Overview

The project aims to showcase the safety features and efficiency of the QNX RTOS by utilizing the FLEXIV Rizon 4 robot arm. The task involves constructing a pyramid using Aruco tagged blocks while dynamically detecting a prosthetic arm entering its workspace and continues the task while ensuring safe operation around the intruding arm.

Requirements

The project requires multiple key components that need to operate together for the demo to be successful.

1. Robot must be able to complete the basic task of constructing the pyramid

   a) Recognize the next cube to grab
   b) Move to the location and grab the cube
   c) Determine the destination for the cube to place
   d) Move to the location and place
   e) Repeat process until pyramid is built

2. Incorporate the dynamic arm detection and alter the movement functionality accordingly.

   a) Recognize intruding arm
   b) determine the space that the intruder occupies c. determine a path around the object

Pyramid Construction Plan

The robot is already able to recognize the Aruco cubes and publish data about their location and orientation. We will take this data and move the robot using Flexiv primitives which gets passed (x,y,z) coordinates as a string.The gripper functionality is already implemented which will be copied over to our demo. The ROS2 nodes will publish and subscribe to each other to access the topic data while the Robot Control will be a service so continuous feedback is sent to Mediator to determine the next course of action for the arm.

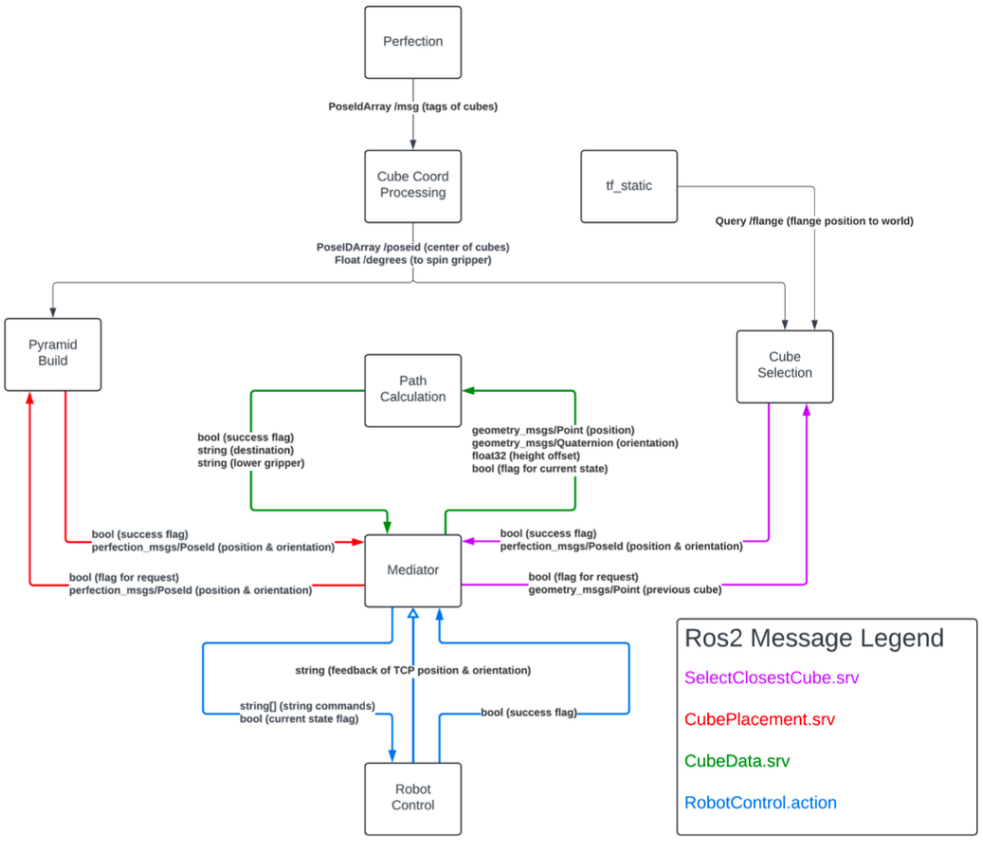


Figure 1: Message Passing of Block_Construction ROS2 Nodes

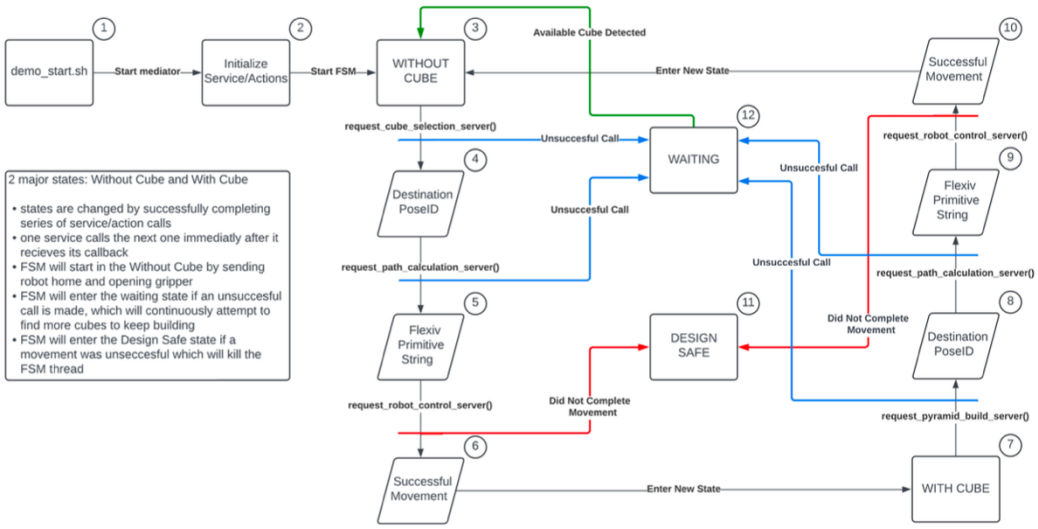


Figure 2: Finite State Machine in Mediator Node

NODAL DECOMPOSITION

Perfection: collects cube face recognition data based off of the tag visualizer, which further employs the use of Aruco markers. The tags data published from this module is fed into our first module - 'CubeCoordProcessing'.

CubeCoordProcessing - The CubeCoordProcessing node processes cube coordinates detected by a camera. It subscribes to /cam1/tags3d, filters and transforms these PoseIds from the camera frame to the world frame, and publishes the results.

Key Functionalities -

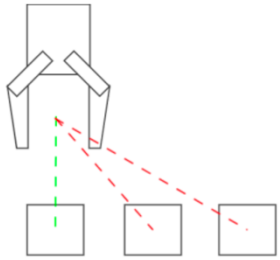
1. Camera Data Subscription: 'subscription_tags3d_' Subscribes to the '/cam1/tags3d' topic, which publishes PoseIdArray messages containing cube positions, orientations and IDs. The data will then be processed by this nodes tags_cb function
2. Cube PoseId Processing (tags_cb function):
   a. Filtering: Filters incoming PoseIds based on cube orientation, retaining only the side faces of cubes.
   b. Transforming: Converts filtered PoseIds from the camera frame to the world frame using tf2 transformation library. c. Publishing: Publishes transformed PoseIds to the /transformed_tags3d topic.
3. PoseId Adjustment (offset_pose_z): Adjusts the Z-axis of a PoseId to get the cube's physical center before transformation.
4. PoseId Filtering (cube_filter): Filters out unwanted PoseIds based on the cube's Z-axis alignment, ensuring only relevant PoseId are processed.
5. TF2 Transformation: Manages transformations between coordinate frames using tf2_ros::Buffer and tf2_ros::TransformListener.

ClosestCubeSelect: The ClosestCubeSelect node identifies the closest available cube to a robot's gripper. It processes incoming 3D pose data and provides a service to select the nearest cube, ensuring that each cube is used only once.

Key Functionalities -

1. Subscription to Transformed Cube Data: 'subscription_tags3d_' Subscribes to the /transformed_tags3d topic, receiving the PoseIds of cubes in the world frame. The data will then be processed by this nodes tags_cb function
2. Service to Select Closest Cube: 'service_' provides the select_closest_cube service, which identifies the closest unused cube to the robot's gripper and returns it.
3. Cube PoseId Processing (tags_cb function):
   a. Updates the list of available cubes with all received PoseIds.
4. Closest Cube Selection (handle_service):
   a) Gripper Position Retrieval: Retrieves the current position of the gripper in the world frame, using a transform_lookup.
   b) Cube Selection: Determines the closest cube by calculating the Euclidean distance between the gripper and each cube. Ensures that previously selected cubes are not reused.

5. Thread Safety: Uses a mutex to ensure thread-safe access to the list of 'available cubes'.



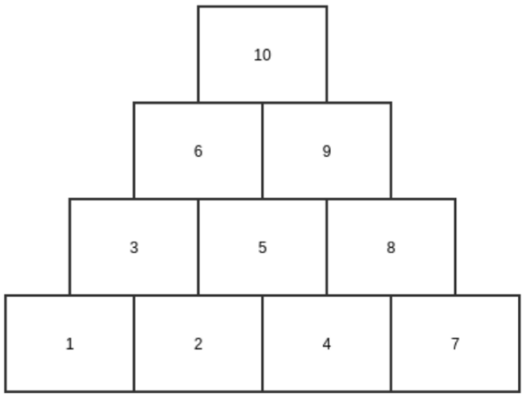
Pyramid Build: Tells the robot where to place a single cube in order to assemble a pyramid.

Key Functionalities -

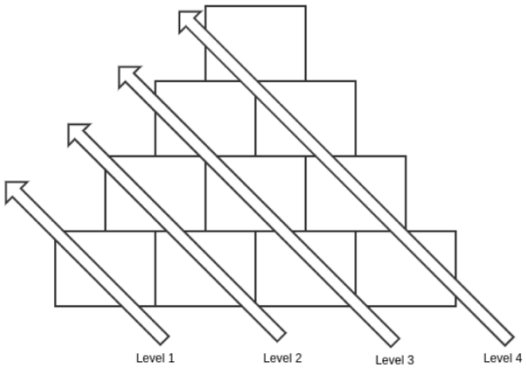
1. Service: pyrm_build service of type CubePlacementSrv. Get bool request and PoseID previous_cube to process
2. Callback: destination_response calculates a PoseId where the next cube will be placed to build the pyramid

    a. If it is the first block of the pyramid, it will be placed on the build_tag
    b. If it is the first block of a level, it will be placed beside the first block of the previous level c. If it is any other block, it will be placed with respect to the previous block

3. The build order and levels are described as the following:



Order in which blocks will be placed.



PathCalculation: The PathCalculation node calculates the path for a robot gripper based on cube position, height offset, and orientation. It offers a service to determine the necessary movements for the gripper to reach and interact with the cube.

Key Functionalities -

1. Service to Calculate Path: 'path_calc_' provides the path_calculator_service, which formulates a Flexiv Primitive approved string with a destination coordinate and TCP orientation based on the cube's position, height offset, and orientation.

2. Path Response Handling (path_response):

a) Request Handling: Responds to the service request by calculating the destination and the gripper's path.
b) Conditional Logic: Adjusts the response based on whether the calculation involves a cube being gripped.
c) Success Determination: Sets result_success to true if the calculated paths are valid.
d) Path Calculation uses the path_calculator.hpp and gripper_control.hpp header files in the inlude directory to build the movement strings and determine the correct degrees for the gripper to spin.

Mediator: The MediatorClient class serves as the central controller for overseeing the robot's interactions with cubes and associated tasks. It establishes connections with various ROS2 services and actions to execute operations like path calculation, pyramid construction, and cube selection. Additionally, the class employs a finite state machine (FSM) to manage different operational states, including robot movement (with or without a cube), waiting periods, and error handling. To ensure responsiveness, it utilizes separate threads for FSM execution and robot movement control.

Key Functionalities -

1. Initialization of ROS2 Node and Clients:

a) The MediatorClient class inherits from rclcpp::Node and is responsible for coordinating various ROS2 services and actions.
b) The constructor initializes multiple clients for interacting with different services and actions, such as CubeDataSrv, CubePlacementSrv, SelectClosestCubeSrv, and Robot_control.

2. Finite State Machine (FSM) Implementation:

a) The FSM runs in a separate thread and cycles through various robot states, including:

I. ROBOT_STATE_TO_DESTINATION_WITHOUT_CUBE
II. ROBOT_STATE_TO_DESTINATION_WITH_CUBE
III. ROBOT_STATE_WAITING
IV. ROBOT_STATE_DESIGN_SAFE

b) The FSM handler (fsm_handler()) handles transitions between these states based on the success or failure of various service calls.

3. Service Requests:

a) The Mediator class contains methods for interacting with services such as request_path_calculation_server(), request_pyramid_build_server(), and request_cube_selection_server().
b) Each method sends a request to the respective service, waits for the response saves it to a method variable and error handles the result.

4. Action Request:

a) The request_robot_control_server() method sends movement commands to the robot using the Robot_control action message.
b) Depending on the current FSM state, it adjusts the robot's behavior, such as setting the robot_waiting_state to true when in the ROBOT_STATE_WAITING state

5. Error Handling:

a) The FSM transitions to a safe state (ROBOT_STATE_DESIGN_SAFE) in case of errors during robot_control action calls.
b) If a movement critical service call fails or does not return, appropriate warnings are logged, and the robot may transition to a waiting state.

6. Thread Management: The FSM runs in a dedicated thread (fsm_thread_), which is properly joined in the destructor to ensure a clean shutdown.

7. The conditional variable unlocks the mutex that waits on the robot's completion of its movement

Robot Control: The RobotControl module is an action server designed to control a Flexiv robotic arm equipped with a gripper. It provides an interface for sending a series of commands to the robot, managing the execution of these commands, and offering real-time feedback on the robot's state. This module is intended for applications requiring precise robotic movements, including tasks that involve interacting with objects using the gripper.

Key Functionalities -

1. Action Server Initialization (robot_control_server_): Initializes an action server that listens for goals to control the Flexiv robot. It handles goal reception, cancellation, and execution by spinning a separate thread for each accepted goal.
2. Robot and Gripper Initialization (initialize_robot, initialize_gripper):
   a. Robot Initialization: Connects to the Flexiv robot using specified IP addresses, clears any faults, and ensures the robot is in a ready state. b. Gripper Initialization: Initializes the gripper attached to the robot, enabling it to grasp or release objects as needed.
3. Command Execution (execute):
   a) Sequential Command Processing: Processes a list of commands in the received goal, clearing faults and executing each command on the robot.
   b) Gripper Control: Manages the gripper based on the current gripper state, executing either a grasp or release action.
4. Feedback and Result Handling (execute):
   a) Real-time Feedback: Continuously provides feedback to the client, including the robot’s TCP position and orientation during command execution.
   b) Result Handling: Sets the result (finished_movement) to indicate whether the goal was successfully completed or aborted due to errors.
5. Fault Detection and Handling (Check_fault): Monitors the robot for faults during operation, attempts to clear detected faults, and stops the robot if the fault cannot be resolved.
6. Primitive Movement Completion Check (completedPrimitiveMovement): Checks whether the robot has completed a primitive movement within a specified timeout blocking the thread, ensuring that the robot does not progress without finishing or get stuck during execution.
7. Joint State Publishing (publishJointStates): Periodically publishes the robot's joint states, including positions, velocities, and efforts, to the /joint_states ROS topic, enabling other components to monitor the robot’s state.
8. Robot Operations Management (Commence_operation): Executes initial operations such as moving the robot to a home position, zeroing force /torque sensors, and ensuring the gripper is in the open position before processing any commands.