ROS Industrial

LIU Yao

ECNU Intelligent Robot Lab

Introduction

- ROS-Industrial

  The ROS-Industrial packages comes with a solution to interface industrial robot manipulators to ROS and controlling it using the power of ROS.

Introduction

- Goal

  - Combine strengths of ROS to the existing industrial technologies for exploring advanced capabilities of ROS in the manufacturing process.

  - Developing a reliable and robust software for industrial robots application.

  - Provide an easy way for doing research and development in industrial robotics.

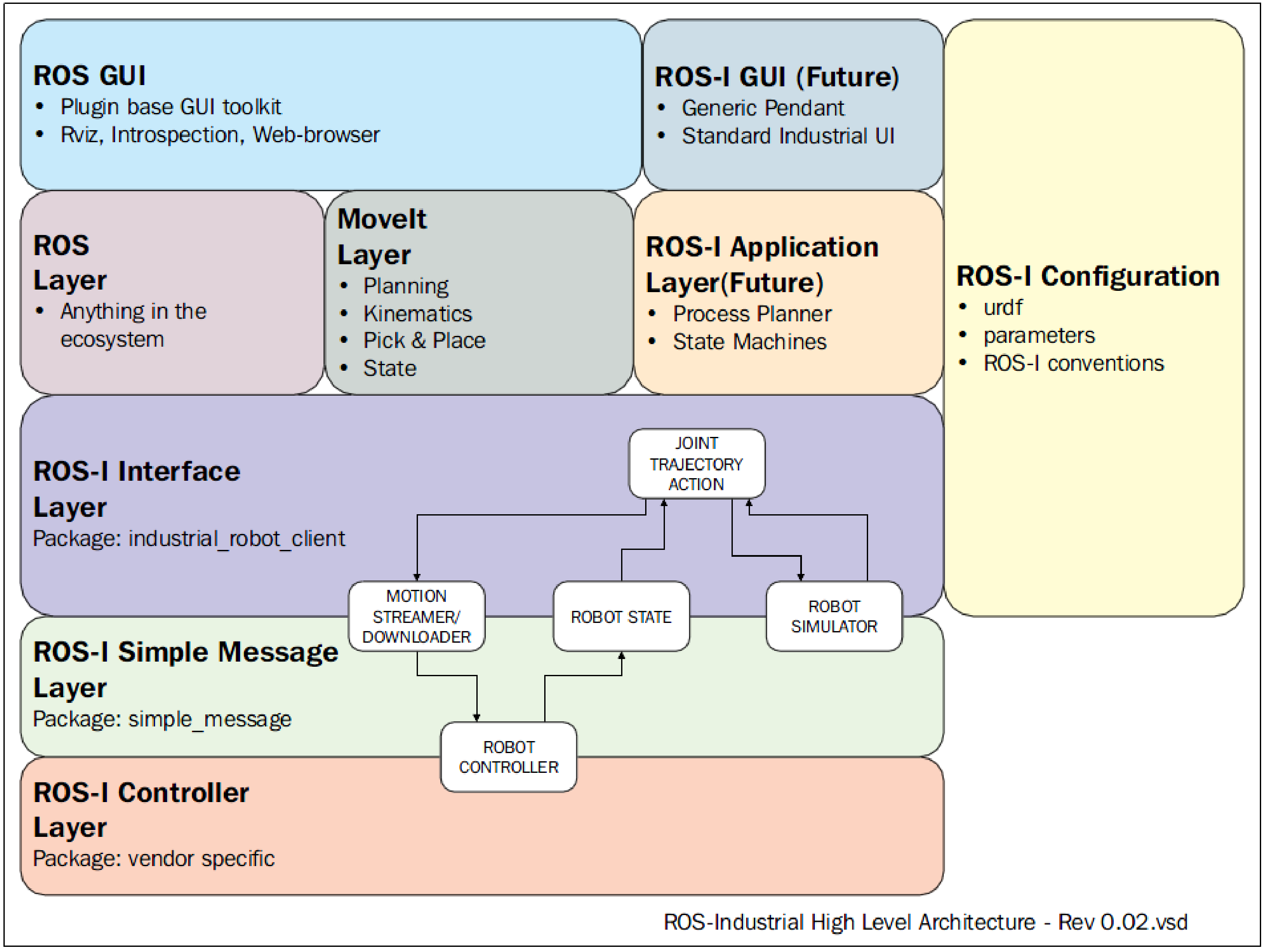
Introduction

- Install

  $ sudo apt-get install ros-indigo-industrial-*

- Packages
  - industrial_robot_simulator
  - industrial_robot_client
  - …

Introduction

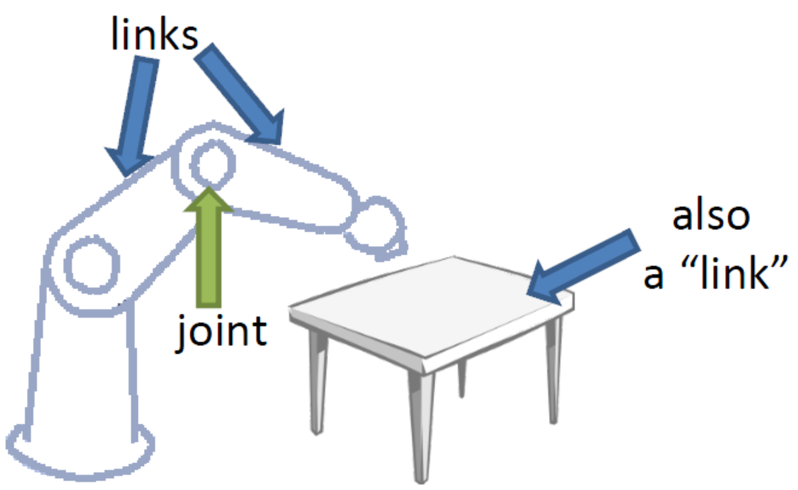


Model

- ROS packages for robot modeling
  - robot_model:
    - urdf
    - joint_state_publisher
    - kdl_parser
  - robot_state_publisher
  - xacro

URDF: Overview

- URDF is an **XML**-formatted file containing:
  - **Links** : coordinate frames and associated geometry
  - **Joints :** connections between links

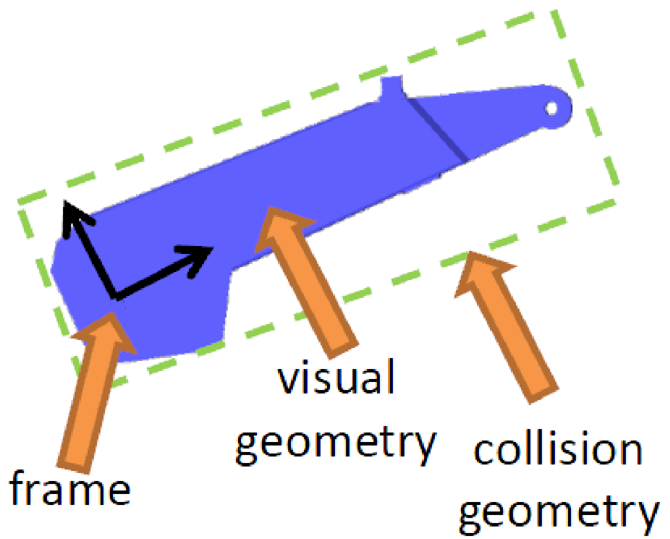


URDF: Link

- A **Link** describes a physical or virtual object
  - Physical : robot link, end-effector, ...
  - Virtual : TCP, robot base frame, ...
- Each link becomes a TF frame
- Can contain visual/collision geometry

URDF: Link

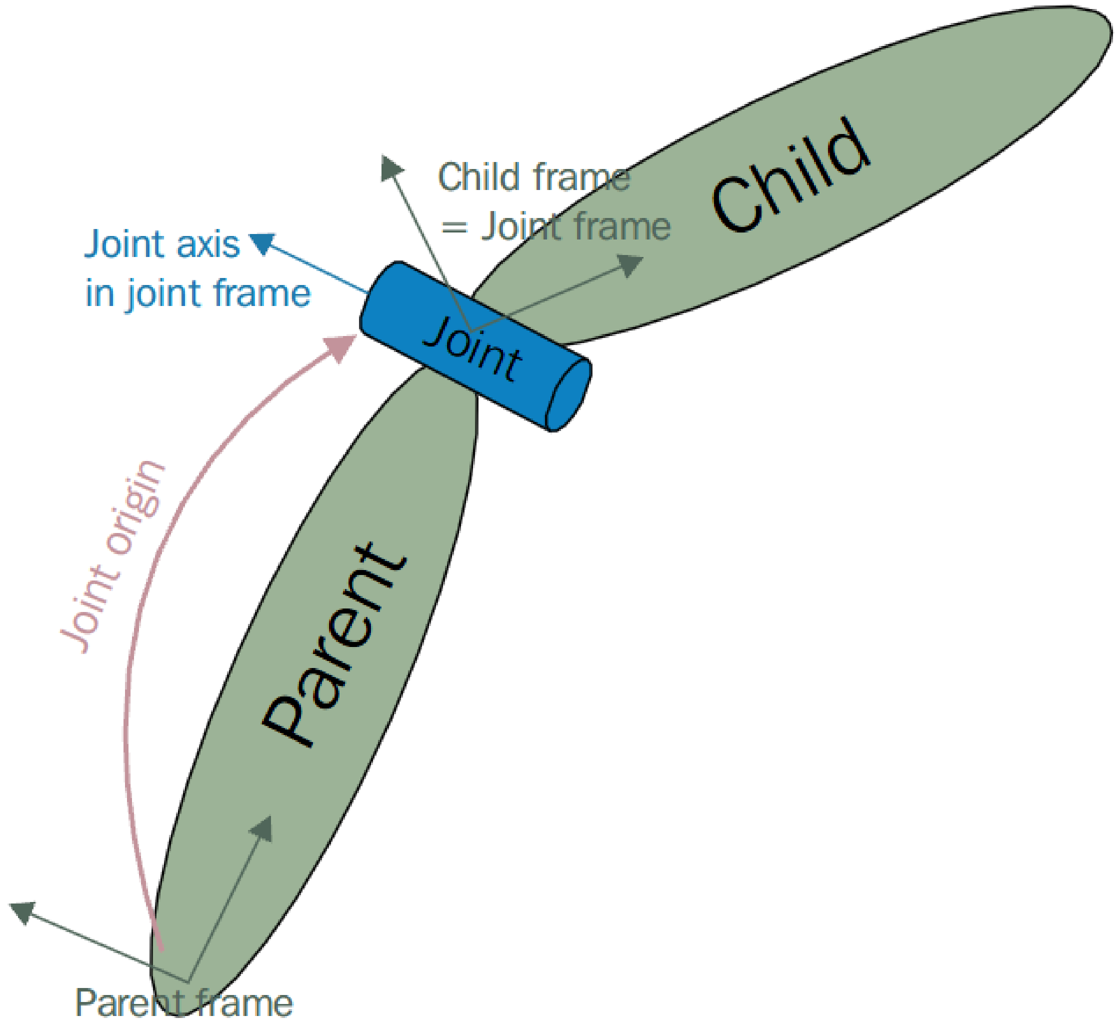
```
<link name="link_4">
    <visual>
        <geometry>
            <mesh filename="link_4.stl"/>
        </geometry>
        <origin xyz="0 0 0" rpy="0 0 0" />
    </visual>
    <collision>
        <geometry>
            <cylinder length="0.5" radius="0.1"/>
        </geometry>
        <origin xyz="0 0 -0.05" rpy="0 0 0" />
    </collision>
</link>
```



URDF: Joint

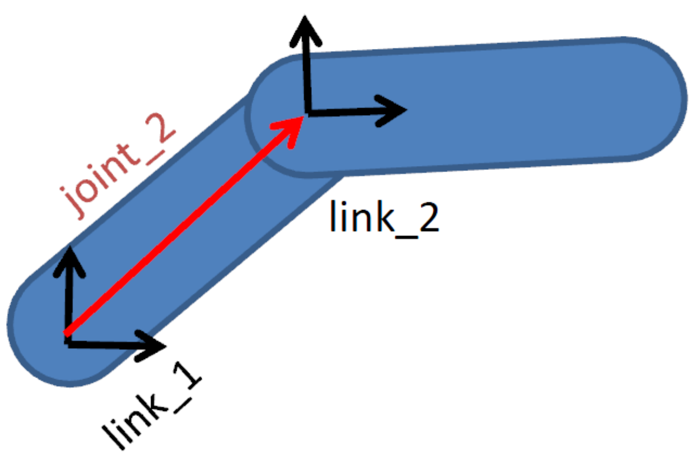
- A Joint connects 2 Links
  - Defines a transform between parent and child frames
    - Types: fixed, revolute, free, floating, planar
  - Denotes axis of movement (for linear / rotary)
  - Contains joint limits on position and velocity
- ROS-I conventions
  - X-axis front, Z-Axis up
  - Keep all frames similarly rotated when possible

URDF: Joint


Child frame
= Joint frame
Child
Joint axis
in joint frame
Joint
Joint origin
Parent
Parent frame

URDF: Joint

```
<joint name="joint_2" type="revolute">
   <parent link="link_1"/>
   <child link="link_2"/>
   <origin xyz="0.2 0.2 0" rpy="0 0 0"/>
   <axis xyz="0 0 1"/>
   <limit lower="-3.14" upper="3.14" velocity="1.0"/>
</joint>
```



URDF: Check

- Check urdf
  - $ check_urdf planar_3dof.urdf
  - $ urdf_to_graphiz planar_3dof.urdf
  - $ evince planar_3dof.pdf

- Show in rviz
  - roslaunch urdf_tutorial display.launch model:=`rospack find lesson_urdf`/urdf/planar_3dof.urdf gui:=true

URDF: XACRO

- XACRO is an XML-based “macro language” for building URDFs
  - <Include> other XACROs, with parameters
  - Simple expressions: math, substitution
- Used to build complex URDFs
  - multi-robot workcells
  - reuse standard URDFs (e.g. robots, tooling)
- Convert

  $ rosrun xacro xacro.py -o <urdf_file> <xacro_file>

URDF: gazebo

- The collision and inertia parameters are required in each link
- Transmission tag
  - Relate a joint to a controller
- gazebo_ros_control plugin

```
<!-- ros_control plugin -->
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.
so">
    <robotNamespace>/seven_dof_arm</robotNamespace>
  </plugin>
</gazebo>
```

URDF: Example

- ABB + gripper + workpiece
  - $ roslaunch lesson_xacro lesson_xacro.launch
- Keyboard Control
  - modify launch file

Actions

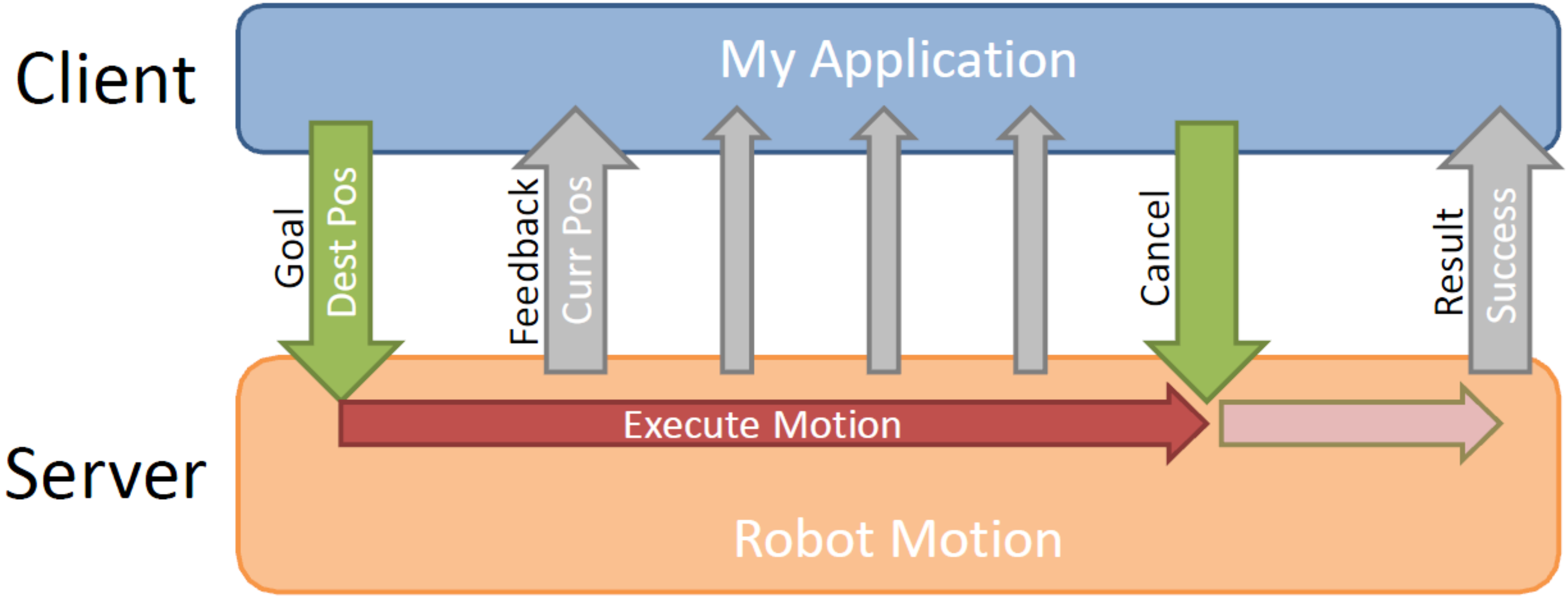
Type	Strengths	Weaknesses
Message	•Good for most sensors (streaming data) •One - to - Many	•Messages can be dropped without knowledge •Easy to overload system with too many messages
Service	•Knowledge of missed call •Well-defined feedback	•Blocks until completion •Connection typically re-established for each service call (slows activity)
Action	•Monitor long-running processes •Handshaking (knowledge of missed connection)	•Complicated

Actions

- **Message**:
  Robot teleoperation, publishing odometry, sending robot transform(TF), and sending robot joint states
- **Service**:
  This saves camera calibration parameters to a file, saves a map of the robot after SLAM, and loads a parameter file
- **Action**:
  This is used in motion planners and ROS navigation stacks

Actions : Overview

Actions manage **Long-Running Tasks**



Actions: Detail

- Each action is made up of 3 components:
  - Goal, sent by client, received by server
  - Result, generated by server, sent to client
  - Feedback, generated by server

- Examples
  - Goal:

    If a robot arm joint wants to move from 45 degrees to 90 degrees, the goal here is 90 degrees.
  - Result:

    The result can be anything indicating it finished the goal.
  - Feedback:

    The intermediate value between 45 and 90 degrees in which the arm is moving.

Actions: Detail

- Non-blocking in client
  - Can monitor feedback or cancel before completion
- Typical Uses:
  - "Long" Tasks: Robot Motion, Path Planning
  - Complex Sequences: Pick Up Box, Sort Widgets

Actions: Syntax

- Action definition
  - Defines Goal, Feedback and Result data types
    - Any data type(s) may be empty. Always receive handshakes.
  - Auto-generates C++ Class files (.h/.cpp), Python, etc.

CalcPi.action

```
# Calculate Pi
int32 digits        <- Goal Data
---
string pi           <- Result Data
---
string pi           <- Feedback Data
int32 iter
```

Actions: Syntax

- Action **Server**
  - Defines **Execute Callback**
  - Periodically **Publish Feedback**
  - Advertises available action *(Name, Data Type)*

Callback Function

Goal Data (IN)

Check for Cancel

Feedback

Result

```
void executeCB(const CalcPiGoalConstPtr &goal) {
  loop {
    if (as_.isPreemptRequested() || !ros::ok())
      as_.setPreempted();

    as_.publishFeedback(…);
  }
  as_.setSucceeded(result_);
}
SimpleActionServer<CalcPiAction> as_ ("calcPi", &executeCB);
```

Action Name

Callback Ref

Server Object

Actions: Syntax

- Action **Client**
  - Connects to specific Action *(Name / Data Type)*
  - Fills in Goal data
  - Initiate Action / Waits for Result

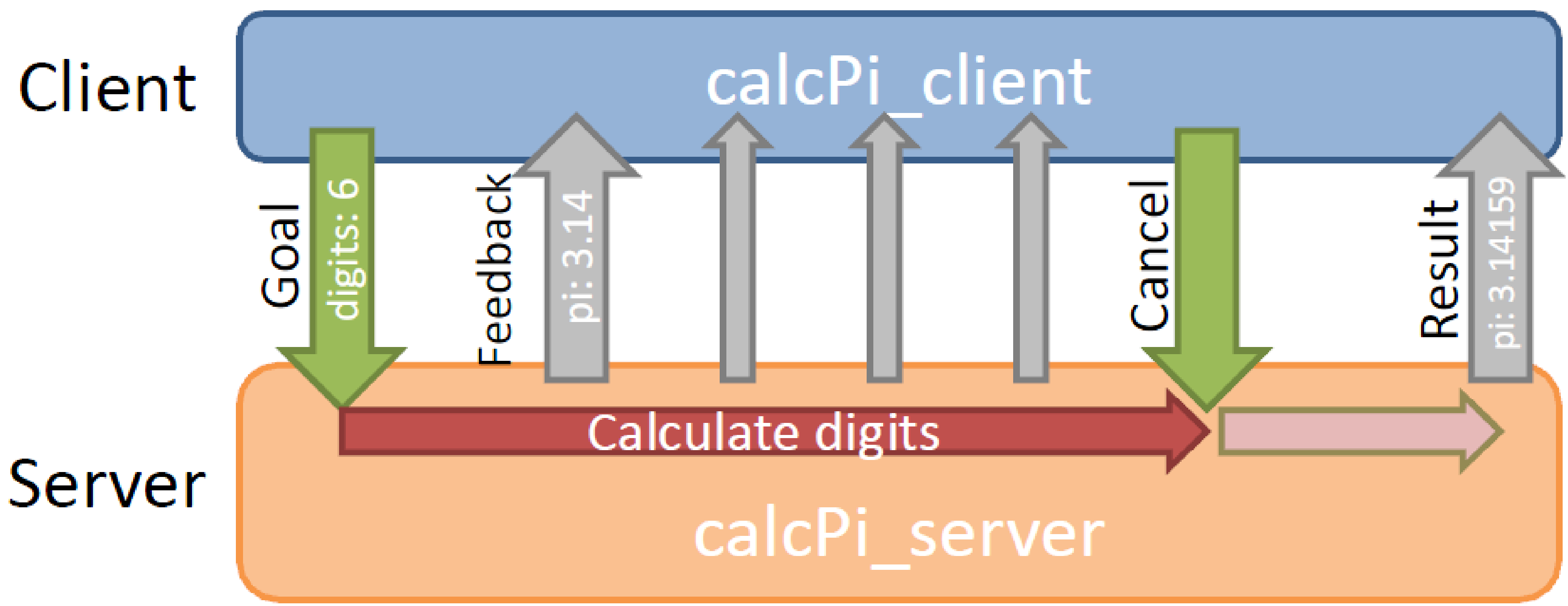
Action Type — Client Object — Action Name

```
SimpleActionClient<CalcPiAction> ac("calcPi");

CalcPiGoal goal;
goal.digits = 7;

ac.sendGoal(goal);

ac.waitForResult();
```

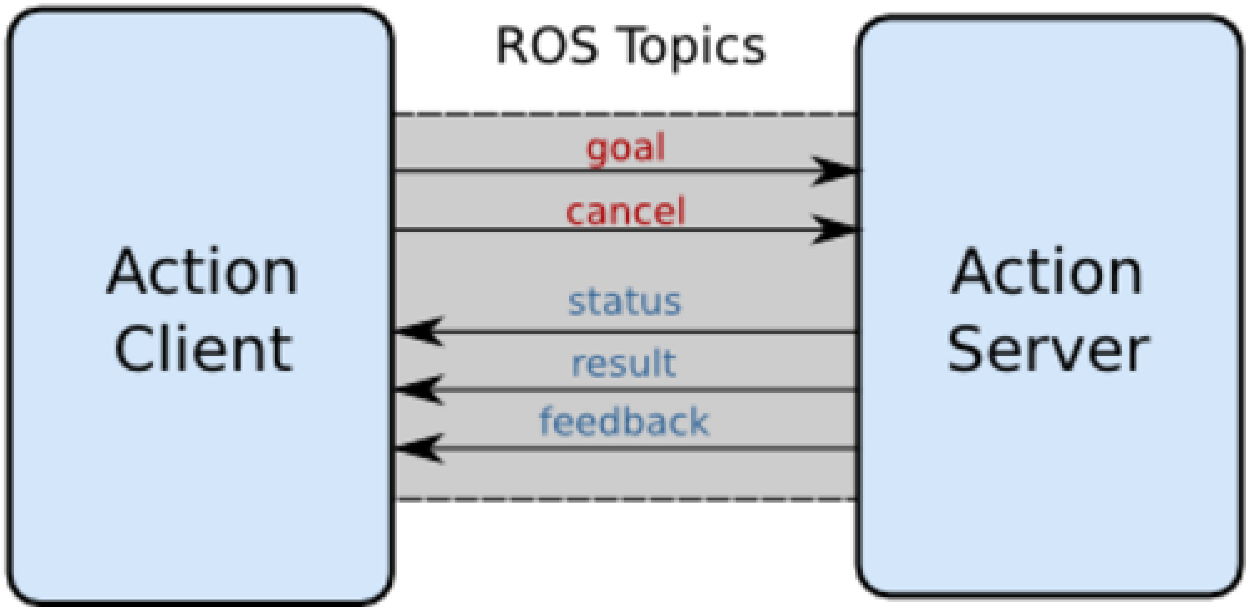
- `CalcPiGoal goal; goal.digits = 7;` ← Goal Data
- `ac.sendGoal(goal);` ← Initiate Action
- `ac.waitForResult();` ← Block Waiting

Actions: Examples

- Example



Actions: Examples



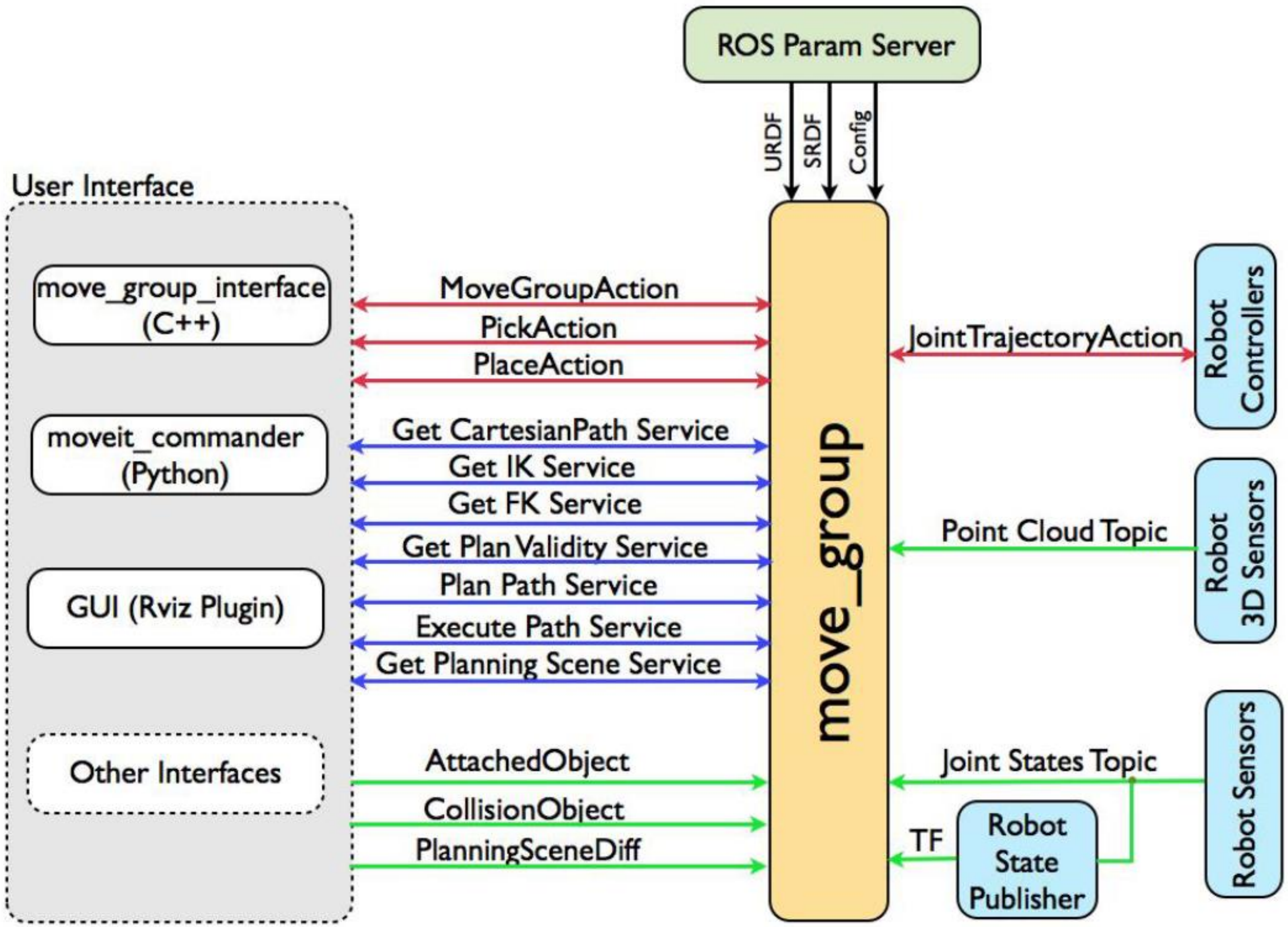
MoveIt!: Overview

- MoveIt! is a set of packages and tools for doing mobile manipulation in ROS.
- MoveIt! contains state of the art software for motion planning, manipulation, 3D perception, kinematics, collision checking, control, and navigation.

MoveIt!: Overview

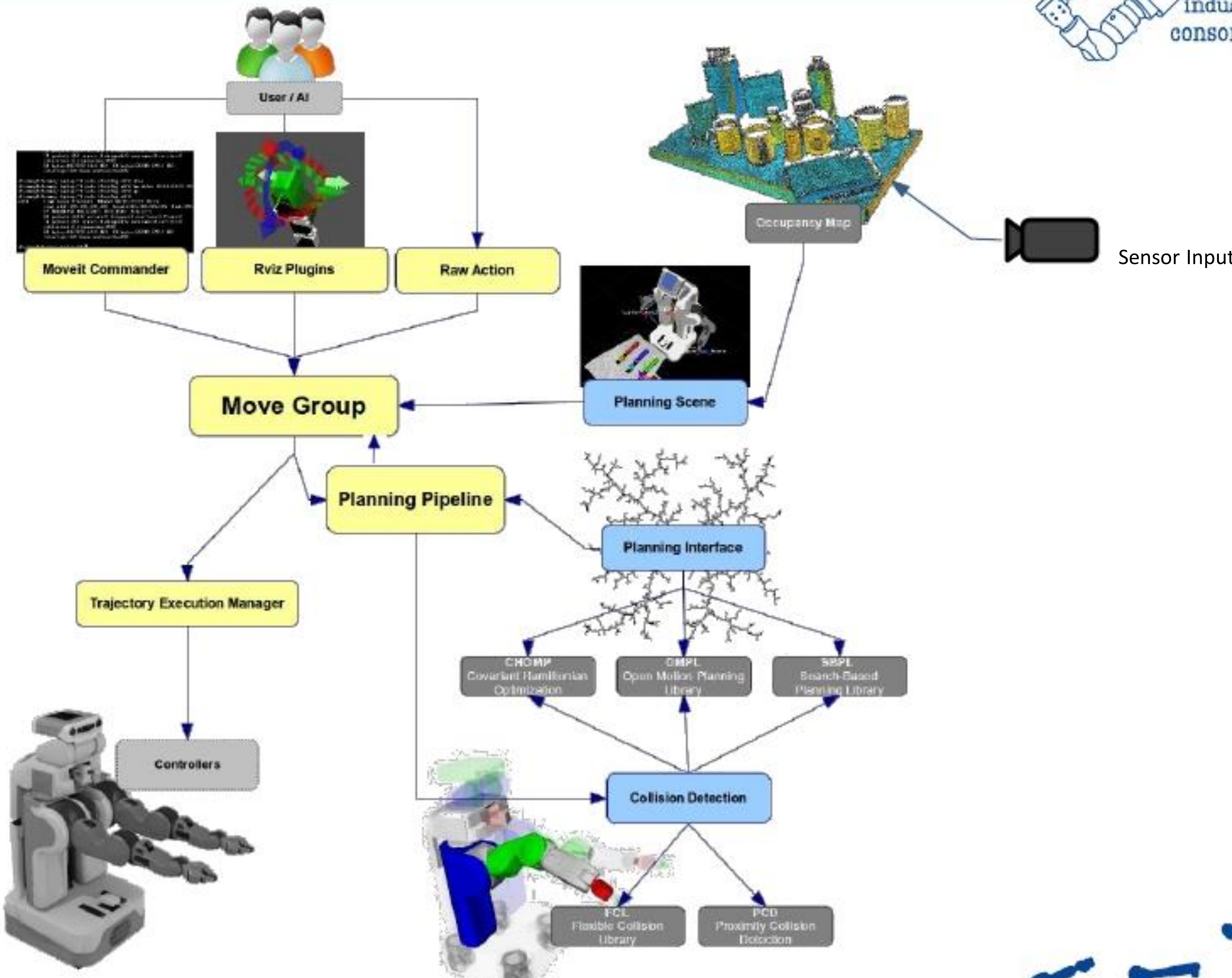
- Motion Planning for industrial robot

MoveIt!: Overview



http://moveit.ros.org/documentation/concepts/

MoveIt!: Overview



MoveIt! / Robot Integration

- A MoveIt! Package...
  - includes all required nodes, config, launch files
    - motion planning, filtering, collision detection, etc.
  - is unique to each individual robot model
    - includes references to URDF robot data
  - uses a standard interface to robots
    - publish trajectory, listen to joint angles
  - can (optionally) include workcell geometry
    - e.g. for collision checking

MoveIt Example

- A lot goes into making the UR5 move:
  - Joint states
  - Robot drivers
  - Path planners
  - Execution monitoring
- This is why MoveIt is valuable

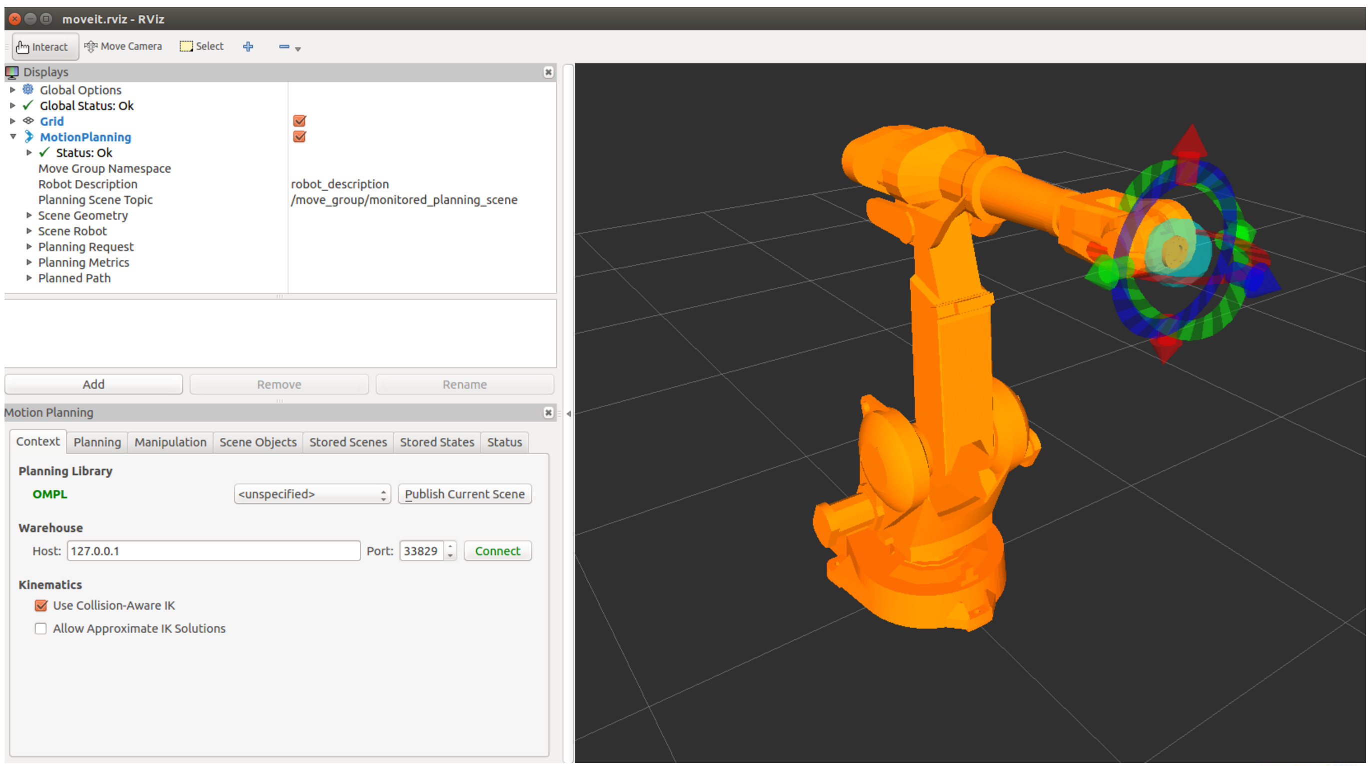
MoveIt!: Overview

- Install
  - $ sudo apt-get install ros-indigo-moveit-full

- $ sudo apt-get install ros-indigo-universal-robot
  - ur_description
  - ur_driver
  - ur_bringup
  - ur_gazebo
  - ur_msgs
  - ur10_moveit_config/ur5_moveit_config
  - ur_kinematics
- $ sudo apt-get install ros-<distro>-abb

MoveIt!: Overview

- Planning Environment
  - $ roslaunch abb_irb2400_moveit_config demo.launch



For each new robot model...

create a new MoveIt! package

- Kinematics
  - physical configuration, lengths, etc.
- MoveIt! configuration
  - plugins, default parameter values
  - self-collision testing
  - pre-defined poses
- Robot connection
  - FollowJointTrajectory Action name

HowTo:
Set Up a New Robot

1. Create a URDF
2. Create a MoveIt! Package
3. Update MoveIt! Package for ROS-I
4. Test on ROS-I Simulator
5. Test on “Real” Robot

Create a URDF

- Previously covered URDF basics

Verify the URDF

- It is **critical** to verify that your URDF matches the physical robot:
  - each joint moves as expected
  - joint-coupling issues are identified
  - min/max joint limits
  - joint directions (pos/neg)
  - correct zero-position, etc.

Create a MoveIt! Package

- Use the MoveIt! Setup Assistant
  - can create a new package or edit an existing one

Create a MoveIt! Package

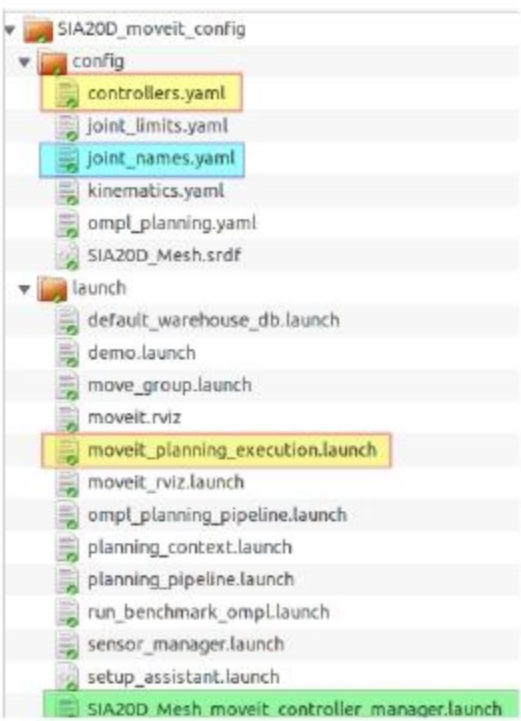
- Launch the MoveIt Setup Assistant:
  - $roslaunch moveit_setup_assistant setup_assistant.launch
  - $rospack find abb_irb2400_support
- Calculate Self-Collisions
- Add a Virtual Joint
  - FixedBase: base_link -> world, type: fixed

Create a MoveIt! Package

- Add arm joints to Planning Group
  - Group Name: manipulator
  - Kinematic Solver: KDL
  - Add Kin Chain: base_link and tool0
- Add robot poses
- Generate configuration files

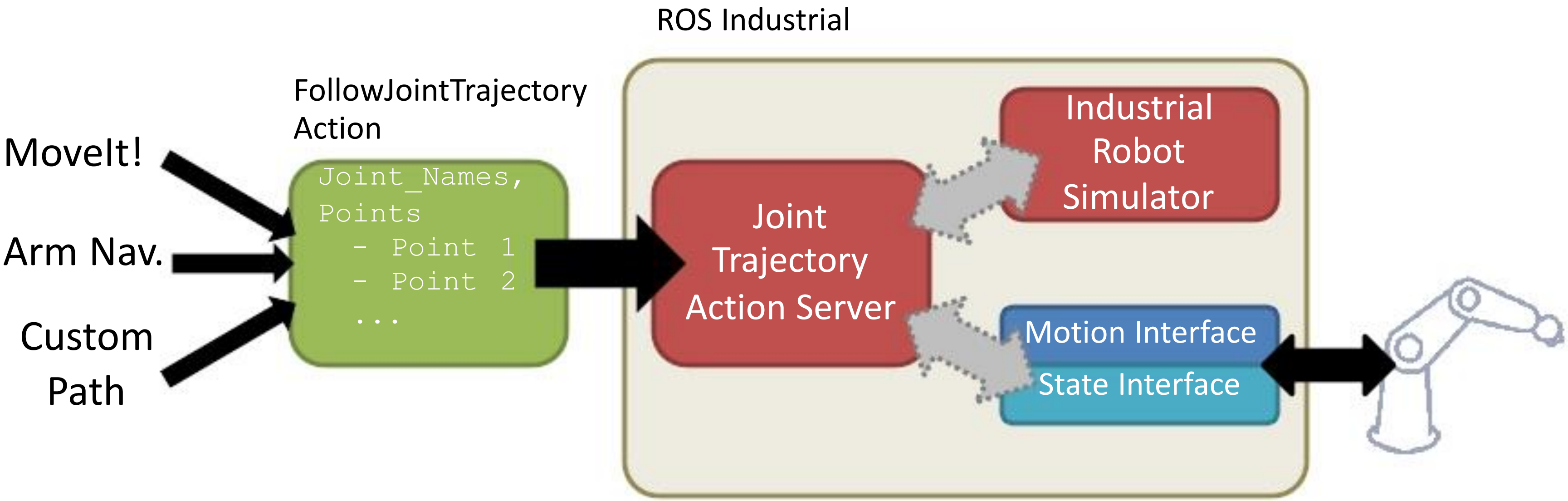
Update MoveIt! Package

- Setup Assistant generates a *generic* package
  - missing config. data to connect to a specific robot
  - ROS-I robots use a *standard* interface



Update MoveIt! Package

- We'll generate launch files to run both:
  - **simulated** ROS-I robot
  - **real** robot-controller interface



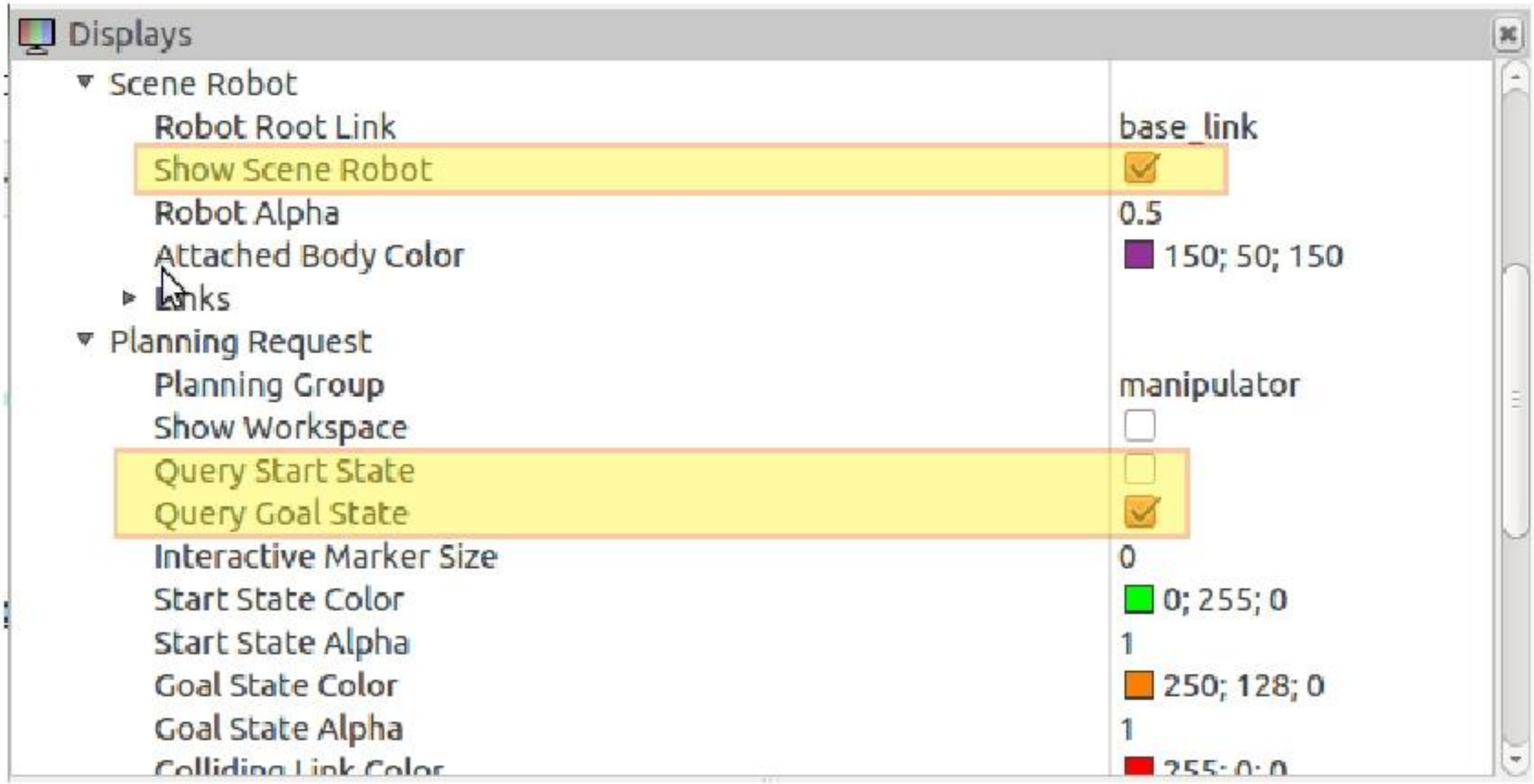
Update MoveIt! Package

- Check
  - $ roslaunch irb2400_moveit_cfg moveit_planning_execution.launch

HowTo:
Motion Planning using MoveIt!

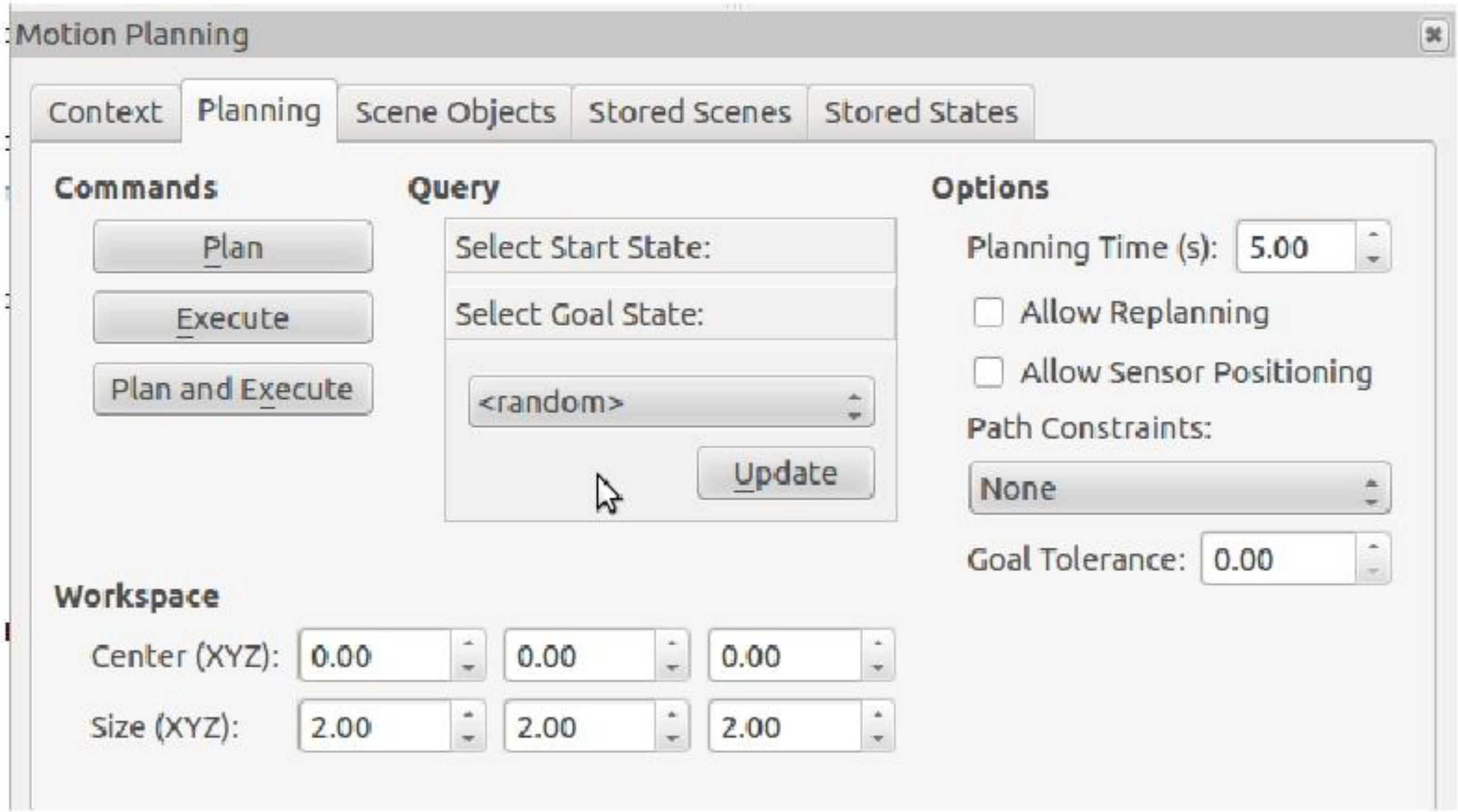
1. Motion Planning using RViz
2. Motion Planning using C++

Display Options



Motion Planning in RViz

Planning Options



Motion Planning using C++

- $ roslaunch irb2400_moveit_cfg moveit_planning_execution.launch
- $ rosrun lesson_move_group lesson_move_group_1

demo

Moveit！ controller ros_control

Pick & Place

demo

- Rviz add , display model

demo

- Ur5 + gripper