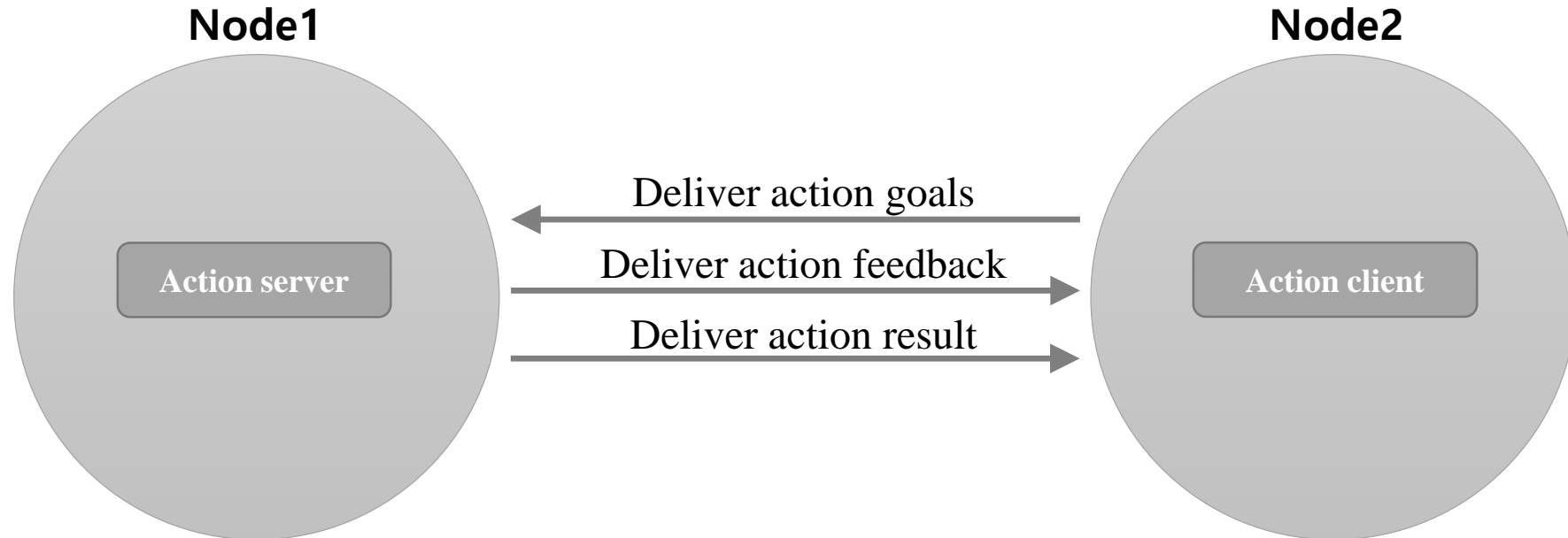
Topic



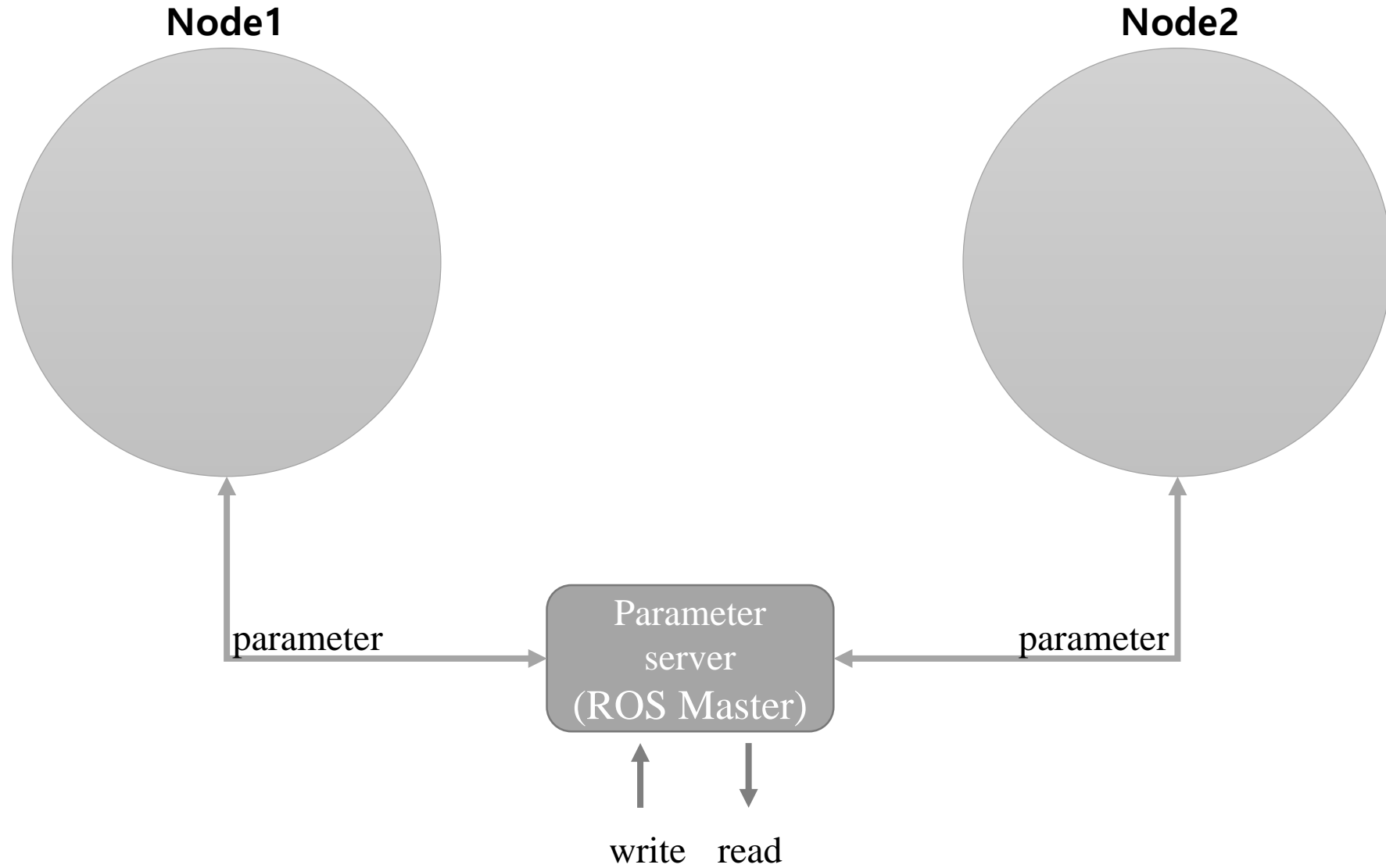
Service



Action



Parameter

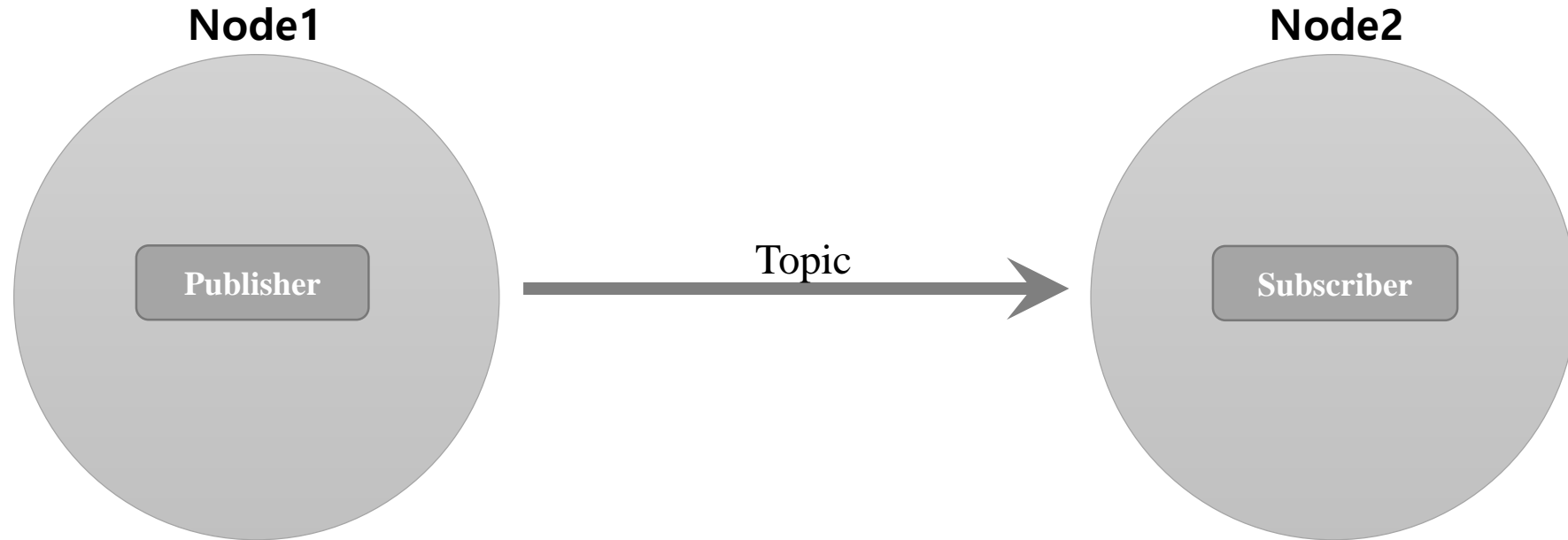


Now let's write codes

Topic: publisher, subscriber

Service: service server, service client

Topic



Topic / Publisher / Subscriber

- ROS uses 'Topic' message communication for unidirectional communication. In this tutorial, the transmitter is called "Publisher" and the receiver is called "Subscriber".

1) Creating the package

```
$ cd ~/catkin_ws/src  
$ catkin_create_pkg ros_tutorials_topic message_generation std_msgs roscpp
```

```
$ cd ros_tutorials_topic  
$ ls  
include          → header file folder  
src              → source code folder  
CMakeLists.txt   → build configuration file  
package.xml      → package configuration file
```

Topic / Publisher / Subscriber

2) Modify the package configuration file (package.xml)

- One of the required ROS configuration files, package.xml, is an XML file containing package information that describes the package name, author, license, and dependency package.

```
$ gedit package.xml
```

```
<?xml version="1.0"?>
<package>
  <name>ros_tutorials_topic</name>
  <version>0.1.0</version>
  <description>ROS tutorial package to learn the topic</description>
  <license>Apache License 2.0</license>
  <author email="pyo@robotis.com">Yoonseok Pyo</author>
  <maintainer email="pyo@robotis.com">Yoonseok Pyo</maintainer>
  <url type="bugtracker">https://github.com/ROBOTIS-GIT/ros_tutorials/issues</url>
  <url type="repository">https://github.com/ROBOTIS-GIT/ros_tutorials.git</url>
  <url type="website">http://www.robotis.com</url>
```

Topic / Publisher / Subscriber

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>std_msgs</build_depend>
<build_depend>message_generation</build_depend>
<run_depend>roscpp</run_depend>
<run_depend>std_msgs</run_depend>
<run_depend>message_runtime</run_depend>
<export></export>
</package>
```

Topic / Publisher / Subscriber

3) Modify build configuration file (CMakeLists.txt)

```
$ gedit CMakeLists.txt
```

```
cmake_minimum_required(VERSION 2.8.3)
project(ros_tutorials_topic)
```

```
## This is the component package required for catkin build.
```

```
## dependency packages are message_generation, std_msgs, and roscpp. If these packages do not exist, an error occurs when you build.
```

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

```
## Message declaration: MsgTutorial.msg
```

```
add_message_files(FILES MsgTutorial.msg)
```

```
## This is an option to configure dependent messages.
```

```
## If std_msgs is not installed, an error occurs when you build.
```

```
generate_messages(DEPENDENCIES std_msgs)
```

```
## The catkin_package option describes the library, catkin build dependencies, and system dependent packages.
```

```
catkin_package(
```

```
  LIBRARIES ros_tutorials_topic
```

```
  CATKIN_DEPENDS std_msgs roscpp
```

```
)
```


Topic / Publisher / Subscriber

Set the include directory.

```
include_directories(${catkin_INCLUDE_DIRS})
```

Build option for the topic_publisher node.

Configure the executable file, target link library, and additional dependencies.

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

```
add_dependencies(topic_publisher ${${PROJECT_NAME}_EXPORTED_TARGETS}  
${catkin_EXPORTED_TARGETS})
```

```
target_link_libraries(topic_publisher ${catkin_LIBRARIES})
```

Build option for the topic_subscriber node.

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

```
add_dependencies(topic_subscriber ${${PROJECT_NAME}_EXPORTED_TARGETS}  
${catkin_EXPORTED_TARGETS})
```

```
target_link_libraries(topic_subscriber ${catkin_LIBRARIES})
```

Topic / Publisher / Subscriber

4) Create message file

- Add the following option in the CMakeLists.txt file.

```
add_message_files(FILES MsgTutorial.msg)
```

- This commands means a message should be built based on MsgTutorial.msg when it is built.

```
$ roscd ros_tutorials_topic      → Move to package folder
$ mkdir msg                     → Create a message folder named msg in the package
$ cd msg                        → Move to the msg folder you created
$ gedit MsgTutorial.msg         → Create new MsgTutorial.msg file and modify contents
```

- Time (message format), stamp (message name)
- int32 (message type), data (message name)
- In addition to time and int32, message types include message basic types such as bool, int8, int16, float32, string, time, duration, and common_msgs which collect messages used in ROS. Here we use time and int32 to create a simple example. (See Appendix C and <http://wiki.ros.org/msg>)

```
time stamp
int32 data
```

Topic / Publisher / Subscriber

5) Creating the Publisher Node

- Add the option to generate the following executable file in the CMakeLists.txt file.

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

- Build a file called topic_publisher.cpp in the src folder to create an executable called topic_publisher

```
$ roscd ros_tutorials_topic/src      → Move to the src folder, which is the source folder of the package
$ gedit topic_publisher.cpp          → New source file and modify contents
```

```
#include "ros/ros.h"                // ROS basic header file
#include "ros_tutorials_topic/MsgTutorial.h" // MsgTutorial message file header(Auto-generated after build)

int main(int argc, char **argv)      // Node main function
{
    ros::init(argc, argv, "topic_publisher"); // Node name initialization
    ros::NodeHandle nh;                // Node handle declaration for communication with ROS system
```

Topic / Publisher / Subscriber

```
// Publisher declaration, using MsgTutorial message file from ros_tutorials_topic package
// Create the publisher ros_tutorial_pub. The topic name is "ros_tutorial_msg"
// set the publisher queue size to 100
ros::Publisher ros_tutorial_pub = nh.advertise<ros_tutorials_topic::MsgTutorial>("ros_tutorial_msg", 100);

// Set the loop period. & Quot; 10 & quot; refers to 10 Hz and repeats at 0.1 second intervals
ros::Rate loop_rate(10);

// Declare msg message in MsgTutorial message file format
ros_tutorials_topic::MsgTutorial msg;

// Declare variable to be used in message
int count = 0;
```

Topic / Publisher / Subscriber

```
while (ros::ok())
{
    msg.stamp = ros::Time::now();           // Put the current time in the msg's stamp message
    msg.data = count;                       // Put the value of the variable count in the lower data message of msg

    ROS_INFO("send msg = %d", msg.stamp.sec); // Display the stamp.sec message
    ROS_INFO("send msg = %d", msg.stamp.nsec); // Display the stamp.nsec message
    ROS_INFO("send msg = %d", msg.data);      // Display data message

    ros_tutorial_pub.publish(msg);           // Publish message

    loop_rate.sleep();                       // Go to sleep according to the loop cycle defined above

    ++count;                                // Increment count variable by 1
}

return 0;
}
```

Topic / Publisher / Subscriber

6) Create subscriber node

- In the CMakeLists.txt file, add the option to generate the following executable file.

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

- That is, build a file named topic_subscriber.cpp to create an executable called topic_subscriber

```
$ roscd ros_tutorials_topic/src      → Move to the src folder, which is the source folder of the package
$ gedit topic_subscriber.cpp          → New source file and modify contents
```

```
#include "ros/ros.h"                // ROS basic header file
#include "ros_tutorials_topic/MsgTutorial.h" // MsgTutorial message file header (Auto-generated after build)
// A message callback function, named ros_tutorial_msg below
// This is a function that works when a message is received
// The input message is supposed to receive the MsgTutorial message from the ros_tutorials_topic package
void msgCallback(const ros_tutorials_topic::MsgTutorial::ConstPtr& msg)
{
    ROS_INFO("recieve msg = %d", msg->stamp.sec); // Display stamp.sec message
    ROS_INFO("recieve msg = %d", msg->stamp.nsec); // Display stamp.nsec message
    ROS_INFO("recieve msg = %d", msg->data);       // Display data message
}
```

Topic / Publisher / Subscriber

```
int main(int argc, char **argv)           // Node main function
{
    ros::init(argc, argv, "topic_subscriber"); // Initialize the node name

    ros::NodeHandle nh;                    // Declare a node handle to communicate with the ROS system

    // Subscriber declaration, using MsgTutorial message file from ros_tutorials_topic package
    // Create subscriber ros_tutorial_sub. The topic name is "ros_tutorial_msg"
    // Set the subscriber queue size to 100
    ros::Subscriber ros_tutorial_sub = nh.subscribe("ros_tutorial_msg", 100, msgCallback);

    // callback function will be waiting for message to be received,
    // Execute callback function when message is received
    ros::spin();

    return 0;
}
```

Topic / Publisher / Subscriber

7) Build the ROS node

- Build the message file, the publisher node, and the subscriber node in the `ros_tutorials_topic` package with the following command:

```
$ cd ~/catkin_ws      → Move to catkin folder  
$ catkin_make         → Execute catkin build
```

- [Reference] File system

- Source code file for `ros_tutorials_topic` package: `~/catkin_ws/src/ros_tutorials_topic/src`
- **Message file** for `ros_tutorials_topic` package: `~/catkin_ws/src/ros_tutorials_topic/msg`
- Built files are located in the `/build` and `/devel` folders in `/catkin_ws`
 - `/build` folder saves the settings used in the catkin build
 - `/devel/lib/ros_tutorials_topic` folder saves **executable file**
 - `/devel/include/ros_tutorials_topic` folder saves **message header file** that is automatically generated from message file

Topic / Publisher / Subscriber

8) Execute Publisher [Note: Do not forget to run roscore before running the node.]

- Below command runs the topic_publisher node in the ros_tutorials_topic package

```
$ roslaunch ros_tutorials_topic topic_publisher
```

```
File Edit View Search Terminal Help
pyo@pyo ~ $ roslaunch ros_tutorials_topic topic_publisher
[ INFO] [1499699973.660967562]: send msg = 1499699973
[ INFO] [1499699973.661016263]: send msg = 660910231
[ INFO] [1499699973.661026591]: send msg = 0
[ INFO] [1499699973.760999003]: send msg = 1499699973
[ INFO] [1499699973.761026640]: send msg = 760971041
[ INFO] [1499699973.761035687]: send msg = 1
[ INFO] [1499699973.861023149]: send msg = 1499699973
[ INFO] [1499699973.861052777]: send msg = 860995018
[ INFO] [1499699973.861061286]: send msg = 2
[ INFO] [1499699973.961021536]: send msg = 1499699973
[ INFO] [1499699973.961051473]: send msg = 960993409
[ INFO] [1499699973.961060450]: send msg = 3
[ INFO] [1499699974.061026451]: send msg = 1499699974
[ INFO] [1499699974.061070222]: send msg = 60993262
[ INFO] [1499699974.061080597]: send msg = 4
[ INFO] [1499699974.161000942]: send msg = 1499699974
[ INFO] [1499699974.161039542]: send msg = 160967676
[ INFO] [1499699974.161054694]: send msg = 5
[ INFO] [1499699974.261001301]: send msg = 1499699974
[ INFO] [1499699974.261039961]: send msg = 260968286
[ INFO] [1499699974.261054164]: send msg = 6
[ INFO] [1499699974.361024242]: send msg = 1499699974
[ INFO] [1499699974.361052420]: send msg = 360996035
```

Topic / Publisher / Subscriber

- [Reference] `rostopic`
 - You can use `rostopic` command to check the topic list, cycle, data bandwidth, and content of the current ROS network.

```
$ rostopic list  
/ros_tutorial_msg  
/rosout  
/rosout_agg
```

```
$ rostopic echo /ros_tutorial_msg
```

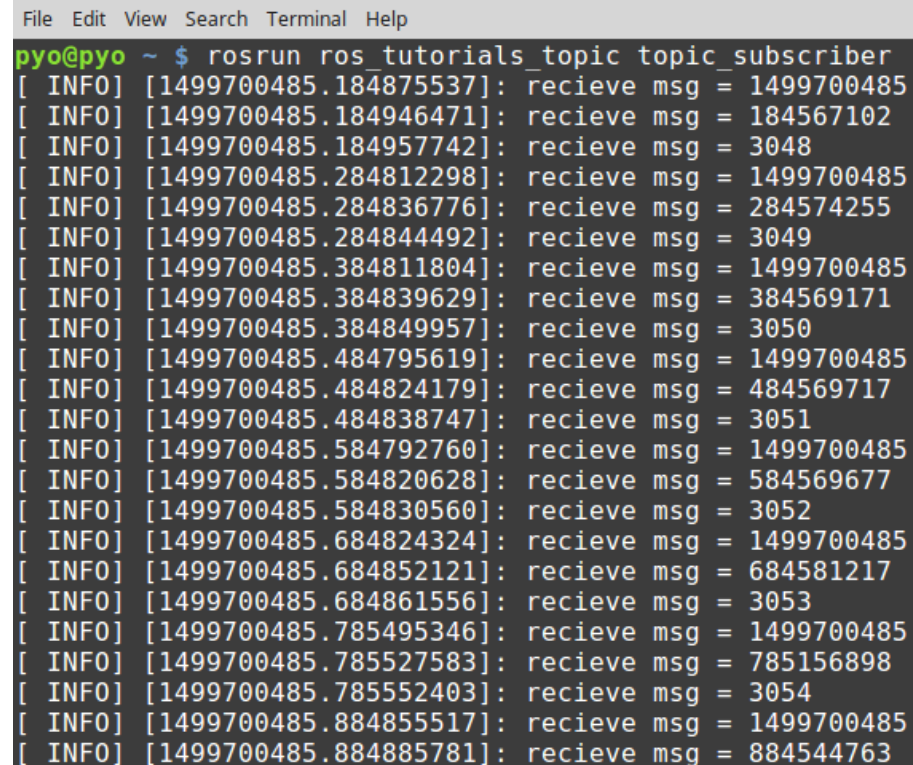
```
File Edit View Search Terminal Help  
pyo@pyo ~ $ rostopic echo /ros_tutorial_msg  
stamp:  
  secs: 1499700351  
  nsecs: 684514825  
data: 1713  
---  
stamp:  
  secs: 1499700351  
  nsecs: 784542724  
data: 1714  
---  
stamp:  
  secs: 1499700351  
  nsecs: 884544453  
data: 1715  
---  
stamp:  
  secs: 1499700351  
  nsecs: 984543934  
data: 1716  
---  
stamp:  
  secs: 1499700352  
  nsecs: 84543178
```

Topic / Publisher / Subscriber

9) Execute Subscriber

- Below command runs the topic_subscriber node of the ros_tutorials_topic package

```
$ roslaunch ros_tutorials_topic topic_subscriber
```

A terminal window with a menu bar (File, Edit, View, Search, Terminal, Help) and a dark background. The prompt is 'pyo@pyo ~'. The command 'roslaunch ros_tutorials_topic topic_subscriber' has been executed. The output consists of 20 lines of log messages, each starting with '[INFO]' followed by a timestamp and the text 'recieve msg ='. The timestamps are in the format [1499700485.timestamp]. The message values are: 1499700485, 184567102, 3048, 1499700485, 284574255, 3049, 1499700485, 384569171, 3050, 1499700485, 484569717, 3051, 1499700485, 584569677, 3052, 1499700485, 684581217, 3053, 1499700485, 785156898, 3054, 1499700485, 884544763.

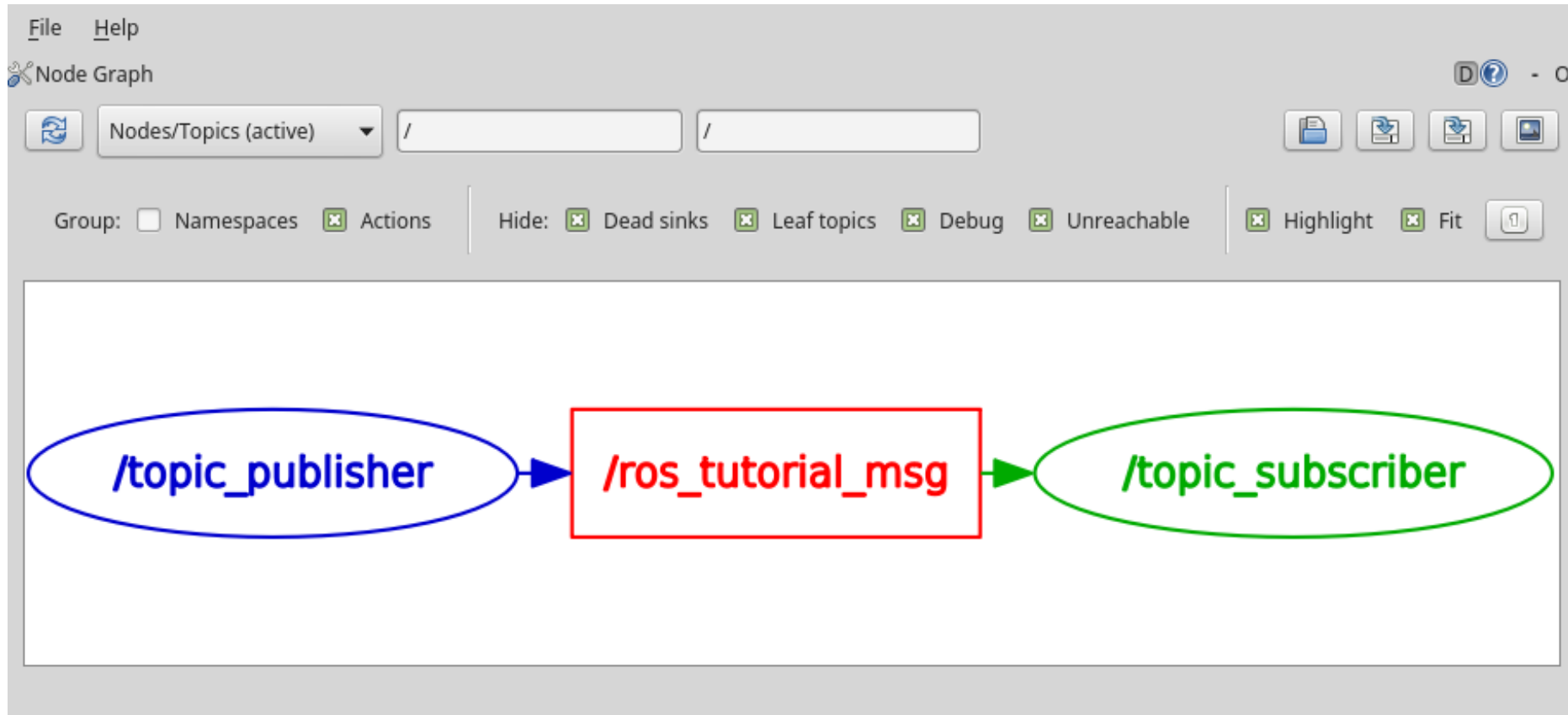
```
File Edit View Search Terminal Help
pyo@pyo ~ $ roslaunch ros_tutorials_topic topic_subscriber
[ INFO] [1499700485.184875537]: recieve msg = 1499700485
[ INFO] [1499700485.184946471]: recieve msg = 184567102
[ INFO] [1499700485.184957742]: recieve msg = 3048
[ INFO] [1499700485.284812298]: recieve msg = 1499700485
[ INFO] [1499700485.284836776]: recieve msg = 284574255
[ INFO] [1499700485.284844492]: recieve msg = 3049
[ INFO] [1499700485.384811804]: recieve msg = 1499700485
[ INFO] [1499700485.384839629]: recieve msg = 384569171
[ INFO] [1499700485.384849957]: recieve msg = 3050
[ INFO] [1499700485.484795619]: recieve msg = 1499700485
[ INFO] [1499700485.484824179]: recieve msg = 484569717
[ INFO] [1499700485.484838747]: recieve msg = 3051
[ INFO] [1499700485.584792760]: recieve msg = 1499700485
[ INFO] [1499700485.584820628]: recieve msg = 584569677
[ INFO] [1499700485.584830560]: recieve msg = 3052
[ INFO] [1499700485.684824324]: recieve msg = 1499700485
[ INFO] [1499700485.684852121]: recieve msg = 684581217
[ INFO] [1499700485.684861556]: recieve msg = 3053
[ INFO] [1499700485.785495346]: recieve msg = 1499700485
[ INFO] [1499700485.785527583]: recieve msg = 785156898
[ INFO] [1499700485.785552403]: recieve msg = 3054
[ INFO] [1499700485.884855517]: recieve msg = 1499700485
[ INFO] [1499700485.884885781]: recieve msg = 884544763
```

Topic / Publisher / Subscriber

10) Checking communication status of executed nodes

```
$ rqt_graph
```

```
$ rqt [Plugins] → [Introspection] → [Node Graph]
```



Source code

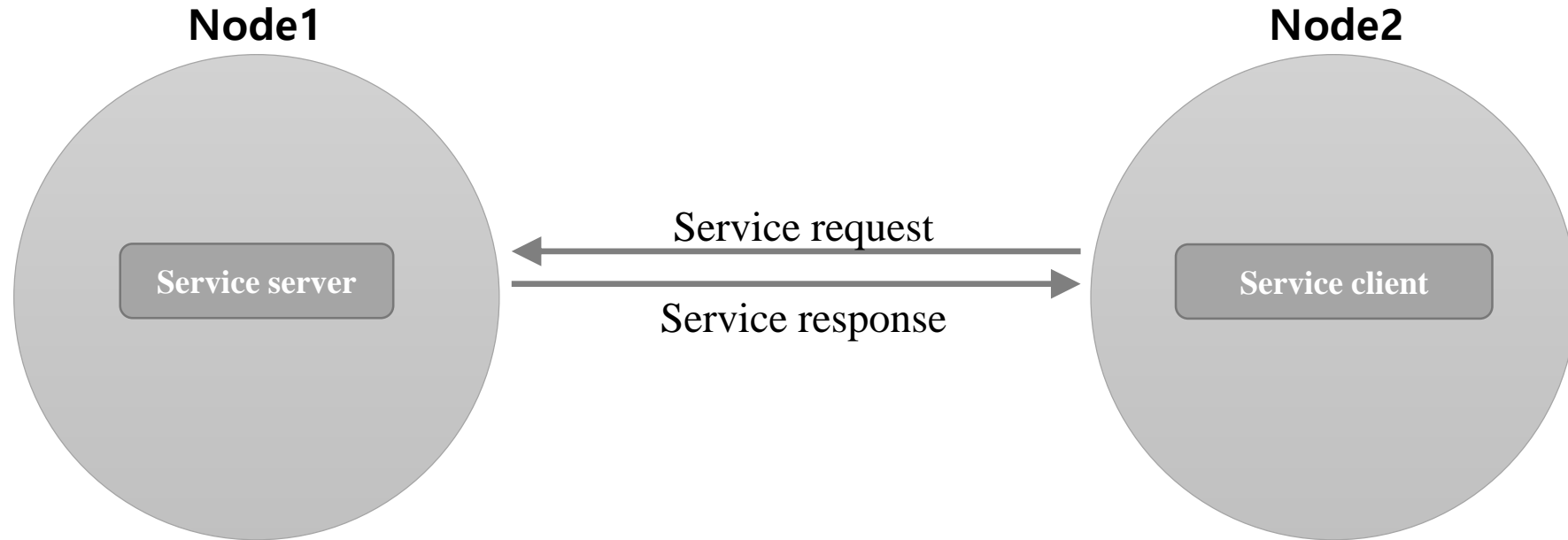
- We have created a publisher and a subscriber node that use topic, and executed them to learn how to communicate between nodes. The relevant sources can be found at the github address below.
- https://github.com/ROBOTIS-GIT/ros_tutorials/tree/master/ros_tutorials_topic
- If you want to apply it right away, you can clone the source code with the following command in the catkin_ws/src folder and build it. Then run the topic_publisher and topic_subscriber nodes.

```
$ cd ~/catkin_ws/src  
$ git clone https://github.com/ROBOTIS-GIT/ros_tutorials.git  
$ cd ~/catkin_ws  
$ catkin_make
```

```
$ rosrn ros_tutorials_topic topic_publisher
```

```
$ rosrn ros_tutorials_topic topic_subscriber
```

Service



Service / Service server / Service client

- A **service** is divided into the **service server** that responds only when there is a request and the **service client** that requests and responds. Unlike the topic, the service is a **one-time message communication**. Therefore, when the request and the response of the service are completed, the two connected nodes are disconnected.
- This service is often used as an instruction when a robot should perform a specific operation or a node should generate events according to certain conditions. In addition, since it is a one-time communication method, it is very useful communication that can replace the topic because of the small network load.
- In this lecture, it is aimed at creating a simple service file and creating and executing a service server node and a service client node.

Service / Service server / Service client

- ROS uses message communication called '**Service**' when bi-directional(or synchronized) communication is needed. '**service server**' responds only when there is a request & '**service client**' requests and responds.

1) Creating the package

```
$ cd ~/catkin_ws/src  
$ catkin_create_pkg ros_tutorials_service message_generation std_msgs roscpp
```

```
$ cd ros_tutorials_service  
$ ls  
include          → header file folder  
src              → source code folder  
CMakeLists.txt   → build configuration file  
package.xml      → package configuration file
```


Service / Service server / Service client

2) Modify the package configuration file (package.xml)

- One of the required ROS configuration files, package.xml, is an XML file containing package information, which describes the package name, author, license, and dependency package.

```
$ gedit package.xml
```

```
<?xml version="1.0"?>
<package>
  <name>ros_tutorials_service</name>
  <version>0.1.0</version>
  <description>ROS tutorial package to learn the service</description>
  <license>Apache License 2.0</license>
  <author email="pyo@robotis.com">Yoonseok Pyo</author>
  <maintainer email="pyo@robotis.com">Yoonseok Pyo</maintainer>
  <url type="bugtracker">https://github.com/ROBOTIS-GIT/ros_tutorials/issues</url>
  <url type="repository">https://github.com/ROBOTIS-GIT/ros_tutorials.git</url>
  <url type="website">http://www.robotis.com</url>
```

Service / Service server / Service client

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>std_msgs</build_depend>
<build_depend>message_generation</build_depend>
<run_depend>roscpp</run_depend>
<run_depend>std_msgs</run_depend>
<run_depend>message_runtime</run_depend>
<export></export>
</package>
```

Service / Service server / Service client

3) Modify the build configuration file (CMakeLists.txt)

```
$ gedit CMakeLists.txt
```

```
cmake_minimum_required(VERSION 2.8.3)
project(ros_tutorials_service)
```

```
## This is the component package required for catkin build
```

```
## dependency packages are message_generation, std_msgs, and roscpp. If these packages do not exist, an error occurs when you build.
```

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

```
## Service declaration: SrvTutorial.srv
```

```
add_service_files(FILES SrvTutorial.srv)
```

```
## This is an option to configure dependent messages.
```

```
## If std_msgs is not installed, an error occurs when you build.
```

```
generate_messages(DEPENDENCIES std_msgs)
```

```
## The catkin package option describes the library, catkin build dependencies, and system dependent packages.
```

```
catkin_package(
```

```
  LIBRARIES ros_tutorials_service
```

```
  CATKIN_DEPENDS std_msgs roscpp
```

```
)
```

Service / Service server / Service client

Set the include directory.

```
include_directories(${catkin_INCLUDE_DIRS})
```

Build option for the service_server node.

Configure the executable file, target link library, and additional dependencies.

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

```
add_dependencies(service_server ${${PROJECT_NAME}_EXPORTED_TARGETS}  
${catkin_EXPORTED_TARGETS})
```

```
target_link_libraries(service_server ${catkin_LIBRARIES})
```

Build option for the service_client node.

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

```
add_dependencies(service_client ${${PROJECT_NAME}_EXPORTED_TARGETS}  
${catkin_EXPORTED_TARGETS})
```

```
target_link_libraries(service_client ${catkin_LIBRARIES})
```

Service / Service server / Service client

4) Write service file

- Add the following options in the CMakeLists.txt file.

```
add_service_files(FILES SrvTutorial.srv)
```

- This commands means a service should be built based on SrvTutorial.srv

```
$ roscd ros_tutorials_service      → Move to package folder
$ mkdir srv                       → Create a new service folder named srv in the package
$ cd srv                          → Go to the created srv folder
$ gedit SrvTutorial.srv           → Create new SrvTutorial.srv file and modify contents
```

- int64 (message format), a, b (request for service), result (service response: response), '---'(delimiter of request and response)

```
int64 a
int64 b
---
int64 result
```

Service / Service server / Service client

5) Writing service server node

- In the CMakeLists.txt file, add the option to generate the following executable file.

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

- Build a file named service_server.cpp in the src folder to create an executable called service_server

```
$ roscd ros_tutorials_service/src    → Move to the src folder, which is the source folder of the package  
$ gedit service_server.cpp           → New source file and modify contents
```

```
#include "ros/ros.h"                // ROS basic header file  
#include "ros_tutorials_service/SrvTutorial.h" // SrvTutorial service file header (Auto-generated after  
build)
```

Service / Service server / Service client

```
// If there is a service request, do the following
// service request is req, service response is res
bool calculation(ros_tutorials_service::SrvTutorial::Request &req,
                 ros_tutorials_service::SrvTutorial::Response &res)
{
    // Add a and b values received in service request and store them in service response value
    res.result = req.a + req.b;

    // Display the values of a and b used in the service request and the result value corresponding to the service
    response
    ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
    ROS_INFO("sending back response: %ld", (long int)res.result);

    return true;
}
```

Service / Service server / Service client

```
int main(int argc, char **argv)           // Node main function
{
    ros::init(argc, argv, "service_server"); // Initialize the node name
    ros::NodeHandle nh;                     // Declare node handle

    // service server declaration, using SrvTutorial service file from ros_tutorials_service package
    // Declare the service server ros_tutorials_service_server
    // The service name is ros_tutorial_srv and when there is a service request,
    // Set up a function called calculation
    ros::ServiceServer ros_tutorials_service_server =
nh.advertiseService("ros_tutorial_srv", calculation);

    ROS_INFO("ready srv server!");

    ros::spin(); // Wait for service request

    return 0;
}
```


Service / Service server / Service client

6) Write service client node

- In the CMakeLists.txt file, add the option to generate the following executable file.

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

- Build a file called service_client.cpp in the src folder to create an executable called service_client

```
$ roscd ros_tutorials_service/src
```

→ Move to the src folder, which is the source folder for the package

```
$ gedit service_client.cpp
```

→ New source file and modify contents

```
#include "ros/ros.h"
```

// ros basic header file

```
#include "ros_tutorials_service/SrvTutorial.h" // SrvTutorial service file header (auto-generated after build)
```

```
#include <cstdlib>
```

// library for using atoll function

Service / Service server / Service client

```
int main(int argc, char **argv)           // Node main function
{
    ros::init(argc, argv, "service_client"); // Initialize the node name

    if (argc != 3)                         // process input value error
    {
        ROS_INFO("cmd : rosrn ros_tutorials_service service_client arg0 arg1");
        ROS_INFO("arg0: double number, arg1: double number");
        return 1;
    }

    ros::NodeHandle nh;                   // Declare a node handle to communicate with the ROS system

    // service client declaration, using SrvTutorial service file in ros_tutorials_service package
    // Declare service client ros_tutorials_service_client
    // The service name is "ros_tutorial_srv"
    ros::ServiceClient ros_tutorials_service_client =
    nh.serviceClient<ros_tutorials_service::SrvTutorial>("ros_tutorial_srv");
```

Service / Service server / Service client

```
// Declare a service that uses the SrvTutorial service file with the name srv
ros_tutorials_service::SrvTutorial srv;

// stores the parameters used as input when the node is executed as a service request value in each of a and b
srv.request.a = atoll(argv[1]);
srv.request.b = atoll(argv[2]);

// Request the service, and if the request is accepted, display the response value
if (ros_tutorials_service_client.call(srv))
{
    ROS_INFO("send srv, srv.Request.a and b: %ld, %ld", (long int)srv.request.a, (long int)srv.request.b);
    ROS_INFO("receive srv, srv.Response.result: %ld", (long int)srv.response.result);
}
else
{
    ROS_ERROR("Failed to call service ros_tutorial_srv");
    return 1;
}
return 0;
}
```

Service / Service server / Service client

7) Build the ROS node

- Use the following command to build the service file, service server node, and client node of the `ros_tutorials_service` package

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

→ move to catkin folder and run catkin build

- [Reference] File system
 - **Source code file** of `ros_tutorials_service` package: `~/catkin_ws/src/ros_tutorials_service/src`
 - **Message file** of `ros_tutorials_service` package: `~/catkin_ws/src/ros_tutorials_service/msg`
 - Built files are located in the `/build` and `/devel` folders in `/catkin_ws`
 - The `/build` folder stores the settings used in the catkin build
 - The `/devel/lib/ros_tutorials_service` folder contains executable files
 - The `/devel/include/ros_tutorials_service` folder contains **message header files** that are automatically generated from the message files

Service / Service server / Service client

8) Run the service server [Note: Do not forget to run roscore before running the node.]

- The service server has programmed to wait until there is a service request. Therefore, when the following command is executed, the service server waits for a service request.

```
$ roslaunch ros_tutorials_service service_server  
[INFO] [1495726541.268629564]: ready srv server!
```

Service / Service server / Service client

9) Run the service client

- After running the service server, run the service client with the following command:

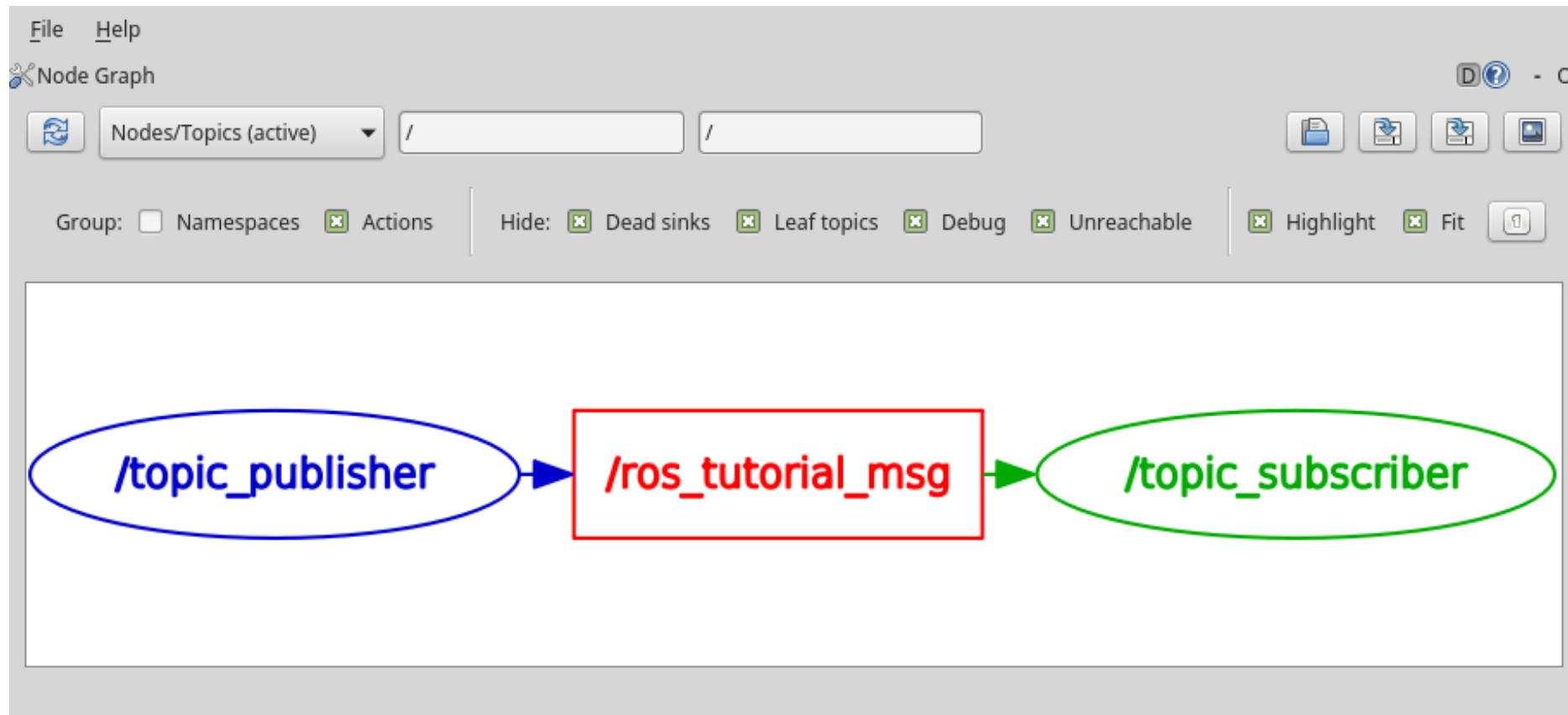
```
$ roslaunch ros_tutorials_service service_client 2 3  
[INFO] [1495726543.277216401]: send srv, srv.Request.a and b: 2, 3  
[INFO] [1495726543.277258018]: receive srv, srv.Response.result: 5
```

- The client is programmed to send parameter 2 and 3 as the service request value.
- 2 and 3 correspond to a and b values in the service, respectively, and receive a sum of them(5) as a response value.
- In this case, numbers are used as execution parameters, but in actual use, an instruction, a value to be calculated, a variable for a trigger may be used as a service request value.

Service / Service server / Service client

[Reference] `rqt_graph`

- Service is one-time unlike the topic in the figure below, so it can not be visualized in `rqt_graph`.



Service / Service server / Service client

[Reference] How to use rosservice call command

- The service request can be executed by the service client node, but there is also a method called "rosservice call" or using rqt's ServiceCaller.
- Let's look at how to use rosservice call.

```
$ rosservice call /ros_tutorial_srv 10 2  
result: 12
```

```
$ rosservice call /ros_tutorial_srv 5 15  
result: 20
```


Service / Service server / Service client

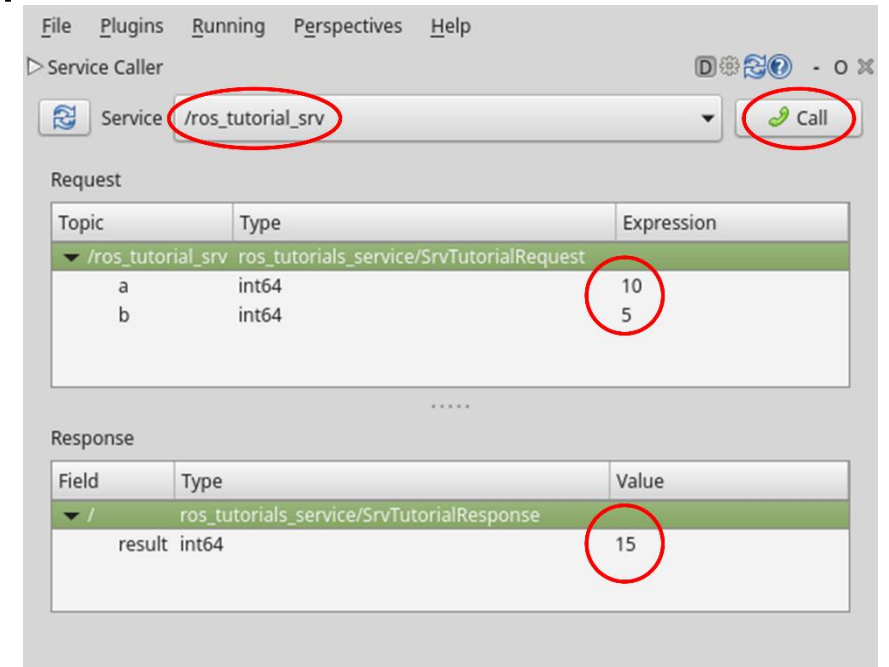
[Reference] How to use GUI tool, Service Caller

- Run the ROS GUI tool rqt.

```
$ rqt
```

- Select [Plugins] → [Service] → [Service Caller] from the rqt program's menu. The following screen appears.

- (1) Enter / ros_tutorial_srv in servic
- (2) Input a = 10, b = 5
- (3) Press the Call button.
- (4) A value of 15 is displayed in Result.



One more!!!

- A single node can act as multiple publishers, subscribers, service servers, and service clients!
- Use the node as you wish

```
ros::NodeHandle nh;  
  
ros::Publisher topic_publisher = nh.advertise<ros_tutorials::MsgTutorial>("ros_tutorial_msg", 100);  
ros::Subscriber topic_subscriber = nh.subscribe("ros_tutorial_msg", 100, msgCallback);  
ros::ServiceServer service_server = nh.advertiseService("ros_tutorial_srv", calculation);  
ros::ServiceClient service_client = nh.serviceClient<ros_tutorials::SrvTutorial>("ros_tutorial_srv");
```

Source code

- We have seen how to communicate services between nodes by creating service servers and client nodes and running them. The relevant source can be found at the address of GitHub below.
- https://github.com/ROBOTIS-GIT/ros_tutorials/tree/master/ros_tutorials_service
- If you want to try it right away, you can clone the source code with the following command in the catkin_ws/src folder and build. Then run the service_server and service_client nodes.

```
$ cd ~/catkin_ws/src  
$ git clone https://github.com/ROBOTIS-GIT/ros_tutorials.git  
$ cd ~/catkin_ws  
$ catkin_make
```

```
$ rosrun ros_tutorials_service service_server
```

```
$ rosrun ros_tutorials_service service_client
```

Parameters

Parameters

1) Create nodes using parameters

- In this section, we are going to modify the service_server.cpp to use a parameter to select an arithmetic operator, rather than merely adding a and b entered as service requests.
- Let's modify the service_server.cpp source in the following order:

```
$ roscd ros_tutorials_service/src
```

→ Go to the src folder, which is the source code folder of the package

```
$ gedit service_server.cpp
```

→ Modify source file contents

```
#include "ros/ros.h"                // ROS basic header file
#include "ros_tutorials_service/SrvTutorial.h" // SrvTutorial.h" // SrvTutorial Service file header (auto-generated after build)

#define PLUS          1 // Add
#define MINUS         2 // Subtract
#define MULTIPLICATION 3 // Multiply
#define DIVISION      4 // Divide

int g_operator = PLUS;
```

Parameters

```
// If there is a service request, do the following
// service request is req, service response is res
```

```
bool calculation(ros_tutorials_service::SrvTutorial::Request &req,
                 ros_tutorials_service::SrvTutorial::Response &res)
{
    // The values of a and b received when the service is requested are changed according to the parameter value.
    // Calculate and store in service response value switch(g_operator)
    {
        case PLUS:
            res.result = req.a + req.b; break;
        case MINUS:
            res.result = req.a - req.b; break;
        case MULTIPLICATION:
            res.result = req.a * req.b; break;
```

Parameters

```
case DIVISION:
    if(req.b == 0){
        res.result = 0; break;
    }
    else{
        res.result = req.a / req.b; break;
    }
default:
    res.result = req.a + req.b; break;
}
// Display the values of a and b used in the service request and the result value corresponding to the service response
ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
ROS_INFO("sending back response: [%ld]", (long int)res.result);

return true;
}
```

Parameters

```
int main(int argc, char **argv)           // Node main function
{
    ros::init(argc, argv, "service_server"); // Initialize the node name

    ros::NodeHandle nh;                    // Declare a node handle to communicate with the ROS
    system                                 system

    nh.setParam("calculation_method", PLUS); // Parameter Initialization

    // service server declaration, using SrvTutorial service file from ros_tutorials_service package
    // Create the service server service_server. The service name is "ros_tutorial_srv"
    // Set up to execute a function called calculation when there is a service request.
    ros::ServiceServer ros_tutorial_service_server = nh.advertiseService("ros_tutorial_srv", calculation);

    ROS_INFO("ready srv server!");
```


Parameters

```
ros::Rate r(10);                                // 10 hz

while (1)
{
    nh.getParam("calculation_method", g_operator); // Change the operator to the value received from the parameter
    ros::spinOnce();                               // Callback function processing routine
    r.sleep();                                      // sleep processing for routine iteration
}

return 0;
}
```

- Note the usage of setParam and getParam regarding parameters.

[Reference] Available as parameter Type

- Parameters can be set as integers, floats, boolean, string, dictionaries, list, and so on.
- For example, 1 is an integer, 1.0 is a float, "Internet of Things" is a string, true is a boolean, [1,2,3] is a list of integers, a: b, c: d is a dictionary.

Parameters

2) Build and run the node

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

```
$ rosrun ros_tutorials_service service_server
[INFO] [1495767130.149512649]: ready srv server!
```

3) View parameter list

- The "rosparam list" command shows a list of parameters currently used in the ROS network, where /calculation_method is the parameter we used.

```
$ rosparam list
/calculation_method
/rosdistro
/rosversion
/run_id
```

Parameters

4) Example of parameter usage

- Set the parameters according to the following command, and observe the change of service process when requesting the same service each time.
- In ROS, parameters can change the flow, setting, and processing of nodes from outside the node. It's a very useful feature so be familiar with it.

```
$ rosservice call /ros_tutorial_srv 10 5      → Input variables a and b of arithmetic operation result: 15
```

→ The addition arithmetic sum, which is the default arithmetic operation

```
$ rosparam set /calculation_method 2          → subtract
```

```
$ rosservice call /ros_tutorial_srv 10 5
```

result: 5

```
$ rosparam set /calculation_method 3          → multiply
```

```
$ rosservice call /ros_tutorial_srv 10 5
```

result: 50

```
$ rosparam set /calculation_method 4          → divide
```

```
$ rosservice call /ros_tutorial_srv 10 5
```

result: 2

Source code

- We have already discussed how to modify the existing service server and use parameters.
- The related source has been renamed as `ros_tutorials_parameter` package in order to distinguish it from the service source code that was created previously, and can be found at the github address below.
- https://github.com/ROBOTIS-GIT/ros_tutorials/tree/master/ros_tutorials_parameter
- If you want to try it right away, you can clone the source code with the following command in the `catkin_ws/src` folder and build. Then run the `service_server` and `service_client` nodes.

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/ROBOTIS-GIT/ros_tutorials.git
$ cd ~/catkin_ws
$ catkin_make
```

```
$ rosrn ros_tutorials_parameter service_server_with_parameter
```

```
$ rosrn ros_tutorials_parameter service_client_with_parameter 2 3
```

roslaunch

How to use roslaunch

- **roslaunch** is a command to execute a node.
- **roslaunch** can run one or more defined nodes.
- In addition, roslaunch command allows you to **specify options** such as changing package parameters or node names, configuring node namespaces, setting ROS_ROOT and ROS_PACKAGE_PATH, and changing environment variables when running a node.
- roslaunch uses the file '* .launch' to set up an executable node, which **is XML-based and provides tag-specific options**.
- The execution command is "roslaunch [package name] [roslaunch file]".

How to use roslaunch

1) Use of roslaunch

- Rename the previously created topic_publisher and topic_subscriber nodes and run them. In order to understand the roslaunch, let's setup two separate message communication by running two public nodes and two subscriber nodes.
- First, let's write a *.launch file. The file used for roslaunch has a file name of *.launch, and you need to create a folder called launch in the package folder and put the launch file in the folder. The folder can be created with the following command and create a new file called union.launch.

```
$ roscd ros_tutorials_topic  
$ mkdir launch  
$ cd launch  
$ gedit union.launch
```

How to use roslaunch

```
<launch>
  <node pkg="ros_tutorials_topic" type="topic_publisher" name="topic_publisher1"/>
  <node pkg="ros_tutorials_topic" type="topic_subscriber" name="topic_subscriber1"/>
  <node pkg="ros_tutorials_topic" type="topic_publisher" name="topic_publisher2"/>
  <node pkg="ros_tutorials_topic" type="topic_subscriber" name="topic_subscriber2"/>
</launch>
```

- The **<launch>** tag describes required tags to run the node with roslaunch command. The **<node>** tags describe the node to be run by roslaunch. Options include pkg, type, and name.
 - **pkg** Package name
 - **type** The name of the node to actually execute (node name)
 - **name** Set the name (executable name) to be appended when the node corresponding to the above type is executed, usually set to be the same as type, but you can set it to change the name when you need it.

How to use roslaunch

- Once you have created the roslaunch file, run union.launch as follows.

```
$ roslaunch ros_tutorials_topic union.launch --screen
```

- [Reference] How to display status when using roslaunch
 - When the roslaunch command runs several nodes, the output (info, error, etc.) of the running nodes is not displayed on the terminal screen and it makes debugging difficult. If you add the --screen option, the output of all nodes running on that terminal will be displayed on the terminal screen.

How to use roslaunch

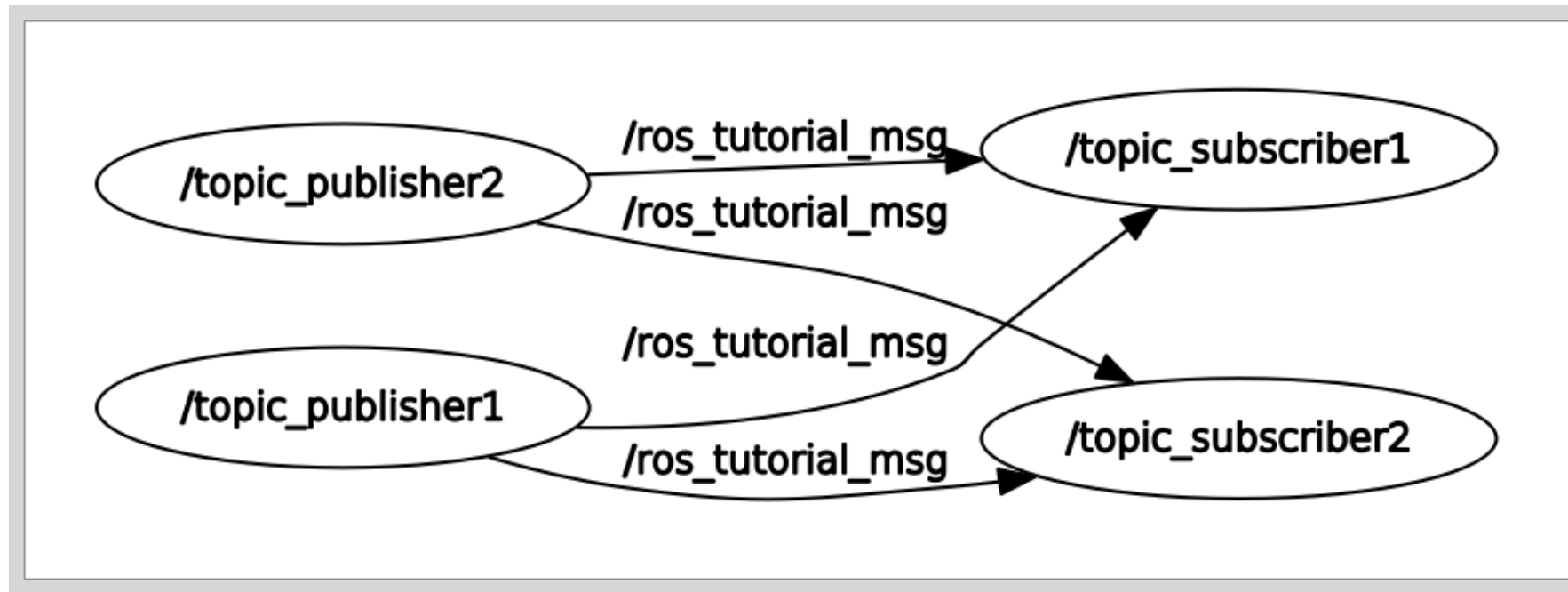
- Execution result?

```
$ rosnode list  
/topic_publisher1  
/topic_publisher2  
/topic_subscriber1  
/topic_subscriber2  
/rosout
```

- As a result, the topic_publisher node was renamed as topic_publisher1 and topic_publisher2, and two nodes are running.
- The topic_subscriber node has also been renamed as topic_subscriber1 and topic_subscriber2.

How to use roslaunch

- The problem is that, unlike the initial intention that "two separate messages are communicated by driving two publisher nodes and subscriber nodes," rqt_graph shows that they are subscribing to each other's messages.
- This is because we simply changed the name of the node to be executed, but did not change the name of the message to be used.
- Let's fix this problem with another roslaunch namespace tag.



How to use roslaunch

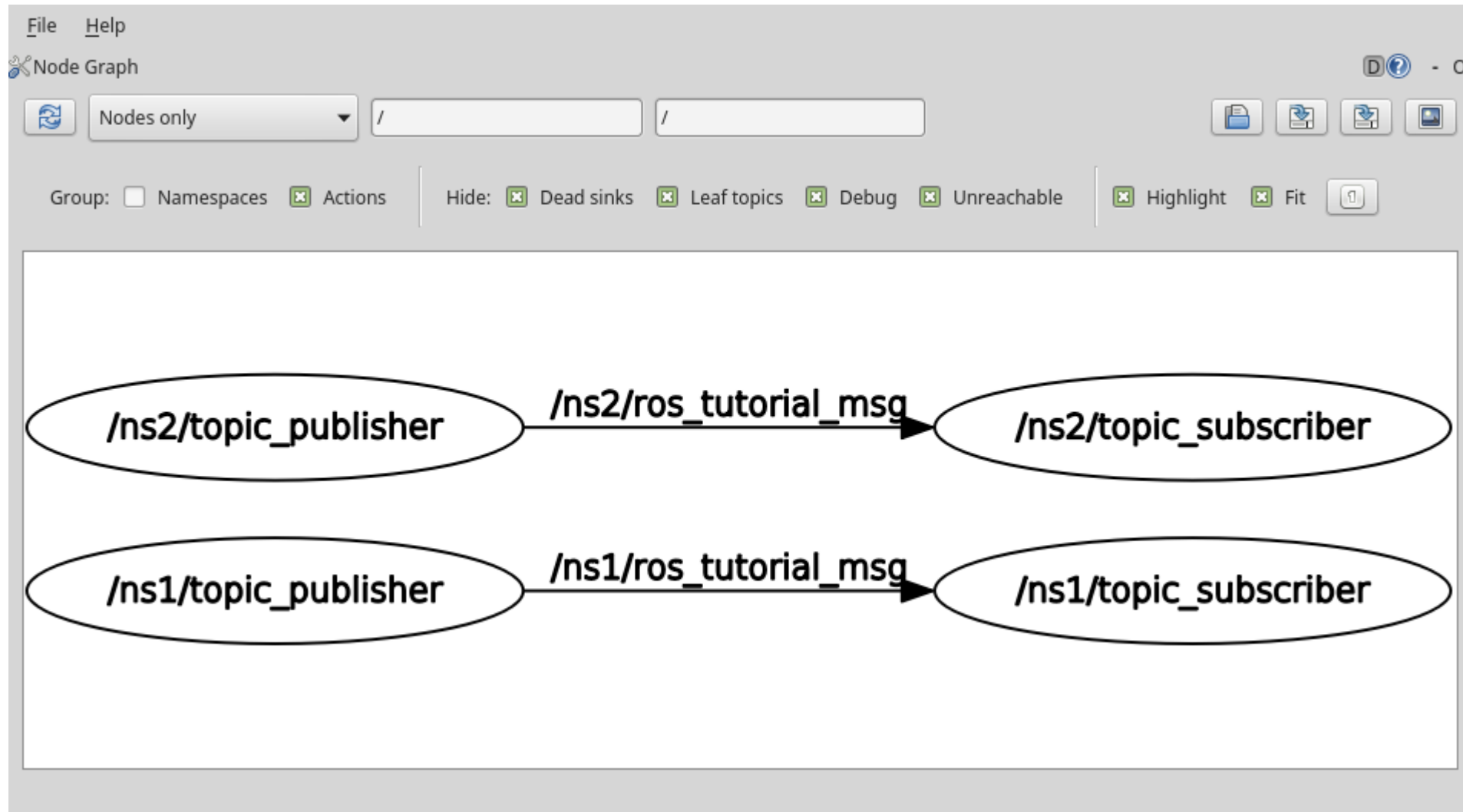
- Let's modify union.launch

```
$ roscd ros_tutorials_service/launch  
$ gedit union.launch
```

```
<launch>  
  <group ns="ns1">  
    <node pkg="ros_tutorials_topic" type="topic_publisher" name="topic_publisher"/>  
    <node pkg="ros_tutorials_topic" type="topic_subscriber" name="topic_subscriber"/>  
  </group>  
  <group ns="ns2">  
    <node pkg="ros_tutorials_topic" type="topic_publisher" name="topic_publisher"/>  
    <node pkg="ros_tutorials_topic" type="topic_subscriber" name="topic_subscriber"/>  
  </group>  
</launch>
```

How to use roslaunch

- The appearance of executed nodes after change



How to use roslaunch

2) launch tag

<launch>	Points to the beginning and end of the roslaunch syntax.
<node>	This is a tag for node execution. You can change the package, node name, and execution name.
<machine>	You can set the name, address, ros-root, and ros-package-path of the PC running the node.
<include>	You can import another package or another launch belonging to the same package and run it as a launch file.
<remap>	You can change the name of the ROS variable in use by the node name, topic name, and so on.
<env>	Set environment variables such as path and IP. (Almost never used)
<param>	Set the parameter name, type, value, etc.
<rosparam>	Check and modify parameter information such as load, dump, and delete as the rosparam command..
<group>	Set the parameter name, type, value, etc.
<test>	Used to test the node Similar to <node>, but with options available for testing.
<arg>	You can define a variable in the launch file, so you can change the parameter when you run it like this.

```
<launch>
  <arg name="update_period" default="10" />
  <param name="timing" value="$(arg update_period)"/>
</launch>
```

```
roslaunch my_package my_package.launch update_period:=30
```

Repository of this course

- Example Repository
 - https://github.com/ROBOTIS-GIT/ros_tutorials
- Although it is possible to read or download directly from the web, if you want to try it right away, you can copy and build the repository with the following command:

```
$ cd ~/catkin_ws/src
$ git https://github.com/ROBOTIS-GIT/ros_tutorials.git
$ cd ~/catkin_ws
$ catkin_make
```

You are not a beginner of ROS
anymore!

Question Time!

Advertisement #1



Free

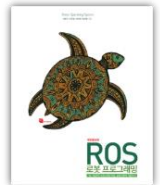


Download link



Language:

English, chinese, Japanese, Korean



“ROS Robot Programming”

A Handbook is written by TurtleBot3 Developers

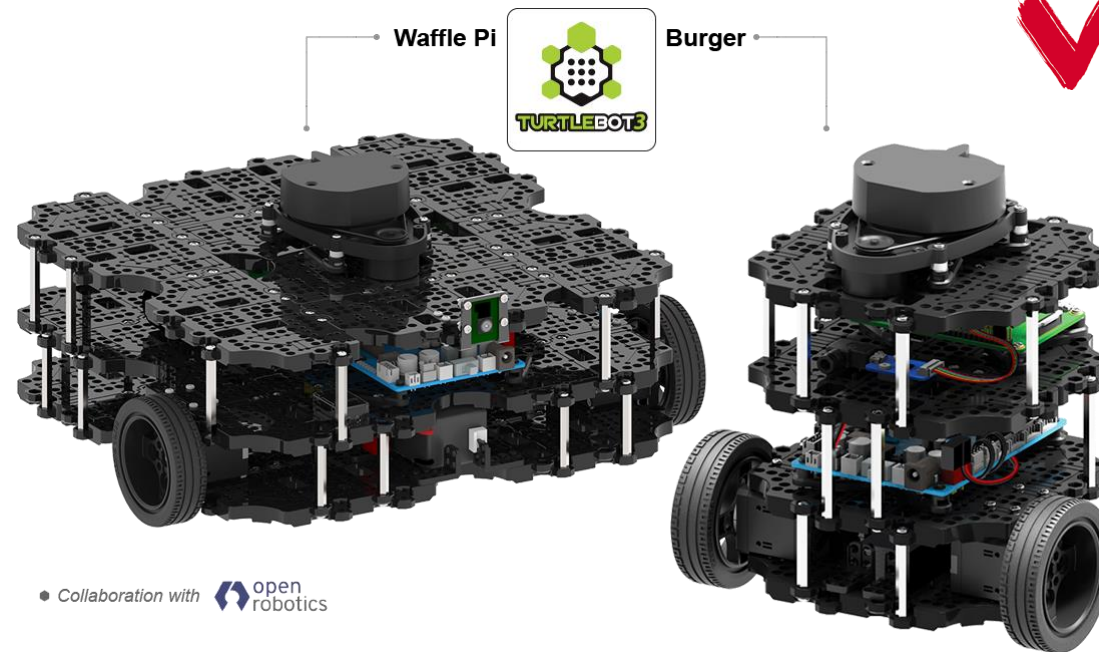
Advertisement #2

TURTLEBOT3

AI Research Starts Here
ROS Official Platform

TurtleBot3 is a new generation mobile robot that's modular, compact and customizable. Let's explore ROS and create exciting applications for education, research and product development.

✓ [Direct Link](#)



Advertisement #3



www.robotsource.org

The 'RobotSource' community is the space for people making robots.

We hope to be a community where we can share knowledge about robots, share robot development information and experiences, help each other and collaborate together. Through this community, we want to realize open robotics without distinguishing between students, universities, research institutes and companies.

Join us in the Robot community ~

END.