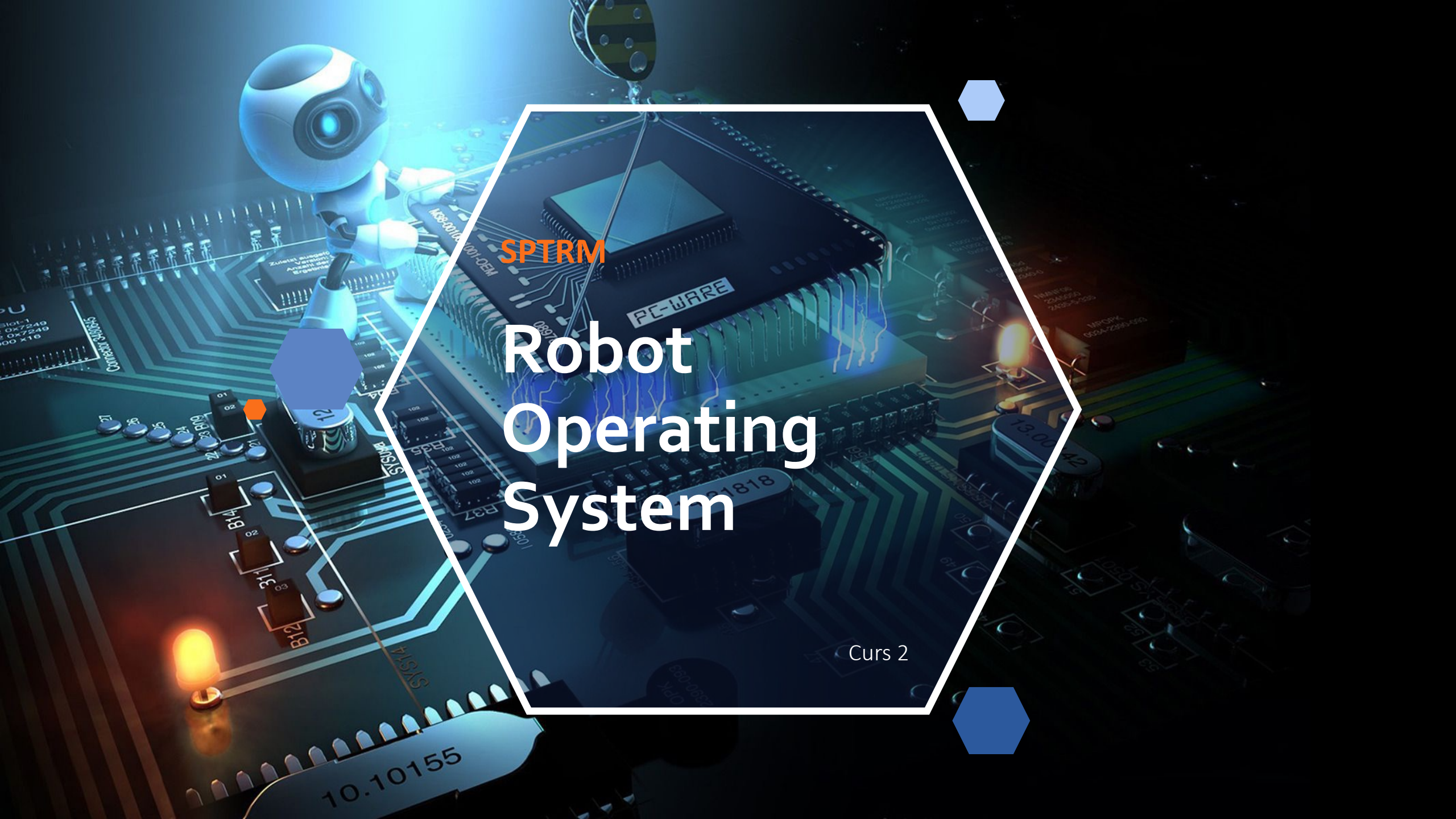


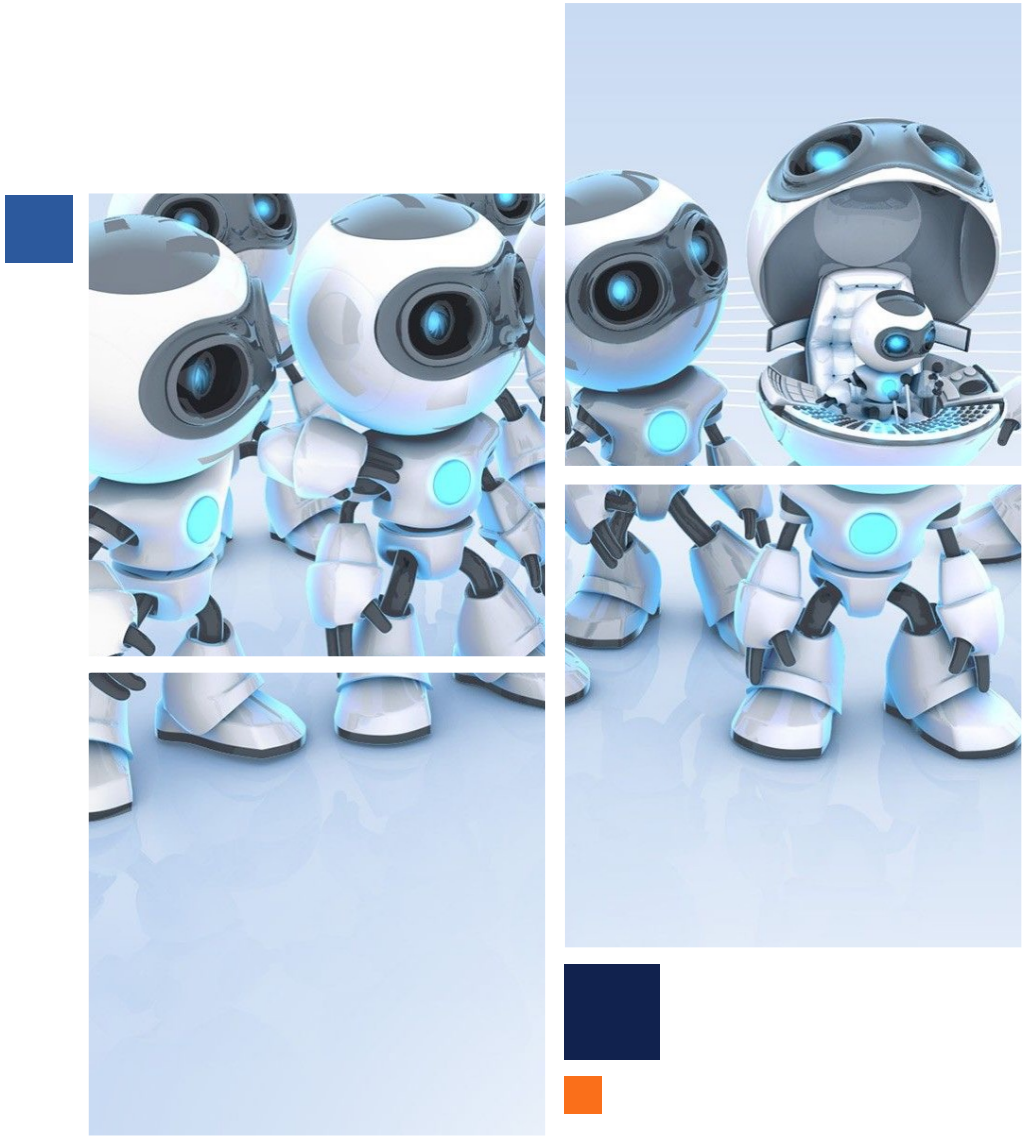
SPTRM

Robot Operating System

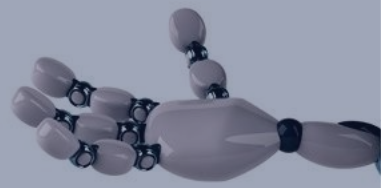
Curs 2

Agenda

- Communication in ROS
- Naming
  - Namespaces
  - Naming conventions
- Packages
- ROSCPP
  - C++ Library
  - Publisher-Subscriber use case

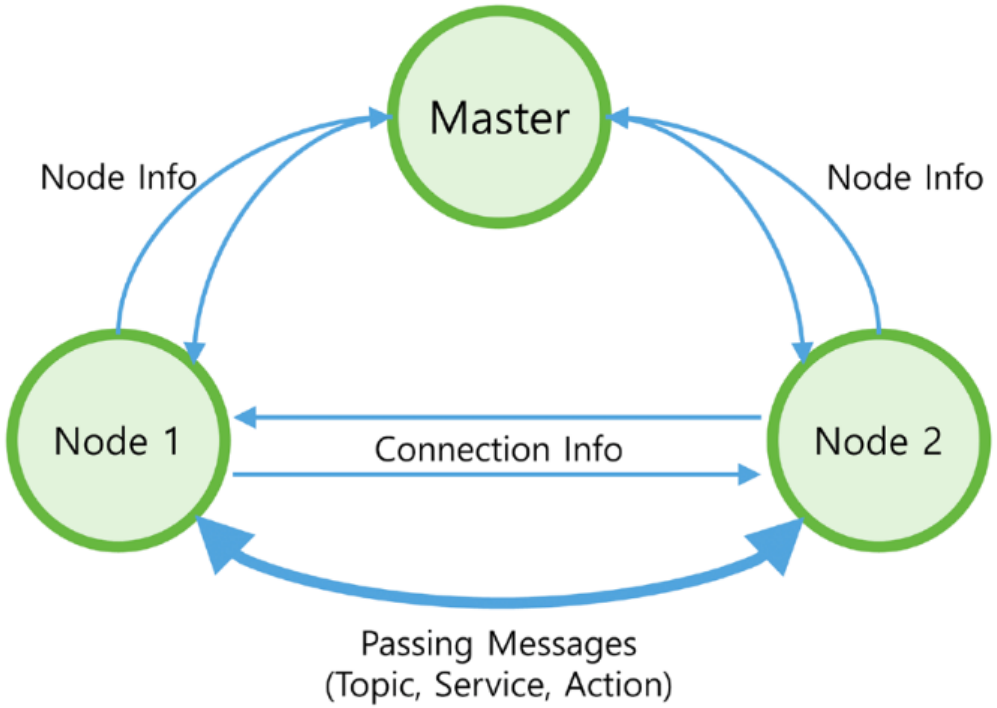


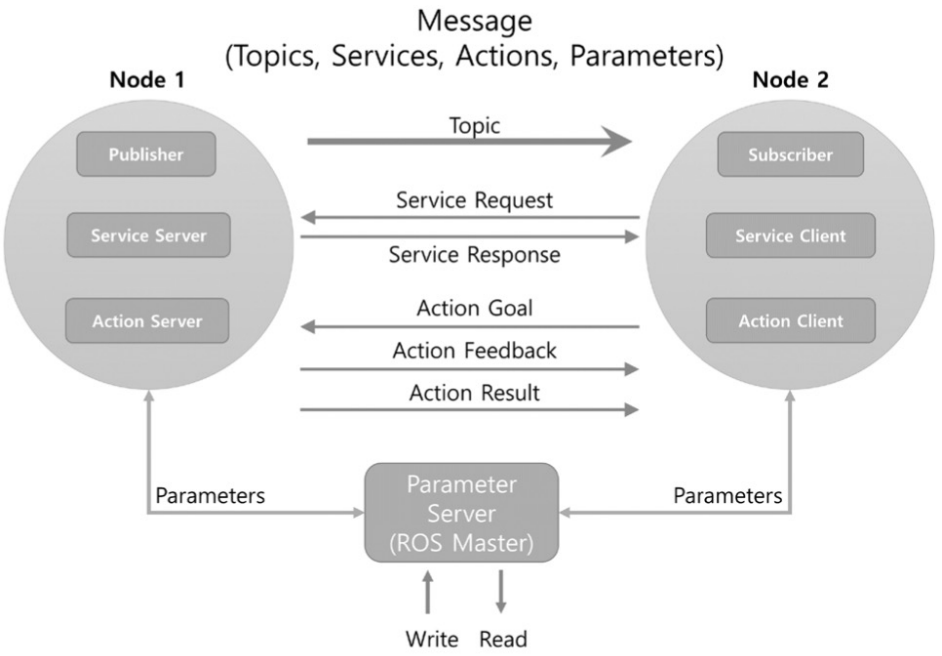
Communication in ROS



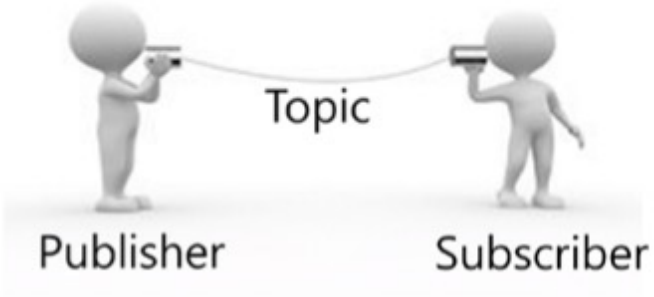
Topics. Services. Actions.
Messages. Parameter Server

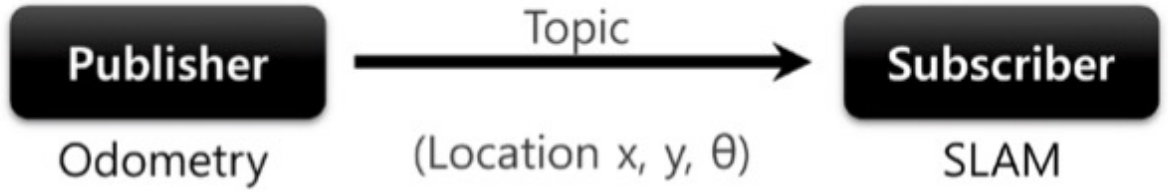
ROS Communication

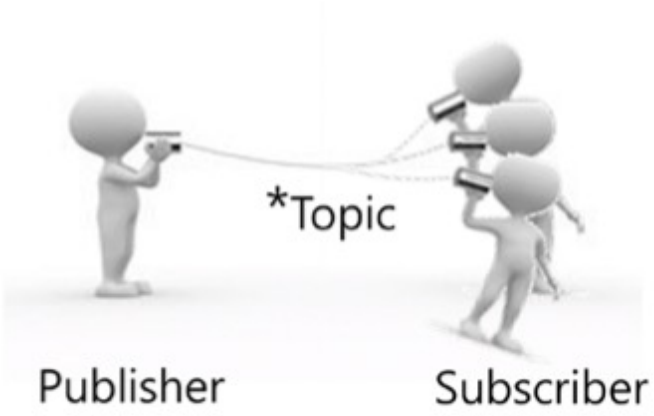


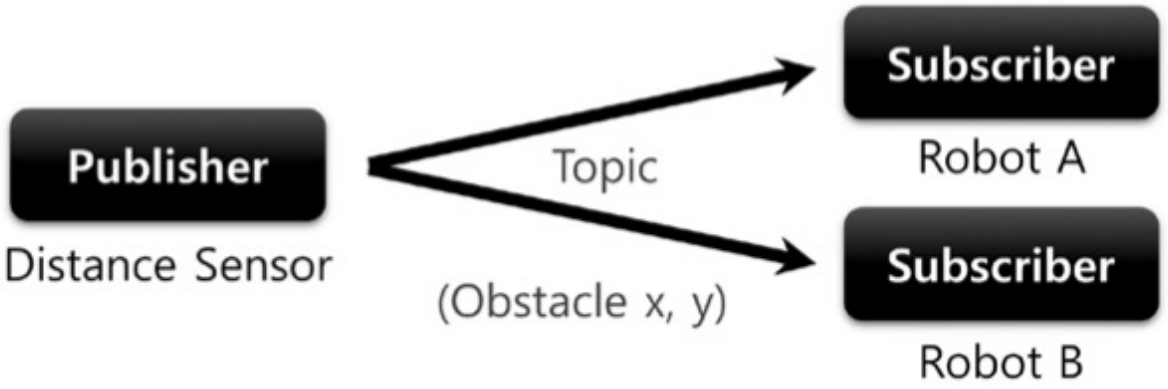


Topic

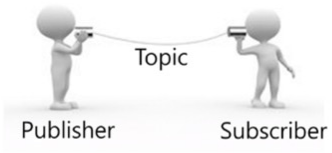

Topic
Publisher
Subscriber

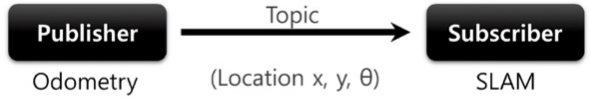

Publisher
Topic
Subscriber
Odometry
(Location x, y, θ)
SLAM

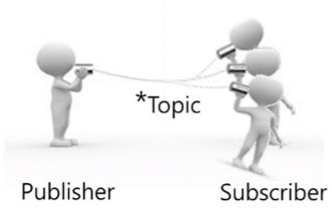

*Topic
Publisher
Subscriber

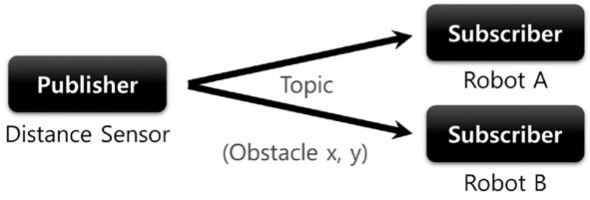

Publisher
Distance Sensor
Topic
(Obstacle x, y)
Subscriber
Robot A
Subscriber
Robot B

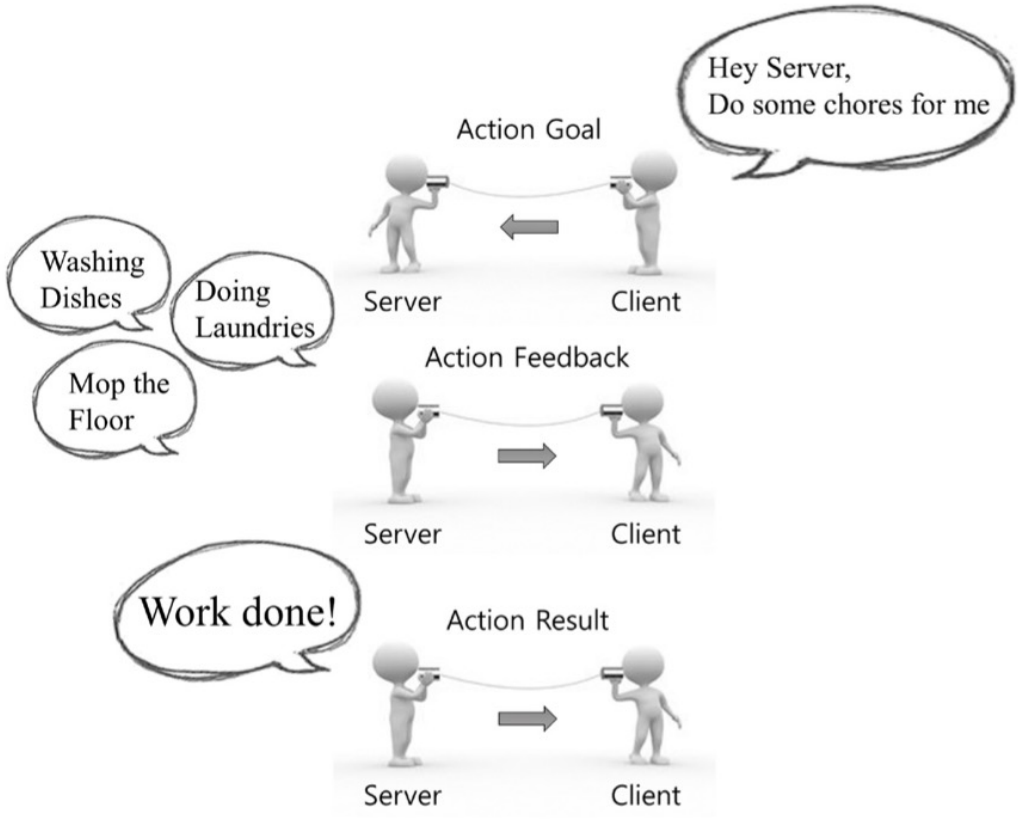
Service vs Action











Service vs Action

Type	Features	Direction	Description
Topic	Asynchronous	Unidirectional	Used when exchanging data continuously
Service	Synchronous	Bidirectional	Used when request processing requests and responds current states
Action	Asynchronous	Bidirectional	Used when it is difficult to use the service due to long response times after the request or when an intermediate feedback value is needed

1. Running the Master

- A master that manages connection information in a message communication between nodes is an essential element that must be run first in order to use RO
- The ROS master is run by using the ‘roscore’ command and runs the server with XMLRPC
- The master registers the name of nodes, topics, services, action, message types, URI addresses and ports for node-to-node connections, and relays the information to other nodes upon request

```
$ roscore
```

XMLRPC (XML-Remote Procedure Call)

RPC protocol

XML as the encoding format

a very simple protocol

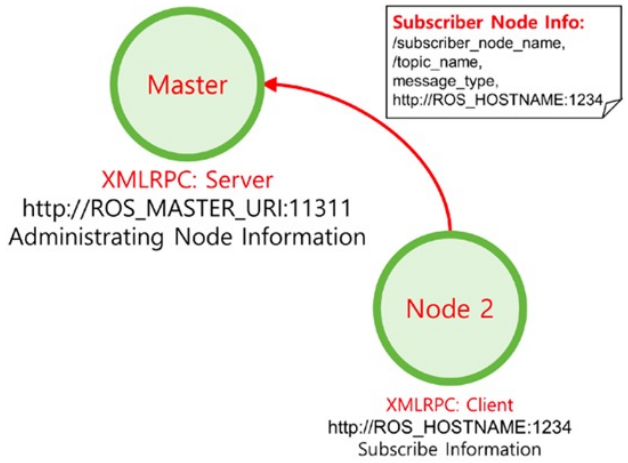
very lightweight

supports multiple programming languages

2. Running the Subscriber Node

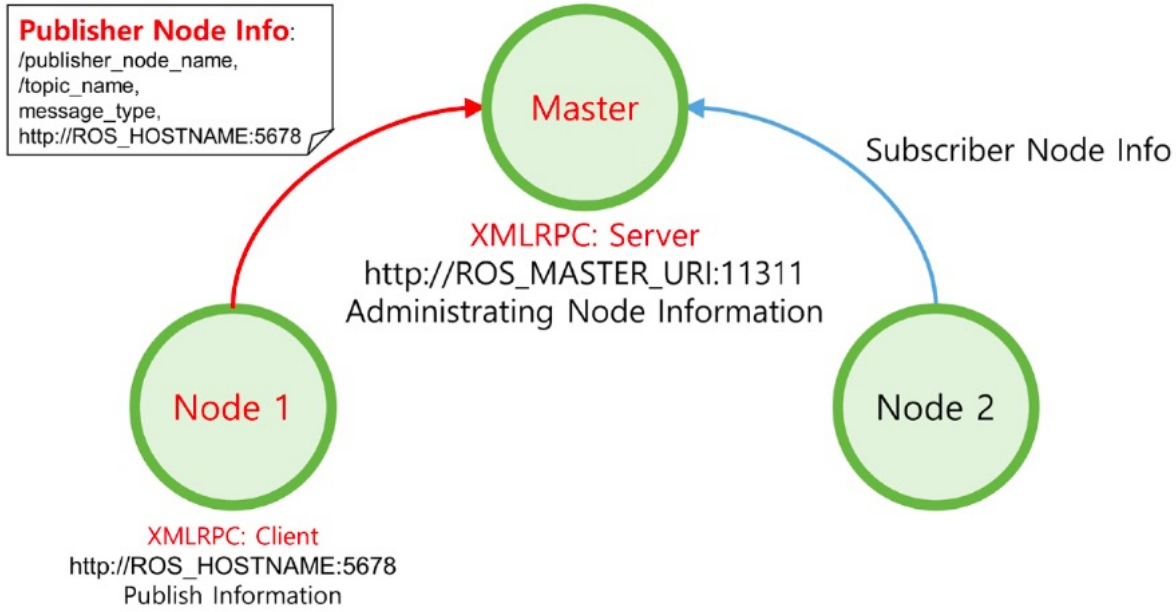
- Subscriber nodes are launched with either a ‘rosrun’ or ‘roslaunch’ commands
- The subscriber node registers its node name, topic name, message type, URI address, and port with the master as it runs
- The master and node communicate using XMLRPC

```
$ rosrun PACKAGE_NAME NODE_NAME
$ roslaunch PACKAGE_NAME LAUNCH_NAME
```



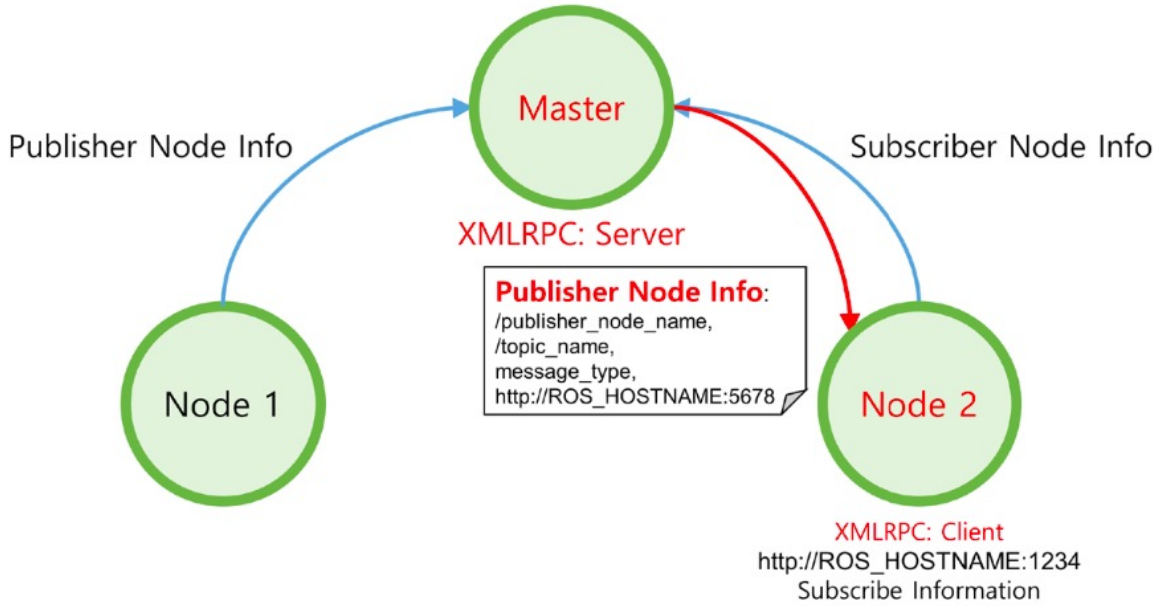
3. Running the Publisher Node

- Publisher nodes, like subscriber nodes, are executed by ‘rosrun’ or ‘roslaunch’ commands.
- The publisher node registers its node name, topic name, message type, URI address and port with the master.
- The master and node communicate using XMLRPC.



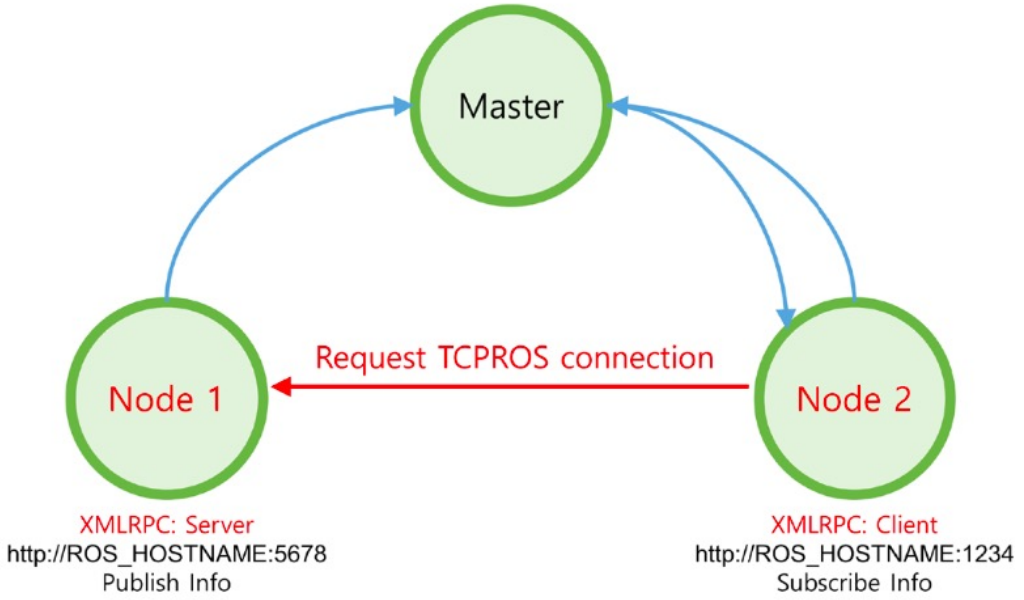
4. Providing Publisher Information

- The master distributes information such as the publisher's name, topic name, message type, URI address and port number of the publisher to subscribers that want to connect to the publisher node
- The master and node communicate using XMLRPC



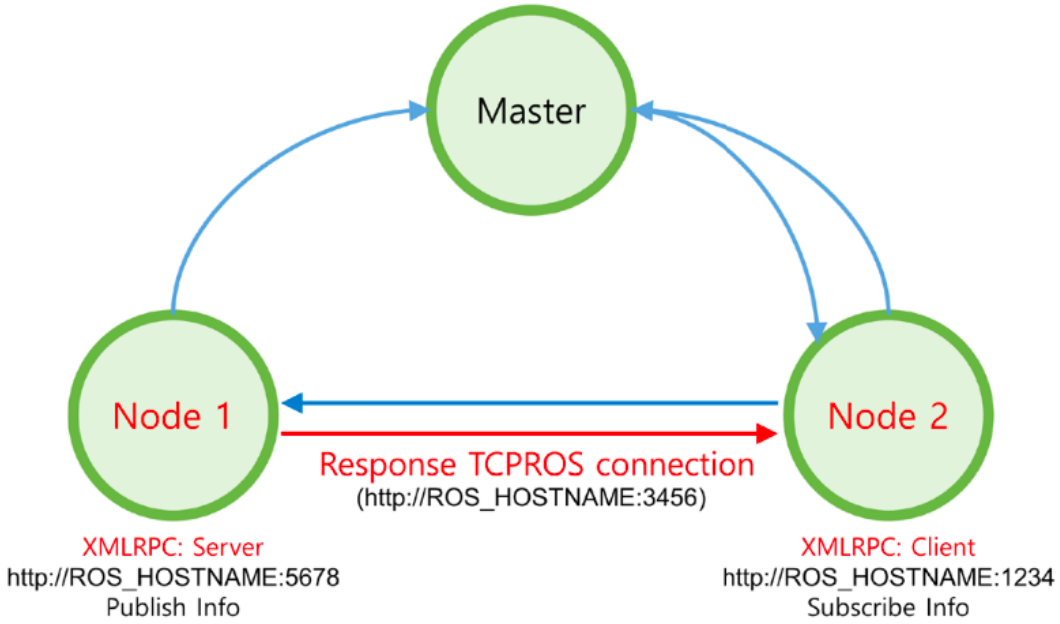
5. Connection Request from the Subscriber Node

- The subscriber node requests a direct connection to the publisher node based on the publisher information received from the master
- The subscriber node transmits information to the publisher node such as the subscriber node's name, the topic name, and the message type
- The publisher node and the subscriber node communicate using XMLRPC.



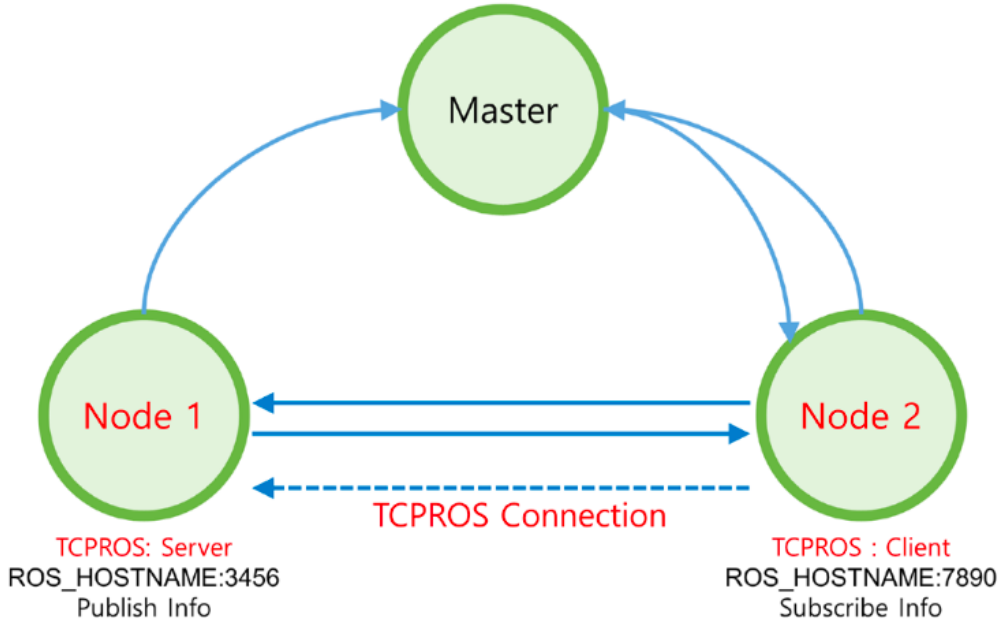
6. Connection Response from the Publisher Node

- The publisher node sends the URI address and port number of its TCP server in response to the connection request from the subscriber node
- The publisher node and the subscriber node communicate using XMLRPC



7. TCPROS Connection

- The subscriber node creates a client for the publisher node using TCPROS, and connects to the publisher node



TCPROS

TCPROS is a message format based on TCP/IP (UDPROS is a message format based on UDP)

TCPROS is a transport layer for ROS Messages and Services.

It uses standard TCP/IP sockets for transporting message data. Inbound connections are received via a TCP Server Socket with a header containing message data type and routing information.

TCPROS for Topics

Subscriber request

- message_definition
- callerid
- topic
- md5sum
- type
- (optional) tcp_nodelay

Publisher response

- md5sum
- type
- (optional) callerid
- (optional) latching

TCPROS for Services

Service client request

- message_definition
- callerid
- topic
- md5sum
- type
- (optional) tcp_nodelay

Service response

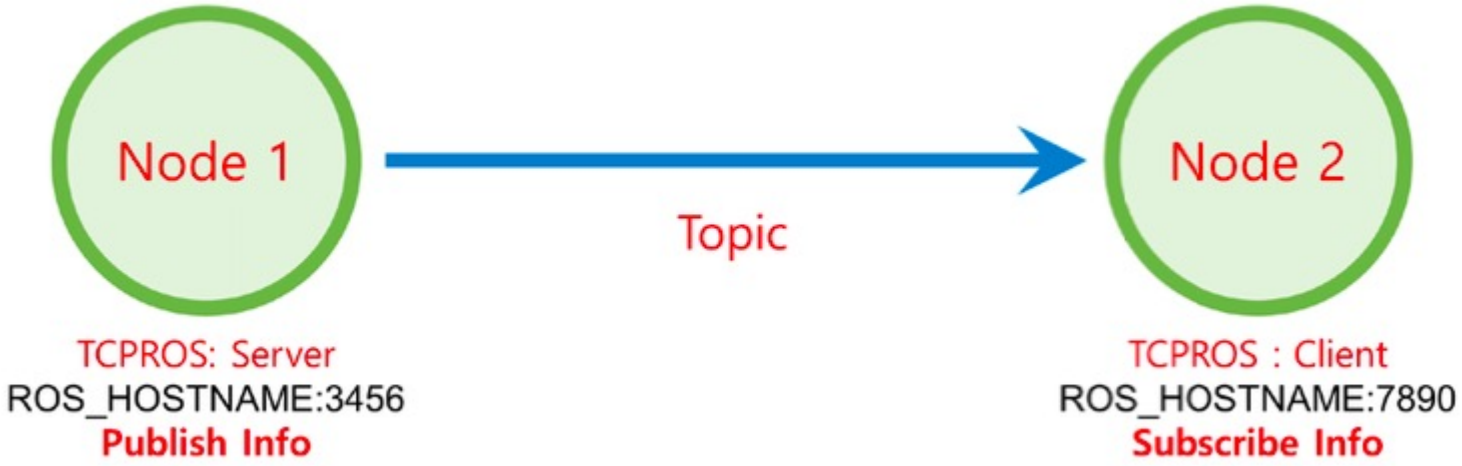
- callerid

Other TCPROS fields:

- Error: human-readable error message
- OK byte

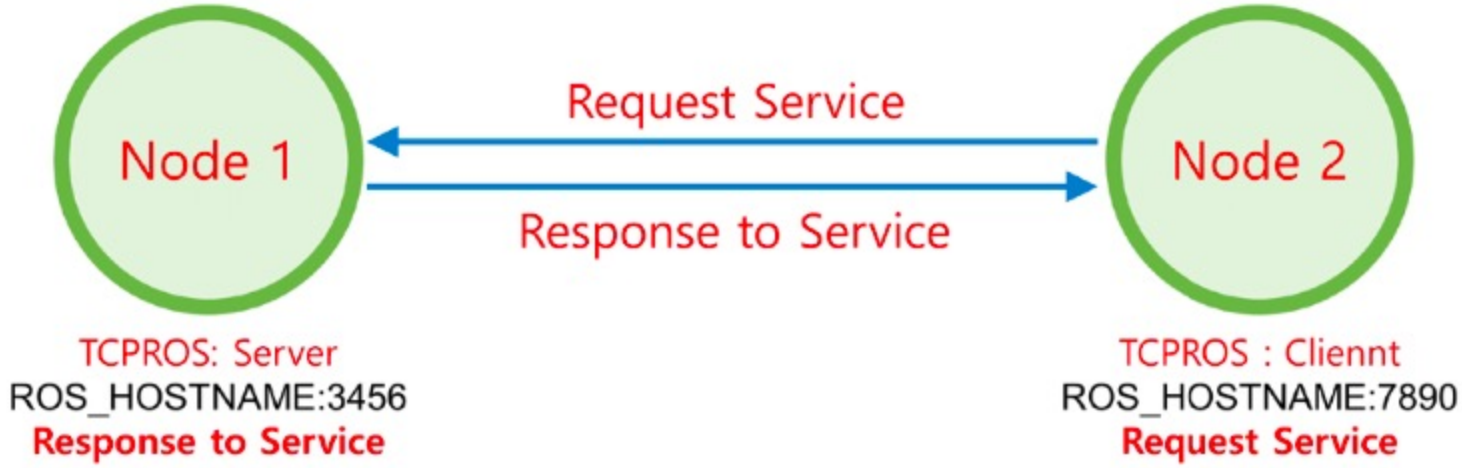
8. Message Transmission

- The publisher node transmits a predefined message to the subscriber node
- The communication between nodes uses TCPROS



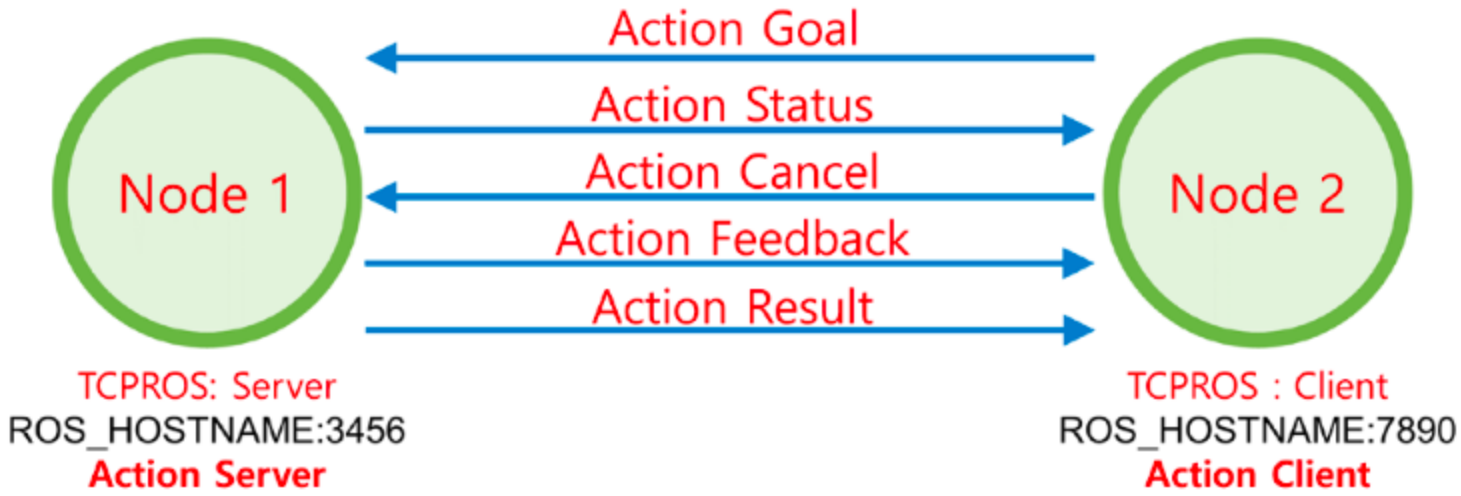
8’. Service Request and Response

- Mostly same as in topic
- Connection is terminated (or not) depending on configuration
- One TCPROS connection can be used for multiple service requests



8’. Action Goal, Result, Feedback

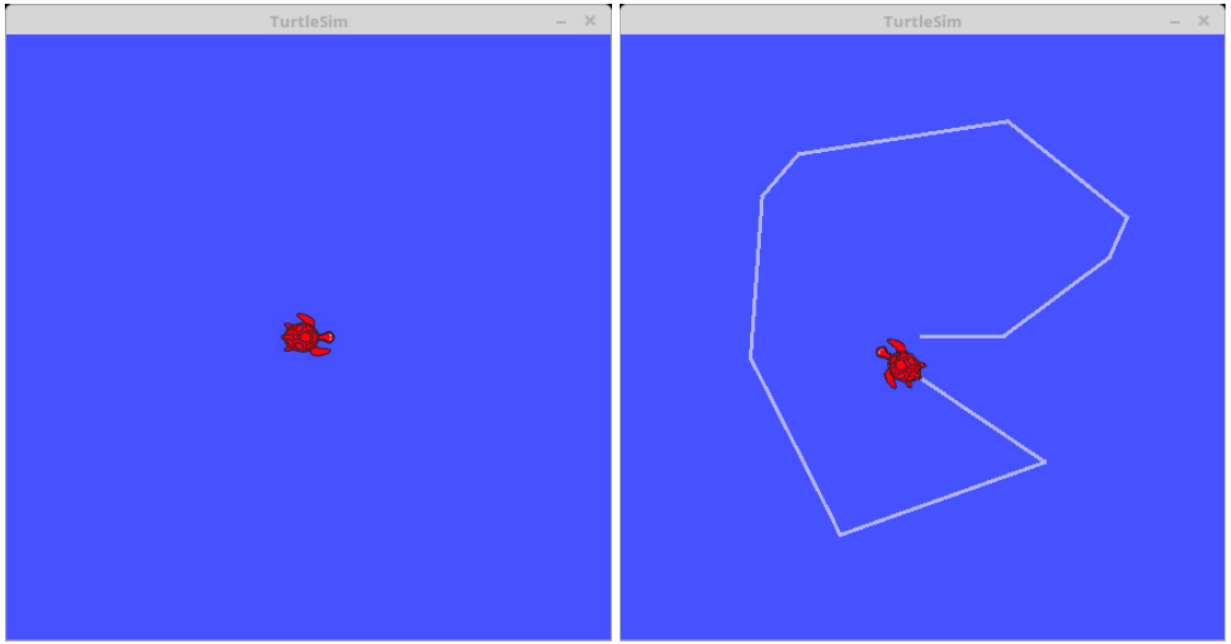
- Similar to service
- Additional feedback messages
- 5 topics used: goal, status, cancel, result and feedback
- Connection is terminated on cancel or on result



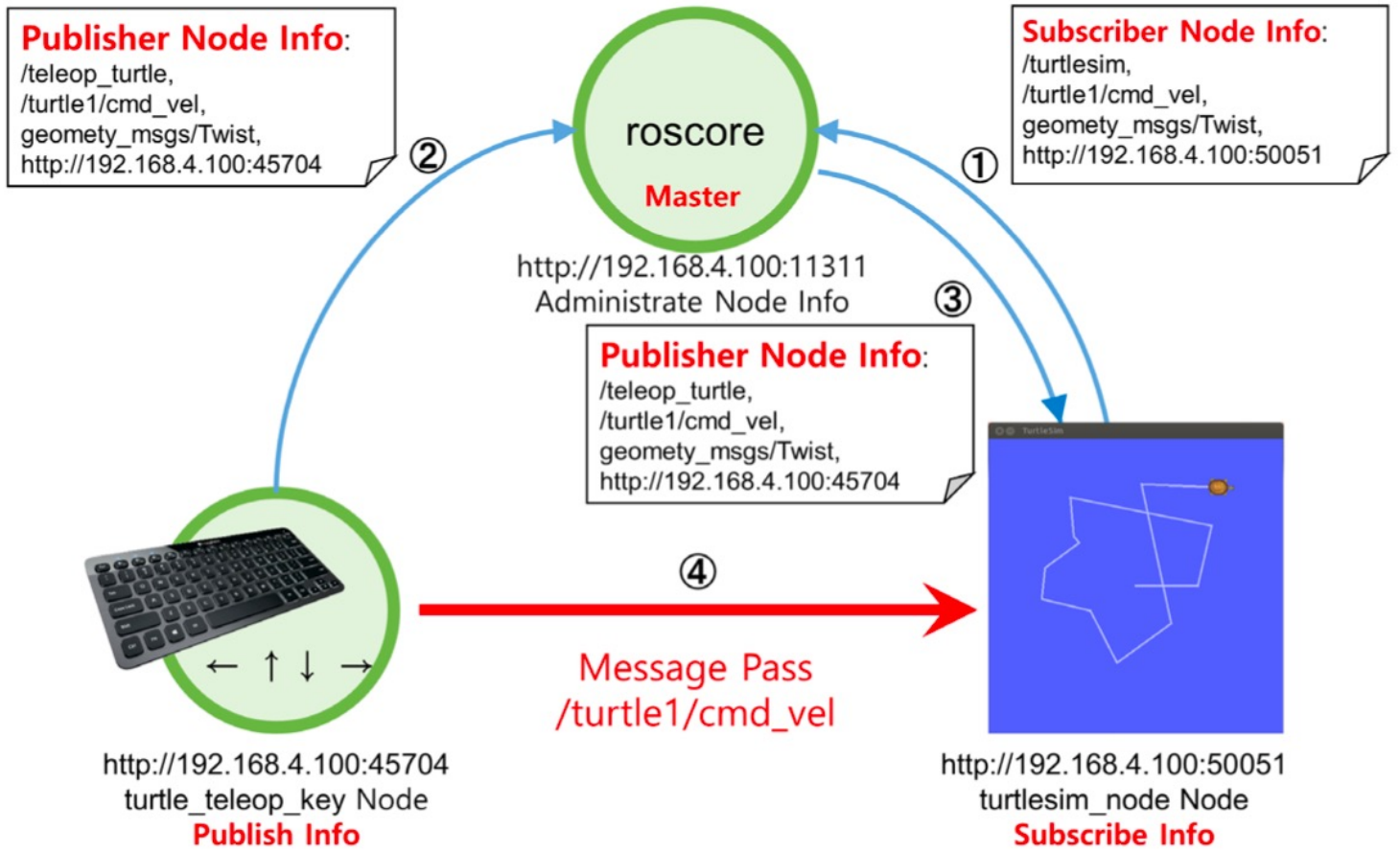
Example

```
$ rosrun turtlesim turtlesim_node
[INFO] [1499182058.960816044]: Starting turtlesim with node name /turtlesim
[INFO] [1499182058.966717811]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445],
theta=[0.000000]
```

```
$ rosrun turtlesim turtle_teleop_key
Reading from keyboard
---------------------------
Use arrow keys to move the turtle.
```



Example explained



Messages

```
fieldtype1 fieldname1
fieldtype2 fieldname2
fieldtype3 fieldname3
```

- Bundle of data to be exchanged between nodes
- Used by topics, services and actions
- Can include basic data type or complex types
- Inheritance
- Inclusion
- Defined in a *.msg file

Basic data types for messages

ROS Data Type	Serialization	C++ Data Type	Python Data Type
bool	unsigned 8-bit int	uint8_t	bool
int8/16/32	signed 8/16/32-bit int	int8_t, int16_t, int32_t	int
uint8/16/32	unsigned 8/16/32-bit int	uint8_t, uint16_t, uint32_t	int
int64	signed 64-bit int	int64_t	long
uint64	unsigned 64-bit int	uint64_t	long
float32	32-bit IEEE float	float	float
float64	64-bit IEEE float	double	float
string	ascii string	std::string	str
time	secs/nsecs unsigned 32-bit ints	ros::Time	rospy.Time
duration	secs/nsecs signed 32-bit ints	ros::Duration	rospy.Duration

Array data types for messages

ROS Data Type	Serialization	C++ Data Type	Python Data Type
fixed-length	no extra serialization	boost::array, std::vector	tuple
variable-length	uint32 length prefix	std::vector	tuple
uint8[]	uint32 length prefix	std::vector	bytes
bool[]	uint32 length prefix	std::vector<uint8_t>	list of bool

ROS Parameter Server

- Shared dictionary accessed via network APIs
- Nodes use this server to store and retrieve parameters at runtime.
- Not designed for high-performance, it is best used for static, non-binary data such as configuration parameters
- Globally viewable so that tools can easily inspect the configuration state of the system and modify if necessary.
- The Parameter Server is implemented using XMLRPC and runs inside of the ROS Master

ROS Parameter Server

config.yaml

```
camera:
  left:
    name: left_camera
    exposure: 1
  right:
    name: right_camera
    exposure: 1.1
```

package.launch

```
<launch>
  <node name="name" pkg="package" type="node_type">
      <rosparam command="load"
                file="$(find package)/config/config.yaml" />
  </node>
</launch>
```

```
ros::NodeHandle nodeHandle("~");
std::string topic;
if (!nodeHandle.getParam("topic", topic)) {
  ROS_ERROR("Could not find topic
             parameter!");
}
```

ROS Parameter Server

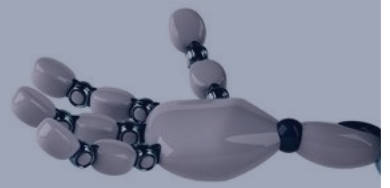
Data Types

- 32-bit integers
- Booleans
- Strings
- Doubles
- ISO8601 dates
- Lists
- Base64-encoded binary data

ROS Param Tools

- rosparam list
- rosparam get <parameter>
- rosparam set <parameter> <value>
- rosparam delete <parameter>
- rosparam dump <file> <namespace>
- rosparam load <file> <namespace>

Naming in ROS



Structure. Node Handles.

Graph resource names

- ROS has a flexible naming system that accepts several different kinds of names
- Nodes, topics, services, and parameters are collectively referred to as graph resources
- Every graph resource is identified by a short string called a **graph resource name**
- Graph Resource Names are an important mechanism in ROS for providing encapsulation
- Each resource is defined within a namespace, which it may share with many other resources.
- In general, resources can create resources within their namespace and they can access resources within or above their own namespace.
- Connections can be made between resources in distinct namespaces, but this is generally done by integration code above both namespaces.
- This encapsulation isolates different portions of the system from accidentally grabbing the wrong named resource or globally hijacking names.

$$\underbrace{/a/b/c/d/e/f}_{\text{default namespace}} + \underbrace{g/h/i/j/k/l}_{\text{relative name}} \Rightarrow \underbrace{/a/b/c/d/e/f/g/h/i/j/k/l}_{\text{global name}}$$

`export ROS_NAMESPACE=`*`default-namespace`*

ROS Naming concept

- Graph-based basic concept
- Everything has a unique name (nodes, topic, services, etc.)
- Domains (global, relative, private)

```
int main(int argc, char **argv)              // Node Main Function
{
  ros::init(argc, argv, "node1");            // Node Name Initialization
  ros::NodeHandle nh;                        // Node Handle Declaration
  // Publisher Declaration, Topic Name = bar
  ros::Publisher node1_pub = nh.advertise<std_msg::Int32>("bar", 10);
```

```
ros::Publisher node1_pub =   nh.advertise<std_msg::Int32>("/bar", 10);
```

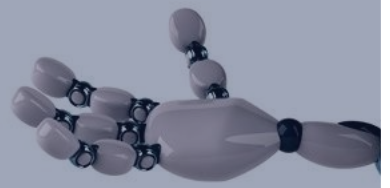
```
ros::Publisher node1_pub =   nh.advertise<std_msg::Int32>("~bar", 10);
```

ROS Naming concept

- Graph-based basic concept
- Everything has a unique name (nodes, topic, services, etc.)

Node	Relative	Global	Private
/node1	bar => /bar	/bar => /bar	~bar => /node1/bar
/wp/node2	bar => /wp/bar	/bar => /bar	~bar => /wp/node2/bar
/wp/node3	Foo/bar => /wg/foo/bar	/foo/bar => /foo/bar	~foo/bar => /wp/node3/foo/bar

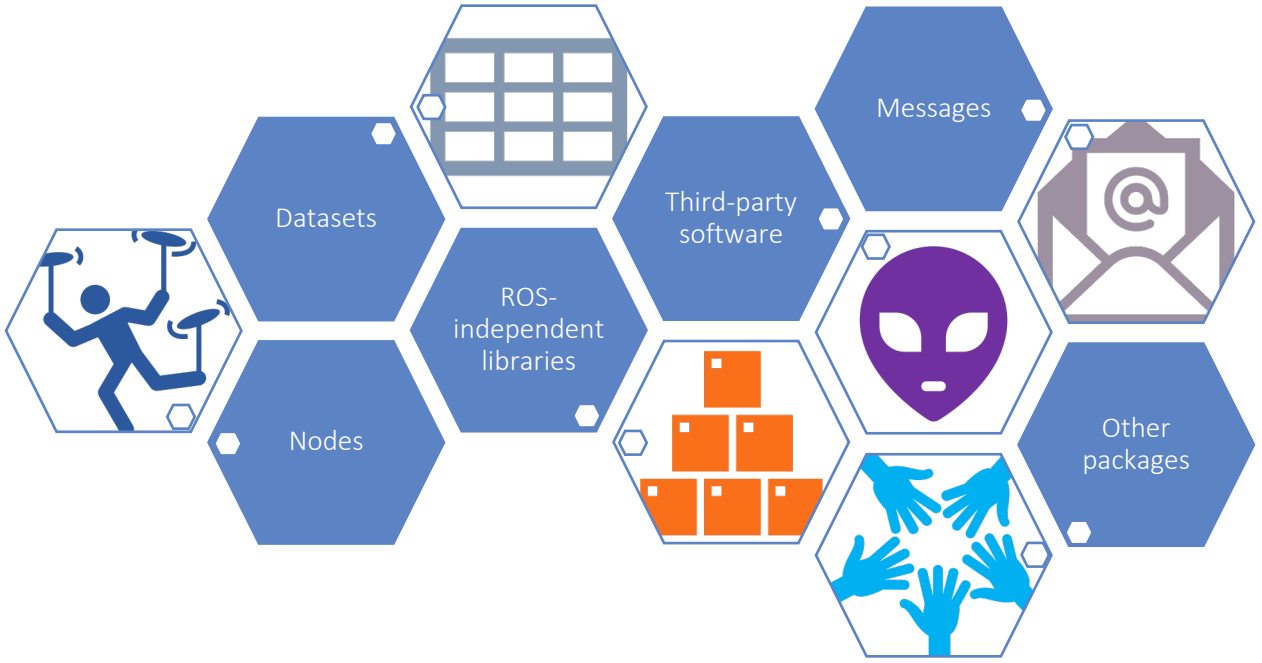
ROS Packages



Structure. Logging

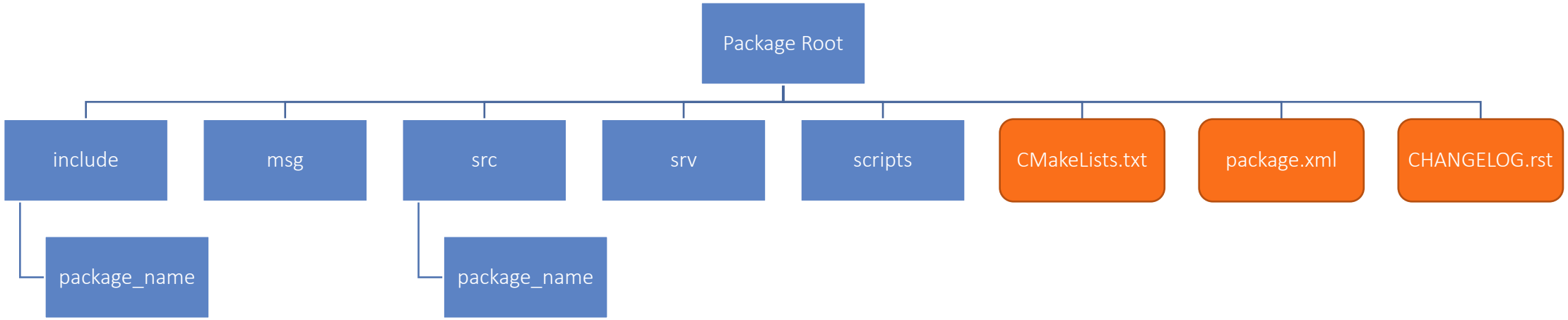
ROS Packages

- The goal of these packages it to provide this useful functionality in an easy-to-consume manner so that software can be easily reused
- Organization unit for software in ROS
- Created by hand or using tools (e.g. catkin_create_pkg)
- Extra tools (e.g. rospack, rosdep, catkin_make)



ROS Package Structure

- Common structure
- Contain source code, launch files, configuration files, message definitions, data, documentation



My First Package... Package.xml

```
$ catkin_create_pkg my_first_ros_pkg std_msgs roscpp
```

```
<?xml version="1.0"?>
<package>
  <name>my_first_ros_pkg</name>
  <version>0.0.1</version>
  <description>The my_first_ros_pkg package</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>
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>roscpp</build_depend>
  <run_depend>std_msgs</run_depend>
  <run_depend>roscpp</run_depend>
  <export></export>
</package>
```

My First Package… CMakeLists.txt

1. Required CMake Version
   cmake_minimum_required
2. Package Name
   project()
3. Find other CMake/Catkin packages needed for build
   find_package()
4. Message/Service/Action Generators
   add_message_files(), add_service_files(), add_action_files()
5. Invoke message/service/action generation
   generate_messages()
6. Specify package build info export
   catkin_package()
7. Libraries/Executables to build
   add_library()/add_executable()/target_link_libraries()
8. Tests to build
   catkin_add_gtest()
9. Install rules
   install()

CMakeLists.txt

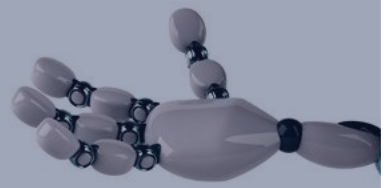
```
cmake_minimum_required(VERSION 2.8.3)
project(ros_package_template)

## Use C++11
add_definitions(--std=c++11)

## Find catkin macros and libraries
find_package(catkin REQUIRED
  COMPONENTS
    roscpp
    sensor_msgs
)

…
```

ROSCPP



Publisher - Subscriber Study case implemented in C++

ROS C++ Client Library (roscpp)

hello_world.cpp

```
#include <ros/ros.h>

int main(int argc, char** argv)
{
  ros::init(argc, argv, "hello_world");
  ros::NodeHandle nodeHandle;
  ros::Rate loopRate(10);

  unsigned int count = 0;
  while (ros::ok()) {
    ROS_INFO_STREAM("Hello World " << count);
    ros::spinOnce();
    loopRate.sleep();
    count++;
  }

  return 0;
}
```

- ROS main header file include
- `ros::init(…)` has to be called before calling other ROS functions
- The node handle is the access point for communications with the ROS system (topics, services, parameters)
- `ros::Rate` is a helper class to run loops at a desired frequency
- `ros::ok()` checks if a node should continue running
  Returns `false` if `SIGINT` is received (`Ctrl + C`) or `ros::shutdown()` has been called
- `ROS_INFO()` logs messages to the filesystem
- `ros::spinOnce()` processes incoming messages via callbacks

More info
http://wiki.ros.org/roscpp
http://wiki.ros.org/roscpp/Overview

ROS Node Handle

- There are four main types of node handles

1. Default (public) node handle:
   `nh_ = ros::NodeHandle();`
2. Private node handle:
   `nh_private_ = ros::NodeHandle("~");`
3. Namespaced node handle:
   `nh_eth_ = ros::NodeHandle("eth");`
4. Global node handle:
   `nh_global_ = ros::NodeHandle("/");`

Recommended

Not recommended

For a ***node*** in ***namespace*** looking up ***topic***, these will resolve to:

```
/namespace/topic

/namespace/node/topic

/namespace/eth/topic

/topic
```

More info
http://wiki.ros.org/roscpp/Overview/NodeHandles

ROS + OOP = ?

my_package_node.cpp

```
#include <ros/ros.h>
#include "my_package/MyPackage.hpp"
int main(int argc, char** argv)
{
  ros::init(argc, argv, "my_package");
  ros::NodeHandle nodeHandle("~");

  my_package::MyPackage myPackage(nodeHandle);

  ros::spin();
  return 0;
}
```

MyPackage.hpp

MyPackage.cpp

`class MyPackage`

Main node class providing ROS interface (subscribers, parameters, timers etc.)

Algorithm.hpp

Algorithm.cpp

`class Algorithm`

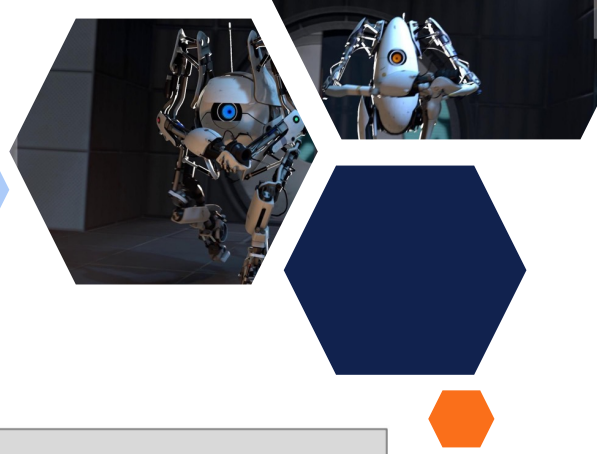
Class implementing the algorithmic part of the node

Note: The algorithmic part of the code could be separated in a (ROS-independent) library

! Specify a function handler to a method from within the class as

```
subscriber_ = nodeHandle_.subscribe(topic, queue_size,
&ClassName::methodName, this);
```

More info
http://wiki.ros.org/roscpp_tutorials/Tutorials/UsingClassMethodsAsCallbacks



Publisher

- Create a publisher with help of the node handle

```
ros::Publisher publisher =
nodeHandle.advertise<message_type>(topic,
queue_size);
```

- Create the message contents
- Publish the contents with

```
publisher.publish(message);
```

```
#include <ros/ros.h>
#include <std_msgs/String.h>

int main(int argc, char **argv) {
  ros::init(argc, argv, "talker");
  ros::NodeHandle nh;
  ros::Publisher chatterPublisher =
    nh.advertise<std_msgs::String>("chatter", 1);
  ros::Rate loopRate(10);

  unsigned int count = 0;
  while (ros::ok()) {
    std_msgs::String message;
    message.data = "hello world " + std::to_string(count);
    ROS_INFO_STREAM(message.data);
    chatterPublisher.publish(message);
    ros::spinOnce();
    loopRate.sleep();
    count++;
  }
  return 0;
}
```

More info
http://wiki.ros.org/roscpp/Overview/Publishers%20and%20Subscribers

Subscriber

- Start listening to a topic by calling the method `subscribe()` of the node handle

```
ros::Subscriber subscriber =
nodeHandle.subscribe(topic, queue_size,
                     callback_function);
```

- When a message is received, callback function is called with the contents of the message as argument
- Hold on to the subscriber object until you want to unsubscribe

`ros::spin()` processes callbacks and will not return until the node has been shutdown

listener.cpp

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

void chatterCallback(const std_msgs::String& msg)
{
  ROS_INFO("I heard: [%s]", msg.data.c_str());
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener");
  ros::NodeHandle nodeHandle;

  ros::Subscriber subscriber =
       nodeHandle.subscribe("chatter",10,chatterCallback);
  ros::spin();
  return 0;
}
```

More info
http://wiki.ros.org/roscpp/Overview/Publishers%20and%20Subscribers

Thank you

Question Time