Grundlagen der Robotik/Medizinrobotik

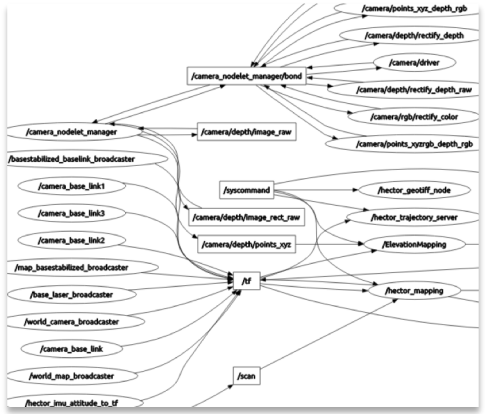
Introduction to the Robot Operating System (a.k.a. ROS)

Universität Augsburg University

What is ROS (Robot Operating System)?

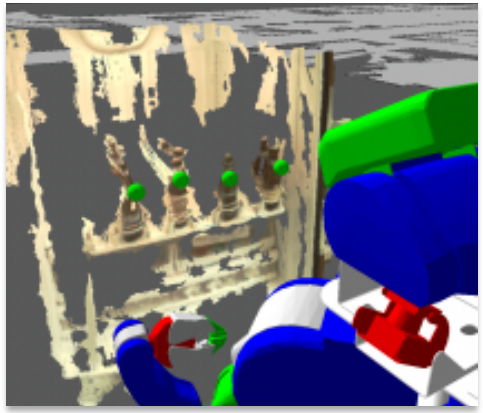
Institute for Software & Systems Engineering

Plumbing



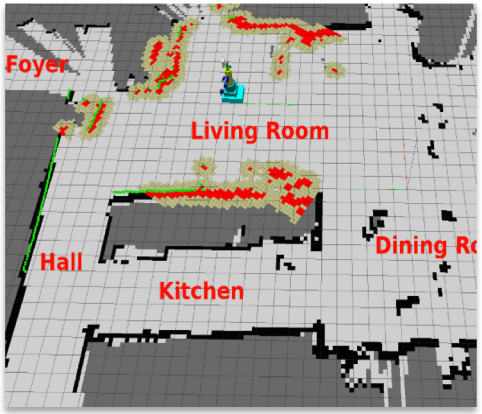
- Process Management
- Inter-process communication
- Device drivers

Tools



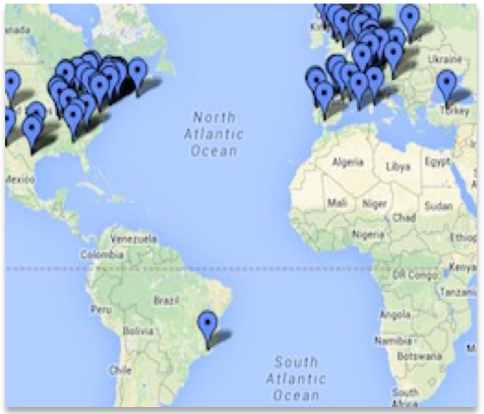
- Simulation
- Visualization
- Graphical User Interface
- Data Logging

Capabilities



- Control
- Planning
- Perception
- Mapping
- Manipulation

Ecosystem



- Package organization
- Software distribution
- Documentation
- Tutorials

What is ROS and what not…

Institute for Software & Systems Engineering

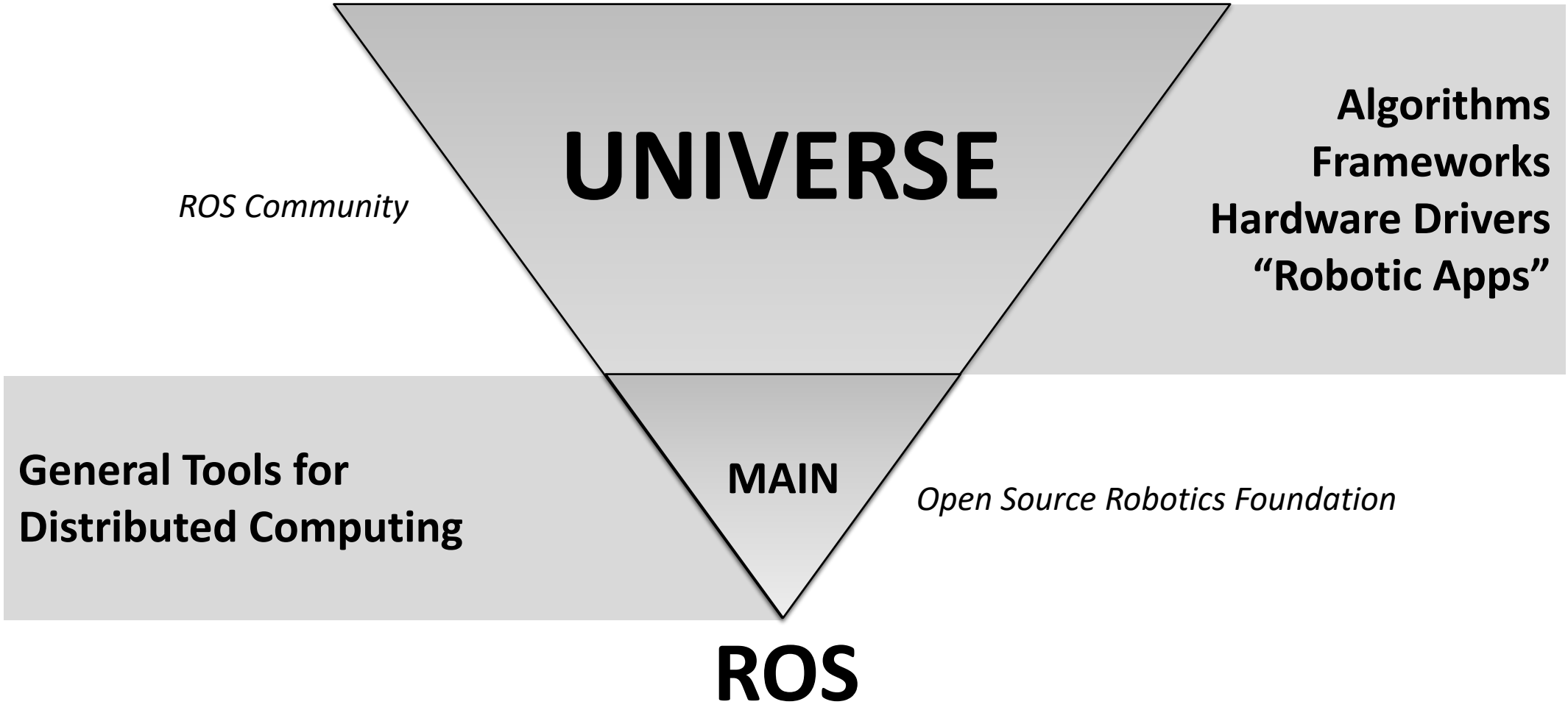
- ROS is
  - a *meta* operating system for robots
  - an architecture for distributed inter-process/inter-machine communication and configuration
  - a collection of development tools for system runtime and data analysis
  - a language-independent architecture (C++, python, lisp, java, and more)
- ROS is not
  - an *actual* operating system
  - a programming language
  - a programming environment / IDE
  - a hard real-time architecture

ROS Philosophy

Institute for Software & Systems Engineering

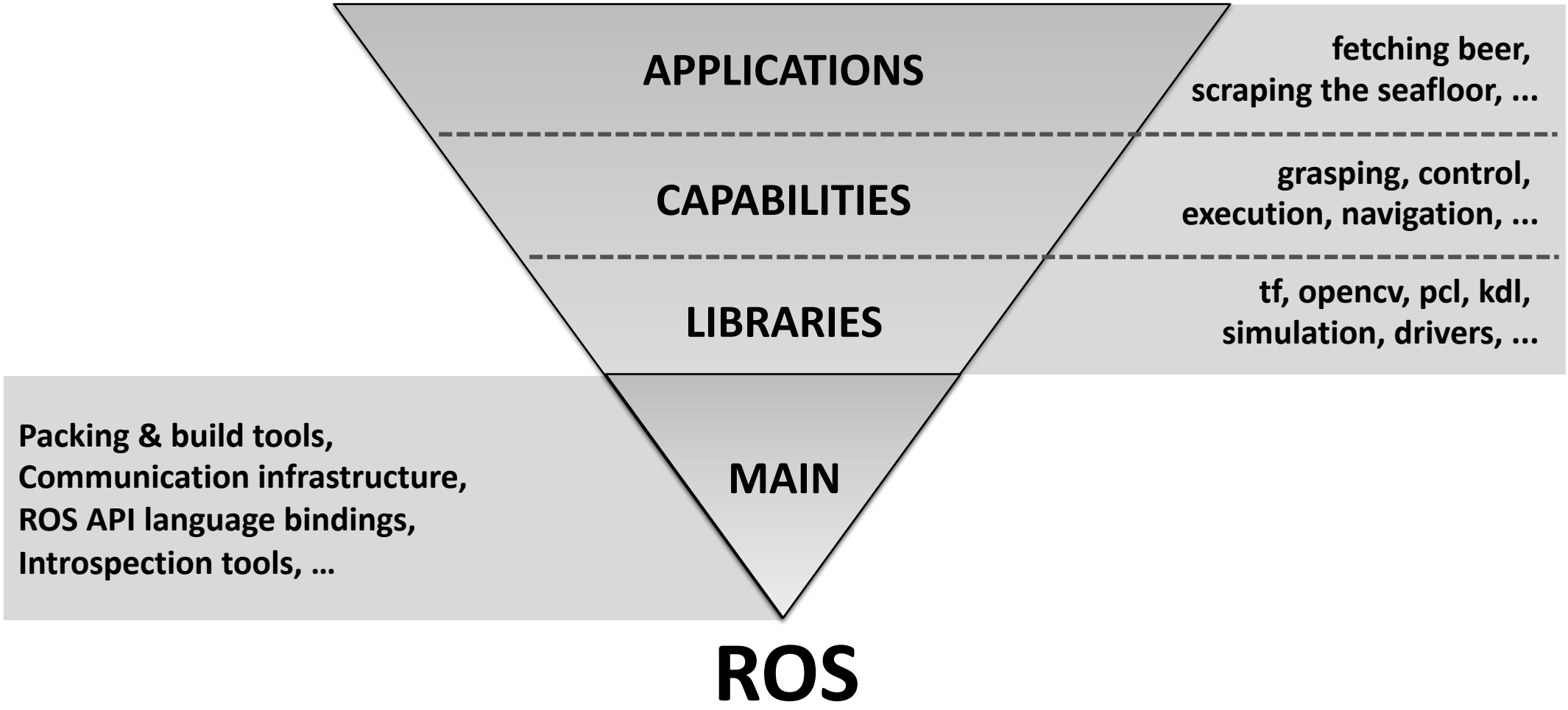
- **Peer to peer**
  Individual programs communicate over defined API (ROS *messages*, *services*, etc.).
- **Distributed**
  Programs can be run on multiple computers and communicate over the network.
- **Multi-lingual**
  ROS modules can be written in any language for which a client library exists (C++, Python, MATLAB, Java, etc.).
- **Light-weight**
  Stand-alone libraries are wrapped around with a thin ROS layer.
- **Free and open-source**
  Most ROS software is open-source and free to use.

What does ROS get you?



What does ROS get you?

Institute for Software & Systems Engineering



ROS Core

Institute for Software & Systems Engineering

- **ROS Master**
  - Manages the communication between nodes
  - Every node registers at startup with the master

- **Parameter Server**
  Stores persistent configuration parameters and other arbitrary data

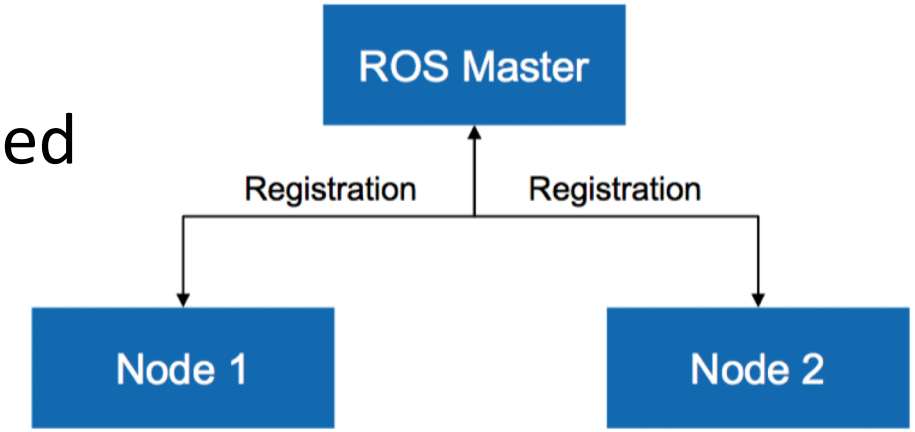
- **rosout**
  Essentially a network-based *stdout* for human-readable messages

More info
http://wiki.ros.org/Master

ROS Nodes

Institute for Software & Systems Engineering

- Single-purpose, executable program
- Individually compiled, executed, and managed
- Organized in *packages*

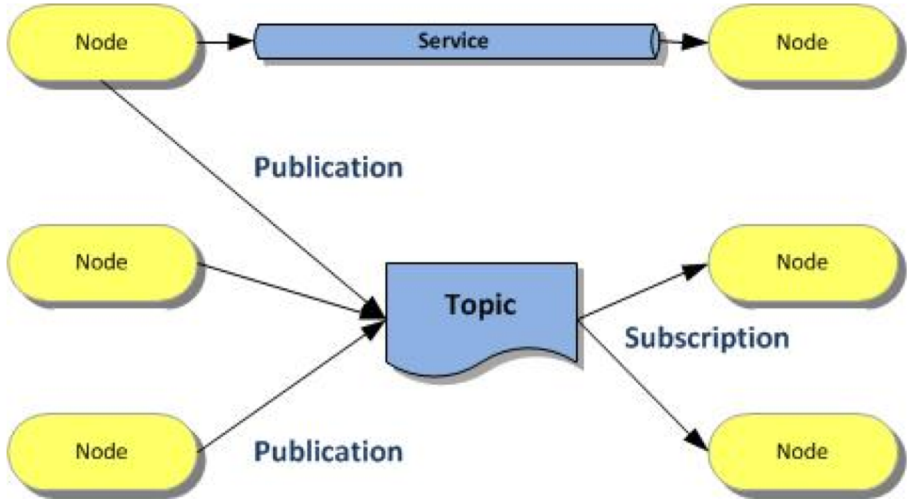


More info
http://wiki.ros.org/rosnode

Communication in ROS

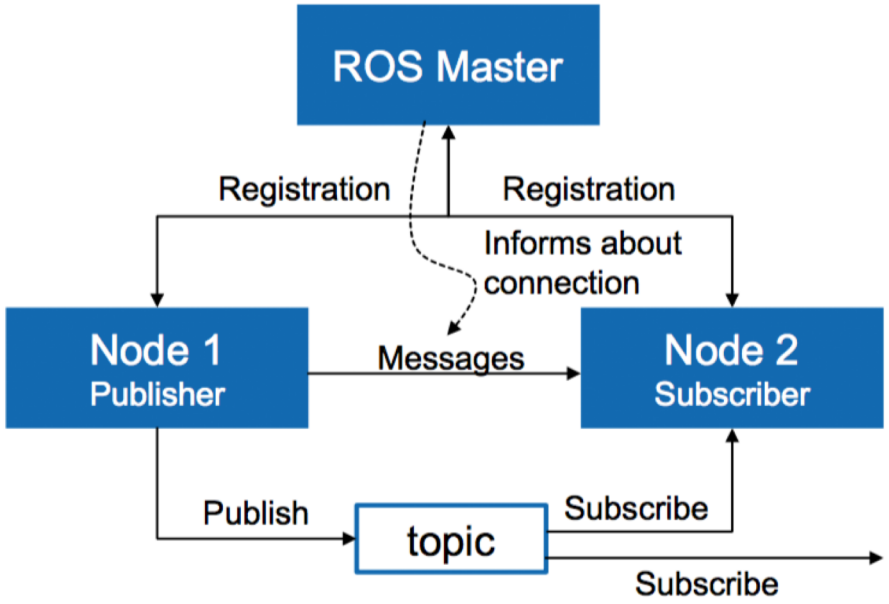
Institute for Software & Systems Engineering

- **ROS Topics**
  - Asynchronous “stream-like” communication
  - Strongly-typed (ROS .msg spec)
  - Can have one or more publishers
  - Can have one or more subscribers
- **ROS Services**
  - Synchronous “function-call-like” communication
  - Strongly-typed (ROS .srv spec)
  - Can have only one server
  - Can have one or more clients
- **Actions**
  - Built on top of topics
  - Long running processes
  - Cancellation



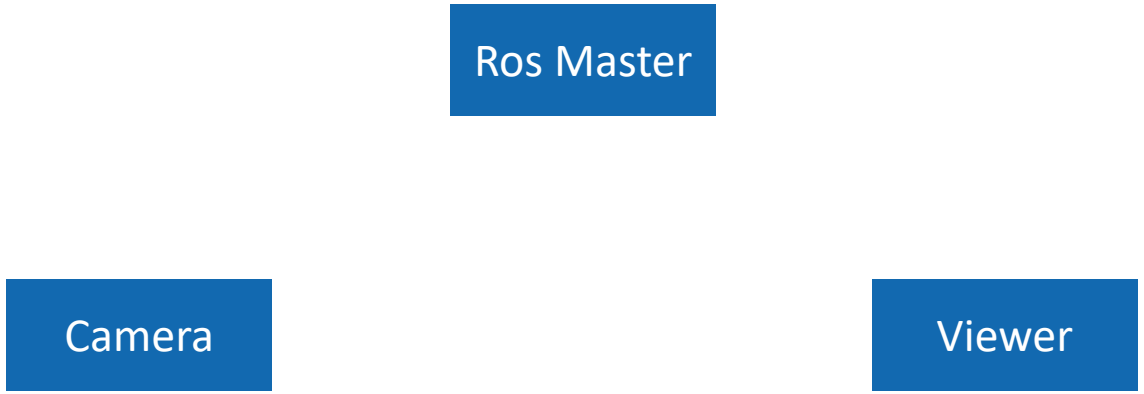
ROS Topics

- Nodes communicate over *topics*
  - Nodes can *publish* or *subscribe* to a topic
  - Typically: 1 publisher and *n* subscribers
- Topic is a name for a stream of *messages*

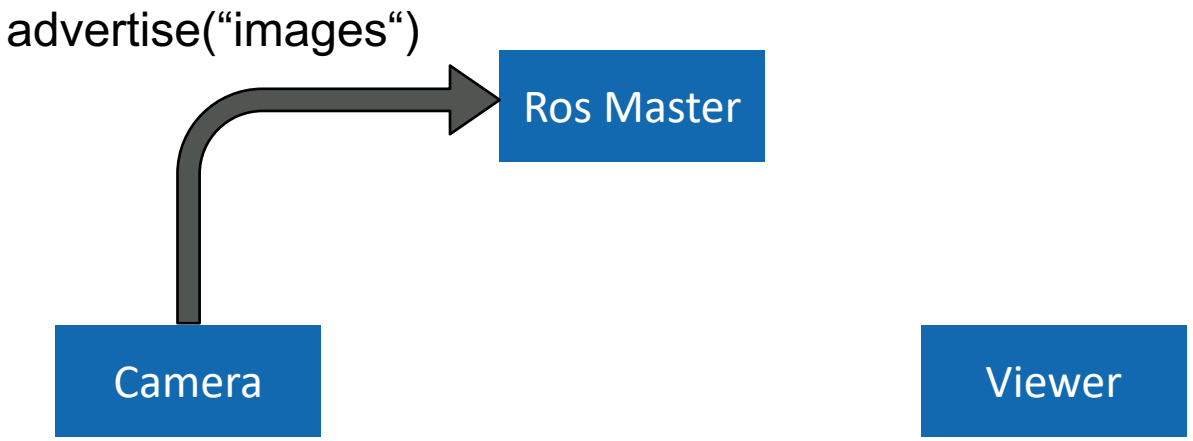


More info
http://wiki.ros.org/rostopic

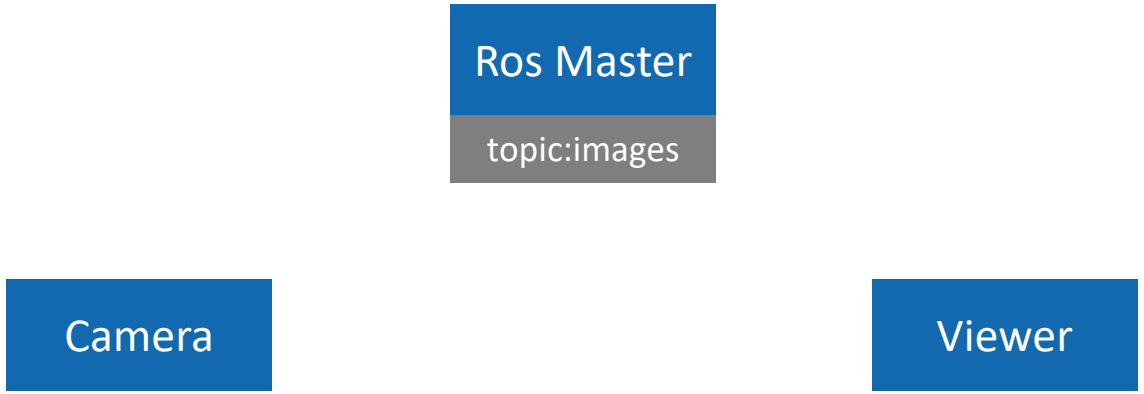
ROS Topics Example



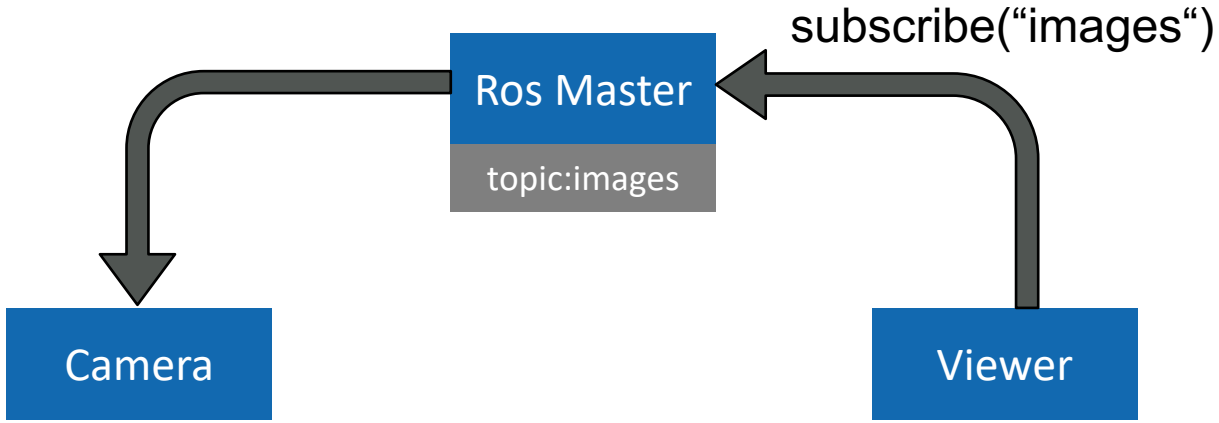
ROS Topics Example



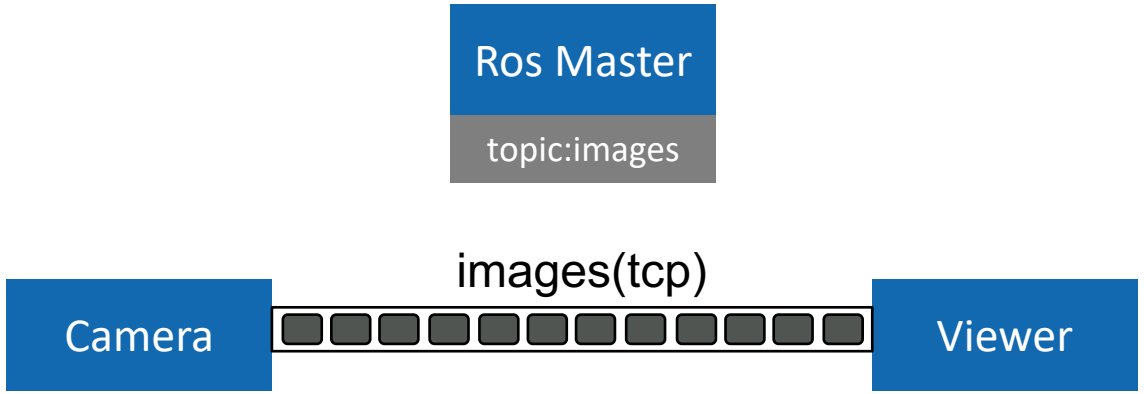
ROS Topics Example



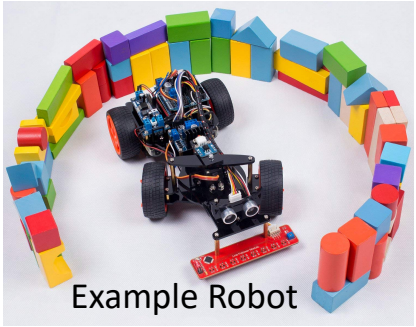
ROS Topics Example

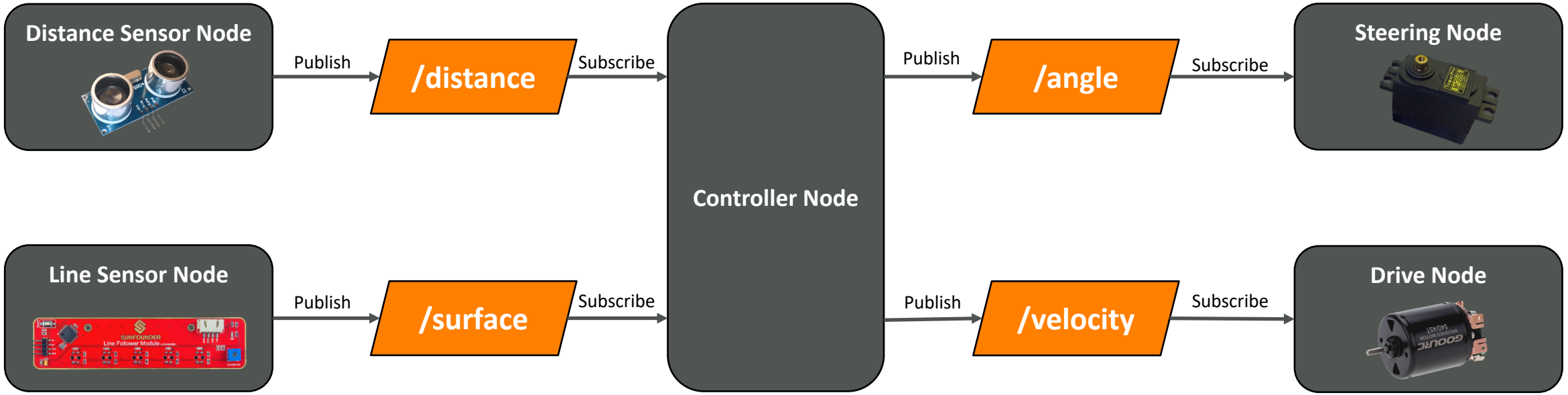


ROS Topics Example



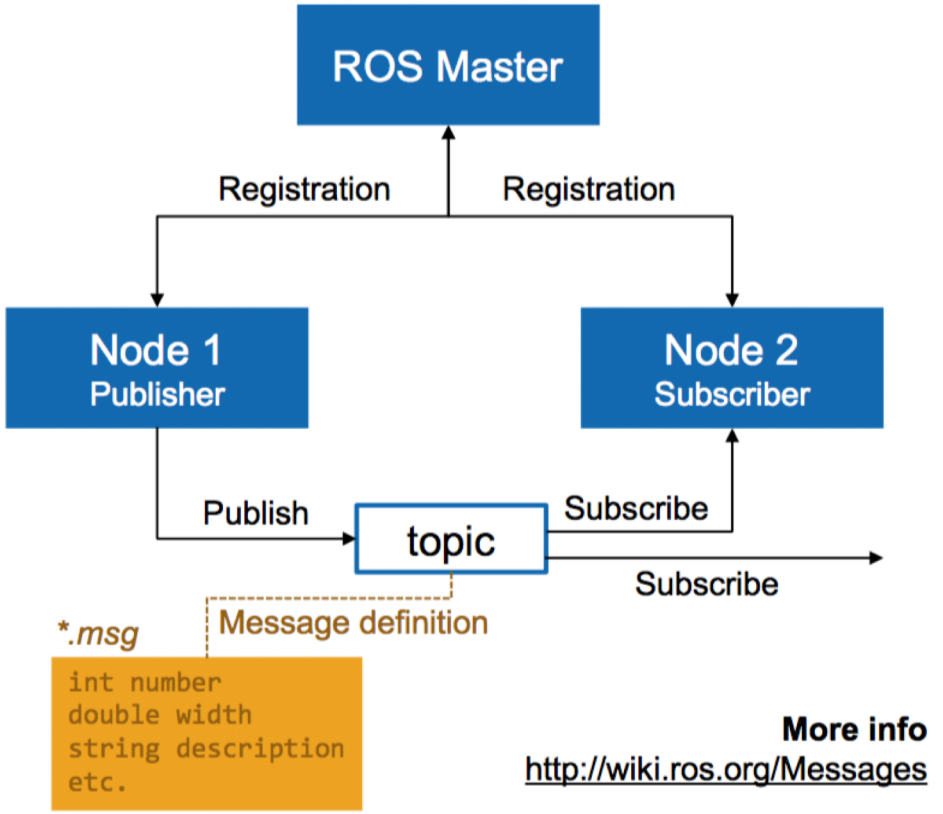
Node and Topic Example





ROS Messages

- Data structure defining the *type* of a topic
- Compromised of a nested structure of
  - integers,
  - floats,
  - booleans,
  - strings etc. and
  - arrays of objects
- Defined in **.msg* files



ROS Messages Example: PoseStamped

Institute for Software & Systems Engineering

geometry_msgs/Point.msg

```
float64 x
float64 y
float64 z
```

sensor_msgs/Image.msg

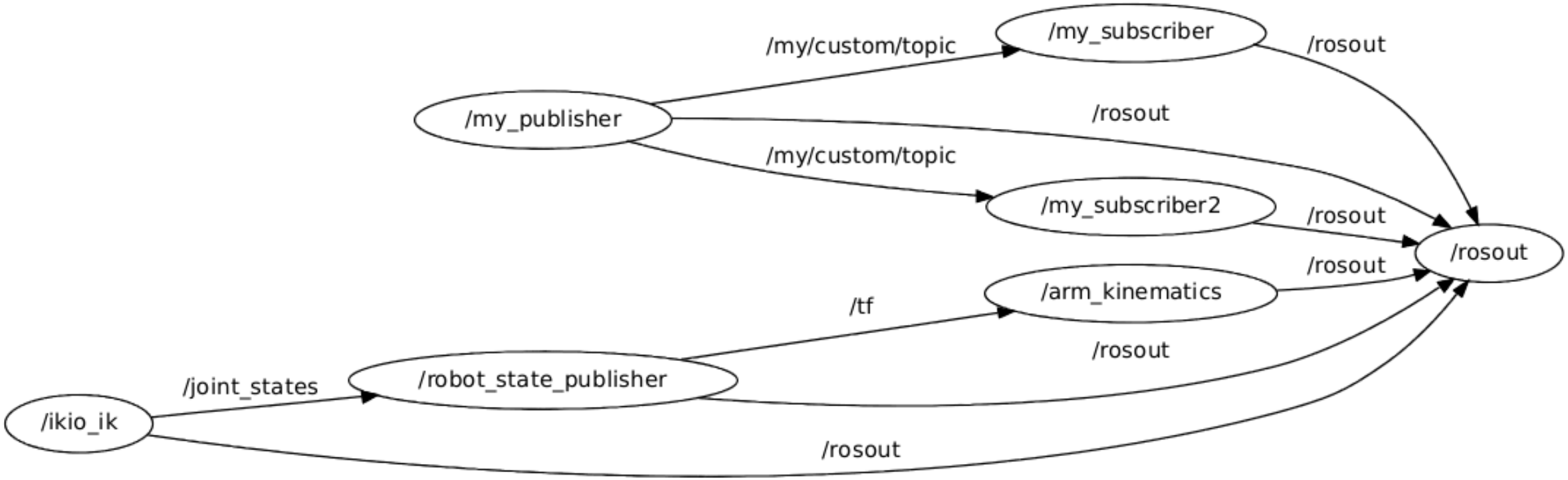
```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

geometry_msgs/PoseStamped.msg

```
std_msgs/Header header
 uint32 seq
 time stamp
 string frame_id
geometry_msgs/Pose pose
  geometry_msgs/Point position
     float64 x
     float64 y
     float64 z
   geometry_msgs/Quaternion
orientation
     float64 x
     float64 y
     float64 z
     float64 w
```

Graph of nodes and topics in ROS

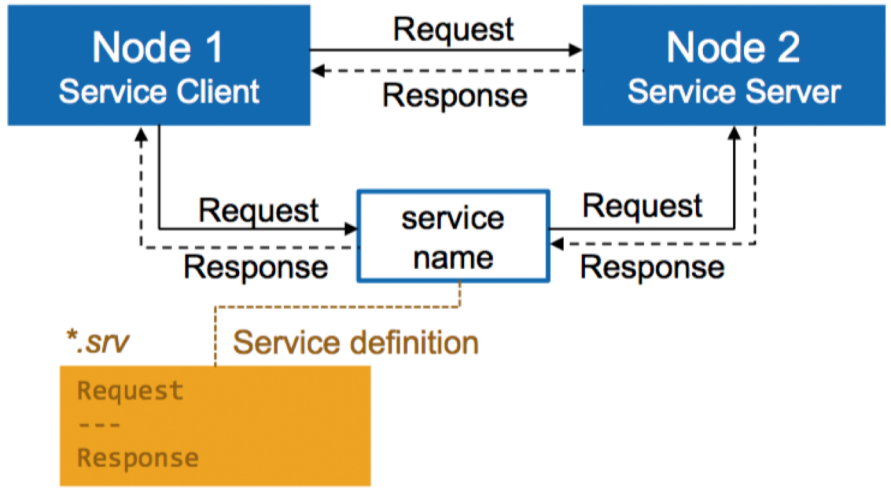
Institute for Software & Systems Engineering



ROS Services

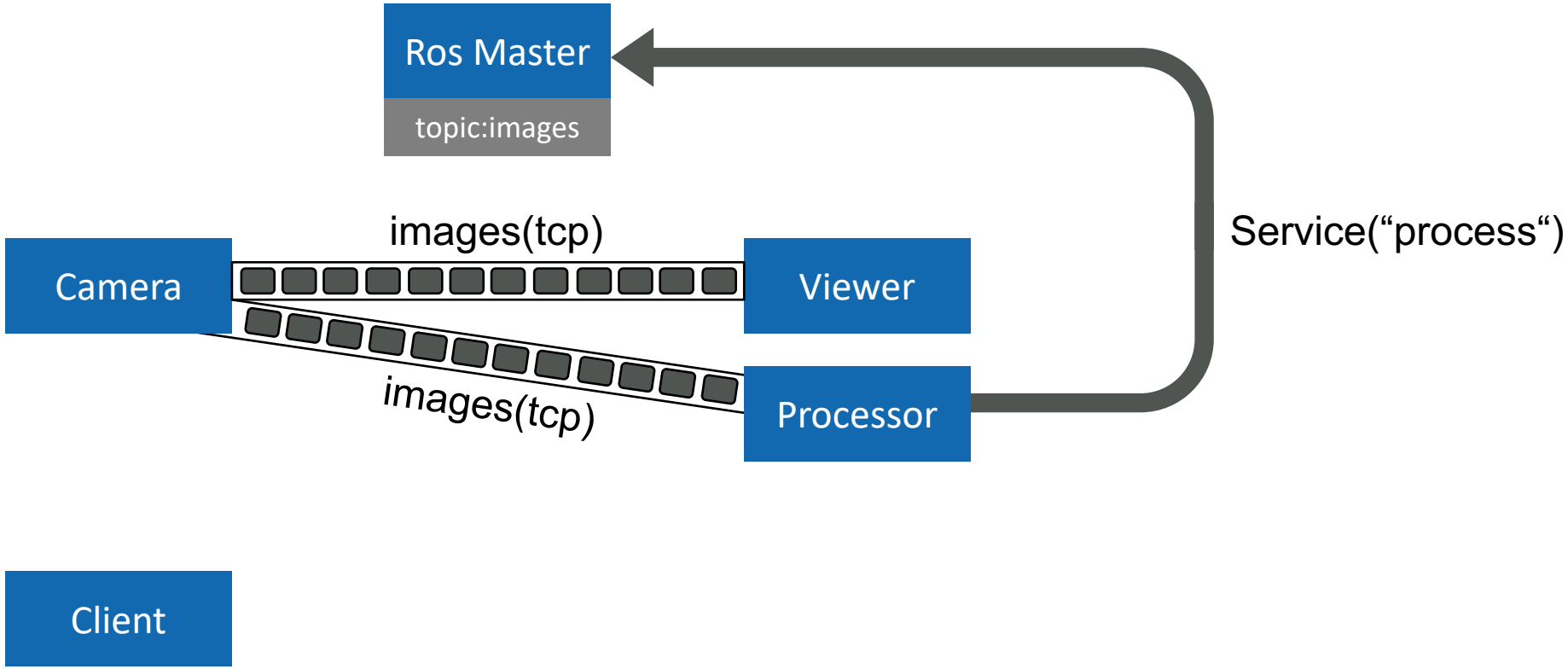
- Request/response communication between nodes is realized with *services*
  - The *service server* advertises the service
  - The *service client* accesses this service

- Similar in structure to messages, services are defined in **.srv* files

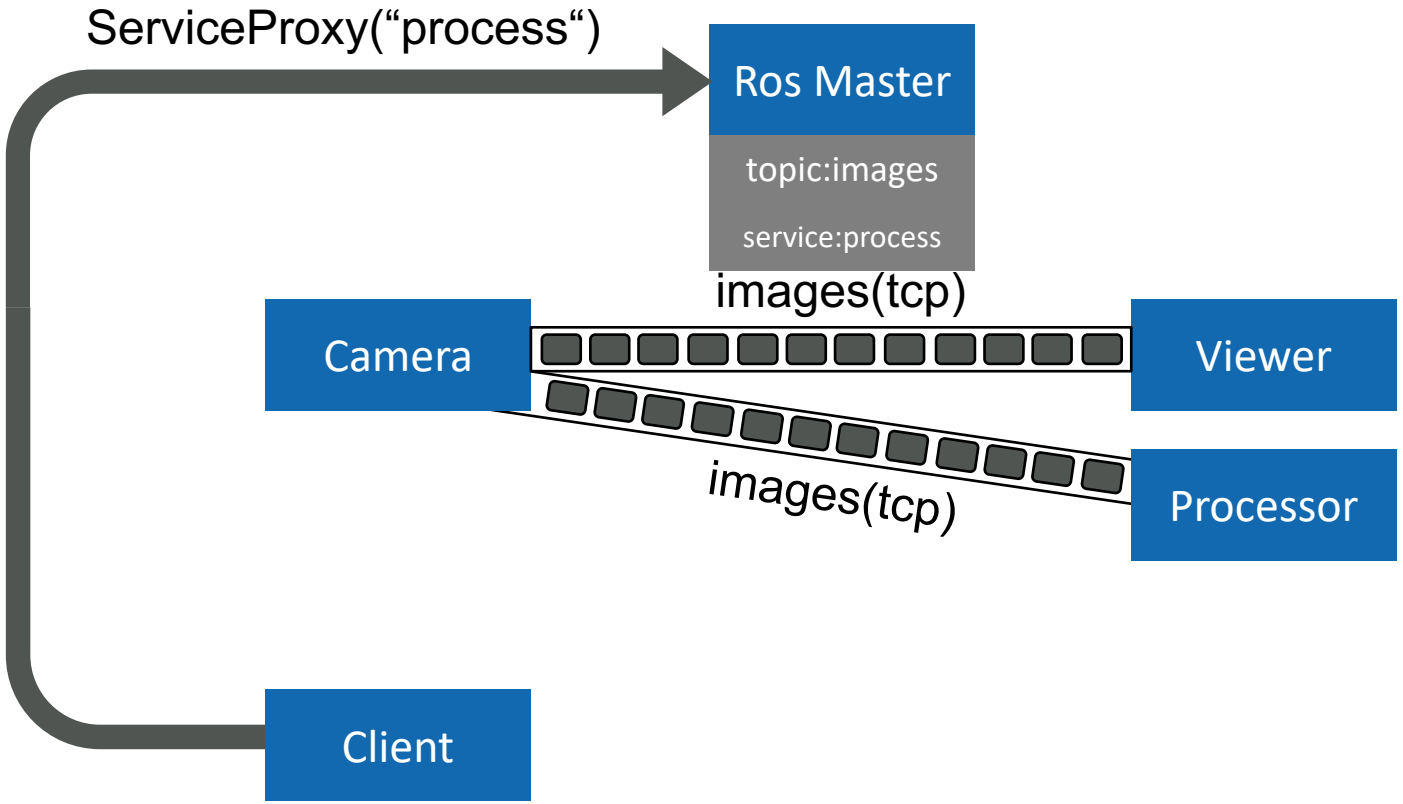


More info
http://wiki.ros.org/Services

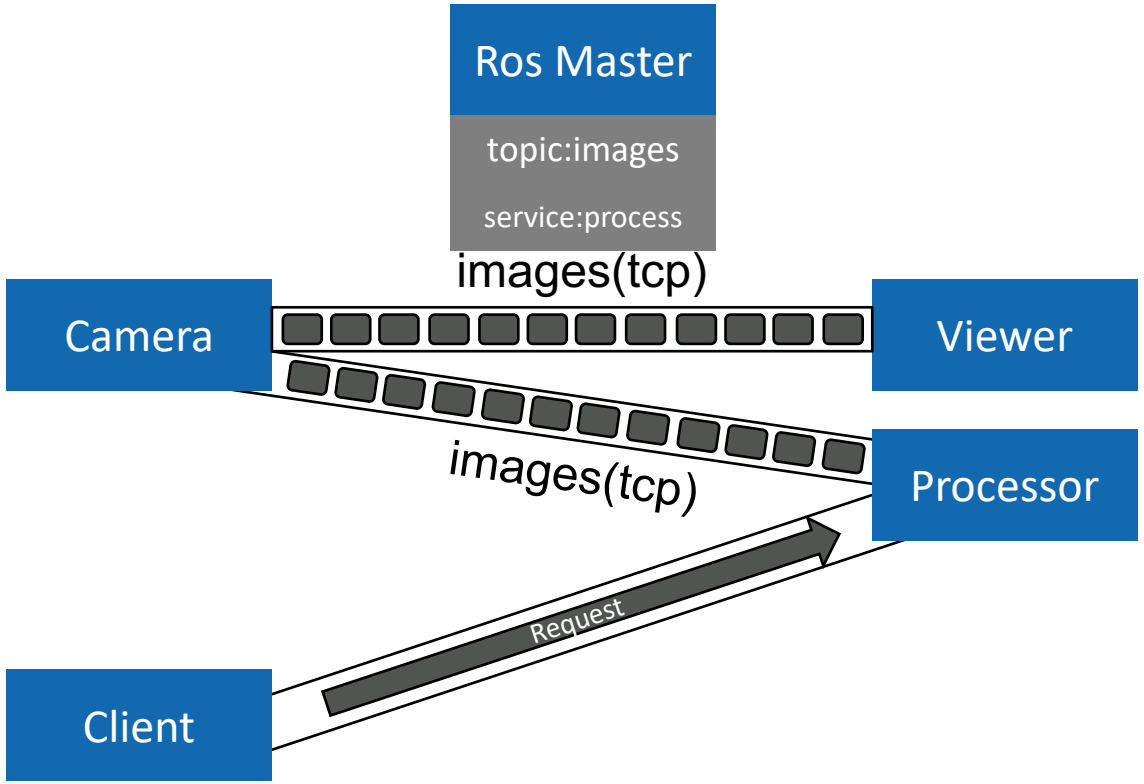
ROS Services Example



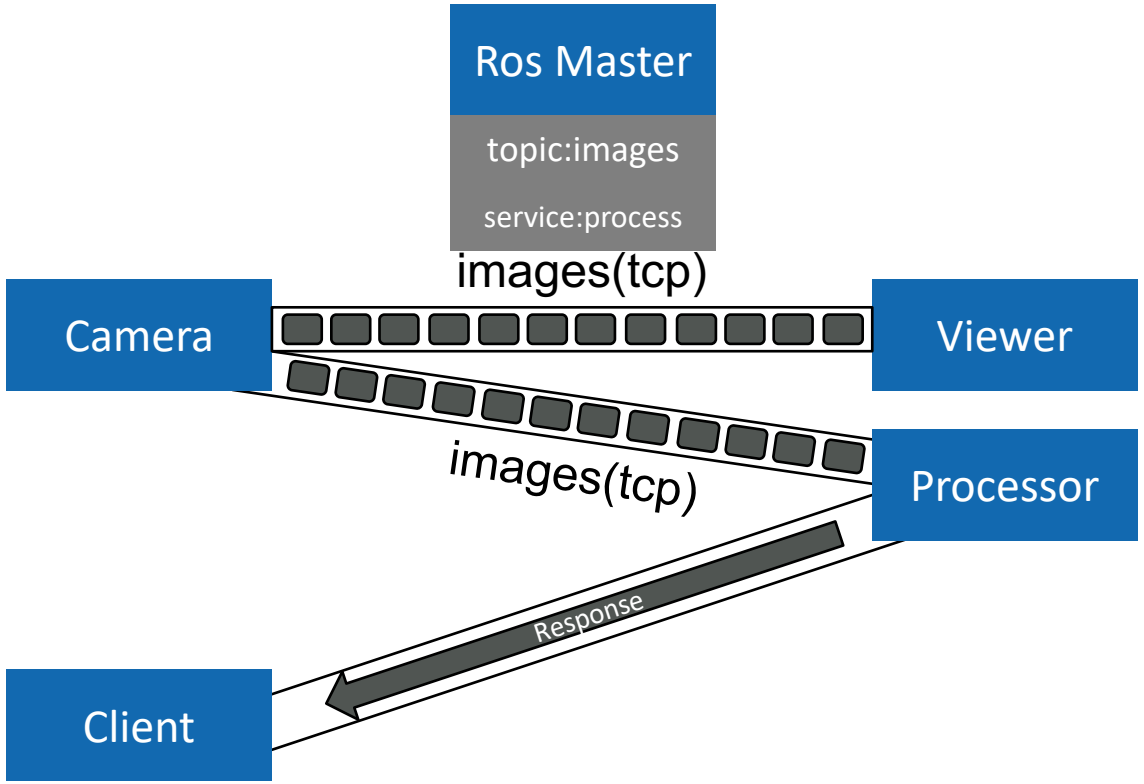
ROS Services Example



ROS Services Example



ROS Services Example



ROS Service Example

Institute for
Software & Systems
Engineering

std_srvs/Trigger.srv

```
---
bool success
string message
```

Request

Response

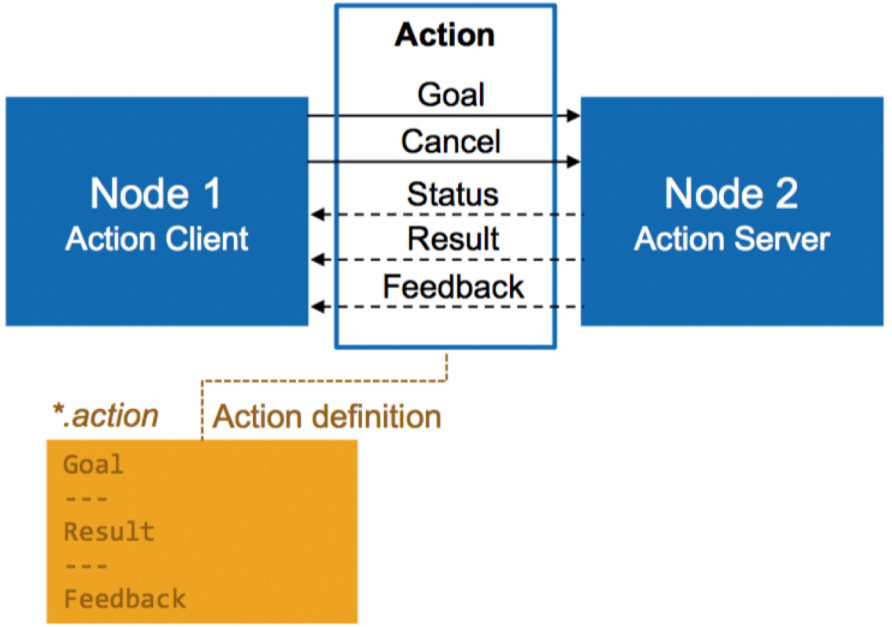
nav_msgs/GetPlan.srv

```
geometry_msgs/PoseStamped start
geometry_msgs/PoseStamped goal
float32 tolerance
---
nav_msgs/Path plan
```

ROS Actions (actionlib)

Institute for Software & Systems Engineering

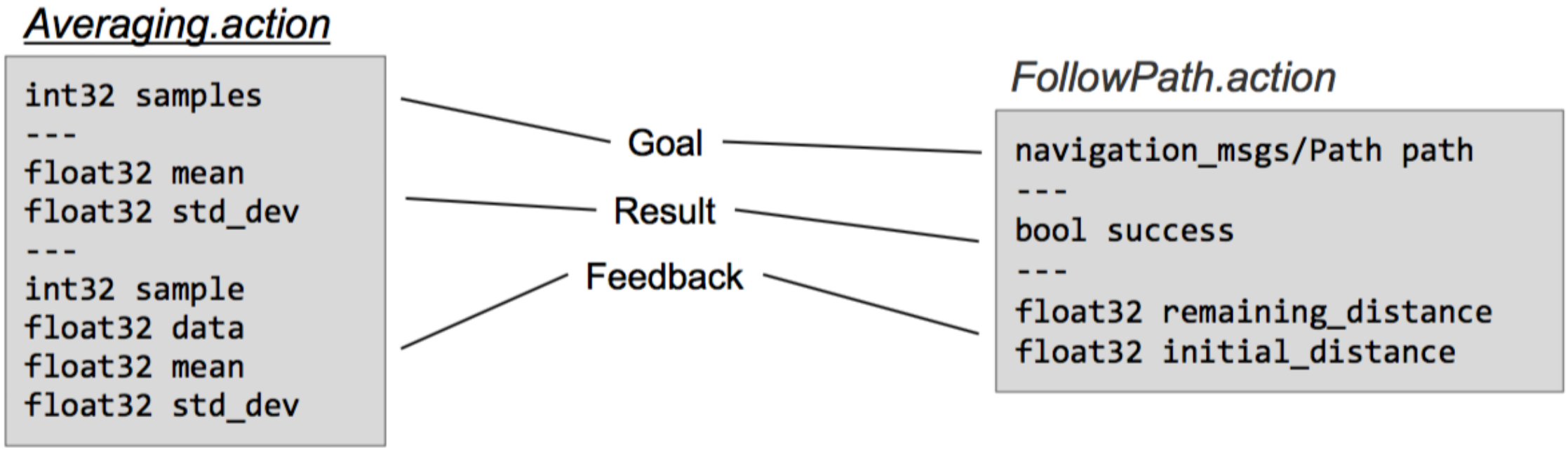
- Similar to service calls, but provide possibility to
  - Cancel the task (preempt)
  - Receive feedback on the progress
- Best way to implement interfaces to time- extended, goal-oriented behaviors
- Similar in structure to services, action are defined in **.action* files
- Internally, actions are implemented with a set of topics



More info
http://wiki.ros.org/actionlib
http://wiki.ros.org/actionlib/DetailedDescription

ROS Actions Example

Institute for
Software & Systems
Engineering



ROS Parameter Server

Institute for Software & Systems Engineering

- Nodes use the *parameter server* to store and retrieve parameters at runtime
- Best used for static data such as configuration parameters
- Parameters can be defined in launch files or separate *YAML* files

config.yaml

```
camera:
  left:
    name: left_camera
    exposure: 1
  right:
    name: right_camera
    exposure: 1.1
```

package.launch

```
<launch>
  <node name="name" pkg="package" type="node_type">
      <rosparam command="load"
                file="$(find package)/config/config.yaml" />
  </node>
</launch>
```

More info
http://wiki.ros.org/rosparam

Comparison

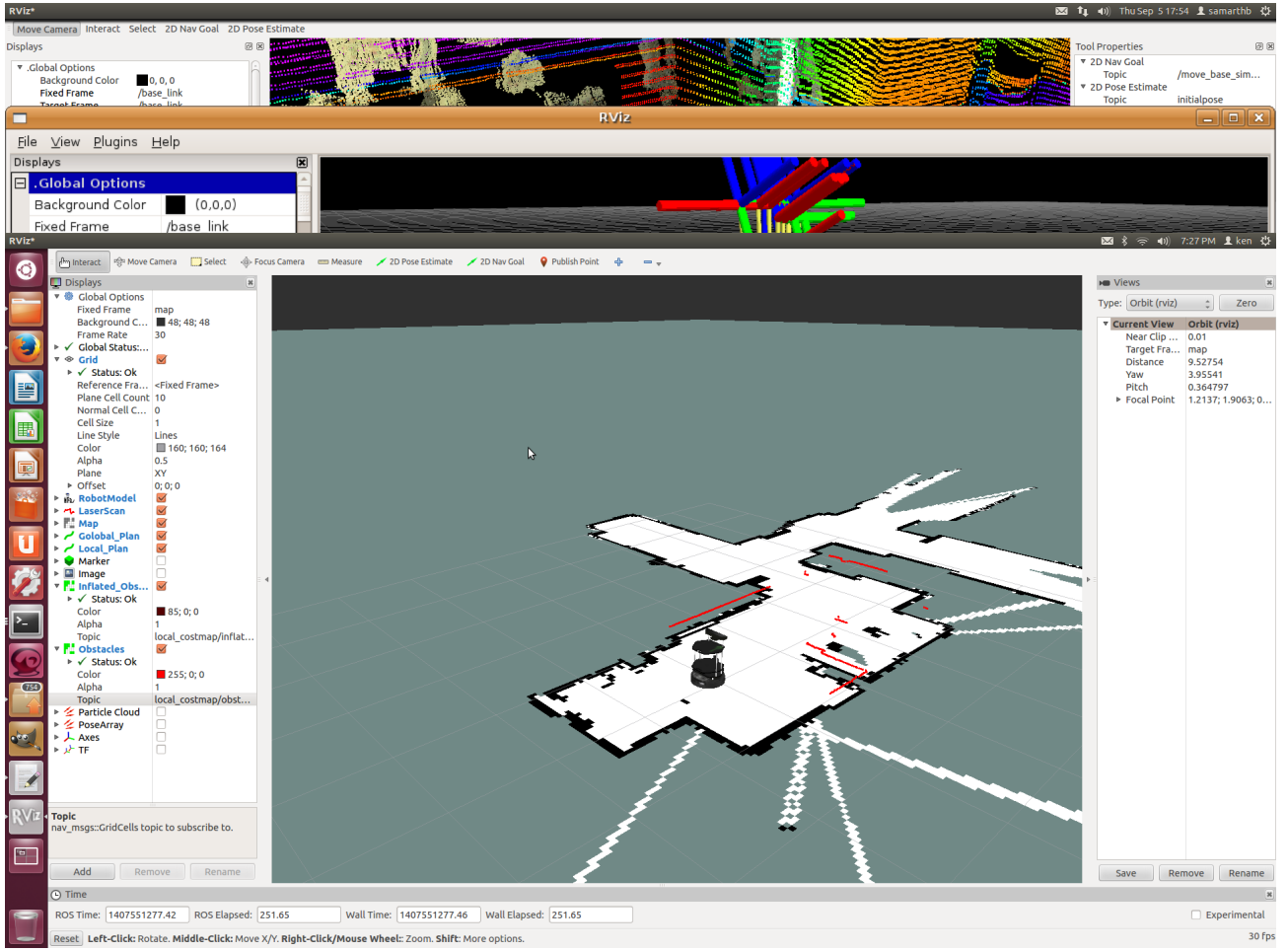
Institute for Software & Systems Engineering

	Parameters	Dynamic Reconfigure	Topics	Services	Actions
Description	Global constant parameters	Local, changeable parameters	Continuous data streams	Blocking call for processing a request	Non-blocking, preemptable goal oriented tasks
Application	Constant settings	Tuning parameters	One-way continuous data flow	Short triggers or calculations	Task executions and robot actions
Examples	Topic names, camera settings, calibration data, robot setup	Controller parameters	Sensor data, robot state	Trigger change, request state, compute quantity	Navigation, grasping, motion execution

More Info
http://wiki.ros.org/dynamic_reconfigure

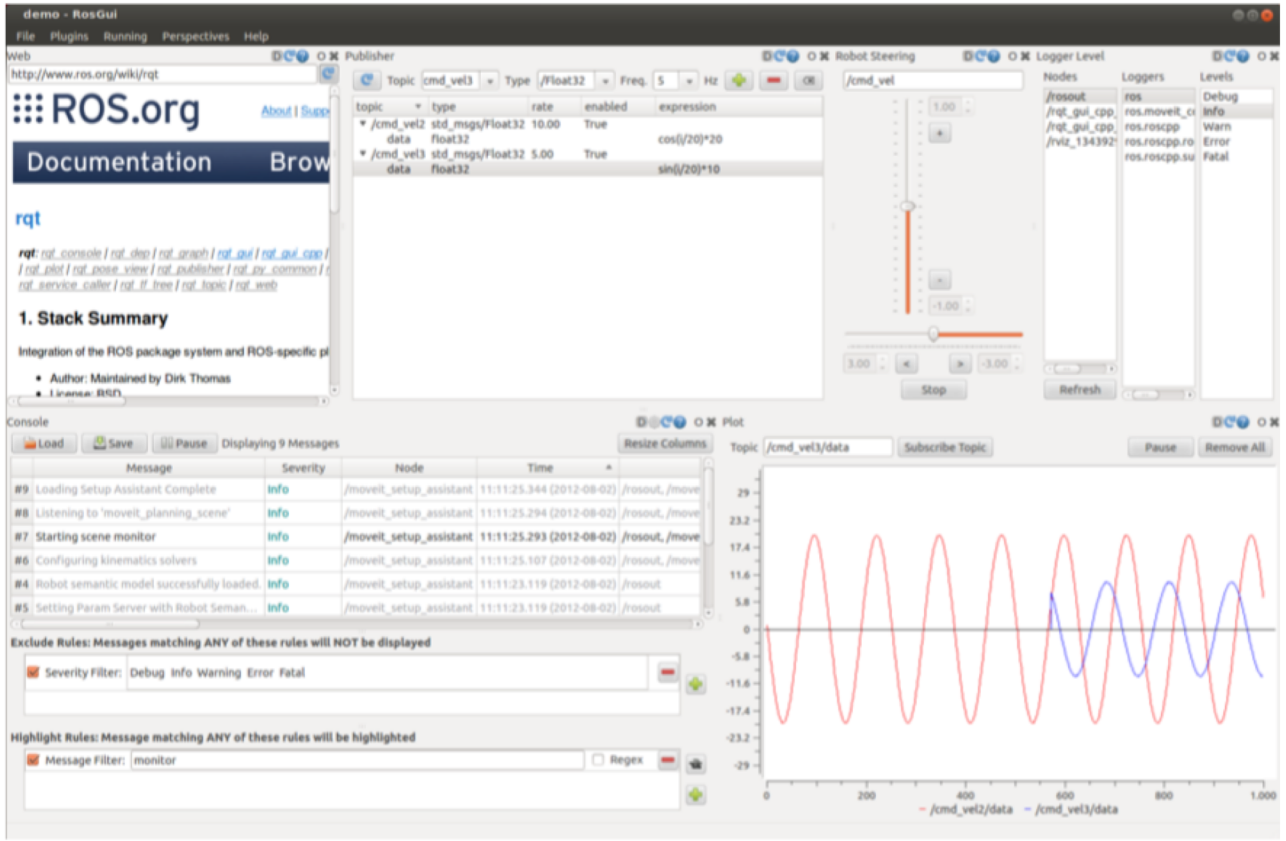
RViz

- 3D visualization tool for ROS
- Subscribes to topics and visualizes the message contents
- Different camera views (orthographic, top- down, etc.)
- Interactive tools to publish user information
- Save and load setup as RViz configuration
- Extensible with plugins



rqt User Interface

- User interface base on Qt
- Custom interfaces can be setup
- Lots of existing plugins
- Simple to write own plugins

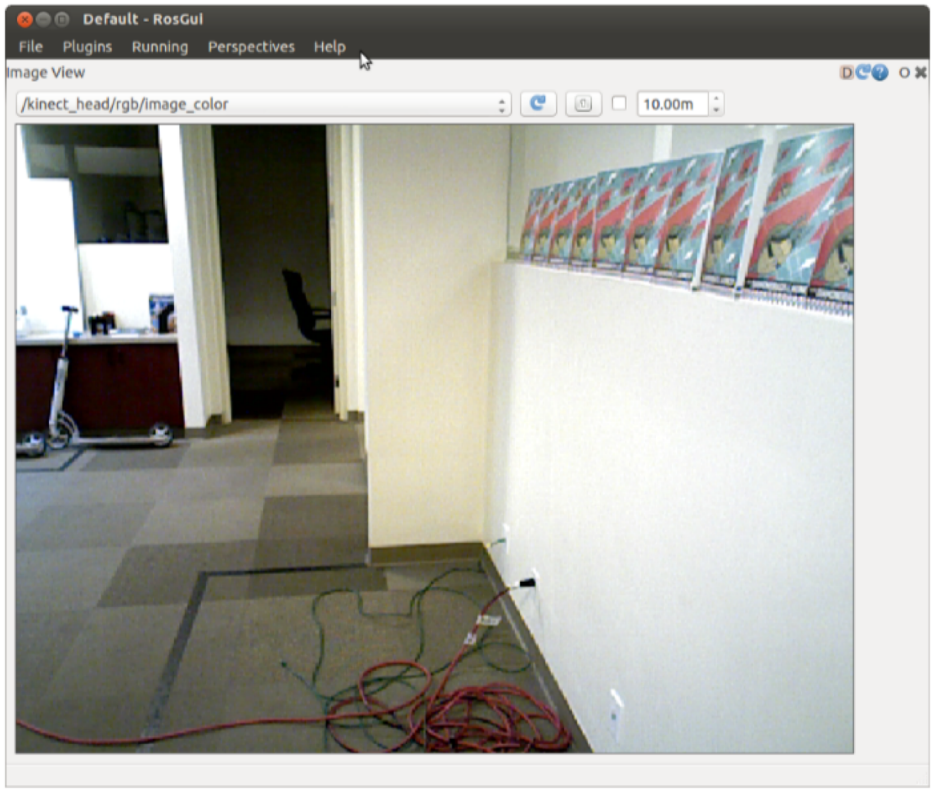


More info
http://wiki.ros.org/rqt/Plugins

rqt User Interface

Institute for Software & Systems Engineering

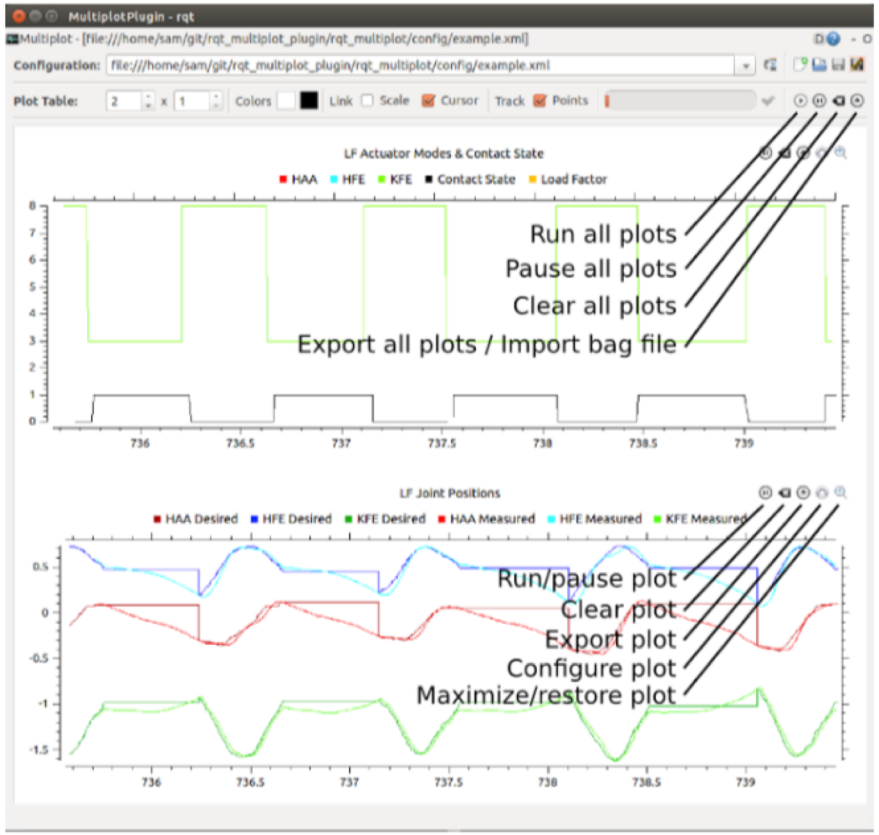
- User interface base on Qt
- Custom interfaces can be setup
- Lots of existing plugins exist
- Simple to write own plugins
- **rqt_image_view**
  Visualizing images



More info
http://wiki.ros.org/rqt_image_view

rqt User Interface

- User interface base on Qt
- Custom interfaces can be setup
- Lots of existing plugins exist
- Simple to write own plugins
- **rqt_multiplot**
  Visualizing numeric values in 2D plots

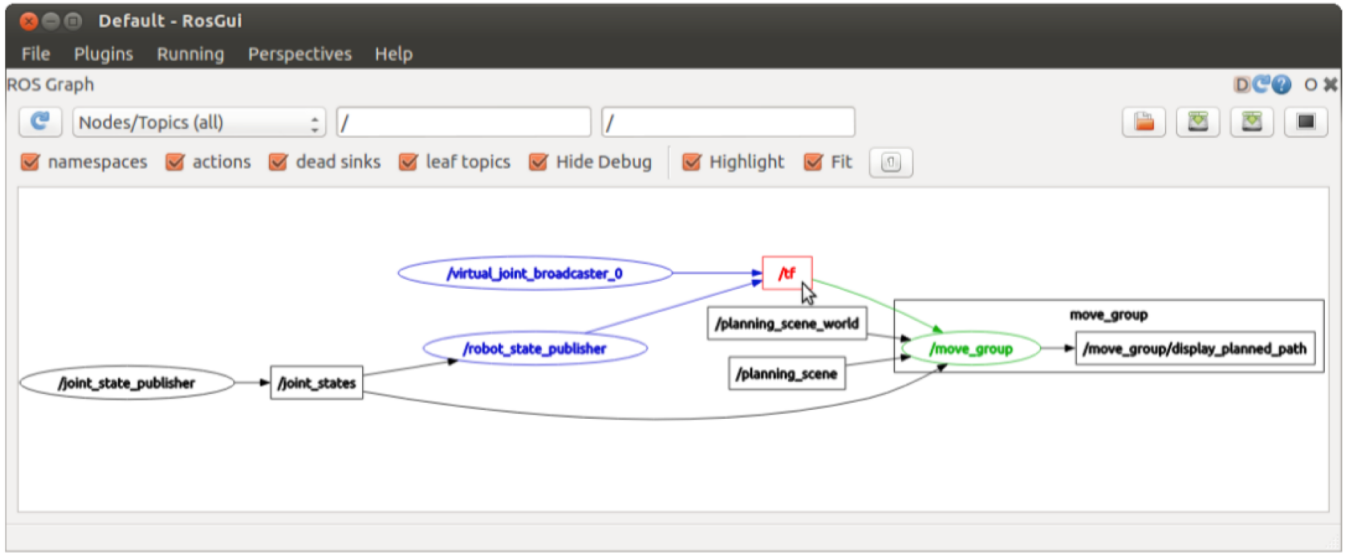


More info
http://wiki.ros.org/rqt_multiplot

rqt User Interface

Institute for Software & Systems Engineering

- User interface base on Qt
- Custom interfaces can be setup
- Lots of existing plugins exist
- Simple to write own plugins
- **rqt_graph**
  Visualizing the ROS computation graph



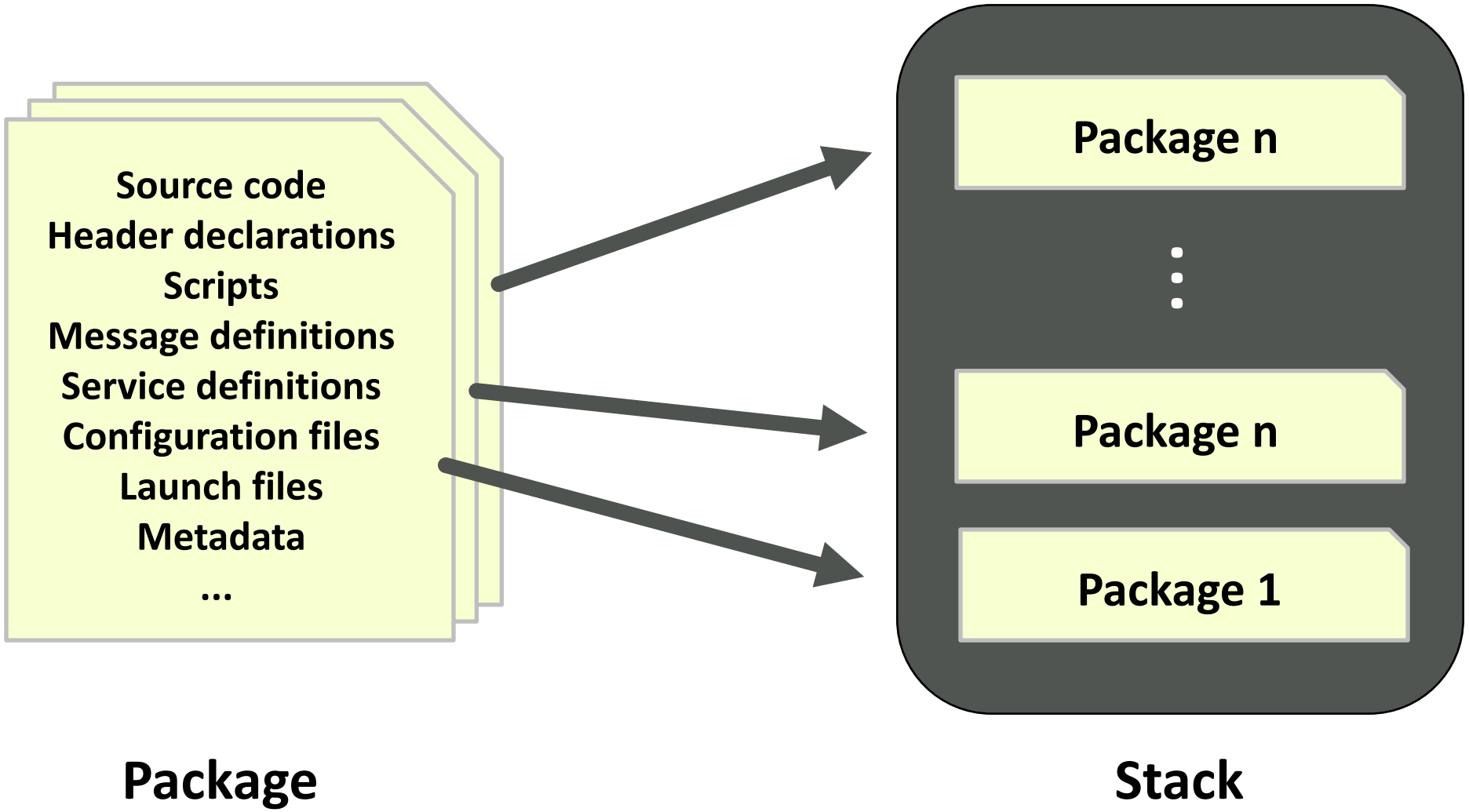
More info
http://wiki.ros.org/rqt_graph

ROS Stacks & Packages

Institute for Software & Systems Engineering

- ROS code is grouped at two different levels:
  - *Packages*
    A named collection of software that is built and treated as an atomic dependency in the ROS build system.
  - *Stacks*
    A named collection of packages for distribution.

ROS Stacks & Packages



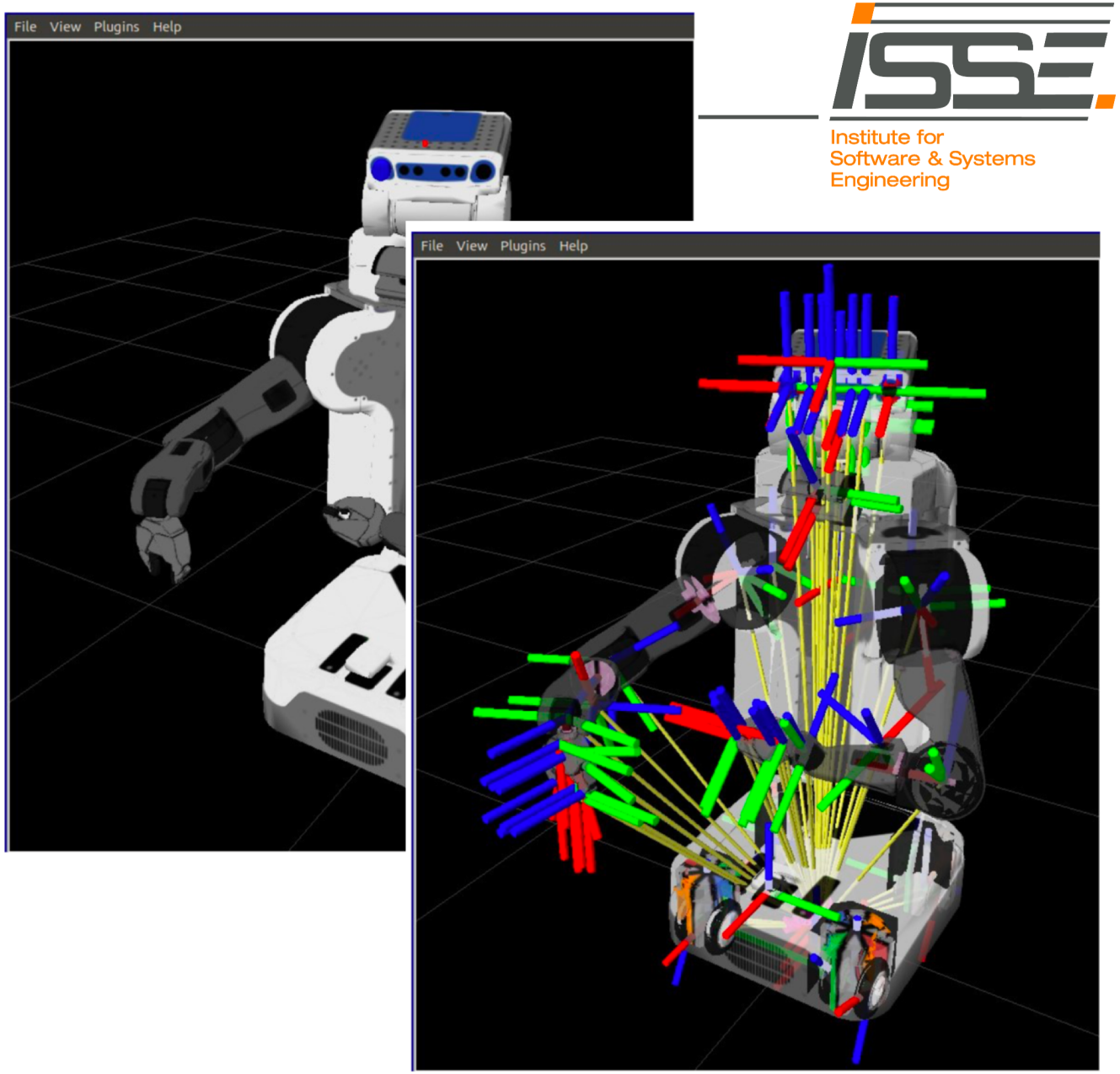
Outlook: ROS Tutorials

Institute for Software & Systems Engineering

- In the exercises we learn ...
  - how to **install & start** ROS
  - to know **ROS console tools**
  - to know the **catkin build system**
  - to create **own messages & topics**
  - to create and run **own nodes**
  - to create and run **own services**
  - to know **graphical ROS tools**

Coordinate frames in ROS

- Robots consist of many *links*
- Every *link* describes its own *coordinate system*
- Sensor measurements are local to the corresponding link
- Links change their position over time



Transforms in ROS

Institute for Software & Systems Engineering

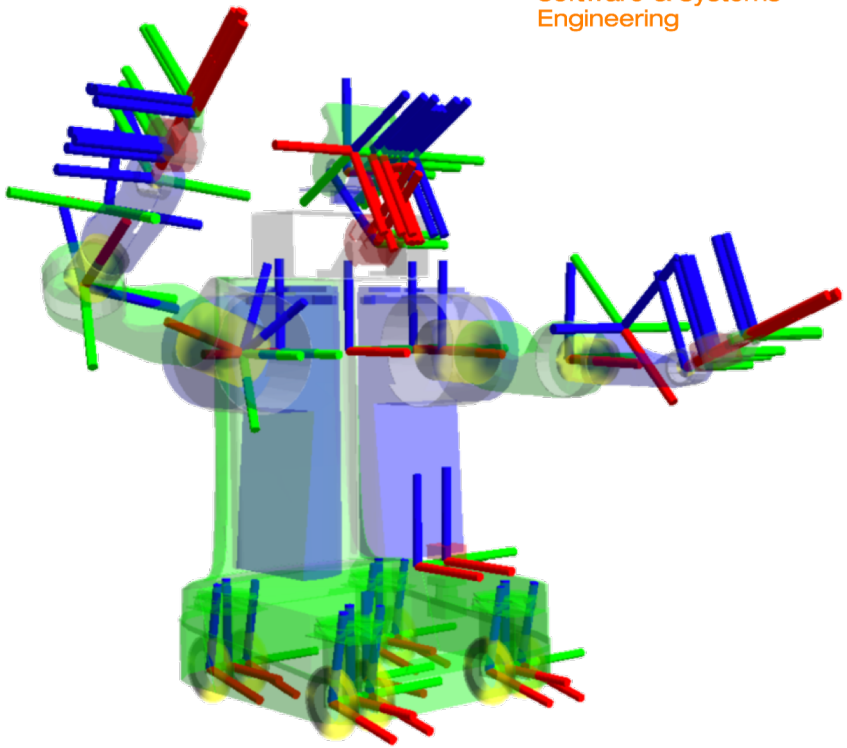
- Transforms are produced by different nodes:
  - Localization in map (AMCL, gmapping)
  - Odometry (base controller)
  - Joint motions (robot controllers and robot state publisher)

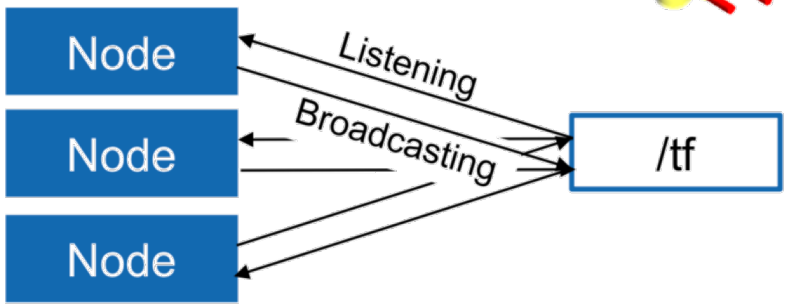
- tf2 is a decentralized coordinate frame tracking system:
  - many publishers, many subscribers
  - Standardized protocol for publishing transforms

tf2

Institute for
Software & Systems
Engineering

- Maintains relationship between coordinate frames in a tree structure buffered in time
- Lets the user transform points, vectors, etc. between coordinate frames at desired time
- Implemented as publish/subscriber model on the topics `/tf` and `/tf_static`





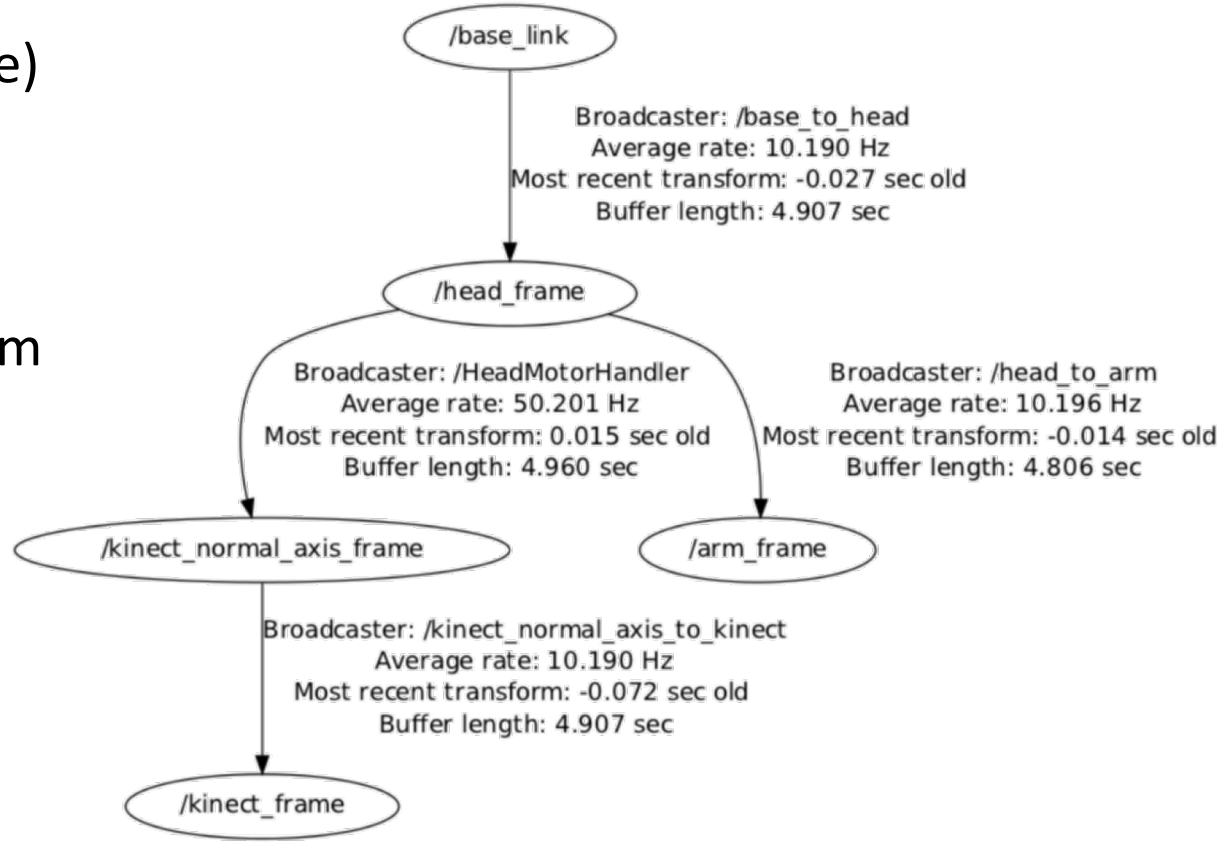
More info
http://wiki.ros.org/tf2

tf2 Transform Tree

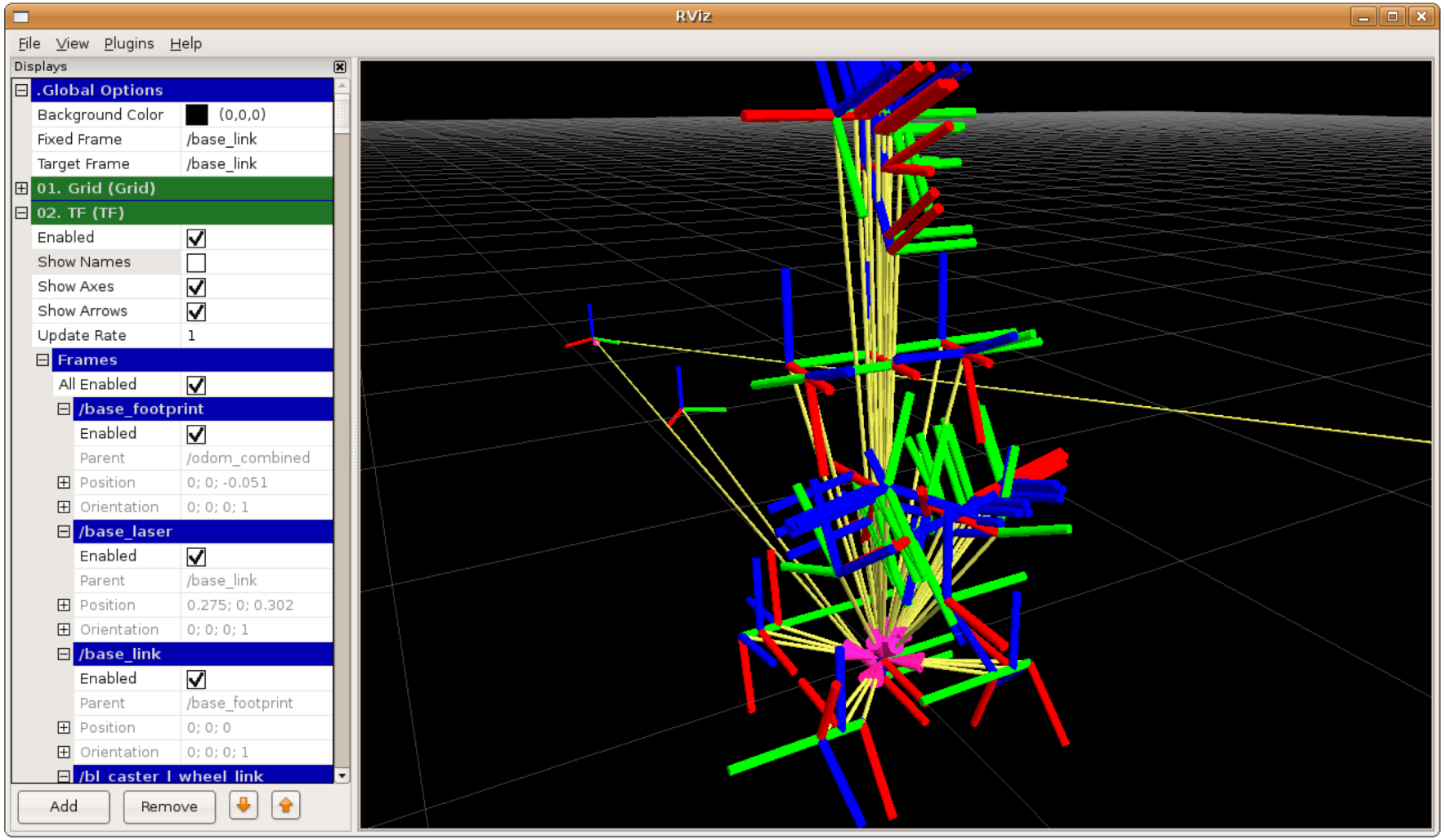
- Consists of frames (links) and the transforms between them.
- Each link is cached (10 secs default caching time)
- TF listeners use a buffer to listen to all broadcasted transforms
- Query for specific transforms from the transform tree

tf2_msgs/TFMessage.msg

```
geometry_msgs/TransformStamped[] transforms
  std_msgs/Header header
    uint32 seqtime stamp
    string frame_id
  string child_frame_id
  geometry_msgs/Transform transform
    geometry_msgs/Vector3 translation
    geometry_msgs/Quaternion rotation
```



tf2 Rviz Plugin



tf2 Data Types

Institute for Software & Systems Engineering

- `Transform` (between frames)
  - `Vector3` for translation
  - `Quaternion` for rotation
- `Pose` (w.r.t. a frame)
  - `Point` for position
  - `Quaternion` for orientation
- Stamped data types (via ROS header)
  - `TransformStamped`, `PoseStamped`, `PointStamped`, …
  - Header contains time stamp and parent frame name
  - `TransformStamped` includes child frame name

Publishing transforms

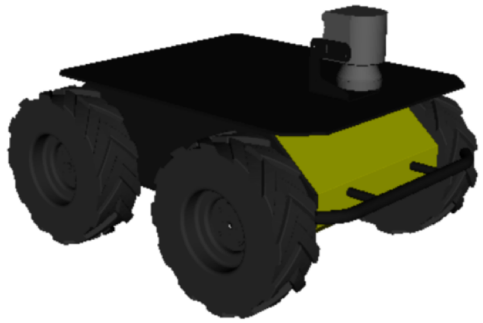
Institute for Software & Systems Engineering

- Nodes that publish transforms
- Static transform publisher (command-line tool)
- Launch files
- URDF, joint states and robot state publisher
  - Make a robot description file (URDF) and load it on the parameter server
  - Implement a node that reads joint states and publishes them
  - Run the robot state publisher node

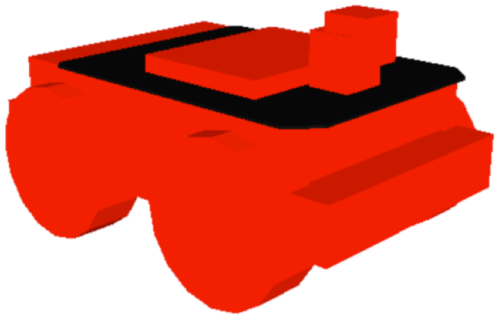
Unified Robot Description Format (URDF)

Institute for Software & Systems Engineering

- Defines an XML format for representing a robot model
  - Kinematic and dynamic description
  - Visual representation
  - Collision model
- URDF generation can be be scripted with *XACRO (XML Macro)*



Mesh for visuals



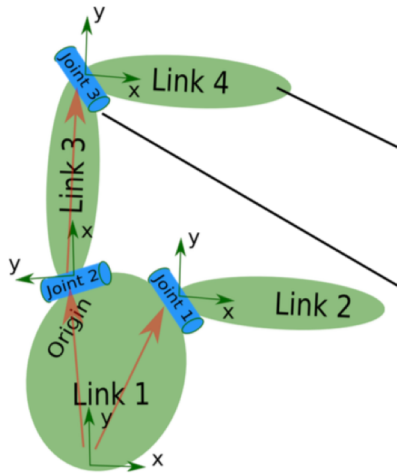
Primitives for collision

More Info
http://wiki.ros.org/xacro

Unified Robot Description Format (URDF)

Institute for Software & Systems Engineering

- Description consists of a set of *link* elements and a set of *joint* elements
- Joints connect the links together



robot.urdf

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

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

```
<link name="link_name">
  <visual>
    <geometry>
      <mesh filename="mesh.dae"/>
    </geometry>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
  </collision>
  <inertial>
    <mass value="10"/>
    <inertia ixx="0.4" ixy="0.0" …/>
  </inertial>
</link>
```

```
<joint name="joint_name" type="revolute">
  <axis xyz="0 0 1"/>
  <limit effort="1000.0" upper="0.548" … />
  <origin rpy="0 0 0" xyz="0.2 0.01 0"/>
  <parent link="parent_link_name"/>
  <child link="child_link_name"/>
</joint>
```

More info
http://wiki.ros.org/urdf/XML/model

URDF in ROS

Institute for Software & Systems Engineering

- URDF can be loaded using a launch file
- The robot description (URDF) is stored on the parameter server (typically) under `/robot_description`
- You can visualize the robot model in Rviz with the RobotModel plugin
- You can visualize the robot model in Rviz with the RobotModel plugin

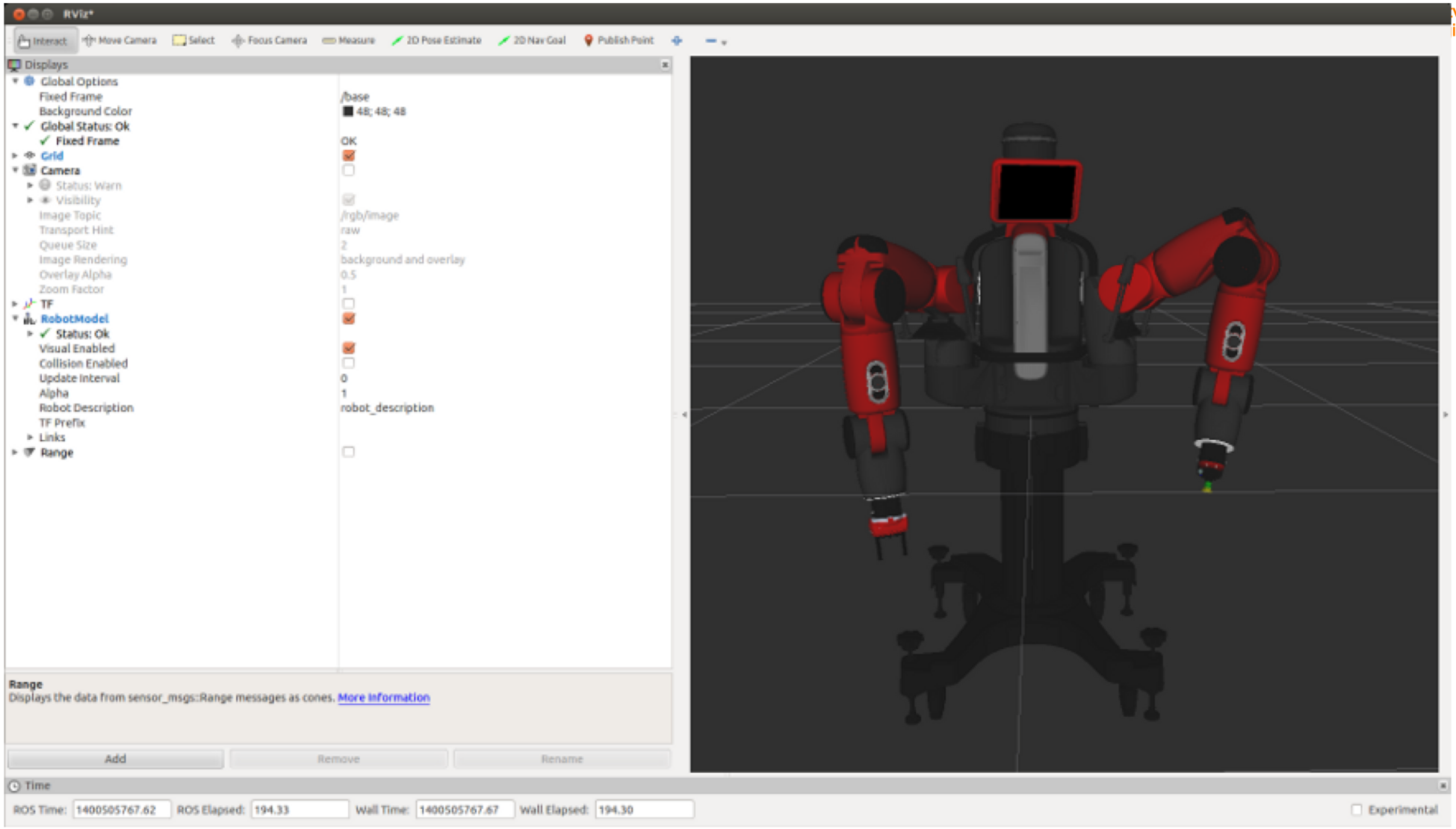
husky_empty_world.launch

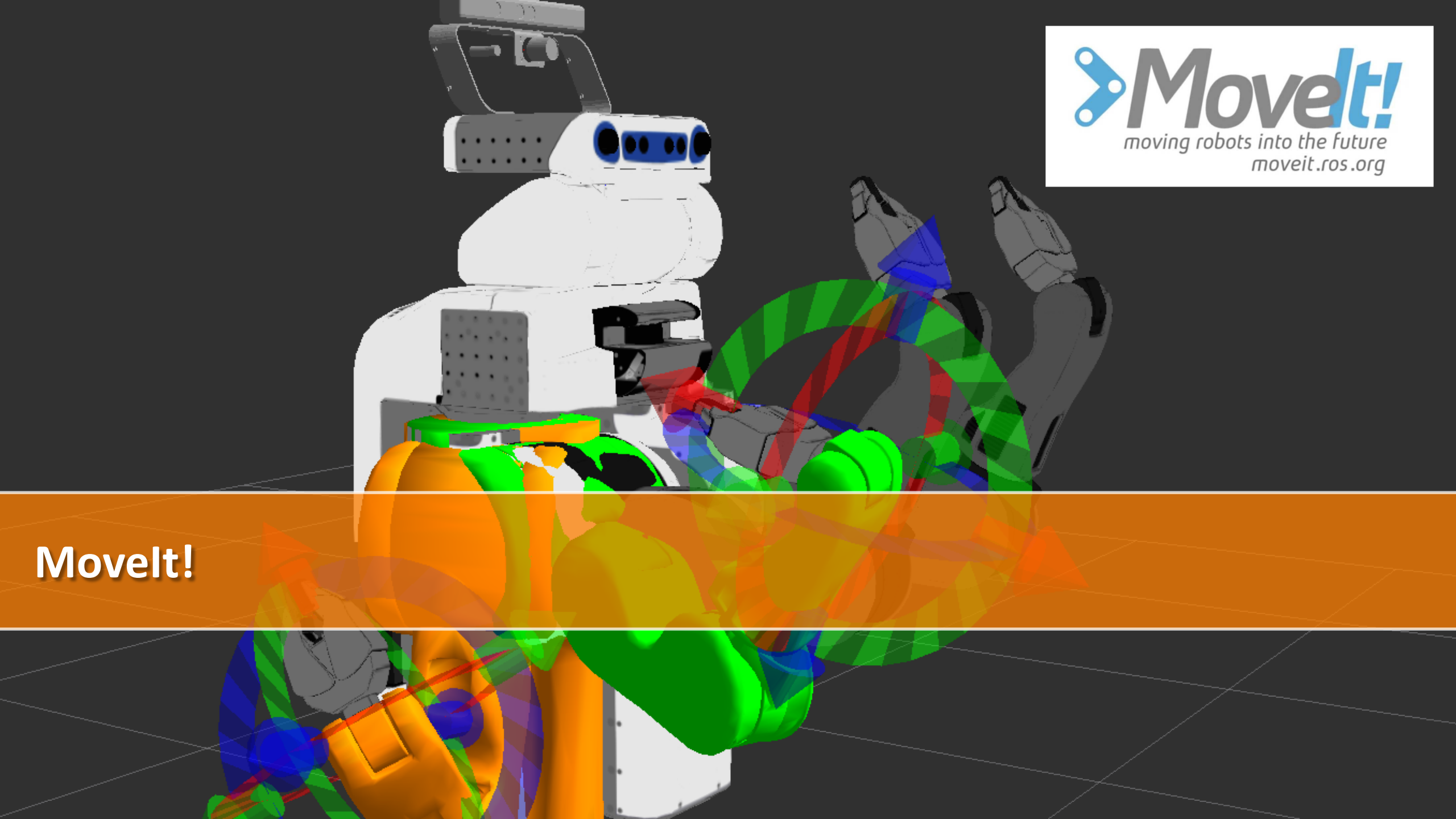
```
…
<include file="$(find husky_gazebo)/launch/spawn_husky.launch">
  <arg name="laser_enabled" value="$(arg laser_enabled)"/>
  <arg name="ur5_enabled" value="$(arg ur5_enabled)"/>
  <arg name="kinect_enabled" value="$(arg kinect_enabled)"/>
</include>
…
```

spawn_husky.launch

```
…
<param name="robot_description" command="$(find xacro)/xacro.py
'$(arg husky_gazebo_description)'
   laser_enabled:=$(arg laser_enabled)
   ur5_enabled:=$(arg ur5_enabled)
   kinect_enabled:=$(arg kinect_enabled)" />
…
```

URDF in ROS


RViz*
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Global Options
Fixed Frame
/base
Background Color
48; 48; 48
Global Status: Ok
Fixed Frame
OK
Grid
Camera
Status: Warn
Visibility
Image Topic
/rgb/image
Transport Hint
raw
Queue Size
2
Image Rendering
background and overlay
Overlay Alpha
0.5
Zoom Factor
1
TF
RobotModel
Status: Ok
Visual Enabled
Collision Enabled
Update Interval
0
Alpha
1
Robot Description
robot_description
TF Prefix
Links
Range
Range
Displays the data from sensor_msgs::Range messages as cones. More Information
Add
Remove
Rename
Time
ROS Time: 1400505767.62
ROS Elapsed: 194.33
Wall Time: 1400505767.67
Wall Elapsed: 194.30
Experimental



MoveIt!

MoveIt! in a nutshell

Institute for Software & Systems Engineering

- Technical Capabilities
  - Motion Planning with Kinematic Constraints
  - Fast and flexible collision checking
  - Integrated Kinematics
  - Integrated Perception for Environment Representation
  - Standardised Interfaces to Robot Controllers
  - Execution and Monitoring of Robot Trajectories

- MoveIt! includes a variety of motion planners:
  - Sampling-based motion planners
  - Search-based motion planners
  - Optimization-based motion planners (CHOMP)

MoveIt! in a nutshell

Institute for Software & Systems Engineering

MoveIt! – Constraints & Collision Detection

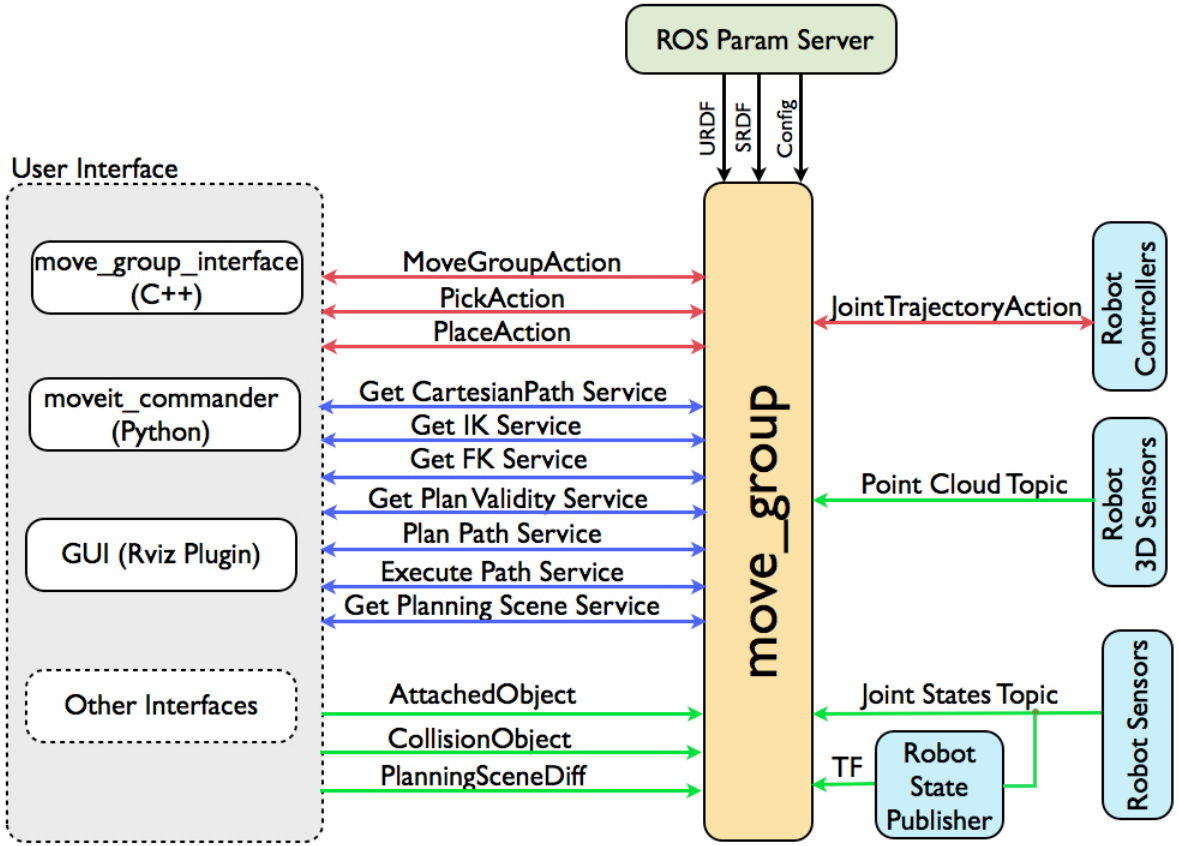
Institute for Software & Systems Engineering

- MoveIt! allows to specify the following kinematic constraints:
  - **Position constraints**: Restrict the position of a link to be within a region of space
  - **Orientation constraints**: Restrict the orientation of a link to be within specified RPY limits
  - **Visibility constraints**: Restrict a point on a link to be within the visibility cone for a sensor
  - **Joint constraints**: Restrict a joint to be between two values
  - **User-specified constraints**: specify your own constraints with a user-defined callback.

- MoveIt! uses Flexible Collision Library (FCL) which supports:
  - Meshes
  - Primitive shapes (boxes, cylinders, cones)
  - Octomap (3D sensor information)

MoveIt! System Architecture

- **User Interface**
  - **C++**: using the `move_group_interface` package that provides an interface to `move_group`
  - **Python**: using the moveit_commander package
  - **GUI**: using the Motion Planning plugin to Rviz
- **Configuration**
  - **URDF**: *`move_group`* looks for *`robot_description`* on the ROS param server to get the URDF.
  - **SRDF** : *`move_group`* looks for the *`robot_description_semantic`* on the ROS param server to get the SRDF.
  - **MoveIt! Configuration**: *`move_group`* will look on the ROS param server for other configuration including joint limits, kinematics, motion planning, perception and other information.
- SRDF and config files are automatically generated by the MoveIt! setup assistant.

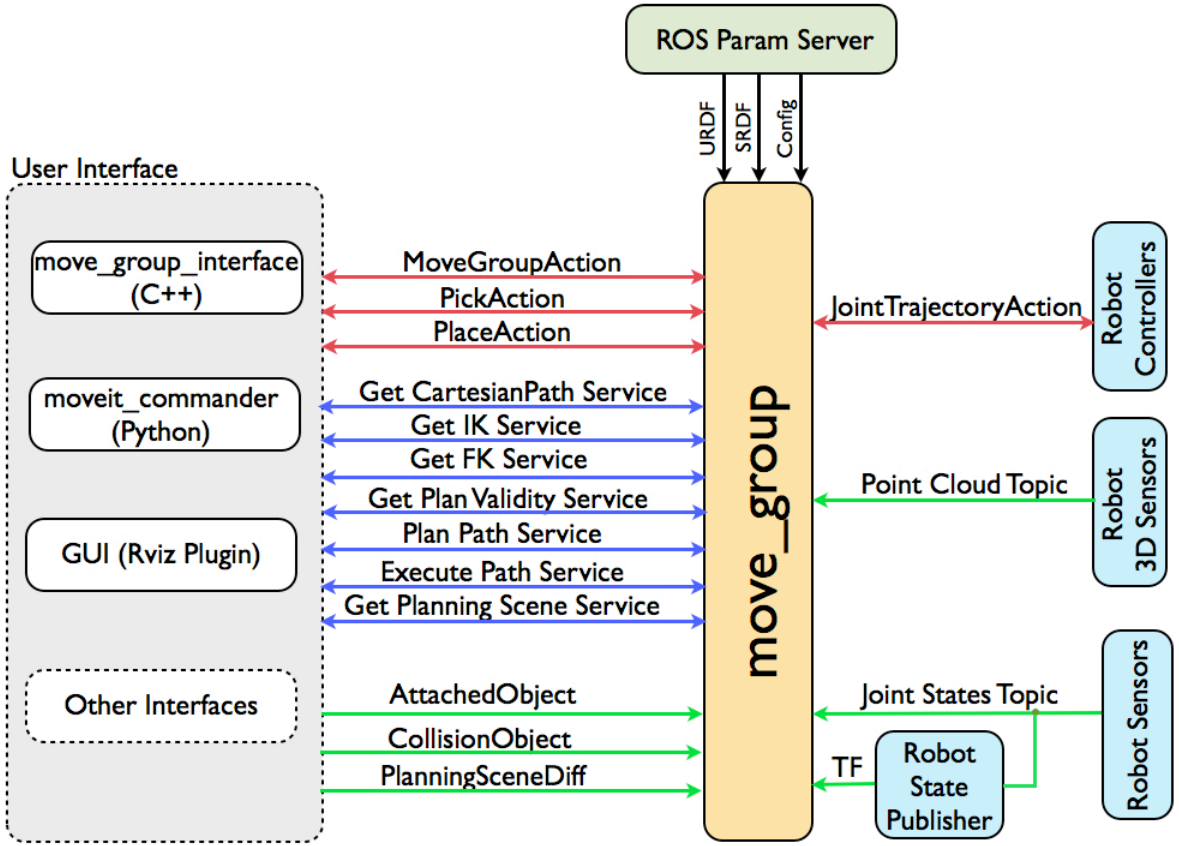


MoveIt! System Architecture

Institute for Software & Systems Engineering

move_group communicates via topics and actions with the robot to get current state information (joints positions, etc.), to get point clouds and other sensor data and to control the robot.

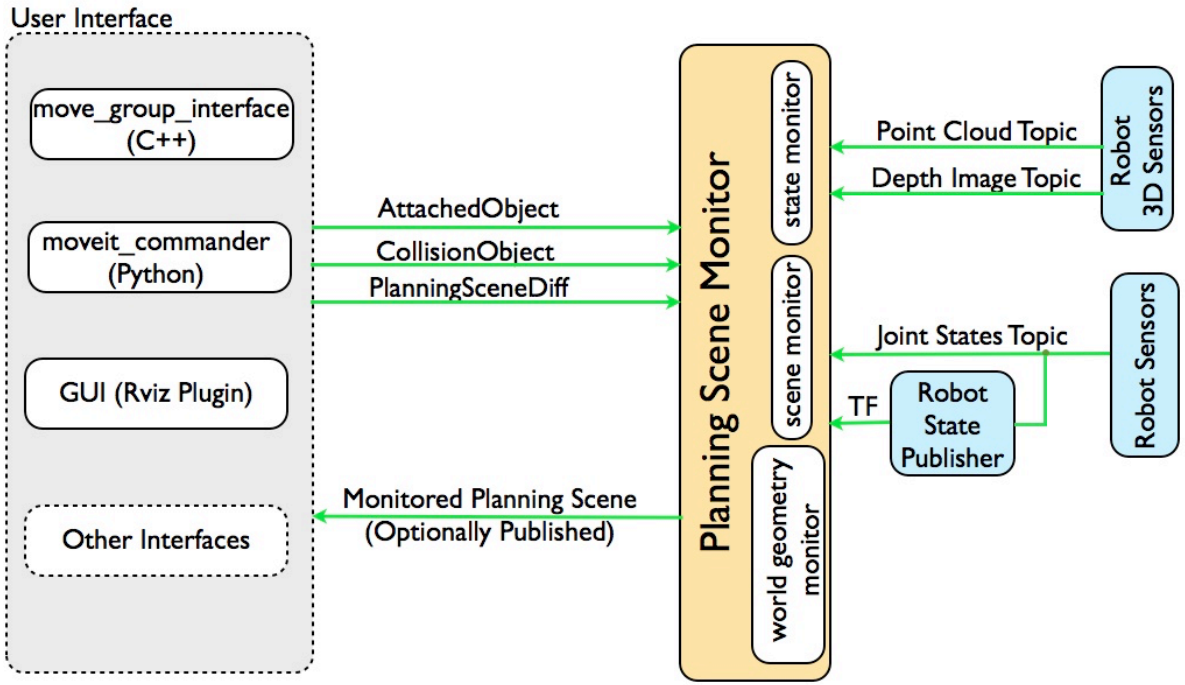
- **Joint State Information**: *move_group* listens on the */joint_states* topic for determining the current state information - i.e. determining where each joint of the robot is.
- **Transform Information**: *move_group* monitors transform information using the ROS TF library. This allows the node to get global information about the robot's pose (among other things).
- **Controller Interface**: *move_group* talks to the controllers on the robot using the standard `FollowJointTrajectoryAction` interface.
- **Planning Scene**: *move_group* uses the Planning Scene Monitor to maintain a *planning scene*, which is a representation of the world and the current state of the robot.



MoveIt! System Architecture

Institute for Software & Systems Engineering

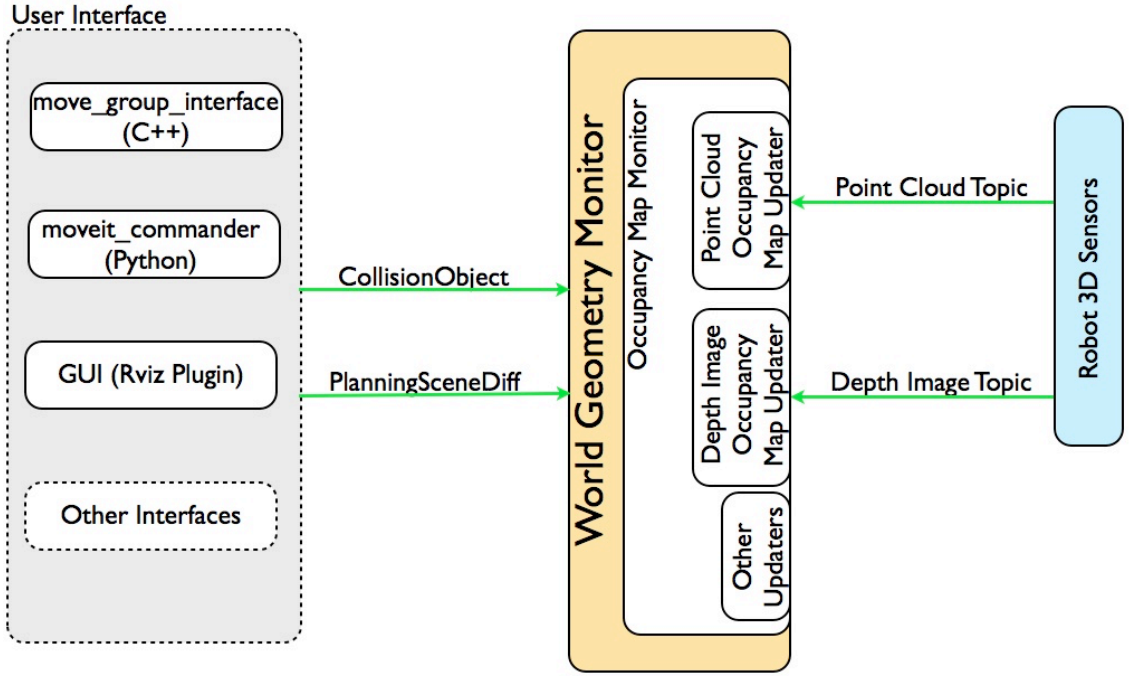
- A *planning scene* is used to represent the world around the robot and also stores the state of the robot itself.
- It is maintained by the **Planning Scene Monitor** inside the `move_group` node.
- The planning scene monitor listens to:
  - **State Information**: on the `joint_states` topic
  - **Sensor Information**: using the *world geometry monitor*
  - **World geometry information**: from user input on the `planning_scene` topic (as a planning scene diff)



MoveIt! System Architecture

Institute for Software & Systems Engineering

- **World Geometry Monitor** builds a 3D representation of the environment around the robot with information on the `planning_scene` topic for adding object information.
- 3D perception is handled by the ***Occupancy Map Monitor*** which uses a plugin architecture to handle different kinds of sensors:
  - **Point clouds** are handled by the *point cloud occupancy map updater* plugin
  - **Depth images** are handled by the *depth image occupancy map updater* plugin
- The Occupancy Map Monitor uses an *Octomap* to maintain the occupancy map of the environment. The *Octomap* can directly be passed into FCL, the collision checking library that MoveIt! uses.
- The **Depth Image Occupancy Map Updater** includes its own *self-filter*, i.e. it will remove visible parts of the robot from the depth map.



MoveIt! Examples

Institute for Software & Systems Engineering