Report 3 - ArUco Detection with ROS2 Server-Client Infrastructure

Pranav Mintri

January 2026

1 Introduction:

ArUco markers are either 4x4, 6x6 or 7x7 markers, which have a black boundary, and have a set number of IDs in their predefined dictionaries (which are commonly used). For example, generally there are 50 IDs for 4x4 markers, numbered from 0 to 49, each of which has a black-and-white ArUco marker intelligently designed to be very distinct from one another, such that even in problematic conditions, there is no error in reading those markers. There are several parity bits included in the ArUco markers as well, for the sake of consistency. Additionally, each ArUco marker must have a specific pre-defined orientation, which must be the sole correct orientation. This leads to the elimination of several symmetric ArUco markers.

Owing to all the reasons listed above (and perhaps more), the number of IDs for a given marker size is significantly less than standard logic would dictate. For example, a 4x4 marker size must lead to 2^{16} or 65,536 possible IDs, but the number of IDs in predefined dictionaries are generally much smaller than that - 50 is an example.

This assignment required a client node in ROS2 to read an image (or in this case a string of images in the form of a video - either passed as an argument, or captured from the device's camera devices) and send that image to a server node.

Further, that server node was required to scan the string of images for ArUco markers. The IDs of those markers, and their bounding boxes' coordinates had to be stored in a 1D array and a 2D array respectively. Those would be sent back to the client, and would be displayed by the client.

2 The Custom Interface:

Before moving to the servers and clients themselves, interfaces is an important topic to touch on. Interfaces in ROS2 can be used to define the types of messages that are to be transferred between nodes - in this case, the client and the server.

For this assignment, I decided to make a custom interface, because that made the project significantly cleaner and easier to follow. Also, it made the transfer of data in the form of 2D arrays possible.

The process for creating a custom interface is as simple as creating a package in your workspace (while creating this package however, `ament_python` cannot be used; `ament_cmake` has to be used, but Python codes can be included in the package). This process was outlined in the official ROS2 Humble documentation.

First, we create a package:

```
ros2 pkg create --build-type ament_cmake --license Apache-2.0 aruco_interface
```

where `aruco_interface` is the name of the custom interface.

Further, we created two directories `msg` and `srv` in our package/interface `aruco_interface`.

We wish for the transfer of an image from the client to the server, the transfer of an array from the server to the client, and finally the transfer of an array of arrays (2D array) to the client as well. Here is how we made that possible:

In the directory `msg`, we made a file `BoundingBox.msg`, with the following content:

```
float32[] corners
```

In the directory `srv`, we made a file `SendImage.srv`, with the following content:

```
sensor_msgs/Image image_data
---
int32[] ids
aruco_interface/BoundingBox[] bboxs
```

The `---` separates the service sent by the client to the server (above), and the message to be sent by the server to the client (below).

To send an image from the client to the server, we use the standard ROS2 package `sensor_msgs`, which is all we require.

To send the 1D array of the IDs, we use a 32-bit integer array.

To send the coordinates of the bounding boxes, we send an array of the "datatype", or more accurately message type `aruco_interface/BoundingBox`, which is itself a message that consists of an array (32-bit floating point array). `corners[]` stores the four corners' coordinates, and `bboxs[]` stores various values of `corners[]`.

The `CMakeLists.txt` file also has to be edited, to account for the change in dependencies and the build of the interface. For approximately the same reason,

`package.xml` also has to be changed. The edited files have been given in the GitHub remote repository.

3 The Server Node:

We shall create a new package `aruco_srvcli`, within which we will create the two nodes - `server_node` and `client_node`. Let us first discuss `server_node`.

```
#!/usr/bin/env python3
import sys
import rclpy
from rclpy.node import Node
import cv2
import cv2.aruco as aruco
import numpy as np
import os
from cv_bridge import CvBridge
from aruco_interface.msg import BoundingBox
from aruco_interface.srv import SendImage
# We are importing cv_bridge to convert the data received from the client into an image.

# Get the image from the client.
# Convert the image to grayscale.
# Get the dictionary.
# Create detector parameters.
# Create the detector object.
# Store bboxs, ids, rejected.
# Append bboxs and ids to the ids and bboxs arrays.
# Send bboxs and ids to the client.

class ServerNode(Node):
    def __init__(self):
        super().__init__('server_node')
        # Create the service
        self.srv = self.create_service(SendImage, 'send_image', self.aruco_callback)
        self.bridge = CvBridge()
        # self.get_logger().info("ArUco Service Server is ready.")

    def aruco_callback(self, request, response):
        # Get the image from the client and convert to OpenCV format.
        cv_img = self.bridge.imgmsg_to_cv2(request.image_data, desired_encoding='bgr8')

        # Convert the image to grayscale.
        imggray = cv2.cvtColor(cv_img, cv2.COLOR_BGR2GRAY)
```

```
        # Get the dictionary (Using OpenCV 4.7+ syntax).
        # Assuming 4x4 markers with 50 total markers.
        aruco_dict = aruco.getPredefinedDictionary(aruco.DICT_4X4_50)

        # Create detector parameters.
        parameters = aruco.DetectorParameters()

        # Create the detector object.
        detector = aruco.ArucoDetector(aruco_dict, parameters)

        # Store bboxs, ids, rejected.
        bboxs, ids, rejected = detector.detectMarkers(imggray)

        # Prepare lists for the response.
        ros_ids = []
        ros_bboxes = []

        # Append bboxs and ids to the arrays if any are detected.
        if ids is not None:
            # ids is a 2D numpy array [[id1], [id2]], flatten it to [id1, id2].
            # For this Python libraries related stuff, AI was used by me.
            ros_ids = ids.flatten().astype(int).tolist()

            for marker_corner in bboxs:
                # Each 'marker_corner' is shape (1, 4, 2).
                new_bbox = BoundingBox()
                # Flatten the (1, 4, 2) into a 1D list of 8 floats.
                new_bbox.corners = marker_corner.flatten().astype(float).tolist()
                ros_bboxes.append(new_bbox)

        # Send bboxs and ids to the client.
        response.ids = ros_ids
        response.bboxs = ros_bboxes

        self.get_logger().info(f"Detected {len(ros_ids)} markers.")
        return response

def main():
    rclpy.init()
    node = ServerNode()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    rclpy.shutdown()
```

```
if __name__ == '__main__':
    main()
```

I have tried to outline the function of all lines through comments in the program (which has led to the ridiculous size of the program).

First, we have imported the necessary libraries and facilities which were required in this program. An important point to note is that we also imported the message and service components of the interface we created earlier into this program.

In `main()`, we have created an object of `ServerNode()`, and then we have executed the node. Two functions have been defined as part of its functionality - `__init__` and `aruco_callback`.

`__init__` is just used to create a server node. A message can also be displayed on the terminal using `self.get_logger().info("")`, which I have commented out here.

`aruco_callback` performs the core functionality of the node. Three arguments have been passed into the function - `request`, `response` and `self`.

`self` is standard Python. This is used to reference the node itself, and is used to call functions like `get_logger` (to print messages on the terminal) and `bridge` (to convert the image into a proper format for OpenCV to decode and process).

`request` is the data that is sent to the server by the client. In this case, that data is an image.

`response` is an empty ROS2 object that matches the datatype of the response which will be given by the server to the client.

The program starts by getting the image from the client, and converting it into OpenCV format. That image is, by convention, converted to grayscale. This is done for convenience in reading the ArUco markers. Colour does not matter in ArUco markers, only the grayscale intensity does. We then get a predefined dictionary with 4x4 markers and 50 defined IDs. Then, as per the new OpenCV syntax and methods, we create the detector parameters and then the detector object using those parameters. This used to be done in a different way in earlier versions of OpenCV, but now this method is used.

We proceed by storing the bounding box coordinates and the IDs, and sending them to the client. Some processing needs to be done to ensure that the format of the arrays is managed well. The server also prints a message on the terminal at a particular frequency, stating the number of markers detected at any point of time.

4 The Client Node:

```
#!/usr/bin/env python3
import gdown # This library is for automatic downloading and
# processing of Google drive links.
import sys
import os
import rclpy
from rclpy.node import Node
import cv2
import numpy as np
from cv_bridge import CvBridge
from sensor_msgs.msg import Image
from aruco_interface.srv import SendImage

class ClientNode(Node):
    def __init__(self, video_source): # to pass an argument into the client,
    # we have added the parameter video_source in the function
        super().__init__('client_node')

        # Setup Service Client
        self.client = self.create_client(SendImage, 'send_image')
        while not self.client.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('Waiting for ArUco Server...')


        if video_source.isdigit():
            # If it's a digit (e.g., "0"), convert to int.
            source = int(video_source)
        elif "drive.google.com" in video_source:
            # If it is a drive link, download it.
            source = self.download_from_drive(video_source)
        else:
            # This is if it's a local video.
            source = video_source

        self.cap = cv2.VideoCapture(source)
        # Instead of directly opening the webcam, above, we have tried
        to pass an argument into the client.
        # self.cap = cv2.VideoCapture(0) # Open webcam.

        self.bridge = CvBridge()

        # Timer to drive the capture loop (approx 30 FPS)
        self.timer = self.create_timer(0.03, self.process_frame)
        self.get_logger().info("Client started.")
```

```
def download_from_drive(self, url):
    # Create a temporary filename.
    output = 'temp_drive_video.mp4'
    # gdown handles the extraction of the file ID automatically.
    gdown.download(url, output, quiet=False, fuzzy=True)
    # fuzzy=True allows the script to extract the FILE_ID even if
    you provide the full "view" link rather than the direct download link.
    return output

def process_frame(self):
    success, frame = self.cap.read()
    if not success:
        return

    # Convert OpenCV image to ROS 2 message. For this, AI was used by me.
    ros_image = self.bridge.cv2_to_imgmsg(frame, encoding="bgr8")

    # Prepare and send request.
    request = SendImage.Request()
    request.image_data = ros_image

    # We use call_async so the UI (imshow) doesn't freeze.
    future = self.client.call_async(request)
    future.add_done_callback(lambda f: self.response_callback(f, frame))

def response_callback(self, future, frame):
    try:
        response = future.result()

        # Process the nested response.
        if response.ids:
            self.get_logger().info(f"IDs found: {response.ids}")
            # Loop through each detection.
            for i in range(len(response.ids)):
                marker_id = response.ids[i]

                # Unpack the nested BoundingBox.msg.
                # Reconstruct from flat list [8] to NumPy array [4, 2]. Once again,
                # for this Python-libraries related stuff, I used AI.
                corners = np.array(response.bboxs[i].corners).reshape((4, 2)).
                                                              astype(np.int32)
                # Printing the coordinates of the bounding boxes:
                print(f"\nMarker ID: {marker_id}")
                print(f"  Top-Left:     ({corners[0][0]:.2f}, {corners[0][1]:.2f})")
                print(f"  Top-Right:    ({corners[1][0]:.2f}, {corners[1][1]:.2f})")
```

```
                    print(f"  Bottom-Right: ({corners[2][0]:.2f}, {corners[2][1]:.2f})")
                    print(f"  Bottom-Left:  ({corners[3][0]:.2f}, {corners[3][1]:.2f})")

        except Exception as e:
            self.get_logger().error(f"Service call failed: {e}")

def main():
    rclpy.init()
    video_source = sys.argv[1] if len(sys.argv) > 1 else '0'
    node = ClientNode(video_source)
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.cap.release()
        cv2.destroyAllWindows()
        # Clean up the download file if it exists.
        if os.path.exists('temp_drive_video.mp4'):
            os.remove('temp_drive_video.mp4')
            # This ensures that the large video file is deleted from the device
            once the ROS2 node is closed.
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

We start by importing. An important thing to note here is that we have imported a library called `gdown`. This is for downloading Google Drive video files whose links can be passed as arguments into the client node. 0 can be passed as an argument, if the webcam video feed has to be captured. The argument number may be different depending on the device, but in general, it is 0.

We have added a parameter `video_source` in `__init__`, to facilitate the passing of arguments while running the node via the terminal. This time, `__init__` has more functions than just setting up the client node. If the argument passed into the client node is a digit, that argument will be converted into an integer, and that will become the source of the video. (for example, `cv2.VideoCapture(0)` captures the string of images from the webcam feed) If the argument is a drive link, the video will get downloaded onto the device using the `gdown` library. A function `download_from_drive` will get implemented to download the video from the URL passed as an argument into the client node, and further into the function `download_from_drive` itself. If the argument is the link/path to a file stored locally on the device, the video can be played directly from the source. `gdown`, as expected, is not required.

`process_frame` is a function which transfers the string of images from the client to the server. To start with, the OpenCV image is converted to a ROS2 message, which can be transferred via the ROS2 transfer package `sensor_msgs` without issues. The message is then sent to the server.

The next line, wherein we use `call_async`, is especially important, because this ensures that the code doesn't freeze in waiting for the response. If this was not used, the code would freeze to wait for the response, which will have created a deadlock. Instead, by using `call_async`, after sending the request, the control moves to the next line, without freezing to wait for the response.

In the function `response_callback`, we simply deconstruct the message sent by the server, and display the coordinates of the bounding boxes, and the IDs of the ArUco markers.

The `main()` function works as normal, with one important detail. If, we keep on downloading files from Google Drive onto our local device, without deleting them, it would be an impediment to the memory of our device. Hence, after the node has been spun (i.e. the processing has been done), removal of the downloaded file is essential. This is exactly the additional function that has been carried out in `main()`.

5 Changes to `setup.py` and `package.xml`:

We must add the required dependencies to the `package.xml` file, and the required entry points to the `setup.py` file.

Additions to `setup.py`:

```
entry_points={
    'console_scripts': [
        'server_node = aruco_srvcli.server_node:main',
        'client_node = aruco_srvcli.client_node:main',
    ],
},
```

The format of addition of entry points remains the same:

```
'<name_of_executable> = <name_of_package>.<name_of_file>:main'
```

Additions to `package.xml`:

```
<depend>cv_bridge</depend>
<depend>sensor_msgs</depend>
<depend>aruco_interface</depend>
<depend>rclpy</depend>
```

6 Demonstration:

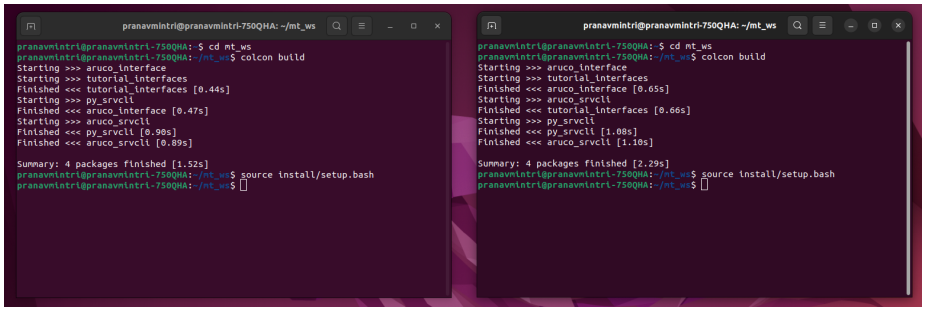

Figure 1: Opening two terminal windows, building the packages from the root of the workspace `mt_ws`, and sourcing `install/setup.bash` in both terminals.

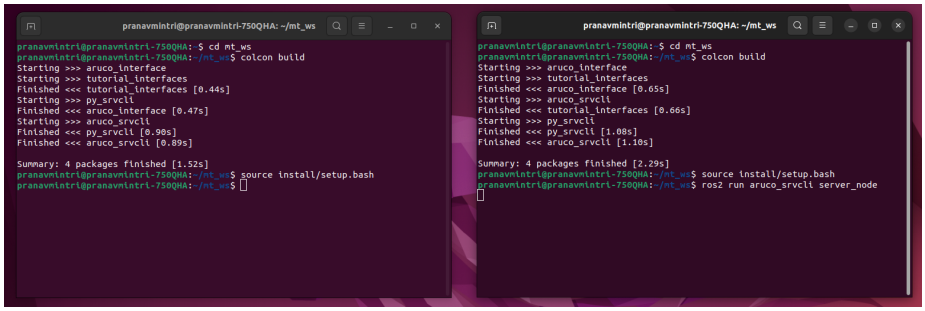

Figure 2: Running the server node in one terminal window.

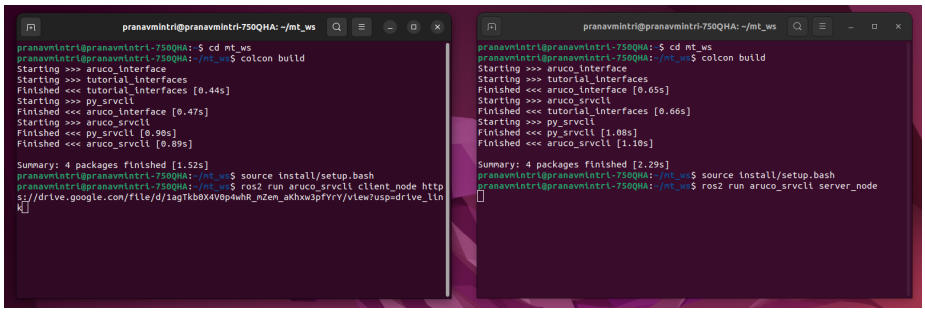

Figure 3: Entering the run command for the client node, with the drive link of a video as the argument.

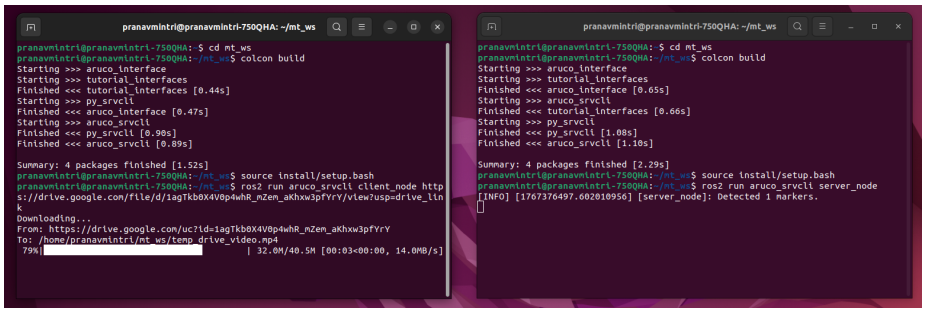

Figure 4: Running the client node. (video downloading)

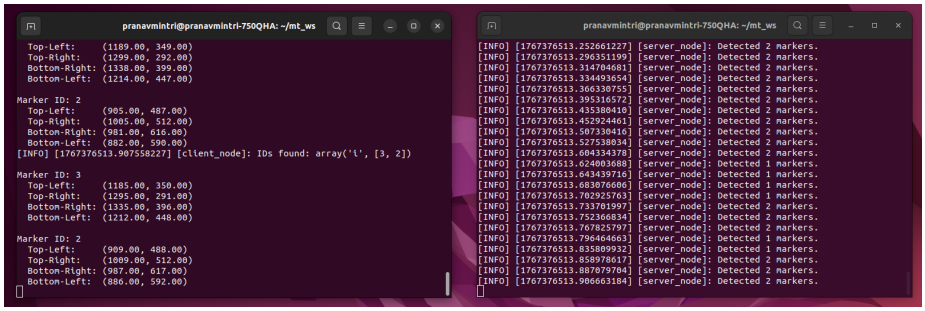

Figure 5: Detection of ArUco markers.