Report on Implementation of a PD Controller for Actuator 4

Group 18 : Manav Mepani, Prasham Soni, Siddharth Shrotri

1. Introduction

This report describes the development and testing of a Proportional-Derivative (PD) controller for Actuator 4 of the Open Manipulator-X robot using ROS 2. The actuator operates in the current (effort) control mode, while the remaining actuators remain in position control mode. The goal of this project is to implement a PD controller from scratch to regulate Actuator 4's position to user-defined references.

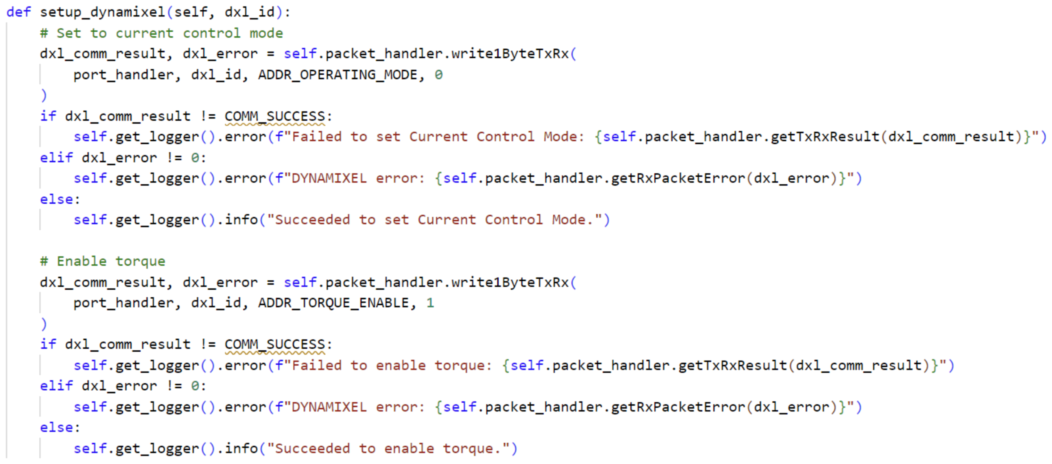
2. Objectives

- Develop a ROS 2 service to accept position reference values for Actuator 4.
- Continuously read and publish joint effort values at high sampling rates using ROS 2 publishers and subscribers.
- Implement a PD controller for Actuator 4 in current control mode using ROS 2.
- Tune the PD controller to achieve fast convergence with minimal overshoot.
- Record and visualize performance data.

3. Implementation Steps

3.1 Switching Actuator Modes

- Configured Actuator 4 to operate in current (effort) control mode using ROS 2 interface code by implementing the following sections:



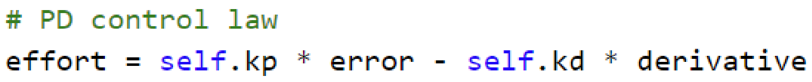
```
def setup_dynamixel(self, dxl_id):
    # Set to current control mode
    dxl_comm_result, dxl_error = self.packet_handler.write1ByteTxRx(
        port_handler, dxl_id, ADDR_OPERATING_MODE, 0
    )
    if dxl_comm_result != COMM_SUCCESS:
        self.get_logger().error(f"Failed to set Current Control Mode: {self.packet_handler.getTxRxResult(dxl_comm_result)}")
    elif dxl_error != 0:
        self.get_logger().error(f"DYNAMIXEL error: {self.packet_handler.getRxPacketError(dxl_error)}")
    else:
        self.get_logger().info("Succeeded to set Current Control Mode.")

    # Enable torque
    dxl_comm_result, dxl_error = self.packet_handler.write1ByteTxRx(
        port_handler, dxl_id, ADDR_TORQUE_ENABLE, 1
    )
    if dxl_comm_result != COMM_SUCCESS:
        self.get_logger().error(f"Failed to enable torque: {self.packet_handler.getTxRxResult(dxl_comm_result)}")
    elif dxl_error != 0:
        self.get_logger().error(f"DYNAMIXEL error: {self.packet_handler.getRxPacketError(dxl_error)}")
    else:
        self.get_logger().info("Succeeded to enable torque.")
```

- Left the other actuators-maintained position control mode.

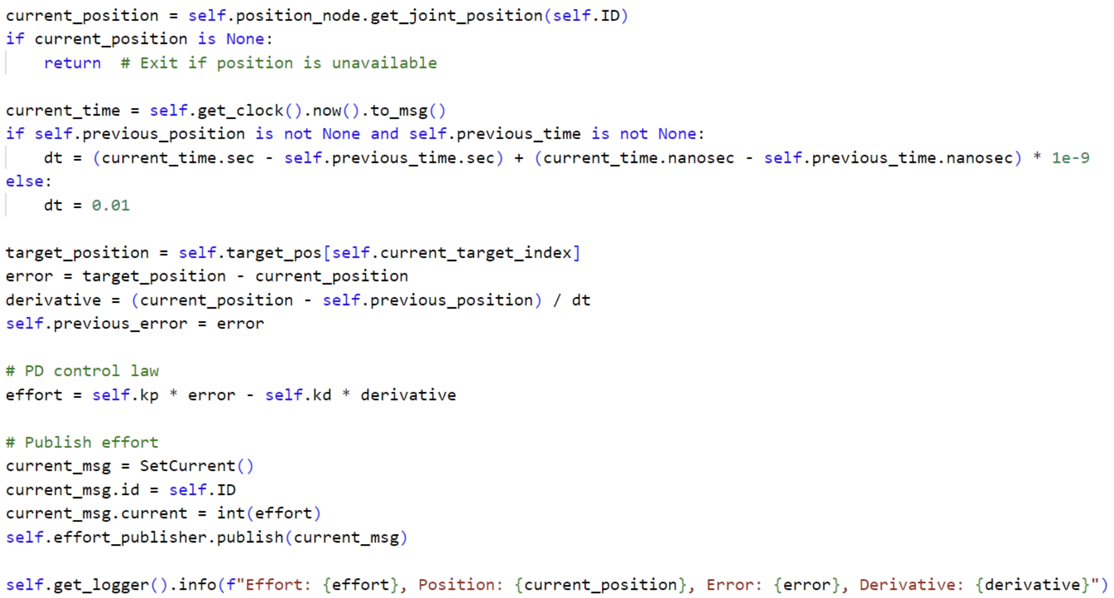
3.2 Controller Design

- Designed the PD controller:

Control law:



```
# PD control law
effort = self.kp * error - self.kd * derivative
```

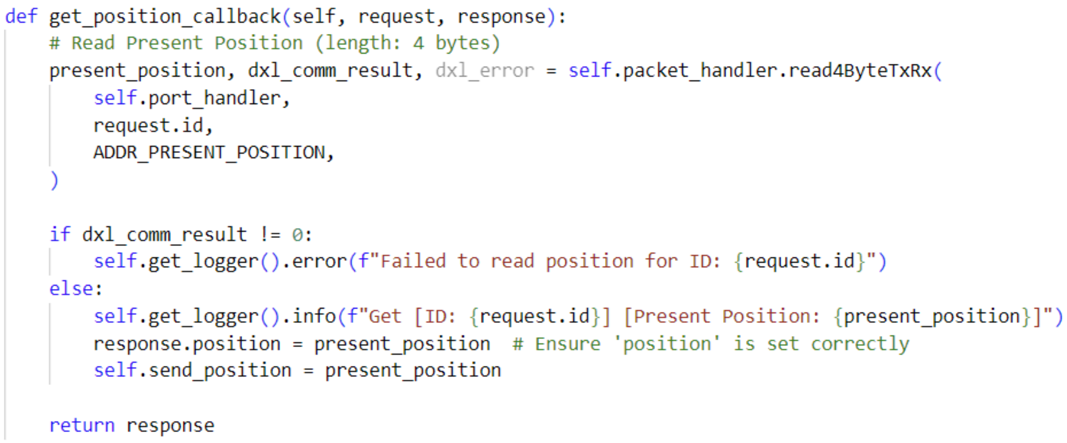


```
current_position = self.position_node.get_joint_position(self.ID)
if current_position is None:
    return  # Exit if position is unavailable

current_time = self.get_clock().now().to_msg()
if self.previous_position is not None and self.previous_time is not None:
    dt = (current_time.sec - self.previous_time.sec) + (current_time.nanosec - self.previous_time.nanosec) * 1e-9
else:
    dt = 0.01

target_position = self.target_pos[self.current_target_index]
error = target_position - current_position
derivative = (current_position - self.previous_position) / dt
self.previous_error = error

# PD control law
effort = self.kp * error - self.kd * derivative

# Publish effort
current_msg = SetCurrent()
current_msg.id = self.ID
current_msg.current = int(effort)
self.effort_publisher.publish(current_msg)

self.get_logger().info(f"Effort: {effort}, Position: {current_position}, Error: {error}, Derivative: {derivative}")
```

3.3 Node Implementation

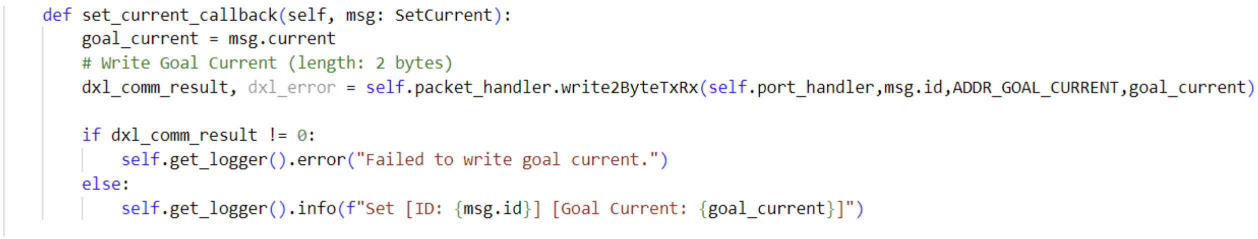
Developed a ROS 2 node with the following functionalities:

- **Service**: A ROS 2 service to accept position reference values for Actuator 4.
- **Subscriber**: A ROS 2 subscriber to read the current joint position values from Actuator 4.



```
def get_position_callback(self, request, response):
    # Read Present Position (length: 4 bytes)
    present_position, dxl_comm_result, dxl_error = self.packet_handler.read4ByteTxRx(
        self.port_handler,
        request.id,
        ADDR_PRESENT_POSITION,
    )

    if dxl_comm_result != 0:
        self.get_logger().error(f"Failed to read position for ID: {request.id}")
    else:
        self.get_logger().info(f"Get [ID: {request.id}] [Present Position: {present_position}]")
        response.position = present_position  # Ensure 'position' is set correctly
        self.send_position = present_position

    return response
```

- **Publisher**: A ROS 2 publisher to send calculated effort values to Actuator 4 at a high sampling rate.



```
def set_current_callback(self, msg: SetCurrent):
    goal_current = msg.current
    # Write Goal Current (length: 2 bytes)
    dxl_comm_result, dxl_error = self.packet_handler.write2ByteTxRx(self.port_handler,msg.id,ADDR_GOAL_CURRENT,goal_current)

    if dxl_comm_result != 0:
        self.get_logger().error("Failed to write goal current.")
    else:
        self.get_logger().info(f"Set [ID: {msg.id}] [Goal Current: {goal_current}]")
```

3.4 PD Gain Tuning

- Started with small K_p.
- Incrementally increased until the overshoot was observed.
- Added K_d and adjusted both gains to minimize overshoot and improve convergence.

3.5 Testing

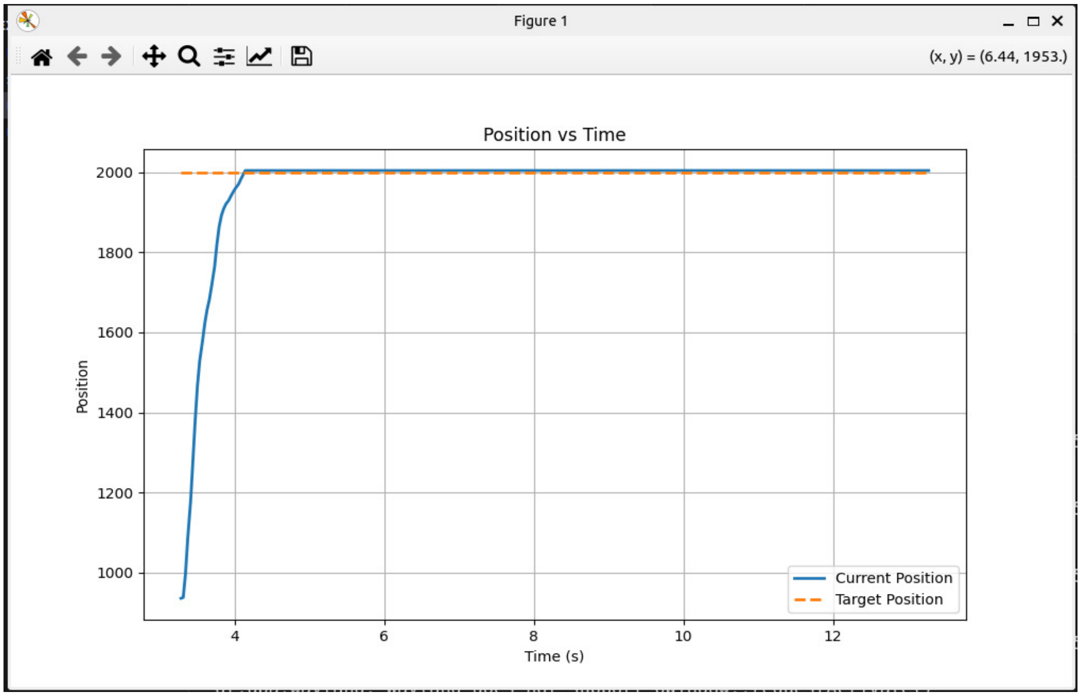
- Tested the controller with three different reference positions:
    1. Reference 1: From position 930 to 2000 units
    2. Reference 2: From position 2180 to 1500 units
    3. Reference 3: From position 930 to 1800 units
- Collected data for 10 seconds with a sampling time of 0.01 seconds.

3.6 Data Recording and Visualization

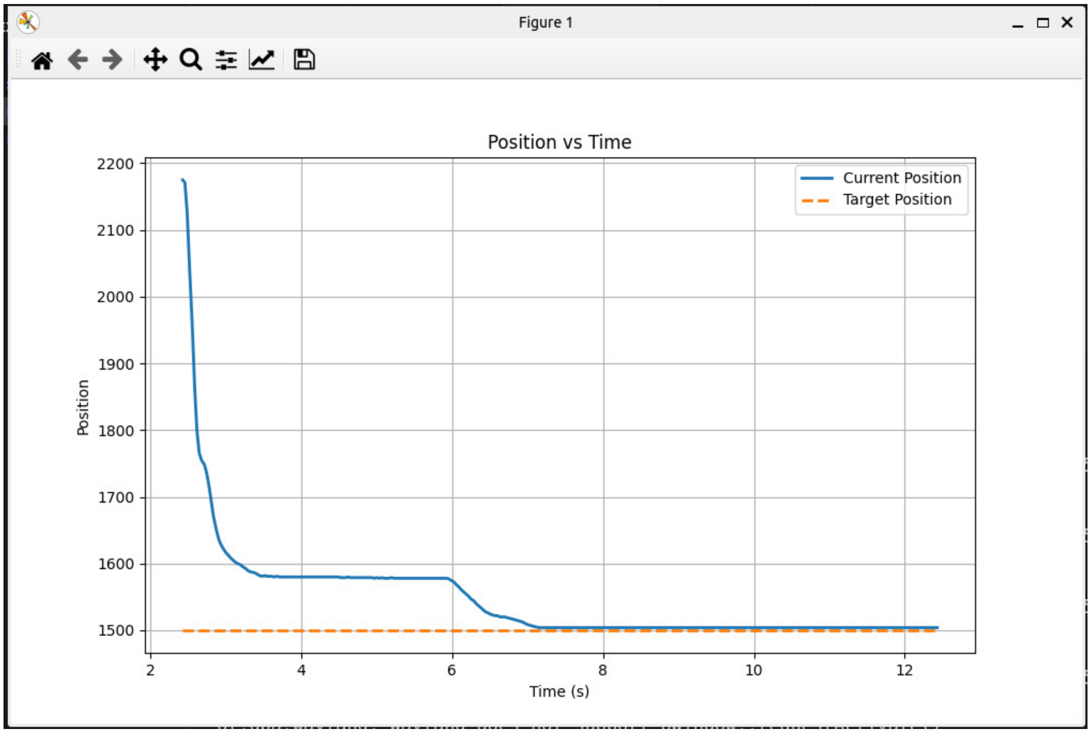
- Recorded the reference and actual joint positions in a text file using ROS 2 logging utilities.
- Plotted the results using Python.

4. Results

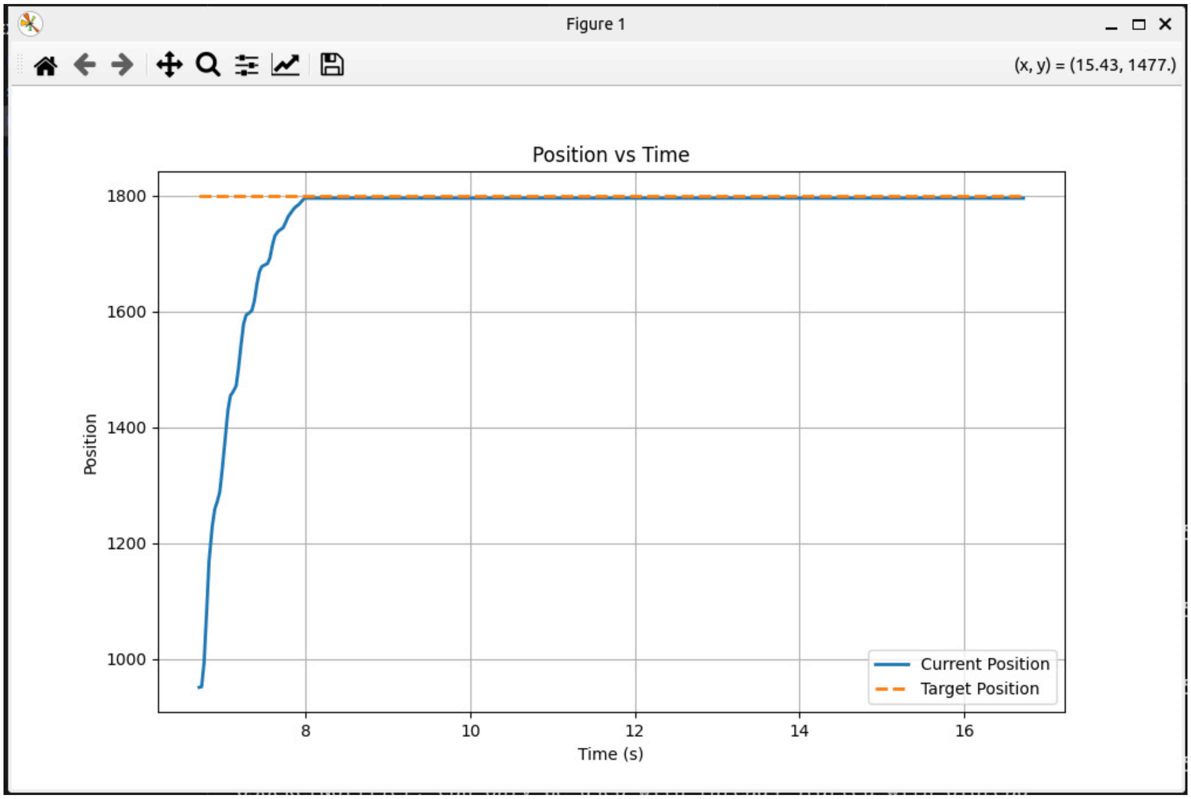
4.1 Plots : Reference 1:



Reference 2:



Reference 3:



4.2 Observations

- Observed K_p, K_d values

K_p	0.19
K_d	0.03

- Describe how the controller performed in terms of:
  - Convergence speed
    The convergence speed of the actuator to a steady position was good with these values.
  - Overshoot
    With these values hardly any overshoot was seen.
  - Steady-state accuracy
    There was a steady-state error in one direction (i.e. going up) while it went to the specified position smoothly and with great accuracy when going downwards.

---

5. Challenges and Improvements

- It was struggling a bit to go from a downwards facing position to an upwards position as it had a steady state error at the end. This suggests that for a better result we will have to integrate the Integral part too.

Appendices

A. Code

PD control code of robot

```
class GetCurrentPosition(Node):
    def __init__(self):
        super().__init__('get_current_position')
        self.pos_client = self.create_client(GetPosition, 'get_position')
        self.request = GetPosition.Request()

        while not self.pos_client.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('GetPosition service not available, waiting again...')

    def get_joint_position(self, motor_id):
        """Send a request to get the joint position and return the position."""
        self.request.id = motor_id
        future = self.pos_client.call_async(self.request)
        rclpy.spin_until_future_complete(self, future)
        try:
            response = future.result()
            if response is not None:
                return response.position
        except Exception as e:
            self.get_logger().error(f"Failed to get position: {e}")
        return None

class PDControl(Node):
    def __init__(self, position_node):
        super().__init__('pd_control')
        self.position_node = position_node
        self.previous_time = None
        self.previous_position = None
        self.flag = True

        # PD controller constants
        self.kp = 0.19
        self.kd = 0.03
        self.ID = 14

        # Effort publisher and service for setting targets
        self.effort_publisher = self.create_publisher(SetCurrent, 'set_current', 10)
        self.create_service(FloatPos, 'currentcalc', self.start_calculation)

        # Timer for control loop
        self.timer = self.create_timer(0.01, self.process_loop)  # 100 Hz control loop
        self.target_pos = None
        self.current_target_index = 0
        self.previous_error = 0.0

        # Logging setup
        self.log_file = open("position_log.txt", "w")
        self.start_time = time.time()
        self.logging_duration = 10  # seconds

    def start_calculation(self, request, response):
        """Service callback to start the current calculation."""
        self.target_pos = np.array(request.pos4)
        self.current_target_index = 0
        self.get_logger().info(f"Starting calculation with target positions: {self.target_pos}")
        response.success = True
        return response

    def process_loop(self):
        """Process the control loop for current calculation."""
        if self.target_pos is None or self.current_target_index >= len(self.target_pos):
            return
        if self.flag:
            self.previous_position = self.position_node.get_joint_position(self.ID)
            self.flag = False

        current_position = self.position_node.get_joint_position(self.ID)
        if current_position is None:
            return  # Exit if position is unavailable

        current_time = self.get_clock().now().to_msg()
        if self.previous_position is not None and self.previous_time is not None:
            dt = (current_time.sec - self.previous_time.sec) + (current_time.nanosec -
self.previous_time.nanosec) * 1e-9
        else:
            dt = 0.01

        target_position = self.target_pos[self.current_target_index]
        error = target_position - current_position
        derivative = (current_position - self.previous_position) / dt
        self.previous_error = error

        # PD control law
        effort = self.kp * error - self.kd * derivative

        # Publish effort
        current_msg = SetCurrent()
        current_msg.id = self.ID
        current_msg.current = int(effort)
        self.effort_publisher.publish(current_msg)

        self.get_logger().info(f"Effort: {effort}, Position: {current_position}, Error: {error},
Derivative: {derivative}")

        # Log current and target positions
        elapsed_time = time.time() - self.start_time
        if elapsed_time <= self.logging_duration:
            self.log_file.write(f"Time: {elapsed_time:.2f}s, Current Position: {current_position}, Target
Position: {target_position}\n")
            self.log_file.flush()  # Ensure data is written to the file immediately
        elif not self.log_file.closed:
            self.log_file.close()
            self.get_logger().info("Stopped logging positions after 10 seconds.")

        self.previous_position = current_position
        self.previous_time = current_time

        # Check if target position is reached
        if abs(error) < 5:  # Tolerance for reaching the target
            self.current_target_index += 1

        if self.current_target_index >= len(self.target_pos):
            self.get_logger().info("Calculation complete.")
            self.target_pos = None
```

Motor control code

```
# Constants for DYNAMIXEL motors
ADDR_OPERATING_MODE = 11
ADDR_TORQUE_ENABLE = 64
ADDR_GOAL_CURRENT = 102
ADDR_PRESENT_CURRENT = 126
PROTOCOL_VERSION = 2.0
BAUDRATE = 1000000
DEVICE_NAME = "/dev/ttyUSB0"
MAX_RETRIES = 5  # Maximum retry count for communication failures
RETRY_DELAY = 2  # Delay in seconds between retries
# Initialize the port handler
port_handler = PortHandler(DEVICE_NAME)


def reset_port():
    """ Function to reset the serial port and reinitialize the communication """
    port_handler.closePort()
    time.sleep(1)  # Short delay to ensure the port is released
    if not port_handler.openPort():
        print("Failed to open the port!")
        return False
    if not port_handler.setBaudRate(BAUDRATE):
        print("Failed to set the baudrate!")
        return False
    return True

class CurrentReadWriteNode(Node):
    def __init__(self):
        super().__init__('current_read_write_node')
        self.get_logger().info("Run read_write_node")

        # Initialize packet_handler as an instance variable
        self.packet_handler = PacketHandler(PROTOCOL_VERSION)

        # Create a subscriber for setting motor current
        self.create_subscription(SetCurrent,'set_current',self.set_current_callback,10)

        # Create a service for getting motor current
        self.create_service(GetCurrent,'get_current',self.get_current_callback)
        self.setup_dynamixel(14)

    def set_current_callback(self, msg):
        dxl_error = 0
        goal_current = msg.current
        ID = msg.id

        # Retry logic for setting current
        retries = 0

        while retries < MAX_RETRIES:
            dxl_comm_result, dxl_error = self.packet_handler.write2ByteTxRx( port_handler,ID,
                ADDR_GOAL_CURRENT,goal_current)

            if dxl_comm_result == COMM_SUCCESS and dxl_error == 0:
                self.get_logger().info(f"Set [ID: {msg.id}] [Goal Current: {msg.current}]")

            else:
                self.get_logger().error(f"Attempt {retries + 1}/{MAX_RETRIES} failed:
{self.packet_handler.getTxRxResult(dxl_comm_result)} - {self.packet_handler.getRxPacketError(dxl_error)}")
                retries += 1
                time.sleep(RETRY_DELAY)

                # If the communication fails due to a disconnection, reset the port
                if "no status packet" in str(self.packet_handler.getTxRxResult(dxl_comm_result)):
                    if reset_port():
                        self.get_logger().info("Port reset successful. Retrying...")
                    else:
                        self.get_logger().error("Failed to reset the port.")
                        break

        self.get_logger().error("Failed to set current after multiple attempts.")

    def get_current_callback(self, request, response):
        dxl_error = 0
        present_current = 0

        # Retry logic for getting current
        retries = 0
        while retries < MAX_RETRIES:
            dxl_comm_result, present_current, dxl_error = self.packet_handler.read2ByteTxRx(
                port_handler,
                request.id,
                ADDR_PRESENT_CURRENT
            )

            if dxl_comm_result == COMM_SUCCESS and dxl_error == 0:
                self.get_logger().info(f"Get [ID: {request.id}] [Present Current: {present_current}]")
                response.current = present_current
                return response
            else:
                self.get_logger().error(f"Attempt {retries + 1}/{MAX_RETRIES} failed: 
{self.packet_handler.getTxRxResult(dxl_comm_result)} - {self.packet_handler.getRxPacketError(dxl_error)}")
                retries += 1
                time.sleep(RETRY_DELAY)

                # If the communication fails due to a disconnection, reset the port
                if "no status packet" in str(self.packet_handler.getTxRxResult(dxl_comm_result)):
                    if reset_port():
                        self.get_logger().info("Port reset successful. Retrying...")
                    else:
                        self.get_logger().error("Failed to reset the port.")
                        break

        self.get_logger().error("Failed to get current after multiple attempts.")
        response.current = -1  # Return -1 if unable to get current
        return response

    def setup_dynamixel(self, dxl_id):
        # Set to current control mode
        dxl_comm_result, dxl_error = self.packet_handler.write1ByteTxRx(
            port_handler, dxl_id, ADDR_OPERATING_MODE, 0
        )
        if dxl_comm_result != COMM_SUCCESS:
            self.get_logger().error(f"Failed to set Current Control Mode: 
{self.packet_handler.getTxRxResult(dxl_comm_result)}")
        elif dxl_error != 0:
            self.get_logger().error(f"DYNAMIXEL error: {self.packet_handler.getRxPacketError(dxl_error)}")
        else:
            self.get_logger().info("Succeeded to set Current Control Mode.")

        # Enable torque
        dxl_comm_result, dxl_error = self.packet_handler.write1ByteTxRx(
            port_handler, dxl_id, ADDR_TORQUE_ENABLE, 1
        )
        if dxl_comm_result != COMM_SUCCESS:
            self.get_logger().error(f"Failed to enable torque: 
{self.packet_handler.getTxRxResult(dxl_comm_result)}")
        elif dxl_error != 0:
            self.get_logger().error(f"DYNAMIXEL error: {self.packet_handler.getRxPacketError(dxl_error)}")
        else:
            self.get_logger().info("Succeeded to enable torque.")


def main(args=None):
    rclpy.init(args=args)
```

```
if not port_handler.openPort():
    print("Failed to open the port!")
    return
if not port_handler.setBaudRate(BAUDRATE):
    print("Failed to set the baudrate!")
    return

node = CurrentReadWriteNode()
rclpy.spin(node)
node.destroy_node()
rclpy.shutdown()

# Disable torque before exiting
node.packet_handler.write1ByteTxRx(port_handler, 254, ADDR_TORQUE_ENABLE, 0)
port_handler.closePort()
```

Publisher / Subscriber service

```
# Control table address for X series (except XL-320)
ADDR_OPERATING_MODE = 11
ADDR_TORQUE_ENABLE = 64
ADDR_GOAL_POSITION = 116
ADDR_PRESENT_POSITION = 132
ADDR_GOAL_CURRENT = 102
ADDR_PRESENT_CURRENT = 126

# Protocol version
PROTOCOL_VERSION = 2.0  # Default Protocol version of DYNAMIXEL X series.
BAUDRATE = 1000000  # Default Baudrate of DYNAMIXEL X series
DEVICE_NAME = "/dev/ttyUSB0"  # [Linux]: "/dev/ttyUSB*", [Windows]: "COM*"

class ReadWriteNode(Node):
    def __init__(self):
        super().__init__('read_write_node')
        self.get_logger().info("Run read write node")
        self.ID = 14

        # Subscriber for the 'set_position' topic
        self.set_position_subscriber_ =
self.create_subscription(SetPosition,'set_position',self.set_position_callback,10)

        # Subscriber for the 'set_current' topic
        self.set_current_subscriber_ =
self.create_subscription(SetCurrent,'set_current',self.set_current_callback,10)

        # Service for getting the 'get_position' service
        self.get_position_server_ =
self.create_service(GetPosition,'get_position',self.get_position_callback)

        # Service for getting the 'get_current' service
        self.get_current_server_ =
self.create_service(GetCurrent,'get_current',self.get_current_callback)

        self.port_handler = PortHandler(DEVICE_NAME)
        self.packet_handler = PacketHandler(PROTOCOL_VERSION)

        self.position_publish = self.create_publisher(Int32, 'set_newpos', 10)
        timer_period = 0.001  # seconds
        self.timer = self.create_timer(timer_period, self.send_currentpos)
        self.send_position = None

        # Open Serial Port
        if not self.port_handler.openPort():
            self.get_logger().error("Failed to open the port!")
            return
        else:
            self.get_logger().info("Succeeded to open the port.")

        # Set the baudrate of the serial port
        if not self.port_handler.setBaudRate(BAUDRATE):
            self.get_logger().error("Failed to set the baudrate!")
            return
        else:
            self.get_logger().info("Succeeded to set the baudrate.")

        self.setup_dynamixel()
    def setup_dynamixel(self):
        # Set the operating mode for Current Control Mode
        dxl_comm_result, dxl_error =
self.packet_handler.write1ByteTxRx(self.port_handler,self.ID, ADDR_OPERATING_MODE,0)

        if dxl_comm_result != 0:
            self.get_logger().error("Failed to set Current Control Mode.")
        else:
            self.get_logger().info("Succeeded to set Current Control Mode.")

        # Enable Torque of DYNAMIXEL
        dxl_comm_result, dxl_error =
self.packet_handler.write1ByteTxRx(self.port_handler,self.ID, ADDR_TORQUE_ENABLE,1)

        if dxl_comm_result != 0:
            self.get_logger().error("Failed to enable torque.")
        else:
            self.get_logger().info("Succeeded to enable torque.")

    def set_position_callback(self, msg: SetPosition):
        goal_position = msg.position
        # Write Goal Position (length: 4 bytes)
        dxl_comm_result, dxl_error =
self.packet_handler.write4ByteTxRx(self.port_handler,msg.id,ADDR_GOAL_POSITION,goal_position)

        if dxl_comm_result != 0:
            self.get_logger().error("Failed to write goal position.")
        else:
            self.get_logger().info(f"Set [ID: {msg.id}] [Goal Position: {goal_position}]")

    def set_current_callback(self, msg: SetCurrent):
        goal_current = msg.current
        # Write Goal Current (length: 2 bytes)
        dxl_comm_result, dxl_error =
self.packet_handler.write2ByteTxRx(self.port_handler,msg.id,ADDR_GOAL_CURRENT,goal_current)

        if dxl_comm_result != 0:
            self.get_logger().error("Failed to write goal current.")
        else:
            self.get_logger().info(f"Set [ID: {msg.id}] [Goal Current: {goal_current}]")

    def get_position_callback(self, request, response):
        # Read Present Position (length: 4 bytes)
        present_position, dxl_comm_result, dxl_error =
self.packet_handler.read4ByteTxRx(self.port_handler,request.id,ADDR_PRESENT_POSITION,)

        if dxl_comm_result != 0:
            self.get_logger().error(f"Failed to read position for ID: {request.id}")
        else:
            self.get_logger().info(f"Get [ID: {request.id}] [Present Position:
{present_position}]")
            response.position = present_position  # Ensure 'position' is set correctly
            self.send_position = present_position

        return response

    def send_currentpos(self):

        if self.send_position is not None:
            pos_msg = Int32()
            pos_msg.data = self.send_position
            self.get_logger().info(f"pos_msg : {pos_msg}")
            self.position_publish.publish(pos_msg)
        else:
            self.get_logger().warn("Send position is not set. Waiting for data...")




          # Explicitly return the response object

    def get_current_callback(self, request, response):
        # Read Present Current (length: 2 bytes)
        present_current, dxl_comm_result, dxl_error = 
self.packet_handler.read2ByteTxRx(self.port_handler,request.id,ADDR_PRESENT_CURRENT,)

        if dxl_comm_result != 0:
            self.get_logger().error(f"Failed to read current for ID: {request.id}")
        else:
            self.get_logger().info(f"Get [ID: {request.id}] [Present Current: 
{present_current}]")
            response.current = present_current  # Ensure 'current' is set correctly
        return response  # Explicitly return the response object

def main(args=None):
    rclpy.init(args=args)
    read_write_node = ReadWriteNode()
    rclpy.spin(read_write_node)
    read_write_node.destroy_node()
    rclpy.shutdown()
```