Chapter 11

Build Dual Arms

When people hear the word "robots", most naturally think of humanoid robots like the Terminator. In reality, humanoid robots are advancing rapidly driven by major breakthroughs in generative AI, improved dexterity, and lower costs that are making mass production and commercial deployment feasible. They are moving from science fiction movies into the research labs, and increasingly, into practical applications. Both Boston Dynamics' Atlas and Tesla's Optimus represent leading efforts in humanoid robotics, though they differ in their design and commercial focus. In the foreseeable future, such robots are more likely to work alongside humans at home as assistants and can take on repetitive and physically demanding tasks like lifting and moving boxes or cleaning.

However, these humanoid robots are too expensive for hobbyists like us to afford. It is more realistic and much more affordable to build our own dual-arm robot platform to mimic human actions. In this chapter, we will explore how to realize such a kind of dual-arm system by installing two single arms on a common `T-type` torso, as well as how to extend ROS programming across multiple robot systems.

11.1 Dual Arm System Setup

The two `Hex7bot` robot arms can be mounted onto a T-type torso, which can be built with 2 T-slotted aluminum extruded bars as shown in Figure 11.1:

The following diagram in Figure 11.2 will show all connections between different components.

For simplicity's sake, two stand-alone `Hex7bot` systems (i.e., two host machines) are directly interconnected in the same network. We are going to run two of the same ROS launch files on the two host machines (Ubuntu 18.04) but with two different namespaces: `left_arm` and `right_arm` . This launch file should start its robot controller and hardware interface node at boot-up and subscribe to position commands from the master machine. The two host machines communicate with the master machine in the ROS space over a network connection (WiFi or Ethernet connection). Of course, you can combine two arms' host machines to one and run one launch file for both arms, which is not covered in this book.

In addition, we need to create a startup script that automatically run the roslaunch file on both arm's host machines automatically at system startup. Once both connecting to the master machine successfully, the system enters standby state. To reduce overall boot time, we can configure both Raspberry Pi machines to run headless by disabling the X Window System and booting the OS in text mode only. Only the

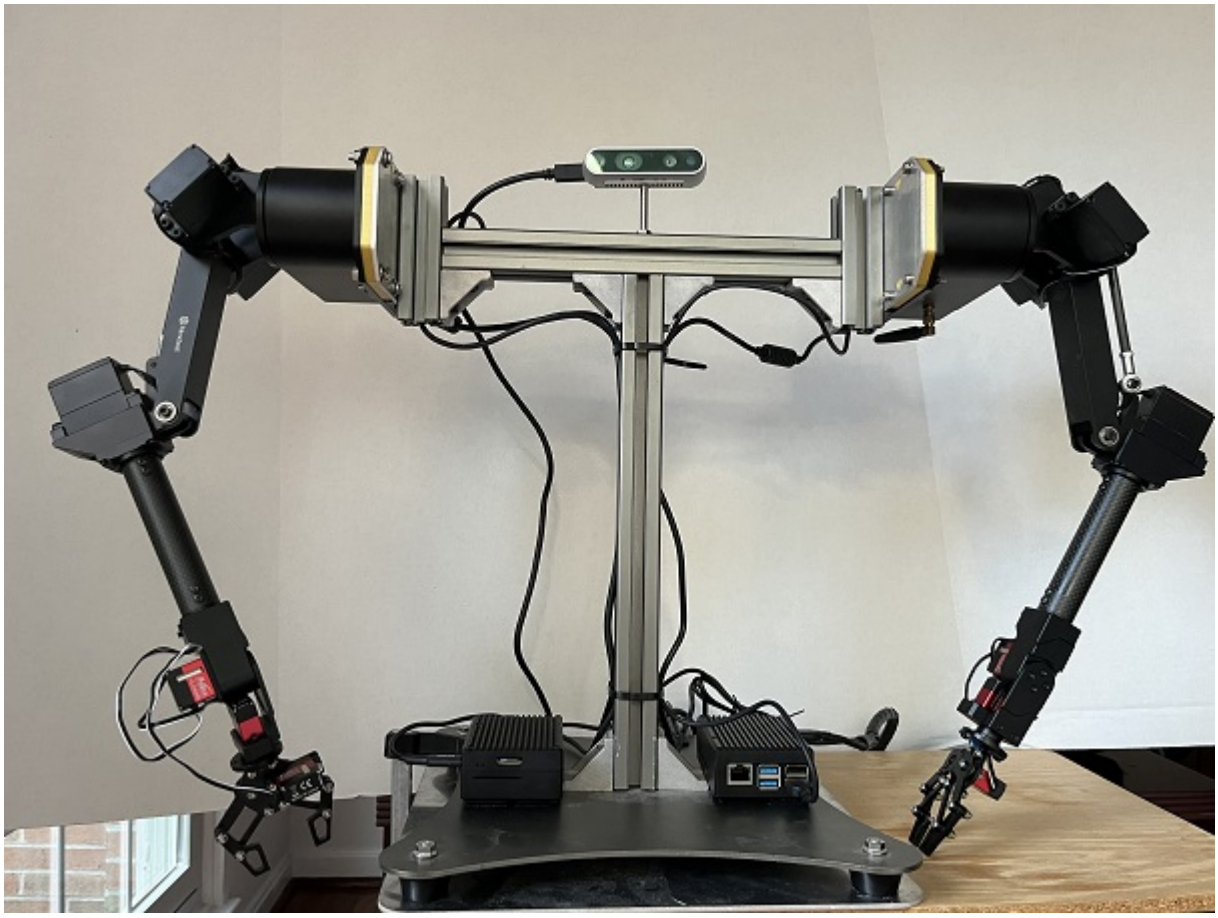

Figure 11.1: Dual robot arm platform

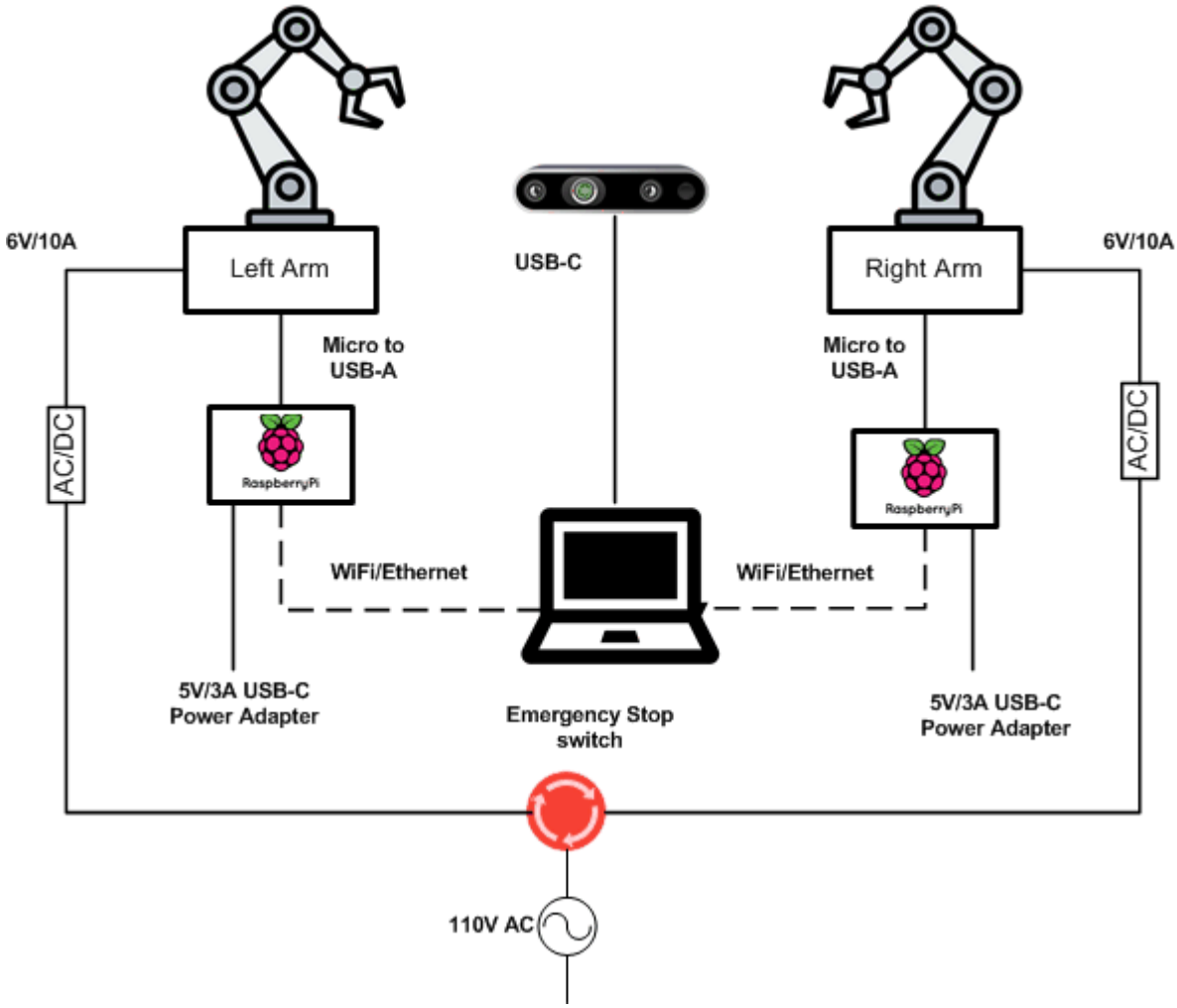

Figure 11.2: Dual robot arm connection diagram

master machine(a laptop) runs X windows, you can still remotely log into either machine via SSH for troubleshooting.

The master machine is a high-performance computer (e.g., a laptop) with a USB 3.0 port connecting the `Realsense D435` depth camera directly. This machine has the same ROS workspace but runs the master launch file for perception and sending commands to the left and right arms.

Of course, we can use either arm's host machine to connect the depth camera without the master host machine. The advantage of introducing the master machine:

(1) Compared with an embedded Raspberry Pi mini PC, a laptop or desktop as the master machine has more computation power and is capable of implementing advanced algorithms (e.g., perception/vision/machine learning) and real-time control either arm over ROS commands.
(2) ROS allows nodes to exchange messages across different machines via topics and services. ROS nodes on the arm's host machine subscribe to topics from the master machine, execute the commands, and report the arm status by publishing status messages back to the master machine.
(3) The master machine software evolution/updating is more frequent, while the arm host machine is seldom updated.

So let's start detailed configurations and installations.

All three computers (one master and two arm's host machines) must be configured in the same network, such as:

Hostname	Master	Left arm	Right arm
IP	192.168.2.1	192.168.2.2	192.168.2.3

Note: For reliable networking communication, it is better to set up a wired Ethernet network connection rather than a WiFi network. The WiFi network is reserved for remote `ssh` troubleshooting interface.

11.2 Configuring ROS Master

The ROS master is a central node in ROS that manages communication between other nodes. It needs to be running and accessible for ROS to function correctly.

There should be 1 master/1 ROS core for handling the system. In our case, a laptop running Ubuntu 18.04 is the ROS master. There is actually not much to do other than to just run `roscore` :

master:$ roscore

We can create the ROS master environment with a script file, such as *ros_master_env.sh*

```
# Define the static IP address of your ROS master
ROS_MASTER_STATIC_IP="192.168.2.1"
export ROS_IP="192.168.2.1"
export ROS_HOSTNAME="master" # localhost or the correct hostname IP for your setup
# ROS master URI (using IP address)
export ROS_MASTER_URI="http://${ROS_MASTER_STATIC_IP}:11311"
echo "ROS_MASTER_URI set to : ${ROS_MASTER_URI}"
# Add any environment variables you need here
KINETIC_DIRECTORY="/opt/ros/kinetic"
MELODIC_DIRECTORY="/opt/ros/melodic"
if [ -d "$KINETIC_DIRECTORY" ]; then
    echo "source ${KINETIC_DIRECTORY}/setup.bash"
    source ${KINETIC_DIRECTORY}/setup.bash
elif [ -d "$MELODIC_DIRECTORY" ]; then
    echo "source ${MELODIC_DIRECTORY}/setup.bash"
    source ${MELODIC_DIRECTORY}/setup.bash
else
    echo "Neither $KINETIC_DIRECTORY nor $MELODIC_DIRECTORY folder exists"
fi

source ~/projects/ros_ws/devel/setup.bash
cd ~/projects/ros_ws/
echo "current folder: $(pwd)"
```

Copy this script to `/etc/ros` with `sudo` and then create an auto start service called `roscore.service` to start this script.

To automatically start a roslaunch file on boot in Linux, a common and robust method is to wrap it as a `systemd` service. This ensures that the ROS environment is properly sourced and the launch file is executed after necessary system services are available.

- Create a systemd Service file: Create a .service file in */etc/systemd/system/* e.g., *roscore.service*

```
[Unit]
Description=ROSCore service
After=network-online.target

[Service]
Type=forking
User=master
Group=master
ExecStart=/bin/sh -c "\. /opt/ros/melodic/setup.sh; \. /etc/ros/ros_master_env.sh;
  roscore & while ! rostopic list | nc localhost 11311 > /dev/null; do sleep 1; done"
ExecStop=/usr/bin/pkill -f rosmaster
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
```

Deploy this service to */etc/systemd/system* fold with the following command

```
sudo cp /your/path/to/roscore.service /etc/systemd/system
```

Then reload, enable, and start this service:

```
sudo systemctl daemon-reload
sudo systemctl enable roscore.service
sudo systemctl start my-application.service
```

Verify the Service is running correctly with:.

```
systemctl status my-application.service
```

11.3 Configuring Left/Right Arms

To start a `roslaunch` file on boot-up, we do the same settings on the left/right host machines but as ROS clients.

1. Create a ROS client startup script, such as ros_startup.sh

```
#!/bin/bash

# Check if at least one argument is provided
if [ -z "$1" ]; then
    echo "Usage: $0 <argument>"
    exit 1
fi
#Define the expected value
LEFT_ARM="left_arm"
RIGHT_ARM="right_arm"
SCRIPT_DIR=$(dirname "${BASH_SOURCE[0]}")
echo "script folder is $SCRIPT_DIR"
# Check if the first argument ($1) is equal to the expected value
if [ "$1" = "$LEFT_ARM" ]; then
    echo "source '$1' enviroments."
    source ${SCRIPT_DIR}/left_client_env.sh
else
    echo "source '$1' environment."
    source ${SCRIPT_DIR}/right_client_env.sh
fi
# wait for ROS master node up running first
echo "run check_master_node..."
rosrun rbt_startup check_master_node
echo "recheck ROS_MASTER_URI :${ROS_MASTER_URI}"
# Execute roslaunch file
if [ "$1" = "$LEFT_ARM" ]; then
    roslaunch rbt_controllers hex7bot_left_arm_service.launch --wait
else
    roslaunch rbt_controllers hex7bot_right_arm_service.launch --wait
fi
```

Where *hex7bot_{left,right}_arm_service.launch* is described in 11.3.2.

and make the script executable:

```
chmod +x ~/scripts/ros_startup.sh
```

Please note that ROS_IP= “192.168.2.2” for the left arm and “192.168.2.3” for the right arm.

2. Create and auto start service file (*/etc/systemd/ros_serivce*) so that the ROS client can auto connect ROS master when system boots up.

```
[Unit]
Description=ROS Launch Auto-start service
After=network-online.target
After=getty.target
After=systemd-user-sessions.service

[Service]
Type=Simple   # simple or forking if your script fors a process
ExecStart=/home/ubuntu/projects/start_scripts/ros_startup.sh <left_arm/right_arm>
StandardOutput=file:/run/log/ros_out.log
StandardError=file:/run/log/ros_err.log
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
```

In addition, you may need to modify "/etc/hosts" file for the left and right arm machines to resolve the master's name to IP, e.g., for the left arm, add the following line to /etc/hosts in the format of "mastersIP address master name":

```
left_arm: $ sudo gedit /etc/hosts
    192.168.2.1    master
    192.168.2.2    left_arm
    192.168.2.3    right_arm
```

Configure the left/right arm's system service to start automatically during bootup using `systemd`, such as

```
$ sudo systemctl enable < service_name>
# check if the service is enable
systemcl is-enabled < service_name>
```

11.3.1 AutoConnect to ROS Master

To make the ROS client start up properly, we need to make sure the ROS master is active first and reachable before proceeding with any ROS-related operations (e.g., launching any ROS launching file). We can create a ROS node called check_master_node, whose role is to check and wait until the ROS Master is up and running.

When a ROS client attempts to connect to the ROS master (`roscore`) using `ros::master::check()` and the master is not yet running, the function call can appear to "hang" or block indefinitely. This behavior occurs because ros::master::check() is designed to establish a connection with the master and will wait until that connection is successful. If the master is not available, the connection attempt will continuously retry, causing the blocking behavior. To prevent this indefinite blocking and allow the client to wait for the master to become ready, a common approach involves implementing a loop with a delay, repeatedly calling `ros::master::check()` until it returns true. This allows the client to periodically check for the master's availability without completely freezing the application. Here is an example of how this can be implemented:

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

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

   ROS_INFO("Waiting for ROS Master...");

   // Loop until ros::master::check() returns true
   while (!ros::master::check()) {
     ROS_INFO_THROTTLE(5, "ROS Master not found. Retrying in 1 second...");
     ros::Duration(1.0).sleep(); // Sleep for 1 second
   }

   ROS_INFO("ROS Master is ready!");

   ros::spin(); // Keep the node alive

   return 0;
}
```

In this example, the while loop continuously calls `ros::master::check()`. If `ros::master::check()` returns false (meaning the master is not found), a message is printed (throttled to avoid excessive output), and the program pauses for 1 second using `ros::Duration(1.0).sleep()`. Once `ros::master::check()` returns true, the loop exits, and the client can proceed with its normal operations, knowing that the ROS Master is available (Figure 11.3).

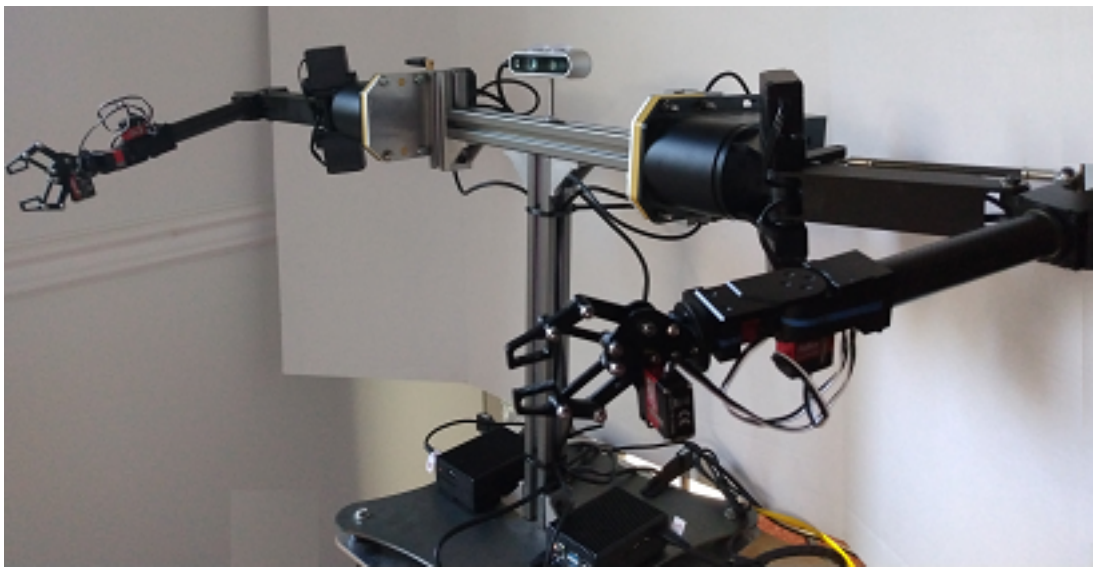

Figure 11.3: Dual arm start up

11.3.2 Establish communication between arms and master.

Next, we need to create a new launch file for the left and right arms so that when the system boots up, the client machine (Raspberry) can initialize its robot arm, run a self-diagnostic test (`BIT`) and establish communication with the master. To achieve this goal, we probably have to update all existing nodes and create new nodes if necessary. Especially, we need to distinguish the same type of ROS topics from left or right by pre-appending a new namespace such as `/left_arm` , or `/right_arm` .

For example, the right arm is running *hex7bot_right_arm_service.launch*

```
<launch>
  <arg name="robot_name" value="left_arm" />
  <arg name="show_rviz" default="false" />
  <arg name="add_gripper" default="true" />
  <arg name="velocity_control" default="false"/>
  <arg name="config"
    default="$(find motion_servo)/config/hex7bot_servo_config.yaml"/>
  <arg name="controller_to_spawn" default="joint_position_controller" />

  <!-- individual parameters in the /robot_name namespace -->
  <group ns="/$(arg robot_name)">
     <param name="robot_description" command="$(find xacro)/xacro --inorder
         '$(find rbt_descriptions)/urdf/hex7bot_left_arm.urdf.xacro'" />
     <!-- load hex7bot controller parameters into ROS parameter server
          for later controller_manager -->
     <rosparam
       file="$(find rbt_controllers)/config/hex7bot/hex7bot_controllers.yaml"
              command="load" />
     <rosparam
         file="$(find rbt_controllers)/config/hex7bot/hex7bot_params.yaml"
         command="load" />
     <rosparam
          file="$(find rbt_controllers)/config/hex7bot/hex7bot_jlimits.yaml"
         command = "load"  />
     <param name="gripper_control" type="bool" value="$(arg add_gripper)"/>
     <param name="velocity_control" type="bool"
                  value="$(arg velocity_control)"/>
```

```
    <!-- webservice server -->
    <node name="webservice_server" pkg="teleop" type="webservice_server"
                  output="screen"/>
    <!-- start HTTP server -->
    <node name="remote_control_web_server" pkg="webserver"
        type="webserver.py" output="screen"  args="8080" cwd="node">
       <param name="ros_queue_size" type="int" value="1" />
       <param name="step_gain" type="double" value="1.25" />
       <param name="joint_vel" type="double" value="0.5" />
       <param name="linear_pos" type="double" value="0.2" /> </node>
    <!-- command (position and velocity) checking with low pass filter-->
    <node name="motion_servo" pkg="motion_servo" type="servo_server"
       output = "screen" ><rosparam command="load" file="$(arg config)"/>
    </node>
    <!-- load  hardware_interface::RobotHW:
    <node  name="hex7bot_ros_controller" pkg="rbt_controllers"
         type="hex7bot_controller_node"  respawn="false" output="screen"/>
    <!-- load the controllers -->
    <node name="position_controller_spawner" pkg="controller_manager"
          type="spawner" respawn="false"
        output="screen"  args="joint_state_controller
          $(arg controller_to_spawn)"/>
    <!-- load gripper action controller -->
    <node if="$(eval add_gripper)" name="gripper_controller_spawner"
          pkg="controller_manager" type="spawner" respawn="false"
          output="screen"  args="gripper_controller"/>

     <!-- publish TF using joint_states from hex7bot_controller_node
        remap joints_states, listen "/joint_state", actually listen
           "$(arg robot_name)/joint_states"-->
     <node name="robot_state_publisher"  pkg="robot_state_publisher"
               type="robot_state_publisher" >
        <remap from="/robot_description"
               to="/$(arg robot_name)/robot_description"/>
        <remap from="/joint_states" to="/$(arg robot_name)/joint_states"/>
      </node>
  </group>

  <!-- robot rviz display -->
  <node if="$(arg show_rviz)"  name="rviz" pkg="rviz" type="rviz"
    args="-d $(find rbt_descriptions)/rviz/urdf.rviz"  required="true"/>
    <param name="loop_rate" value="50.0" />
</launch>
```

Please note that in this launch file, all nodes except Rviz are running under the namespace of `left_arm` or `right_arm` .

11.3.3 Test communication between arms and master

1. First check both robot arms are up and running by their topics, such as

```
ptshi@peitao-Latitude-E5500:~/projects/start_scripts$ rostopic list
/left_arm/delta_joint_cmds
/left_arm/delta_twist_cmds
/left_arm/gripper_controller/gripper_action/cancel
/left_arm/gripper_controller/gripper_action/feedback
/left_arm/gripper_controller/gripper_action/goal
/left_arm/gripper_controller/gripper_action/result
/left_arm/gripper_controller/gripper_action/status
/left_arm/joint_position_controller/command
/left_arm/joint_states
/left_arm/motion_servo/delta_joint_cmds
/left_arm/motion_servo/delta_twist_cmds
/left_arm/motion_servo/status
/right_arm/delta_joint_cmds
/right_arm/delta_twist_cmds
/right_arm/gripper_controller/gripper_action/cancel
/right_arm/gripper_controller/gripper_action/feedback
/right_arm/gripper_controller/gripper_action/goal
/right_arm/gripper_controller/gripper_action/result
/right_arm/gripper_controller/gripper_action/status
/right_arm/joint_position_controller/command
/right_arm/joint_states
/right_arm/motion_servo/delta_joint_cmds
/right_arm/motion_servo/delta_twist_cmds
/right_arm/motion_servo/status
/rosout
/rosout_agg
/tf
/tf_static
```

We can open two terminals on the ROS master machine and verify whether the two arms have established communication with the master or not, such as by running left_arm key joy node script:

```
#!/bin/bash

# Define the static IP address of your ROS master
ROS_MASTER_STATIC_IP="192.168.2.1"

export ROS_IP="192.168.2.1"

export ROS_MASTER_URI="http://${ROS_MASTER_STATIC_IP}:11311"
echo "ROS_MASTER_URI set to : ${ROS_MASTER_URI}"
# Add any environment variables you need here

source /opt/ros/melodic/setup.bash

echo "current folder: $(pwd)"
source /home/master/projects/ros_ws/devel/setup.bash
cd ~/projects/ros_ws/
source devel/setup.bash
# Execute roslaunch file
rosrun teleop keyboard_jog __ns:=/left_arm
```

Similarly for the right arm, in either window, the user can control the robot arm (either left or right), individual joints in joint space or Cartesian space, the gripper open/close actions etc. Please refer to chapter 9 for detailed command information.

If both robot arms can be controlled as you expect, although they are slow and not so responsive (which is another topic: how to improve the system response by fine-tuning every ROS node with a higher loop rate, etc.). Congratulations, your dual arms are ready for advanced tasks/tasks.

11.3.4 Improve system response

- Disable Arms' X Windows when booting up. The following commands will do what you need:

```
sudo systemctl set-default multi-user.target
```

To enable or revert the GUI on boot:

```
sudo systemctl set-default graphical.target
```

To disable the GUI temporarily:

```
sudo systemctl isolate multi-user.target
```

To enable the GUI temporarily:

```
sudo systemctl isolate graphical.target
```

- Adjust all nodes' loop rate. Warning: it will take a lot of effort to fine-tune all running nodes' looping rates to improve response.
- Disable log output, such as changing the ROS log level from INFO to ERROR to suppress INFO, WARNING level messages.

Other advanced topics including build special version Ubuntu kernel by disabling some unnecessary process and modules, which is out of this book topics.

11.4 Summary

In this chapter, we focused primarily on software configurations for building a dual-arm platform, though many detailed construction techniques beyond our scope. To simplify deployment, we simply duplicated one robot arm setup - without detailing the process. (In practice, you could use a single Raspberry Pi to control both arms). You might also consider using `Yocto` to maintain a custom OS image and packages for future products. Regarding network connectivity, you must decide between a wired Ethernet connection(requiring cables) or a wireless Wi-Fi connection. However, Wi-Fi is less reliable than Ethernet, especially for high-bandwidth data streams such as point clouds, and is therefore not recommended for such applications.

If you are building a dual-arm platform, one crucial topic that cannot be ignored is workspace design: Two arms' working spaces are overlapped slightly to allow for collaboration without excessive interference. If possible, a 3D depth sensor should be mounted at a location that it can cover nearly the entire working area. Additionally, you will need to collect the 3D position information and parameters of both arms and the sensor, create a new configuration file. In most cases, you will also need to calibrate the sensor relative to the robot base. This can be done using known reference points measured with a ruler or laser tools to determine the transformation matrix between the sensor and robot base. Once these steps are completed, the platform is ready for advanced task execution and motion planning - topics that will be covered in the following chapters.