Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

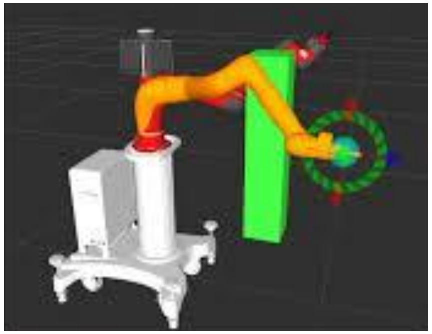
Learning ROS for Robotics Programming

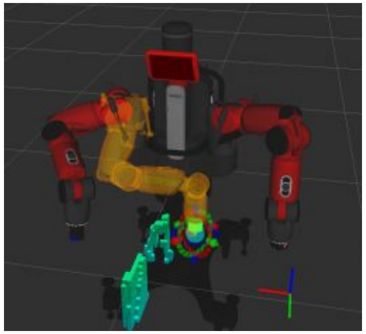
- MoveIt! -

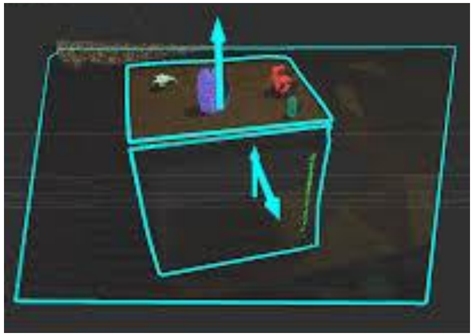
alberto.gottardi@phd.unipd.it
alberto.bacchin.1@phd.unipd.it
anna.polato@unipd.it

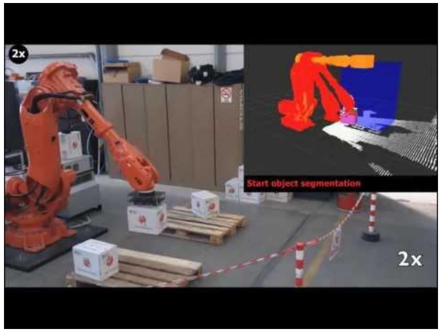
- About MoveIt!
- MoveIt! Pipeline
  - The Planning Scene
  - move_group
  - Motion planning
  - Kinematics
  - Collision Checking
- Simple Motion Planning tutorials
  - Motion Planning with RViz
  - Move Group Interface
  - Main Node

- **MoveIt!** is a set of tools for
  - Motion planning (OMPL, CHOMP, STOMP, Pilz);
  - Kinematics;
  - Mobile manipulation;
  - Trajectory proceeding and execution.



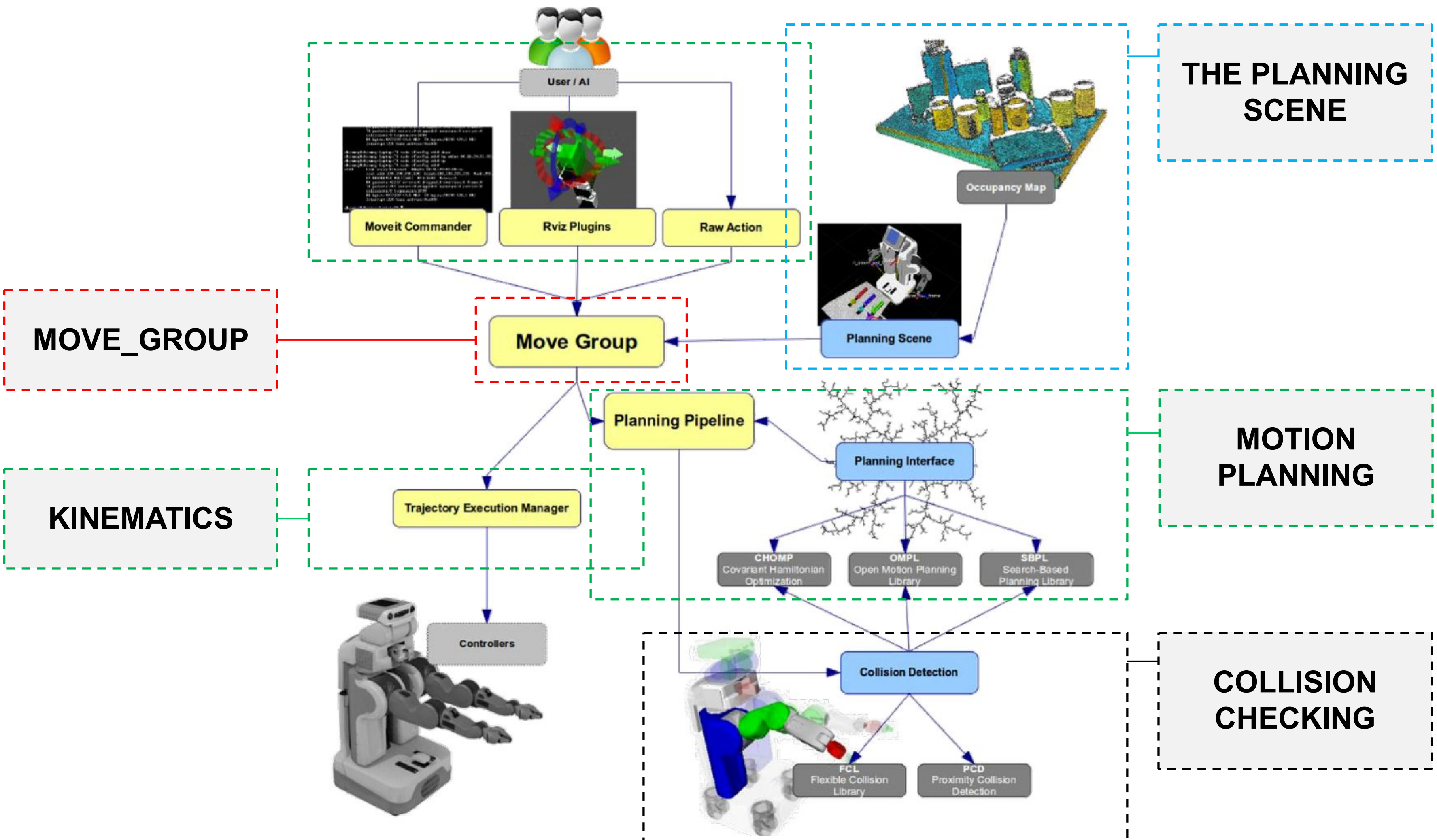






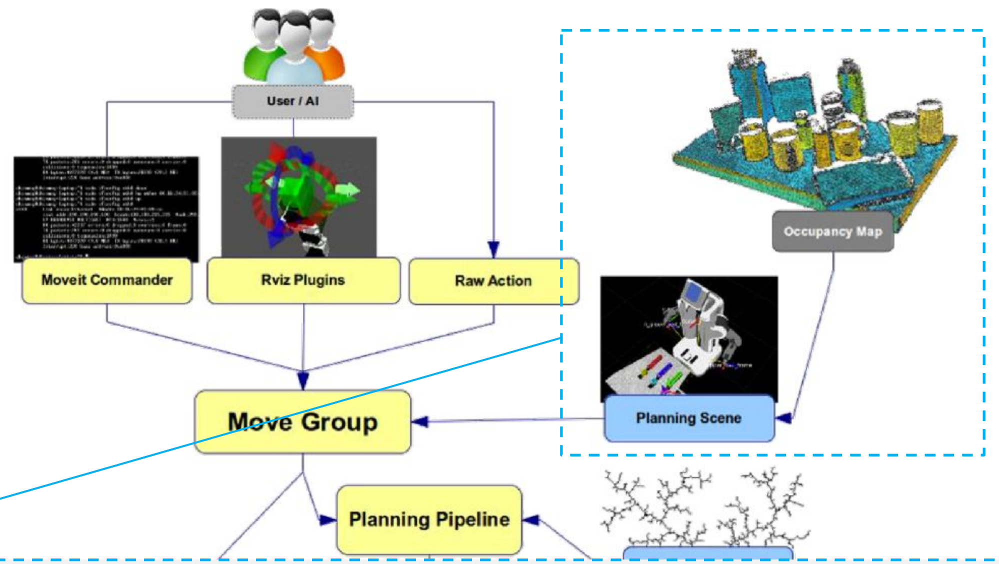
- About MoveIt!
- MoveIt! Pipeline
  - The Planning Scene
  - move_group
  - Motion planning
  - Kinematics
  - Collision Checking
- Simple Motion Planning tutorials
  - Motion Planning with RViz
  - Move Group Interface
  - Main Node

MoveIt! PIPELINE



THE PLANNING SCENE

MoveIt! Pipeline



Components:

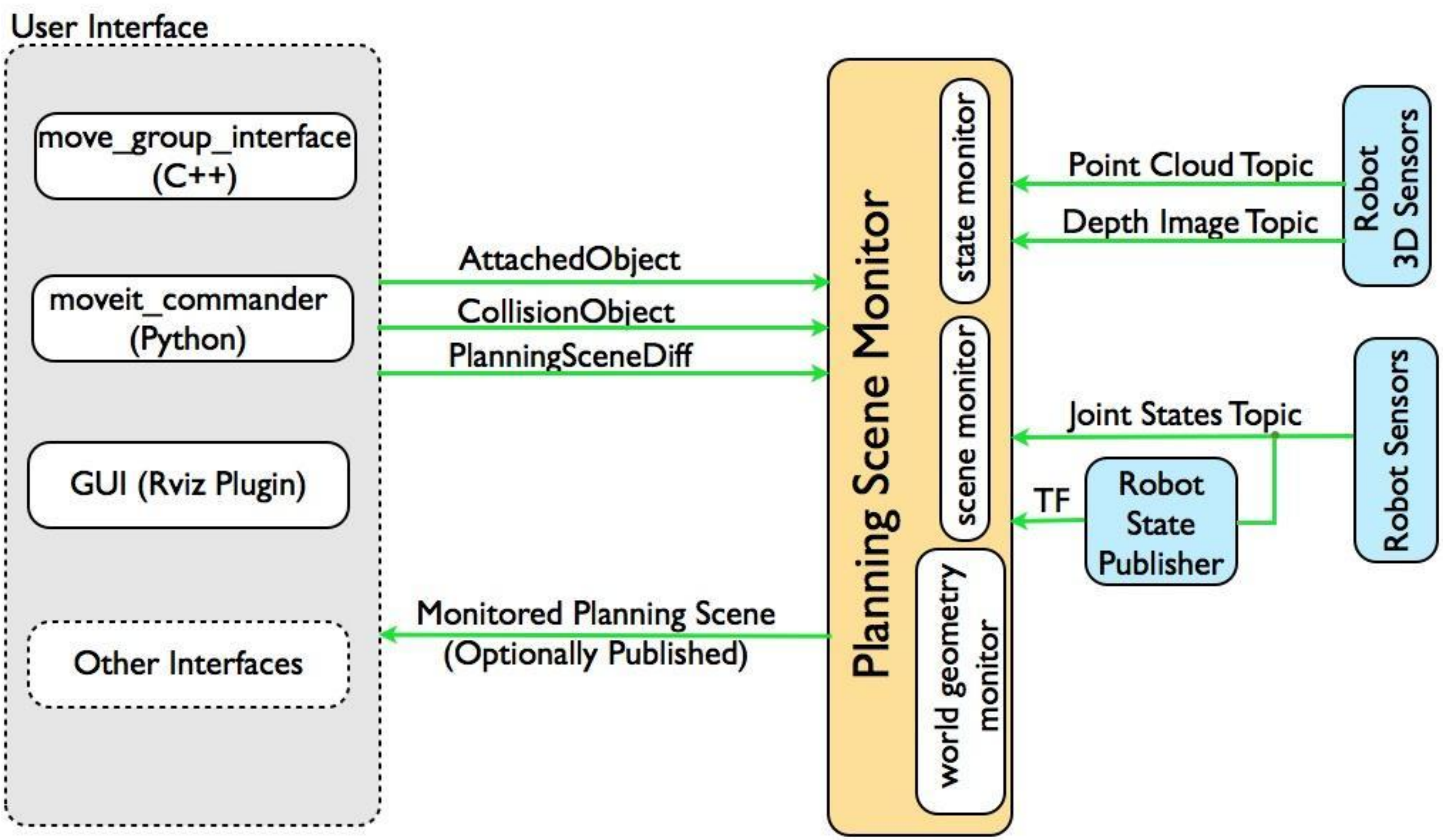
- The **world around the robot**
- The **state of the robot**

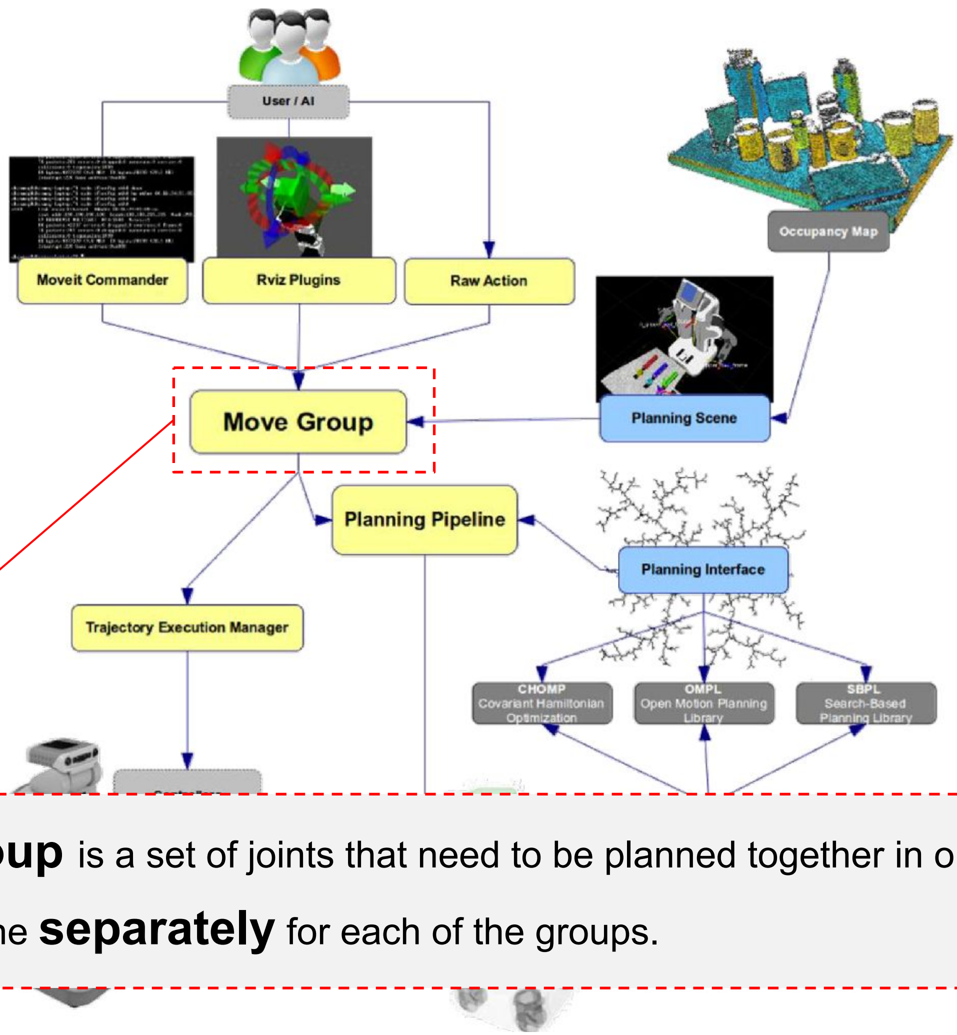
It is a subpart of `move_group`, which listens to:

- `joint_states`
- Sensor information (e.g., point clouds)
- The world geometry (provided by the user input on the `planning_scene` topic)

THE PLANNING SCENE

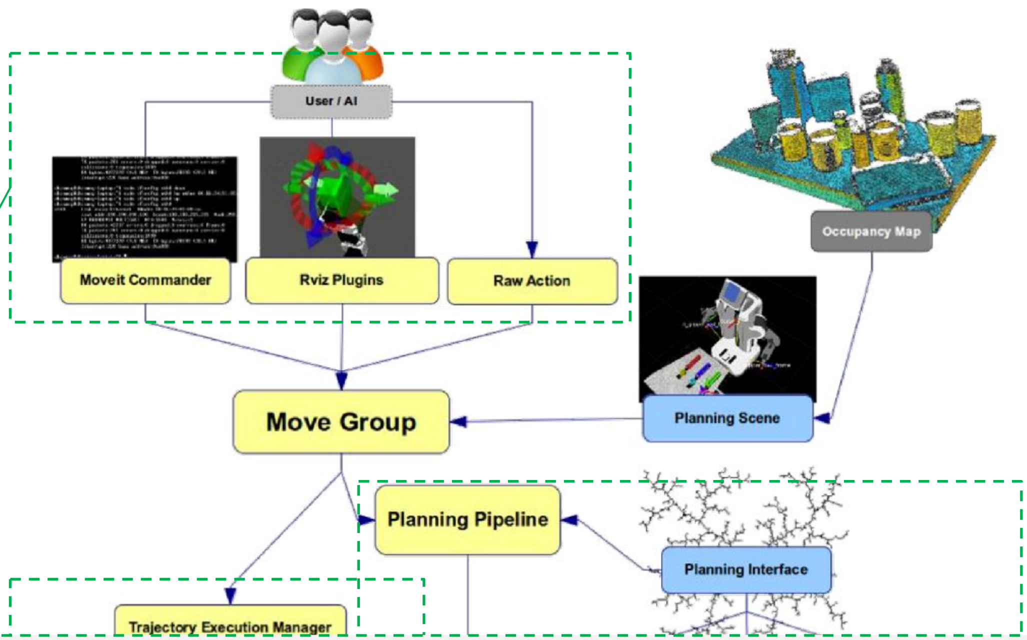
MoveIt! Pipeline





- A **planning group** is a set of joints that need to be planned together in order to achieve a goal;
- The planning is done **separately** for each of the groups.

REQUEST



1

A motion plan request is sent to the motion planner for a group.

Goal:

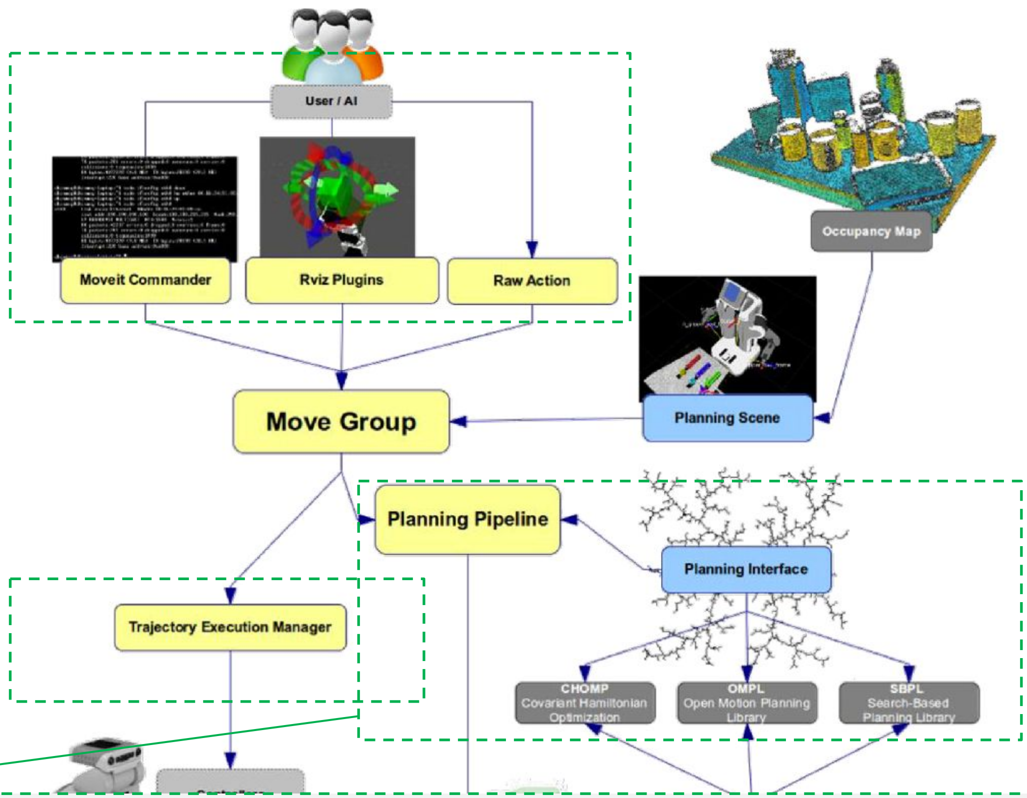
- Pose in the joint space;
- End Effector pose in the cartesian space;

Kinematics Constraints:

- Position;
- Orientation;
- Visibility;
- User-specified.

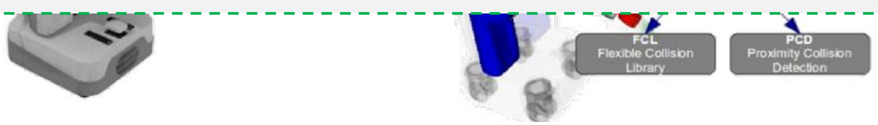
REQUEST

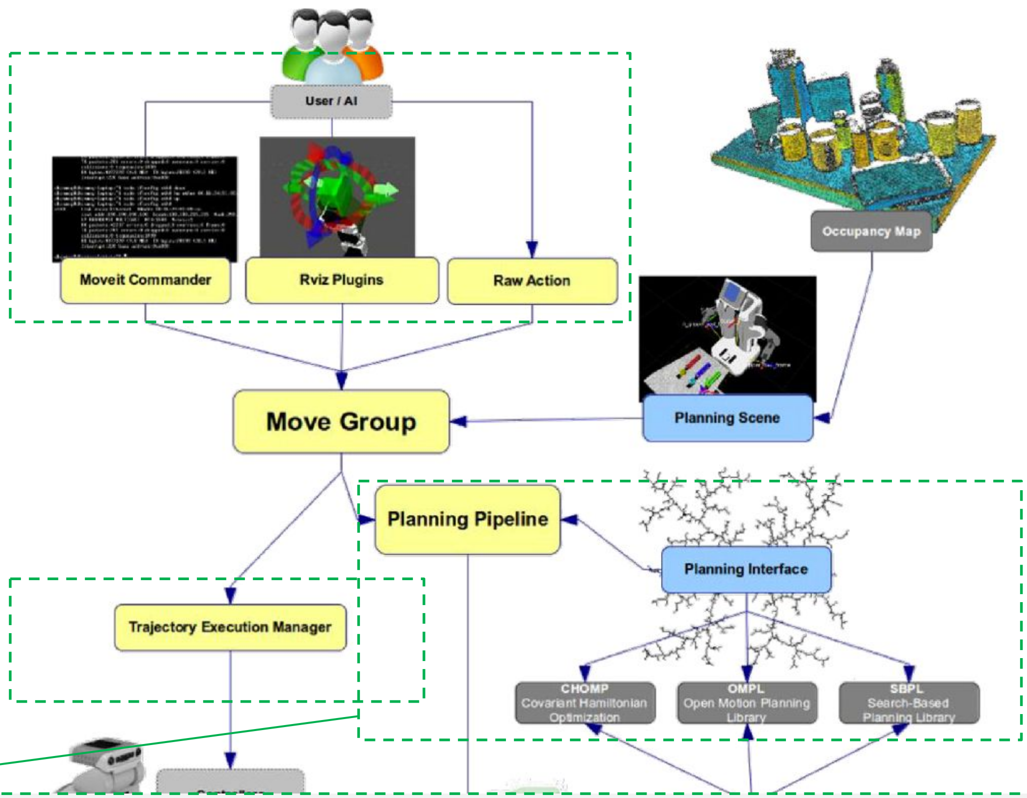
COMPUTATION

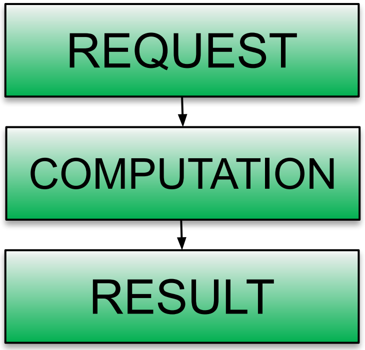


2

The Planning Pipeline searches for a trajectory for all the joints in the group that move the arm so that it reaches the goal.







3

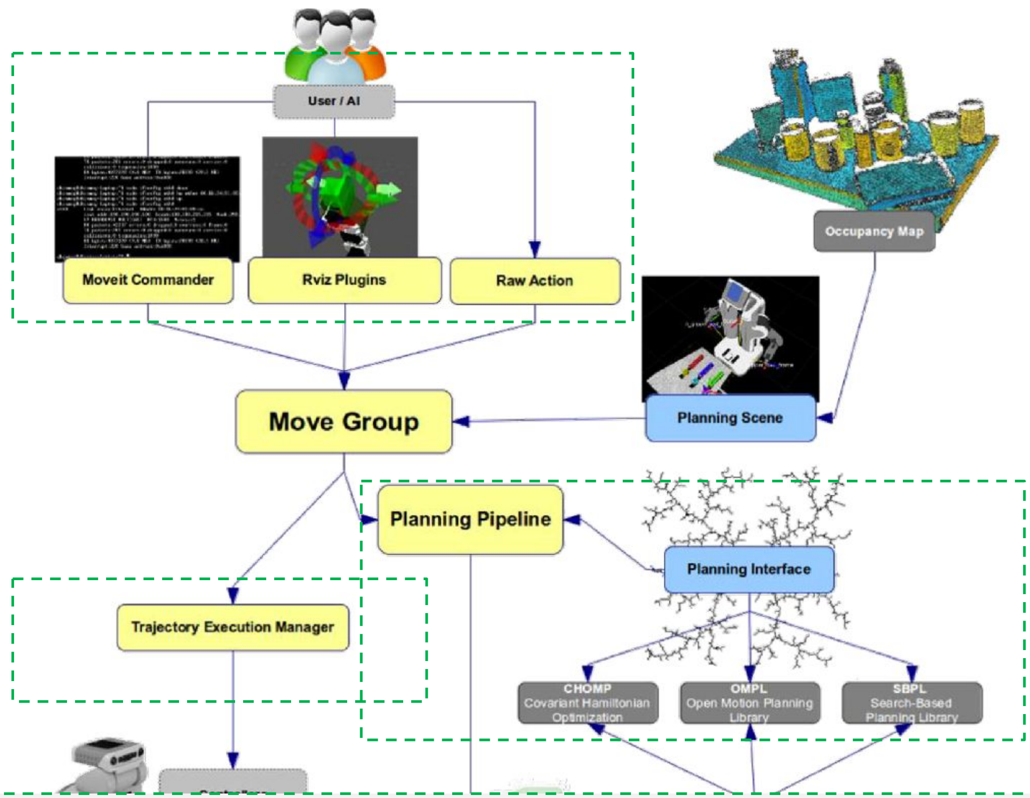
A trajectory that moves the arm to the target goal location is computed.

The trajectory:

- **Avoids collisions**;
- **Satisfies** the velocity and acceleration **constraints** at the joint level.

KINEMATICS

MoveIt! Pipeline



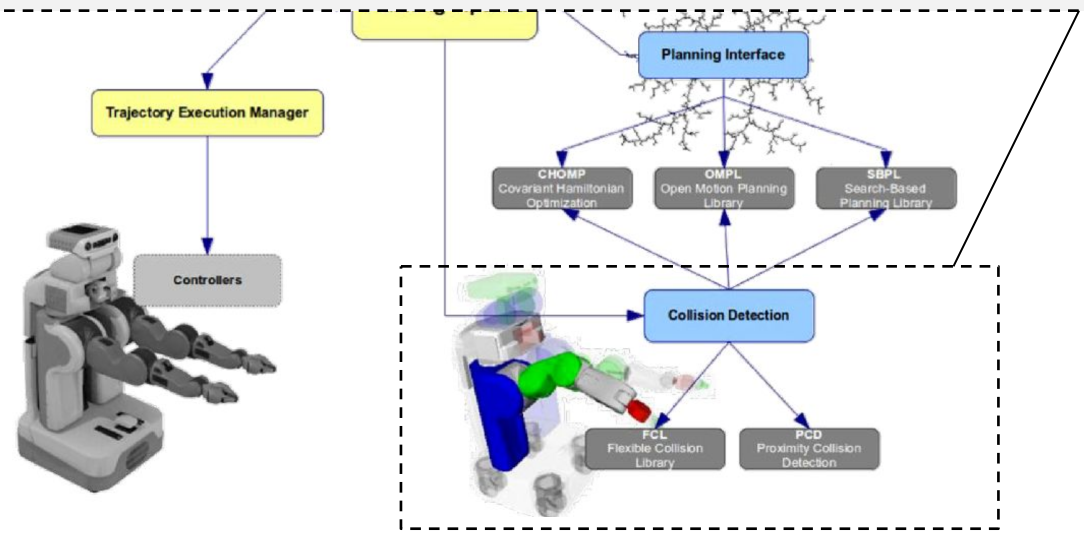
- **Forward Kinematics:** (and its Jacobians) is integrated in the `RobotState` class
- **Inverse Kinematics:** MoveIt! provides a default plugin that uses a numerical Jacobian-based solver that is automatically configured by the MoveIt! Setup Assistant.
- You can select the solver that you prefer. KDL and Trac-IK are two of them

- Collision checking is configured through the `CollisionWorld` object of the planning scene;

- Collision checking is an expensive operation, then

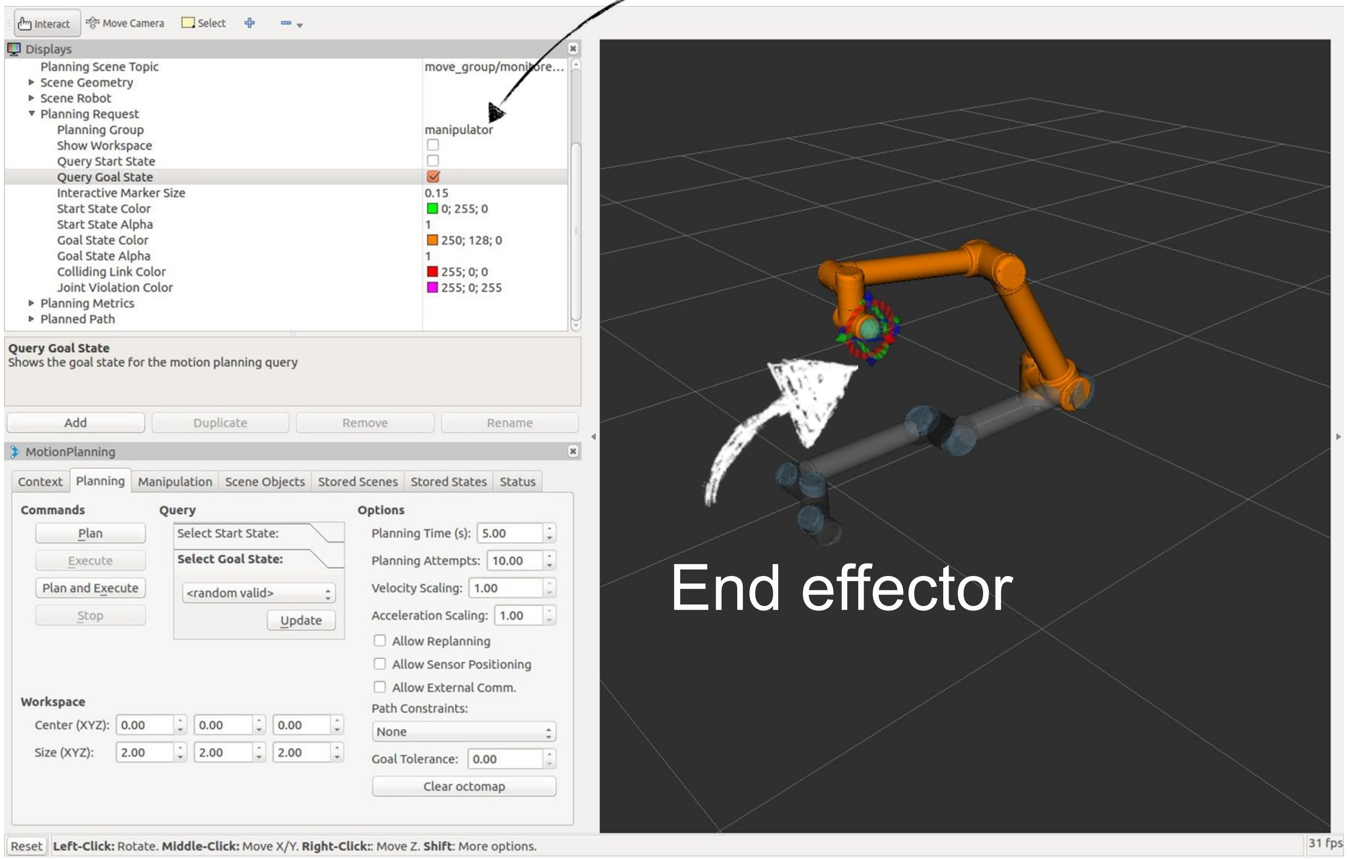
- **Allowed Collision Matrix:** Matrix used to encode a Boolean value that indicates whether collision checking is needed for two pairs of bodies:
  - value=1: collision checking is not needed (e.g., for bodies that are very far from each other, so they would never collide, or bodies that are adjacent).

(see Setup Assistant)



- About MoveIt!
- MoveIt! Pipeline
  - The Planning Scene
  - move_group
  - Motion planning
  - Kinematics
  - Collision Checking
- Simple Motion Planning tutorials
  - Motion Planning with RViz
  - Move Group Interface
  - Main Node

Planning group



- Setup

- Different Planning Options:
  - Pose Goal (Cartesian)
  - Joint-space Goal
  - Path Constraints
  - Cartesian Paths

- Collision Objects

- Attached/Detached Objects

Setup

- MoveIt operates on sets of joints called "planning groups" and stores them in an object called the **JointModelGroup**. Throughout MoveIt the terms "**planning group**" and "**joint model group**" are used interchangeably.

- The MoveGroup class is setup using the name of the planning group you would like to control
    - *moveit::planning_interface::MoveGroupInterface move_group(PLANNING_GROUP);*

- We will use the PlanningSceneInterface class to add and remove collision objects in our "virtual world" scene
    - *moveit::planning_interface::PlanningSceneInterface planning_scene_interface;*

- Raw pointers are frequently used to refer to the planning group for improved performance.
    - *const moveit::core::JointModelGroup* joint_model_group =*
      move_group_interface.getCurrentState()->getJointModelGroup(PLANNING_GROUP);

- We can plan a motion for this group to a desired pose for the end-effector:
  - Build a geometry msgs that represent the goal pose
    ```
    geometry_msgs::Pose target_pose1;
    target_pose1.orientation.w = 1.0;
    target_pose1.position.x = 0.28;
    target_pose1.position.y = -0.2;
    target_pose1.position.z = 0.5;
    ```
  - Set the Pose Target for the move_group_interface
    ```
    move_group_interface.setPoseTarget(target_pose1);
    ```
  - Call the planner to compute the plan
    ```
    moveit::planning_interface::MoveGroupInterface::Plan my_plan;
    auto success = move_group_interface.plan(my_plan);
    ROS_INFO_NAMED("tutorial", "Visualizing plan 1 (pose goal) %s", success ? "" : "FAILED");
    ```

Planning and Execution

- The function plan is only for compute a plan
  - *moveit::core::MoveItErrorCode plan_success = move_group_interface.plan(my_plan);*
  - *bool success = (move_group_interface.plan(my_plan) == moveit::core::MoveItErrorCode::SUCCESS);*

- To execute the trajectory stored in my_plan, you could use the execute method
  - *move_group_interface.execute(my_plan);*

- If you do not want to inspect the planned trajectory, the following is a more robust combination of the two-step plan+execute pattern shown above and should be preferred.
  - *move_group_interface.move();*

Path Constraints

Simple Motion Planning

- Path constraints can easily be specified for a link on the robot. Let's specify a path constraint and a pose goal for our group.
    - First define the path constraint.
        - *moveit_msgs::OrientationConstraint ocm;*
        - *ocm.link_name = "panda_link7";*
        - *ocm.header.frame_id = "panda_link0";*
        - *ocm.orientation.w = 1.0;*
        - *ocm.absolute_x_axis_tolerance = 0.1;*
        - *ocm.absolute_y_axis_tolerance = 0.1;*
        - *ocm.absolute_z_axis_tolerance = 0.1;*
        - *ocm.weight = 1.0;*

- Now, set it as the path constraint for the group.
    - *moveit_msgs::Constraints test_constraints;*
    - *test_constraints.orientation_constraints.push_back(ocm);*
    - *move_group_interface.setPathConstraints(test_constraints);*

- We will reuse the old goal that we had and plan to it. Note that this will only work if the current state already satisfies the path constraints. So we need to set the start state to a new pose.
  - *moveit::core::RobotState start_state(*move_group_interface.getCurrentState());*
  - *geometry_msgs::Pose start_pose2;*
  - *start_pose2.orientation.w = 1.0;*
  - *start_pose2.position.x = 0.55;*
  - *start_pose2.position.y = -0.05;*
  - *start_pose2.position.z = 0.8;*
  - *start_state.setFromIK(joint_model_group, start_pose2);*
  - *move_group_interface.setStartState(start_state);*

- Now we will plan to the earlier pose target from the new start state that we have just created.
  - *move_group_interface.setPoseTarget(target_pose1);*

- Planning with constraints can be slow because every sample must call an inverse kinematics solver. Lets increase the planning time from the default 5 seconds to be sure the planner has enough time to succeed.
    - *move_group_interface.setPlanningTime(10.0);*
    - *success = move_group_interface.plan(my_plan);*

- When done with the path constraint be sure to clear it.
    - *move_group_interface.clearPathConstraints();*

- Create an pointer that references the current robot's state. RobotState is the object that contains all the current position/velocity/acceleration data.
  - *moveit::core::RobotStatePtr current_state = move_group_interface.getCurrentState();*
- Get the current set of joint values for the group.
  - *std::vector<double> joint_group_positions;*
  - *current_state->copyJointGroupPositions(joint_model_group, joint_group_positions);*
- Modify one of the joints, plan to the new joint space goal and visualize the plan.
  - *joint_group_positions[0] = -1.0; // radians*
  - *move_group_interface.setJointValueTarget(joint_group_positions);*
- Plan and execute
  - *bool success = move_group_interface.plan(my_plan);*
  - *move_group_interface.execute(my_plan);*

- Plan a Cartesian path directly by specifying a list of waypoints for the end-effector to go through. Note that we are starting from the new start state above. The initial pose (start state) does not need to be added to the waypoint list but adding it can help (with visualizations)
  - std::vector<geometry_msgs::Pose> waypoints;
  - waypoints.push_back(start_pose2);

  - geometry_msgs::Pose target_pose3 = start_pose2;
  - target_pose3.position.z -= 0.2;
  - waypoints.push_back(target_pose3); // down

  - target_pose3.position.y -= 0.2;
  - waypoints.push_back(target_pose3); // right

  - target_pose3.position.z += 0.2;
  - target_pose3.position.y += 0.2;
  - target_pose3.position.x -= 0.2;
  - waypoints.push_back(target_pose3); // up and left

- We want the Cartesian path to be interpolated at a resolution of 1 cm which is why we will specify 0.01 as the max step in Cartesian translation. We will specify the jump threshold as 0.0, effectively disabling it.
  - *const double eef_step = 0.01;*
  - *const double jump_threshold = 0.0;*

- **Warning:** disabling the jump threshold while operating **real hardware** can cause large unpredictable motions of redundant joints and could be a safety issue
  - *moveit_msgs::RobotTrajectory trajectory;*
  - *double fraction = move_group_interface.computeCartesianPath(waypoints, eef_step, trajectory);*

- Fraction is a value between 0.0 and 1.0 indicating the fraction of the path achieved as described by the waypoints. Return -1.0 in case of error.

Adding collision objects

- Define a collision object ROS message for the robot to avoid.
  - *moveit_msgs::CollisionObject collision_object;*
  - *collision_object.header.frame_id = move_group_interface.getPlanningFrame();*

- The id of the object is used to identify it.
  - *collision_object.id = "box1";*

- Define a box to add to the world.
  - *shape_msgs::SolidPrimitive primitive;*
  - *primitive.type = primitive.BOX;*
  - *primitive.dimensions.resize(3);*
  - *primitive.dimensions[primitive.BOX_X] = 0.1;*
  - *primitive.dimensions[primitive.BOX_Y] = 1.5;*
  - *primitive.dimensions[primitive.BOX_Z] = 0.5;*

Adding collision objects

- Define a pose for the box (specified relative to frame_id)
  - *geometry_msgs::Pose box_pose;*
  - *box_pose.orientation.w = 1.0;*
  - *box_pose.position.x = 0.5;*
  - *box_pose.position.y = 0.0;*
  - *box_pose.position.z = 0.25;*

  - *collision_object.primitives.push_back(primitive);*
  - *collision_object.primitive_poses.push_back(box_pose);*
  - *collision_object.operation = collision_object.ADD;*

  - *std::vector<moveit_msgs::CollisionObject> collision_objects;*
  - *collision_objects.push_back(collision_object);*

- Now, let's add the collision object into the world (using a vector that could contain additional objects)
  - *planning_scene_interface.addCollisionObjects(collision_objects);*

- Adding collision objects into the world (using a vector that could contain additional objects)
    - *planning_scene_interface.addCollisionObjects(collision_objects);*

- Remove the collision object from the world
    - *planning_scene_interface.removeCollisionObjects(collision_objects);*

- Attach the collision object to the robot
    - *move_group_interface.attachObject(object_to_attach.id);*

- Detach the collision object to the robot
    - *move_group_interface.detachObject(object_to_attach.id);*

- ROS spinning must be running for the MoveGroupInterface to get information about the robot's state. One way to do this is to start an AsyncSpinner beforehand.
    - *ros::init(argc, argv, "move_group_interface_tutorial");*
    - *ros::NodeHandle node_handle;*
    - *ros::AsyncSpinner spinner(uint32_t thread_count);*
    - *spinner.start();*

- **thread_count**: The number of threads to use. A value of 0 means to use the number of processor cores

ASSIGNMENT 2