Introduction ROS - 2

Jaeseok Kim
The BioRobotics Institute - Service Robotics and Ambient Assisted Living Lab

Contents

- **Things to know before programming ROS**
- **Create and run publishers and subscriber nodes**
- **Create and run service servers and client nodes**

- **Things to know before programming ROS**
- **Create and run publishers and subscriber nodes**
- **Create and run service servers and client nodes**

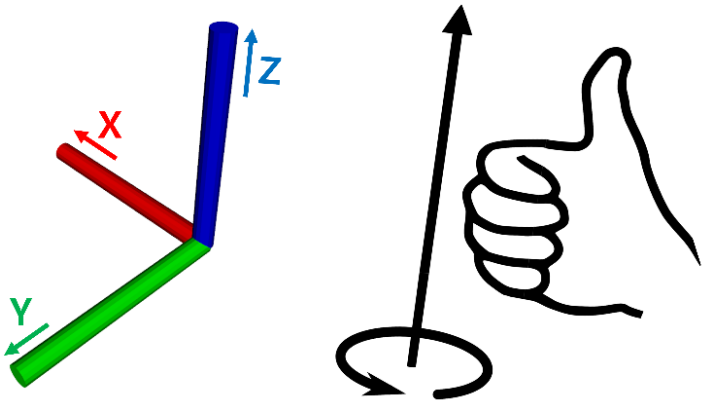
Things to know before programming ROS

- **Standard unit**
  - SI unit

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

Quantity	Unit
angle	radian
frequency	hertz
force	newton
power	watt
voltage	volt

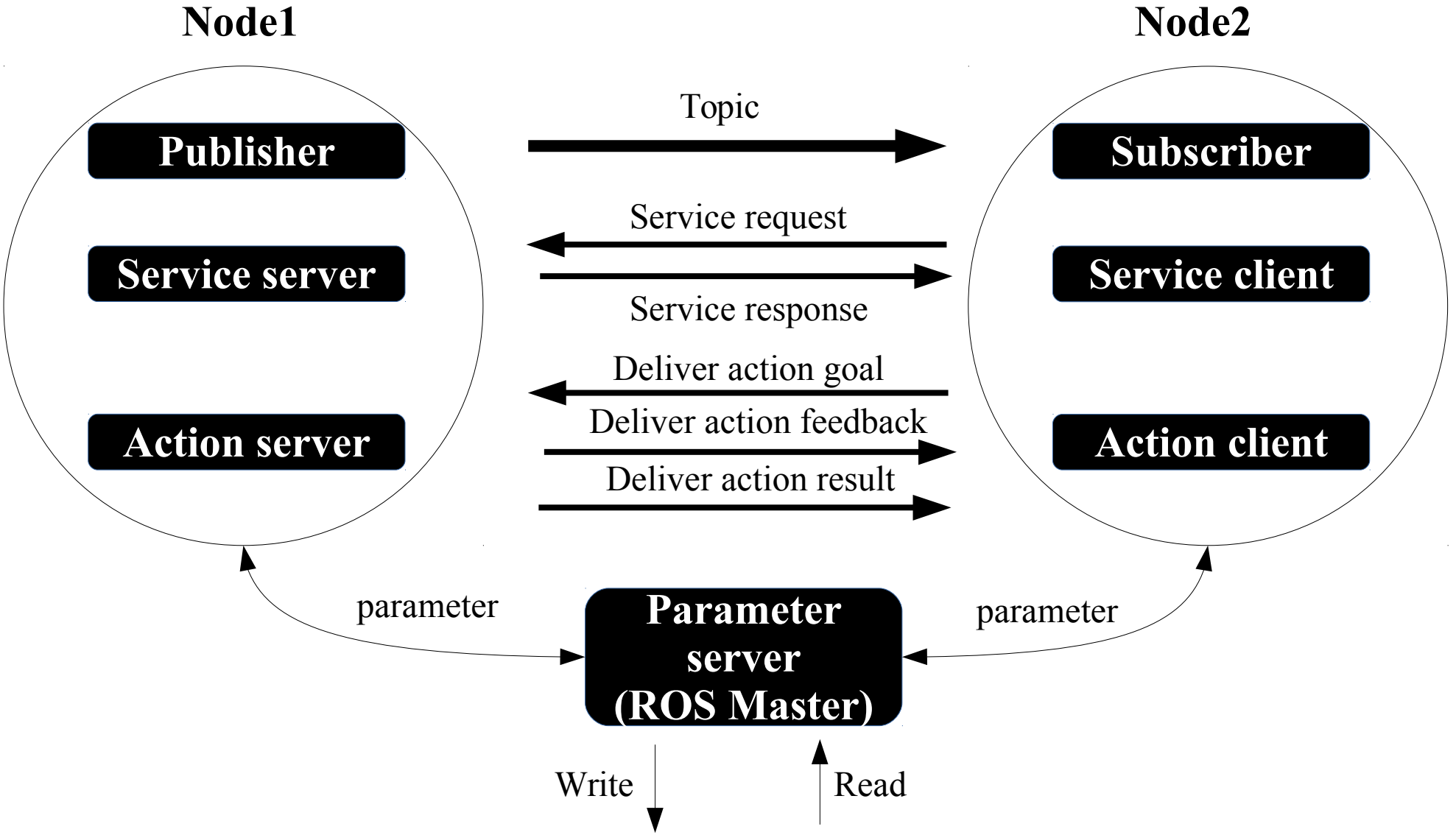
Quantity	Unit
length	meter
mass	kilogram
time	second
current	ampere
temperature	celsius



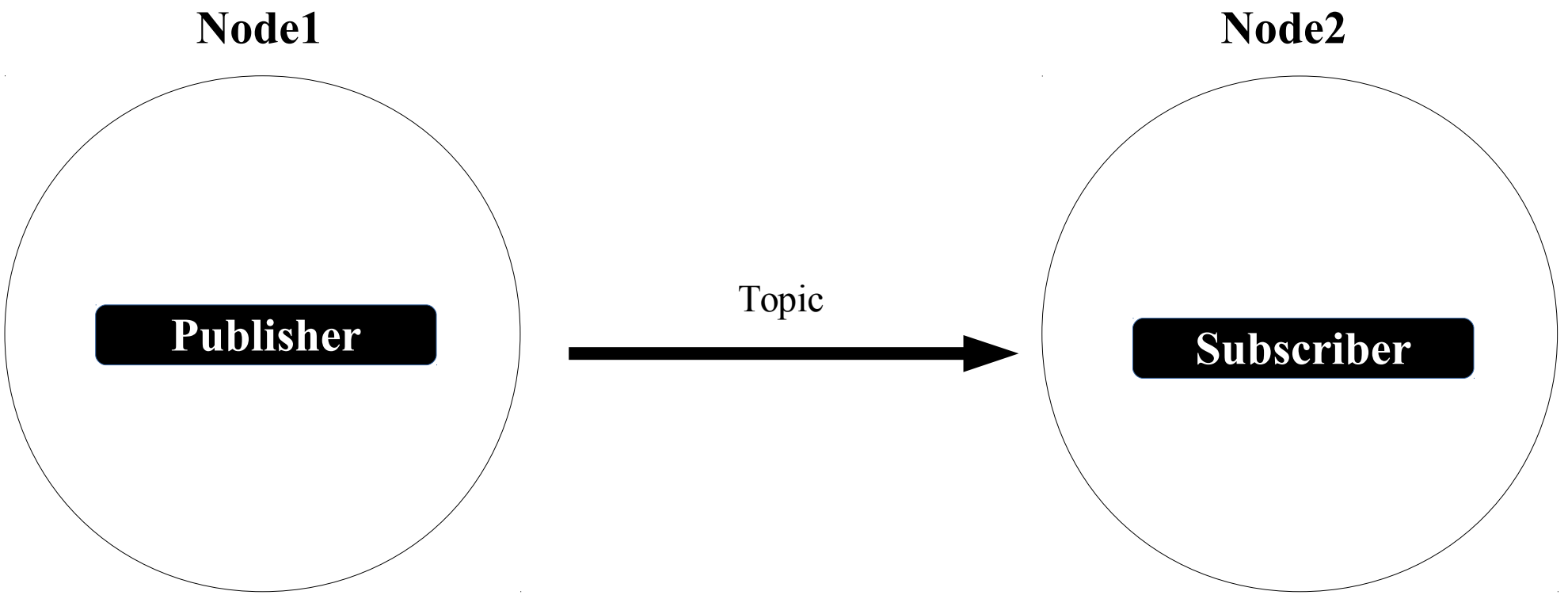
- Things to know before programming ROS
- **Create and run publishers and subscriber nodes**
- Create and run service servers and client nodes

ROS Message

Message communication
(Topic, Service, Action, Parameter)



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 turtorial package to learn the topic</description>
<license>Apache License 2.0</license>
<author email="j.kim@santannapisa.it">Jaeseok Kim</author>
<maintainer email="j.kim@santannapisa.it">Jaeseok Kim</maintainer>
<url type="bugtracker">https://github.com/Alchemist77/ros_tutorials/issues</url>
<url type="repository">https://github.com/Alchemist77/ros_tutorials.git</url>
<url type="website">http://www.santannapisa.it</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 substamp
        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);   // Disaplay stamp.sec message
        ROS_INFO("recieve msg = %d", msg→stamp.nsec); // Disaplay stamp.nsec message
        ROS_INFO("recieve msg = %d", msg→data);           // Disaplay 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();

    rereturn 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
```

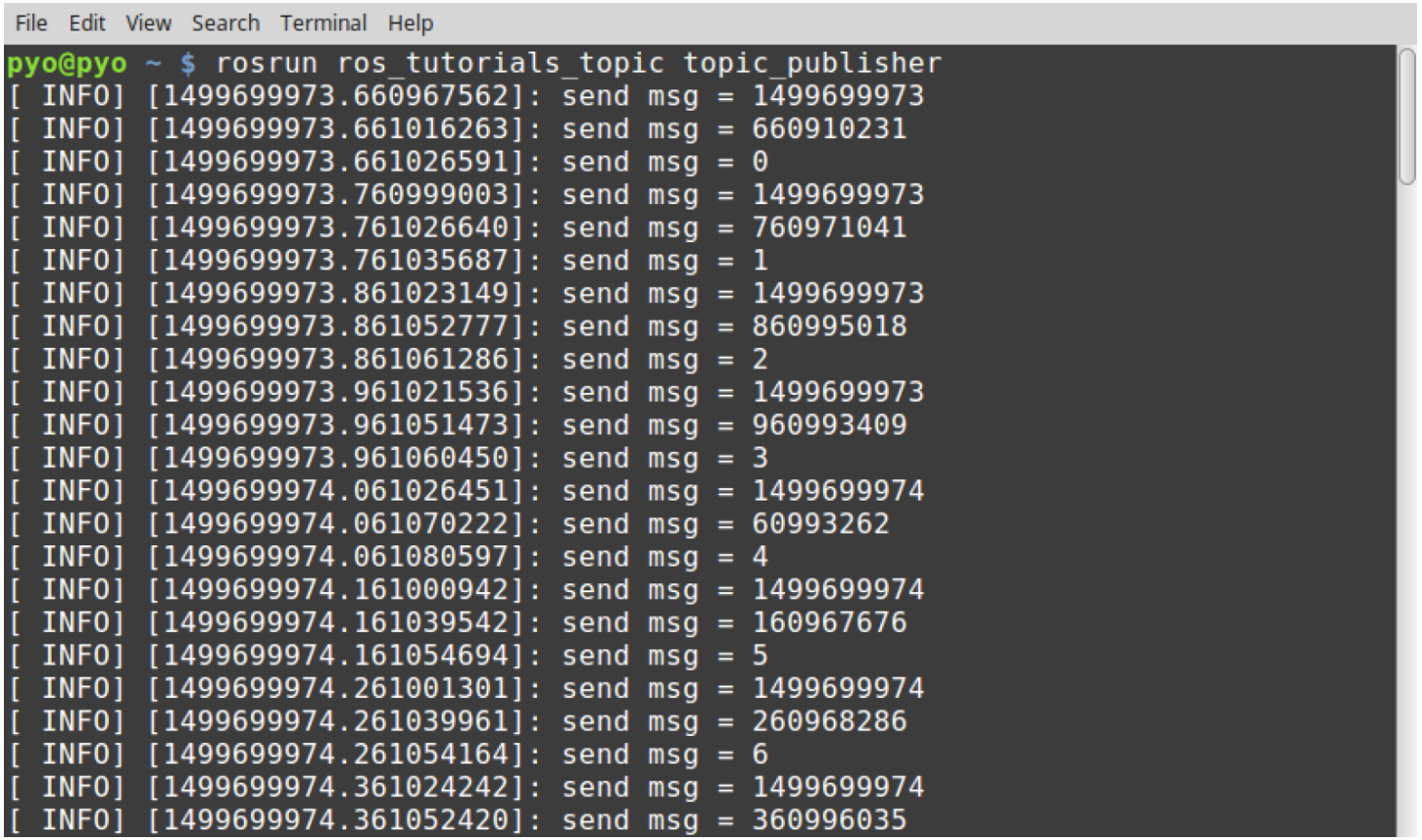
• [Reference] File system
  • Source code file for ros_tutorials_topic package:
  ~/catkin_ws/src/ros_tutorials_topic/src
  • **Message file** for ros_tutorials_topic package:
  ~/catkin_ws/src/ros_tutorials_topic/msg
  • Built files are located in the /build and /devel folders in /catkin_ws
    • /build 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

```
$ rosrun ros_tutorials_topic topic_publisher
```

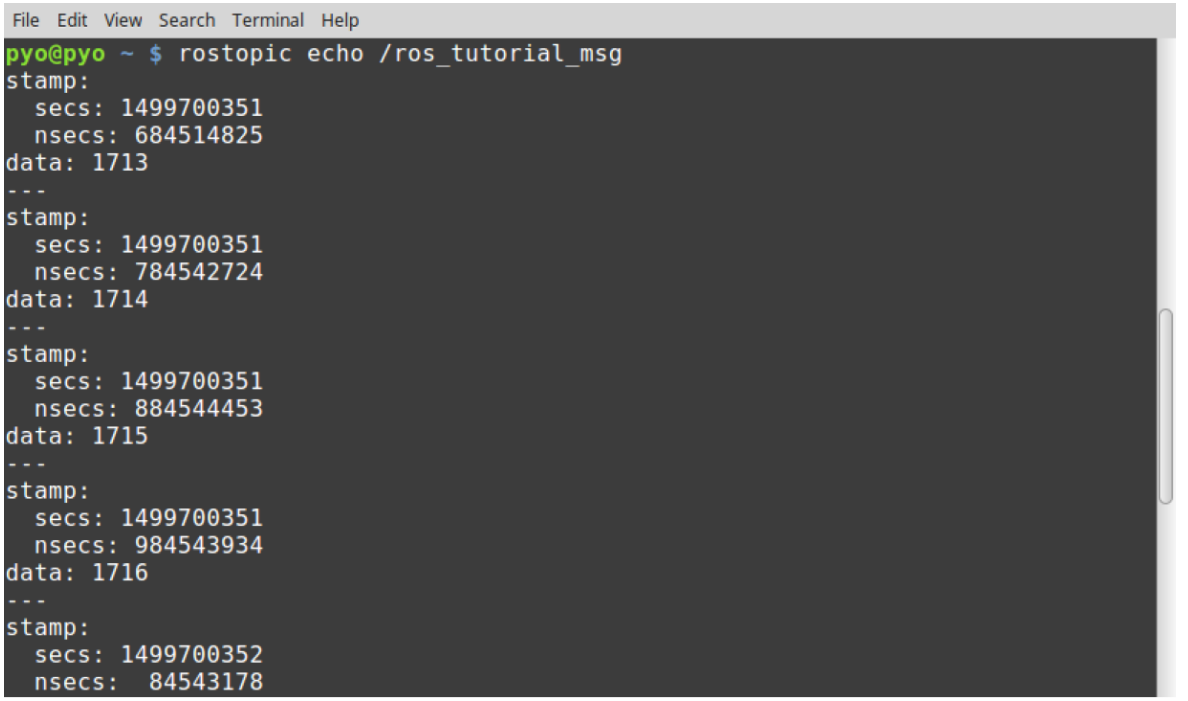


Topic / Publisher / Subscriber

- [Reference] rostopic
  - You can use rostopiccommand 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
```

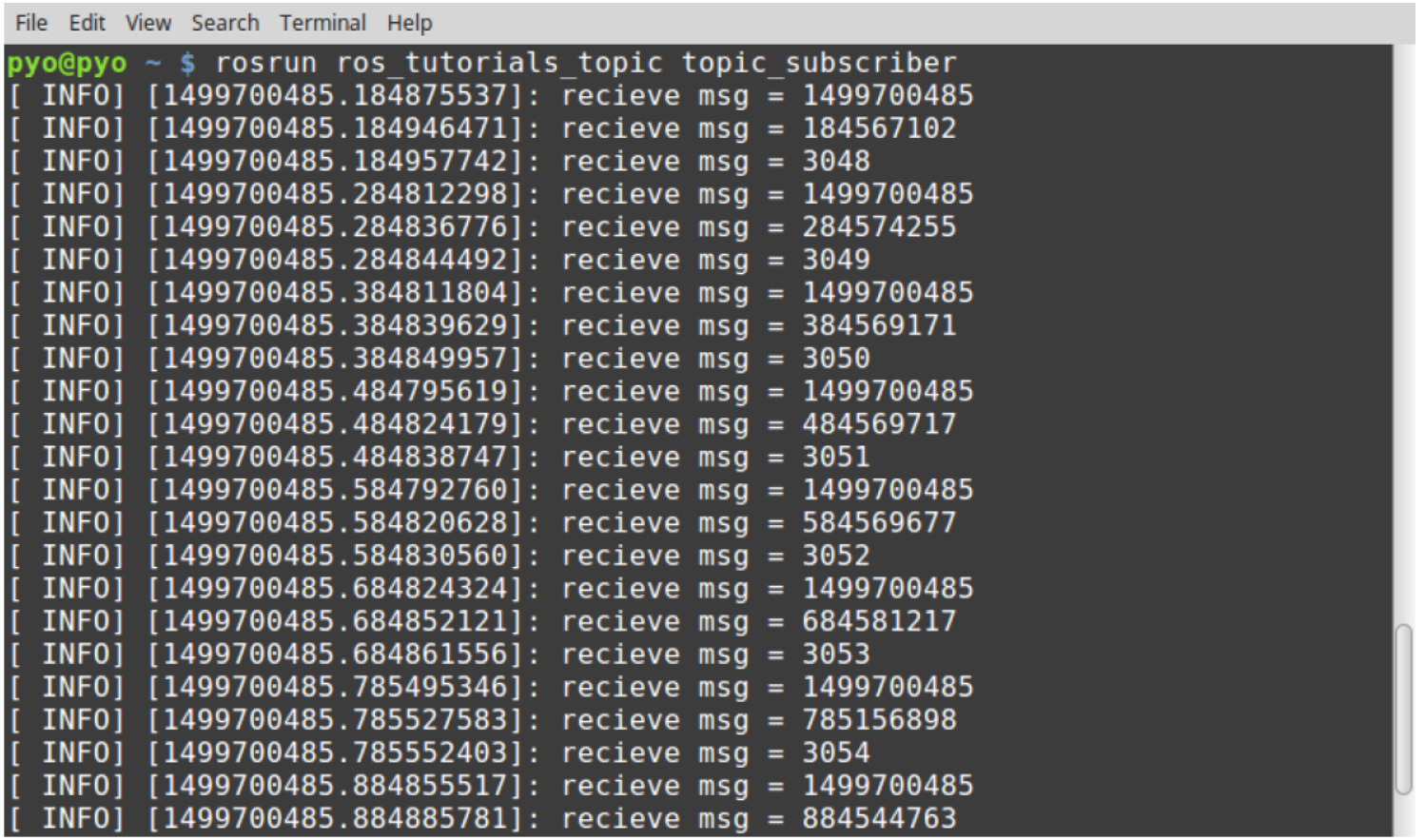


Topic / Publisher / Subscriber

9) Execute Subscriber

• Below command runs the topic_subscriber node of the ros_tutorials_topic package

```
$ rosrun ros_tutorials_topic topic_subscriber
```

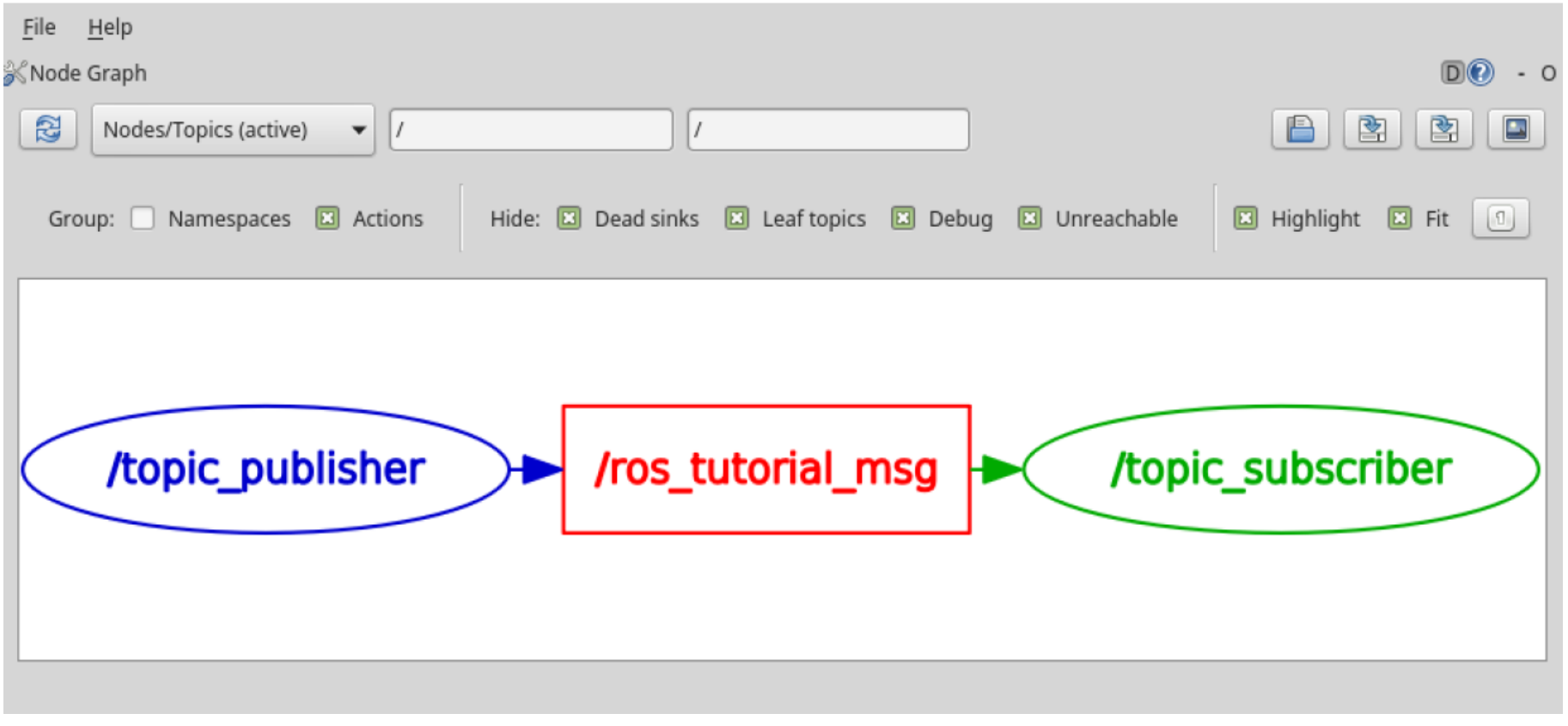


Topic / Publisher / Subscriber

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.

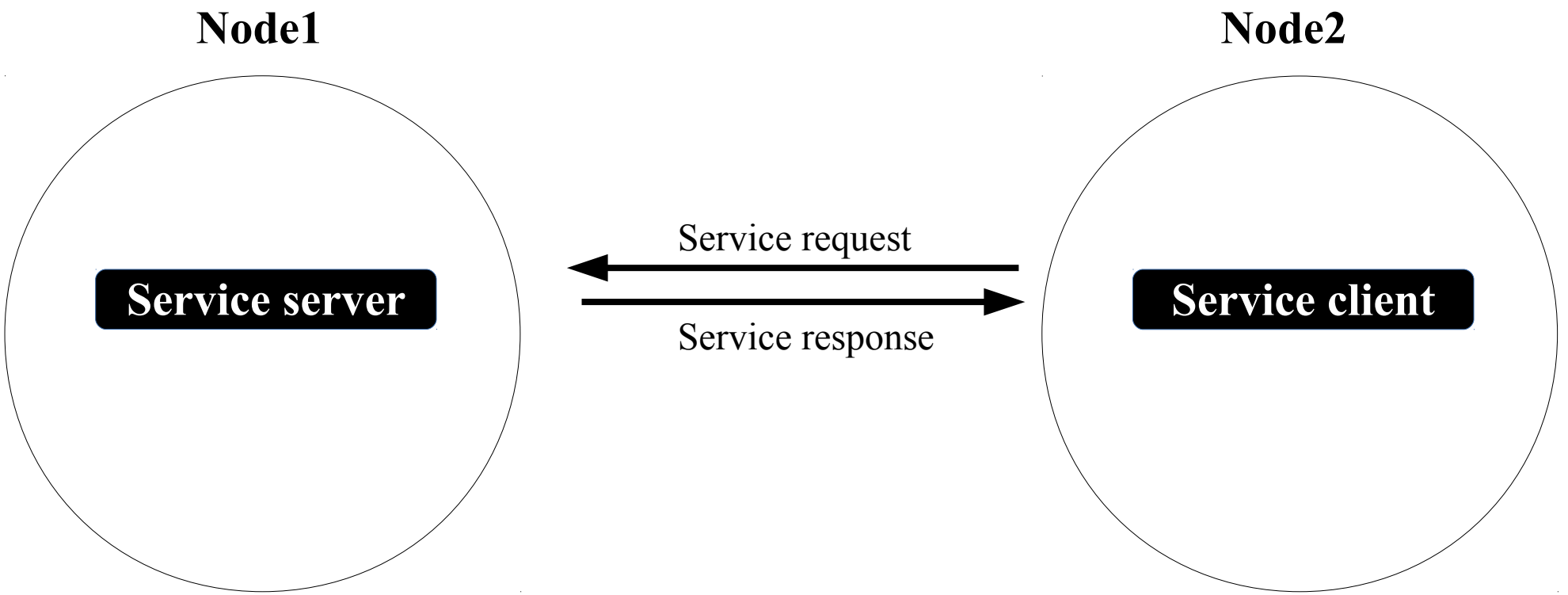
```
$ cd ~/catkin_ws/src
$ git clone https://github.com/ROBOTIS-GIT/ros_tutorials.git
$ cd ~/catkin_ws
$ catkin_make
```

```
$ rosrun ros_tutorials_topic topic_publisher
```

```
$ rosrun ros_tutorials_topic topic_subscriber
```

- Things to know before programming ROS
- Create and run publishers and subscriber nodes
- **Create and run service servers and client nodes**

Service


Node1
Node2
Service server
Service request
Service response
Service client

Service / Service server / Service client

• ROS uses message communication called '**Service**' when bidirectional(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_topic</name>
<version>0.1.0</version>
<description>ROS turtorial package to learn the service</description>
<license>Apache License 2.0</license>
<author email="j.kim@santannapisa.it">Jaeseok Kim</author>
<maintainer email="j.kim@santannapisa.it">Jaeseok Kim</maintainer>
<url type="bugtracker">https://github.com/Alchemist77/ros_tutorials/issues</url>
<url type="repository">https://github.com/Alchemist77/ros_tutorials.git</url>
<url type="website">http://www.santannapisa.it</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 : rosrun 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.

```
$ rosrun 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:

```
$ rosrun 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.

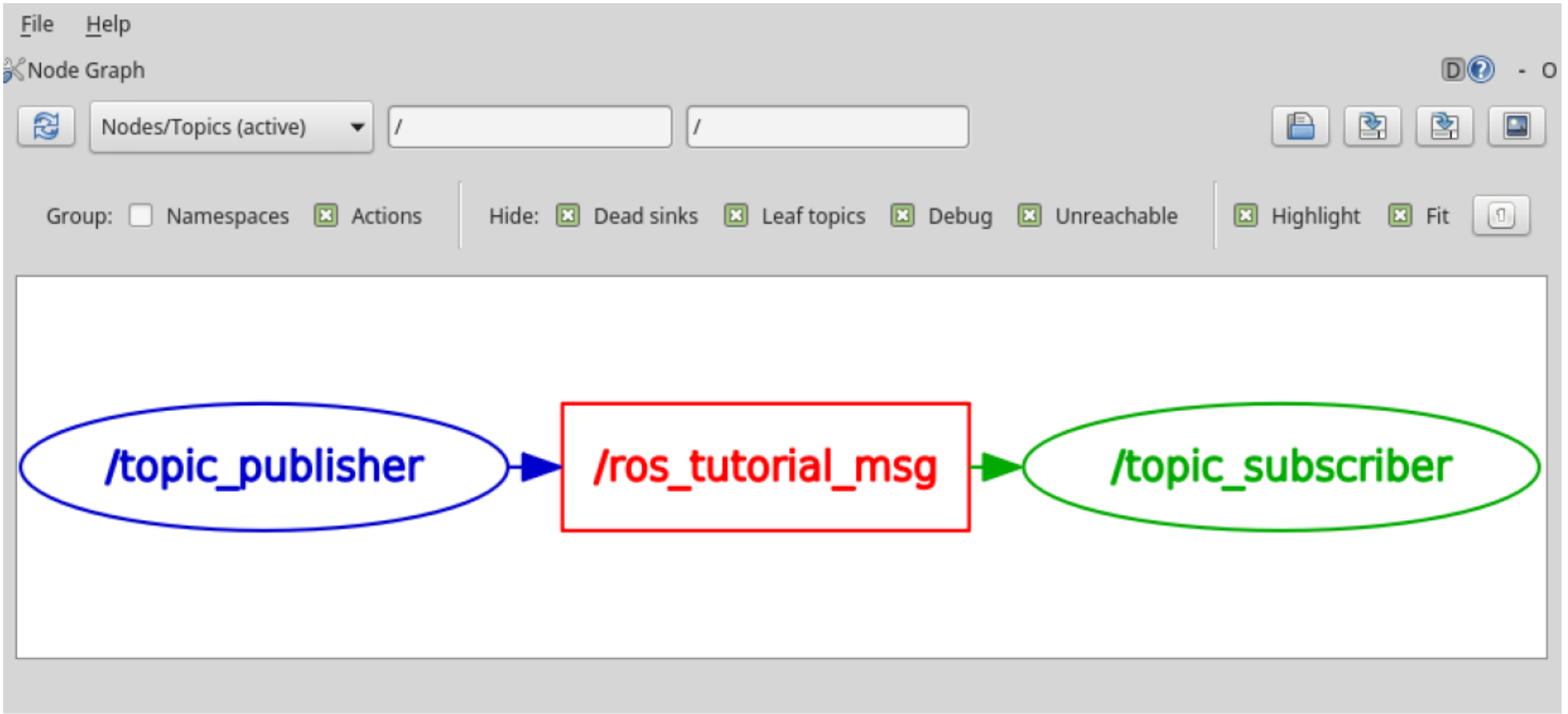
• 2 and 3 correspond to a and b values in the service, respectively, and receive a sum of them(5) as a response value.

• 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
```

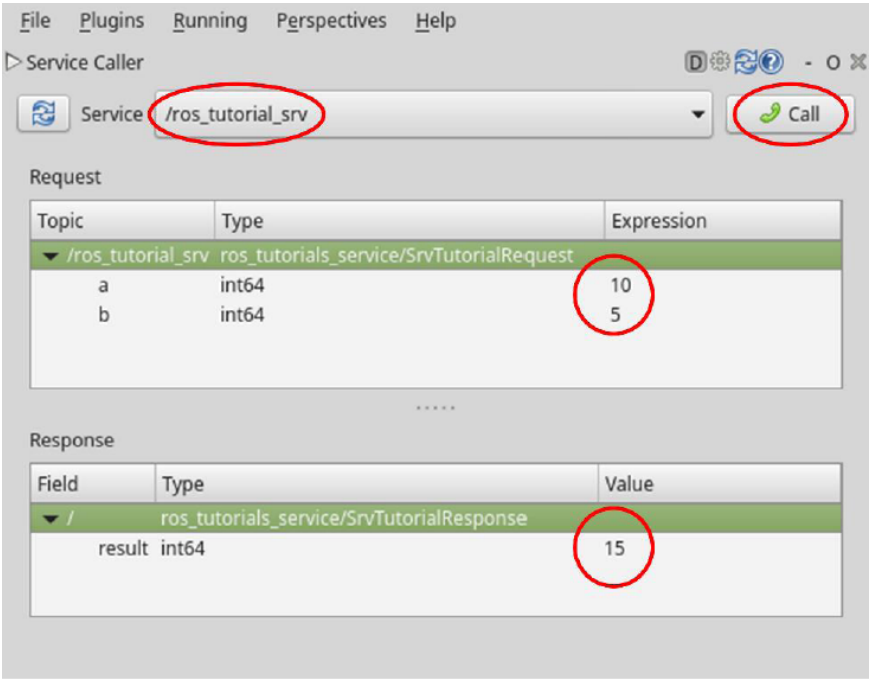
• Select [Plugins] → [Service] → [Service Caller] from the rqt program's menu. The following screen appears.

(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
```

Reference

- **Book**

  - ROS_Robot_Programming_EN
- http://wiki.ros.org/Documentation
- http://wiki.ros.org/ROS/Tutorials

Question Time!