

Dual-Arm Robotic Pick-and-Place System: A ROS-Based Implementation

**AUT.841-2024-2025-1 Robot Manipulators: Modeling, Control and Programming**

**Mikhail Silaev (151861415)**

**Juha Jalkanen (H291863)**

**Amirmohammad Daareyni (152308980)**

Table of contents

[Table of contents 2](#_Toc185357660)

[Introduction 3](#_Toc185357661)

[System Architecture 4](#_Toc185357662)

[Scene Spawner Node 4](#_Toc185357663)

[Random Pose Generator Node 5](#_Toc185357664)

[Motion Control Node 5](#_Toc185357665)

[Execution Workflow: 6](#_Toc185357666)

[Motion Planning: 6](#_Toc185357667)

[Object Manipulation: 7](#_Toc185357668)

[Feedback and Monitoring: 7](#_Toc185357669)

[Unified Launch File 7](#_Toc185357670)

[ROS Action for Pick-and-Place 7](#_Toc185357671)

[Setup Details 8](#_Toc185357672)

[Step 1: Create Workspace 8](#_Toc185357673)

[Step 2: Download and Configure Motoman Robot 8](#_Toc185357674)

[Step 3: Create Project Packages 8](#_Toc185357675)

[Step 4: Launch and Test the System 9](#_Toc185357676)

# Introduction

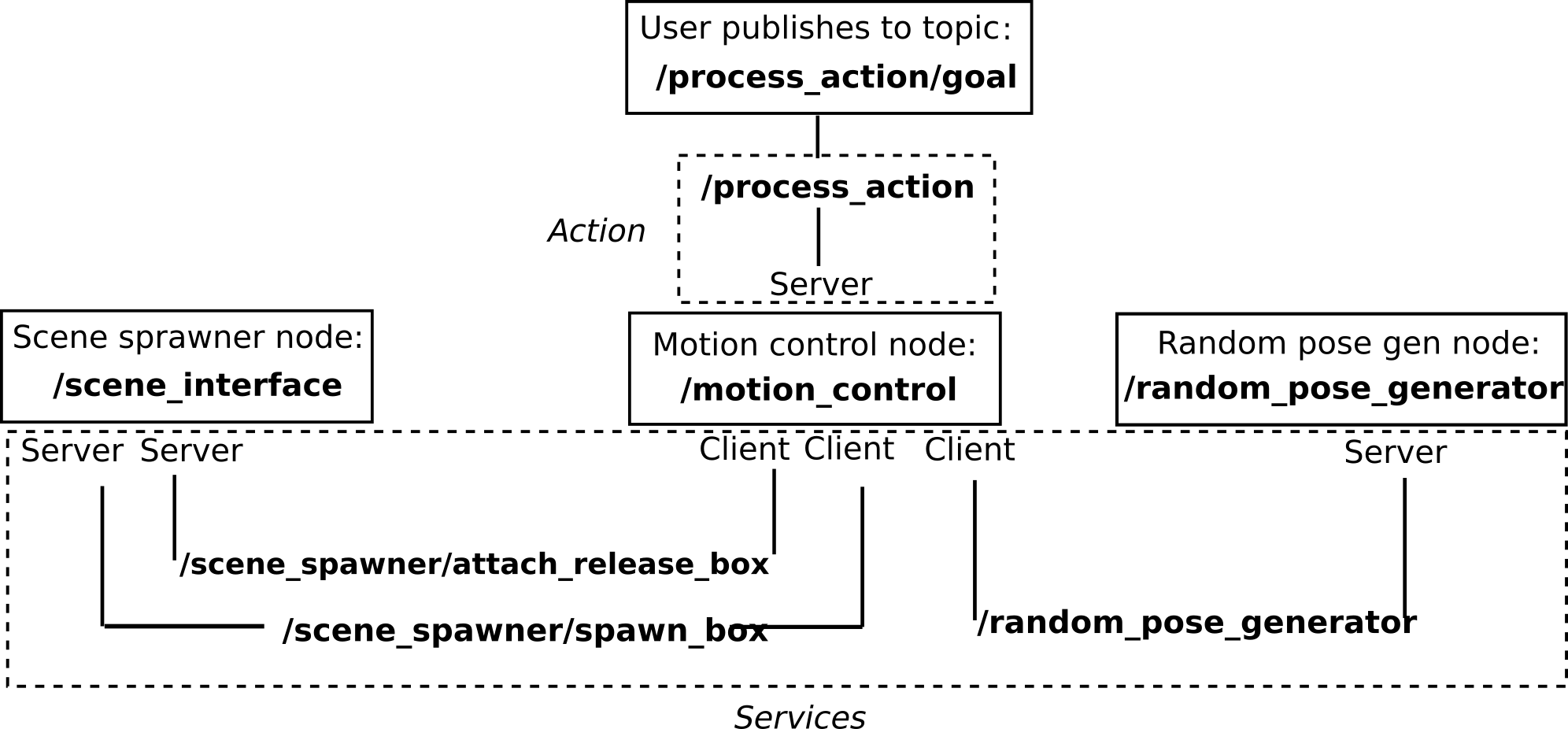
The dual-arm pick-and-place application represents a practical implementation of advanced robotic concepts, leveraging the Robot Operating System (ROS) to coordinate motion and object manipulation. The primary objective of this assignment was to design and implement a system where two robotic arms collaborate to perform pick-and-place tasks involving two boxes spawned at random positions. This task required the integration of multiple ROS nodes to ensure seamless operation, including scene setup, pose generation, motion control, and action feedback mechanisms.

The application builds upon the dual-arm robotic setup developed in the first part of Assignment 1. Besides the general **motoman** package the application includes four packages

* **my\_sia20d:** Contains URDF and XACRO configurations for the dual-arm setup consisting of two SIA20D robots and gripper defined in **wsg50\_gripper**.
* **wsg50\_gripper:** Contained URDF and configurations for the grippers attached to the robotic arms.
* **my\_moveit2** contains SRDF file with definition of Moveit motion groups for the dual-arm robot with grippers. It contains in particular definition of some common poses like home position, closed and open gripper. There are also other configuration and controller files.
* **motion\_test\_pkg** has the functionality related to the pick-and-place action specified in the assignment.

Packages **my\_moveit2** and **my\_sia20d** were developed by us in the Part 1 of the Assignment. The gripper package **wsg50\_gripper** was provided by the teachers. For this Part 2 of the Assignment we have developed a new package (**motion\_test\_pkg**) which includes three nodes responsible for (i) adding boxes to the scene – large box for green table, “base box” and “top box” at random coordinates; attaching “top box” to the robot links, (ii) planning and implementing robot motion, (iii) generating random coordinates of “top box” and “base box ”. Communication between these nodes is implemented using three Services. Communication between user and system is mplemented using ROS action which allows setting goals and to monitor the system's progress in real-time.

This report documents the steps taken to design, implement, and test the dual-arm pick-and-place application. It highlights the technical challenges encountered and the solutions developed to achieve a reliable and functional system.

Figure 1 System structure. The boxes represent ROS nodes, and the arrows show how they have to exchange information.

# System Architecture

The dual-arm pick-and-place application was developed using a modular architecture, comprising several interconnected ROS nodes. Each node performed a specific function to ensure the system's overall efficiency and functionality. Figure 1 illustrates the architecture of our robotic application. It has the same general structure as in the Assignment description, and details specific parts. The application consists of the following key components:

## Scene Spawner Node

The **scene spawner node (/scene\_interface)** was provided as part of the assignment package and formed the foundation of the system. This node interacts with MoveIt Planning Scene Interface to manage objects in the robot's environment. It clears pre-existing objects in the scene and adds a "table" object to serve as a base reference. It also provides two server interfaces to interact with **motion control node** **(/motion\_control) through** Services, which are

* + /scene\_spawner/spawn\_box: Accepts requests to add boxes to the scene at specific positions and dimensions.
  + /scene\_spawner/attach\_release\_box: Allows attaching or releasing objects from the robot's end effectors.

Message types for these services are **motion\_test\_pkg/BoxSpawner** and **motion\_test\_pkg/BoxAttach**, respectively defined in the corresponding .srv files in /srv folder.

## Random Pose Generator Node

This node (**/random\_pose\_generator**) has been developed by us to generate random X, Y coordinates upon request from the **/motion\_control** node.

1. Functionality:
   * The node generates random (x, y) coordinates within specified boundaries (X\_MIN, X\_MAX, Y\_MIN, Y\_MAX) and sets a fixed z value (Z\_HEIGHT = 0.7), representing the height of a table.
   * The pose is returned to the client as a service response.
2. Service:
   * Service Name: /random\_pose\_generator
   * Service Type: RandomPose (imported from motion\_test\_pkg.srv)
   * Request: No fields in the request are explicitly used; it only triggers the random pose generation.
   * Response: Contains three fields: x, y, and z (the generated pose coordinates).
3. Core Functions:
   * handle\_random\_pose\_request(req): Handles service requests by generating random (x, y) values within the defined workspace boundaries and setting z to a constant height. Logs the generated pose and returns it as a response using the RandomPoseResponse message

The algorithm for checking that box coordinates are reachable by robot and there are no collisions between boxes is implemented on the client side in **/motion\_control node.** In case the coordinates are not reachable or overlap with other objects conflicts the request to **random\_pose\_generator** nodeis repeated

## Motion Control Node

This node (**/motion\_control)** was developed by us.It acts as the central decision-making component of the system. It utilizes the MoveIt framework for planning and executing trajectories to handle objects in a coordinated manner. In order to provide the action interface with external user this node is realized as a **class MoveRobotServer** withaction server instance as an attribute:

self.server = actionlib.SimpleActionServer('/process\_action', MoveRobotAction, self.execute, False)

The main functionality is implemented in the callback class member function execute(goal), where goal is the argument passed to this function by the action server when the callback is invoked, that is when the goal prompt is resewed by action server from a client or corresponding topic.

### **Execution Workflow**:

The node waits for services **/scene\_spawner/spawn\_box** and **/scene\_spawner/attach\_release\_box** for object spawning and manipulation and service **/random\_pose\_generator** to generate random x,y coordinates for the base and top box objects. After the random coordinates are received, they boxes are spawned and robot arm1 or arm2 can reach them. This check is done in the class member function **execute()** by trying to build motion plan, such as

arm2.set\_pose\_target(pose\_stamped\_target\_offset\_wrist\_base)

success, plan\_arm2, \_, \_ = arm2.plan()

In case if box object cannot be reached, that is i success = False new coordinates are generated and the process repeats until success = True. This ensures that boxes are withing the robot workspace and that they don’t collide with robot parts. Separate check is done to ensure that “base box” and “top box” do not overlap.

The robotic arm to handle operation with top box is assigned dynamically based on the position of “base box” and “top box” relative to the base links of the robot 1 and robot 2. Coordinates of base links are found using **tf.listener** realized in the static method of the class

@staticmethod

def get\_robot\_coordinates(world\_frame = "world", base\_link\_frame = "one\_base\_link"): tf\_listener = tf.TransformListener()

tf\_listener.waitForTransform(world\_frame, base\_link\_frame, rospy.Time(0), rospy.Duration(3.0))

(trans, rot) = tf\_listener.lookupTransform(world\_frame, base\_link\_frame, rospy.Time(0))

return trans

### **Motion Planning**:

Trajectories for each arm and gripper are planned to use Moveit functionality. Targets are defined using “top box” and “base box” coordinates with precise offsets for grasping, as well as several poses defined in the Moveit SRDF configuration file in **my\_moveit2** package. To calculate poses relative to the robot's world frame we use kinematic transformations frame transformation as object\_frame\_top.p = PyKDL.Vector(top\_x, top\_y, top\_z) and frame to pose transform function frame\_to\_pose provided in course materials.

Depending on the relative positions between boxes and robot arms two types of motion are possible. (i) Both pick and place actions are implemented by one arm. It is realized when both boxes are closer to one of the arms. (ii) Initially “top box” is picked by one arm but the final place of ”top box” on the “base box” is implemented by the other arm. It is realized in case if “top box” is spawned closer to one arm and “base box” is closer to the other one.

In case (ii) the “top box” is trasferred between arms by implementing an intermediate pick and place operation in the middle point with coordinates between robot bases transfer\_x, transfer\_y, transfer\_z = (0, 0, 0.7) .

Object Manipulation:

Grippers are opened and closed at appropriate stages for object grasping and releasing. Objects are attached to or detached from the robot's grippers via service calls. The top box is picked by one arm and transferred to a base position managed by the other arm. If arms need to pass the box, an intermediate transfer pose is defined and executed.

### **Feedback and Monitoring**:

Regular feedback on status and actions are published in the action server topics **/process\_action/feedback** and **process\_action/result**. In principle it is possible to setup a client which listens to these topics, but it is not necessary for the present Assignment.

## Unified Launch File

A single **launch file (src/motion\_test\_pkg/launch/combined\_action.launch)** was created to streamline the execution of the system.

This launch file:

1. Initializes MoveIt 2 for motion planning and Rviz for visualization by including launch file of **my\_moveit2** package.
2. Launches three nodes from **motion\_test\_pkg** package shown in Figure 1, that is **\scene\_interface**, **\motion\_control** and **\random\_pose\_generator** nodes.

## ROS Action for Pick-and-Place

To manage the pick-and-place process, a ROS action was implemented. It offered the following functionalities:

* **Goal Execution:** Allowed external nodes or users to trigger the process.
* **Feedback Mechanism:** Published the status of the operation (e.g., picking, placing, transferring).
* **Result Reporting:** Communicated the successful completion or failure of the process.

This structured approach ensured modularity, enabling each node to focus on its specific task while contributing to the overall system functionality. The communication between nodes via ROS topics, services, and actions was critical in achieving a seamless workflow.

# Setup Details

The implementation of the dual-arm pick-and-place application involved multiple steps, beginning with the workspace setup and package creation and culminating in the execution of the system components. Below is a detailed description of each step:

## Step 1: Create Workspace

To organize the project, a dedicated workspace named assignment\_ws was created:

mkdir assignment\_ws

cd assignment\_ws

This workspace served as the root directory for all packages and configurations used in the project.

## Step 2: Download and Configure Motoman Robot

The Motoman robot package was downloaded and configured to set up the robotic arms:

git clone -b kinetic-devel https://github.com/ros-industrial/motoman.git src/motoman

rosdep update

rosdep install --from-paths src/ --ignore-src --rosdistro noetic

## Step 3: Create Project Packages

Copy project packages from the submitted zip file, that is folders **motion\_test\_pkg,**  **my\_moveit2, my\_sia20d, wsg50\_gripper**  to same **/src** floder where **motoman** package installed on previous step is located. Then run

catkin\_make

source devel/setup.bash

A blue and white object on a green surface

Description automatically generated

Figure 2 Dual-Arm Robotic Pick-and-Place System.

## Step 4: Launch and Test the System

After setting up the workspace and configuring the nodes, the system was tested using multiple ROS commands. These steps involved:

**1 Launching the Combined System:**

roslaunch motion\_test\_pkg combined\_action.launch

This command started the scene spawner, random pose generator, and motion control nodes, as well as the visualization environment. Figure 2 shows the robotic system.

**2 Sending a Pick-and-Place Command:** A goal (string “start”) is sent to the motion control action server /**process\_action** without using client, just directly publishing the following message to the topic **/process\_action/goal**:

rostopic pub /process\_action/goal motion\_test\_pkg/MoveRobotActionGoal "header:

seq: 0

stamp:

secs: 0

nsecs: 0

frame\_id: ''

goal\_id:

stamp:

secs: 0

nsecs: 0

id: ''

goal:

command: 'start'"

**Monitoring Feedback:** The feedback from the pick-and-place action was monitored using topic **/process\_action/feedback**:

rostopic echo /process\_action/feedback

**Checking Results:** The results of the pick-and-place operation were validated reading messages in the topic /**process\_action/result**:

rostopic echo /process\_action/result

**Final Testing and Demonstration**

After integrating all components, the system was tested for:

1. Robot 1 performing both pick-and-place operations.
2. Robot 2 performing both pick-and-place operations.
3. Robot 1 picking the box, with Robot 2 placing it (and vice versa).

The results were recorded in a demonstrator video, available in the recordings folder, to showcase the system's functionality.