Manipulation

ROS Training for Industry: Day 5

Karl Kruusamäe

20.09.2019

Tartu, Estonia

A quick recap...

ROS terminology: package, node, message, topic

ROS skills: creating ROS nodes, publishers, subscribers, running and launching

ROS tools: RViz, rqt, tf, geometry_msgs/PoseStamped

Agenda: Day 5 (20.09)

- 09:15 Robot Description (URDF), MoveIt!
- 10:00 Coffee Break
- 10:10 Workshop
  - MoveIt GUI
  - URDF
  - MoveIt Setup Assistant
- 12:00 Lunch Break
- 13:00 Workshop: MoveGroup C++ Interface
- 14:30 Coffee Break
- 14:45 Workshop: Motion planning with multiple robots
- 16:15 Conclusions, feedback, ROS2
- 17:00 End of Day 5

Motion planning for manipulators

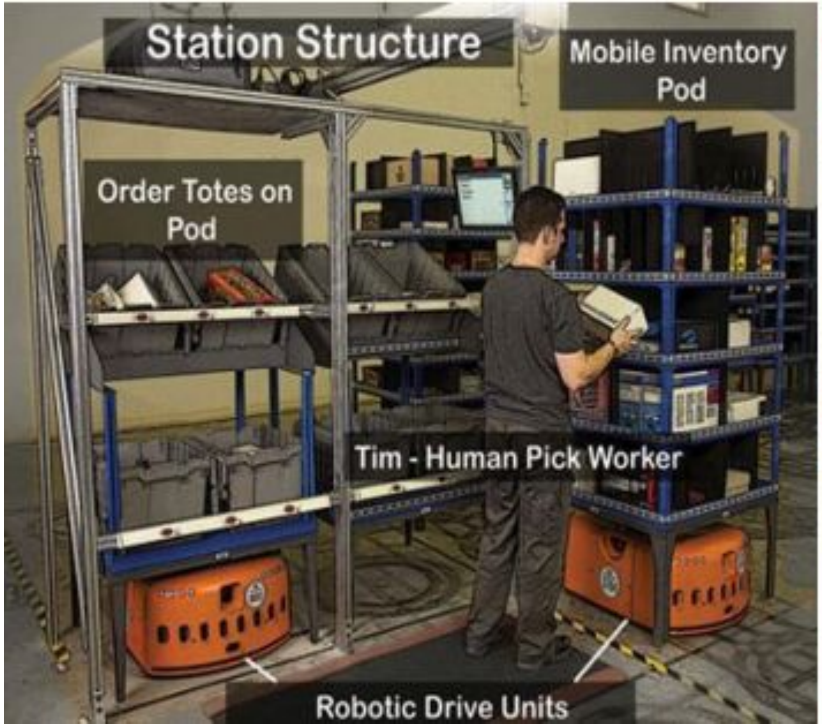
Agenda in terms of content

1. What is motion planning?

2. What is needed for motion planning?

3. **How ROS helps us with motion planning?**
   a. **URDF** (Unified Robot Description Format)
   b. **MoveIt** Motion Planning Framework
   c. MoveIt Move Group **C++ Interface**

The task



doi.org/10.1145/3132446.3134904

Motion planning

Objective: Getting the tool from A to B (Cartesian space)

Task: Calculating a path (sequence of all the joint values, joint space)

Constraint: Avoiding collisions (incl self-collisions)

For calculations we need A and B as well as the **kinematic description of the robot**

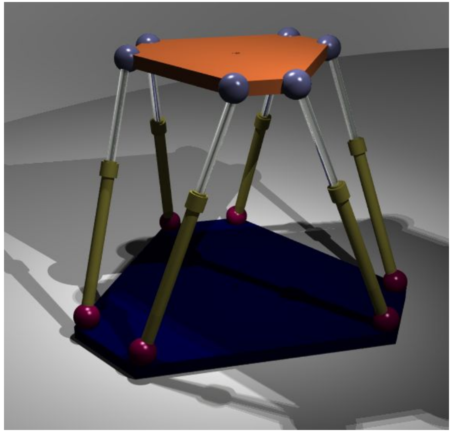
- The **position and orientation** of a rigid body in space are collectively termed the **pose**.

Robot kinematics describes the pose, velocity, acceleration, and all higher-order derivatives of the pose of the bodies that comprise a mechanism.

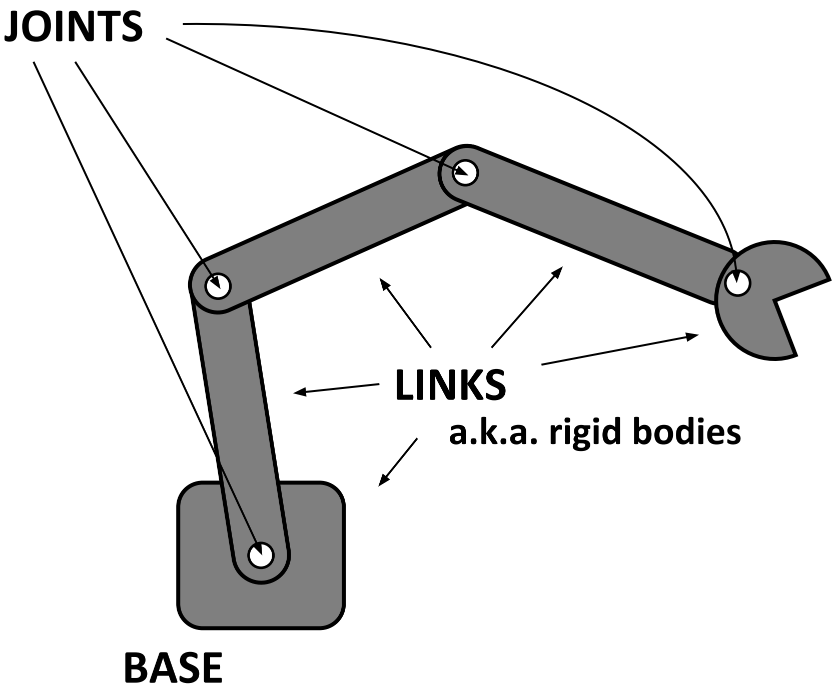
Among the many **possible topologies** in which systems of bodies can be connected, two are of particular importance in robotics:

1) serial chains

2) parallel mechanisms

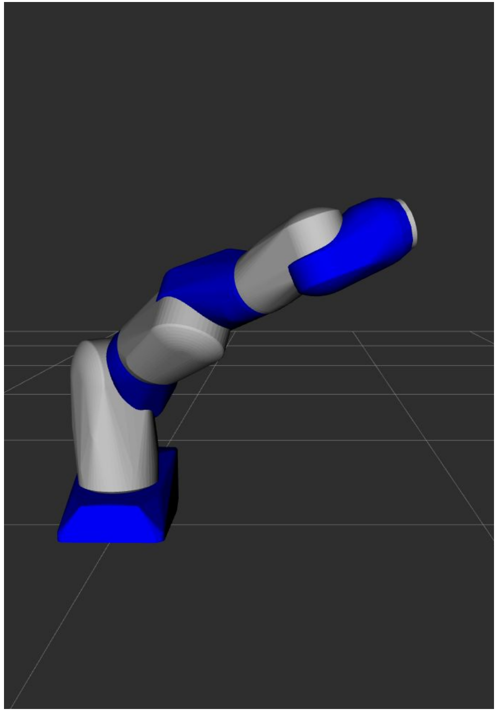


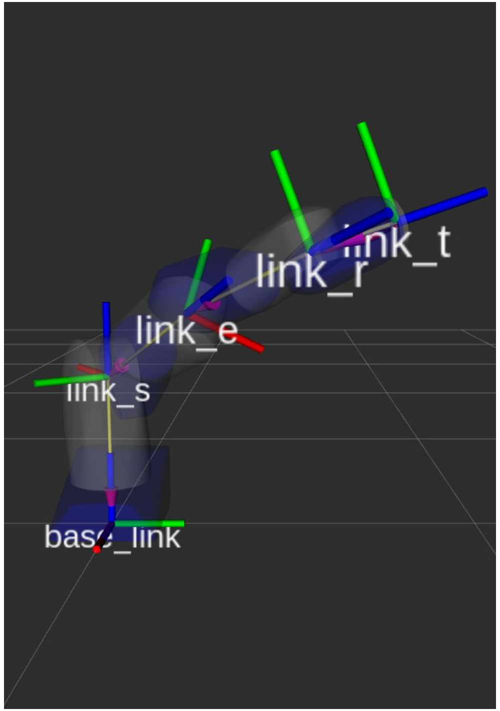
Links, joints, base, and end-effector

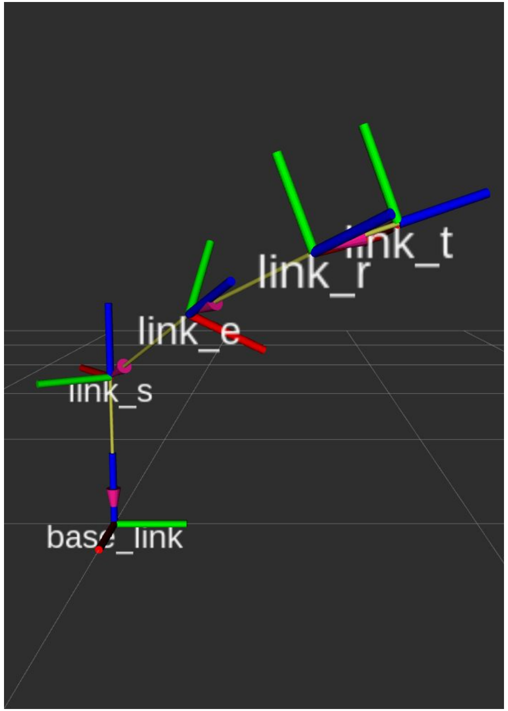


An **END EFFECTOR** is the device at the end of a robotic arm, designed to interact with the environment. The exact nature of this device depends on the application of the robot.

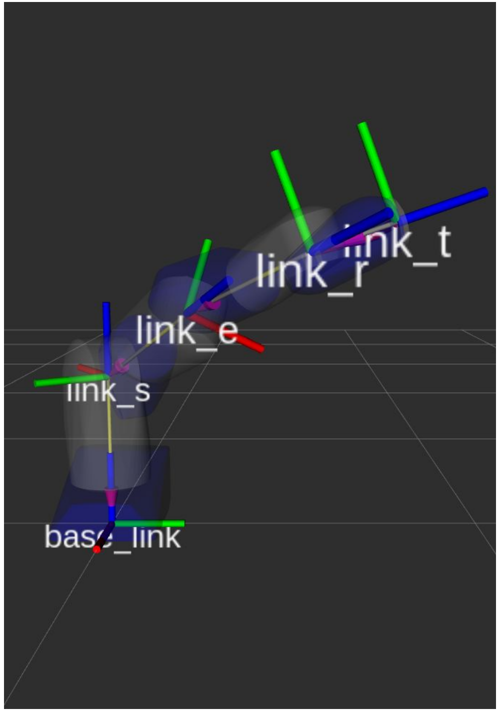
Robot manipulator as a series of frames

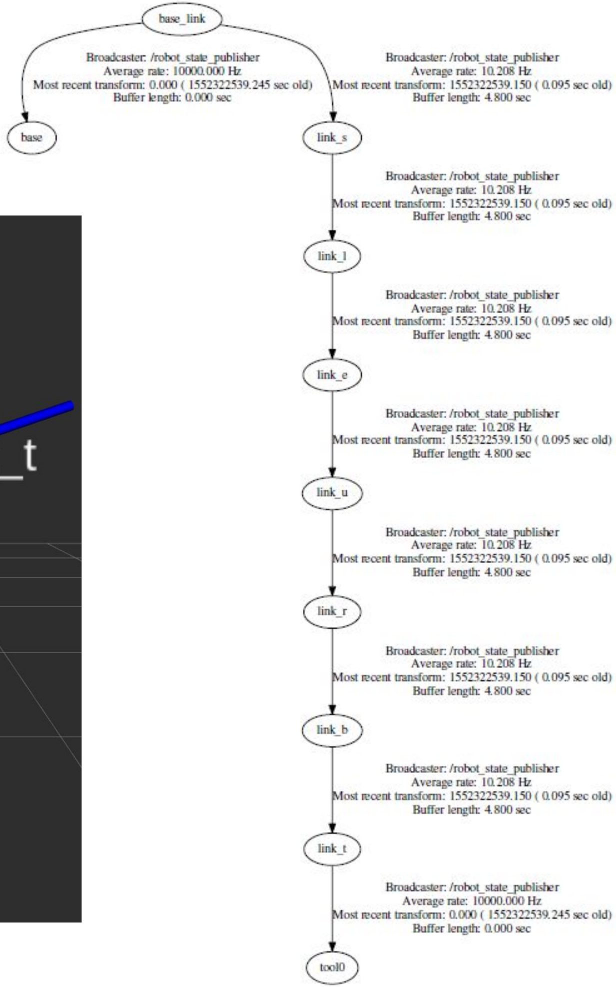






tf tree


link_r
link_e


base_link
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1552322539.245 sec old)
Buffer length: 0.000 sec
base
Broadcaster: /robot_state_publisher
Average rate: 10.208 Hz
Most recent transform: 1552322539.150 (0.095 sec old)
Buffer length: 4.800 sec
link_s
Broadcaster: /robot_state_publisher
Average rate: 10.208 Hz
Most recent transform: 1552322539.150 (0.095 sec old)
Buffer length: 4.800 sec
link_l
Broadcaster: /robot_state_publisher
Average rate: 10.208 Hz
Most recent transform: 1552322539.150 (0.095 sec old)
Buffer length: 4.800 sec
link_e
Broadcaster: /robot_state_publisher
Average rate: 10.208 Hz
Most recent transform: 1552322539.150 (0.095 sec old)
Buffer length: 4.800 sec
link_u
Broadcaster: /robot_state_publisher
Average rate: 10.208 Hz
Most recent transform: 1552322539.150 (0.095 sec old)
Buffer length: 4.800 sec
link_r
Broadcaster: /robot_state_publisher
Average rate: 10.208 Hz
Most recent transform: 1552322539.150 (0.095 sec old)
Buffer length: 4.800 sec
link_b
Broadcaster: /robot_state_publisher
Average rate: 10.208 Hz
Most recent transform: 1552322539.150 (0.095 sec old)
Buffer length: 4.800 sec
link_t
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1552322539.245 sec old)
Buffer length: 0.000 sec
tool0

Relative poses of links

link_2 relative to link_1

- Position and orientation of link_2 relative to the ***origin*** of link_1

However, as robot manipulators have links moving relative to others, it is important to describe the degree of freedom for those movements

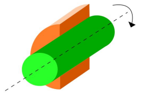
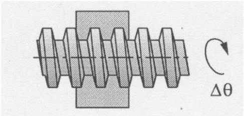
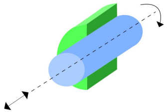
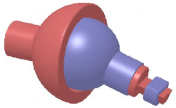
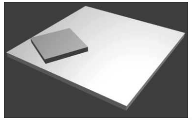
- link_2 is relative to link_1 but able to move in predetermined fashion
- there is a **joint** between link_2 and link_1

Joint kinematics

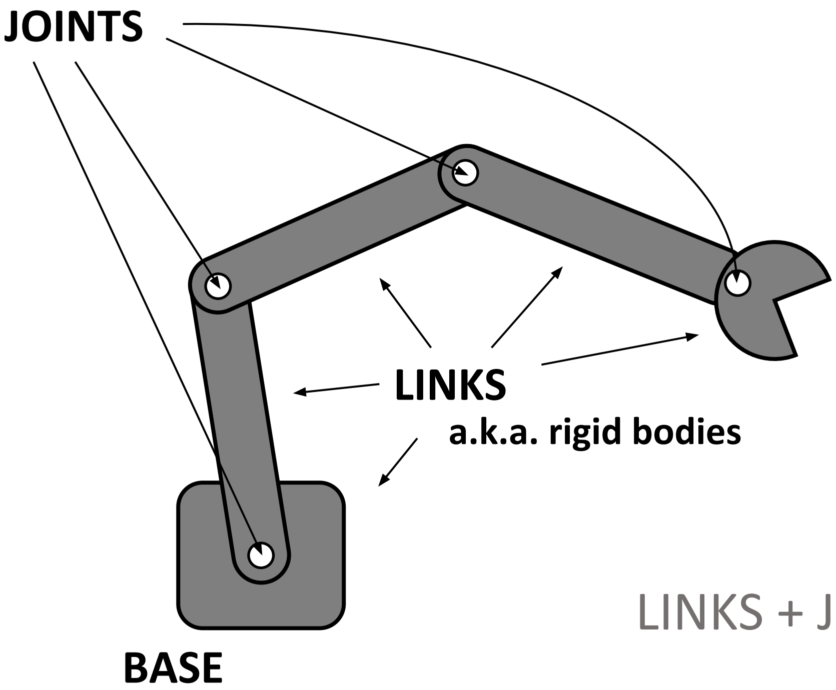
- A kinematic joint is a connection between two bodies that constrains their relative motion

- **There are only 6 forms of lower pair joints:**
    - *revolute, prismatic, helical, cylindrical, spherical, and planar joints*

Lower-pair joints

Joint		Geometry	DOF	Equivalent to
Revolute	R		1	
Prismatic	P		1	
Helical	H		1	
Cylindrical	C		2	R+P
Spherical	S		3	3xR w/ concurrent axes
Planar			3	3xR in series

Representation of robot



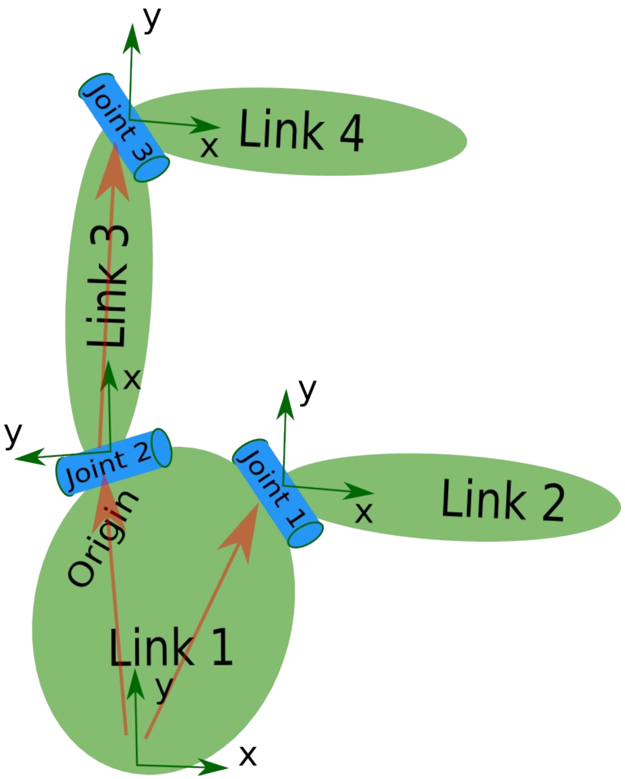
LINKS + JOINTS = **KINEMATIC CHAIN**

ROS uses URDF

Unified Robot Description Format (**URDF**) is an XML specification to describe a robot

The specification covers:

- Kinematic and dynamic description of the robot
- Visual representation of the robot
- Collision model of the robot



URDF basics

- The description of a robot consists of a set of link elements, and a set of joint elements connecting the links together.

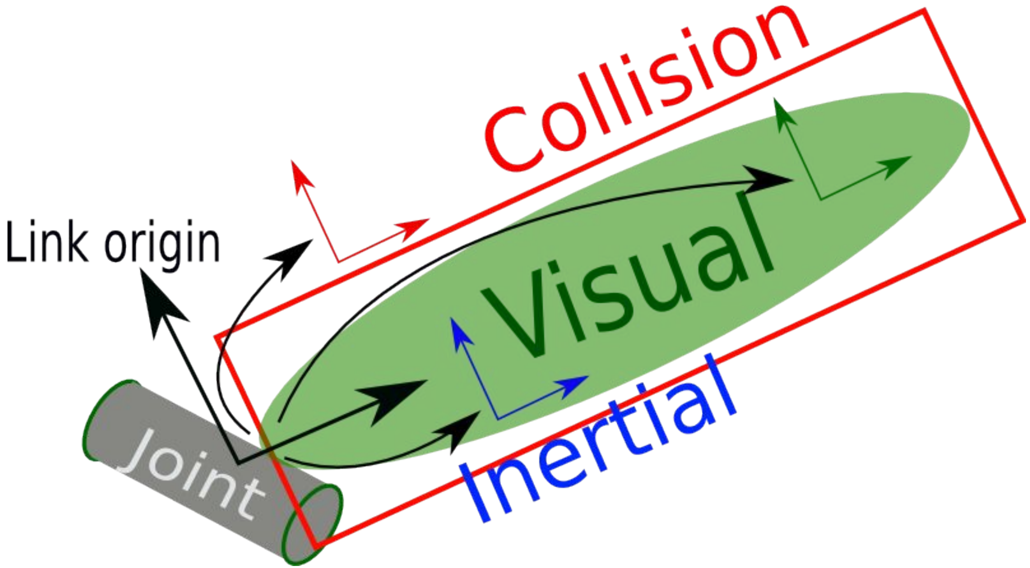
- A typical robot description looks something like this:

```
<robot name=„my_robot">
  <link> ... </link>
  <link> ... </link>
  <link> ... </link>

  <joint>  ....  </joint>
  <joint>  ....  </joint>
  <joint>  ....  </joint>
</robot>
```

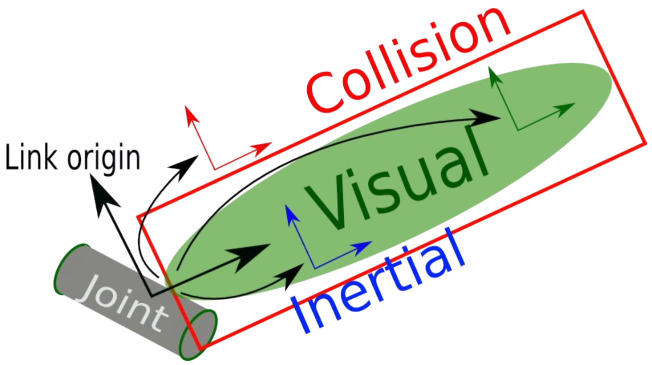
<link> element

The link element describes a rigid body with an **inertia**, **visual** **features**, and **collision** **space**



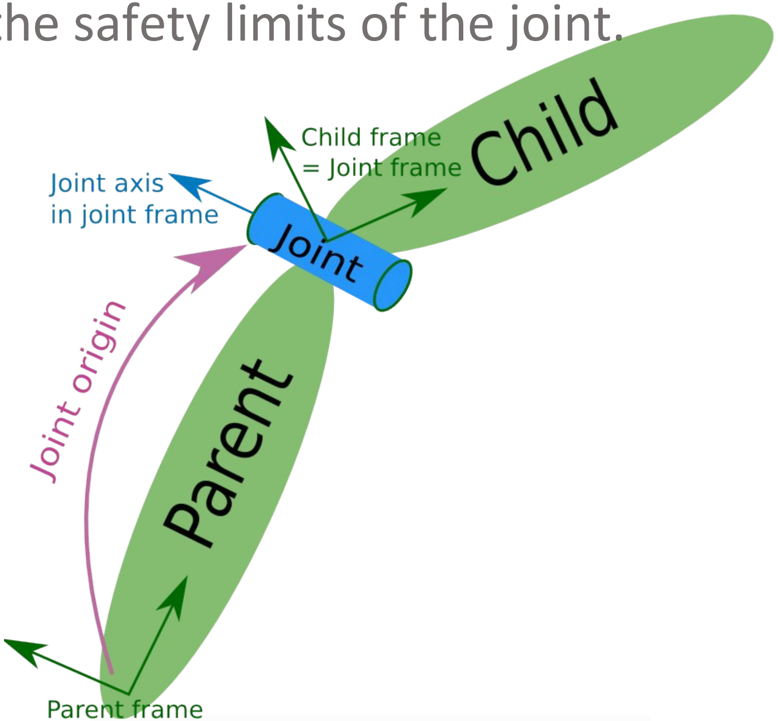
Example of link element

```
<link name="my_link">
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <box size="1 1 1" />
    </geometry>
    <material name="Cyan">
      <color rgba="0 1.0 1.0 1.0"/>
    </material>
  </visual>
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <cylinder radius="1" length="0.5"/>
    </geometry>
  </collision>
  <inertial>
    <origin xyz="0 0 0.5" rpy="0 0 0"/>
    <mass value="1"/>
    <inertia ixx="100"  ixy="0"  ixz="0" iyy="100" iyz="0" izz="100" />
  </inertial>
</link>
```



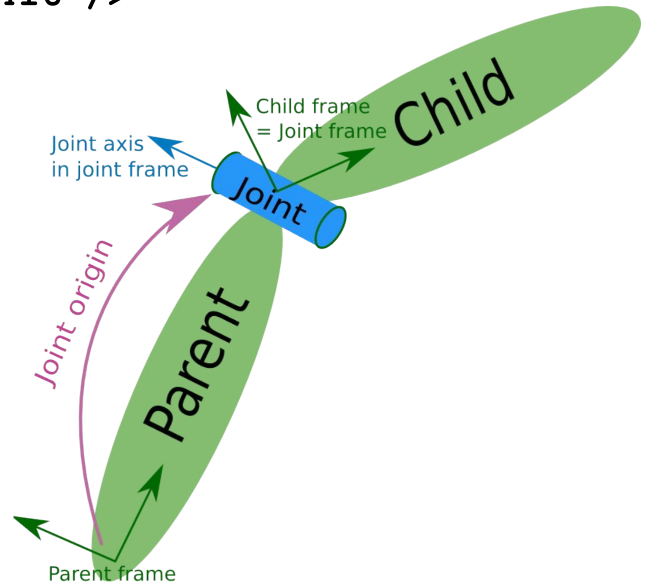
<joint> element

The joint element describes the **kinematics** and **dynamics** of the joint and also specifies the safety limits of the joint.



Example of simple joint element

```
<joint name="my_joint" type="floating">
   <origin xyz="0 0 1" rpy="0 0 3.1416"/>
   <parent link="link1"/>
   <child link="link2"/>
</joint>
```



MoveIt

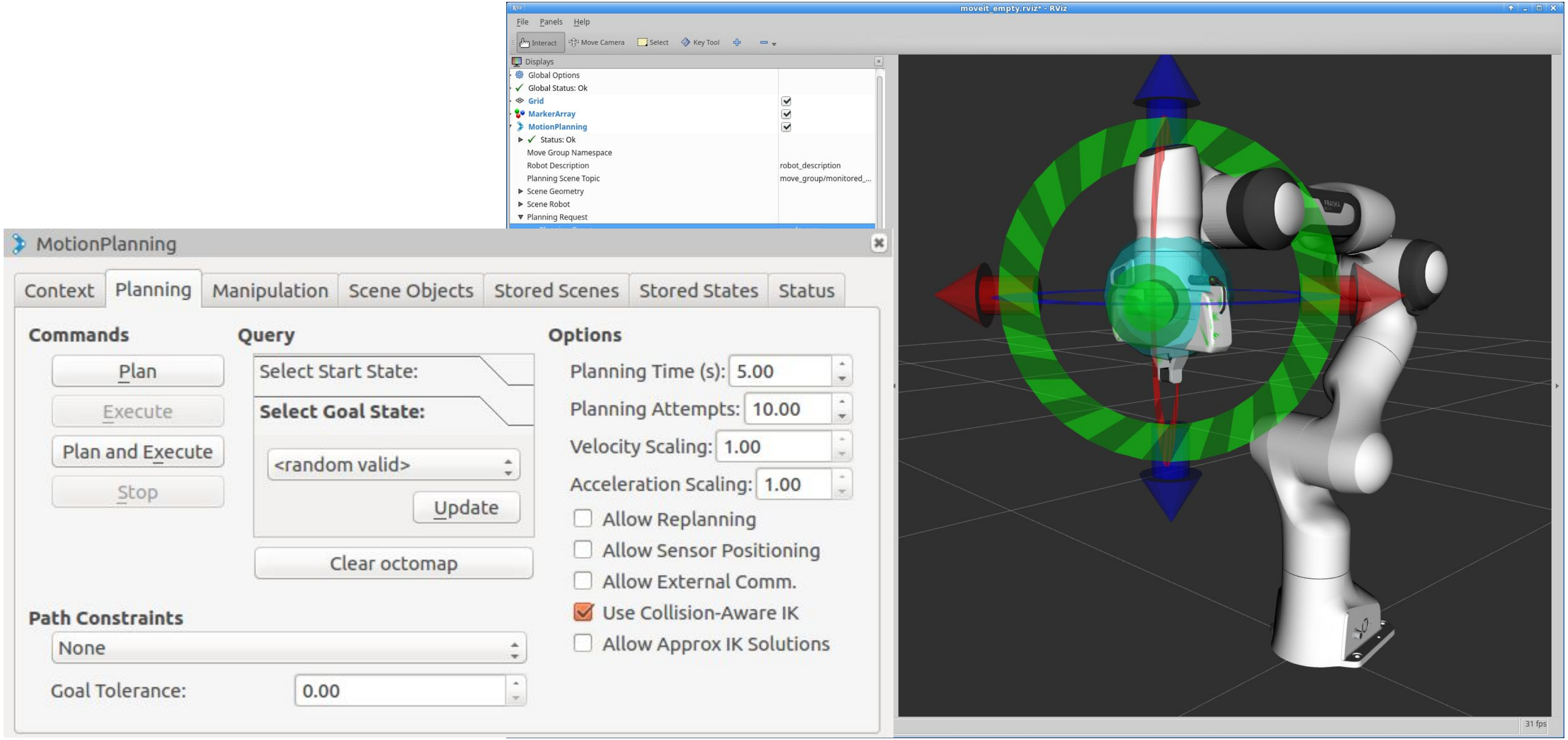
moveit.ros.org

Easy-to-use robotics manipulation platform for developing applications, evaluating designs, and building integrated products

It binds together **robot description**, computations of **kinematics**, algorithms of **motion planning**, graphical user **interface**, and **ROS**

To use MoveIt, one needs a **MoveIt configuration package** or a URDF to generate MoveIt configuration and a **ROS driver** for the robot

MoveIt GUI



moveit_empty.rviz* - RViz
File Panels Help
Interact Move Camera Select Key Tool
Displays
Global Options
Global Status: Ok
Grid
MarkerArray
MotionPlanning
Status: Ok
Move Group Namespace
Robot Description
robot_description
Planning Scene Topic
move_group/monitored_...
Scene Geometry
Scene Robot
Planning Request
MotionPlanning
Context
Planning
Manipulation
Scene Objects
Stored Scenes
Stored States
Status
Commands
Plan
Execute
Plan and Execute
Stop
Query
Select Start State:
Select Goal State:
<random valid>
Update
Clear octomap
Options
Planning Time (s): 5.00
Planning Attempts: 10.00
Velocity Scaling: 1.00
Acceleration Scaling: 1.00
Allow Replanning
Allow Sensor Positioning
Allow External Comm.
Use Collision-Aware IK
Allow Approx IK Solutions
Path Constraints
None
Goal Tolerance: 0.00
31 fps

Move Group C++ Interface

Within a source file for ROS node:

1. Initialize ROS
2. Create an instance of MoveGroupInterface for the robot
3. Run-time configuration (optional)
4. Set goal state for the robot
5. Trigger motion-planning
6. Execute successfully generated motion plan

Move Group C++ Interface

Example

1. Initialize ROS
2. Create an instance of MoveGroupInterface for the robot
```
moveit::planning_interface::MoveGroupInterface mg("xarm7");
```
3. Set Goal State for the robot
```
mg.setNamedTarget("home_pose");
```
4. Trigger motion-planning
```
moveit::planning_interface::MoveGroupInterface::Plan my_plan;
mg.plan(my_plan);
```
5. Exectute sucessfully generated motion plan
```
mg.execute(my_plan);
```

Summary

- **Pose** describes the position and orientation simultaneously
- **Quaternion** is a 4-number representation of orientation/rotation
- A **kinematic joint** is a connection between two bodies that constrains their relative motion
- **URDF** allows describing robot kinematics in a uniform way
- **MoveIt** is the go-to ROS tool for making manipulators move safely

Workshops

1. MoveIt GUI
2. URDF
3. MoveIt configuration package
4. MoveGroup C++ interface
5. Multi-robot motion planning


2

ROS 2

More turtles bring bigger changes

"*ardent*"

12/2017

"*bouncy*"

07/2018

"*crystal*"

12/2018

"*dashing*"

05/2019

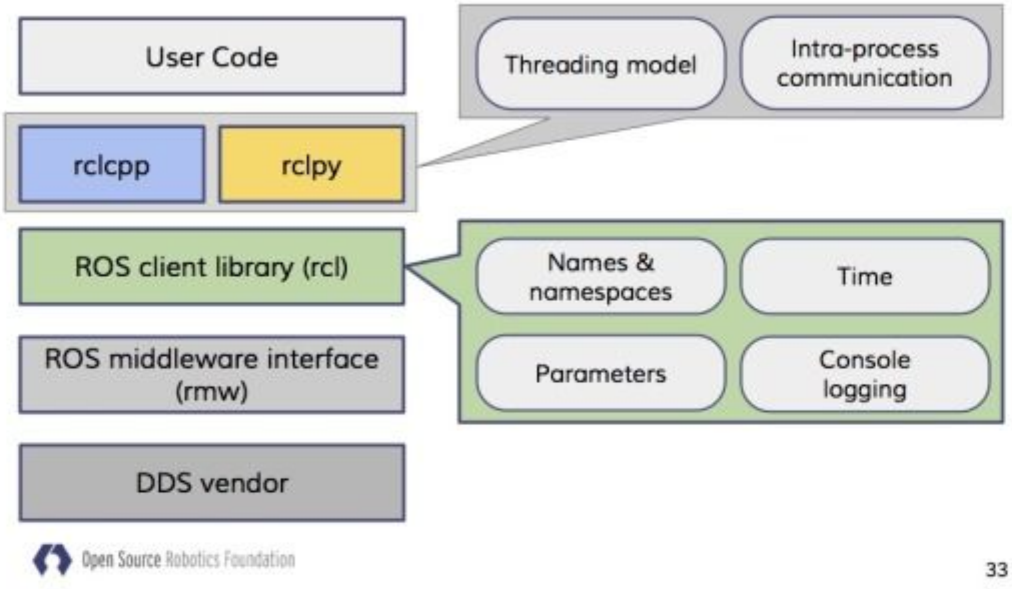
Eloquent Elusor

"*eloquent*"

11/2019

ROS Master replaced by DDS!

ROS Client Libraries



Launch files

- No XML
- Python3 API
- Granular execution models (executors)
- Real-time nodes (when using RTOS)

sourcing

Works like in ROS1

```
source ~/catkin_ws/install/setup.bash
```

ROS2 also has local sourcing

```
source ~/catkin_ws/install/local_setup.bash
```

DOES NOT INCLUDE PARENT WORKSPACES!

Build tools

Goodbye `catkin`, welcome `ament`.

To build everything in the "ros2_ws"

```
colcon build
```

TTo compile a single package

```
colcon build --symlink-install --packages-select clearbot_driver
```

Backwards compatibility

CMakeLists.txt

Leave the stone-age: `set(CMAKE_CXX_STANDARD 14)`

```
find_package(ament_cmake REQUIRED)
find_package(component1 REQUIRED)
# ...
find_package(componentN REQUIRED)
```

```
CATKIN_DEPENDS   →  ament_export_dependencies(...)
INCLUDE_DIRS     →  ament_export_include_directories(...)
LIBRARIES        →  ament_export_libraries(...)
```

Build differences

- Only isolated builds are supported
- No devel space - in ROS 2 a package must be installed after building it before it can be used
- Symlinks are still used in install space!
- CMake API is Restructured

From ROS1 pkg to ROS2 pkg?

Use the migration guide:

https://index.ros.org/doc/ros2/Contributing/Migration-Guide/

Changes of conventions

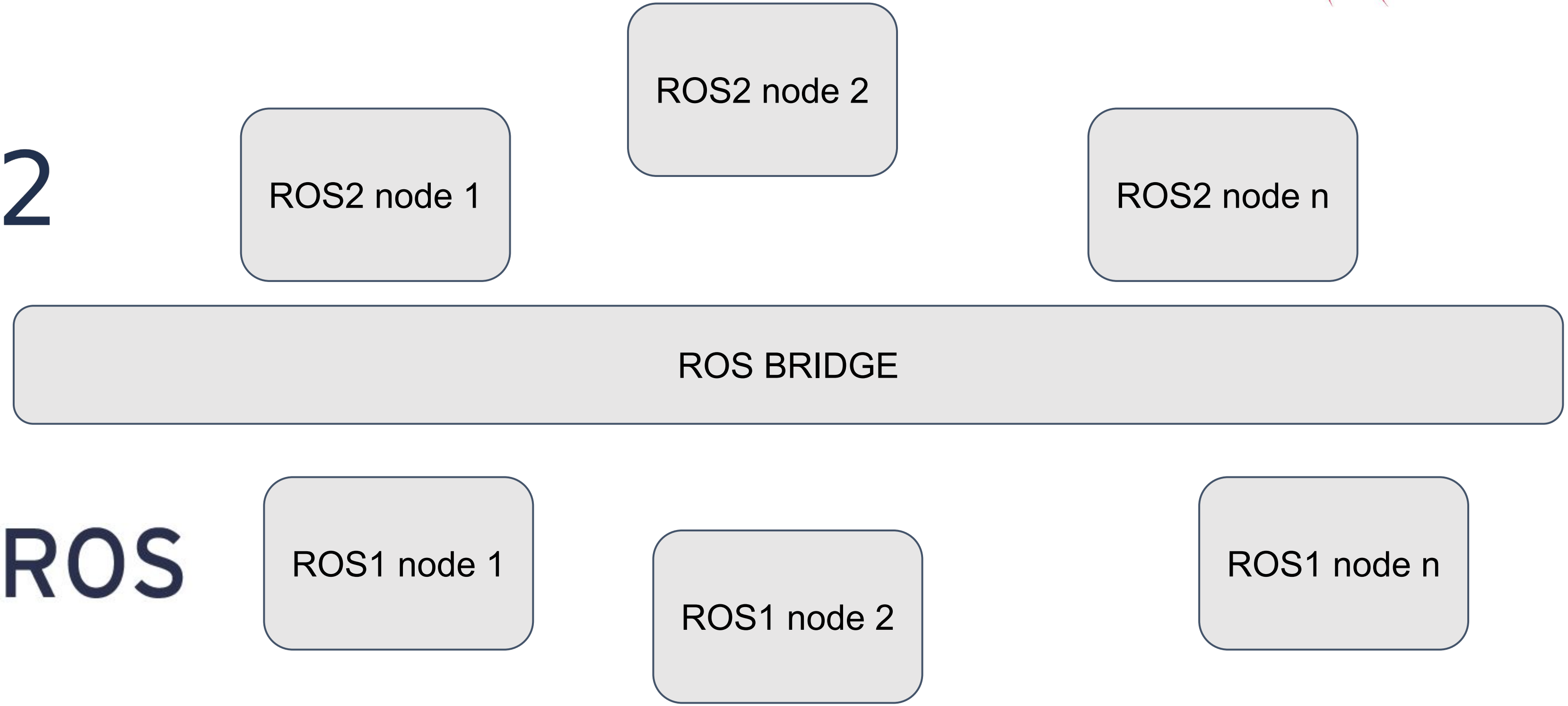
Messages are namespaced:

```
geometry_msgs::msg::PointStamped point_stamped;
```

Name header files `*.hpp`

```
void service_callback(
  const std::shared_ptr<nav_msgs::srv::GetMap::Request> request,
  std::shared_ptr<nav_msgs::srv::GetMap::Response> response)
{
  // return true;  // or false for failure
}
```

ROS Bridge


ROS2 node 2
ROS2 node 1
ROS2 node n
ROS BRIDGE
ROS1 node 1
ROS1 node 2
ROS1 node n

Feedback & discussion

https://tinyurl.com/y6pbjnxh

Any other feedback welcome:

veiko.vunder@ut.ee