

# 23. ROS2 TF2 Coordinate Transformation

## 1. Introduction to TF2

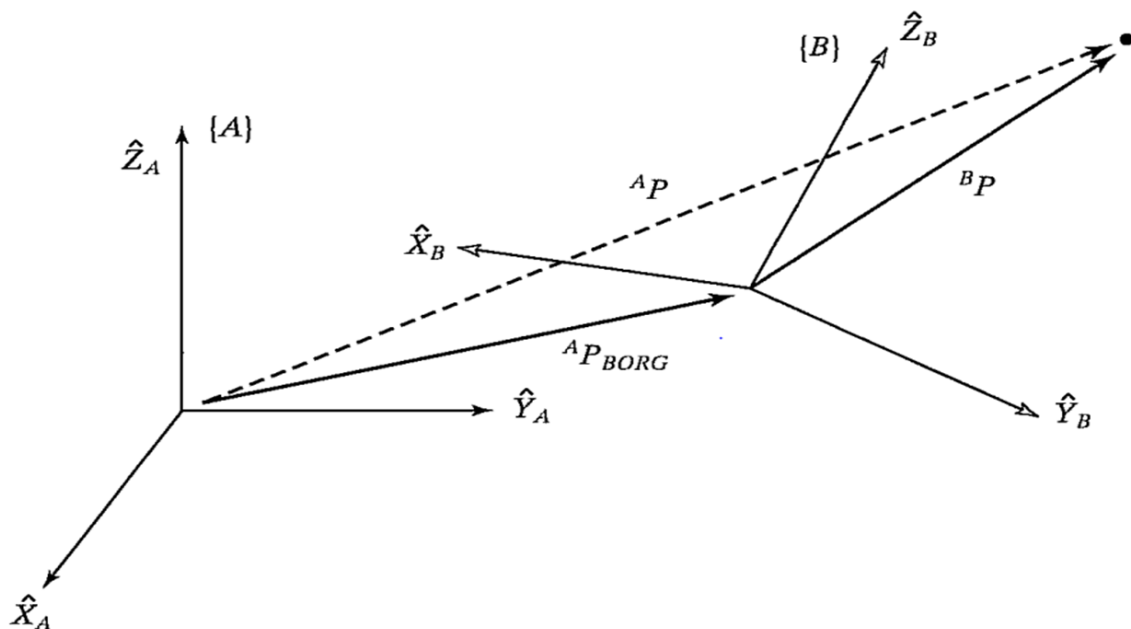
Coordinate systems are a very familiar concept and a key foundation in robotics. In a complete robotic system, there are many coordinate systems. How do we manage the positional relationships between these coordinate systems? ROS provides a powerful coordinate system management tool: TF2.

TF System Reference: [tf: The transform library | IEEE Conference Publication | IEEE Xplore](#)

## 2. Coordinate Systems in Robotics

Coordinate systems are also crucial in mobile robot systems. For example, the center point of a mobile robot is the base coordinate system (Base Link), and the position of the radar is called the laser link. As the robot moves, the odometry accumulates its position. The reference system for this position is called the odom coordinate system (odom). The odometry itself is subject to accumulated errors and drift. The reference system for absolute position is called the map coordinate system (map).

The relationships between layers of coordinate systems are complex. Some are relatively fixed, while others are constantly changing. Even seemingly simple coordinate systems can become complex within a spatial context, making a good coordinate system management system extremely important.



The basic theory of coordinate system transformations is explained in every robotics textbook. It can be broken down into two components: translation and rotation. These components are described by a four-by-four matrix. When plotting a coordinate system in space, the transformation between these two components is essentially a mathematical description of vectors.

The underlying principles of the TF functionality in ROS encapsulate these mathematical transformations. For detailed theoretical knowledge, please refer to robotics textbooks. We will primarily explain how to use the TF coordinate management system.

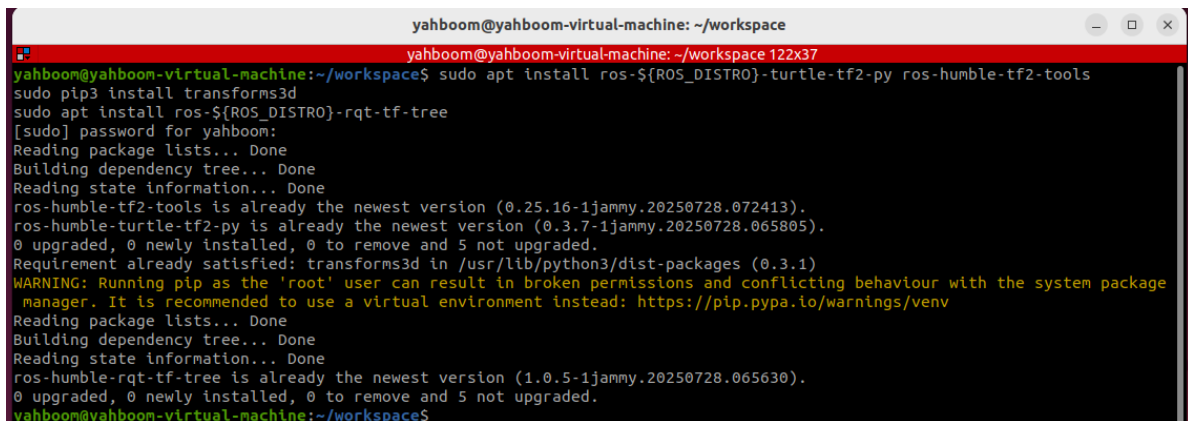
## 3. TF Command Line Operations

Let's first use an example with two turtles to understand a robot following algorithm based on coordinate systems. **For ease of demonstration, this lesson is best conducted in a virtual machine**

### 3.1 Installing Related Tools

This example requires installing the corresponding function packages, the TF turtle simulator example, and the TF tree visualization tool.

```
sudo apt install ros-${ROS_DISTRO}-turtle-tf2-py ros-humble-tf2-tools
sudo pip3 install transforms3d
sudo apt install ros-${ROS_DISTRO}-rqt-tf-tree
```

A terminal window titled 'yahboom@yahboom-virtual-machine: ~/workspace' showing the execution of installation commands. The user runs 'sudo apt install ros-\${ROS\_DISTRO}-turtle-tf2-py ros-humble-tf2-tools', followed by 'sudo pip3 install transforms3d', and finally 'sudo apt install ros-\${ROS\_DISTRO}-rqt-tf-tree'. The terminal output shows package lists being read, dependency trees being built, and state information being read. It confirms that 'ros-humble-tf2-tools' is the newest version (0.25.16-1jammy.20250728.072413), 'ros-humble-turtle-tf2-py' is already the newest version (0.3.7-1jammy.20250728.065805), and 'ros-humble-rqt-tf-tree' is already the newest version (1.0.5-1jammy.20250728.065630). A warning message from pip is also visible: 'WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system package manager. It is recommended to use a virtual environment instead: https://pip.pypa.io/warnings/venv'.

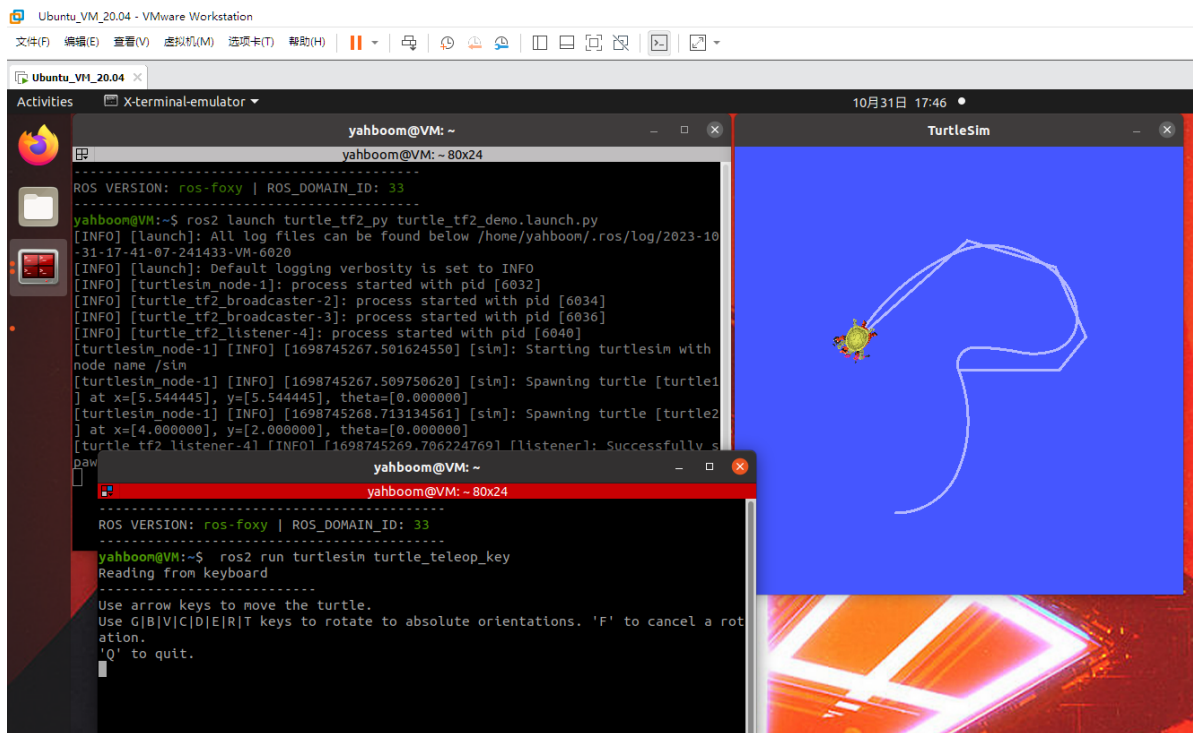
```
yahboom@yahboom-virtual-machine: ~/workspace
yahboom@yahboom-virtual-machine: ~/workspace 122x37
yahboom@yahboom-virtual-machine:~/workspace$ sudo apt install ros-${ROS_DISTRO}-turtle-tf2-py ros-humble-tf2-tools
sudo pip3 install transforms3d
sudo apt install ros-${ROS_DISTRO}-rqt-tf-tree
[sudo] password for yahboom:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
ros-humble-tf2-tools is already the newest version (0.25.16-1jammy.20250728.072413).
ros-humble-turtle-tf2-py is already the newest version (0.3.7-1jammy.20250728.065805).
0 upgraded, 0 newly installed, 0 to remove and 5 not upgraded.
Requirement already satisfied: transforms3d in /usr/lib/python3/dist-packages (0.3.1)
WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system package
manager. It is recommended to use a virtual environment instead: https://pip.pypa.io/warnings/venv
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
ros-humble-rqt-tf-tree is already the newest version (1.0.5-1jammy.20250728.065630).
0 upgraded, 0 newly installed, 0 to remove and 5 not upgraded.
yahboom@yahboom-virtual-machine:~/workspace$
```

### 3.2 Starting Up

Then you can start the system using a launch file. You can then control one of the turtles, and the other will automatically follow.

```
ros2 launch turtle_tf2_py turtle_tf2_demo.launch.py
ros2 run turtlesim turtle_teleop_key
```

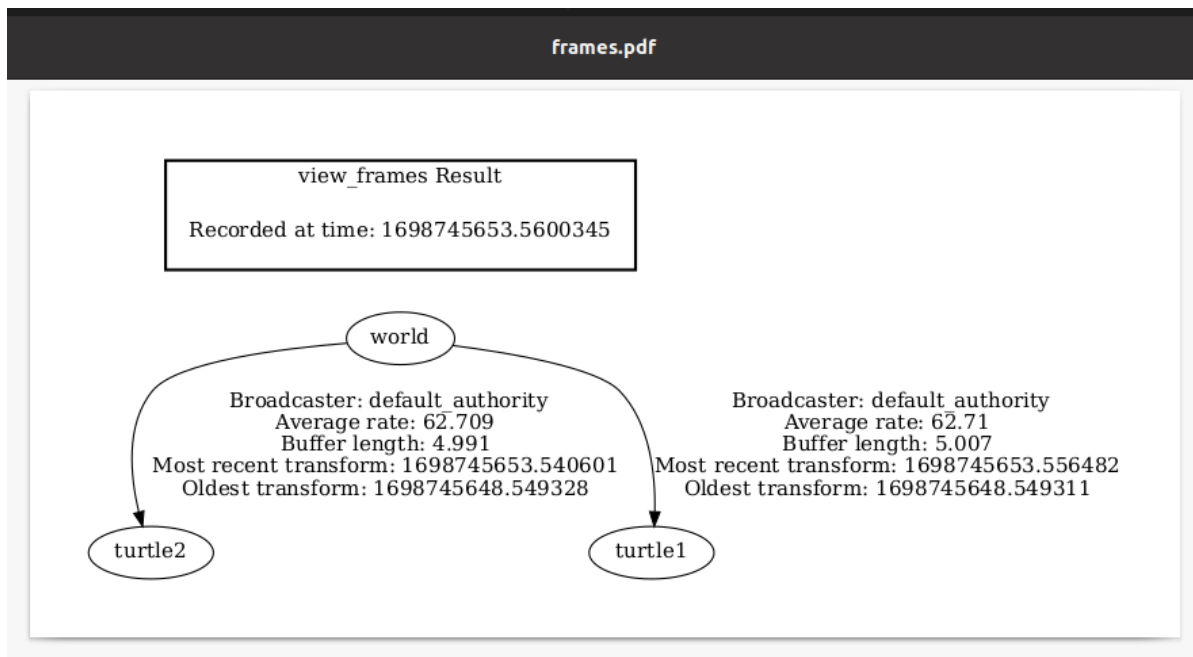
When we control the movement of one turtle, the other turtle will follow.



### 3.3 Viewing the TF Tree

```
ros2 run rqt_tf_tree rqt_tf_tree
```

You can view the TF transformation tree in the rqt window.



### 3.4 Querying Coordinate Transformation Information

Just viewing the coordinate system structure is not enough. If we want to know the specific relationship between two coordinate systems, we can use the tf2\_echo tool:

```
ros2 run tf2_ros tf2_echo turtle2 turtle1
```

After running successfully, the terminal will repeatedly print the coordinate system transformation values.

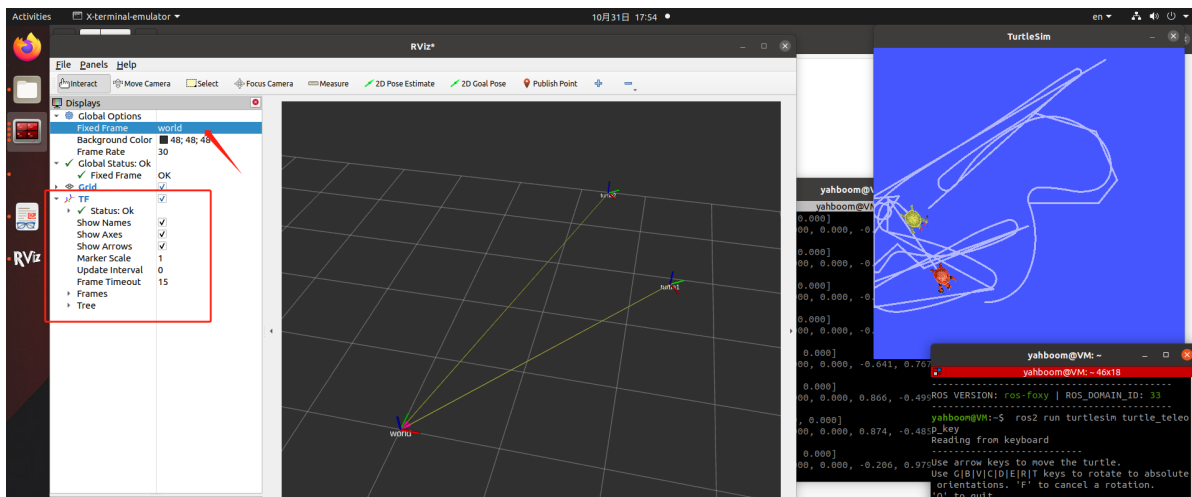
```
yahboom@VM: ~
yahboom@VM: ~ 80x24
yahboom@VM:~$ ros2 run tf2_ros tf2_echo turtle2 turtle1
[INFO] [1698745750.637285593] [tf2_echo]: Waiting for transform turtle2 -> turtle1: Invalid frame ID "turtle2" passed to canTransform argument target_frame - frame does not exist
At time 1698745751.620676614
- Translation: [0.000, 0.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, -0.187, 0.982]
At time 1698745752.628624962
- Translation: [0.000, 0.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, -0.187, 0.982]
At time 1698745753.621259108
- Translation: [0.000, 0.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, -0.187, 0.982]
At time 1698745754.629166540
- Translation: [0.000, 0.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, -0.187, 0.982]
At time 1698745755.620967109
- Translation: [0.000, 0.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, -0.187, 0.982]
At time 1698745756.613158127
- Translation: [0.000, 0.000, 0.000]
- Rotation: in Quaternion [0.000, 0.000, -0.187, 0.982]
```

### 3.5 Coordinate System Visualization

- Run rviz2 and add the TF display plugin.

```
rviz2
```

Set the reference coordinate system in rviz2 to "world," add the TF display plugin, and then animate the turtle. The coordinate axes in rviz will begin to move. Isn't this more intuitive?



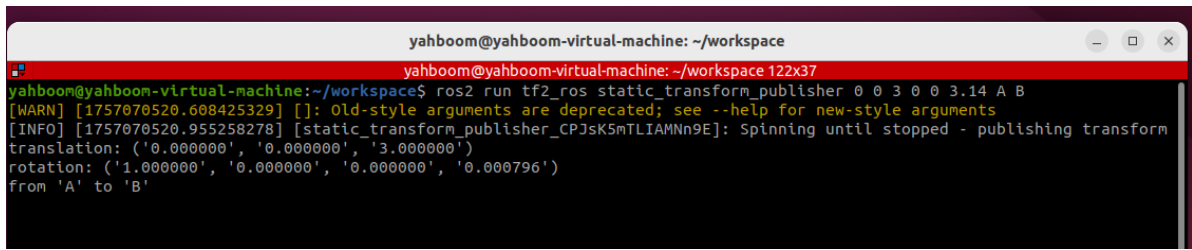
### 4. Static Coordinate Transformation

A static coordinate transformation refers to a fixed relative position between two coordinate systems. For example, the position between the radar and base\_link is fixed.

Example: **For ease of demonstration, this lesson is best performed in a virtual machine**

## 4.1. Publishing the pose from A to B

```
ros2 run tf2_ros static_transform_publisher 0 0 3 0 0 3.14 A B
```



```
yahboom@yahboom-virtual-machine: ~/workspace
yahboom@yahboom-virtual-machine: ~/workspace 122x37
yahboom@yahboom-virtual-machine:~/workspace$ ros2 run tf2_ros static_transform_publisher 0 0 3 0 0 3.14 A B
[WARN] [1757070520.608425329] []: Old-style arguments are deprecated; see --help for new-style arguments
[INFO] [1757070520.955258278] [static_transform_publisher_CPJsK5mTLIAMN9E]: Spinning until stopped - publishing transform
translation: ('0.000000', '0.000000', '3.000000')
rotation: ('1.000000', '0.000000', '0.000000', '0.000796')
from 'A' to 'B'
```

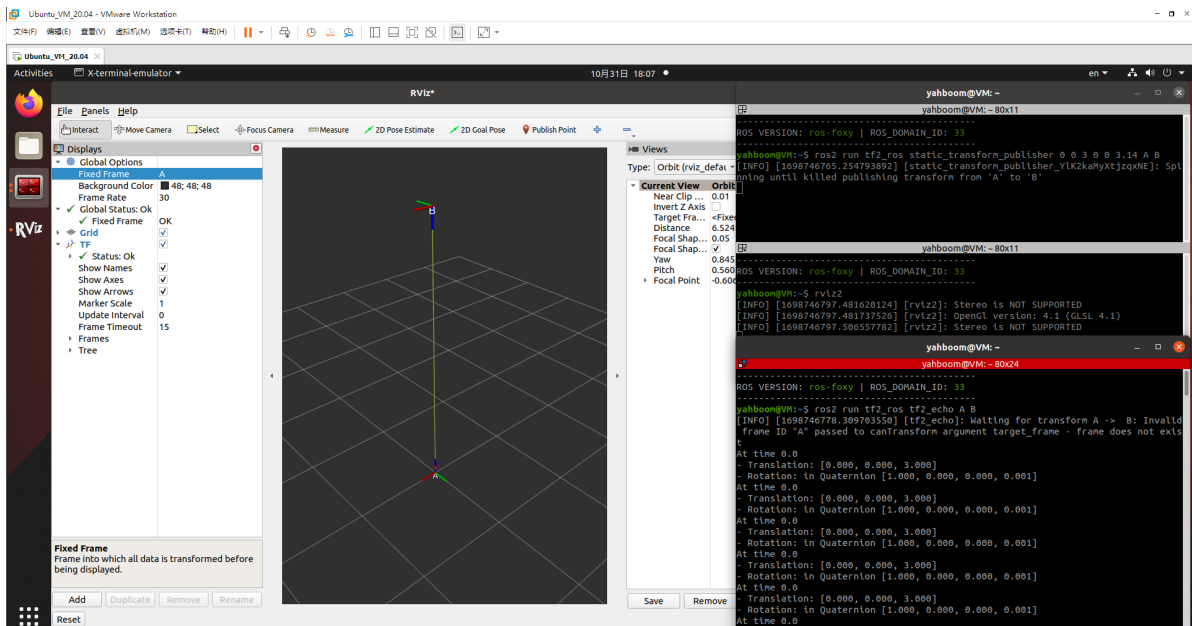
## 4.2. Monitoring/Getting TF Relationships

```
ros2 run tf2_ros tf2_echo A B
```

## 4.3. RIVZ Visualization

- Run rviz2 and add the TF display plugin.

```
rviz2
```



## 5. Case Introduction

In the previous lesson, we explained the TF relationship in the system-provided turtle following example. In this lesson, we will implement this functionality ourselves.

Course Content:

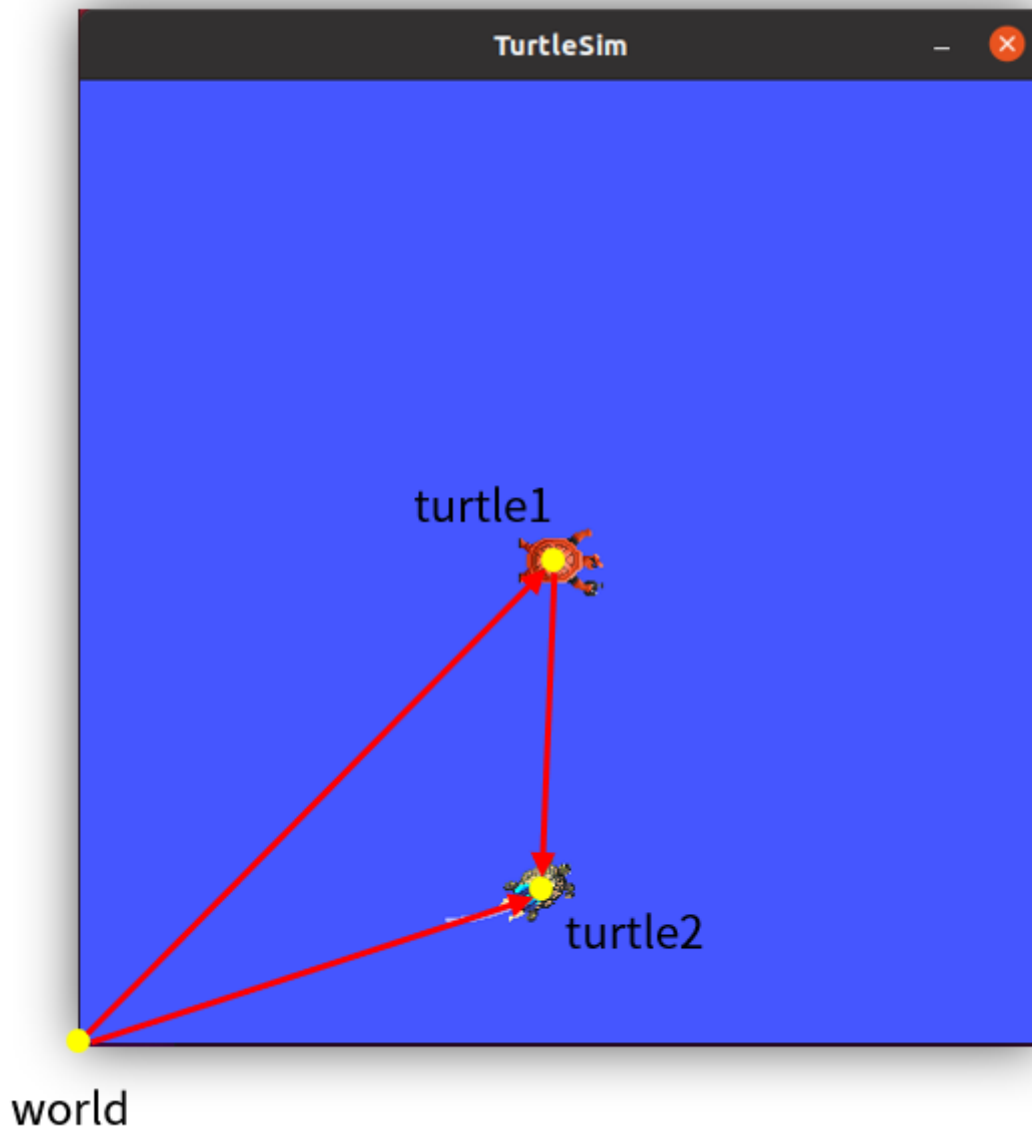
- Implementing a turtle-following example through programming
- Mastering the implementation of a dynamic broadcaster through programming
- Mastering the implementation of coordinate transformations between monitoring coordinate systems through programming
- Mastering the use of PID control to convert physical quantities (distance, angle) into speed control variables

Advanced Content:

- Understanding the TF system's ability to transform coordinate systems across time dimensions

## 6. Analysis of the Principles of the Turtle-Following Example

---



In a two-turtle simulator, we can define three coordinate systems. For example, the simulator's global reference system is called "world," and the turtle1 and turtle2 coordinate systems are at the center of the two turtles. Thus, the relative position of turtle1 and the world coordinate system represents the position of turtle1, and similarly for turtle2.

To make Turtle 2 move toward Turtle 1, we draw a line connecting the two and add an arrow. Does this remind you of vector calculations learned in high school? Vectors are used to describe coordinate transformations, so in this following example, TF can be used to solve this problem perfectly.

The length of a vector represents distance, and the direction represents angle. With distance and angle, we can calculate the speed by setting a random time. Then, we encapsulate and publish the speed topic, and Turtle 2 can start moving.

So the core of this example is to perform vector calculations using the coordinate system. Since the two turtles are constantly moving, the vectors must also be calculated according to a certain period. This requires the use of TF's dynamic broadcast and monitoring.

## 7. Create a new package

1. Create a new package in the src directory of the workspace to store our files.

```
ros2 pkg create pkg_tf --build-type ament_python --dependencies rclpy --node-name turtle_tf_broadcaster
```

Executing the above command will create the pkg\_tf package and a turtle\_tf\_broadcaster node. The relevant configuration files will be configured. Add the following code to the turtle\_tf\_broadcaster.py file:

```
import rclpy                                # ROS2 Python interface
library
from rclpy.node import Node                 # ROS2 node class
from geometry_msgs.msg import TransformStamped # Coordinate transformation
message
import tf_transformations                   # TF coordinate
transformation library
from tf2_ros import TransformBroadcaster    # TF coordinate
transformation broadcaster
from turtlesim.msg import Pose              # Turtlesim turtle position
message

class TurtleTFBroadcaster(Node):

    def __init__(self, name):
        super().__init__(name)              # Initialize the
ROS2 node parent class

        self.declare_parameter('turtle_name', 'turtle') # Create a
parameter for the turtle name
        self.turtle_name = self.get_parameter(          # Externally set
parameter value takes precedence; otherwise, default value is used.
'turtle_name').get_parameter_value().string_value

        self.tf_broadcaster = TransformBroadcaster(self) # Create and
initialize a TF coordinate transformation broadcast object

        self.subscription = self.create_subscription(   # Create a
subscriber to subscribe to turtle position messages
Pose,
f'/{self.turtle_name}/pose',                        # Use the turtle
name obtained from the parameters
self.turtle_pose_callback, 1)

    def turtle_pose_callback(self, msg):              # Create a
callback function to handle turtle position messages and convert them into
coordinate transformations
        transform = TransformStamped()                # Create a
coordinate transformation message object
```

```

        transform.header.stamp = self.get_clock().now().to_msg()      # Set the
timestamp of the coordinate transformation message
        transform.header.frame_id = 'world'                          # Set the
source coordinate system for the coordinate transformation
        transform.child_frame_id = self.turtlename                  # Set the
target coordinate system for the coordinate transformation
        transform.transform.translation.x = msg.x                    # Set the
X, Y, and Z translations in the coordinate transformation
        transform.transform.translation.y = msg.y
        transform.transform.translation.z = 0.0
        q = tf_transformations.quaternion_from_euler(0, 0, msg.theta) # Convert
Euler angles to quaternions (roll, pitch, yaw)
        transform.transform.rotation.x = q[0]                        # Set the
X, Y, and Z rotations (quaternions) in the coordinate transformation
        transform.transform.rotation.y = q[1]
        transform.transform.rotation.z = q[2]
        transform.transform.rotation.w = q[3]

        # Send the transformation
        self.tf_broadcaster.sendTransform(transform)                # Broadcast the
coordinate transformation. When the turtle's position changes, the coordinate
transformation information will be updated in time.

def main(args=None):
    rclpy.init(args=args)                                           # Initialize the ROS2
Python interface
    node = TurtleTFBroadcaster("turtle_tf_broadcaster")             # Create and initialize
a ROS2 node object
    rclpy.spin(node)                                                # Loop and wait for
ROS2 to exit
    node.destroy_node()                                             # Destroy the node
object
    rclpy.shutdown()                                                # Shut down the ROS2
Python interface

```

2. Next, create a new file [turtle\_following.py] in the same directory as turtle\_tf\_broadcaster.py and add the following code:

```

import math
import rclpy                                                        # ROS2 Python
interface library
from rclpy.node import Node                                         # ROS2 node class
import tf_transformations                                           # TF coordinate
transformation library
from tf2_ros import TransformException                             # TF left-side
transformation exception class
from tf2_ros.buffer import Buffer                                    # Buffer class for
storing coordinate transformation information
from tf2_ros.transform_listener import TransformListener            # Listener class for
coordinate transformations
from geometry_msgs.msg import Twist                                # ROS2 velocity
control message
from turtlesim.srv import Spawn                                    # Turtle spawning
service interface
class TurtleFollowing(Node):

```



[illegible]

```

        scale_rotation_rate = 1.0                                # Calculate
angular velocity based on the turtle's angle.
        msg.angular.z = scale_rotation_rate * math.atan2(
            trans.transform.translation.y,
            trans.transform.translation.x)

        scale_forward_speed = 0.5                                # Calculate
linear velocity based on the turtle's distance.
        msg.linear.x = scale_forward_speed * math.sqrt(
            trans.transform.translation.x ** 2 +
            trans.transform.translation.y ** 2)

        self.publisher.publish(msg)                              # Publish
velocity command for the turtle to follow.
    else:                                                         # If the
following turtle is not spawned.
        if self.result.done():                                    # Check if
the turtle is spawned.
            self.get_logger().info(
                f'successfully spawned {self.result.result().name}')
            self.turtle_spawned = True
        else:                                                     # Still no
following turtle spawned.
            self.get_logger().info('spawn is not finished')
    else:                                                         # If no
turtle spawning service is requested
        if self.spawner.service_is_ready():                     # If the
turtle spawning server is ready
            request = Spawn.Request()                            # Create a
request data
            request.name = 'turtle2'                             # Set the
content of the request data, including turtle name, xy position, and posture
            request.x = float(4)
            request.y = float(2)
            request.theta = float(0)

            self.result = self.spawner.call_async(request)       # Send
service request
            self.turtle_spawning_service_ready = True          # Set the
flag to indicate that the request has been sent
        else:
            self.get_logger().info('Service is not ready')      # Turtle
spawning server is not ready

def main(args=None):
    rclpy.init(args=args)                                        # ROS2 Python interface
initialization
    node = TurtleFollowing("turtle_following")                  # Create and initialize a ROS2
node object.
    rclpy.spin(node)                                            # Loop and wait for ROS2 to
exit.
    node.destroy_node()                                         # Destroy the node object
    rclpy.shutdown()                                           # Shut down the ROS2 Python
interface.

```

3. Create a new launch folder under the pkg\_tf package, create a new file [turtle\_following.launch.py] in the launch folder, and add the following content:

```
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node

def generate_launch_description():

    return LaunchDescription([
        DeclareLaunchArgument('source_frame', default_value='turtle1',
description='Target frame name.'),
        Node(
            package='turtlesim',
            executable='turtlesim_node',
        ),
        Node(
            package='pkg_tf',
            executable='turtle_tf_broadcaster',
            name='broadcaster1',
            parameters=[
                {'turtle_name': 'turtle1'}
            ]
        ),
        Node(
            package='pkg_tf',
            executable='turtle_tf_broadcaster',
            name='broadcaster2',
            parameters=[
                {'turtle_name': 'turtle2'}
            ]
        ),
        Node(
            package='pkg_tf',
            executable='turtle_following',
            name='listener',
            parameters=[
                {'source_frame': LaunchConfiguration('source_frame')}
            ]
        ),
    ])
])
```

## 8. Edit the configuration file

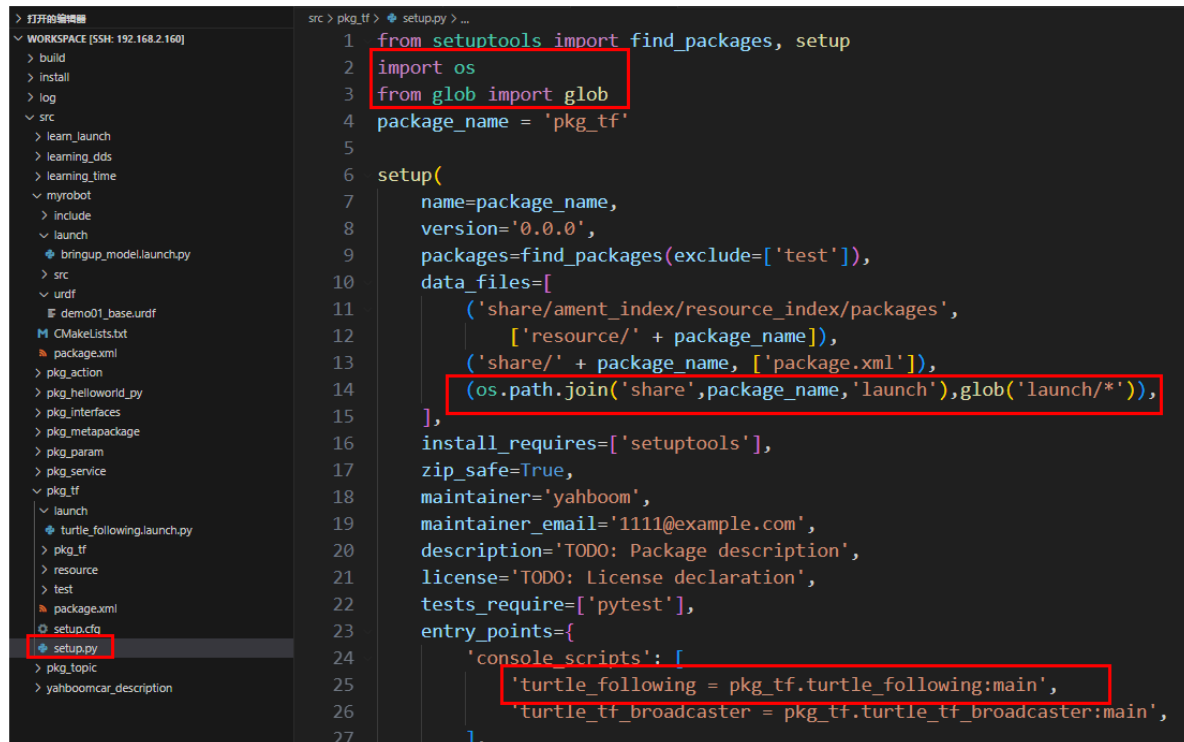
### 8.1. Configuration in setup.py

- Import relevant libraries

```
import os
from glob import glob
```

- Add the turtle\_following node information and add the following command to copy the launch file to the shared directory in the install.

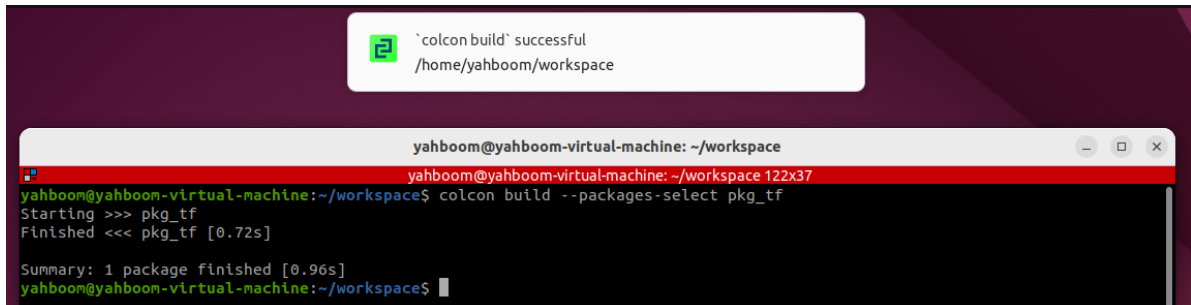
```
(os.path.join('share', package_name, 'launch'), glob('launch/*'))),
```



```
src > pkg_tf > setup.py > ...
1 from setuptools import find_packages, setup
2 import os
3 from glob import glob
4 package_name = 'pkg_tf'
5
6 setup(
7     name=package_name,
8     version='0.0.0',
9     packages=find_packages(exclude=['test']),
10    data_files=[
11        ('share/ament_index/resource_index/packages',
12         ['resource/' + package_name]),
13        ('share/' + package_name, ['package.xml']),
14        (os.path.join('share', package_name, 'launch'), glob('launch/*')),
15    ],
16    install_requires=['setuptools'],
17    zip_safe=True,
18    maintainer='yahboom',
19    maintainer_email='1111@example.com',
20    description='TODO: Package description',
21    license='TODO: License declaration',
22    tests_require=['pytest'],
23    entry_points={
24        'console_scripts': [
25            'turtle_following = pkg_tf.turtle_following:main',
26            'turtle_tf_broadcaster = pkg_tf.turtle_tf_broadcaster:main',
27        ],
28    },
29 )
```

## 9. Compile the package

```
colcon build --packages-select pkg_tf
```

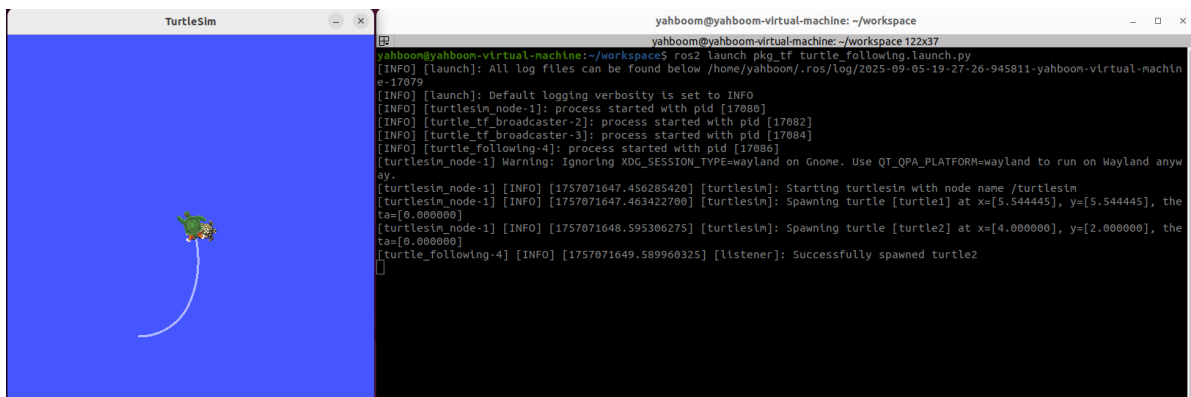


```
yahboom@yahboom-virtual-machine: ~/workspace
colcon build --packages-select pkg_tf
Summary: 1 package finished [0.96s]
```

## 10. Run the Program

- Refresh the terminal environment variables, then run

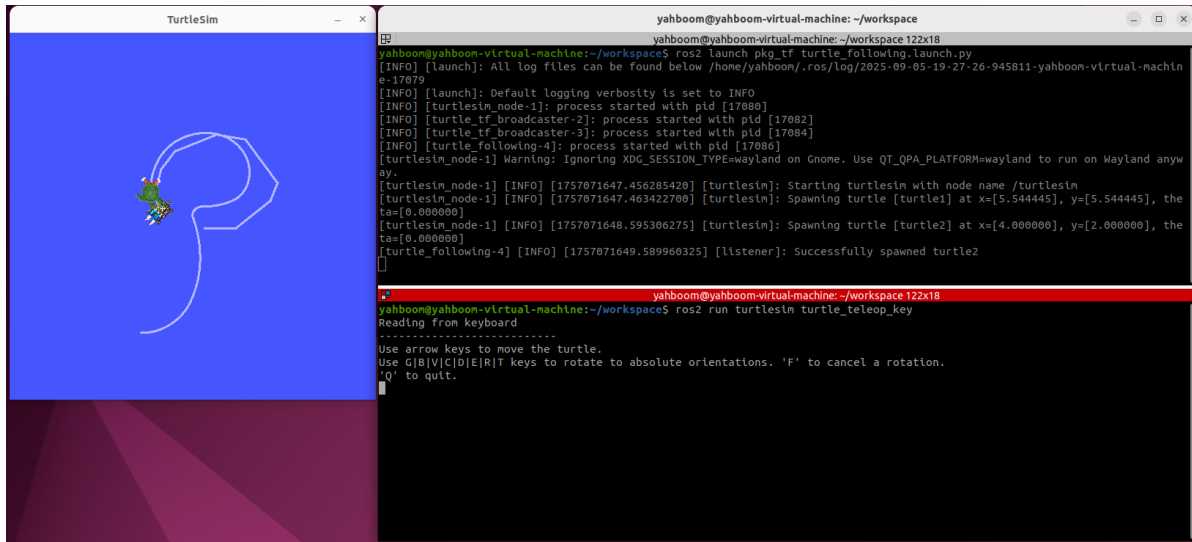
```
ros2 launch pkg_tf turtle_following.launch.py
```



```
yahboom@yahboom-virtual-machine: ~/workspace
ros2 launch pkg_tf turtle_following.launch.py
[INFO] [launch]: All log files can be found below /home/yahboom/.ros/log/2025-09-05-19-27-26-945811-yahboom-virtual-machine-17879
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [turtlesim_node-1]: process started with pid [17880]
[INFO] [turtle_tf_broadcaster-2]: process started with pid [17882]
[INFO] [turtle_tf_broadcaster-3]: process started with pid [17884]
[INFO] [turtle_following-4]: process started with pid [17886]
[turtlesim_node-1] Warning: Ignoring XDG_SESSION_TYPE=wayland on Gnome. Use QT_QPA_PLATFORM=wayland to run on Wayland anyway.
[turtlesim_node-1] [INFO] [1757071647.456285420] [turtlesim]: Starting turtlesim with node name /turtlesim
[turtlesim_node-1] [INFO] [1757071647.463422700] [turtlesim]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], the
ta=[0.000000]
[turtlesim_node-1] [INFO] [1757071648.595306275] [turtlesim]: Spawning turtle [turtle2] at x=[4.000000], y=[2.000000], the
ta=[0.000000]
[turtle_following-4] [INFO] [1757071649.589960325] [listener]: Successfully spawned turtle2
```

- Start the turtle keyboard control node, controlling the movement of the first turtle; the second turtle will automatically follow.

```
ros2 run turtlesim turtle_teleop_key
```



In this terminal, press the up, down, left, and right keys on your keyboard to control the movement of one turtle. The other turtle will follow until they overlap.

## 11. Advanced Content

- Understanding TF's Cross-Time Transformation Capabilities

Buffer automatically caches all TF transformation relationships within the TF system over the past 10 seconds (you can set any desired cache duration using the Buffer constructor). All transformations in the buffer are timestamp-based and traceable. Even if two coordinate systems are at different points in time, the transformation relationship between them can be found. For a detailed explanation of this principle, please refer to the references for designing TF systems (the references are in this section's folder, or you can click the link at the beginning of this section).

```
self.tf_buffer = Buffer() # 创建保存坐标变换信息的缓冲区
self.tf_listener = TransformListener(self.tf_buffer, self) # 创建坐标变换的监听器
```

- The following is a supplement to the turtle following example above. The red arrow indicates that the transformation between two coordinate systems at different points in time can be found across time.

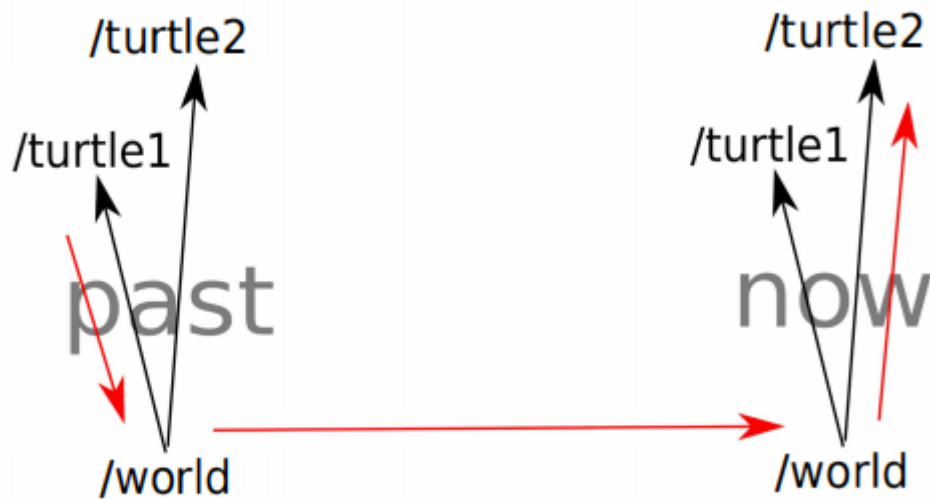


Fig. 4: A simple *tf* tree from a core ROS tutorial, with debugging information.

*Long Term Data Storage:* The above example assumes that you have the ability to transform between frames a and b at time 0 and b and c at time 1. This will only work if the length of the buffer in your *Listener* module is long enough to encompass both time 0 and time 1. To be able to store data for a long time, it should be transformed into the fixed frame, b, when saved. Then the source frame is the same as frame b and consequently the transform between the two is the identity, leaving only  $T_{b@t_1}^{c@t_1}$  needing to be computed to find the current location of any previously observed object. This simple case is common in many robotic systems because they do not have the tools to compute the more complicated cases. This will only work when the assumptions about where the identity transform can be used is maintained.