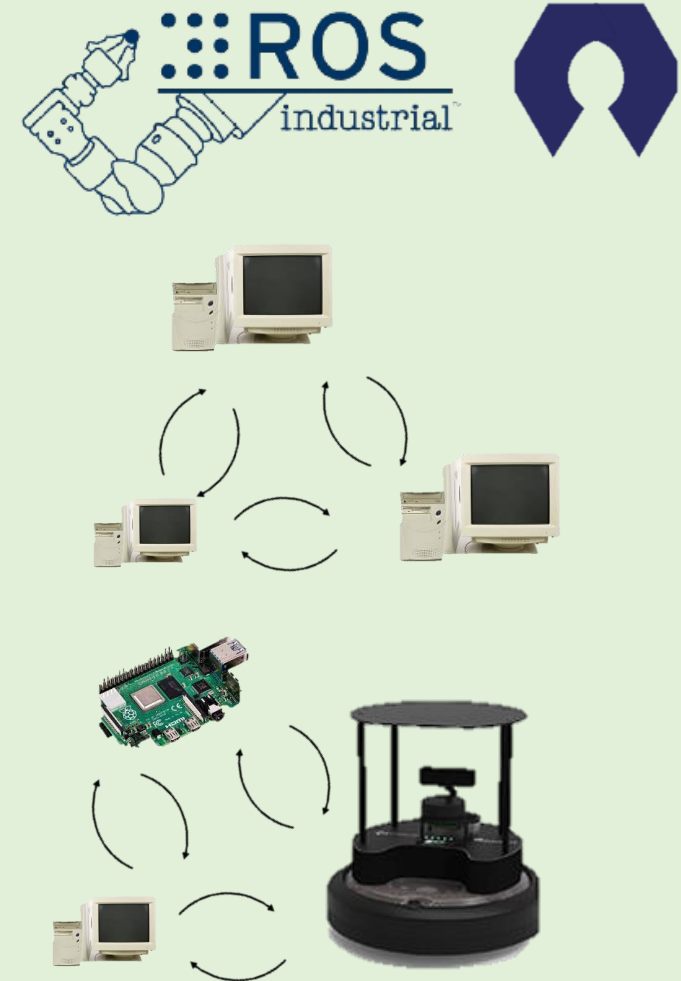


ROS tutorial


Ahmad Alghooneh

ROS tutorial - Contents


- Introduction.
- Communication in ROS.
 - Publishing/Subscription in ROS.
 - Grouping in ROS through ROS_DOMAIN.
 - Quality of Service in ROS.
- RVIZ
- TF-Tree
- Synchronization.
- Communication in ros, req/resp
 - Services.
 - Actions.
- Example




ROS tutorial - Introduction

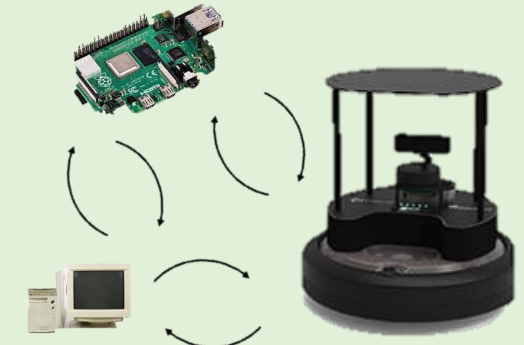
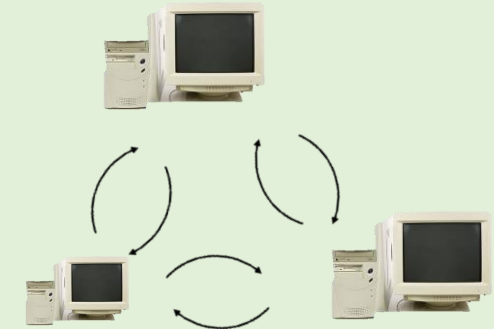
-  A Process: In a computer, a process refers to an executing instance of a program or software. It is an independent unit of execution that contains the program code and its current activity. Each process has its memory space, system resources, and execution context, separating it from other processes running on the same computer.

Google Chrome (20) 1.7% 1,233.5 MB 0.1 MB/s 0.1 Mbps
2346 ahmad 20 0 396K 30836 30296 S 0.7 0.3 0:00.32 /usr/libexec/gnome-terminal-server

-  A Process: Therefore, your code compiled and run is also a process.



-  A Middleware: is the one who makes the communication between the processes on a local/external machine.



ROS tutorial - Introduction



ROS is a **middleware**, a collection of **libraries/headers/definitions/executables** that help to connect processes both on the local machine or machines across a network.

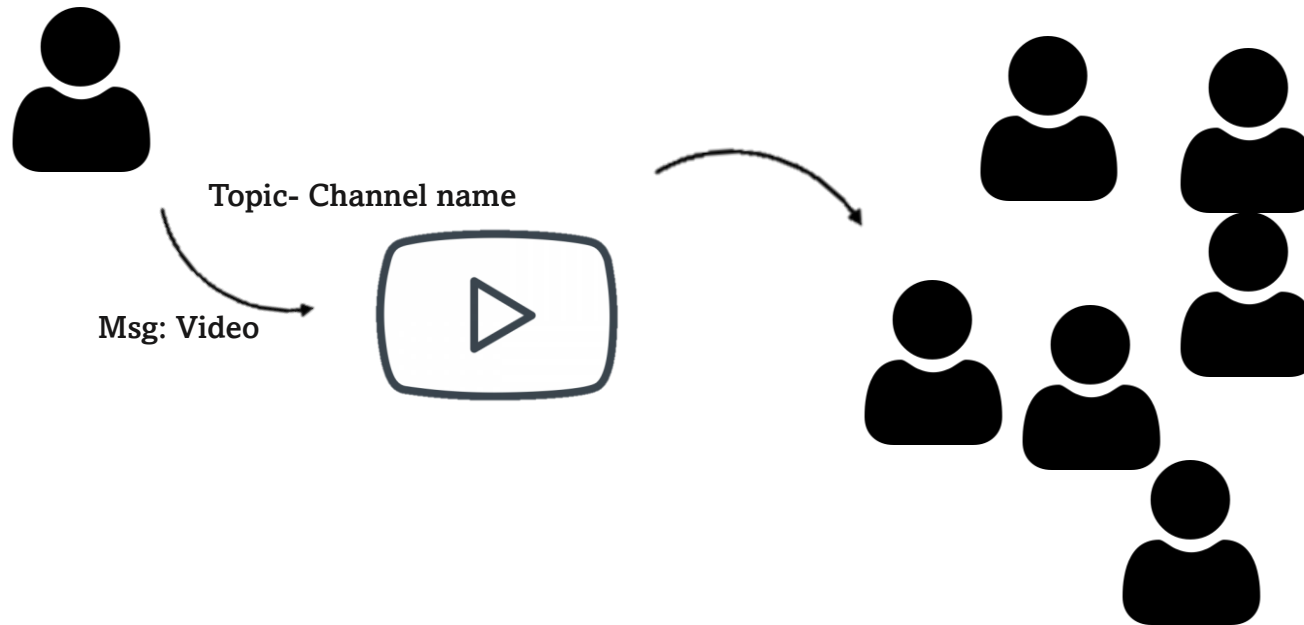
Other than being a middleware, it comes with other **toolboxes and libraries** that make a **system** able to **operate robotics**, hence they call it Robotic Operating System

ROS tutorial – Communication



Publishers/Subscriber Scheme

Publisher^s



Server/Client Scheme



ROS tutorial – Interface, sub/pub



- Information in ros goes **under rostopics**
- It is the medium under which we send information.
- e.g. we can use **rostopics** to send commands to the robot or receive the position of the wheels.
- After targeting the topic, we should look for the type of message, that we send over the network, we call it **rosmmsg**.
- To see the available topics over the network, we do **\$>ros2 topic list**
- To see the content of a topic, **\$>ros2 topic echo \$topic**
- To see the processes that are interfacing with ROS, **\$>ros2 node list**

ROS tutorial – Interface, sub/pub



- To publish a msg on a topic using terminal you can

```
$>ros2 topic pub -r $rate $topic $msgType $msg
```

- If the topic has different fields that we want to see only we should

```
do: $>ros2 topic echo $topic --field field1.field2..fieldn
```

e.g. show the position of a odom

```
$>ros2 topic echo /odom --field pose.pose
```

- To check the rate of a topic

```
$>ros2 topic hz $topic
```
- To see the information about the topic, the msg type, it's publisher and subscriber:

```
$>ros2 topic info $topic
```

ROS tutorial – Interface, sub/pub



- To find out about the interfaces used for your ROS installation
`$>ros2 interface list`
- You will see three type of interfaces: `msgs`, `services`, and `actions`.
- For sub/pub section we go through `msgs`. The `services` and `actions` are for server/client communication type.
- The `msgs` are structures defined so it can be parsed on both subscriber and publisher side.
- To show the content of the msg you can do `$>ros2 interface show $msg`
`ros2 interface show std_msgs/msg/String`

ROS tutorial – Interface, sub/pub



- Creating a ROS node is only about writing a script with ros headers included, linked against its libraries.
- The ROS libraries are primarily written in C++, and python, therefore they can be linked against Java, JavaScript, or other languages.
- However, sometimes, we're after creating a package that contains couple of nodes, our own msg/service/action definition, and easy to ship around.
- There we go after **ros packages**.

ROS tutorial – Script, sub/pub



- In this course, we only write codes with python.
- The most of the ros related classes/interfaces are included in `rclpy`.
- Now let's write a simple publisher that publishes a msg across ros network.
- This stand alone node can be created with `$>vim minPublisher.py`
- On top we start by importing the necessary libraries.
- `-import rclpy`: this will help us to initiate and run the node
- `-from rclpy.node import Node` this is the type that contains subscription/publisher methods.
- `-from std_msgs.msg import String` this will help us to send the a standard structure of information, so it's easy to parse on both sides

ROS tutorial – Script, sub/pub



- Now that the necessary libraries are included, we move to the next step, which is creating the publisher.
- It is advised to use classes when writing a Node, this will help us:
 - Inherit from Node imported, so we can use its methods/properties
 - We can share variables across the class instance
- Start by `-class minimalPublisher(Node)`
- - `def __init__(self):\ super().__init__("nodeName")` this will initialize the parent class.
- -`self.publisher=self.create_publisher(String,'topic',10),` this line will write the publisher, that takes the msg type as the first argument, the topic as the second, and `QoS` as the last. More follows!

ROS tutorial – Script, sub/pub

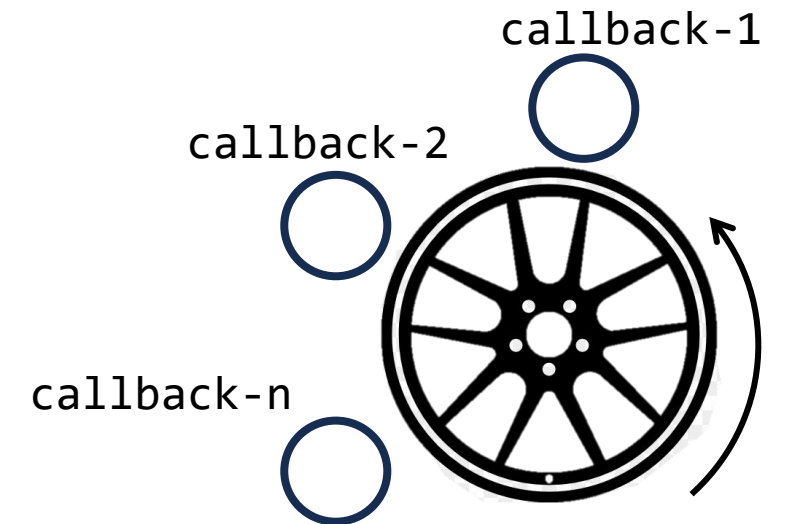


- To publish the message, we should decide on a **rate**.
- For this purpose, we define a timer.
- The timer is an **event-loop** that is triggered at a **timer_period**
- Just as every other **event-loop**, it needs a callback.
- The **callback** will **trigger** each time the event is happened, which in here is in each timer period.
- **-self.timer=self.create_timer(timer_period, self.timerCallback)**
- **-def timerCallback(self)**
- **-msg=String(); msg.data='MTE544 is awesome'; self.publisher_.publish(msg)**

ROS tutorial – Script, sub/pub



- In the main method, `-def main(args=None)`
- We initialize the node; this will build the socket to listen on `ip/port` combination.
- `-rclpy.init(args=args); minPublisherInstance=minimalPublisher()`
- `-rclpy.spin(minPublisherInstance);` this will take care of running the ros engine, hence the term spin
- Save the changes by `:wq` in vim.



ROS tutorial – Script, sub/pub

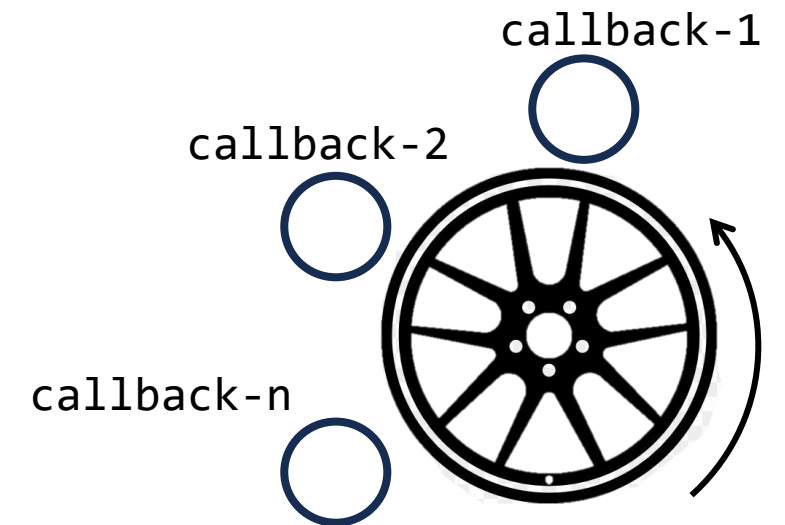


- On another terminal, `$>vim minSubscriber.py`
- Start by `-class minimalSubscriber(Node)`
- - `def __init__(self):\ super().__init__("nodeName")` this will initialize the parent class.
- - `self.subscription=self.create_subscription(String,'topic',self.subsCallback,10),` this line will write the publisher, that takes the msg type as the first argument, the topic as the second, and QoS as the last. More follows!
- - `def subsCallback(self, msg);`
- - `self.get_logger().info('I heard: "%s"' % msg.data)`

ROS tutorial – Script, sub/pub



- In the main method, `-def main(args=None)`
- We initialize the node; this will build the socket to listen on `ip/port` combination.
- `-rclpy.init(args=args); minSubscriberInstance=minimalPublisher()`
- `-rclpy.spin(minSubscriberInstance)`; this will take care of running the ros engine, hence the term spin
- Save the changes by `:wq` in vim.



ROS tutorial – Script, sub/pub



- Open a terminal and then `$>python minPublisher.py`
- Open another terminal and then `$>python minSubscriber.py`
- To check the topic is there you can do `$>ros2 topic list`
- And then, you can see the content of a topic `$>ros2 topic echo $topic`
- Now you can see that the two nodes are talking to each other on ros network.
- The last slides should be adopted any time that we want to have a subscriber/publisher written in python.

ROS tutorial – Script, sub+pub

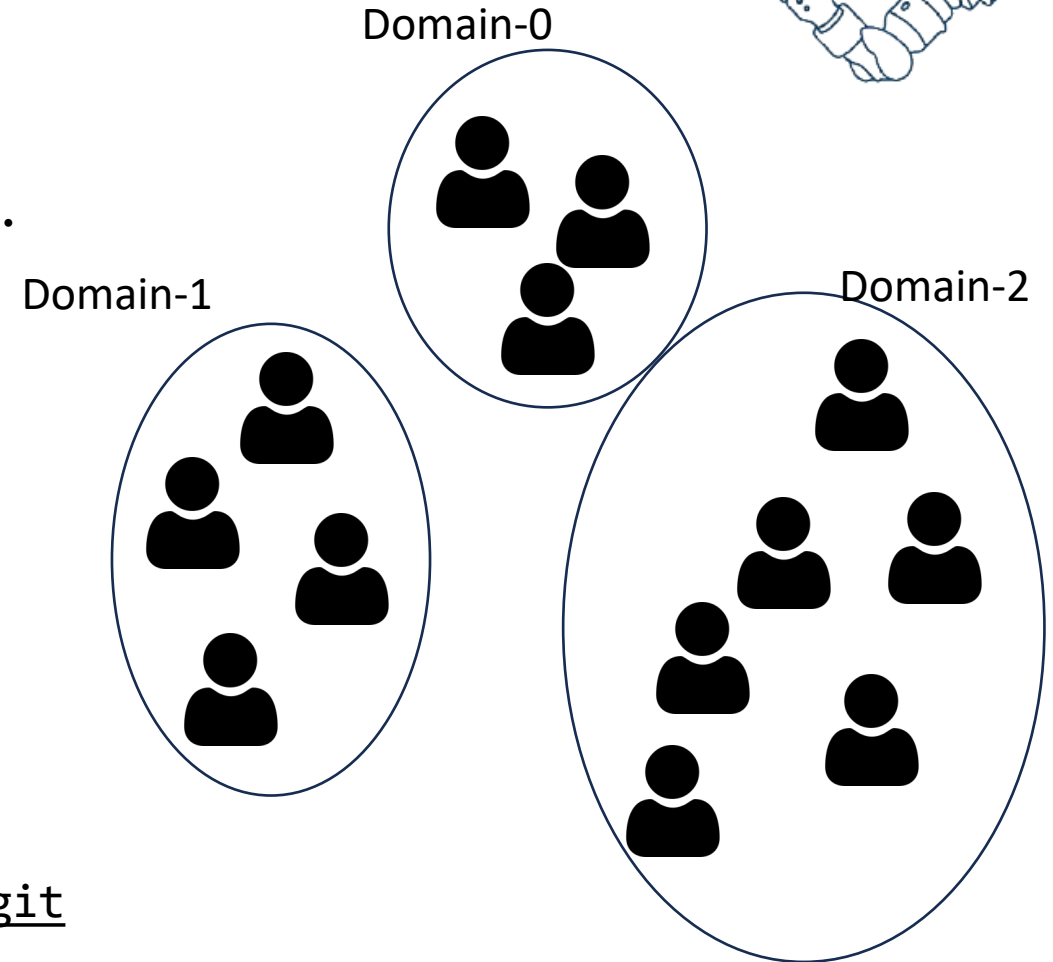


- Adopt the last approach to create your class
- Write your subscriber callback and your publisher object.
- Choose your rate for publishing information
- Use class variable “**self**” to carry information between timer and subscriber callback.

ROS tutorial – ROS DOMAIN



- In ROS2, refers to a logical grouping of nodes that can communicate with each other.
- We set that like an env variable,
“`export ROS_DOMAIN_ID`”
- It keeps communication,
 - Isolated
 - Enables namespacing
 - Enabling communication across domains through gateways.
 - https://github.com/ros2/domain_bridge.git



ROS tutorial – Quality of Service



- Quality of Service (**QoS**) refers to a set of configurable parameters that allow you to control how communication occurs between different nodes in a ROS 2 system.
- The configurable parameters for QoS is,
 - **History Depth**: Defines the number of messages to be held for the late-joining subscriber.
 - A late-joining subscriber: a subscriber that starts listening to the topic after the publisher started sending out msgs.
 - **Durability**: Defines how long should the msgs be held for the late-joining subscriber.

There are two kinds, **VOLATILE**; store none or **TRANSIENT LOCAL**: store msgs as long as there are local subscribers.

ROS tutorial – Quality of Service



- Quality of Service (**QoS**) refers to a set of configurable parameters that allow you to control how communication occurs between different nodes in a ROS 2 system.
- The configurable parameters for QoS is,
 - **Reliability**: Defines the level of reliability for the msgs, it is **BEST_EFFORT**; telling us there is no guarantee for the msgs to deliver. There is **RELIABLE**: at least can deliver one.
 - **Deadline**: sets a expiry time limit for the msgs;
- If the QoS is not set compatible, the subscription will not trigger.

ROS tutorial – Quality of Service



Publisher	Subscription	Compatible	Result
Volatile	Volatile	Yes	New messages only
Volatile	Transient local	No	No communication
Transient local	Volatile	Yes	New messages only
Transient local	Transient local	Yes	New and old messages

Publisher	Subscription	Compatible
Best effort	Best effort	Yes
Best effort	Reliable	No
Reliable	Best effort	Yes
Reliable	Reliable	Yes

ROS tutorial – Example. QoS, tb4 /odom



1.Reliability: RELIABLE (value: 2)

1. This setting ensures that messages are delivered to subscribers at least once. If a message is lost during transmission, the publisher will attempt to resend it until the subscriber acknowledges receipt.

2.Durability: TRANSIENT_LOCAL (value: 2)

1. This setting indicates that messages are stored for late-joining local subscribers within the same ROS 2 domain. Messages are not stored persistently, and they are discarded when all local subscribers have received them.

3.History: KEEP_LAST (value: 1)

1. This setting specifies that a fixed number of the most recent samples should be stored in the history buffer. In this case, the history buffer is configured to keep the most recent samples.

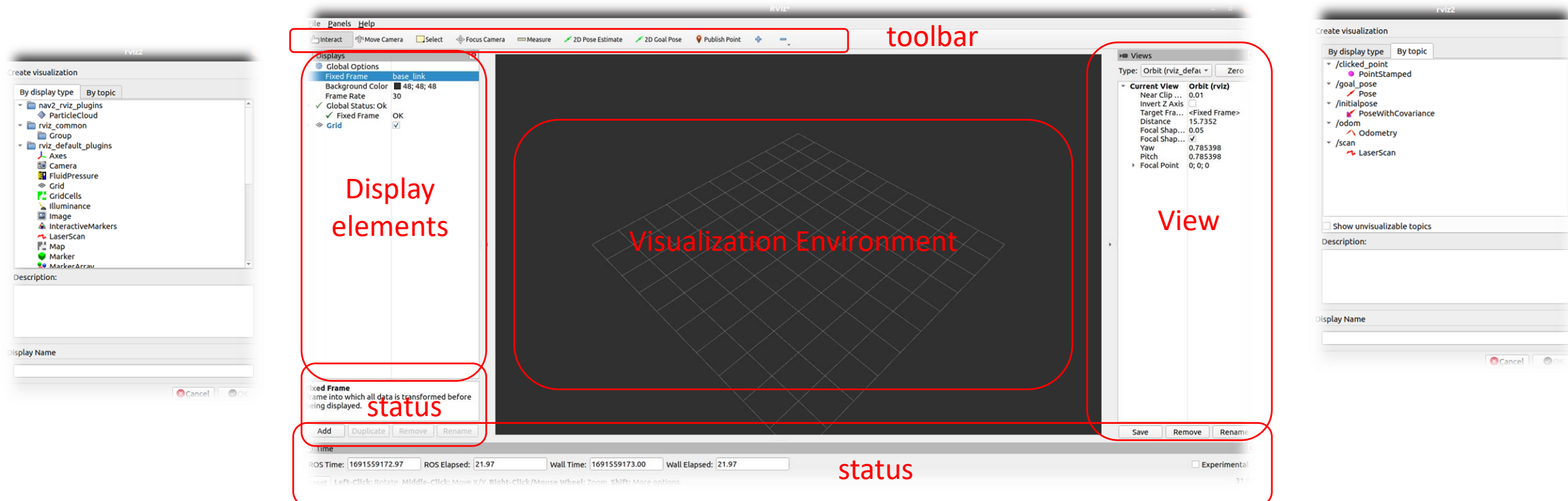
4.Depth: 10

1. The depth parameter indicates how many samples (messages) are stored in the history buffer. In this case, up to 10 recent messages will be stored.

ROS tutorial – RVIZ



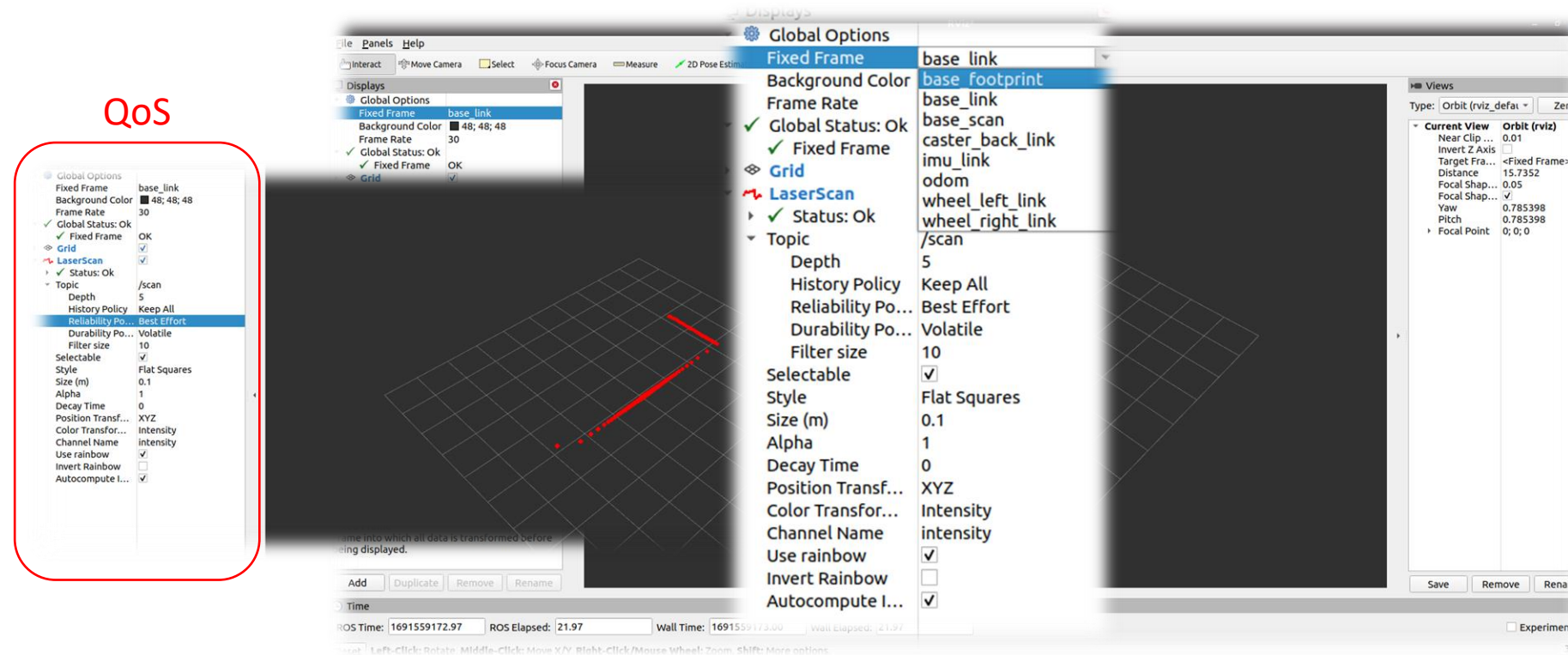
- Stands for Ros-VisualIZation
- A visualization application that is written in Qt, and can help us to visualize information on topics.
- Qt is a cross-platform C++/(has Python wrapper too) framework for developing graphical user interfaces (GUIs) and applications.



ROS tutorial – RVIZ



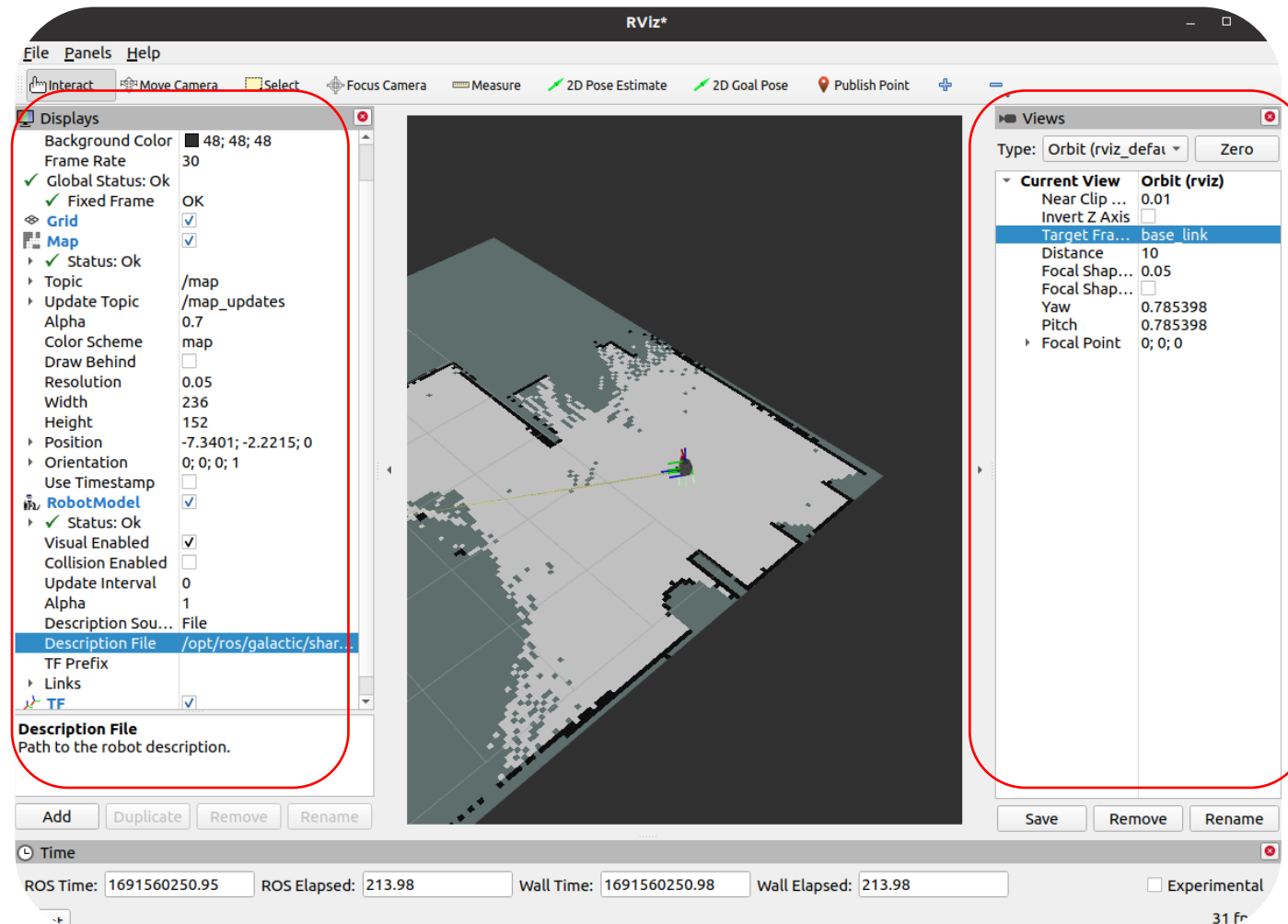
○ Cont'd



ROS tutorial – RVIZ



○ Cont'd

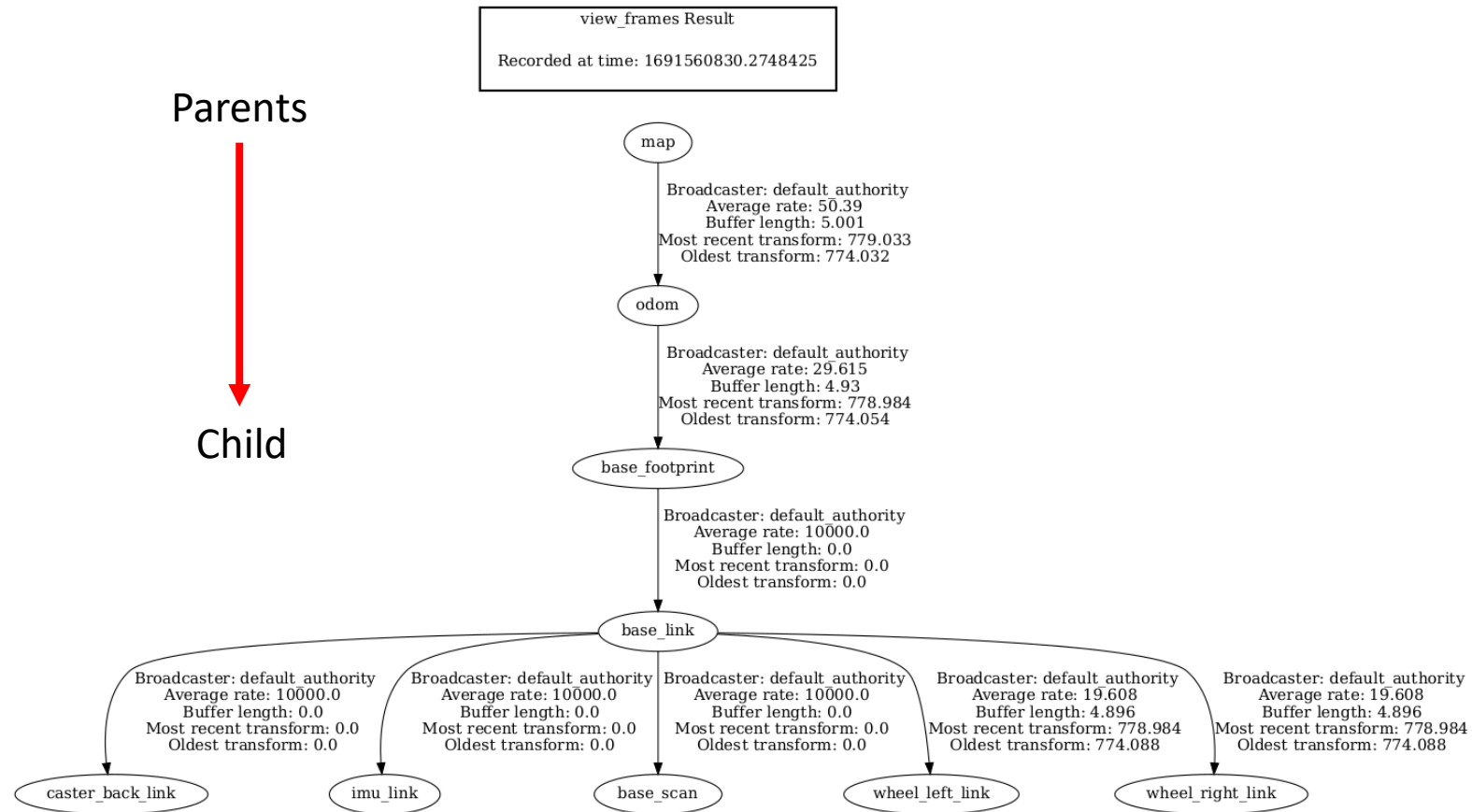


ROS tutorial – tf tree



- The "**tf tree**," short for "**transform tree**," is a fundamental concept in robotics and 3D spatial reasoning.
- It represents a **hierarchical structure** of coordinate frame transformations that describe the spatial relationships between various objects or sensors in a robotic system.
- The **tf tree** helps maintain a **consistent reference frame** for different parts of a robot or its environment, enabling accurate data fusion and coordination.
- It's crucial for tasks like **localization**, **mapping**, and **navigation**, ensuring that data from different sensors can be **properly aligned and interpreted**.

ROS tutorial – tf tree



ROS tutorial – tf tree, some points



- A **child** frame cannot have **two parents**.
- A **parent**, can have **multiple children**.
- The transformations are stamped. So, we can receive the changes in an accurate place and time.
- The process of creating a **tf-tree** is through calibration.
- If there is a path from a child-x to child-y, the tf-toolbox can give us the **transformation (p,q)**.
- This makes the **projections, and transformations** in general very easy.
- The tf-tree is a critical tool used by **rviz for rendering**.

ROS tutorial – soft synchronization



- In applications, we often want to receive **synchronized messages**.
- This can be carried out with hard synchronization; meaning that we can manage to have **hardware triggers activated** on sensors, and then send the hardware trigger to different sensors.
- E.g. we can use **GPS trigger signal to activate cameras and Lidar** in the same time.
- This concept is out of scope for this course.
- One easy way to do it, is through stamping messages, and then doing software **synchronization**.
- Ros provide a library to do so, through **message_filters** namespace.

ROS tutorial – soft synchronization



- For instance, consider we want to fuse **IMU** and **Encoder** data.
- We go ahead with `“import message_filters”`
- Then we define, the subscribers to the topics we want, here the imu and odometry topics, `“self.odom_sub=message_filters.Subscriber(self,odom,”/odom”,qos_profile=odom_qos)”`, and `“self.imu_sub=message_filters.Subscriber(self, Imu, ”/imu”, qos_profile=odom_qos)”`
- Then we define, **timeSynchronizer** object as `time_syncher=message_filters.ApproximateTimeSynchronizer([self.odom_sub, self.imu_sub], queue_size=10, slop=0.1)`. That slop is the amount of delay to synchronize messages between.
- `time_syncher.registerCallback(self.fusion_callback)`, then register your callback

ROS tutorial – services



- services are a communication mechanism that allows nodes to request and receive specific pieces of data or perform actions from other nodes in a synchronous manner.
- It is a request/response manner of communication.

- It consists of,

Service Client (Client Node):

- The service client is a node that sends a request to a service server node and waits for a response.
- The client specifies the service name and the request message type when creating a service request

Service Server (Server Node):

- The service server is a node that provides a service and handles incoming requests from service clients.
- The server specifies the service name, request message type, and response message type when creating a service

Service Type:

- A service type is defined by a pair of message types: one for the request and another for the response. The request message specifies what the client is asking for, and the response message carries the result of the request.

ROS tutorial – services, CLI



- First, to see all the available services, “`ros2 service list`”
- To call a service “`ros2 service call SERVICE_NAME SERVICE_TYPE SERVICE_CONTENT`”
- To get the type of the service, `ros2 service type SERVICE_NAME`
- To get the info about a given service, “`ros2 service info SERVICE_NAME`”
- To get the service name matching the service type “`ros2 service find SERVICE_TYPE`”

Services are often used when we don't want constant polling of the information. We may just use the service, once, or inconsistently

ROS tutorial – actions



- We use action servers when we want to know a return of the status of the service. In short, we need the feedback, status updates, and the ability to cancel or preempt the task if needed. They offer a more sophisticated way of communication that is asynchronous and let us monitor the task progress.
- Same as servers they have “Action type”, “Action client”, “Action server”, but they also come with,
- Goal Handle; used to manage the progress of the task
- Action preemption, and cancellation

ROS tutorial – action, CLI



- First, to see all the available services, “`ros2 action list`”
- To call a service “`ros2 action send_goal ACTION_NAME ACTION_TYPE ACTION_CONTENT`”
- To get the type of the service, `ros2 action type ACTION_NAME`
- To get the info about a given service, “`ros2 action info ACTION_NAME`”
- To get the service name matching the service type “`ros2 action find ACTION_TYPE`”

ROS tutorial – comprehensive example



- Launch turtlebot3 gazebo simulation “`ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py`”
- Launch the rviz node, “`rviz2`”
- Launch the slam toolbox package, “`ros2 slam_toolbox slam_online_synch.launch.py`”
- Run the teleop, “`ros2 run teleop_keyboard_twist teleop_keyboard_twist`”
- Add the map to the rviz.
- Add the robot model to the rviz
- Map the environment by navigating the robot inside the teleop terminal.
- Save the map by calling ros2 service.