Detailed Analysis of ROS 2 Interaction Schema and Repository Structure

Technical Document

October 4, 2025

Abstract

This document provides a detailed analysis of an interaction schema between ROS 2 nodes and the Dobot CR5. The roles of the nodes, topics, and configuration files are clearly distinguished. Implementation specifics and modifications made to the components for the correct functioning of the system are also included. Finally, the structure of the files responsible for the Dobot's movement is presented.

1 Main Components of the ROS 2 System

The system is composed of different types of entities that cooperate for the control and monitoring of the robot.

1.1 Nodes (Executable Processes)

The nodes are the active computational elements of the system, each with a specific task.

- **STATIC_TRANSFORM_PUBLISHER**: This node is responsible for publishing geometric transformations fixed over time. In the context of this system, it is used to establish the static transformation between the `world` frame and the robot's `base_link`, acting as the root for the transformation tree (TF). Transformations are published on the `/tf_static` topic.
- **ROBOT_STATE_PUBLISHER**: A fundamental node for visualization and planning. It subscribes to the `/joint_states` topic (to receive the current joint state) and `/tf_static` (for fixed transformations). Using this information and the robot description (URDF/Xacro), it calculates and publishes the dynamic transformations between all robot links on the `/tf` topic.
- **MOVE_GROUP**: The central component of the MoveIt! framework, the `move_group` node is in charge of planning and executing complex

trajectories for the robot. To do this, it subscribes to several key topics, including `/tf`, `/tf_static`, and `/joint_states`. It also uses specific configuration files such as *KINEMATICS.YAML* (for kinematic parameters), *JOINT_LIMITS.YAML* (for joint operating limits), and *PILZ_CARTESIAN_LIMITS.YAML* (for Pilz Cartesian kinematics). Once a trajectory is generated, the **MOVE_GROUP** sends it to its configured controller, which in this case is the **TRANSLATER** node.

- **TRANSLATER**: This node acts as a low-level interface for the robot. It receives movement commands, specifically *FollowJointTrajectory* messages sent by the **MOVE_GROUP** node (masquerading as its controller, as defined in *ROS2_CONTROLLER.YAML*). The **TRANSLATER** translates these commands into specific actions for the robot's physical hardware, utilizing the services made available by the Dobot manufacturer.
  - *Implementation Note*: To ensure the correct operation of the system, the *feedback.py* file provided by the company was modified. In particular, line 120 was updated so that the Dobot publishes joint positions on the `/joint_states` topic (instead of `/joint_state_robot`), and line 134 was added to allow the correct functioning of **RViz**.
- **RViz**: The main 3D visualizer of ROS (Robot Visualization). Its role is fundamental for monitoring and debugging the system. It subscribes to various topics to receive data and present it graphically in a 3D environment, allowing the user to observe the state of the robot and the surrounding environment.

1.2 Topics, Message Types, and Data Systems

These are the channels and data structures used for communication between nodes.

- `JOINT_STATES` (*sensor_msgs/JointState*): This is both the ROS 2 message type and the conventional topic name (typically `/joint_states`) on which information about the state of each robot joint (such as position, velocity, and/or effort) is published.
- TF (Transform Frames): The ROS transformation system, implemented through the `tf2` library in ROS 2. TF is not a single topic, but a hierarchical data structure that tracks the poses of all frames in space. Nodes publish transformation messages (*geometry_msgs/TransformStamped*) on standard topics like `/tf` and `/tf_static`.
- `/tf`: The standard topic used for publishing dynamic transformations, which change over time (e.g., the pose of moving robot links).
- `/tf_static`: The standard topic for publishing static transformations, which do not vary over time.

1.3 Key Configuration Files

These files define the properties and operating parameters of the robot and its components.

- *cr5_robot.urdf.xacro*: Robot description file that defines its kinematics, links, joints, and physical properties. It is the model on which the **ROBOT_STATE_PUBLISHER** and the **MOVE_GROUP** are based (it calls the *cr5_robot.urdf* file).
- *cr5_robot.srdf*: The Semantic Robot Description Format (SRDF) complements the URDF, defining joint groups, predefined states, and limitations that are useful for movement planning (used by the **MOVE_GROUP**).
- *kinematics.yaml*: Contains specific parameters for the robot's kinematics, used by the **MOVE_GROUP**'s planners for inverse kinematics (IK) and forward kinematics (FK) calculations.
- *joint_limits.yaml*: Defines the operating limits (position, velocity, acceleration) for each robot joint, essential for safe movement planning by the **MOVE_GROUP**.
- *moveit_controllers.yaml*: Fundamental configuration file for MoveIt! which defines the controllers that **MOVE_GROUP** will use for trajectory execution. In this case, it specifies the **TRANSLATER** node as the robot controller.
- *pilz_cartesian_limits.yaml*: Configuration for the Pilz motion planner, often used for more predictable and safer trajectories in industrial environments.

2 Data Flows and Main Interactions

This section describes how the nodes communicate with each other through topics to coordinate robot operations.

2.1 Robot Initialization and State

→ The fixed transformation between `world` and `base_link` is established by the **STATIC_TRANSFORM_PUBLISHER**, which *publishes* on the `/tf_static` topic.

→ The current state of the robot joints (`JOINT_STATES`) is *published* by the **TRANSLATER** (thanks to the modification in *feedback.py*) on the `/joint_states` topic.

→ The **ROBOT_STATE_PUBLISHER** *subscribes* to `/joint_states` and `/tf_static`. Based also on *cr5_robot.urdf.xacro*, it *calculates* the transformations of all robot links and *publishes* them on the `/tf` topic.

2.2 Movement Planning and Execution

→ The **MOVE_GROUP** *subscribes* to the `/tf`, `/tf_static`, and `/joint_states` topics to have a complete view of the current state of the robot and the environment.

→ Using the configuration files (*KINEMATICS.YAML*, *JOINT_LIMITS.YAML*, *PLIZ_CARTESIAN_LIMIT.YAML* etc.), the **MOVE_GROUP** *plans* the trajectories for the robot.

→ Once planned, the trajectory is *sent* by the **MOVE_GROUP** to the **TRANSLATER** in the form of a *FollowJointTrajectory* message. The **TRANSLATER** is configured to read the messages from the controller configured in *moveit_controllers.yaml* (MoveIt uses this file to say which controller is responsible for transmitting the path to be taken, in this case it is configured to use the **FollowJointTrajectory** controller).

→ The **TRANSLATER** *executes* the trajectory, translating the commands into hardware actions for the Dobot (TCP/IP services provided by the company).

→ Generally, the *ros2_controller.yaml* file would be used instead of the **TRANSLATER**, but this is not possible in our case since we do not have a hardware interface.

→ The **RViz** node is a *data consumer*. It *subscribes* to the `/tf` and `/tf_static` topics to build the transformation tree and correctly position all elements in 3D space.

→ **RViz** also *subscribes* to the `/joint_states` topic. This allows it to display the robot's 3D model with the joints in the correct positions and to animate the robot in real-time as it moves.

2.3 Node Launch

This ROS 2 launch script, `full_bringup.launch.py`, is the main entry point for starting the entire CR5 robot software architecture, including the MoveIt configuration and the necessary nodes for control and visualization. The following nodes are launched:

→ **STATIC_TRANSFORM_PUBLISHER**: This node (`static_tf`) from the `tf2_ros` package publishes a static transformation between the `world` and `dummy_link` frames. It is essential for establishing the root of the transformation tree (TF tree) and providing a fixed reference point in space.

→ **ROBOT_STATE_PUBLISHER**: The `robot_state_publisher` node is responsible for reading the robot description (`cr5_robot.urdf.xacro`) and publishing the dynamic transformations between all robot links on

the `/tf` topic. This allows TF to keep the position of every part of the robot updated.

→ **MOVE_GROUP**: This is the central node of the MoveIt! framework. It is configured with the robot's parameters (kinematic description, semantic, joint limits, etc.) through `MoveItConfigsBuilder`. Its main task is to plan trajectories, manage movement requests, and communicate with the robot controllers for execution. It uses planning pipelines like OMPL and Pilz Industrial Motion Planner.

→ **TRANSLATER**: The `translater_node` (from the `dobot_bringup_v3` package) acts as a low-level interface for the physical robot. It receives movement commands generated by **MOVE_GROUP** and translates them into specific instructions for the Dobot CR5 APIs.

→ **LISTENER**: The `listener_node` (also from the `dobot_bringup_v3` package) is responsible for listening to data coming from the ZED camera (see the following section [Software Architecture for Map Management]).

→ **RViz**: The `rviz2` node starts the ROS 3D visualizer. It loads a specific configuration file (`moveit.rviz`) for visualizing the CR5 robot and its environment. It allows real-time monitoring of the robot's state, planned trajectories, and the transformation tree.

3 Implementation Details and Specific Modifications

This section highlights the changes to the configuration files and the code provided by the company to adapt the system.

- *cr5_robot.urdf*: To include the representation of the End-Effector in the robot model, the lines between 264 and 291 of this file were added, extending the robot's kinematic description.
- *feedback.py*: As mentioned, this script was modified (line 120 changed and line 134 added) to ensure that the Dobot correctly publishes joint data on the `/joint_states` topic, essential for the **ROBOT_STATE_PUBLISHER** and **RViz**.

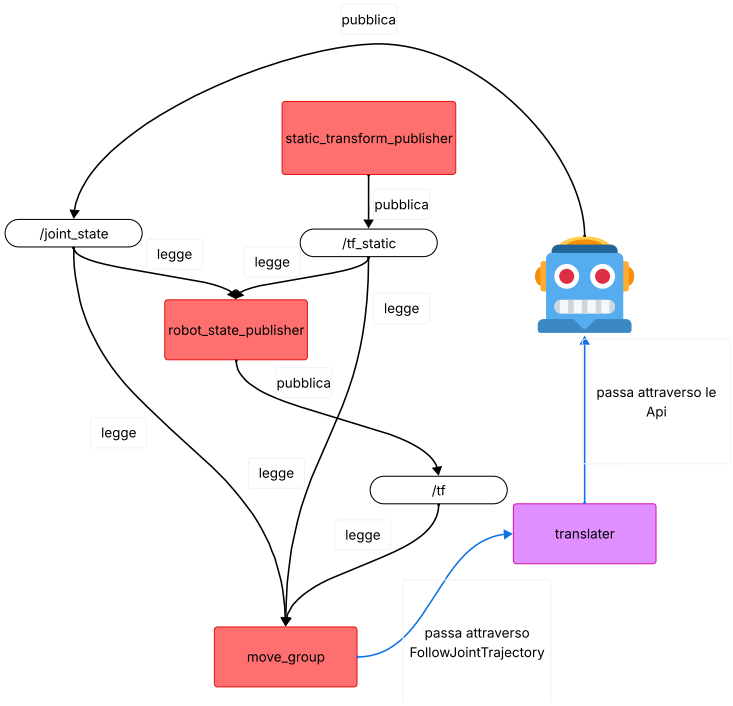

Figure 1: Diagram of ROS 2 node interaction for CR5 robot control.

4 Software Architecture for Map Management

This section describes the software structure and the main files involved in the process of data acquisition (simulated or real from a ZED camera), its reprocessing, and subsequent use within the ROS 2 system. The goal is to provide an overview of the code organization and the responsibilities of each component.

4.1 File Roles

The map management logic is distributed across several files within a ROS 2 package.

- *fake_zed.cpp*: This node is a simulator of the ZED camera. Its main function is to *generate and publish a fictitious initial map* on the `Boing` topic. This map includes the work space and the definition of some objects, including those classified as "target" (e.g., plants) and "obstacles". It is primarily used for testing and development purposes when a physical ZED camera is not available. The message type used for publishing is

custom_messages/msg/Map (if the zed is not used, it is better to remove the **LISTENER** node from the `dobot_bringup_v3` launch file responsible for communication with the real ZED).

- *reworked_map.cpp*: This is the central node for *map reprocessing*. It subscribes to the `Boing` topic to receive the raw map published by **fake_zed** (or by a real ZED). Its main function is to enrich the description of "target" objects within the map. In particular, for each target object, it calculates one or more circumferences of "optimal points" around it. Each point on these circumferences is then evaluated based on:
  - Its reachability within the defined work area.
  - Its position relative to other objects (avoiding collisions).

  The **reworked_map** node publishes the modified and enriched map on the `reworked_map` topic, making it available to other system components.
- *move_cr5_node.cpp*: This script is responsible for planning and executing the dobot's movement. WARNING: see related documentation: modular_scanning.
- *move_cr5_node_modular.cpp*: This script is the same as the one above but is remodeled and divided into modules.

4.2 Launch files

- *scripts_launch.launch.py* (ONLY IN SIMULATION): This is the *ROS 2 launch script* that coordinates the execution of the nodes. It defines which nodes must be started and with which parameters. In the context of this system:
  - It launches the **reworked_map_node** (the executable derived from *reworked_map.cpp*), which handles map reprocessing.
  - It provides the option to launch the **fake_zed_node** (the executable derived from *fake_zed.cpp*) if a connection to a real ZED is not available. This allows testing the entire map processing pipeline in a simulated environment.

  This file is crucial for managing the entire application, allowing the entire system to be started with a single command.