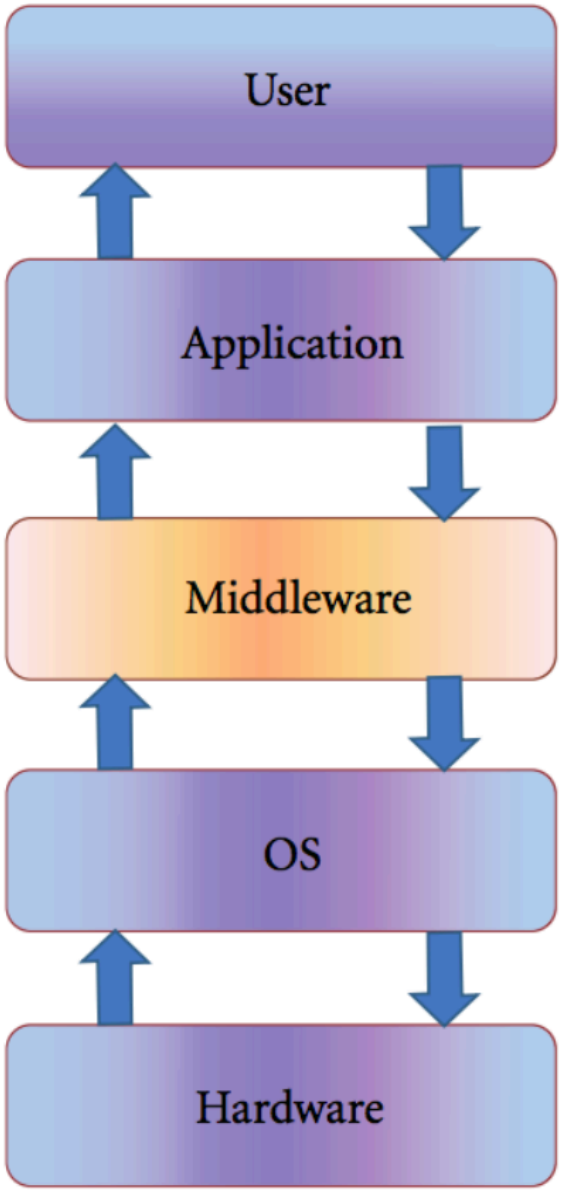
Introduction to Middleware

What is “middleware”?

- **Middleware sits “in the middle”** of software components and facilitates their interaction.
- The purpose is to **provide an abstraction model** for functions such as instantiation, communication, etc.
- Middleware provides the low-level implementation; **you can focus on the business logic.**

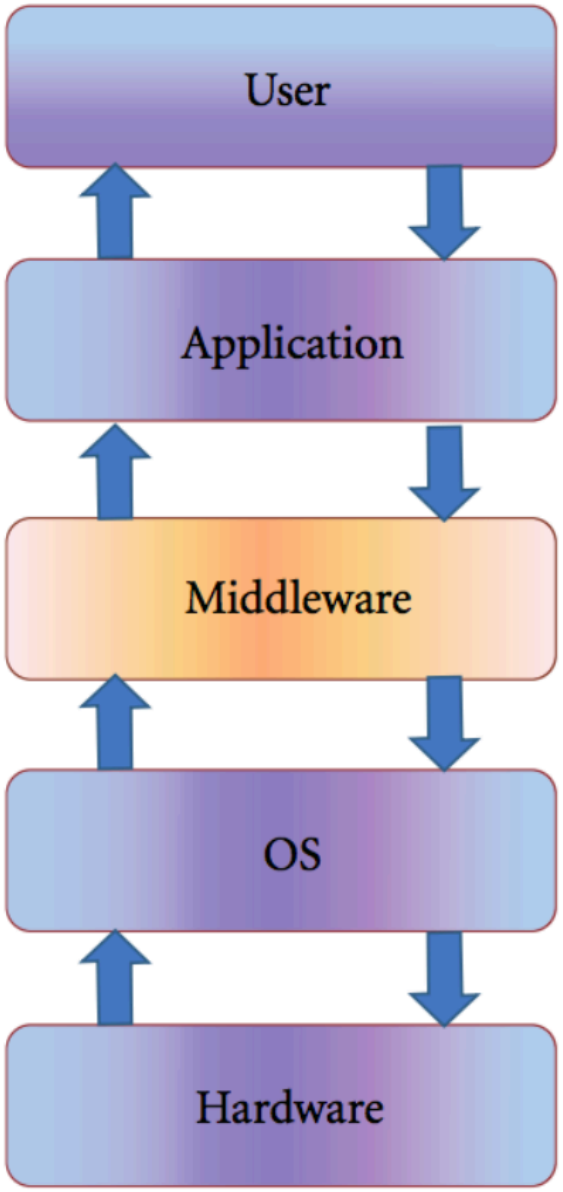


What is robotics "middleware"?

- **Middleware for robotics** also provides specific functionalities for robot development.
    - For example, **message types** specific for robotics.

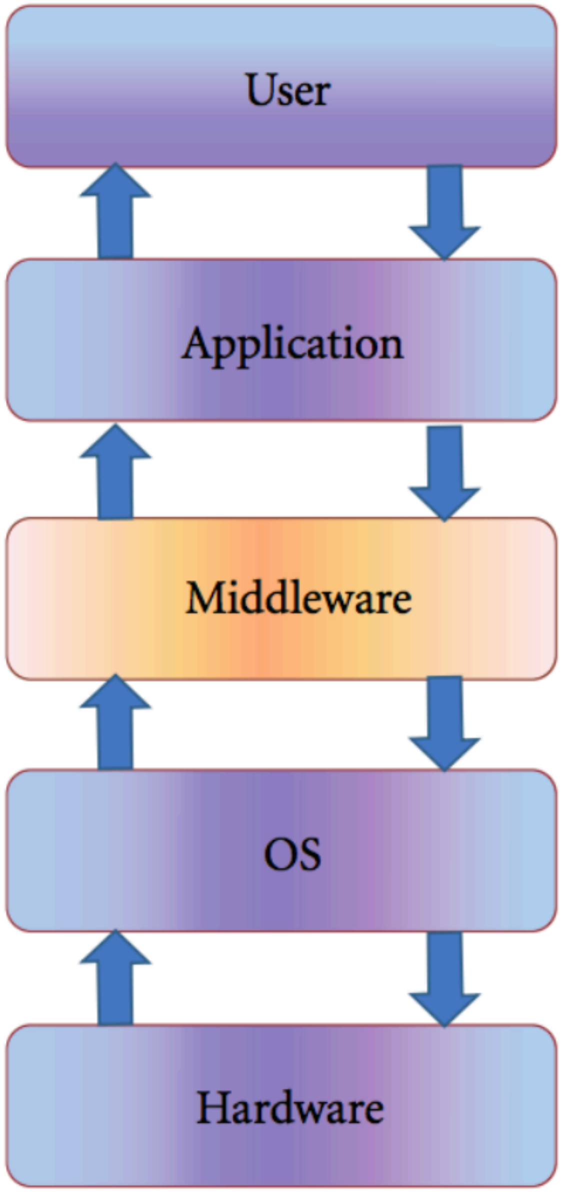
      `Joy, Imu, NavSatFix, PointCloud, LaserScan`

    - **Execution and communication models** that fit the robotics paradigm.



Middleware components

- Every middleware **must** provide:
  - Abstraction from sensors/actuators hardware;
  - Communication protocol for data transport.
- Every middleware **should** have:
  - A tool for taking logs;
  - A tool for playing back logs;
  - Tools for timing analysis (latency/throughput).
  - Simulation tools.



Some popular middleware suites

ROS, LCM, MOOS, JAUS, Orcos, Pyro, Player, Orca, Mira, OpenRTMaist, ASEBA, MARIE, RSCA, MRDS, OPROS, CLARAty, SmartSoft, ERSP, Webots, RoboFrame

From prototype to deployment

- Each middleware is best suited to a **different phase of development.**

learning → prototyping → development → productization

- It is not uncommon to start from a **flexible prototype middleware** and then switch to some **more rigid and performant** deployment solution.
- **Well-designed applications** separate business logic from communication logic.
    - Make “core code” independent of middleware;
      write thin wrapper(s) specific to middleware.

Middleware Comparison Axes

- You can compare middlewares along different dimensions.
  Choose the best one for your use case.

	ROS	LCM	MOOS
Communication Structure	name-/parameter server	decentralized	central database
Communication Mechanism	intra-process, TCP, UDP	UDP multicast	TCP
Data Transport	publisher / subscriber, RPC	publisher / subscriber	store / fetch
Message Types	IDL using PODs	IDL using PODs	string, double
Supported Languages	C++, Java, Python,...	C++ , Java, C#, Python, ...	C++, Java
Supported Platforms	Linux, OS X (partial), and Win (partial)	Linux, Win, OS X	Linux, OS X

- In this course, the choice is made for you.

Introduction to ROS (Robot Operating System)

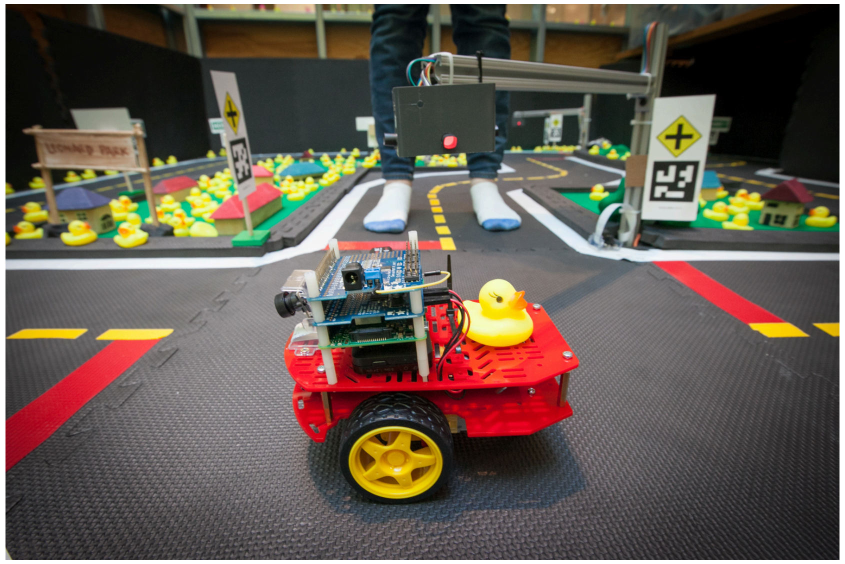
* not an operating system

The History of ROS

- The project began in 2007.
- Funded by National Science Foundation (NSF).
- Later supported by a company called “Willow Garage” (not existing anymore).
- Currently supported by the “Open Source Robotics Foundation” (www.osrfoundation.org)

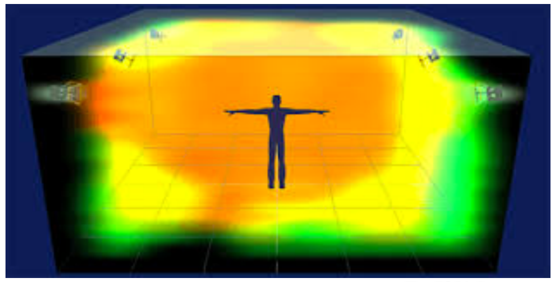
Officially supported "ROS" Research-Education Robots

- Many research/education robots come with ROS drivers support.
    - Easy to get started!



Hardware with Supported ROS Interfaces

- Many sensors for research/development come now with a ROS interface.



ROS Noetic

- For this year we use ROS Noetic.
- This is the latest and last "ROS 1" version.

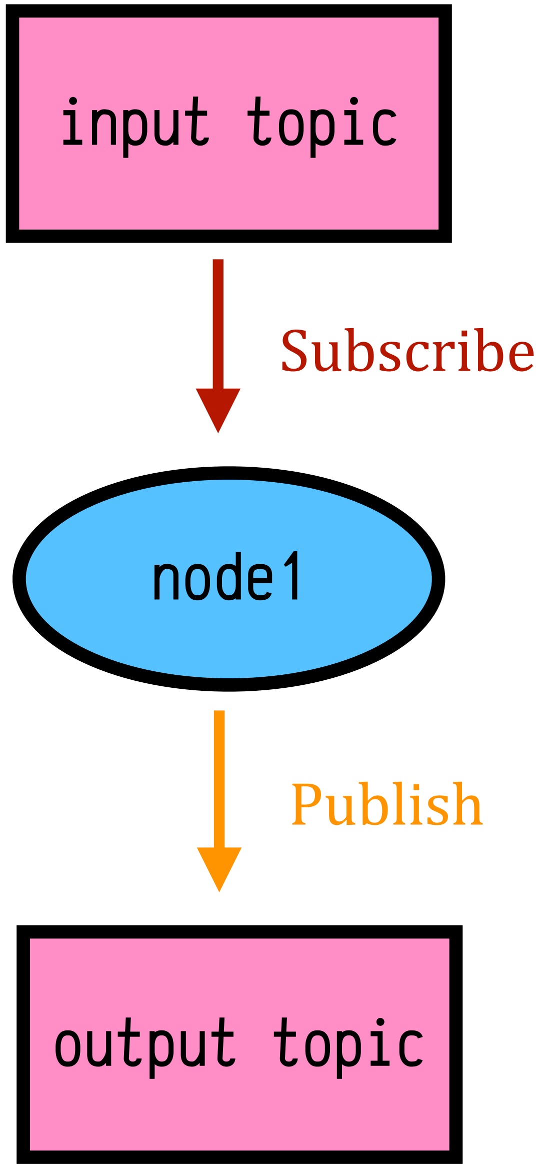
Some ROS vocabulary that we are going to learn

- **Basic concepts:**
  - Nodes
  - Topics
  - Publishing
  - Subscribing
  - The ROS “Master”
  - Messages
- **Intermediate concepts:**
  - Launch files
  - Parameters / parameter server

Nodes

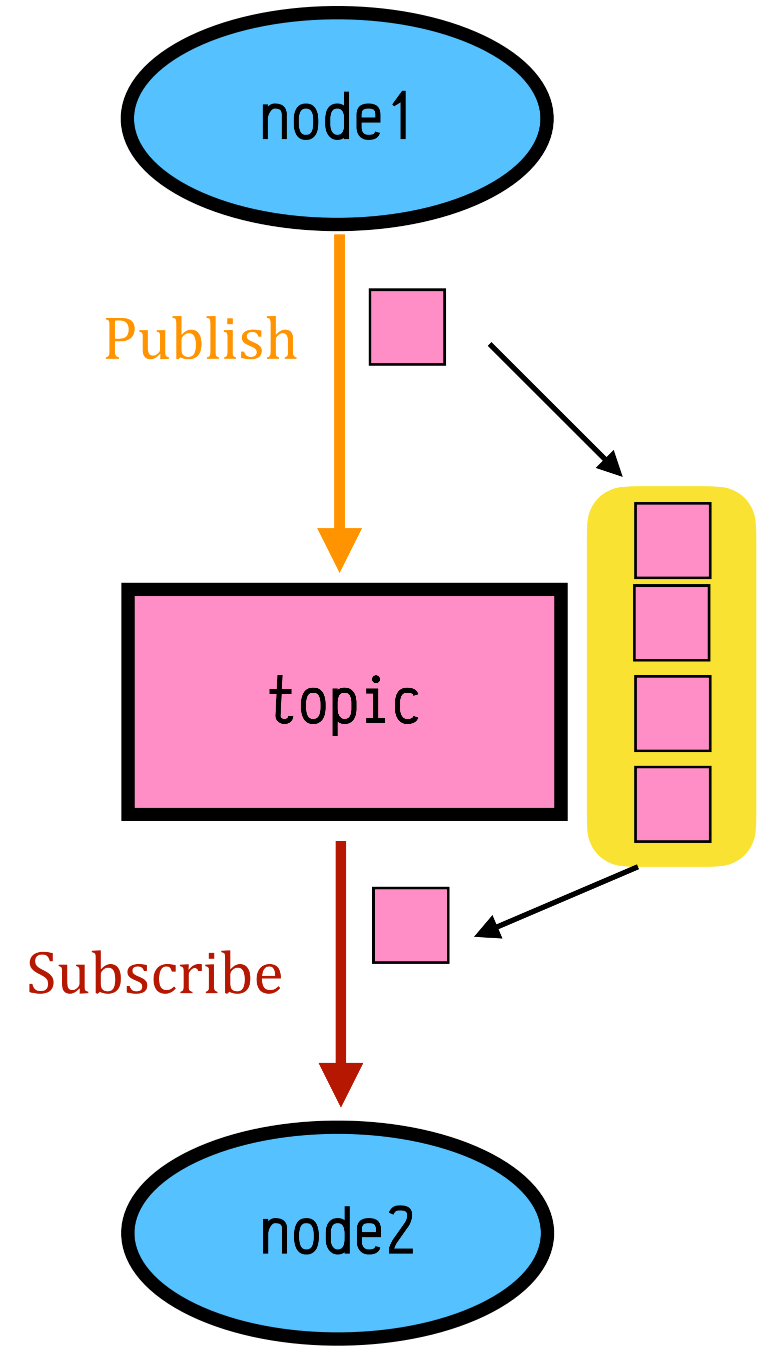
- Nodes are the “executables”.
- ROS handles threading.
  - Nodes can be multi-threaded inside.

- Nodes **subscribe (“read”)** to **topics**.
- Nodes **publish (“write”)** to **topics**.



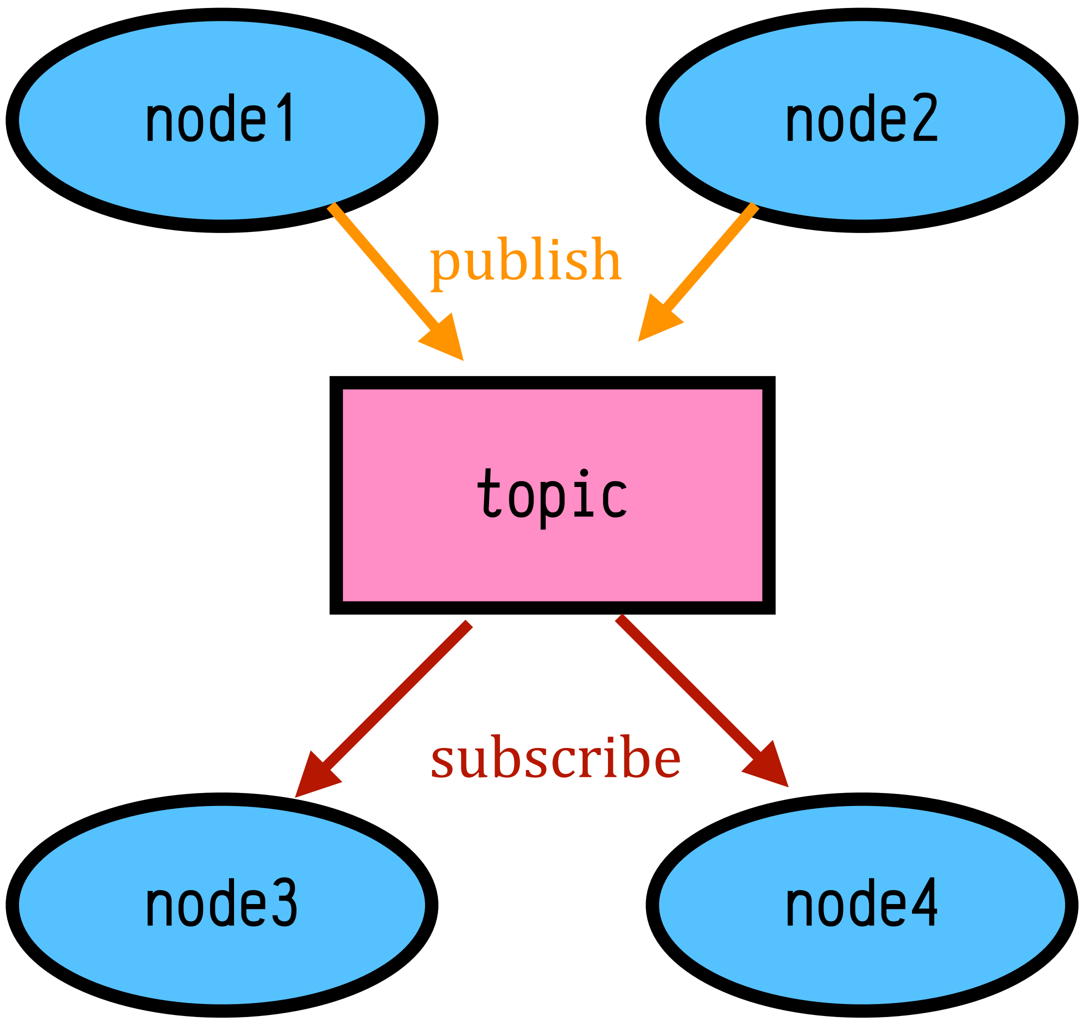
Topics

- **Topics** are used to pass information between **nodes**.
    - there are other ways, but this is the recommended way
- Each **topic** has a "**message type**".
    - e.g. "Image", "Odometry reading".
- Each **topic** maintains **a queue of data** that the publishers append to, and the subscribers read from.
    - We will see that there are different settings for the behavior of the queue.



Multiple-writers and multiple readers

- **Multiple nodes can publish** to a topic.
- **Multiple nodes can subscribe** to a topic.

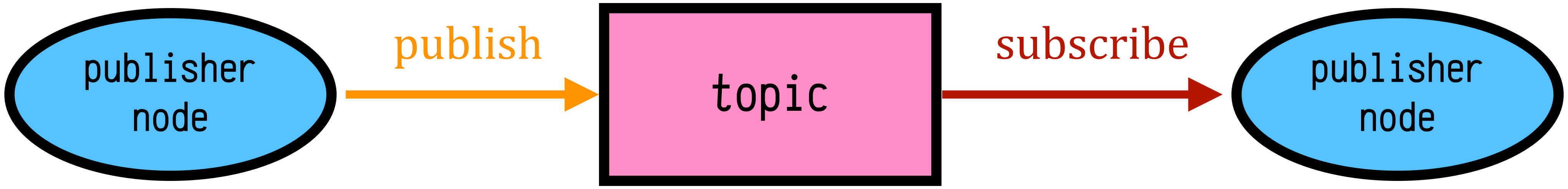


- Not the best way to isolate functionality according to "component-based design", but very low barrier of entry to get something working quickly.

The ROS Master

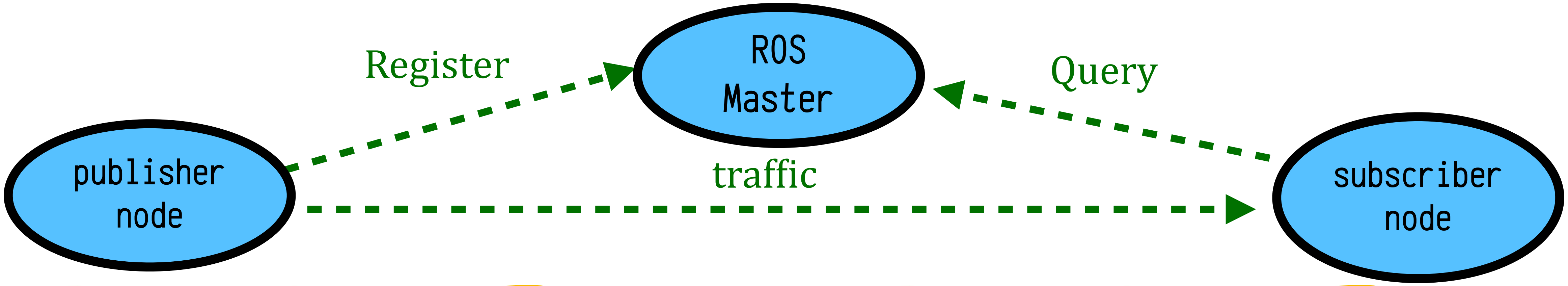
- The ROS Master is a special type of node that curates the communications between nodes.

- Traffic does *not* go through the Master; publishers register their published topics to the Master, and subscribers query the Master for knowing who is publishing the topic using **special control messages.**

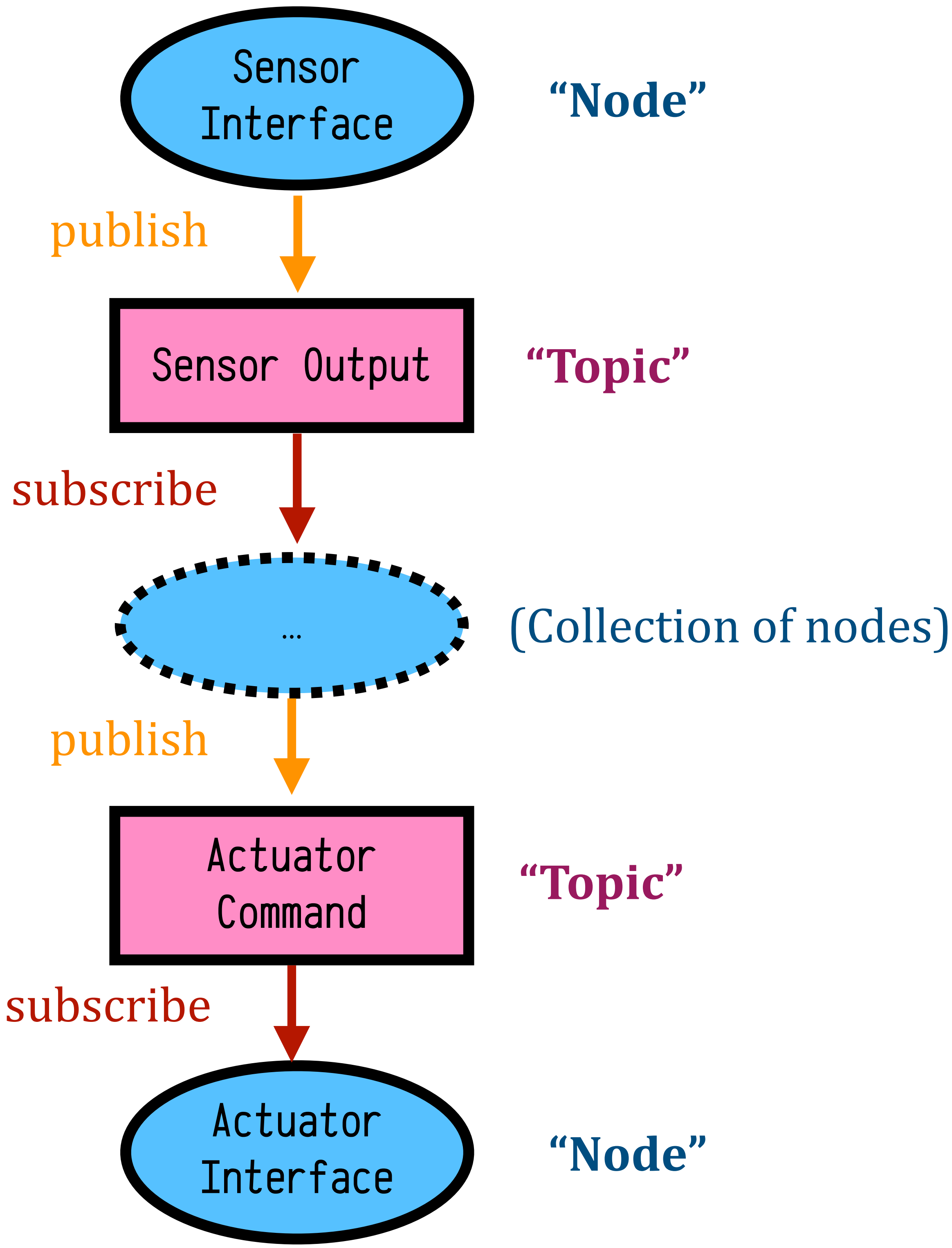


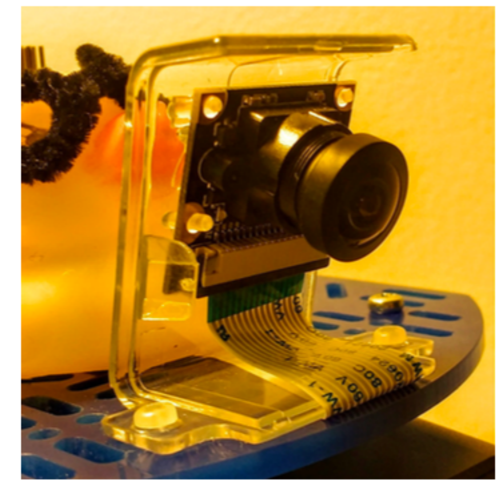
illusion

reality

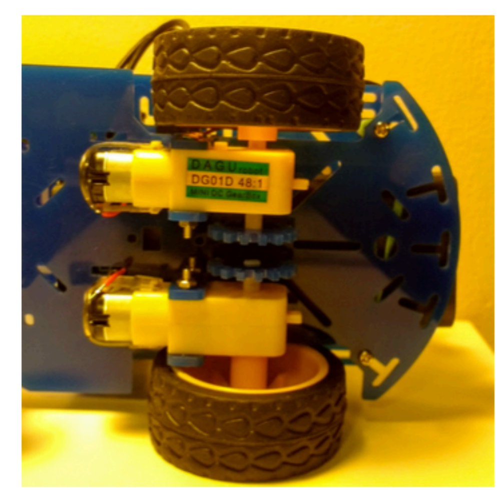


Nodes/topics example for basic Robotics Pipeline



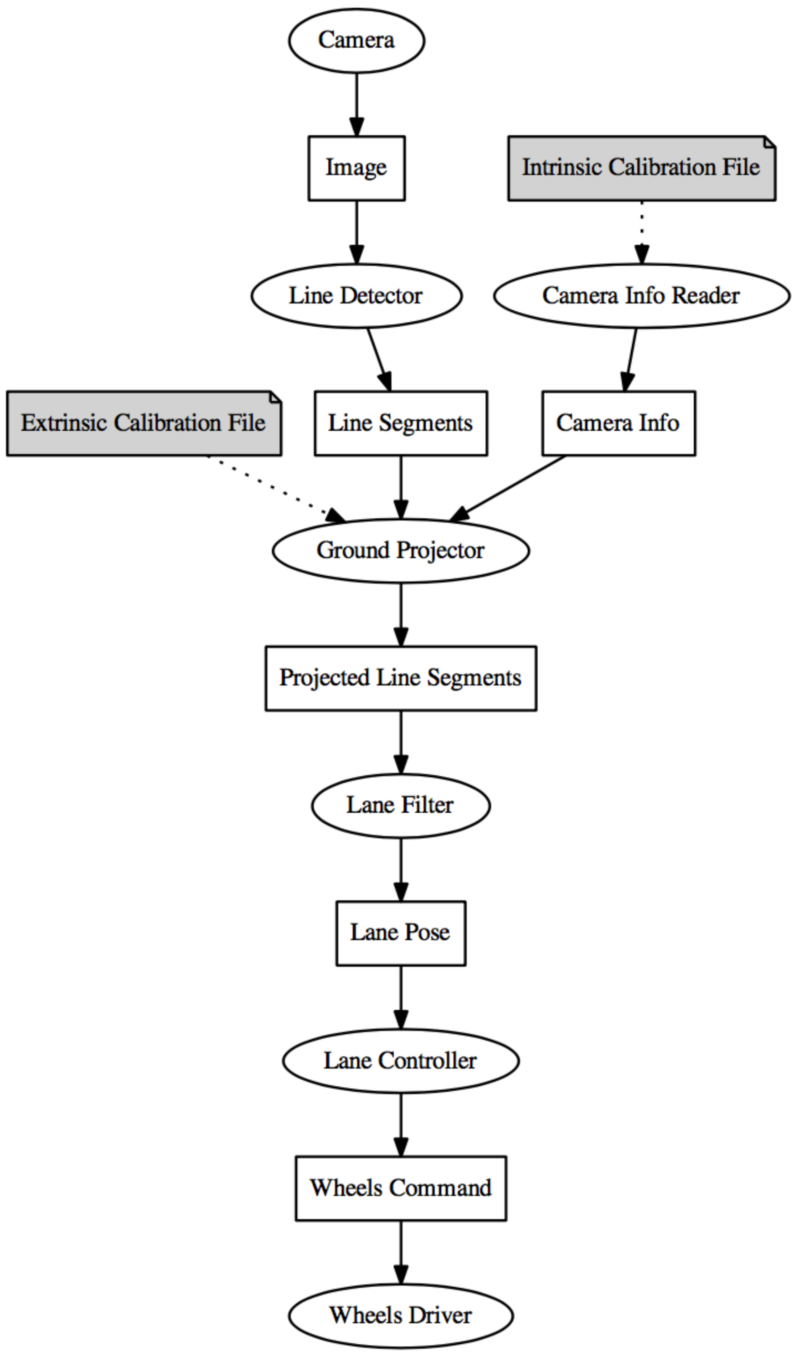


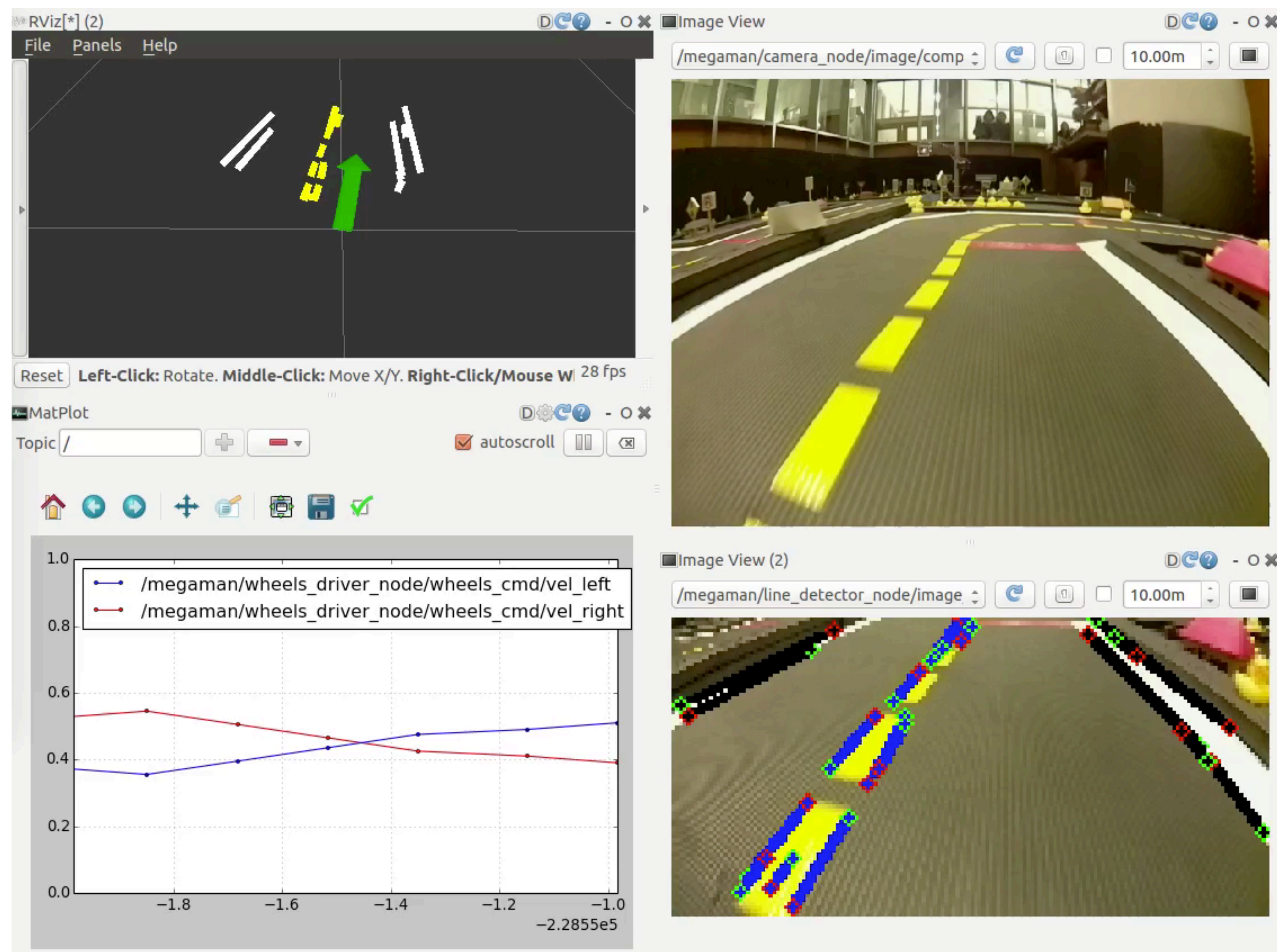
Sensors: Camera



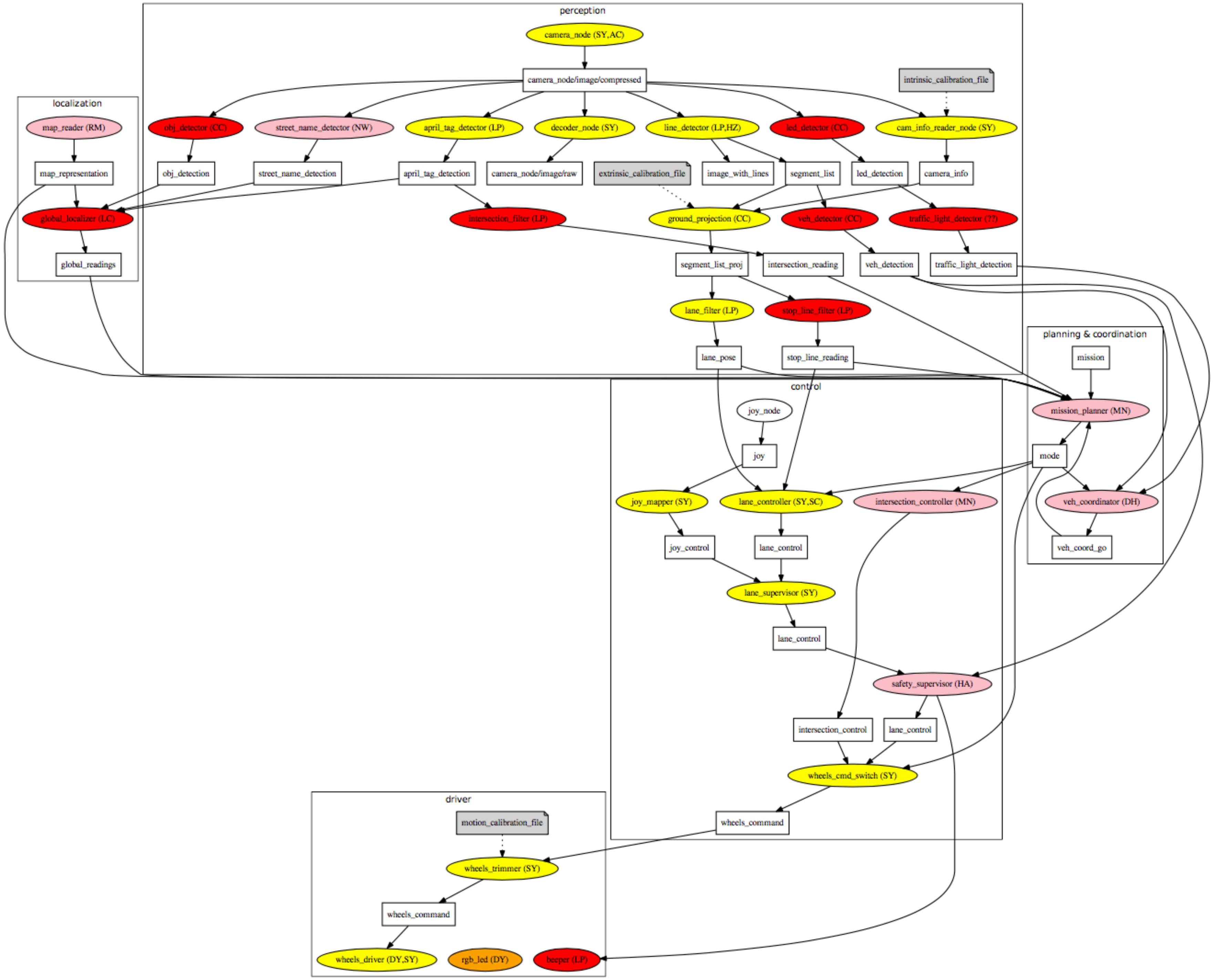
Actuators: Wheel Motors

Lane Following ROS Computation Graph





Things Get Pretty Big



Messages

- **Primitive built-in types** (`std_msgs`)
  - `bool, string, float32, int32, …`
- **Higher-level built in types**:
  - `geometry_msgs: Point, Polygon, Vector, Pose, PoseWithCovariance, …`
  - `nav_msgs: OccupancyGrid, Odometry, Path, …`
  - `sensors_msgs: Joy, Imu, NavSatFix, PointCloud, LaserScan, …`
- **You can make your own messages.**
  - Similar to creating a new "**class**" **in object-oriented programming.**

Example message in duckietown_msgs

- Segment.msg:

```
uint8 WHITE=0
uint8 YELLOW=1
uint8 RED=2
uint8 color
duckietown_msgs/Vector2D[2] pixels_normalized
duckietown_msgs/Vector2D normal
geometry_msgs/Point[2] points
```

- SegmentList.msg:

```
Header header
duckietown_msgs/Segment[] segments
```

Some ROS vocabulary that we are going to learn

- ✅ **Basic concepts:**
    - ✅ Nodes
    - ✅ Topics
    - ✅ Publishing
    - ✅ Subscribing
    - ✅ The ROS “Master”
    - ✅ Messages
- **Intermediate concepts:**
    - Launch files
    - Parameters / parameter server

Launch Files

- They **describe a “subsystem” of many nodes and their interconnections.**
- Specified in **XML format**. Basic Syntax:

```
<tag attribute1="value1" attribute2="value2">
        <element_tag_1 attribute3="value3">
                <!-- Other nesting tags -->
        </element_tag_1>
</tag>
```

- Top-level tags:
    - `<launch>`: Specifies that this is a launch file
    - `<group>`: Apply some settings to a range
    - `<arg>`: Used to pass arguments between launch files
    - `<node>`: Used to run an executable
    - `<include>`: Include the contents of another launch file.

Example: launch file for a single node

launch

```
<launch>
    <arg name="veh"/>
    <arg name="config" default="baseline"/>
    <arg name="param_file_name" default="default" doc="Specify a param file. ex:megaman"/>
    <arg name="local" default="false" doc="true to launch locally on laptop. false to launch of vehicle"/>
    <arg name="pkg_name" default="lane_control" doc="name of the package"/>
    <arg name="node_name" default="lane_controller_node" doc="name of the node"/>
    <group ns="$(arg veh)">
        <!-- Local -->
        <node if="$(arg local)" pkg="$(arg pkg_name)" type="$(arg node_name).py" name="$(arg node_name)" output="screen"
              clear_params="true" required="true">
            <rosparam command="load"
                      file="$(find duckietown)/config/$(arg config)/$(arg pkg_name)/$(arg node_name)/$(arg param_file_name).yaml"/>
        </node>
        <!-- Remote -->
        <include unless="$(arg local)" file="$(find duckietown)/machines"/>
        <node unless="$(arg local)" machine="$(arg veh)" pkg="$(arg pkg_name)" type="$(arg node_name).py"
              name="$(arg node_name)" output="screen" clear_params="true" required="true">
            <rosparam command="load"
                      file="$(find duckietown)/config/$(arg config)/$(arg pkg_name)/$(arg node_name)/$(arg param_file_name).yaml"/>
        </node>
    </group>
</launch>
```

Example: Composing Launch Files

```
<launch>
    <arg name="veh" doc="Name of vehicle. ex: megaman"/>
    <arg name="local" default="false" doc="true for running on laptop. false for running on vehicle."/>
    <arg name="config" default="baseline" doc="Specify a config."/>
    <arg name="param_file_name" default="default" doc="Specify a param file. ex:megaman." />
    <arg name="joy_mapper_param_file_name" default="$(arg param_file_name)" doc="Specify a joy_mapper param
 file. ex:high_speed" />
    <include file="$(find duckietown)/machines"/>
    <!-- joy -->
    <node ns="$(arg veh)" if="$(arg local)" pkg="joy" type="joy_node" name="joy" output="screen">
        <rosparam command="load" file="$(find duckietown)/config/$(arg config)/joy/joy_node/$(arg
 param_file_name).yaml"/>
    </node>
    <node ns="$(arg veh)" unless="$(arg local)" machine="$(arg veh)" pkg="joy" type="joy_node" name="joy"
 output="screen">
        <rosparam command="load" file="$(find duckietown)/config/$(arg config)/joy/joy_node/$(arg
 param_file_name).yaml"/>
    </node>
    <!-- joy_mapper -->
    <include file="$(find joy_mapper)/launch/joy_mapper_node.launch">
        <arg name="veh" value="$(arg veh)"/>
        <arg name="local" value="$(arg local)"/>
        <arg name="config" value="$(arg config)"/>
        <arg name="param_file_name" value="$(arg joy_mapper_param_file_name)"/>
    </include>
    <!-- run inverse_kinematics_node -->
    <remap from="inverse_kinematics_node/car_cmd" to="joy_mapper_node/car_cmd"/>
    <remap from="inverse_kinematics_node/wheels_cmd" to="wheels_driver_node/wheels_cmd" />
    <include file="$(find dagu_car)/launch/inverse_kinematics_node.launch">
        <arg name="veh" value="$(arg veh)"/>
        <arg name="local" value="$(arg local)"/>
        <arg name="config" value="$(arg config)"/>
    </include>
    <!-- ... -->
</launch>
```

Parameters in ROS

- Configurations are loaded at launch time.
- Parameters are stored on the **parameter server** and can be **queried** or **adjusted at any time**
    - Bonus: We can tune the system without restarting the applications.
- Common pitfall: parameters are preserved on the parameter server until the ROS Master is killed.
- What types of things should be parameters?
    - Controller gains;
    - Color thresholds;
    - ...

Checklist of ROS commands to know and use

Commands/tools to become familiar with

- `roscore`

- `roslaunch`

- `rosnode list`
- `rosnode info`

Topics

- `rostopic list`
- `rostopic echo topic_name`
- `rostopic hz topic_name`

- `rqt_graph`

Visualizing data

- rqt_plot
- rviz
- rqt_image_view
- rqt_console

Parameter server

- `rosparam get param_name`
- `rosparam set param_name`
- `rosparam dump file_name [namespace]`
- `rosparam load file_name [namespace]`

Recording and playing logs with `rosbag`

- `rosbag record`
- `rosbag play`

Programming tips

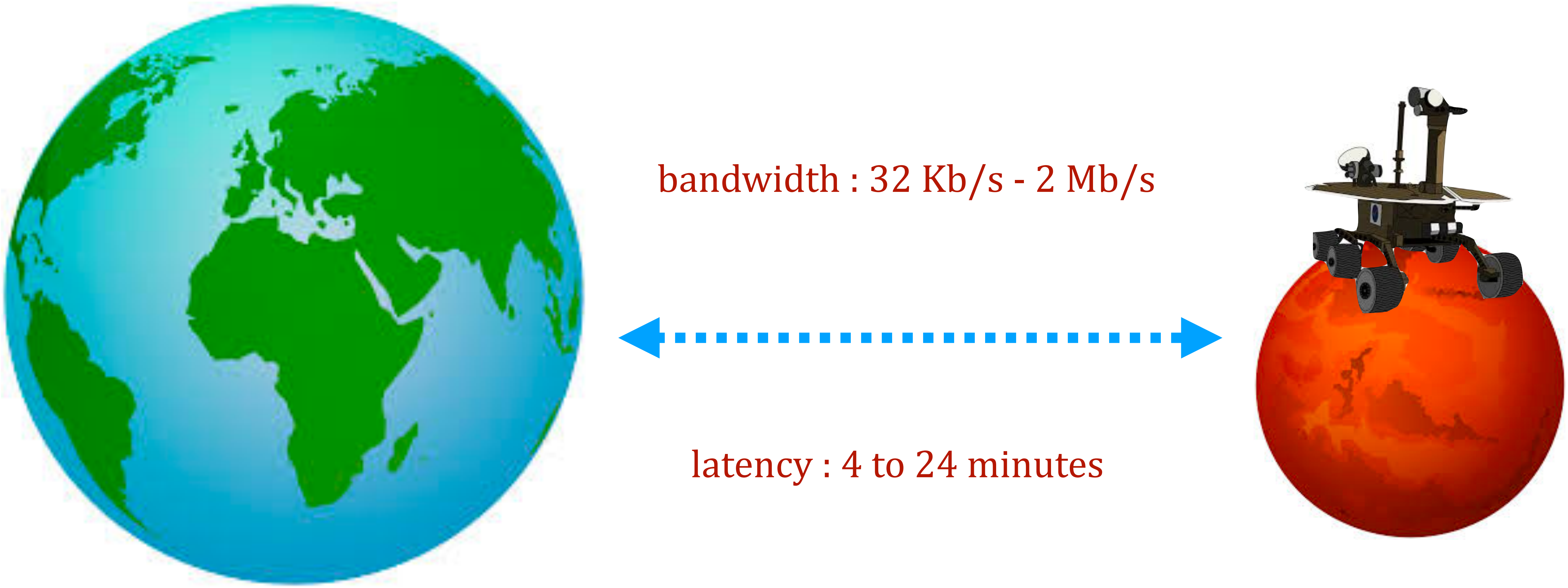
Bandwidth, throughput, latency, jitter

- **Bandwidth** (measured in bits/second) is the maximum rate at which information can be transferred.
- **[Message] Throughput** (measured in Hz) is the rate at which messages arrive.
  - The relation between bandwidth and throughput depends on the size in bits of the packets / messages.
- **Message Latency** (measure in seconds) is the delay between the sender sending the message and the receiver decoding it.
- **Jitter** is variation in delay over time.

- **Mega tip**: it is very intuitive to think of throughput as the main performance metric (how many images can I process per second?) however **latency is what kills you in robotics**.

Latency and throughput are independent

- Isn't latency = 1/throughput? No, think about parallel processing.
- **You could process a message arbitrarily fast** (low latency) but still be bound on the frequency of the data source.
- You could **increase the latency arbitrarily** while keeping same throughput.



Do not complain about your network setup

- Some colleagues have this on their dashboard:

Event-based vs periodic processing

- **Event-Based** processing**: process events as they arrive**

```
def callback(data):
  # process data here

rospy.Subscriber("topic_name", TopicType, callback)
```

- **Periodic** processing**:** process the last data received every **period T.**

```
self.data = TopicType()

def subscriber_callback(data):
  self.data = data

rospy.Subscriber("topic_name", TopicType, subscriber_callback)

def timer_callback(event)
   # process last self.data

rospy.Timer(rospy.Duration(2), timer_callback)
```

All-data vs most up-to-date data

- If (time to process one message) > 1/{throughput of the message data}
  you cannot process all data, and you have a decision to make.
  - Option 1: Always **grab the latest data** and ignore that you may have missed some
  - Option 2: **Make sure to get all the data.** The data will backlog, but each data could be important.
  - Option 3: Figure out what fraction you have to ignore, and **discard as needed** to be as current as possible.
- ROS supports **queue size limits** that could help in these scenarios.
  Read the documentation to understand the semantics.
  - Publisher side:

```
pub = rospy.Publisher('chatter', String, queue_size=10)
```

  - Subscriber side

```
sub = rospy.Subscriber('chatter', String, callback, queue_size=10)
```