Full Robot Operating System (ROS) Training v21

Trainer: Man Guo Chang

TINFOTECH

Website:www.tertiarycourses.com.sg
Email: enquiry@tertiaryinfotech.com

About the Trainer

Mr. Man Guo Chang graduated from Nanyang Technological University, School of Electrical and Electronic Engineering, major in Computer Engineering.

Mr. Man has more than 25 years of working experience in the Semiconductor field, specialized in IC Testing, Product Engineering, Data Analysis, and Software Development.

Mr. Man is an ACTA certified trainer. His skill set includes Website Development, Software Development, Machine Vision, Internet of Things, ROS, Cyber Security, etc.

Let's Know Each Other...

Say a bit about yourself
- Name
- What Industry you are from?
- Do you have any prior knowledge in Python and Linux?
- Why do you want to learn ROS?
- What do you want to learn from this course?

Ground Rules

- Set your mobile phone to silent mode
- Actively participate in the class. No question is stupid.
- Respect each other view
- Exit the class silently if you need to step out for phone call, toilet break

Ground Rules for Virtual Training

- Upon entering, mute your mic and turn on the video. Use a headset if you can
- Use the 'raise hand' function to indicate when you want to speak
- Participant actively. Feel free to ask questions on the chat whenever.
- Facilitators can use breakout rooms for private sessions.

Guidelines for Facilitators

1. Once all the participants are in and introduce themselves
2. Goto gallery mode, take a snapshot of the class photo - makes sure capture the date and time
3. Start the video recording (only for WSQ courses)
4. Continue the class
5. Before the class end on that day, take another snapshot of the class photo - makes sure capture the date and time
6. For NRIC verification, facilitator to create breakout room for individual participant to check (only for WSQ courses)
7. Before the assessment start, take another snapshot of the class photo - makes sure capture the date and time (only for WSQ courses)
8. For Oral Questioning assessment, facilitator to create breakout room for individual participant to OQ (only for WSQ courses)
9. End the video recording and upload to cloud (only for WSQ courses)
10. Assessor to send all the assessment records, assessment plan and photo and video to the staff (only for WSQ courses).

Prerequisite

The following knowledge is assumed

- Basic Python or C++ programming
- Basic Linux

Agenda

Topic 1 Introduction to ROS

- What is ROS
- Why ROS ?
- ROS Applications and Eco Systems
- ROS Installation on Ubuntu
- ROS Master

Topic 2 ROS Packages & Nodes

- ROS Workspace & catkin
- Create a Catkin Workspace
- Create a Package
- Install a Package
- ROS Nodes
- ROS Launch

Agenda

Topic 3 ROS Topics, Services & Actions

- ROS Topics and Messages
- Publishers and Subscribers
- ROS Services
- ROS Actions

Topic 4 ROS Bags

- What is a ROSBag
- Record and Playback a ROSBag
- Visualize ROSBag with RViz

Topic 5 TF and URDF

- What is Transformation System (TF)
- TF Tools
- What is Unified Robotics Description Format (URDF)
- URDF File Format
- Create a URDF File

Agenda

Topic 6 Program ROS Nodes and Topics

- Program ROS Publisher Nodes and Topics (Python)
- Program ROS Subscriber Nodes and Topics (Python)
- Create Launch file

Topic 7 Program ROS Messages

- Create Custom Message Type
- Program Nodes with Custom Message Type (Python)

Topic 8 Program ROS Services

- Create Custom Service Type
- Program ROS Services with Custom Service Type (Python)

Topic 9 Program ROS Actions

- Create Custom Action Type
- Program ROS Actions with Custom Service Type (Python)

Download Course Material

- The resources can be found on the Google classroom
- Goto https://classroom.google.com and enter the class code below
- If you have problem to access the Google Classroom, please inform trainer or the staff
- Goto Classwork > Course Material

yydixqx

Topic 1
Introduction to ROS

What is ROS

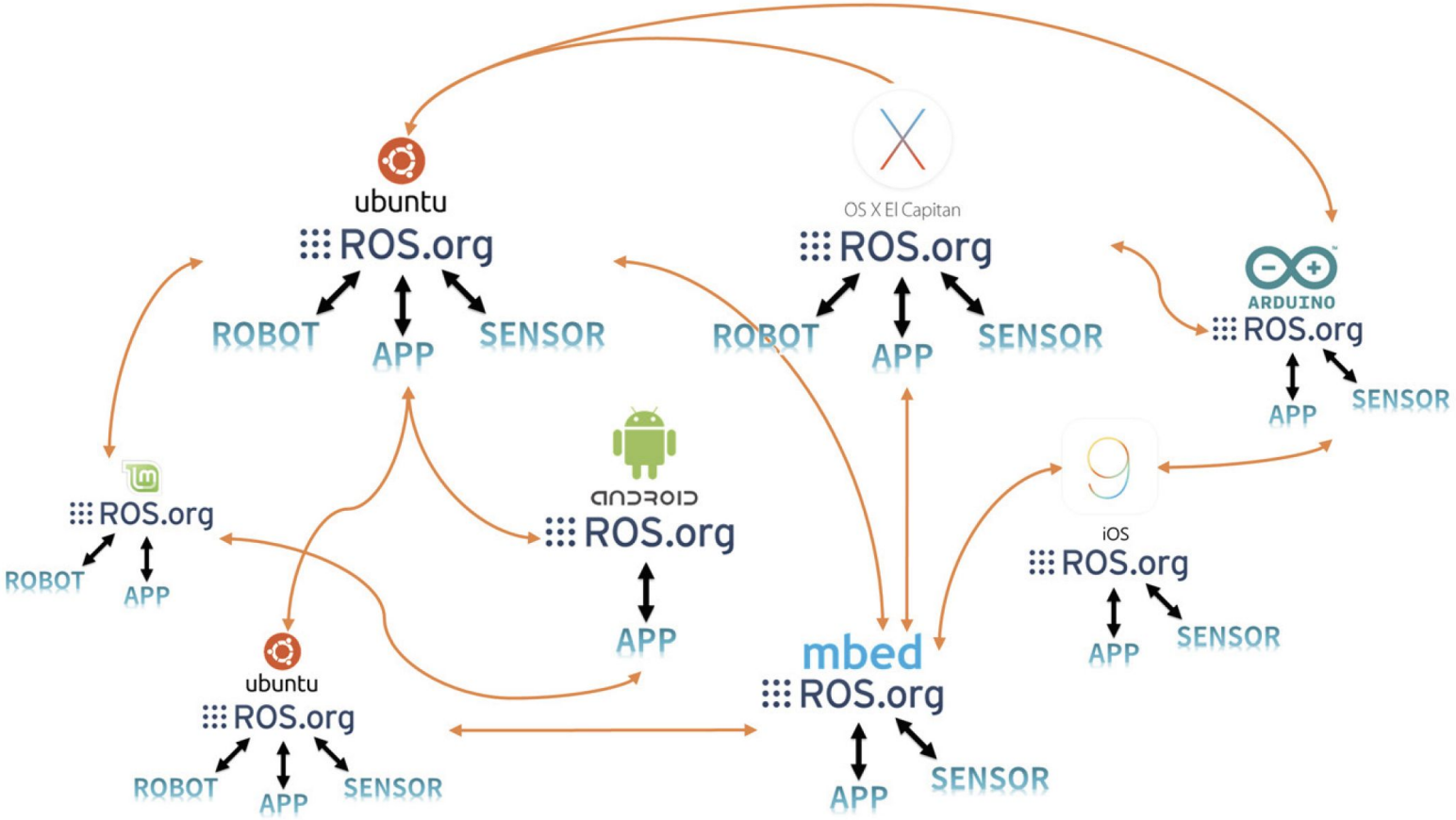
- ROS = Robot Operating System
- ROS is an open-source, meta-operating system for your robot.
- It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes,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

What is Meta Operating System?

- ROS is NOT a conventional OS; ROS is a meta operating system that runs on existing OS such as Window, Mac, Linux etc.

ROS Data Communication

- ROS data communication is supported by multiple operating systems, hardware, and programs, making it highly suitable for robot development where various hardware are combined



ROS Robots

There are hundreds of ROS robots in the market.
https://robots.ros.org/all/

10 Years of ROS

Team MAXed Out

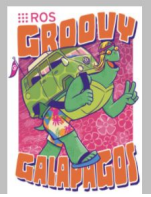
Learning Algorithms and Systems Laboratory (LASA) | EPFL

History of ROS

- 2000's - Robotics projects started at Stanford AI Lab
- 2007 - In May 2007, ROS was started by borrowing the early opens ource robotic software frameworks including switchyard, which is developed by Dr. Morgan Quigley by the Stanford Artificial Intelligence Laboratory in support of the Stanford AI Robot STAIR (STanford AI Robot) project9
- 2007 - In November 2007, U.S. robot company Willow Garage succeeded the development of ROS. Willow Garage is a well-known company in the field for personal robots and service robots. It is also famous for developing and supporting the Point Cloud Library (PCL), which is widely used for 3D devices such as Kinect and the image processing open source library OpenCV.
- 2010 - On January 22, 2010, ROS 1.0 was released
- 2013 - Open Source Robotics Foundation (OSRF) later changed to Open Robotics
- 2013 - Release of Kinetic Kame
- 2014 - Release of Indigo Igloo
- 2017 - Release of Lunar Loggerhead
- 2017 - Release of ROS 2.0
- 2018 - Release of Melodic

ROS Distributions

http://wiki.ros.org/Distributions

Distro	Release date	Poster	*Tuturtle*, turtle in tutorial	EOL date
ROS Noetic Ninjemys (**Recommended**)	May 23rd, 2020			May, 2025 (Focal EOL)
ROS Melodic Morenia	May 23rd, 2018			May, 2023 (Bionic EOL)
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame	May 23rd, 2016			April, 2021 (Xenial EOL)
ROS Jade Turtle	May 23rd, 2015			May, 2017
ROS Indigo Igloo	July 22nd, 2014			April, 2019 (Trusty EOL)
ROS Hydro Medusa	September 4th, 2013			May, 2015
ROS Groovy Galapagos	December 31, 2012			July, 2014

ROS Philosophy

- **Peer to peer**
  Individual programs communicate over defined API (ROS messages, services, etc.).
- **Distributed**
  Programs can be run on multiple computers and communicate over the network.
- **Multi-lingual**
  ROS modules can be written in any language for which a client library exists (C++, Python, MATLAB, Java, etc.).
- **Light-weight**
  Stand-alone libraries are wrapped around with a thin ROS layer.
- **Free and open-source**
  Most ROS software is open-source and free to use.

ROS Components

ROS Ecosystem

- More than 5,000 packages
- 2,818 Official Packages (Indigo, March 2017)
- 13,441,711 .deb Packages Download (July 2017)
- 18,839 Wiki page (July 2017)

Developers, Users

APP

ROS

ROBOT, SENSOR

Robot, Sensor Manufacturer

Applications	fetch beer, elevator ···
Simulators	gazebo, player/stage, STDR Simulator ···
Intelligent Modules	navigation, action, grasping ···
Libraries	tf, PCL, OpenCV, OpenRave ···
Device Drivers	camera_drivers, urg_node ···
Debugging Tools	rviz, rqt_graph, rosbag, rostopic ···
Message Passing	rosmaster, rosmsg, rosservice ···
Execution Tools	rosrun, roslaunch ···
Compile Tools	catkin_make, rosbuild ···
File Systems	roscd, rosls ···
Installer	rosinstall ···
Programming Languages	C++, Python, Lisp, Java, Ruby, MATLAB, etc

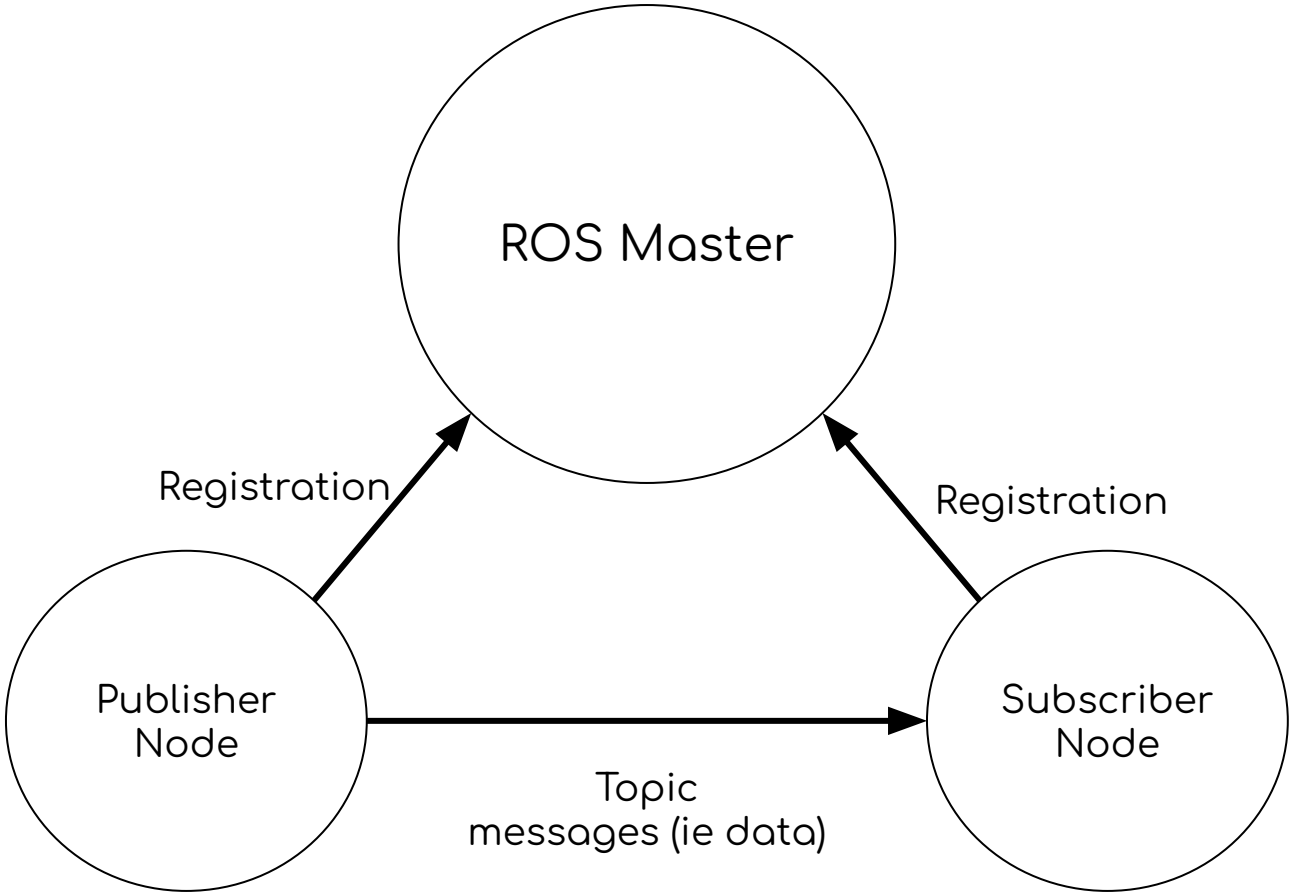
- Supports more than 90 robots and 80 sensors

Install ROS

- To install ROS kinetic distribution on Ubuntu 16.04 http://wiki.ros.org/kinetic/Installation/Ubuntu
- To install ROS on VirtualBox
  - Download and install VirtualBox from https://www.virtualbox.org
  - Download Ubuntu 16.04 image from http://releases.ubuntu.com/16.04/?_ga=2.245247622.286617492.1585822024-1634746839.1585822024.
  - In VirtualBox, click Start -> Install Ubuntu
  - Install ROS as follows:
    $ sudo apt-get update
    $ sudo apt-get upgrade
    $ wget https://raw.githubusercontent.com/ROBOTIS-GIT/robotis_tools/master/install_ros_kinetic.sh && chmod 755
    $ ./install_ros_kinetic.sh && bash ./install_ros_kinetic.sh

ROS Master

- A ROS master manages the communication between nodes
- Every node registers at startup with the master



Start a ROS Master

- To start a ROS master, type on terminal:

  roscore

- roscore will start up:
  - a ROS Master
  - a ROS Parameter Server
  - a rosout logging node
- roscore is the command that runs the ROS master. If multiple computers are within the same network, it can be run from another computer in the network. However, except for special case that supports multiple roscore, only one roscore should be running in the network.

Topic 2
Packages & Nodes

ROS Workspace & Catkin

- Any ROS project begins with making a workspace.
- catkin is the official build system of ROS to generate executables, libraries, and interfaces.
- catkin is the successor to the original ROS build system, rosbuild.
- catkin combines CMake macros and Python scripts to provide some functionality on top of CMake's normal workflow

Create a Workspace

- To create a catkin workspace, type from the following command lines

  mkdir -p catkin_ws/src

  cd catkin_ws

  catkin_make

- After catkin_make, there are two 'build' and 'devel' folders created
- To make the workspace visible to ROS, run the following command
  - **cd ~/catkin_ws**
  - **source devel/setup.bash**

Catkin Build System

catkin_make will create the following folders in the workspace:

Work here

src

The source space contains the source code. This is where you can clone, create, andedit source code for the packages you want to build.

Don't touch

build

The build space is where CMake is invoked to build the packages in the source space. Cache information and other intermediate files are kept here.

Don't touch

devel

The development (devel) space is where built targets are placed (prior to being installed).

Activity: Create a New Workspace

- Create a new workspace on the root directory

```
$ mkdir -p <yr own workspace>/src
$ cd <yr own workspace>
$ catkin_make
$ cd ~/<yr own workspace>
$ source devel/setup.bash
```

Workspace Folder Structure

A typical workspace folder structure might look like this:

```
workspace_folder/        -- WORKSPACE
  src/                   -- SOURCE SPACE
    CMakeLists.txt       -- 'Toplevel' CMake file, provided by catkin
    package_1/
      CMakeLists.txt     -- CMakeLists.txt file for package_1
      package.xml        -- Package manifest for package_1
    ...
    package_n/
      CMakeLists.txt     -- CMakeLists.txt file for package_n
      package.xml        -- Package manifest for package_n
```

ROS Package

- A ROS package is a project. It might contain ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module.
- A ROS package is resided in the src folder of the ROS workspace.
- For a ROS package to be considered a catkin package it must meet a few requirements:
  - The package must contain a catkin compliant package.xml file. The package must contain a CMakeLists.txt which uses catkin.
  - Each package must have its own folder

Steps to Create a ROS Package

- Change to the source space directory of the catkin workspace:
  $ cd ~/catkin_ws/src

- Use the catkin_create_pkg script to create a new package
  catkin_create_pkg <package_name> [depend1], [depend2],....
  $ catkin_create_pkg demo1 std_msgs rospy roscpp

- Build the packages in the catkin workspace
  $ cd ~/catkin_ws
  $ catkin_make

- Add the workspace to your ROS environment by sourcing the generated setup file:
  $. ~/catkin_ws/devel/setup.bash

- catkin_make creates two files - CMakeList.txt and package.xml and two folders include and src in the package folder

Activity: Create a ROS Package

Create a ROS package 'my_robotics' as follows:

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg my_robotics std_msgs rospy roscpp
$ cd ~/catkin_ws
$ catkin_make
$ .  ~/catkin_ws/devel/setup.bash
```

OR

```
$ cs
$ catkin_create_pkg my_robotics std_msgs rospy roscpp
$ cm
$ sb
```

ROS Packages File Structure

ROS packages tend to follow a common structure:

- **src:** Source files (C++)
- **scripts:** Script files (Python)
- **msg:** Folder containing Message (msg) types
- **srv:** Folder containing Service (srv) types
- **CMakeLists.txt:** CMake build file
- **package.xml:** Package catkin/package.xml
- **include:** C++ include headers

Install a ROS Package

- You can enhance the capabilities of ROS by installing packages
- These packages are either downloadable debian packages or github repo
- To install debian packages, use apt-get install, Eg

  sudo apt install <package-name>
- To install github repo, cd to src folder, do a git clone, follow by catkin build and source setup.bash

Activity: Install Github ROS Package

Try to install a github repo for ROS package as follows:

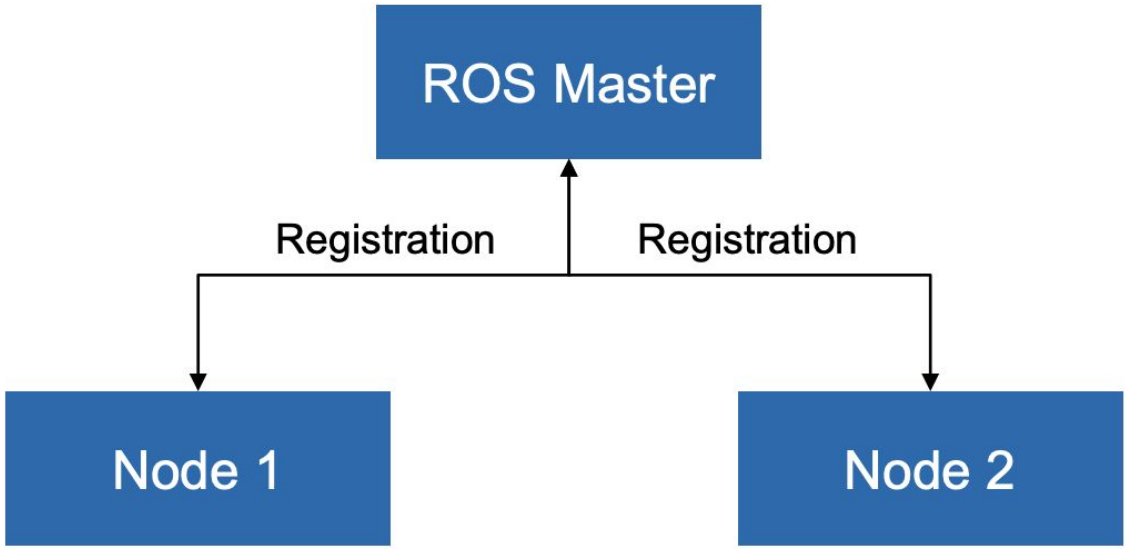
$ cd ~/<your own ws>/src
$ git clone https://github.com/tertiarycourses/beginner_tutorials
$ cd ~/<your own ws>
$ catkin_make
$. devel/setup.bash

OR

$ cs
$ git clone https://github.com/tertiarycourses/beginner_tutorials
$ cm
$ sb

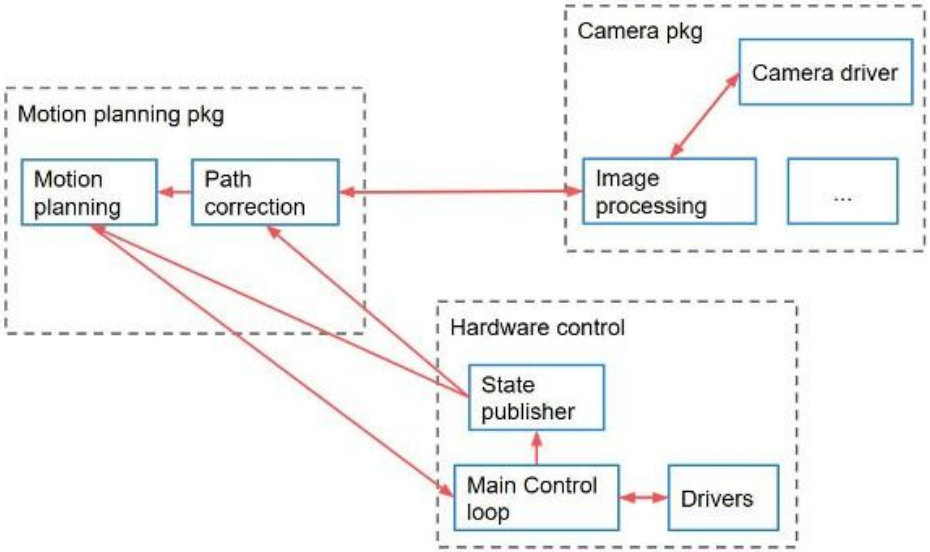
ROS Nodes

- The basis of ROS communication is that multiple executables called nodes are running in an environment and communicating with each other in various ways.
- A node is a single purpose executable program
- These nodes exist within a structure called a package



ROS Node Case Study

- Take an example of a standard robotics application which involves a mobile robot and a camera. The robot has 3 ROS packages :
  - Hardware control package: directly controls the hardware (wheels and other actuators)
  - Motion planning package: monitors and controls the robot trajectory
  - Camera package: processes images and give useful info and commands to the robot
- Each package would have multiple nodes for some processes such as image process, path correction.
- Nodes are combined into a graph and communicate with each other using ROS topics, services, actions, etc.



Start a Node

- To start a node, type the following command
  rosrun <package_name> <node_name>
  Eg: **rosrun turtlesim turtlesim_node**

Node Information

- To list all the active nodes,run
  rosnode list
- To retrieve information about a node with
  rosnode info <node_name>
  Eg: **rosnode info /turtlesim**

Activity: ROS Node

Run the following commands

(terminal 1) $ roscore

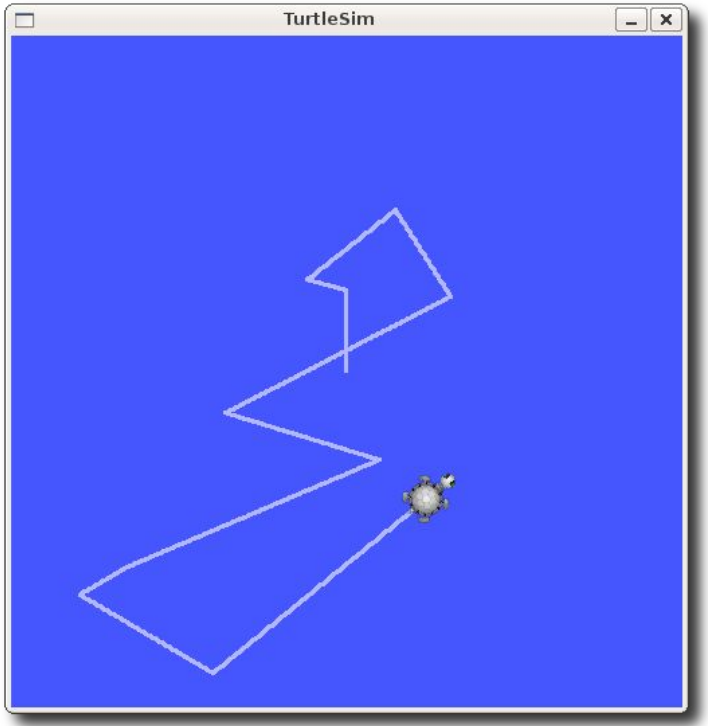
(terminal 2) $ rosrun turtlesim turtlesim_node

(terminal 3) $ rosrun turtlesim turtle_teleop_key

Use the arrow keys to move the turtle

(terminal 4) $ rosnode list

(terminal 4) $ rosnode info /teleop_turtle



ROS Launch

- ROS launch is a tool for launching multiple nodes (as well as setting parameters)
- Launch files are written in XML as *.launch files
- To launch the ROS nodes

  roslaunch <package_name> <launch file name>

Activity: ROS Launch

(terminal 1)
$ cd ~/catkin_ws/src/
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git
$ cd ~/catkin_ws && catkin_make
$. devel/setup.bash

(terminal 1)
$ ~~roscore~~

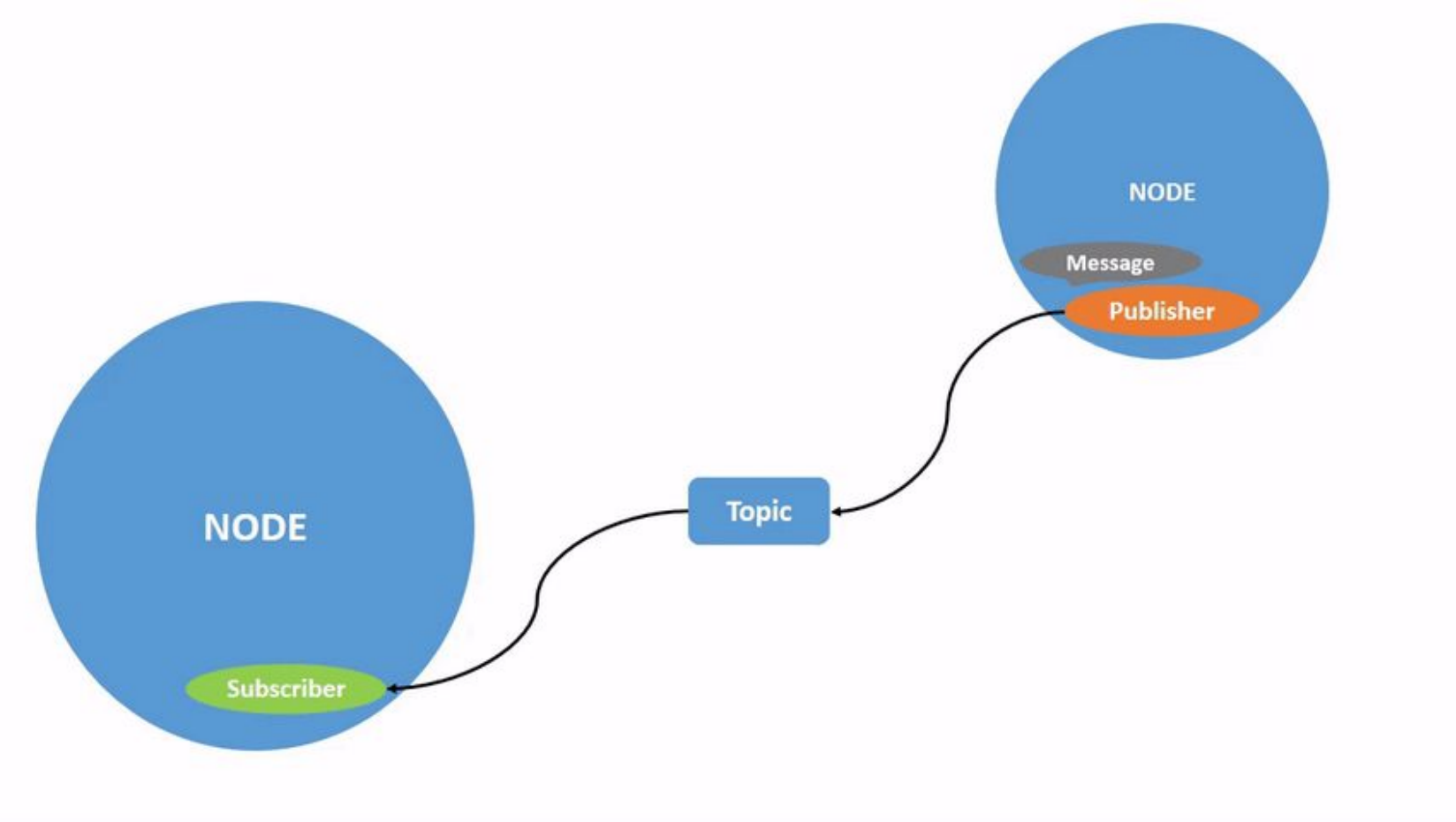
(terminal 2)
~~$ nano ~/.bashrc~~
~~$ export TURTLEBOT3_MODEL=burger~~
~~$ source ~/.bashrc~~
$ roslaunch turtlebot3_fake turtlebot3_fake.launch

(terminal 3)
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch

Topic 3
ROS Topics, Services and Actions

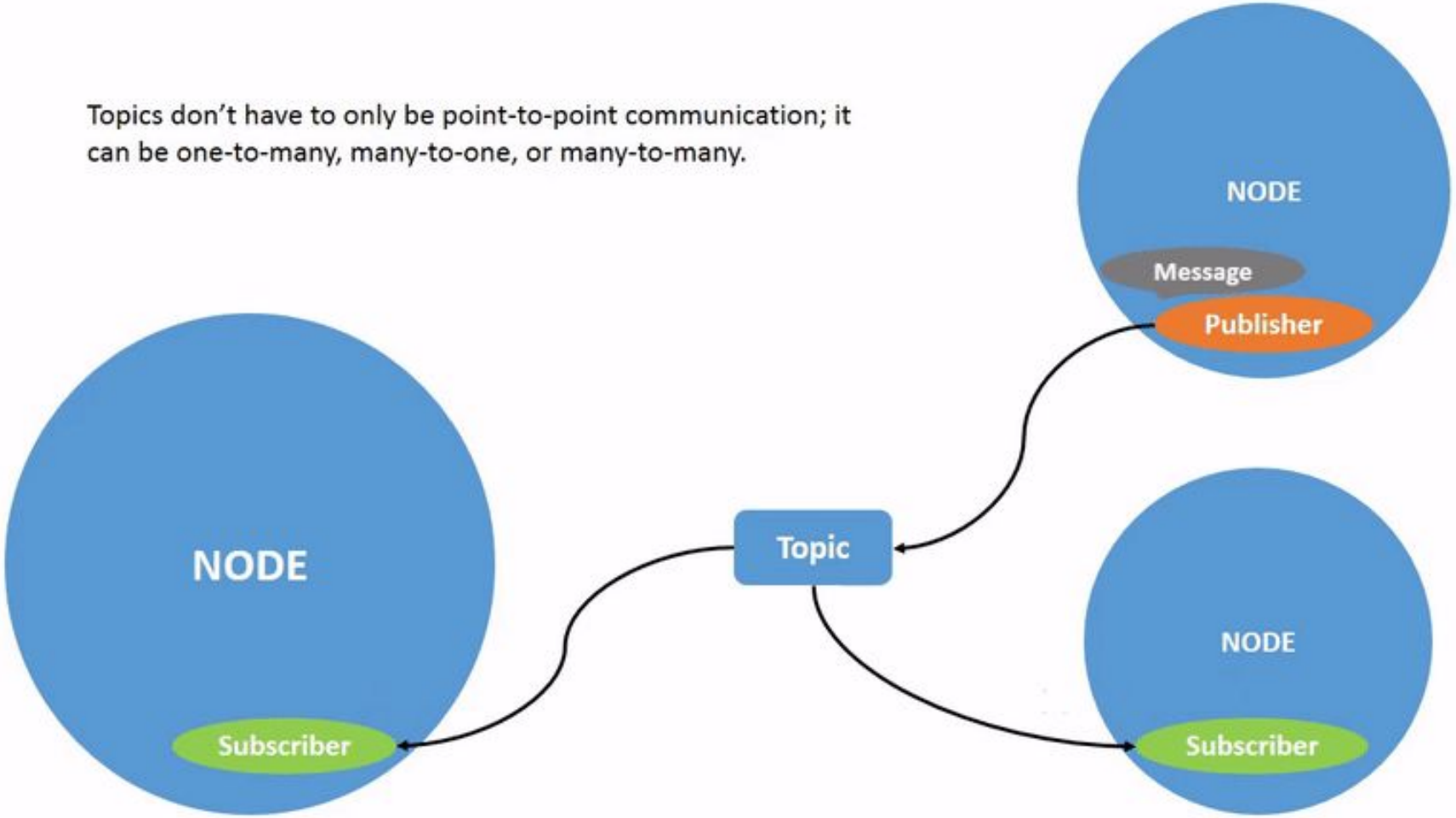
ROS Topics

- A ROS Topic is a channel for communicating messages among the nodes
- A ROS Topic has a fixed Message type
- A Node publishes on a Topic to broadcast Messages
- A Node subscribes to a Topic to get Messages from other Nodes

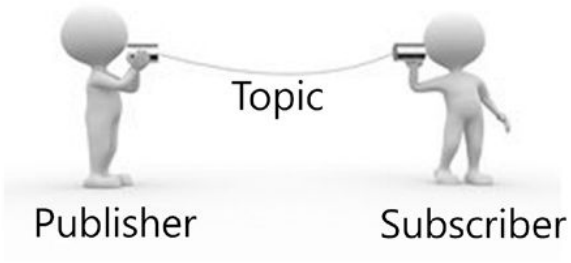


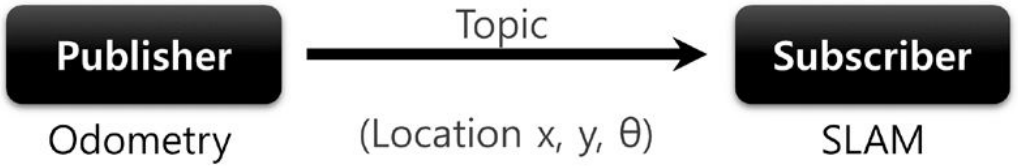
Topics Many-to-Many Communication

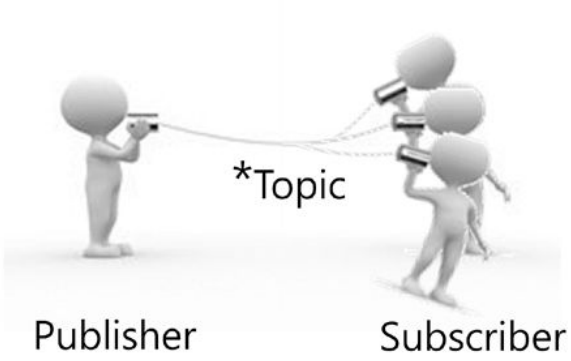
Topics don't have to only be point-to-point communication; it can be one-to-many, many-to-one, or many-to-many.

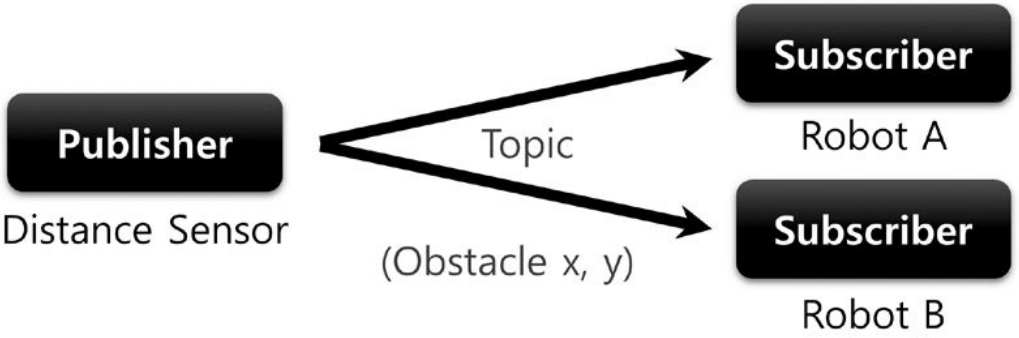


Topic Message Communication









*Topic not only allows 1:1 Publisher and Subscriber communication, but also supports 1:N, N:1 and N:N depending on the purpose.

Topics information

- To list all the active topics, run
  rostopic list
- To show the information on a topic
  rostopic info <topic_name>
- To see the data being published on a topic, use:
- **rostopic echo <topic_name>**

Eg
rostopic info /turtle1/cmd_vel

Activity: ROS Topic

Run the following commands
(terminal 1) $ roscore
(terminal 2) $ rosrun turtlesim turtlesim_node
(terminal 3) $ rosrun turtlesim turtle_teleop_key
Use the arrow keys to move the turtle
(terminal 4)
$ rostopic list
$ rostopic info /turtle1/cmd_vel
$ rostopic echo /turtle1/cmd_vel

Visualize Nodes and Topics

- To visualize the nodes and topics relationship, you can use the package rqt_graph
  rosrun rqt_graph rqt_graph

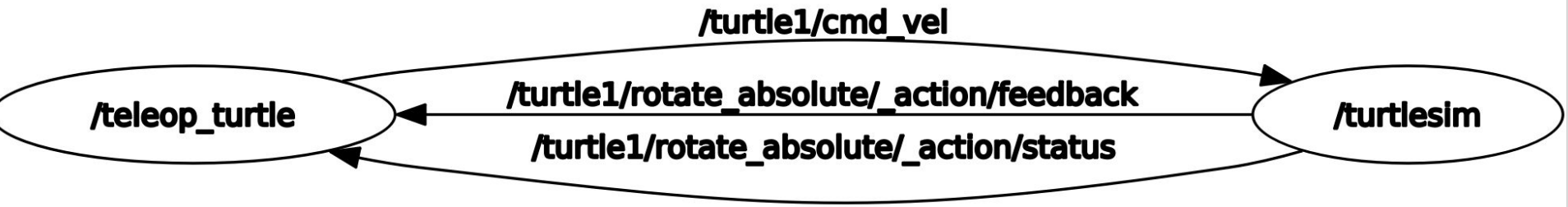
As you can see, the turtlesim_node and the turtle_teleop_key nodes are communicating on the topic named /turtle1/command_velocity.

RQt

- RQt is a graphical user interface framework that implements various tools and interfaces in the form of plugins.
- One can run all the existing GUI tools as dockable windows within RQt! The tools can still run in a traditional standalone method, but RQt makes it easier to manage all the various windows in a single screen layout.
- You can run any RQt tools/plugins easily by:
  rqt

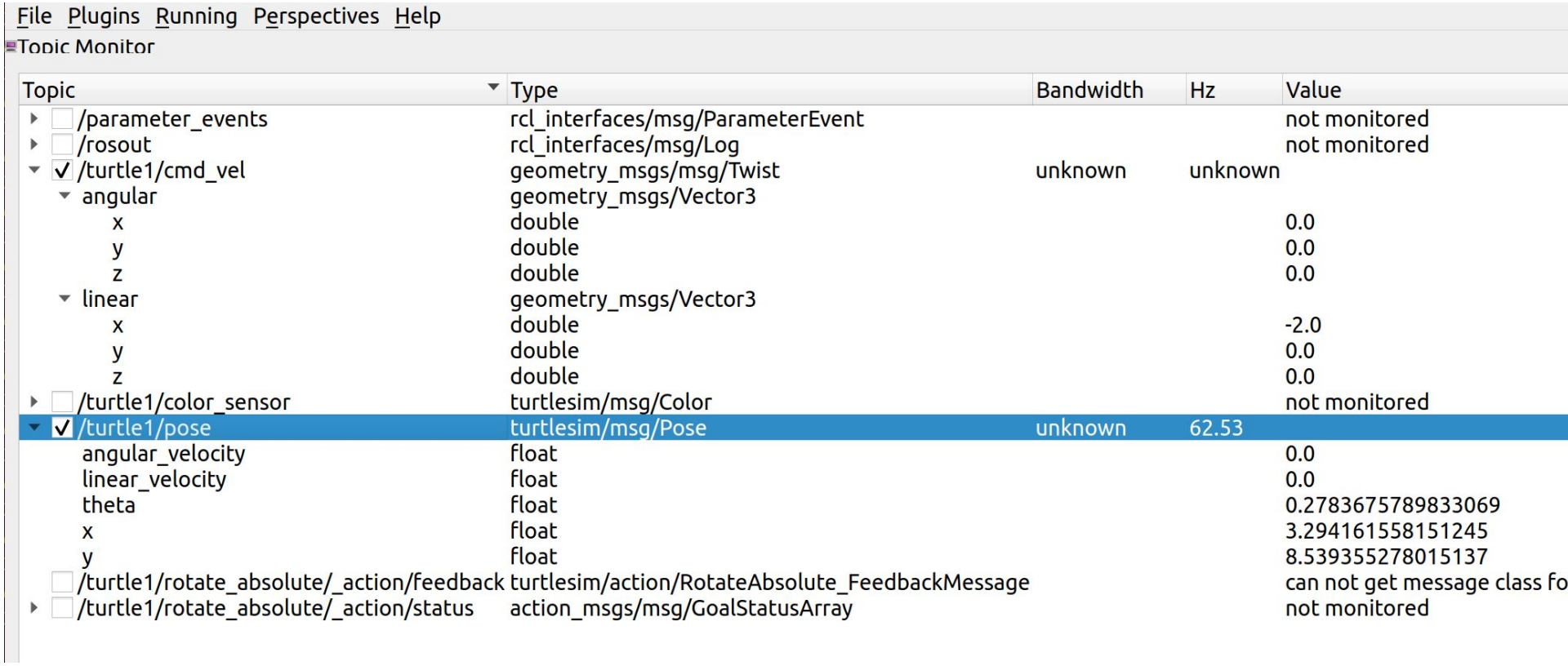
Visualize Nodes and Topics

- To visualize the nodes and topics relationship, you can run rqt and selecting Plugins > Introspection > Nodes Graph.
- As you can see, the turtlesim_node and the turtle_teleop_key nodes are communicating on the topic named /turtle1/cmd_vel



Activity: Monitor Topic with RQt

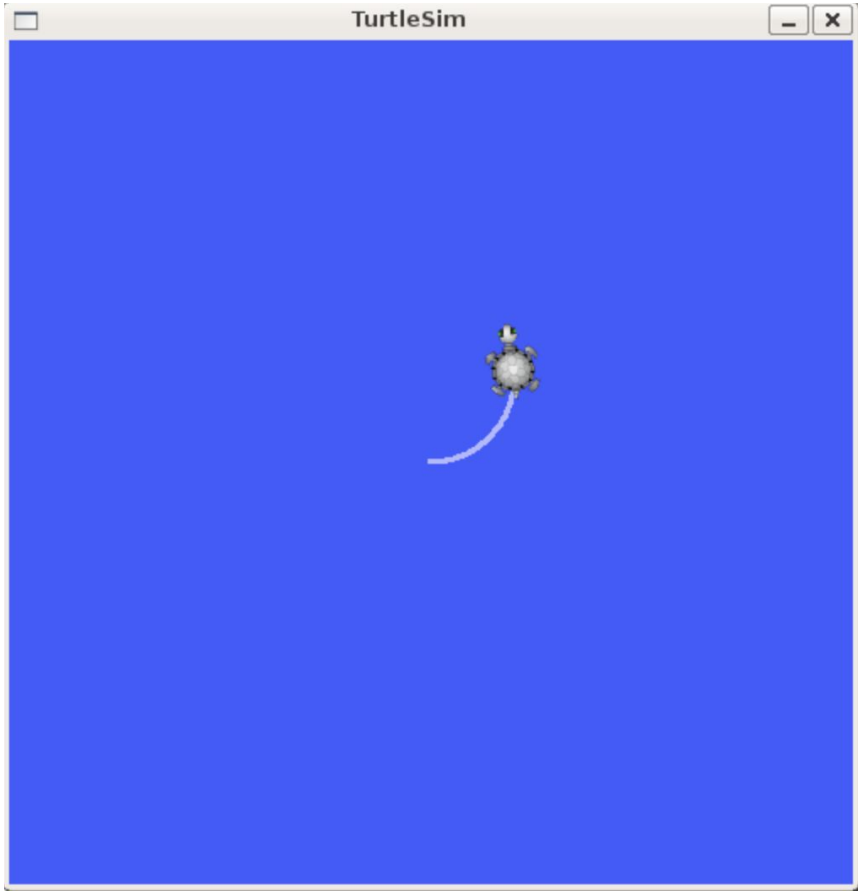
- You can open rqt and select Plugins > Topics > Topics Monitor to monitor the topics information.
- Move the keys to see how the value changes.



Publish ROS Message Data

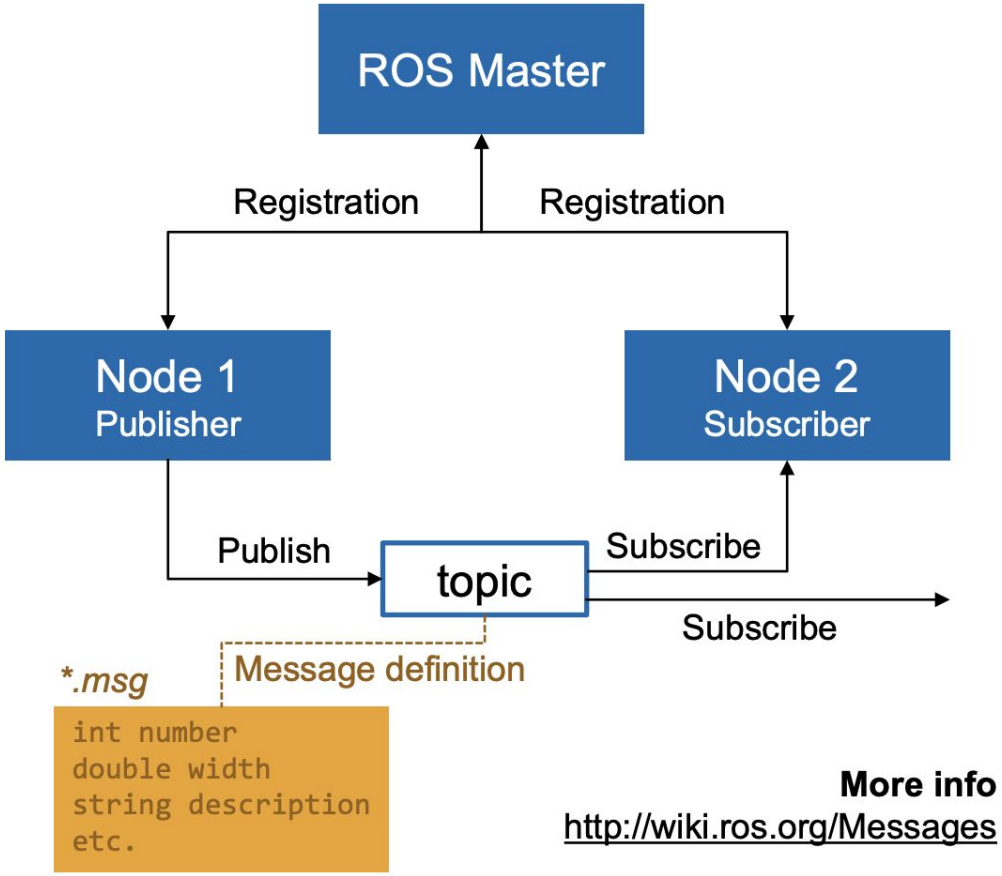
rostopic pub publishes data on to a topic currently advertised. Eg

rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'



ROS Messages

- Communication on topics happens by sending ROS messages between nodes.
- A ROS message is a data type for communication among the nodes



ROS Message Types

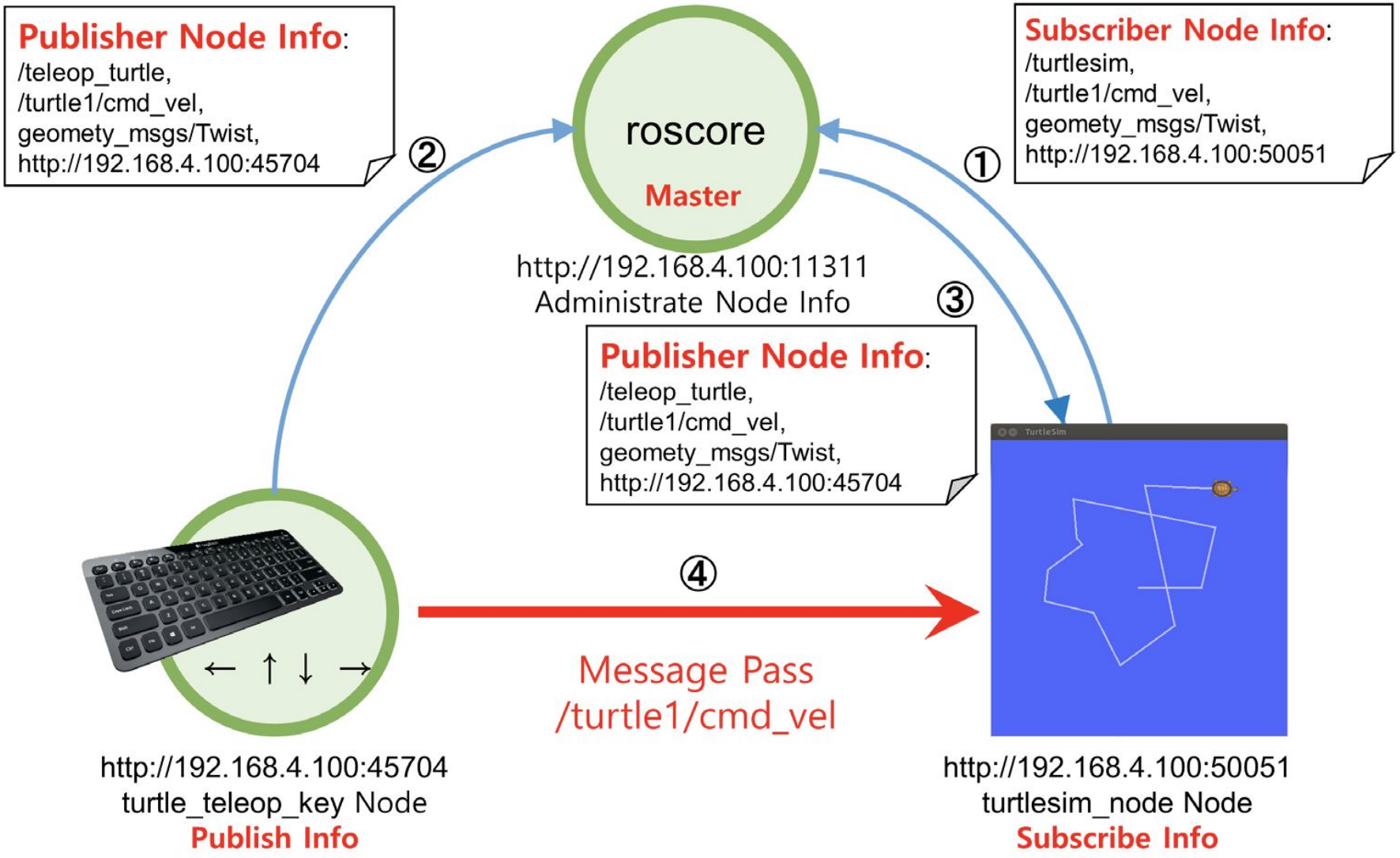
- ROS built in default message types
  - std_msgs - basic data types, such as Int , Float, Char, String, Bool, Byte, Array
  - geometry_msgs - geometry related data types, such as
    - Pose-related: Point, PointStamped, Quaternion, QuaternionStamped, Pose PostStamped, PosewithCovariance, PosewithCovarianceStamped, PoseArray
    - Control-related: Twist, Vector3, Transform
  - sensor_msgs - sensory data types, such as Image, PointCloud, BatteryState, LaserScan
- Customized message type
  - Developer can define a new message type using the existing message types

Show Message Types

- To show information of a message
  rosmsg show <type>
- Eg **rosmsg show geometry_msgs/Twist**
  Its show the following info:

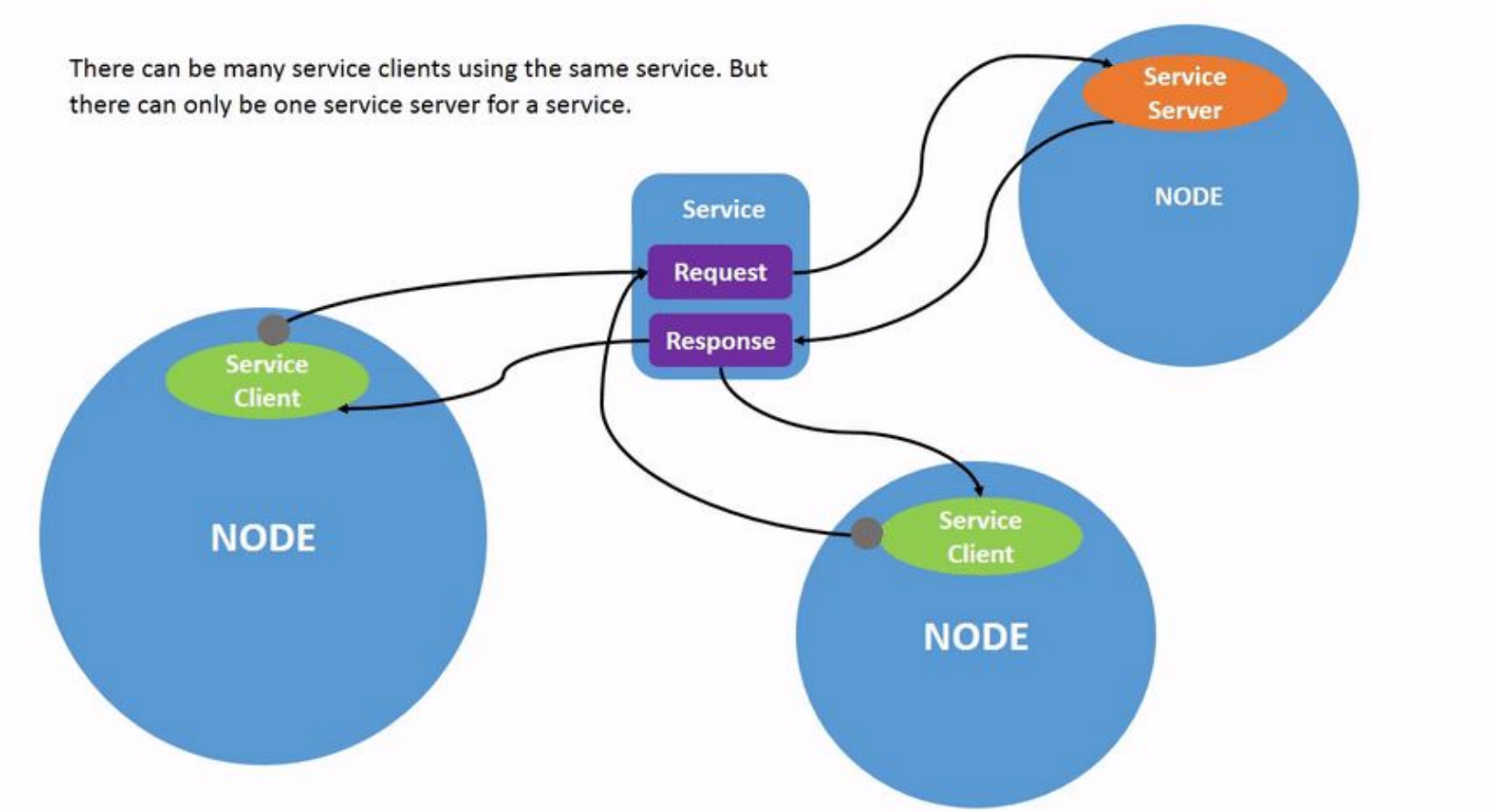
```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Turtlesim Message Communication

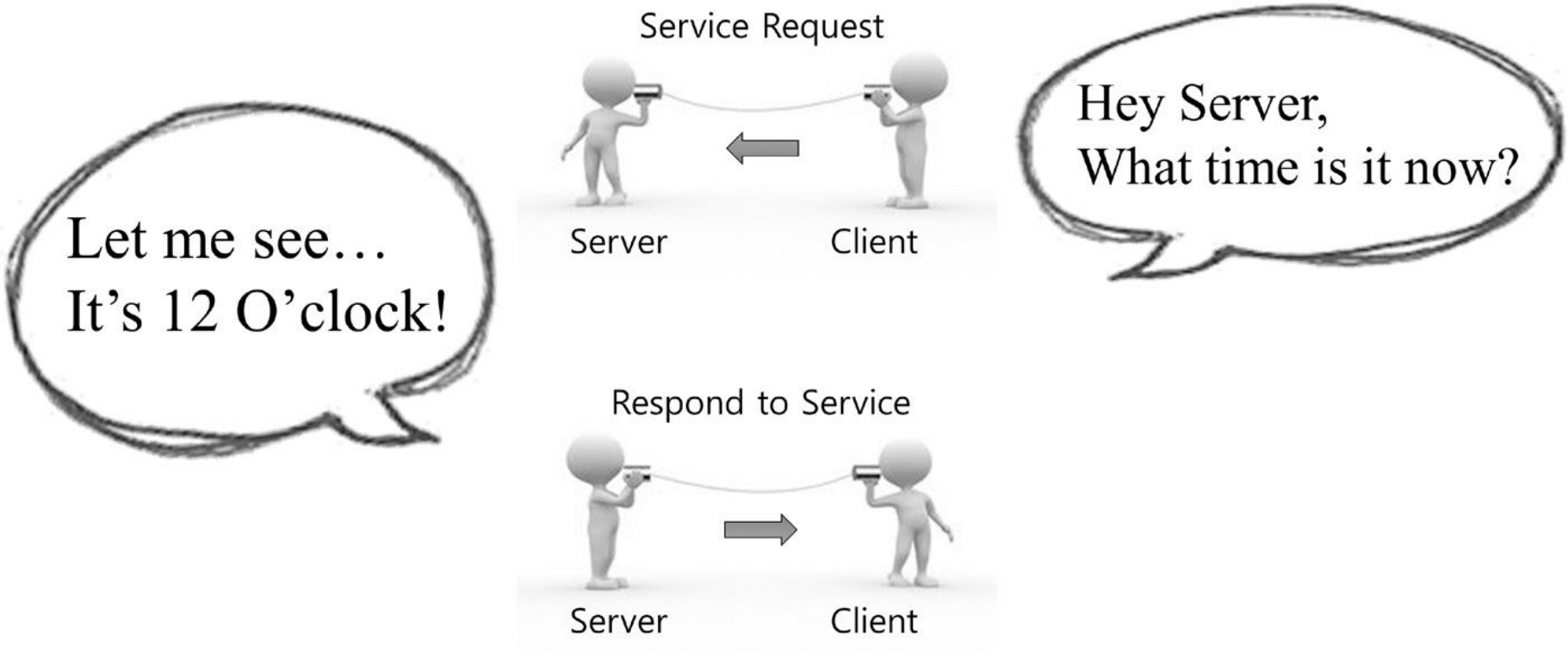


ROS Services

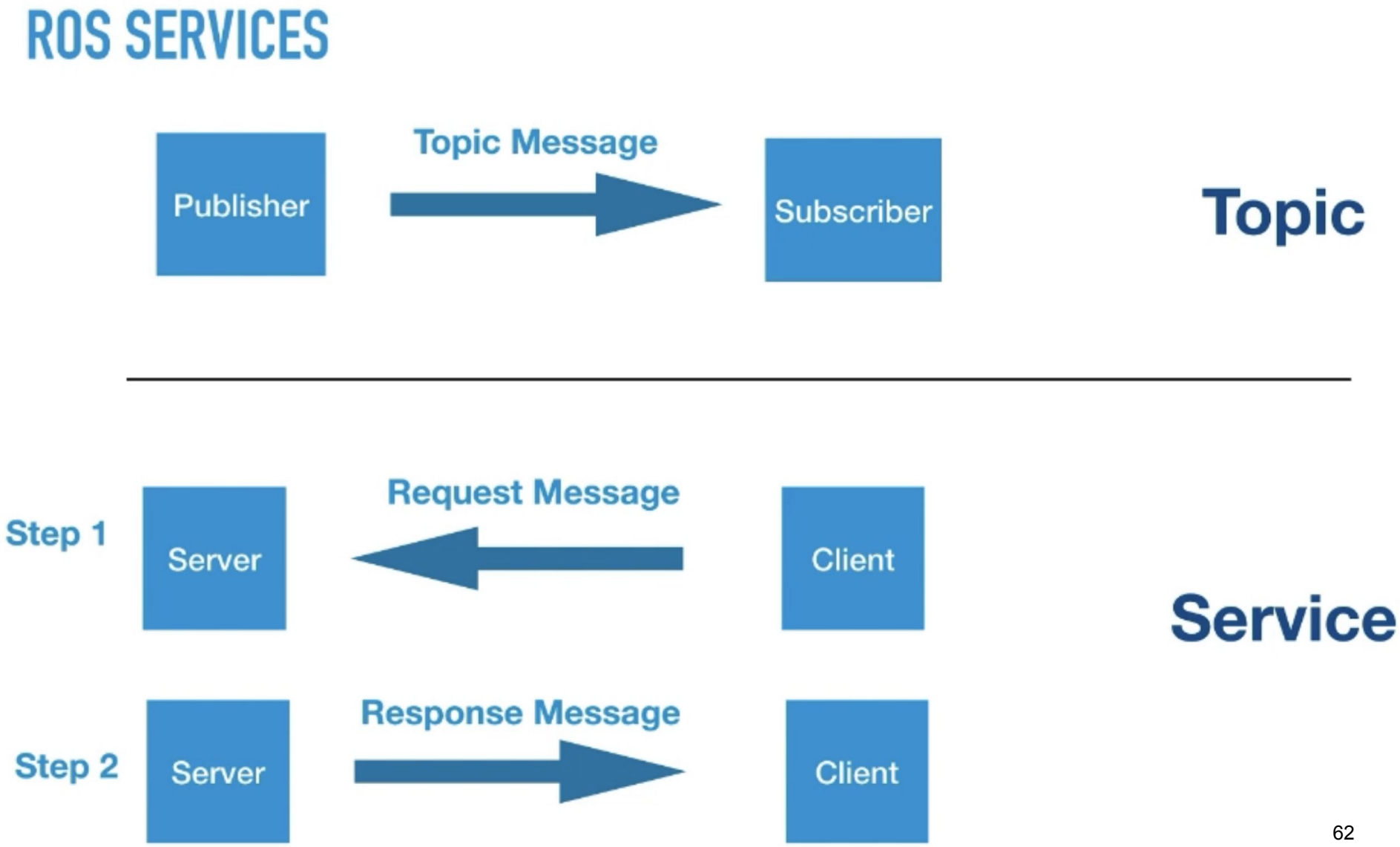
- What is ROS Service?
  - Unlike topic, service is a one time communication
  - A client sends a request, the server sends back a response.
- When to use ROS Service?
  - Request the robot to perform a certain task eg pick up a cup, move from point A to B



Service Message Communication

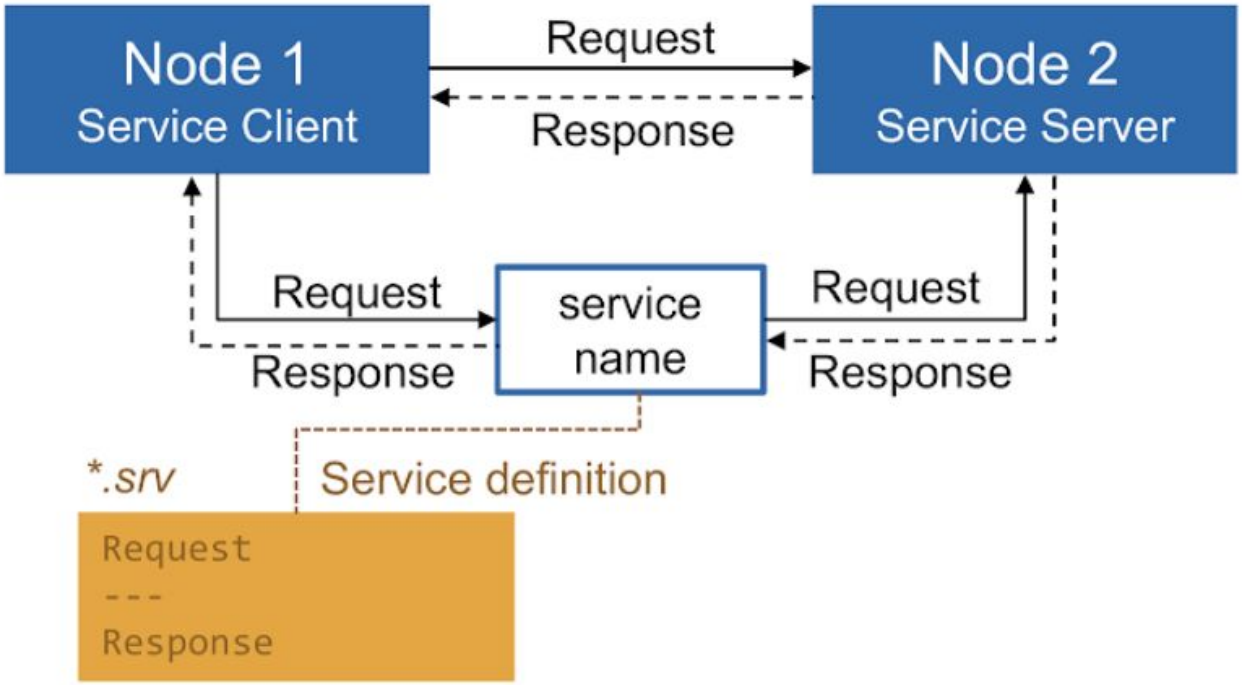


ROS Topic vs Service



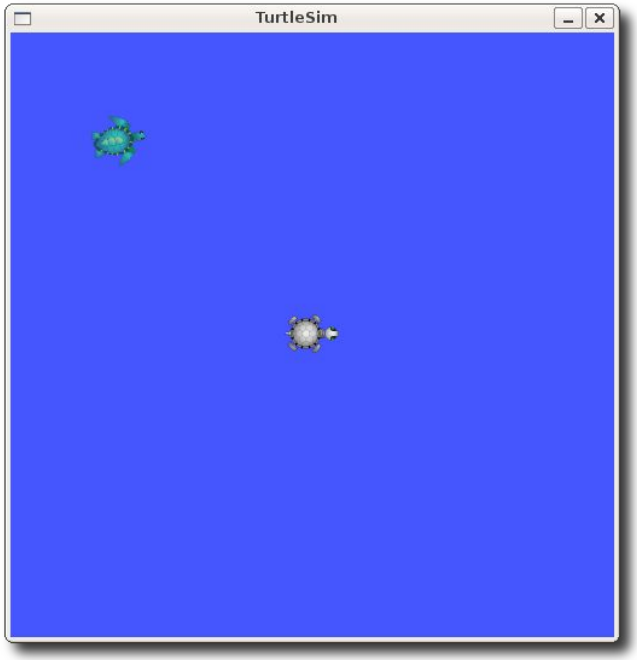
ROS Service Command

- List all available ROS services
  rosservice list
- Show the type of ROS service
  rosservice type /service_name
- Call a service with request
  rosservice call /service_name args



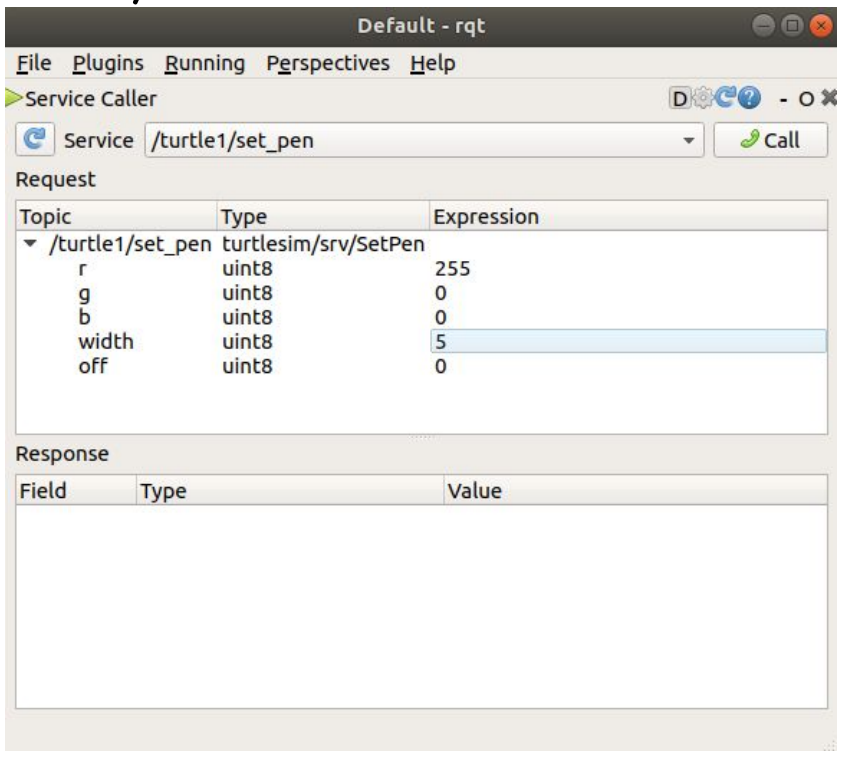
Activity: ROS Service

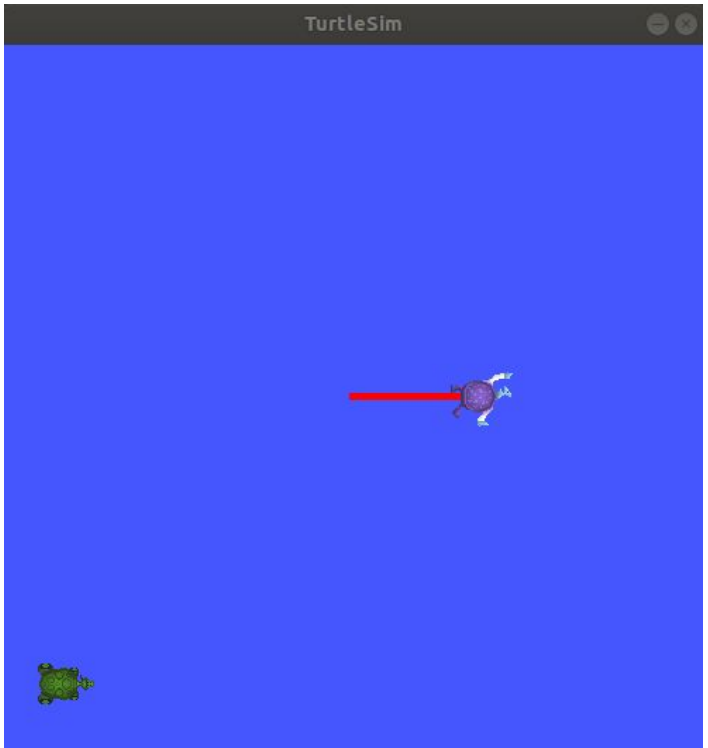
Run the following commands
(terminal 1) $ roscore
(terminal 2) $ rosrun turtlesim turtlesim_node
(terminal 3)
$ rosservice list
$ rosservice call /spawn 2 2 180 t2
$ rosservice call /spawn 8 8 0 t3



Call ROS Services from RQt

- Select Plugins > Services > Service Caller from the menu bar at the top.
- Use rqt to call the /spawn service.
- Give turtle1 a unique pen using the /set_pen service
- To have turtle1 draw with a distinct red line, change the value of r to 255, and the value of width to 50 and y = 1.0





ROS Parameters

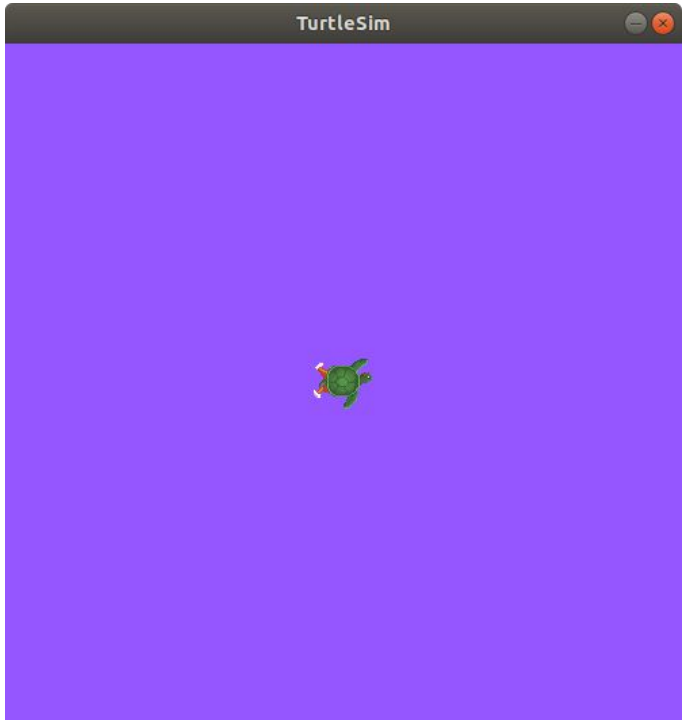
- A parameter server is a shared, multi-variate dictionary that is accessible via network APIs.
- Nodes use this server to store and retrieve parameters at runtime.
- As it is not designed for high-performance, it is best used for static, non-binary data such as configuration parameters.
- It is meant to be 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, which means that its API is accessible via normal XMLRPC libraries
-

ROS Parameter Commands

- To see the parameters belonging to your nodes, open a new terminal and enter the command:
  rosparam list
- To get the current value of a parameter, use the command:
  rosparam get <node_name> <parameter_name>
- To change a parameter's value at runtime, use the command:
  rosparam set <node_name> <parameter_name> <value>

Activity: ROS Parameters

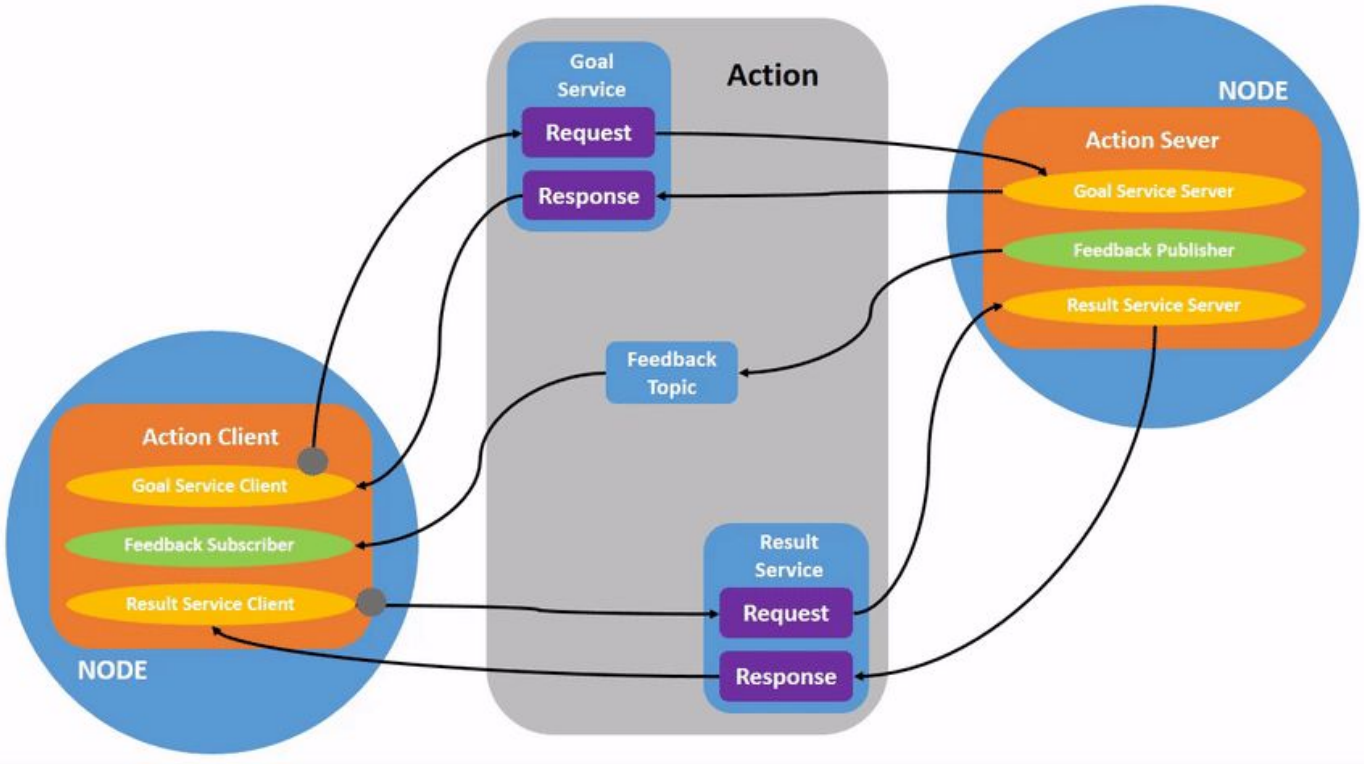
Run the following commands:
(terminal 1) $ roscore
(terminal 2) $ rosrun turtlesim turtlesim_node
(terminal 3)
$ rosparam list
$ rosparam get background_g
$ rosparam set /background_r 150
$ rosservice call /clear



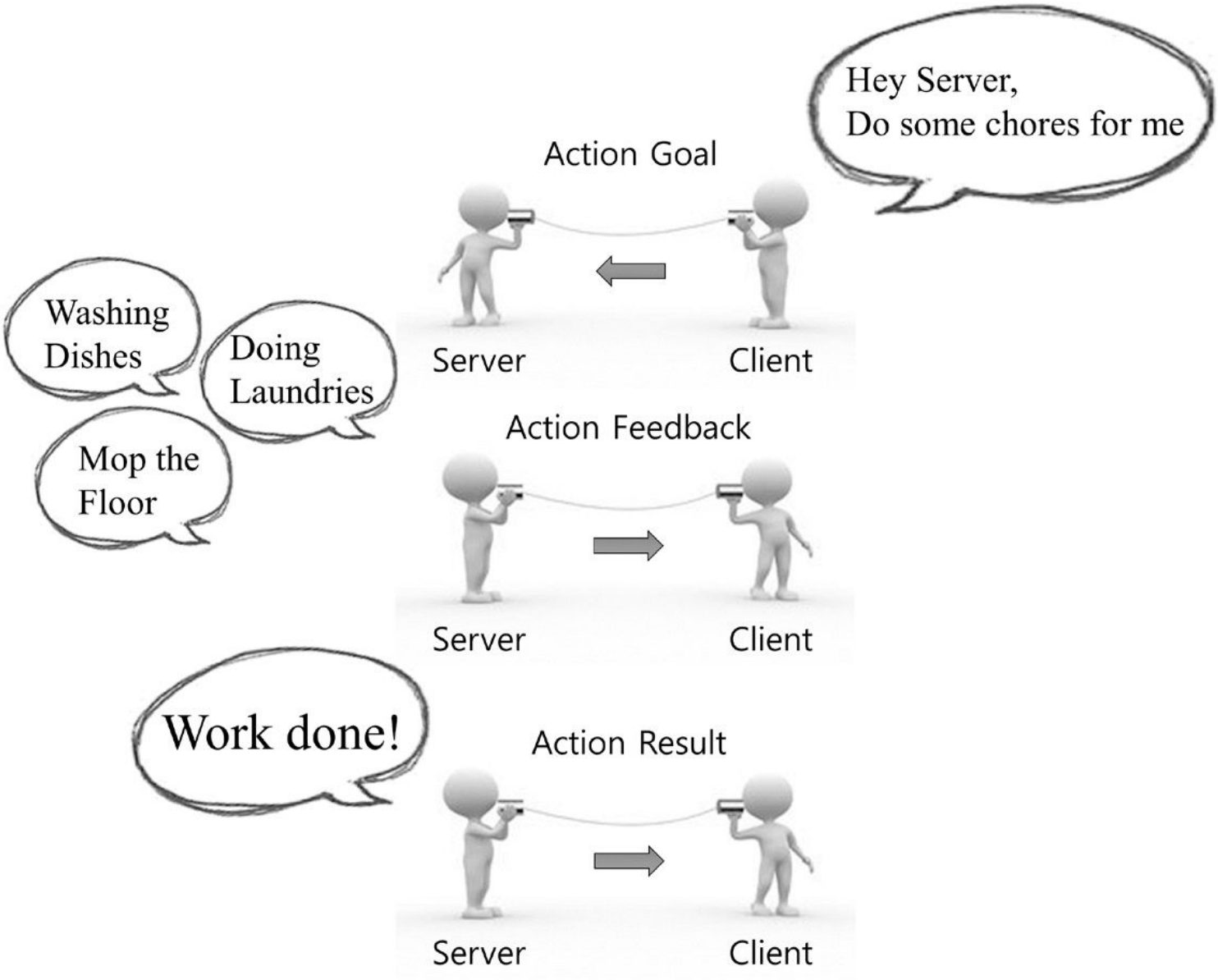
If not working, try
$ rosparam set /turtlesim/background_r 150
$ rosservice call /clear

ROS Actions

- Actions are built on topics and services. Their functionality is similar to services, except actions are preemptable (you can cancel them while executing). They also provide steady feedback, as opposed to services which return a single response.
- Actions use a client-server model, similar to the publisher-subscriber model (described in the topics tutorial). The “action client” node sends a goal to an “action server” node that acknowledges the goal and returns a stream of feedback and a result



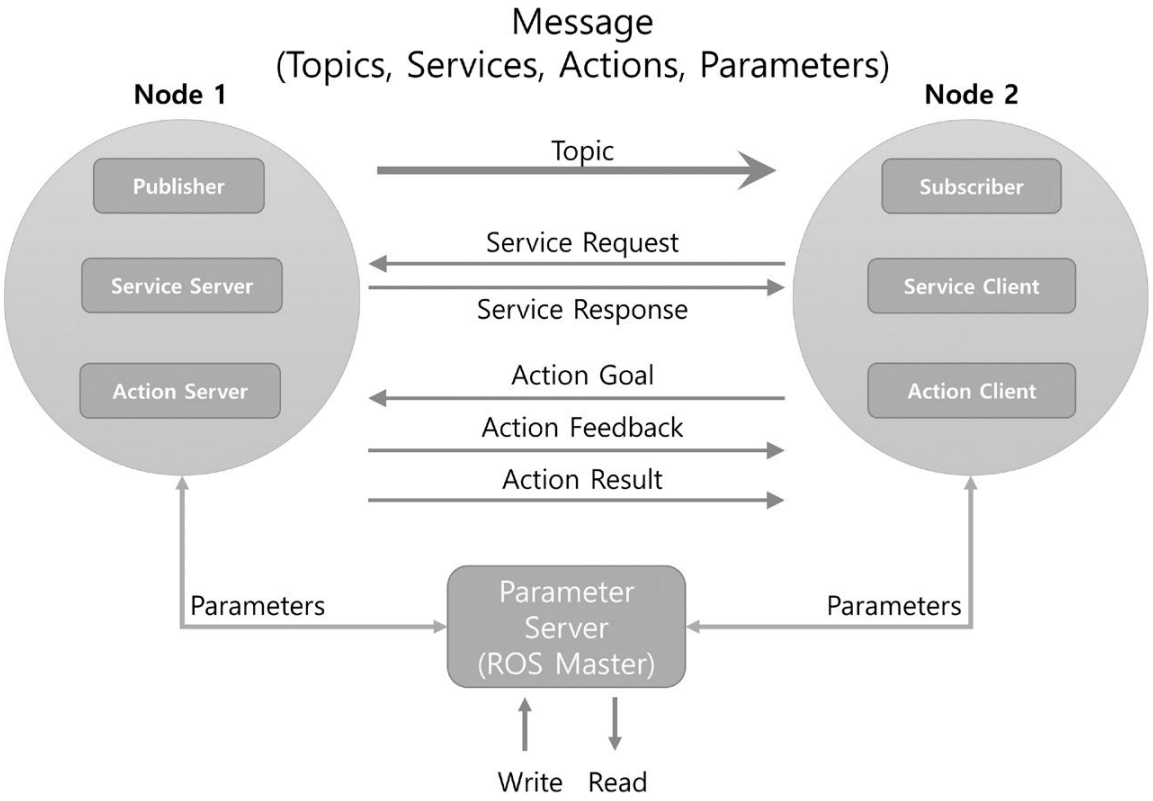
Action Message Communication



ROS Action Commands

- To identify all the actions in the ROS graph, run the commands:
  rosaction list
- To send an action goal from the command line with the following syntax:
  rosaction send_goal <action_name> <action_type> <values>

ROS Communication Summary



Type	Features		Description
Topic	Asynchronous	Unidirectional	Used when exchanging data continuously
Service	Synchronous	Bi-directional	Used when request processing requests and responds current states
Action	Asynchronous	Bi-directional	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

Topic 4
ROS Bags

ROSBag

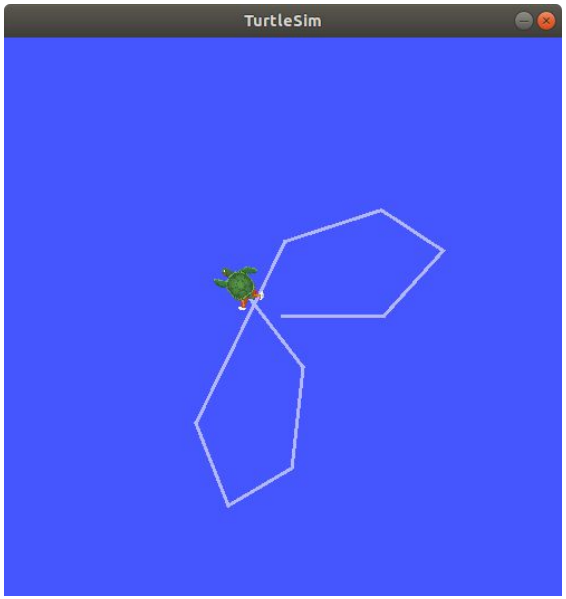
- A rosbag or bag is a file format in ROS for storing message data.
- These bags are often created by subscribing to one or more ROS topics, and storing the received message data in an efficient file structure
- ROS bags is suited for logging and recording dataset for later visualization and analysis

ROS 2 Bag Commands

- To record the data published to a topic:
  rosbag record <topic_name>
- To record and output a specified bag name
  rosbag record <topic_name> -o <bag_file_name>
- To see details about your recording by running:
  rosbag info <bag_file_name>
- To play the recorded data
  rosbag play <bag_file_name>

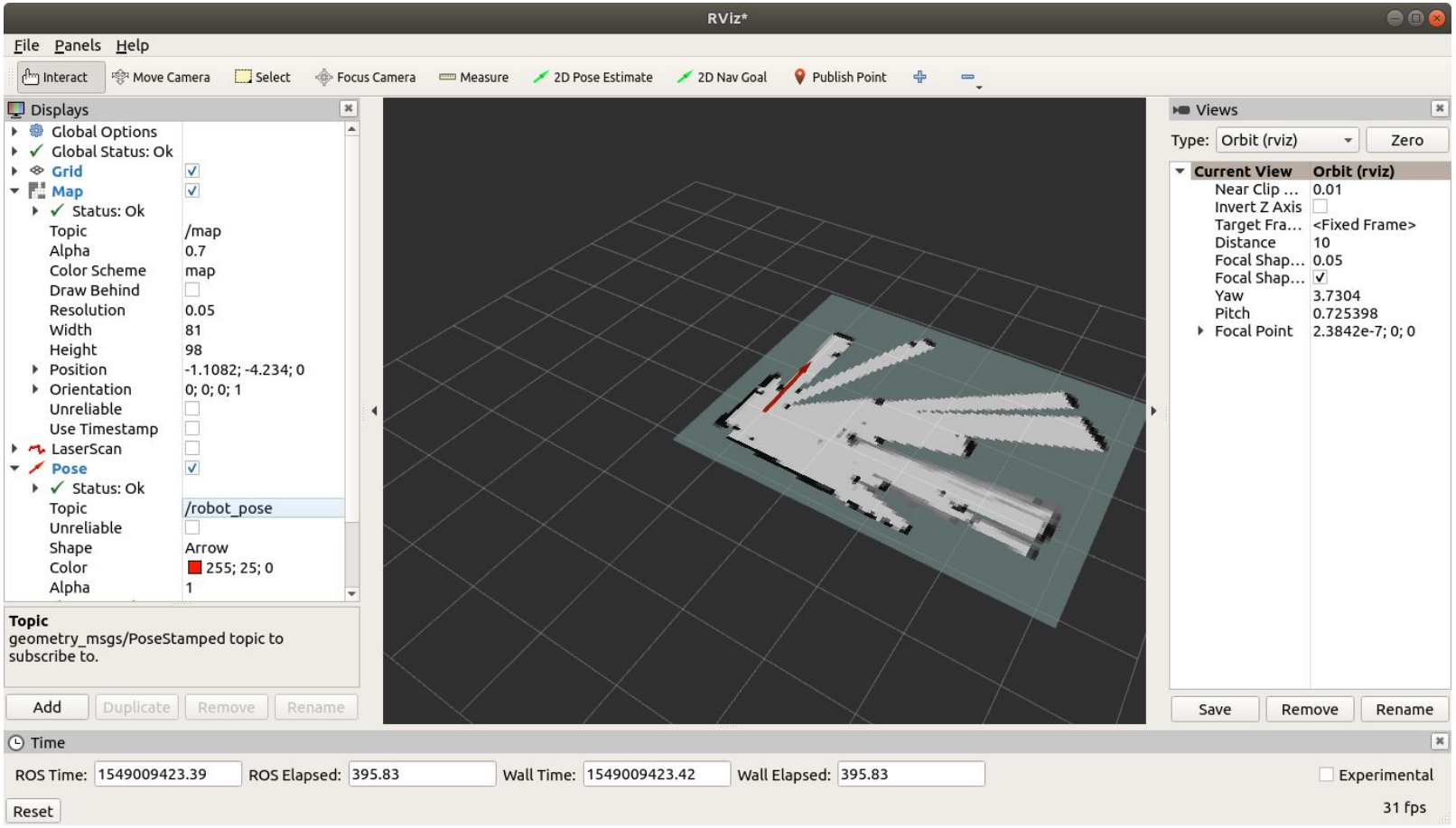
Activity: Record and Play ROS Bag

- (terminal 1) $ roscore
- (terminal 2) $ rosrun turtlesim turtlesim_node
- (terminal 3) $ rosrun turtlesim turtle_teleop_key
- (terminal 4) running rosbag record
- $ mkdir ~/bagfiles
- $ cd ~/bagfiles
- $ rosbag record /turtle1/cmd_vel
- (terminal 3) $ move the turtle with arrow keys
- (terminal 4) $ exit with a Ctrl-C.
- (terminal 4) $ rosbag info <your bagfile>
- (terminal 4) $ rosbag play <your bagfile>

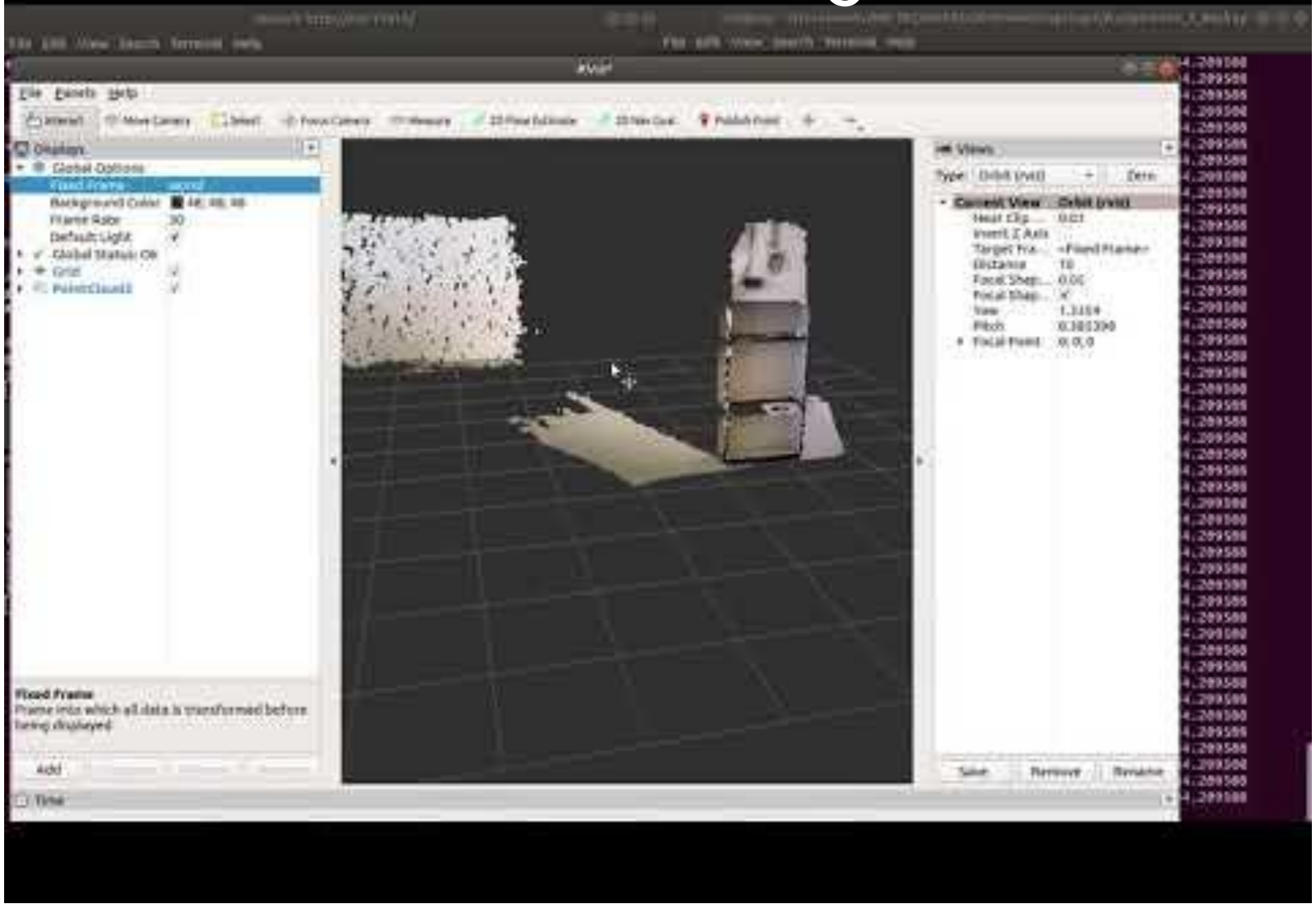


RViz

- RViz (http://wiki.ros.org/rviz) is a 3D visualization tool for ROS
- Subscribes to topics and visualize the message content
- Different camera views
- To run rviz package, type the following command on terminal **rosrun rviz rviz (or rviz)**



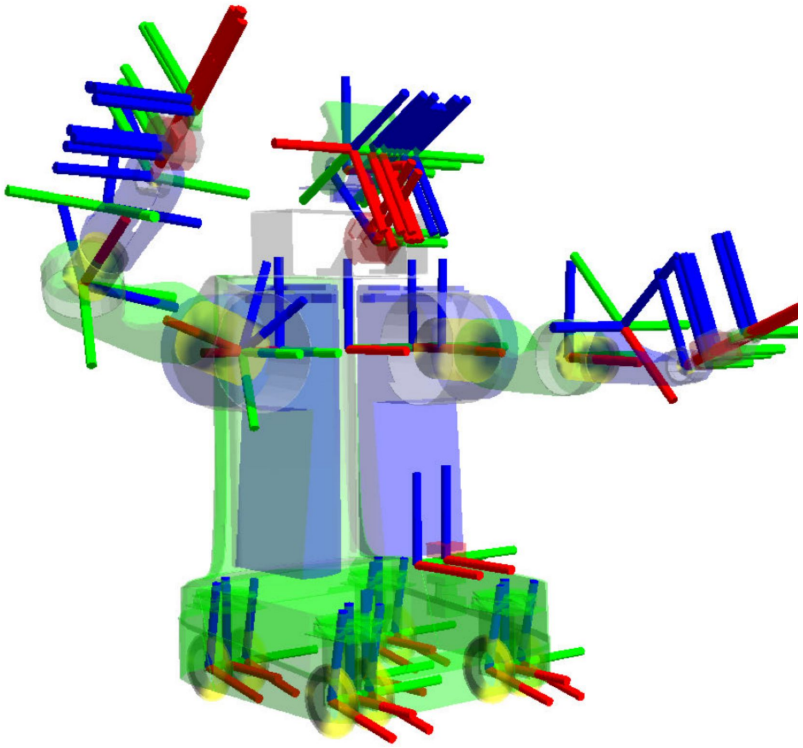
Visual Point Cloud Bags with RViz



Topic 5
TF & URDF

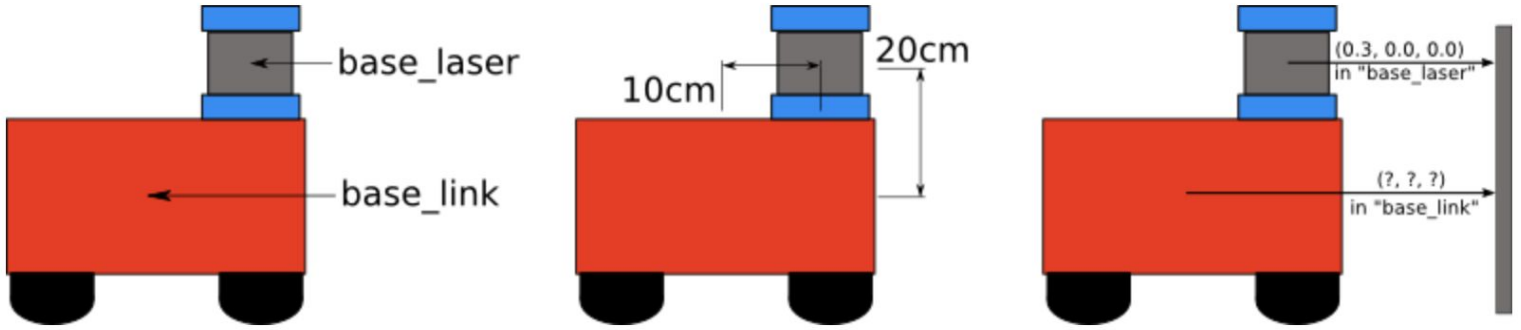
Transformation System (TF)

- A robotic system typically has many 3D coordinate frames that change over time, such as a world frame, base frame, gripper frame, head frame, etc.
- tf (transformation system) keeps track of all these frames over time, and allows you to ask questions like:
  - Where was the head frame relative to the world frame, 5 seconds ago?
  - What is the pose of the object in my gripper relative to my base?
  - What is the current pose of the base frame in the map frame
- tf maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors, etc between any two coordinate frames at any desired point in time.

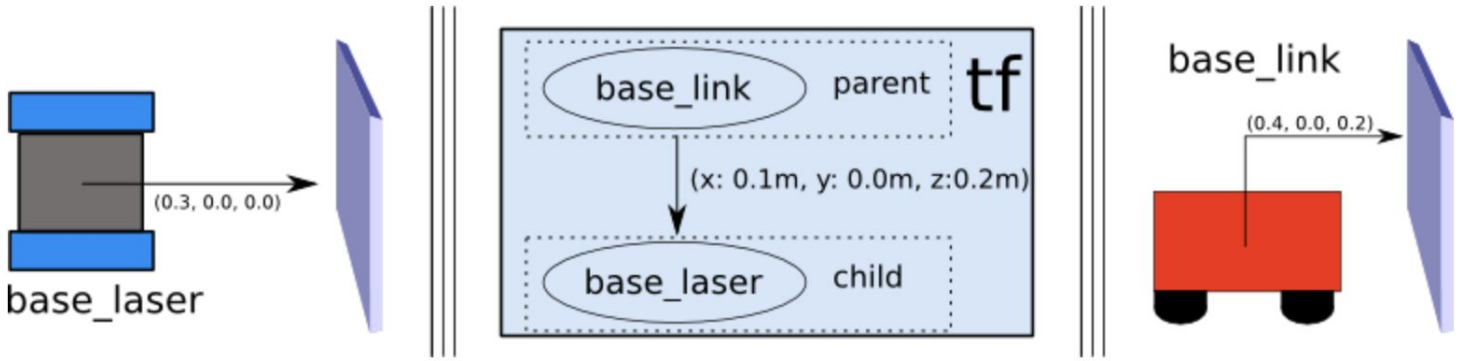


A Simple Robot Example Using TF

- Consider the example of a simple robot (base_link) that has a mobile base (base_laser) with a single laser mounted on top of it.

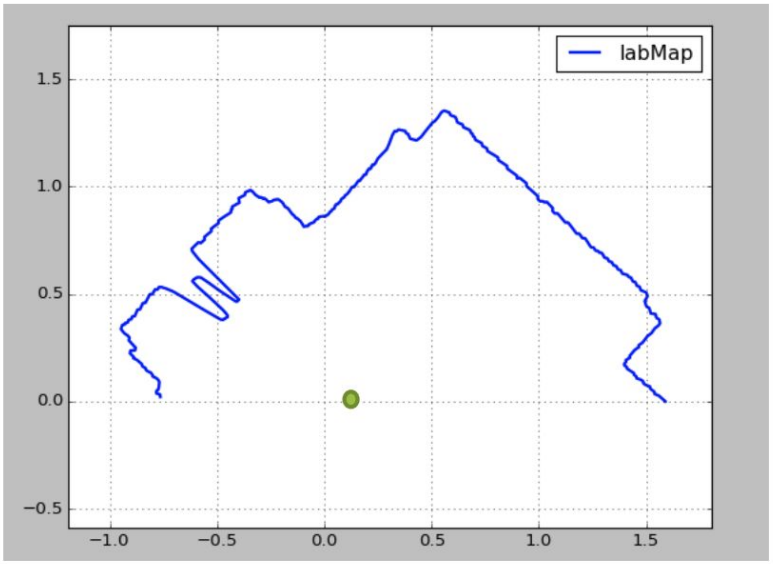


- TF defines and stores the relationship between the "base_link" and "base_laser" frames, and add them to a transform tree.
- To create a transform tree, we create two nodes, one for the "base_link" coordinate frame and one for the "base_laser" coordinate frame.
- To create the edge between them, we need to decide which node will be the parent and which will be the child

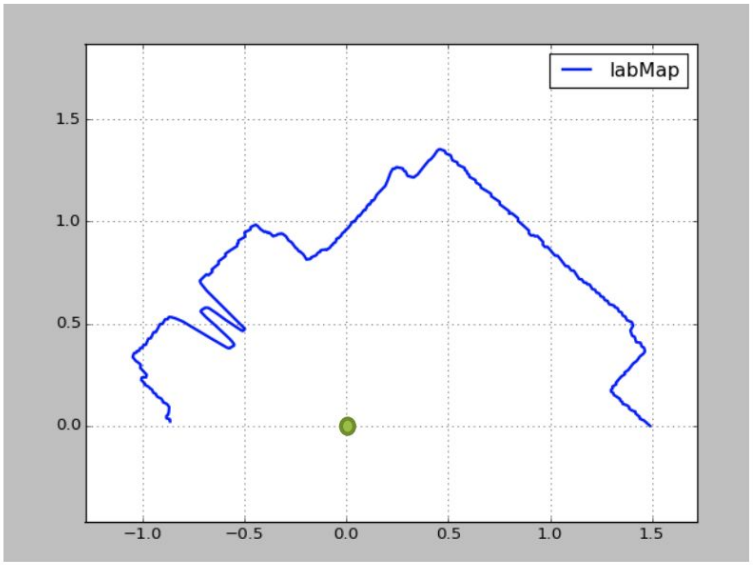


A Simple Robot Example Using TF

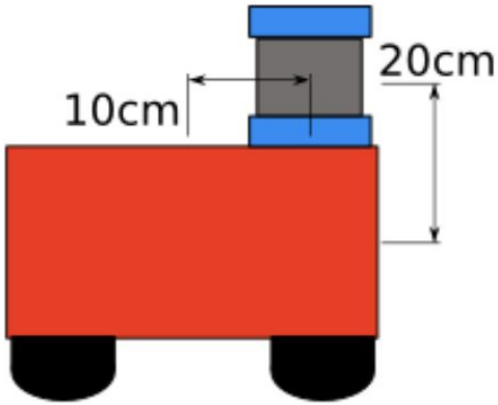
- TF allows one to view the LIDAR maps based on the base_link or base_laser coordinate frames

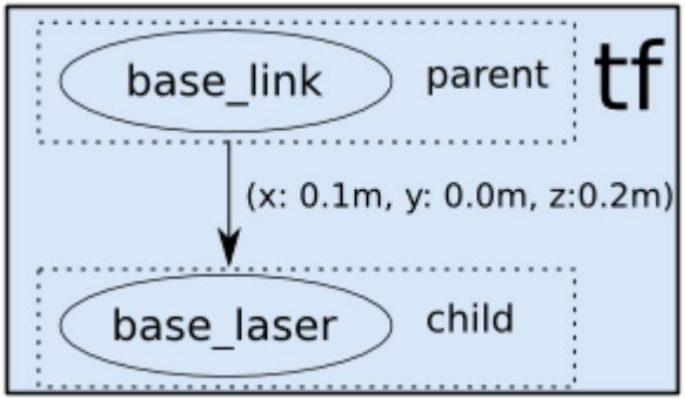


LIDAR reading in base_link coordinate frame



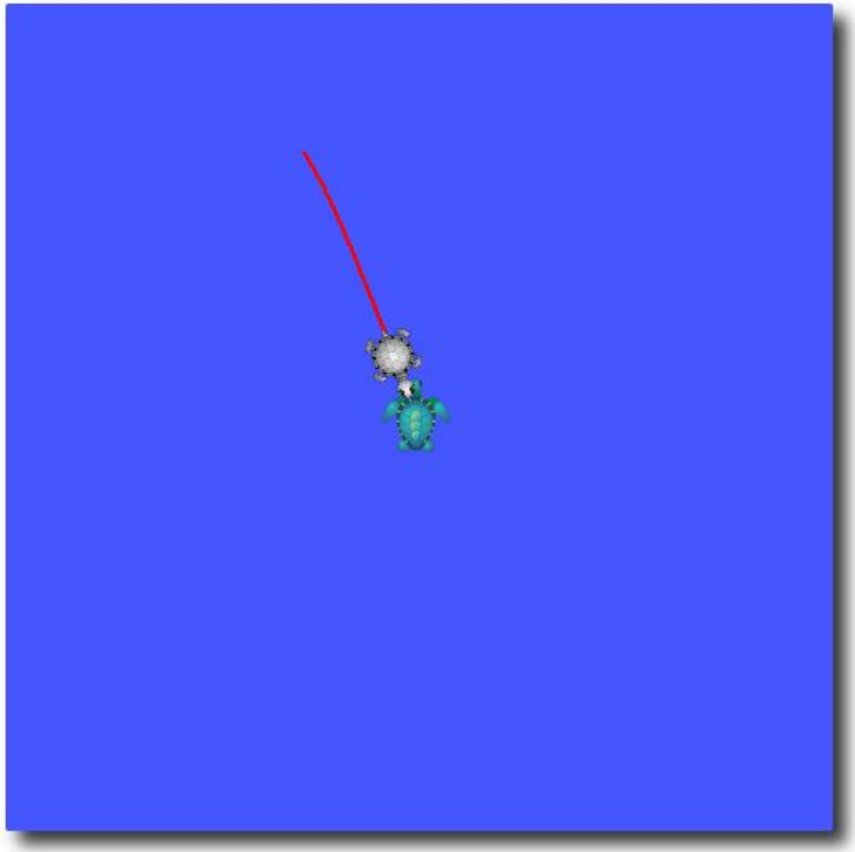
LIDAR reading in base_laser coordinate frame

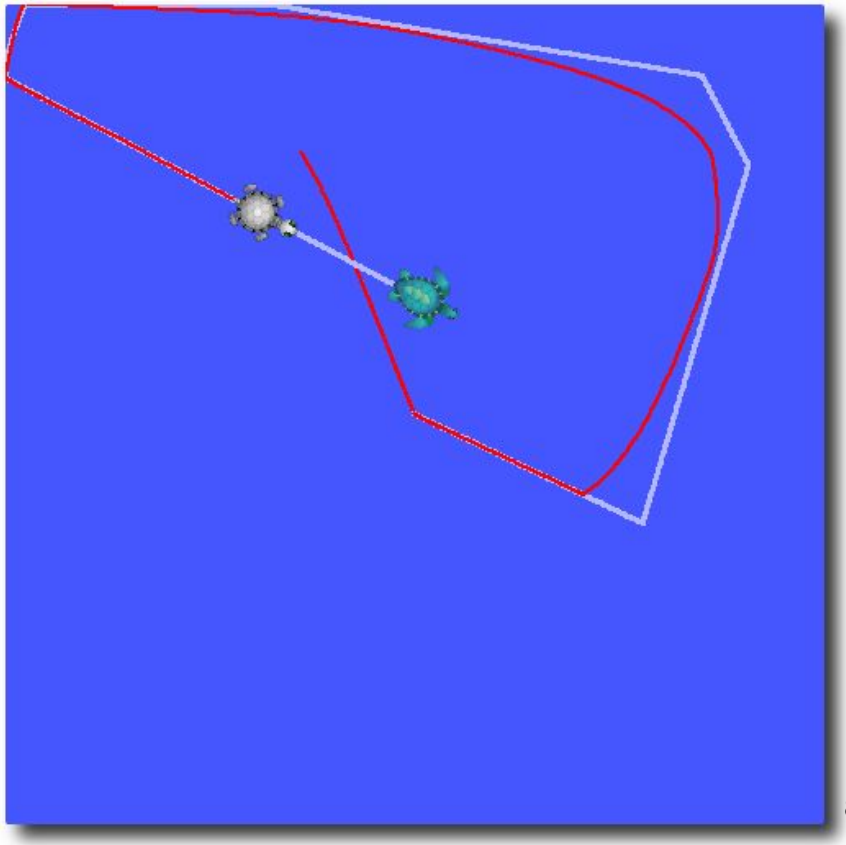




Activity: TF Demo - 1

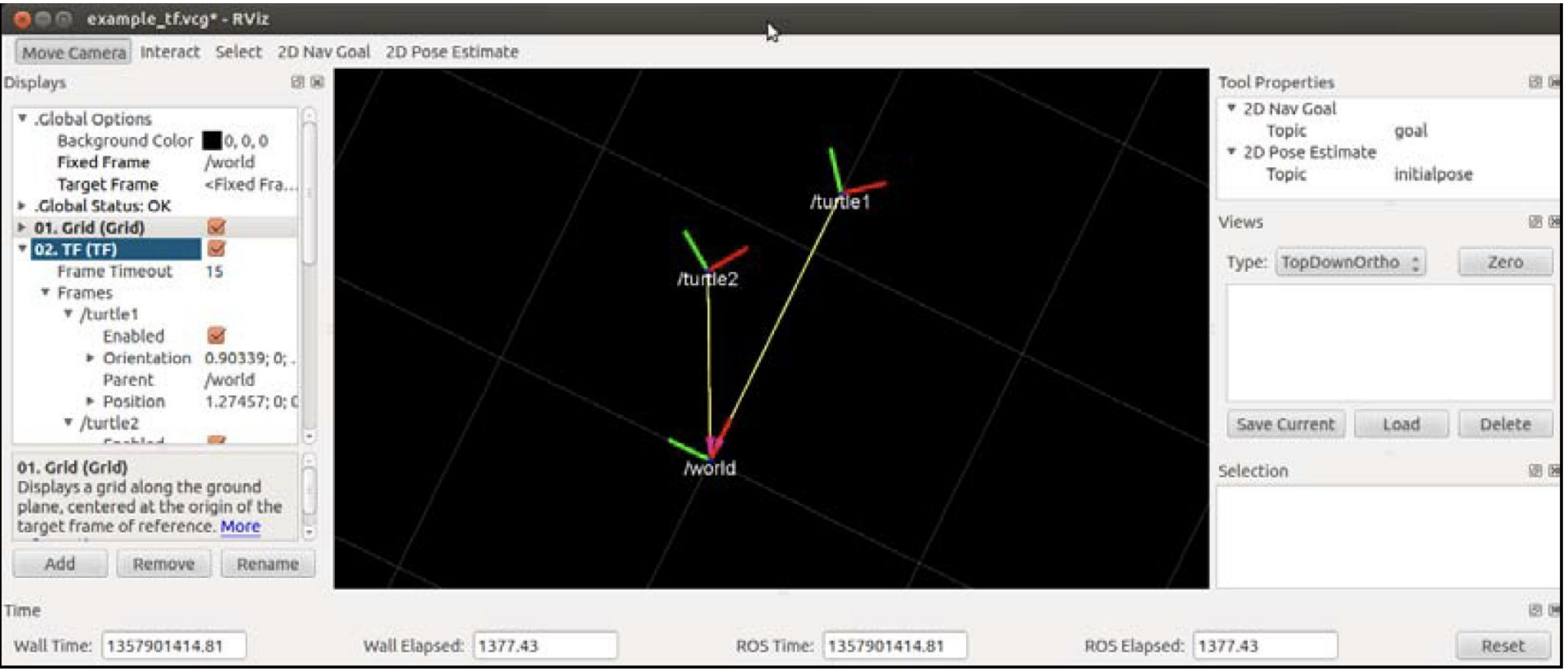
- sudo apt-get install ros-kinetic-turtle-tf2 ros-kinetic-tf2-tools ros-kinetic-tf
- (terminal 1) roslaunch turtle_tf2 turtle_tf2_demo.launch
- (terminal 2) rviz
- Turn on TF
- Observe one turtle tracks the other turtle





Activity: TF Demo - 2

- Set the fixed frame to /world
- Add the TF plugin to the left area. You will see that we have the /turtle1 and /turtle2 frames, both as children of the /world frame.
- Set the view of the world to TopDownOrtho



What is URDF?

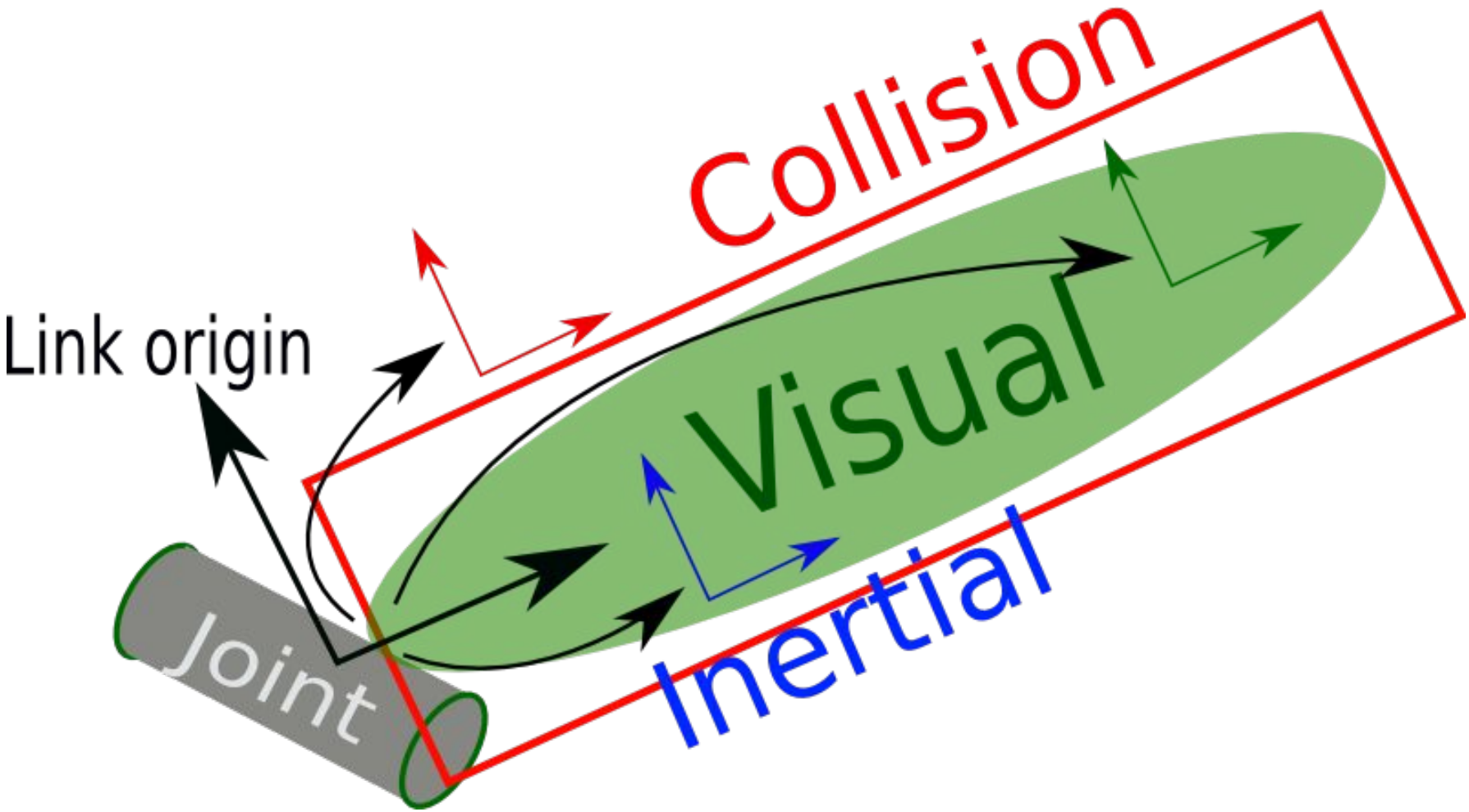
- The Unified Robot Description Format (URDF) is an XML specification to describe a robot.
- Note that the URDF specification cannot describe all robots. The main limitation at this point is that only tree structures can be represented, ruling out all parallel robots. Also, the specification assumes the robot consists of rigid links connected by joints; flexible elements are not supported.
- The URDF specification covers:
  - Kinematic and dynamic description of the robot
  - Visual representation of the robot
  - Collision model of the robot

<robot> Element

- The root element in a robot description file must be a robot, with all other elements must be encapsulated within.

```
<robot name="pr2">
  <!-- pr2 robot links and joints and more -->
</robot>
```

<link> Element

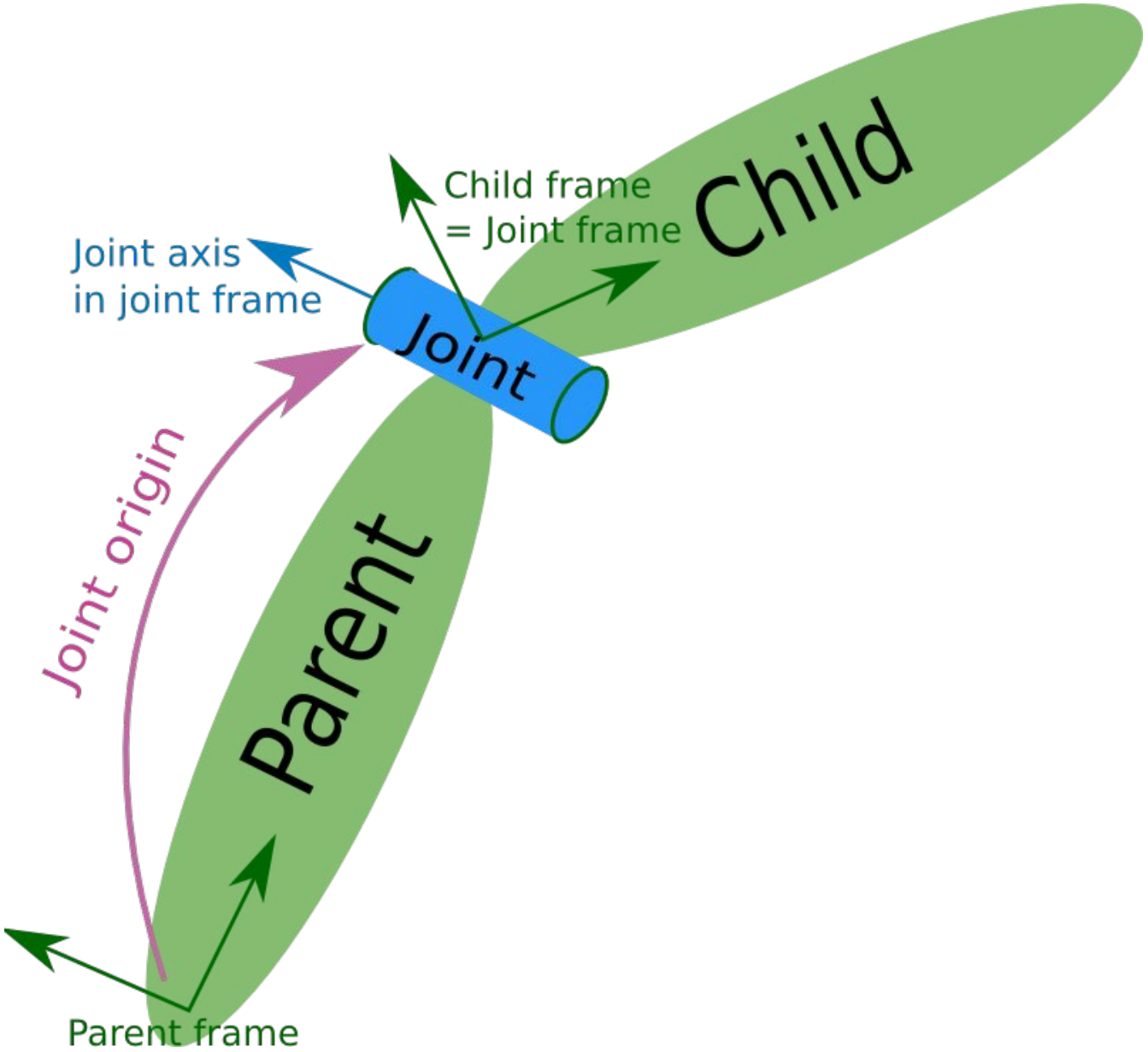


<link> Element

- The link element define the geometry (cylinder, box, , sphere, or mesh), the material (color, texture) and the origin.
- It also describes a rigid body with an inertia, visual features, and collision properties.

```
<link name="base_link">
    <visual>
        <geometry>
            <box size="0.2 .3 .1"/>
        </geometry>
        <origin rpy="0 0 0" xyz="0 0 0.05"/>
        <material name="white">
            <color rgba="1 1 1 1"/>
        </material>
    </visual>
</link>
```

<joint> Element



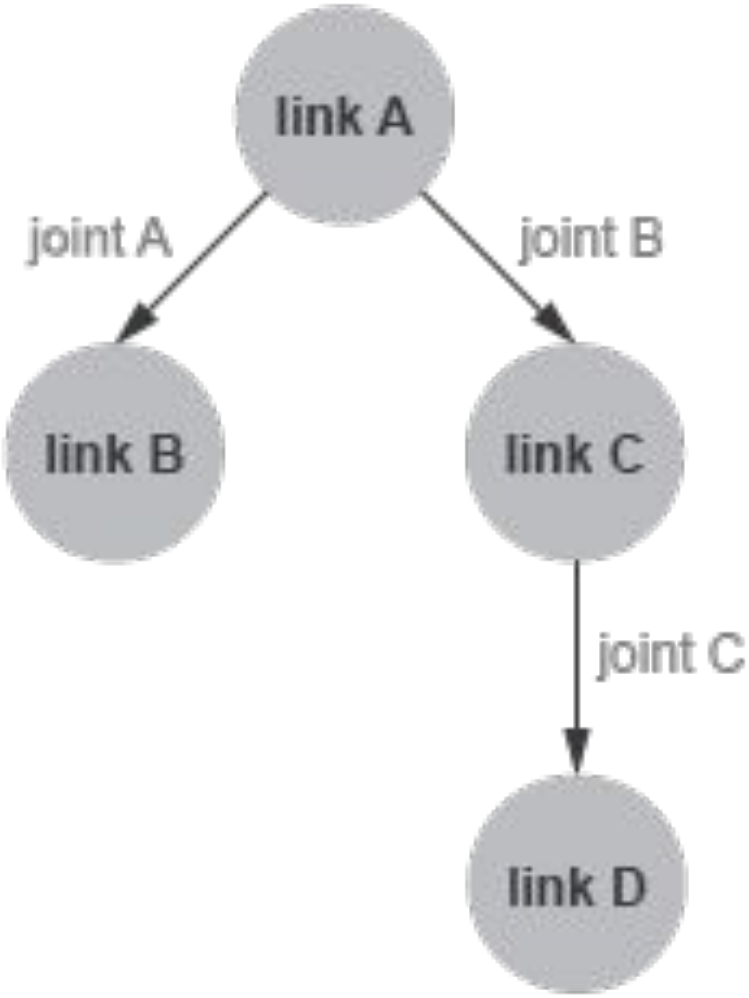
<joint> Element

- The joint element define the type of joint (fixed, revolute, continuous, floating, or planar).
- It also describes the kinematics and dynamics of the joint and also specifies the safety limits of the joint.

```
<joint name="base_to_wheel1" type="fixed">
    <parent link="base_link"/>
    <child link="wheel_1"/>
    <origin xyz="0 0 0"/>
</joint>
```

URDF Structure

```
<robot>
    <joint name = "joint A>
        <parent link = "link A" />
        <child link = "link B" />
    </joint>
    <joint name = "joint B>
        <parent link = "link A" />
        <child link = "link C" />
    </joint>
    <joint name = "joint C>
        <parent link = "link C" />
        <child link = "link D" />
    </joint>
</robot>
```



A Simple Robot URDF

```
<?xml version="1.0"?>
 <robot name="multipleshapes">
     <link name="base_link">
       <visual>
         <geometry>
           <cylinder length="0.6" radius="0.2"/>
         </geometry>
       </visual>
     </link>
    <link name="right_leg">
      <visual>
        <geometry>
          <box size="0.6 0.1 0.2"/>
        </geometry>
      </visual>
    </link>
    <joint name="base_to_right_leg" type="fixed">
      <parent link="base_link"/>
      <child link="right_leg"/>
    </joint>
 </robot>
```

Adding Physics to a URDF Model

- To add physics to a URDF model, you need to add collision and inertia elements. These are required for the gazebo simulation.
- The collision element is a direct subelement of the link object, at the same level as the visual tag
- The inertia element consists of mass (in kg) and 3x3 rotational inertia matrix defined by:

  ixx ixy ixz
  ixy iyy iyz
  ixz iyz izz

Example of Collision and Inertia

```
<link name="base_link">
      <visual>
        <geometry>
          <cylinder length="0.6" radius="0.2"/>
        </geometry>
        <material name="blue">
            <color rgba="0 0 .8 1"/>
        </material>
       </visual>
      <collision>
        <geometry>
          <cylinder length="0.6" radius="0.2"/>
        </geometry>
      </collision>
      <inertial>
        <mass value="10"/>
        <inertia ixx="0.4" ixy="0.0" ixz="0.0" iyy="0.4" iyz="0.0" izz="0.2"/>
      </inertial>
</link>
```

Display on RViz

- To view the URDF in RViz, we can create a launch file with the following code
- Change the red text to your package name

```
<launch>
  <arg name="gui" default="true" />
  <param name="robot_description" command="$(find xacro)/xacro $(arg
model)" />
  <node if="$(arg gui)" name="joint_state_publisher"
pkg="joint_state_publisher_gui" type="joint_state_publisher_gui" />
  <node unless="$(arg gui)" name="joint_state_publisher"
pkg="joint_state_publisher" type="joint_state_publisher" />
  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher" />
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
trobotics_arv)/rviz/urdf.rviz" required="true" />
</launch>
```

Install Joint State Publisher Package

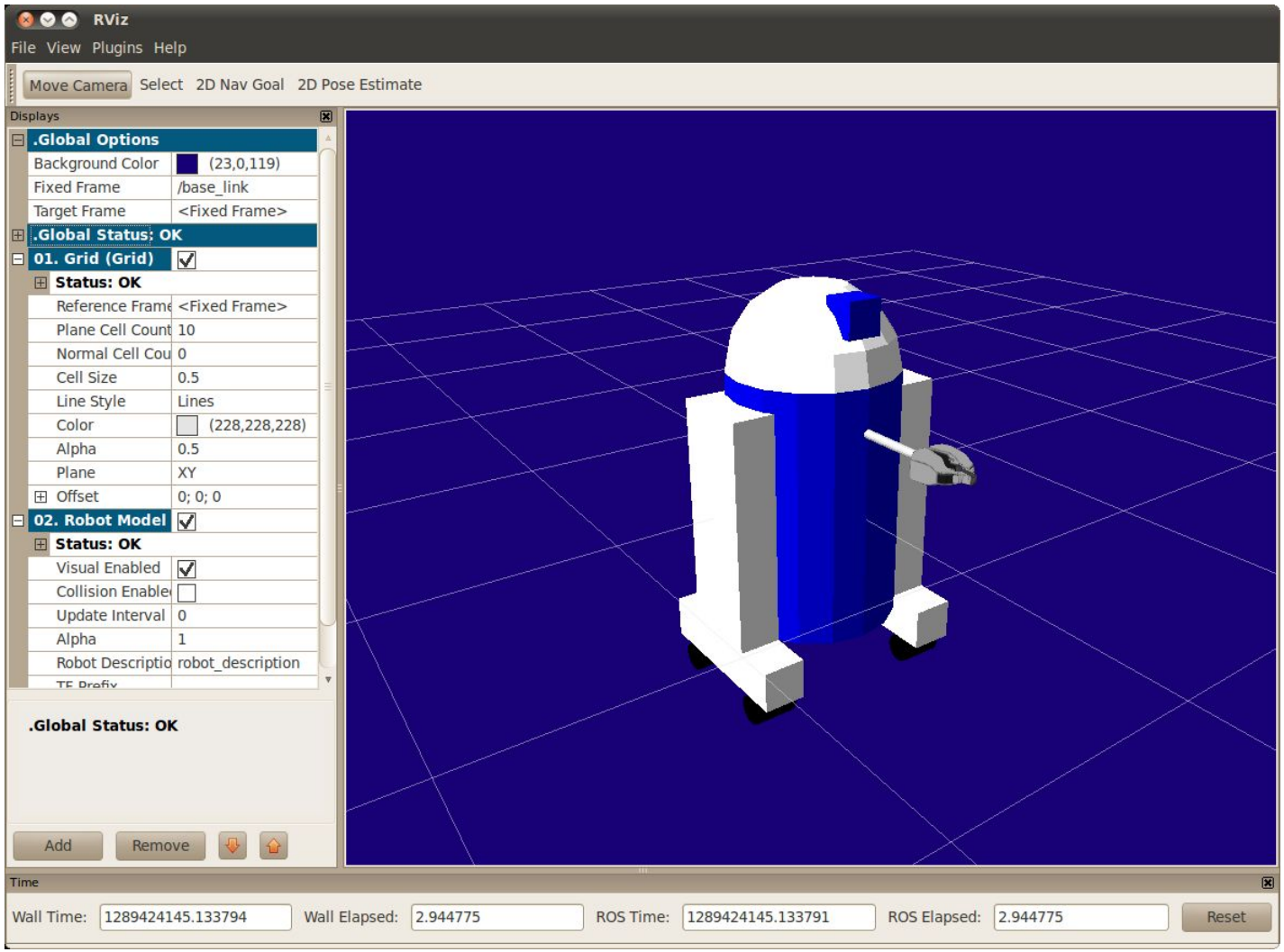
- If you have problem to view the RViz robot correct, install the following packages

sudo apt-get install
ros-kinetic-joint-state-publisher-gui

Activity: Visualize URDF in RViz

- Add the display.launch to the launch folder in your package
- Launch the visual.urdf as follows:

$ roslaunch <your package name> display.launch model:='visual.urdf'



Xacro

- Xacro helps to reduce the overall size of the URDF and make it easier to read and maintain.
- It also allows us to create modules and reutilize them to create repeated structures such as several arms or legs.
- To convert a xacro to urdf file, you can run the following command line:

```
xacro --inorder model.xacro > model.urdf
```

Xacro - Constant

- Xacro can define constants using xacro:property and make it easier to maintain.
- The value of the contents of the ${} construct are then used to replace the ${}. This means you can combine it with other text in the attribute.

```
<link name="base_link">
  <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
      <material name="blue"/>
  </visual>
  <collision>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
  </collision>
</link>
```

```
<xacro:property name="width" value="0.2" />
<xacro:property name="len" value="0.6" />
<link name="base_link">
  <visual>
      <geometry>
        <cylinder radius="${width}" length="${len}"/>
      </geometry>
      <material name="blue"/>
  </visual>
  <collision>
      <geometry>
        cylinder radius="${width}" length="${len}"/>
      </geometry>
  </collision>
</link>
```

Xacro - Math

- You can build up arbitrarily complex expressions in the ${} construct using the four basic operations (+,-,*,/), the unary minus, and parenthesis.
- Examples:

```
<cylinder radius="${wheeldiam/2}" length="0.1"/>
<origin xyz="${reflect*(width+.02)} 0 0.25" />
```

Xacro - Macro

- Xacro can generate a macro using xacro:micro

```
<xacro:macro name="default_origin">
    <origin xyz="0 0 0" rpy="0 0 0"/>
</xacro:macro>
<xacro:default_origin />
```

- This code will generate the following.

```
<origin rpy="0 0 0" xyz="0 0 0"/>
```

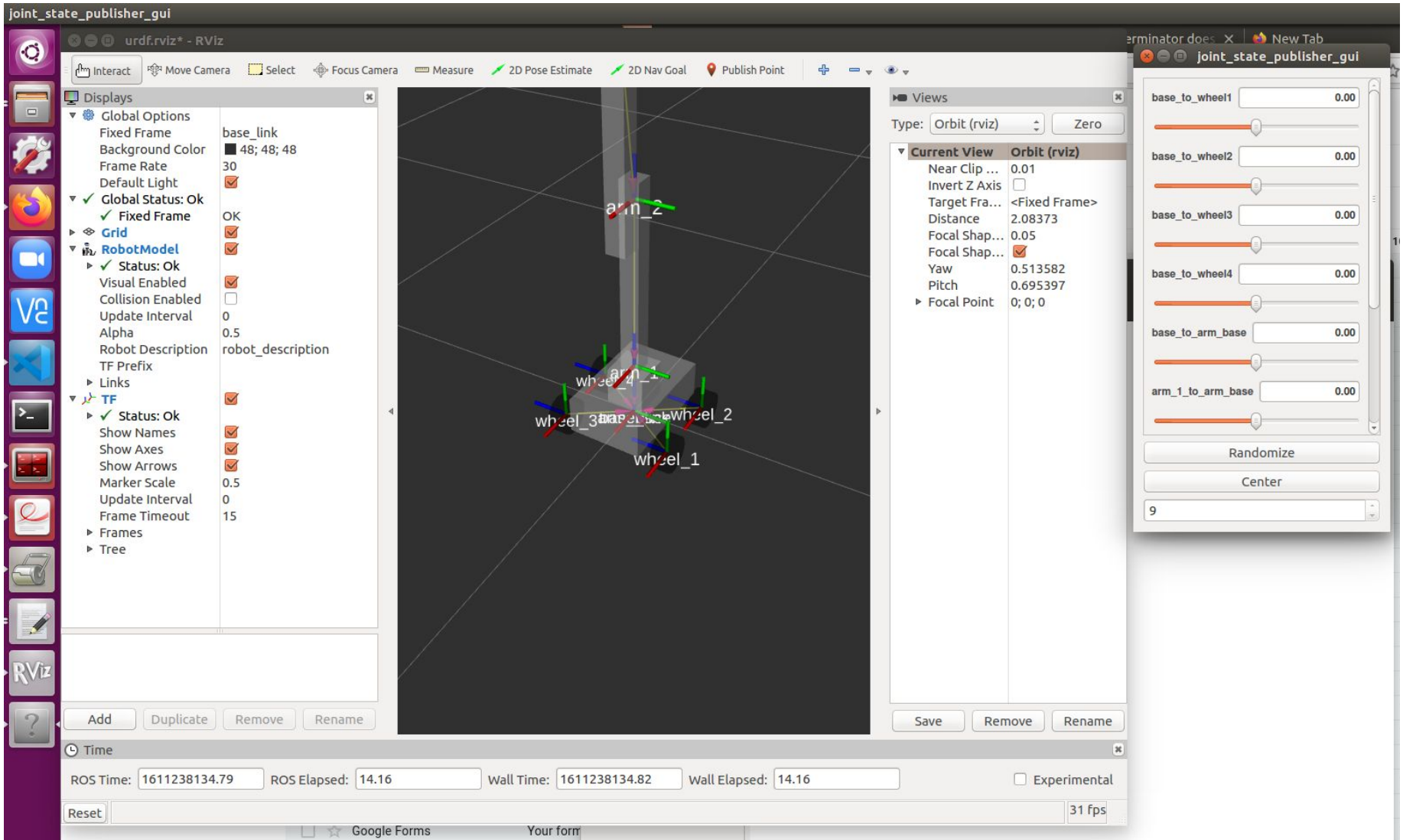
Xacro - Parameterized Macro

- You can also parameterize macros so that they don't generate the same exact text every time

```
<xacro:macro name="default_inertial" params="mass">
    <inertial>
        <mass value="${mass}" />
        <inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0"
        izz="1.0" />
    </inertial>
 </xacro:macro>
 <xacro:default_inertial mass="10"/>
```

Activity: Xacro

$ xacro --inorder robot1.xacro > robot1.urdf
$ roslaunch <your package name> display.launch model:='robot1.urdf'
$ move the sliders to see how the robot arm move



Topic 6
Program ROS Nodes and Topics

Visual Studio Code

To program Python or C++, download and Install Visual Studio Code to Ubuntu

https://code.visualstudio.com/download

Minimal ROS Program for a Node

- Minimal ROS program to create a node:
  - Import rospy library
  - Instantiate a node.
  - Making the node spin.
  - Shutdown ROS communications.

Activity: Create a Simple ROS Node

- Create a simple node in the my_robotics package as shown below
  #!/usr/bin/env python
  import rospy

  rospy.init_node('my_first_node')
  print('Started the node')
  rospy.spin()
  rospy.shutdown()

- To run the program:
  - Make the file executable and run
    rosrun my_robotics py_test1.py
  - Check the node by typing
    rosnode list
- To stop the program, you can do CTRL-Z or CTRL-C

Code Explanation

- To instantiate a node, you first have to initialize ROS communications. rospy.init_node will do that for you and also create the node.
- rospy.spin() will pause the program execution here, waiting for you to request to kill the node (for example CTRL Z or CTRL C in the terminal)
- rospy.shutdown() will basically shutdown what you started when you executed rclpy.init()

Print the Node Log Info

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def talker():
    rospy.init_node('my_second_node')
    while True:
        rospy.loginfo("Hello World")
        rospy.sleep(1)

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

Activity: Print the Node Log Info

- Modify the code to print the Hello World <number>

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def talker():
    rospy.init_node('my_second_node')
    i = 0
    while True:
        rospy.loginfo("Hello World %s" %i)
        i = i+1
        rospy.sleep(1)

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

Create a Publisher and Topics

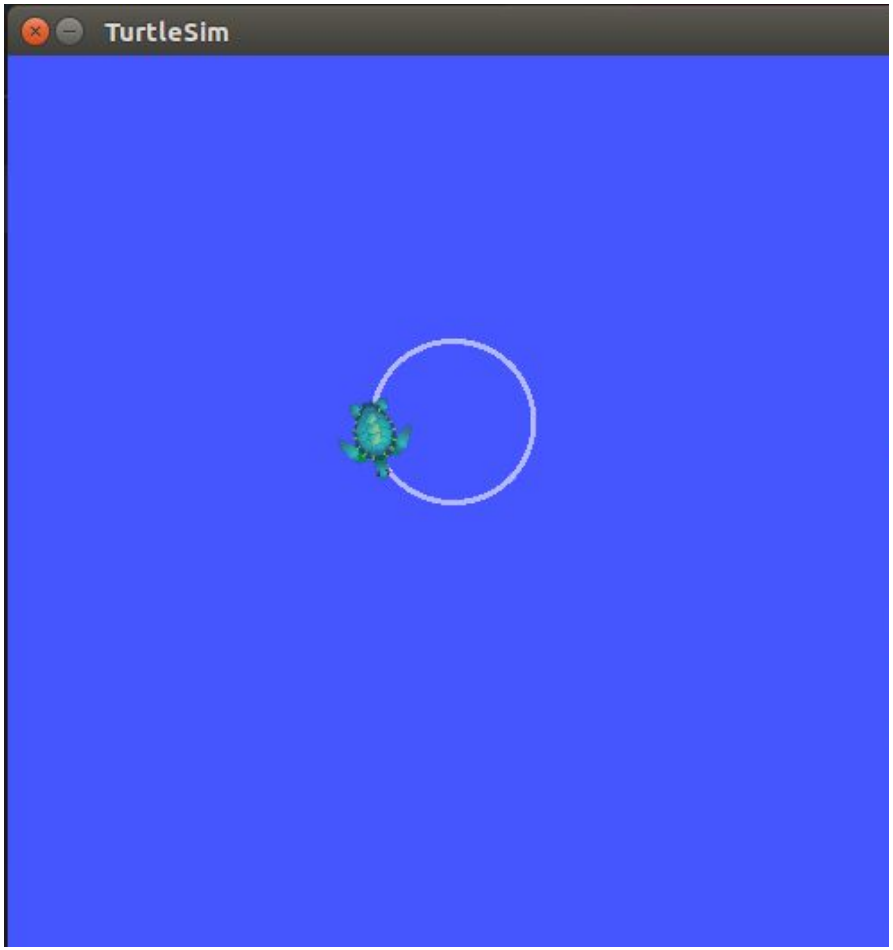
```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def talker():
    rospy.init_node('my_third_node')
    pub = rospy.Publisher('greetings',String,queue_size=10)

    i = 0
    while True:
        msg_data = "Hello World %s" % i
        pub.publish(msg_data)
        rospy.sleep(1)
        i = i+1

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

Code Explanation

- rospy.Publisher('greetings',String,queue_size=10) instantiate a publisher with the topics 'greetings' and the message type is a String.
- pub.publish(msg_data) will publish the data to the topics

Activity: Create a Publisher

- rosrun turtlesim turtlesim_node
- Modify the code move2.py to move the turtlesim in the following circular path continuously
- rosrun <package_name> move2.py



Create a Subscriber Node

```
import rospy
from std_msgs.msg import String

def callback(msg_data):
    rospy.loginfo(msg_data)

def listener():
    rospy.init_node('my_fourth_node')
    rospy.Subscriber('greetings',String,callback)
    rospy.spin()

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

Code Explanation

- rospy.Subscriber('greetings',String,callback)instantiate a subscriber with the topics 'greetings' and the message type is a String.
- The subscriber will listen to the callback function to receive the data from the topics

Activity: Publisher and Subscriber

- Create a temperature sensor publisher node "temp_sensor_node"
- Simulate the temperature data as follows: temp_data = 28.0 + random.random()+2
- Publish the data to the topics 'temperature'
- Create a subscriber node "iot_sensor" to receive the data from the temperature topics
- Run the following checks
  - rosnode list
  - rostopic list
  - rostopic info /temperature

Launch File

- A launch file is XML document which specifies:
  - which nodes to execute
  - their parameters
  - which other launch files to include
- roslaunch is a program that easily launches multiple ROS nodes
- Launch file has a .launch extension
- The syntax is
  <node pkg="<pkg_name>" type="<program>" name="<node_name>"/>

Launch File Example

```
<launch>
    <node pkg="my_robotics" type="py_test1.py"
    name="my_first_node"/>
    <node pkg="my_robotics" type="py_test2.py"
    name="my_second_node"/>
</launch>
```

Activity: Launch File

- Create a launch file to launch the following nodes from previous activities
  - temp_sensor_node
  - iot_sensor
- Run the rosnode list to check the nodes

Topic 7
Program ROS Messages

ROS Messages

- ROS uses a simplified messages description language for describing the data values (aka messages) that ROS nodes publish.
- This description makes it easy for ROS tools to automatically generate source code for the message type in several target languages.
- Message descriptions are stored in .msg files in the msg/ subdirectory of a ROS package.

Message Types

- Each field consists of a type and a name, separated by a space, i.e.:

  fieldtype1 fieldname1

  fieldtype2 fieldname2

  fieldtype3 fieldname3

- For example:

  int32 id

  string name

  float32 temperature

  float32 humidity

General Steps to Code a Message

1. Create a msg folder in your package folder
2. Create a message file with the extension ".msg"
3. Edit the message file by adding the elements (one per line)
4. Update the dependencies
    a. in package.xml
    b. in CMakeLists.txt
5. compile the package using catkin_make
6. make sure your message is created typing rosmsg list

Create Custom Message Type

- Create a msg folder
- Create a message file my_msg.msg file as shown below

```
string name
string greetings
```

Modify CMakeLists.txt

```
find_package(
catkin REQUIRED COMPONENTS
 roscpp
  rospy
  std_msgs
  message_generation
)
 add_message_files(
   FILES
   my_msg.msg
)
generate_messages(
   DEPENDENCIES
   std_msgs
 )
catkin_package(
  INCLUDE_DIRS include
  LIBRARIES my_robotics
  CATKIN_DEPENDS roscpp rospy std_msgs message_runtime
  DEPENDS system_lib
)
```

Modify package.xml

- Add the following to the package.xml

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

- Execute the catkin_make

Publisher with Custom Message Type

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
from demo1.msg import my_msg

def talker():
    rospy.init_node('my_seventh_node')
    pub = rospy.Publisher('greetings',my_msg,queue_size=10)
    i = 0
    msg_data = my_msg()
    while True:
        msg_data.name = "Ally"
        msg_data.greetings = "Hello %s" %i
        pub.publish(msg_data)
        rospy.sleep(1)
        i = i+1

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

Replace with your package name

Subscriber with Custom Message

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import Float32
from demo1.msg import my_msg

def callback(msg_data):
    greeting_msg = msg_data.name + "," + msg_data.greetings
    rospy.loginfo(greeting_msg)

def listener():
    rospy.init_node('my_eighth_node')
    rospy.Subscriber('greetings',my_msg,callback)
    rospy.spin()

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

Replace with your package name

Test the Message

(terminal 1) roscore
(terminal 2) rosrun my_robotics py_test7.py
(terminal 3) rosrun my_robotics py_test8.py

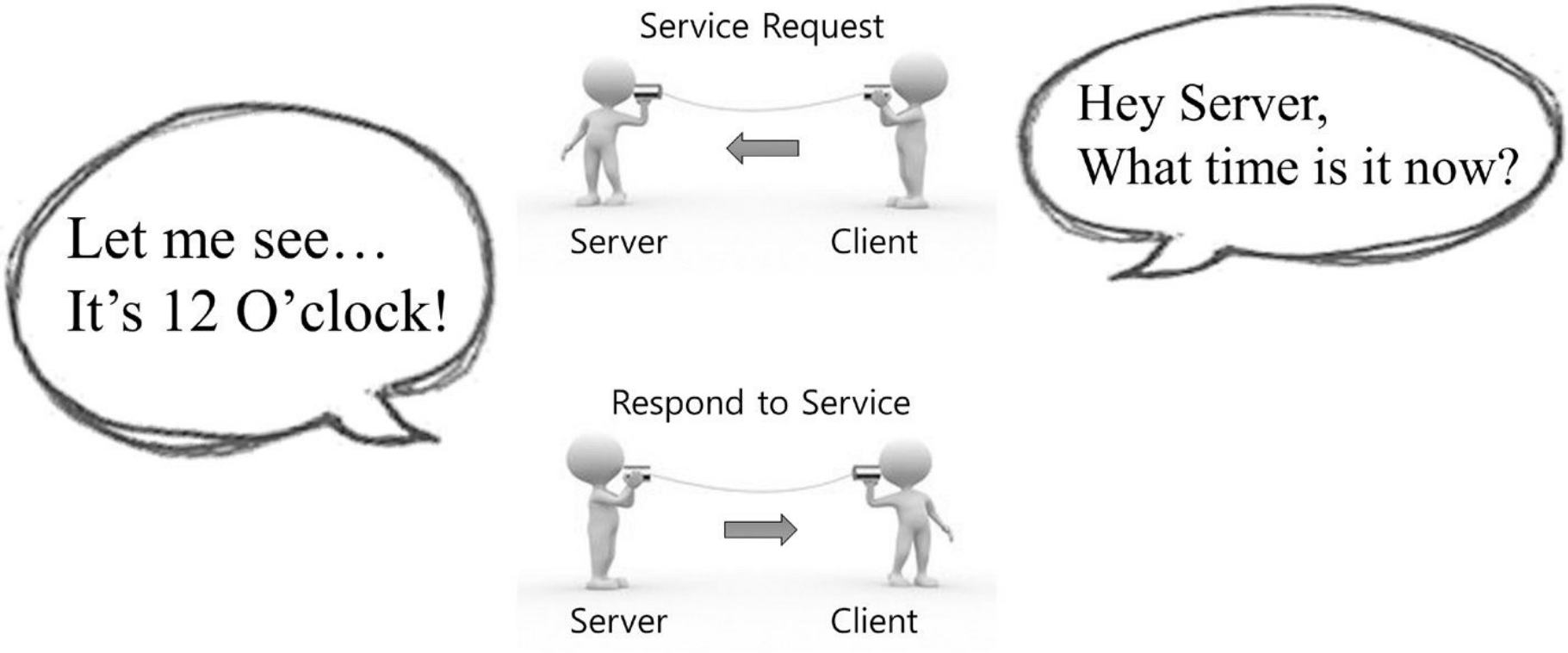
Activity: Custom Message Type

- Create a custom message type 'iot_sensor.msg'
  int32 id
  float32 temperature
- Modify the CMakeLists.txt and do a catkin_make
- Create a publisher code py_test9.py to publish the custom message type
- Create a subscriber py_test10.py to listen to the custom message.
- Testing:

(terminal 1) roscore
(terminal 2) rosrun my_robotics py_test9.py
(terminal 3) rosrun my_robotics py_test10.py

Topic 8
Program ROS Services

Service Message Communication



General Steps to Code a Service

1. Create a srv folder in your package folder
2. Create a service file with the extension ".srv"
3. Edit the service file by adding the elements (one per line)
4. Update the dependencies
    a. in package.xml
    b. in CMakeLists.txt
5. compile the package using catkin_make

Custom Service Type

Create a srv folder
Create a service file my_service.srv file as shown below

```
int32 onezero
---
string switch
```

Modify CMakeLists.txt

```
add_service_files(
  FILES
  my_service.srv
)
```

Service Server Code

```
#!/usr/bin/env python
import rospy
from demo1.srv import *

def service():
    rospy.init_node('my_eleven_node')

rospy.Service('on_off_service',my_service,turn_on_off)
    rospy.spin()

def turn_on_off(req):
    if req.onezero == 1:
        return my_serviceResponse('ON')
    else:
        return my_serviceResponse('OFF')

if __name__ == '__main__':
```

Replace with your package name

Call the Server Service from CMD

(terminal 1) roscore
(terminal 2) rosrun my_robotics py_test11_server.py
(terminal 3)
    rosservice call /on_off_service 0
    rosservice call /on_off_service 1

Service Client Code

```
#!/usr/bin/env python
import sys
import rospy
from demo1.srv import *

def on_off(req):
    rospy.wait_for_service('on_off_service')
    onoff = rospy.ServiceProxy('on_off_service', my_service)
    print(onoff(req))

if __name__ == "__main__":
    req = int(sys.argv[1])
    on_off(req)
```

Replace with your package name

Call the Service from Client File

(terminal 1) roscore
(terminal 2) rosrun my_robotics py_test11_server.py
(terminal 3)
rosrun my_robotics py_test11_client.py 0
rosrun my_robotics py_test11_client.py 1

Activity: Program ROS Service

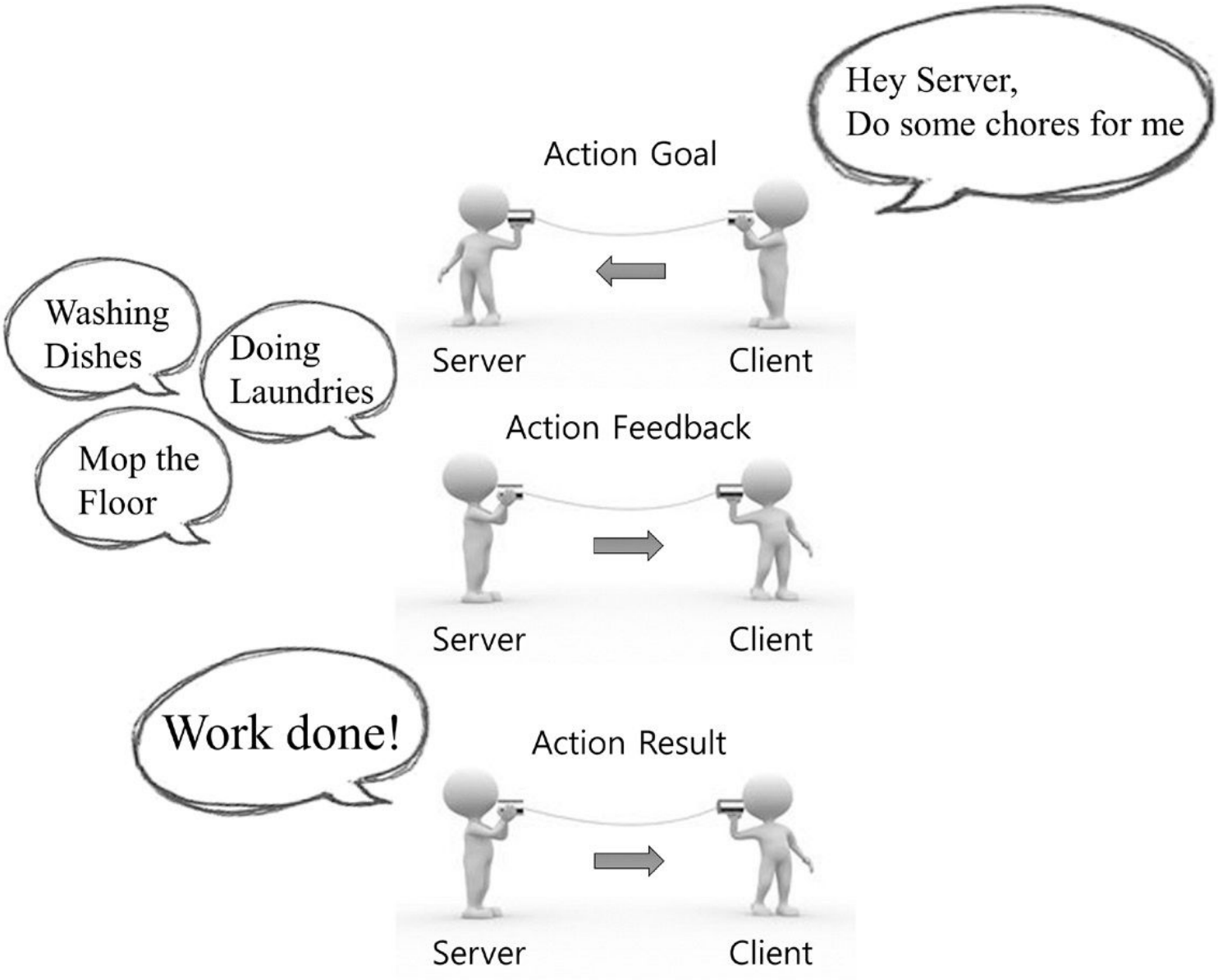
- Create a custom service server file 'my_sum.srv' to add two integers

  int32 a
  int32 b
  ---
  int32 sum
- Modify the CMakeLists.txt
- Create the service server file py_test12_server.py and service client file py_test12_client to add the two integers and return the value
- Testing:

(terminal 1) roscore
(terminal 2) rosrun my_robotics py_test12_server.py
(terminal 3) rosrun my_robotics py_test12_client.py 3 5

Topic 9
Program ROS Actions

Action Message Communication



General Steps to Code a ROS Action

- Create a action folder
- Create a custom action file with extension .action inside the action folder
- Create a ROS Server node
- Create a ROS Client node
- Execute the action
- Consume the action by the client

Create Custom Action Type

- Create a action folder
- Create a action file my_action.action file as shown below

```
#goal
int32 order
---
#result
int32[] sequence
---
#feedback
int32[] sequence
```

Modify CMakeLists.txt

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

add_action_files(
   FILES
   my_action.action
 )

generate_messages(
  DEPENDENCIES
  std_msgs
  actionlib_msgs
 )
```

Modify package.xml

- Add the following to the package.xml

```
<build_export_depend>actionlib_msgs</build_export_depend>
<exec_depend>actionlib_msgs</exec_depend>
```

- Execute the catkin_make

Action Action Code - 1

```
#!/usr/bin/env python
import rospy
import actionlib
from actionlib import SimpleActionServer
from my_robotics.msg import
my_actionAction,my_actionGoal,my_actionResult,my_actionFeedback

class FibonacciActionServer(object):
    # create messages that are used to publish feedback/result
    feedback = my_actionFeedback()
    result = my_actionResult()

    def __init__(self, name):
        self.action_name = name
        self.action_server = SimpleActionServer(self.action_name,
my_actionAction, execute_cb=self.execute_cb, auto_start = False)
        self.action_server.start()

    def execute_cb(self, goal):
        # helper variables
        r = rospy.Rate(1)
        success = True
```

Action Action Code - 3

```
        # append the seeds for the fibonacci sequence
         self.feedback.sequence = []
         self.feedback.sequence.append(0)
         self.feedback.sequence.append(1)

         # publish info to the console for the user
         rospy.loginfo('%s: Executing, creating fibonacci sequence of order %i
with seeds %i, %i' % (self.action_name, goal.order,
self.feedback.sequence[0], self.feedback.sequence[1]))

         # start executing the action
         for i in range(1, goal.order):
             # check that preempt has not been requested by the client
             if self.action_server.is_preempt_requested():
                 rospy.loginfo('%s: Preempted' % self.action_name)
                 self.action_server.set_preempted()
                 success = False
                 break
```

Action Action Code - 3

```
        self.feedback.sequence.append(self.feedback.sequence[i] +
self.feedback.sequence[i-1])
            # publish the feedback
            rospy.loginfo('publishing feedback ...')
            self.action_server.publish_feedback(self.feedback)
            # this step is not necessary, the sequence is computed at 1 Hz for
demonstration purposes
            r.sleep()

        if success:
            self.result.sequence = self.feedback.sequence
            rospy.loginfo('%s: Succeeded' % self.action_name)
            self.action_server.set_succeeded(self.result)

if __name__ == '__main__':
    rospy.init_node('fibonacci')
    server = Fibona
```

Action Client Code - 1

```
#!/usr/bin/env python
import rospy
import actionlib
from my_robotics.msg import
my_actionAction,my_actionGoal,my_actionResult,my_actionFeedback
import sys

def fibonacci_client():
    # Creates the SimpleActionClient, passing the type of the action
    # (FibonacciAction) to the constructor.
    client = actionlib.SimpleActionClient('fibonacci', my_actionAction)

    # Waits until the action server has started up and started
    # listening for goals.
    client.wait_for_server()

    # Creates a goal to send to the action server.
    goal = my_actionGoal(order=20)
```

Action Client Code - 2

```
    # Sends the goal to the action server.
    client.send_goal(goal)
    # Waits for the server to finish performing the action.
    client.wait_for_result()
    # Prints out the result of executing the action
    return client.get_result()  # A FibonacciResult

if __name__ == '__main__':
    try:
        # Initializes a rospy node so that the SimpleActionClient can
        # publish and subscribe over ROS.
        rospy.init_node('fibonacci_client_py')
        result = fibonacci_client()
        print("Result:", ', '.join([str(n) for n in result.sequence]))
    except rospy.ROSInterruptException:
      pass
```

Run Client and Server Action Nodes

- (terminal 1) roscore
- (terminal 2)
  cd ~/catkin_ws
  catkin_make
- (terminal 2)
  rosrun my_robotics py_test13.py
- (terminal 3)
  rosrun my_robotics py_test14.py

Activity: Program ROS Action

- Create a custom action file

```
#goal
int32 order
---
#result
int32[] sequence
---
#feedback
int32[] sequence
```

- Modify the CMakeLists.txt and do a catkin_make
- Create a action server and client programs 'py_test15_server.py' and 'py_test16_client.py'
- Run the action and check the result

Summary Q&A

Feedback

https://goo.gl/R2eumq

TINFOTECH

CERTIFICATE

of ACCOMPLISHMENT

You will receive a digital certificate in your email
after the completion of the class
If you did not receive the digital certificate,
please send your request to
enquiry@tertiaryinfotech.com

Thank You!

Man Guo Chang
gc.man.sg@gmail.com