Lab 2 Report

Robotics Integration Group Project I

Yuwei ZHAO (23020036096)
Group #31 2025-11-14

Abstract

This report documents the setup and experimentation undertaken in Lab 2. The primary objectives are to install and configure the Robot Operating System (*ROS*) environment on Ubuntu, establish a functioning Catkin workspace, and gain practical experience with core ROS communication mechanisms. Through a sequence of hands-on tasks, the lab introduces essential ROS concepts such as node creation, publisher-subscriber communication, topic monitoring, and system visualization using tools like `rqt_graph` and `roscore`. The experiment provides a foundational understanding of how ROS facilitates modular, distributed robotics software development.

See Resources on github.com/RamessesN/Robotics_MIT.

1 Introduction

This laboratory session focuses on the installation, configuration and initial exploration of the Robot Operating System (*ROS*), which serves as the middleware framework for subsequent robotics development. Before implementing perception, planning or control modules, it is essential to understand the *ROS* architecture and its communication mechanisms. The experiment involves setting up the *ROS* environment, creating and managing a catkin workspace, and developing simple publisher and subscriber nodes to exchange data through topics.

2 Procedure

2.1 Part I

2.1.1 Objective

To set up the *ROS* environment on the *Ubuntu 24.04* operating system, including the installation of essential packages and the configuration of a functional Catkin workspace.

2.1.2 Methodology

The initial approach for installing *ROS Noetic* was as follows:

1. Add the *ROS* package sources and install *ROS Noetic*:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc)
↪  main" > /etc/apt/sources.list.d/ros-latest.list'
curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo
↪  apt-key add -
sudo apt update
sudo apt install ros-noetic-desktop-full
```

2. Configure the *ROS* environment and install additional dependencies:

```
echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc
source ~/.bashrc
sudo apt install python3-rosdep python3-rosinstall python3-rosinstall-generator
↪  \
python3-vcstool build-essential python3-catkin-tools python-is-python3
```

3. Initialize *rosdep* to enable dependency management:

```
sudo rosdep init
rosdep update
```

However, the above method is not fully compatible with *Ubuntu 24.04*. To address this, an alternative installation procedure using the *Shrike* repository was adopted, which provided a successful setup.

1. Clone the *Shrike* repository from GitHub:

```
git clone git@github.com:Minoic-Intelligence/shrike.git
```

2. Follow the instructions provided in the *Shrike* repository to complete the installation. The resulting terminal output is shown in Figure 1.

```
./scripts/install_ubuntu24.sh
./src/catkin/bin/catkin_make_isolated --install -DCMAKE_BUILD_TYPE=Release
source ./install_isolated/setup.bash
```

2.1.3 Observations

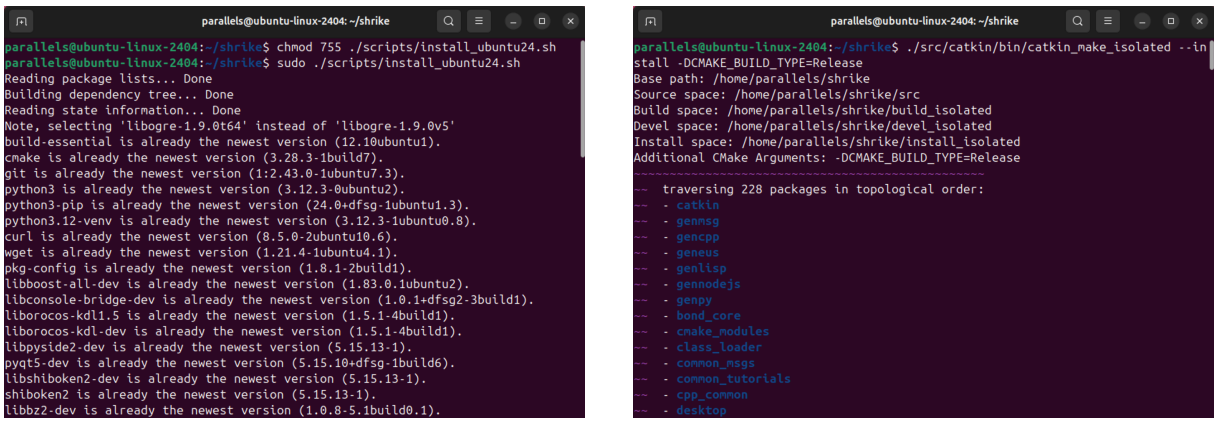


Figure 1: Installation of *ROS Noetic* on *Ubuntu 24.04*

2.1.4 Discussion

Installing *ROS Noetic* on *Ubuntu 24.04* presents several challenges due to incompatibilities between the standard Noetic packages and the latest system libraries. By utilizing the *Shrike* repository—which provides customized build scripts tailored for Ubuntu 24.04—the installation process was completed successfully. This outcome highlights the necessity of adapting traditional installation procedures to evolving system environments and demonstrates the value of community-maintained solutions when official support is unavailable.

2.2 Part II

2.2.1 Objective

To deepen the understanding of the *ROS* communication framework by implementing publisher and subscriber nodes, exploring topic-based message exchange, and visualizing the inter-node topology. This experiment establishes the foundation for modular, event-driven robotic behavior through:

1. Implementing *ROS* nodes that publish and subscribe to custom message types.
2. Demonstrating data flow between nodes via topics and verifying correct message delivery.
3. Utilizing diagnostic and visualization tools (such as `rqt_graph`) to examine node–topic connectivity.
4. Ensuring the workspace is properly configured to build, launch, and manage multiple nodes within a *ROS* ecosystem.

2.2.2 Methodology

- ***ROS* Master**
  Start the *ROS* Master using the following command. The resulting output is shown in Figure 2.

  ```
  roscore
  ```

- ***ROS* Nodes**

  1. Launch the *turtlesim* node in a new terminal:
     ```
     rosrun turtlesim turtlesim_node
     ```
  2. List active nodes using:
     ```
     rosnode list
     ```
  3. Launch the *turtle_teleop_key* node to control the turtle with keyboard input:
     ```
     rosrun turtlesim turtle_teleop_key
     ```
  4. The corresponding runtime output is illustrated in Figure 3.

- ***ROS* Topics**

1. Visualize the communication graph of running nodes and topics:

   ```
   rosrun rqt_graph rqt_graph
   ```

2. Monitor velocity commands sent to the turtle in real time:

   ```
   rostopic echo /turtle1/cmd_vel
   ```

3. Create a simple *C++* example node:

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

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

    ros::Rate loop_rate(50);

    while (ros::ok()) {
        ros::spinOnce();
        loop_rate.sleep();
    }
    return 0;
}
```

To compile and execute this example using *Shrike*, follow these steps:

```
cd ~/shrike/ros_ws/src
catkin_create_pkg ros_sample roscpp std_msgs
```

The resulting workspace structure is:

```
- ros_ws/
  - src/
    - ros_sample/
      - CMakeLists.txt
      - include/
      - package.xml
      - src/
        - ros_sample.cpp
```

Replace *example_node.cpp* with *ros_sample.cpp* and update the *CMakeLists.txt* file as follows:

```
cmake_minimum_required(VERSION 3.0.2)
project(ros_sample)

find_package(catkin REQUIRED COMPONENTS
  roscpp
  std_msgs
)

catkin_package()

include_directories(
  ${catkin_INCLUDE_DIRS}
)

add_executable(ros_sample_node src/ros_sample.cpp)
target_link_libraries(ros_sample_node ${catkin_LIBRARIES})
```

Build and source the workspace:

```
cd ~/shrike/ros_ws
```

```
catkin_make_isolated --install -DCMAKE_BUILD_TYPE=Release
source ~/shrike/ros_ws/devel_isolated/setup.bash
rosrun ros_sample ros_sample_node
```

Verify that the node is active using `rosnode list`. The corresponding results are shown in Figure 4.

4. ***TF* Tools**
   - Launch the Turtle TF demo:
     `roslaunch turtle_tf turtle_tf_demo.launch`
     Note: On Ubuntu 24.04 with *Shrike*, older Noetic packages may depend on Python 2, while only Python 3 is installed. To resolve this compatibility issue, create a symbolic link:
     `sudo ln -s /usr/bin/python3 /usr/bin/python`
   - Visualize the TF tree using:
     `rosrun rqt_tf_tree rqt_tf_tree`
     The visualization shows three coordinate frames: *world* (parent), *turtle1*, and *turtle2*.
   - Inspect transformations in real time:
     `rosrun tf tf_echo /turtle1 /turtle2`
   - The resulting output is illustrated in Figure 5.
5. **Using *RViz***
   Run the following command to launch *RViz*:
   `LIBGL_ALWAYS_SOFTWARE=1 rviz`
   Software rendering is required because *RViz* relies on OpenGL, which may not be fully supported in virtualized environments. The visualization result is shown in Figure 6.

Notice: Modify the *view_frames* script for Python 3.x compatibility:

```
try:
    vstr = subprocess.Popen(args, stdout=subprocess.PIPE,
                            stderr=subprocess.STDOUT).communicate()[0]
    vstr = vstr.decode('utf-8')
except OSError as ex:
    print("Warning: Could not execute `dot -V`. Is graphviz installed?")
    sys.exit(-1)

v = distutils.version.StrictVersion('2.16')
r = re.compile(r".*version (\d+\.?\d*)")
print(vstr)
m = r.search(vstr)
```

2.2.3 Observations

- *ROS* Master

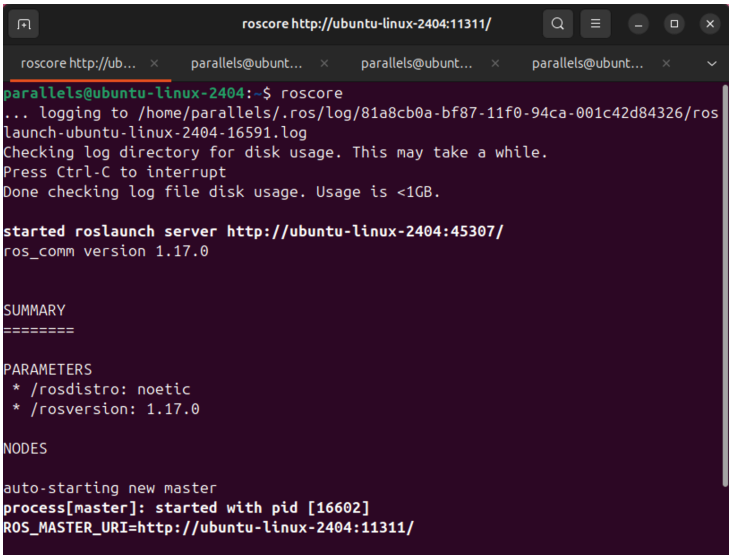
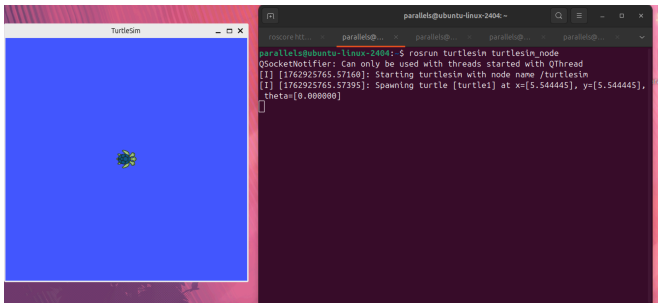
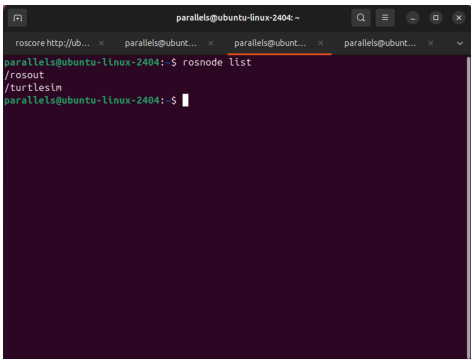


Figure 2: ROS Master

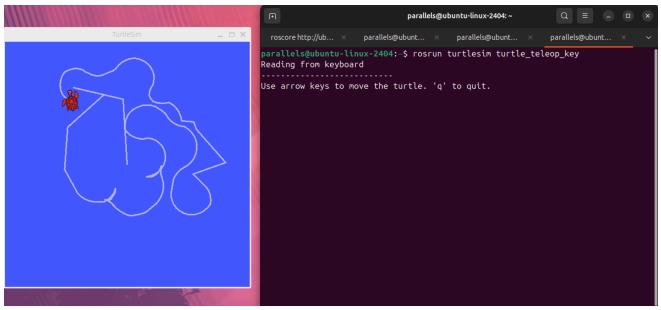
- *ROS* Nodes



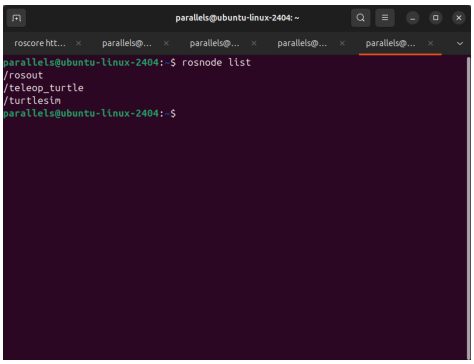
(a) Turtlesim node running



(b) rosnode list output



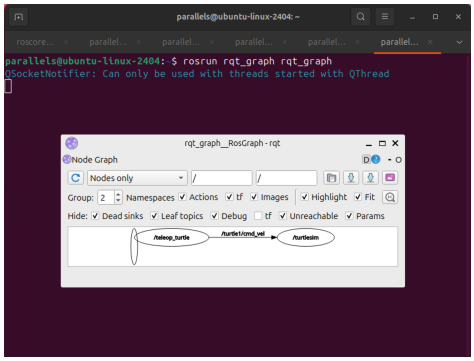
(c) Turtle teleop node running



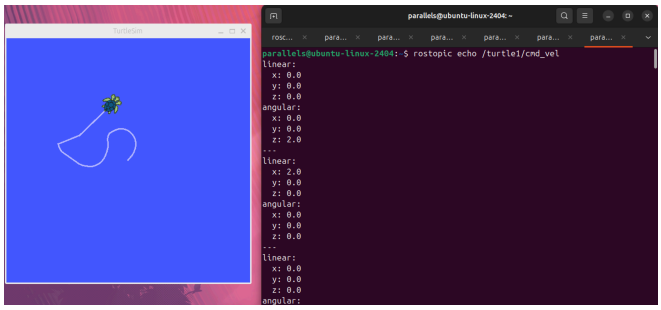
(d) rosnode list output

Figure 3: ROS nodes

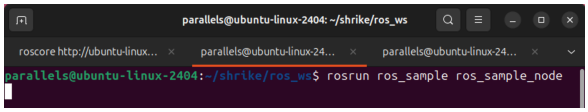
- *ROS* Topics



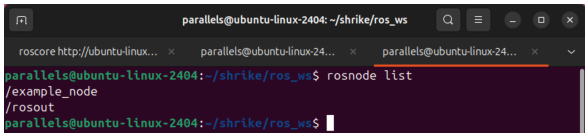
(a) rqt_graph output



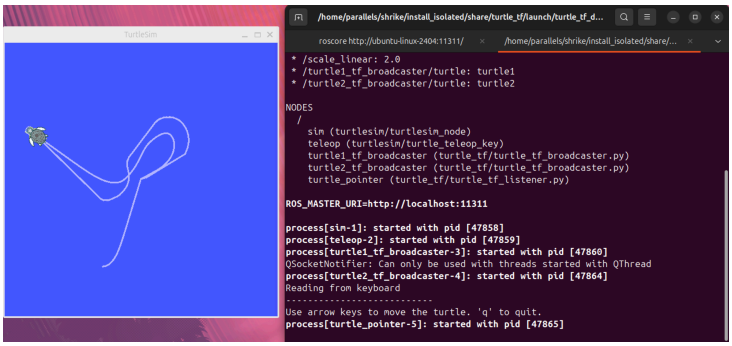
(b) Velocity topic output



(c) ROS node running

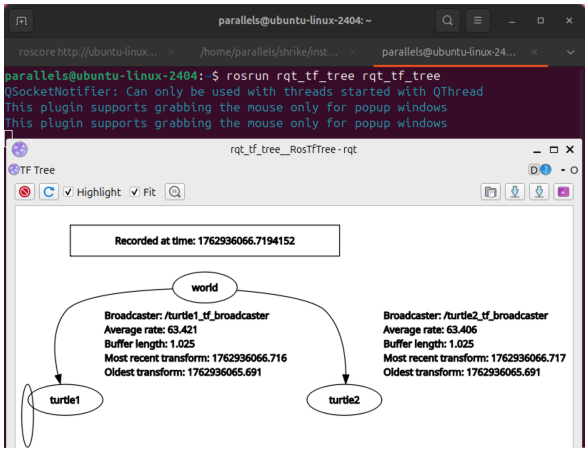


(d) rosnode list output

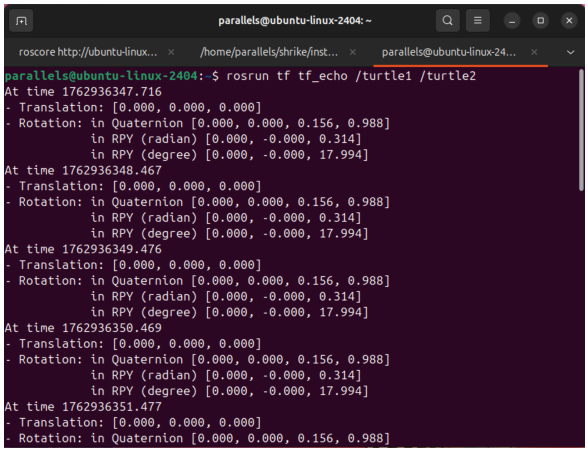


(e) Roslaunch output

Figure 4: ROS topics



(a) rqt_tf_tree output



(b) tf_echo output

Figure 5: TF tools

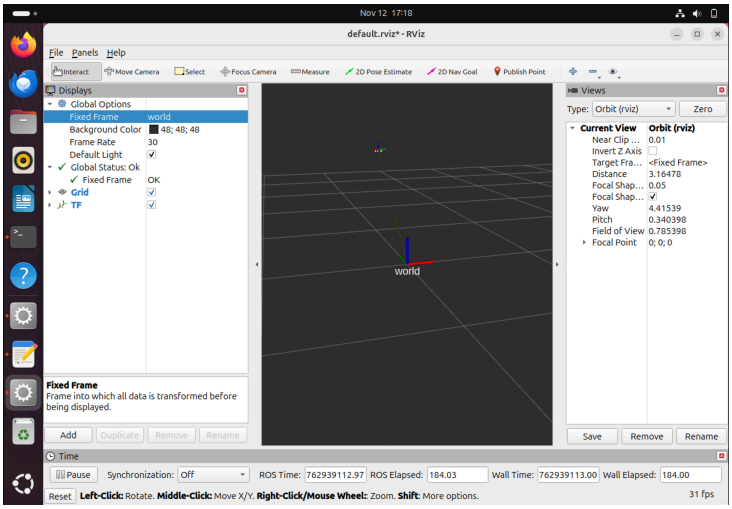


Figure 6: RViz

2.2.4 Discussion

This section demonstrates the core principles of inter-node communication and system visualization within the *ROS* framework. By implementing publisher and subscriber nodes and observing message flow using *rqt_graph* and *rostopic echo*, it becomes clear how topic-based communication enables modular and decoupled node interaction. The ability to monitor live message streams allows developers to validate correct behavior and quickly identify configuration or runtime issues.

The use of *TF* frames further illustrates the importance of coordinate transformation management in robotic systems. The hierarchical relationship among the *world*, *turtle1*, and *turtle2* frames highlights how spatial relationships are maintained and broadcast across the *ROS* network. Tools such as *rqt_tf_tree* and *tf_echo* provide intuitive means of visualizing and debugging frame hierarchies, reinforcing understanding of kinematic relationships and transformation propagation.

Running *RViz* in a virtualized environment emphasizes practical considerations for deploying visualization tools. The need to enforce software rendering with `LIBGL_ALWAYS_SOFTWARE=1` underscores how hardware acceleration and OpenGL support directly affect performance and usability.

2.3 Part III

2.3.1 Objective

To utilize the *TF* package to control the movement of drones and visualize their trajectories dynamically in *RViz*.

2.3.2 Methodology

1. **Setting up the workspace**
   First, create a Catkin workspace named *vnav_ws* under the home directory:

   ```
   mkdir -p ~/vnav_ws/src
   cd ~/vnav_ws/
   catkin init
   ```

   The output should appear as shown in Figure 7. Next, copy the *two_drones_pkg* package into the *src* folder:

   ```
   cp -a /source_path/two_drones_pkg ~/vnav_ws/src
   ```

   Then, build the workspace using:

   ```
   catkin build
   ```

   The build output is shown in Figure 7. After the build completes, source the setup file to refresh the environment:

   ```
   source devel/setup.bash
   ```

   Finally, launch *ROS* to visualize the static drone scene in *RViz*:

   ```
   roslaunch two_drones_pkg two_drones.launch static:=True
   ```

   After adding the *TF* display, three colored axes will appear for each coordinate frame, as illustrated in Figure 8. Remember to save the *RViz* configuration once the setup is complete.

2. **Basic *ROS* commands**

   - **Listing active nodes**
     The active nodes in the static scene can be verified via *rqt_graph* and the output includes:

     ```
     /frames_publisher_node
     /plots_publisher_node
     /rviz
     ```

   - **Running without roslaunch**
     The static scene can also be launched manually using *rosrun*:

     ```
     rosrun two_drones_pkg frames_publisher_node
     rosrun two_drones_pkg plots_publisher_node
     rosrun rviz rviz -d path/to/rviz_config.rviz
     ```

   - **Node–topic relationships**
     The topic connections between nodes are summarized below:

Node	Publishes	Subscribes	Description
frames_publisher_node	/tf	N/A	Broadcasts the coordinate transformations of AV1 and AV2
plots_publisher_node	/visuals	/tf	Generates trajectory *Marker*s and calculates AV2's position relative to AV1
rviz	N/A	/visuals, /tf	Visualizes the drone grid and trajectories in 3D

- **Effect of omitting *static:=True***
  The *static* argument determines whether *TF* uses a static transform broadcaster. If omitted, transforms will be published dynamically at each iteration, which in dynamic scenes can increase message traffic and slightly reduce performance.

3. **Dynamic scene visualization**
   To simulate drone movement, a `TransformBroadcaster` is instantiated to periodically send transform messages:

   ```
   tf2_ros::TransformBroadcaster tf_broadcaster;
   ```

   The elapsed time since startup is computed as:

   ```
   double time = (ros::Time::now() - startup_time).toSec();
   ```

   Each drone has an associated `TransformStamped` message:

   ```
   geometry_msgs::TransformStamped AV1World;
   geometry_msgs::TransformStamped AV2World;
   AV1World.transform.rotation.w = 1.0;
   AV2World.transform.rotation.w = 1.0;
   ```

   AV1 moves along a circular path on the xy-plane centered at the world origin:

   ```
   // --- AV1 ---
   AV1World.header.frame_id = "world";
   AV1World.child_frame_id = "av1";
   AV1World.transform.translation.x = cos(time);
   AV1World.transform.translation.y = sin(time);
   AV1World.transform.translation.z = 0.0;

   tf2::Quaternion q1;
   q1.setRPY(0, 0, time);
   AV1World.transform.rotation.x = q1.x();
   AV1World.transform.rotation.y = q1.y();
   AV1World.transform.rotation.z = q1.z();
   AV1World.transform.rotation.w = q1.w();
   ```

   AV2 follows a sinusoidal path along the x–z plane:

   ```
   // --- AV2 ---
   AV2World.header.frame_id = "world";
   AV2World.child_frame_id = "av2";
   AV2World.transform.translation.x = sin(time);
   AV2World.transform.translation.y = 0.0;
   AV2World.transform.translation.z = cos(2 * time);
   ```

   Finally, both transforms are broadcast:

   ```
   tf_broadcaster.sendTransform(AV1World);
   tf_broadcaster.sendTransform(AV2World);
   ```

   The complete source code for `frames_publisher_node.cpp` is provided in Source Code at the end of this report.

4. **Trajectory visualization**
   To visualize the drones' trajectories, *TF* data is queried using `lookupTransform` to retrieve each drone's position relative to the world frame:

   - *tf_buffer* stores transformation history between coordinate frames.
   - *lookupTransform(ref_frame, dest_frame, ros::Time(0))* retrieves the latest pose.
   - *ros::Duration(0.1)* ensures synchronization by waiting briefly if needed.

   Besides, with *Duration(0.1)* exists but there is no other thread to process the messages from *TF*. Therefore, the transform buffer will never be updated unless a dedicated thread or asynchronous spinner is created to handle incoming *TF* messages. In *tf2_ros*, this can be achieved by enabling the internal thread of the *TransformListener*, which continuously listens to the TF topic and updates the buffer in the background, as shown below:

   ```
   tf2_ros::Buffer tf_buffer{ros::Duration(10.0), true};
   tf2_ros::TransformListener tf_listener{tf_buffer};
   ```

   Here, the second parameter `true` tells the listener to spawn a dedicated thread to process incoming TF data asynchronously. Without this background thread, the buffer would remain empty, and any subsequent calls to `lookupTransform()` would fail with a time extrapolation or "transform not found" error.

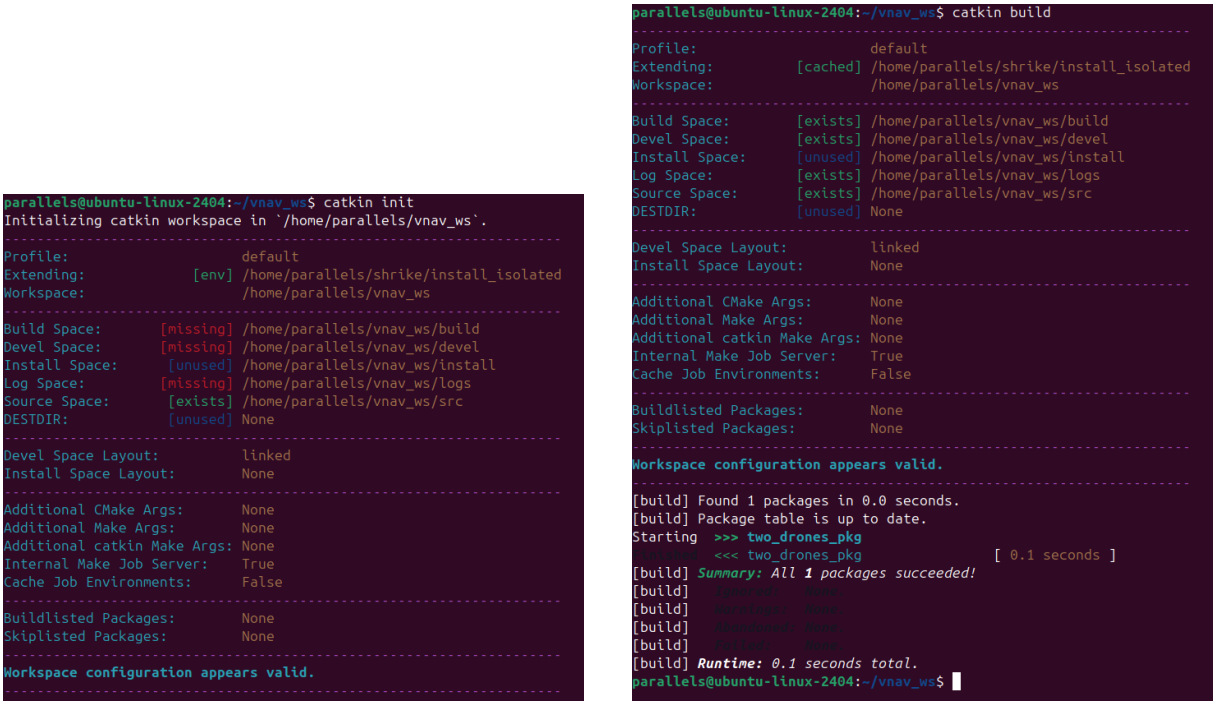
   These steps ensure that *RViz* dynamically updates both trajectories in real time. The resulting visualization is shown in Figure 9.

5. **Mathematical derivations**

   - In the problem formulation ,we mentioned that AV2's trajectory is an arc of parabola in the x-z plane of the world frame. Can you prove this statement?
   - Compute $o_2^1(t)$, i.e., the position of AV2 relative to AV1's body frame as a function of t .
   - Show that $o_2^1(t)$ describes a planar curve and find the equation of its plane Π.
   - Rewrite the above trajectory explicitly using a 2D frame of reference (x_p, y_p) on the plane found before. Try to ensure that the curve is centered at the origin of this 2D frame and that x_p , y_p are axes of symmetry for the curve.
   - Using the expression of $o_2^p(t)$, prove that the trajectory of AV2 relative to AV1 is an ellipse and compute the lengths of its semi-axes.

6. **More properties of quaternions**

2.3.3 Observations



(a) catkin init (b) catkin build

Figure 7: Catkin workspace initialization and build

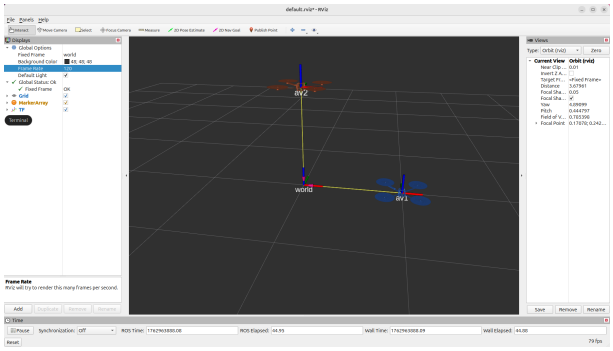


Figure 8: Static visualization of two drones in *RViz*

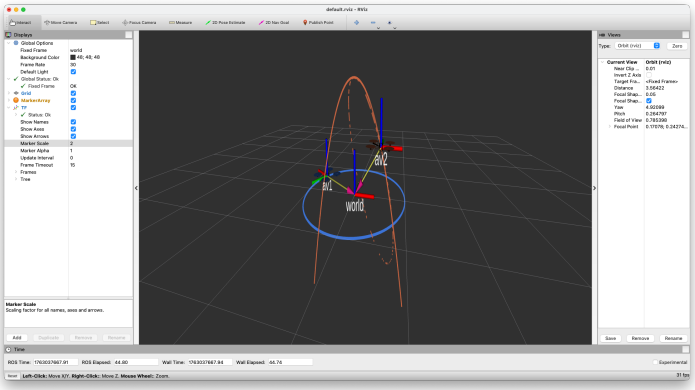


Figure 9: Dynamic trajectories of two drones in *RViz*

2.3.4 Discussion

This session successfully demonstrates how *TF* enables both static and dynamic spatial relationships to be visualized in real time. In the **static** case, each transform is broadcast only once, minimizing system overhead. In contrast, **dynamic** broadcasting continuously updates transforms according to the elapsed time, allowing for smooth motion visualization but at a higher computational and communication cost.

The design of the two distinct motion models—circular motion for AV1 and sinusoidal motion for AV2—illustrates the flexibility of *TF* in representing diverse kinematic behaviors. By combining *tf2_ros::TransformBroadcaster* with *lookupTransform*, the system maintains synchronization between frames and ensures that *RViz* accurately reflects the relative positions and orientations of both drones.

3 Reflection and Analysis

4 Conclusion

5 Source Code

- *frames_publisher_node.cpp*

```cpp
#include <ros/ros.h>
#include <tf2_ros/transform_broadcaster.h>
#include <tf2/LinearMath/Quaternion.h>

#include <iostream>

class FramesPublisherNode {
    private:
    ros::NodeHandle nh;
    ros::Time startup_time;

    ros::Timer heartbeat;

    tf2_ros::TransformBroadcaster tf_broadcaster;

    public:
    FramesPublisherNode() {
        // NOTE: This method is run once, when the node is launched.
        startup_time = ros::Time::now();
        heartbeat = nh.createTimer(ros::Duration(0.02),
        ↪  &FramesPublisherNode::onPublish, this);
        heartbeat.start();
    }

    void onPublish(const ros::TimerEvent&) {
        double time = (ros::Time::now() - startup_time).toSec();

        geometry_msgs::TransformStamped AV1World;
        geometry_msgs::TransformStamped AV2World;

        AV1World.transform.rotation.w = 1.0;
        AV2World.transform.rotation.w = 1.0;

        // --- AV1 ---
        AV1World.header.stamp = ros::Time::now();
        AV1World.header.frame_id = "world";
        AV1World.child_frame_id = "av1";
        // path: [cos(t), sin(t), 0]
        AV1World.transform.translation.x = cos(time);
        AV1World.transform.translation.y = sin(time);
        AV1World.transform.translation.z = 0.0;
        // angle: [0, 0, t]
        tf2::Quaternion q1;
        q1.setRPY(0, 0, time);
        AV1World.transform.rotation.x = q1.x();
        AV1World.transform.rotation.y = q1.y();
        AV1World.transform.rotation.z = q1.z();
        AV1World.transform.rotation.w = q1.w();

        // --- AV2 ---
        AV2World.header.stamp = ros::Time::now();
        AV2World.header.frame_id = "world";
        AV2World.child_frame_id = "av2";
        // path: [sin(t), 0, cos(2t)]
```

```
        AV2World.transform.translation.x = sin(time);
        AV2World.transform.translation.y = 0.0;
        AV2World.transform.translation.z = cos(2 * time);

        // Publish the transforms
        tf_broadcaster.sendTransform(AV1World);
        tf_broadcaster.sendTransform(AV2World);
    }
};

int main(int argc, char** argv) {
    ros::init(argc, argv, "frames_publisher_node");
    FramesPublisherNode node;
    ros::spin();
    return 0;
}
```

- *plots_publisher_node.cpp*

```
#include <geometry_msgs/Point.h>
#include <ros/ros.h>
#include <tf2_ros/transform_listener.h>
#include <visualization_msgs/MarkerArray.h>

#include <iostream>
#include <list>

class PlotsPublisherNode {
  ros::Time startup_time;
  ros::Timer heartbeat;
  ros::NodeHandle nh;
  ros::Publisher markers_pub;
  tf2_ros::Buffer tf_buffer{ros::Duration(10.0), true};
  tf2_ros::TransformListener tf_listener{tf_buffer};
  int num_trails;

  class TrajTrail {
    PlotsPublisherNode* parent;
    static int id;
    std::list<geometry_msgs::Point> poses;
    std::string ref_frame, dest_frame;
    size_t buffer_size;

    std::string ns;
    float r, g, b, a;

    visualization_msgs::Marker marker_out;

    void update() {
      // NOTE: you need to populate this transform
      geometry_msgs::TransformStamped transform;
      try {
        transform = parent->tf_buffer.lookupTransform(
          ref_frame,
          dest_frame,
          ros::Time(0),
          ros::Duration(0.1)
        );

```

```
41          while (poses.size() >= buffer_size) {
42            poses.pop_front();
43          }
44
45          geometry_msgs::Point tmp;
46          tmp.x = transform.transform.translation.x;
47          tmp.y = transform.transform.translation.y;
48          tmp.z = transform.transform.translation.z;
49          poses.push_back(tmp);
50        } catch (const tf2::TransformException& ex) {
51          ROS_ERROR_STREAM("transform lookup failed: " << ex.what());
52        }
53      }
54
55    public:
56      TrajTrail() : parent(nullptr){};
57
58      TrajTrail(PlotsPublisherNode* parent_,
59                const std::string& ref_frame_,
60                const std::string& dest_frame_,
61                int buffer_size_ = 160)
62          : parent(parent_),
63            ref_frame(ref_frame_),
64            dest_frame(dest_frame_),
65            buffer_size(buffer_size_) {
66        if (buffer_size <= 0) {
67          ROS_ERROR_STREAM("invalid buffer size! defaulting to 10");
68          buffer_size = 10;
69        }
70
71        marker_out.header.frame_id = ref_frame;
72        marker_out.ns = "trails";
73        marker_out.id = parent->num_trails++;
74        marker_out.type = visualization_msgs::Marker::LINE_STRIP;
75        marker_out.action = visualization_msgs::Marker::ADD;
76        marker_out.color.a = 0.8;
77        marker_out.scale.x = 0.02;
78        marker_out.lifetime = ros::Duration(1.0);
79      }
80
81      void setColor(float r_, float g_, float b_) {
82        marker_out.color.r = r_;
83        marker_out.color.g = g_;
84        marker_out.color.b = b_;
85      }
86
87      void setNamespace(const std::string& ns_) { marker_out.ns = ns_; }
88
89      void setDashed() { marker_out.type = visualization_msgs::Marker::LINE_LIST;
        ↪  }
90
91      visualization_msgs::Marker getMarker() {
92        update();
93        marker_out.header.stamp = ros::Time::now();
94        marker_out.points.clear();
95        for (auto& p : poses) marker_out.points.push_back(p);
96
97        if (marker_out.type == visualization_msgs::Marker::LINE_LIST &&
        ↪  poses.size() % 2)
```

```
98          marker_out.points.resize(poses.size() - 1);
99
100       return marker_out;
101     }
102   };
103
104   TrajTrail av1trail;
105   TrajTrail av2trail;
106   TrajTrail av2trail_rel;
107
108 public:
109   PlotsPublisherNode() : tf_listener(tf_buffer), num_trails(0) {
110     startup_time = ros::Time::now();
111     markers_pub = nh.advertise<visualization_msgs::MarkerArray>("visuals", 0);
112     heartbeat =
113         nh.createTimer(ros::Duration(0.02), &PlotsPublisherNode::onPublish,
          ↪  this);
114     heartbeat.start();
115     av1trail = TrajTrail(this, "world", "av1", 300);
116     av1trail.setColor(0.25, 0.52, 1.0);
117     av1trail.setNamespace("Trail av1-world");
118     av2trail = TrajTrail(this, "world", "av2", 300);
119     av2trail.setColor(0.8, 0.4, 0.26);
120     av2trail.setNamespace("Trail av2-world");
121     av2trail_rel = TrajTrail(this, "av1", "av2", 160);
122     av2trail_rel.setDashed();
123     av2trail_rel.setColor(0.8, 0.4, 0.26);
124     av2trail_rel.setNamespace("Trail av2-av1");
125
126     ROS_INFO_STREAM("Waiting for av1 and av2 transforms to be broadcast...");
127     while (ros::ok()) {
128       const bool av1_present = tf_buffer.canTransform("av1", "world",
          ↪  ros::Time(0));
129       const bool av2_present = tf_buffer.canTransform("av2", "world",
          ↪  ros::Time(0));
130       if (av1_present && av2_present) {
131         ROS_INFO_STREAM("Necessary frames are present, starting!");
132         break;
133       }
134     }
135   }
136
137   void onPublish(const ros::TimerEvent&) {
138     visualization_msgs::MarkerArray visuals;
139     visuals.markers.resize(2);
140     visualization_msgs::Marker& av1(visuals.markers[0]);
141     visualization_msgs::Marker& av2(visuals.markers[1]);
142     av1.header.frame_id = "av1";
143     av1.ns = "AVs";
144     av1.id = 0;
145     av1.header.stamp = ros::Time();
146     av1.type = visualization_msgs::Marker::MESH_RESOURCE;
147     av1.mesh_resource = "package://two_drones_pkg/mesh/quadrotor.dae";
148     av1.action = visualization_msgs::Marker::ADD;
149     av1.pose.orientation.w = 1.0;
150     av1.scale.x = av1.scale.y = av1.scale.z = av1.color.a = 1.0;
151     av1.color.r = 0.25;
152     av1.color.g = 0.52;
153     av1.color.b = 1.0;
```

```
154     av1.lifetime = ros::Duration(1.0);
155     av2 = av1;
156     av2.header.frame_id = "av2";
157     av2.ns = "AVs";
158     av2.id = 1;
159     av2.color.r = 0.8;
160     av2.color.g = 0.4;
161     av2.color.b = 0.26;
162 
163     // Trails
164     visuals.markers.push_back(av1trail.getMarker());
165     visuals.markers.push_back(av2trail.getMarker());
166     visuals.markers.push_back(av2trail_rel.getMarker());
167     markers_pub.publish(visuals);
168   }
169 };
170 
171 int main(int argc, char** argv) {
172   ros::init(argc, argv, "plots_publisher_node");
173   PlotsPublisherNode node;
174   ros::spin();
175   return 0;
176 }
```