An introduction to ROS

matteo.luperto at unimi.it

ROS: the Robot Operating System

ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. [wiki.ros.org]

Robot software architecture

Low level functionalities as real-time motor controllers, sensors drivers, battery management

+

Core functionalities as mapping, localization, navigation, people detection

+

Reasoning mechanism for path planning, task allocation, self management

ROS

Robot software architecture

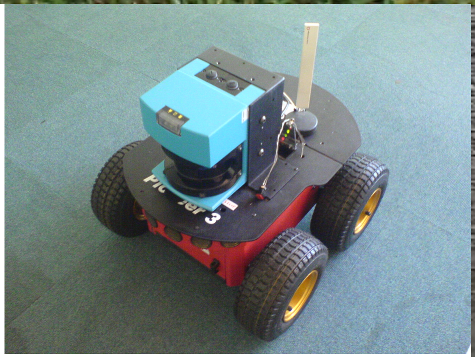
The development of (even a single) robots (functionality) requires both low-level hardware related and high-level AI-based mechanism

Modularity and scalability are consequently core features in a robot software architecture

ROS provide this

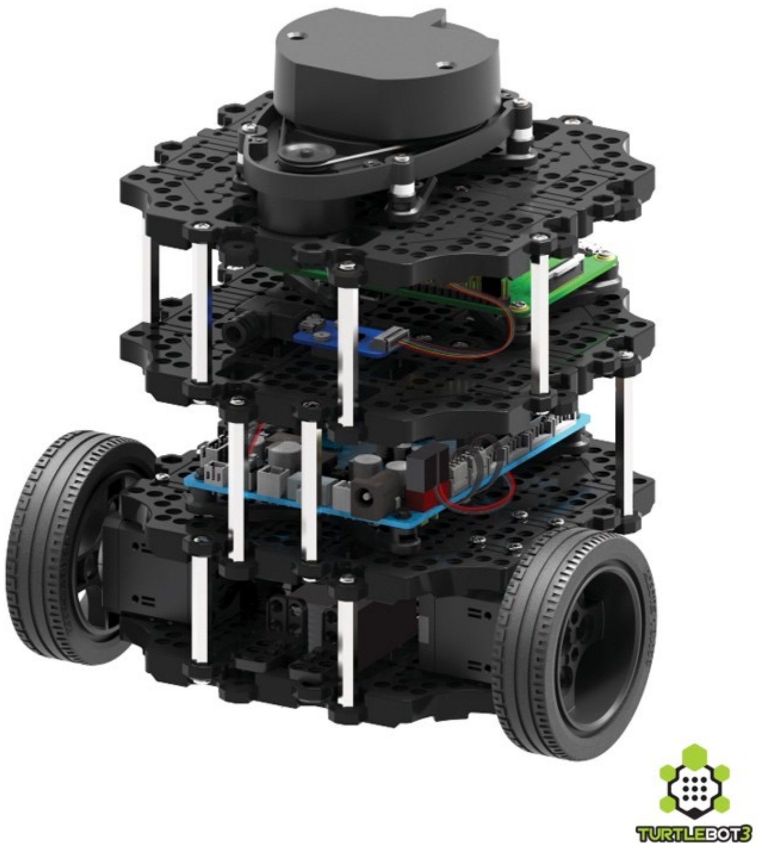
ROS has established itself as the de-facto standard for robot development

ROS



More at:
robots.ros.org

Our ROS robots



Sensors with ROS [wiki.ros.org]

What is ROS?

Is a Meta-Operating System

- Scheduling – loading – monitoring, and error handling
- virtualization layer between applications and distributing computing resources
- runs on top of (multiple) operating system(s)

- is a framework
- not a real-time framework but embed real-time code
- enforce supports a modular software architecture

ROS SW architecture

- distributed framework of processes (*Nodes*)
- enables executables to be individually designed and loosely coupled at runtime.
- processes can be easily shared and distributed.
- supports a federated system of code *Repositories* that enable collaboration to be distributed as well.

This design, from the filesystem level to the community level, enables independent decisions about development and implementation, but all can be brought together with ROS infrastructure tools.

More ROS features

- thin: ROS is designed to be as thin as possible
- easy to integrate with other frameworks and libraries
- language independence
  core languages are Python and C++ but you can use what you want
- easy testing: built in unit/integration test framework and debug tool
- scaling: ROS tools can be distributed across different machines and is appropriate for large development process

The core idea behind all of this is: <u>code reuse + modularity</u>

What ROS provides

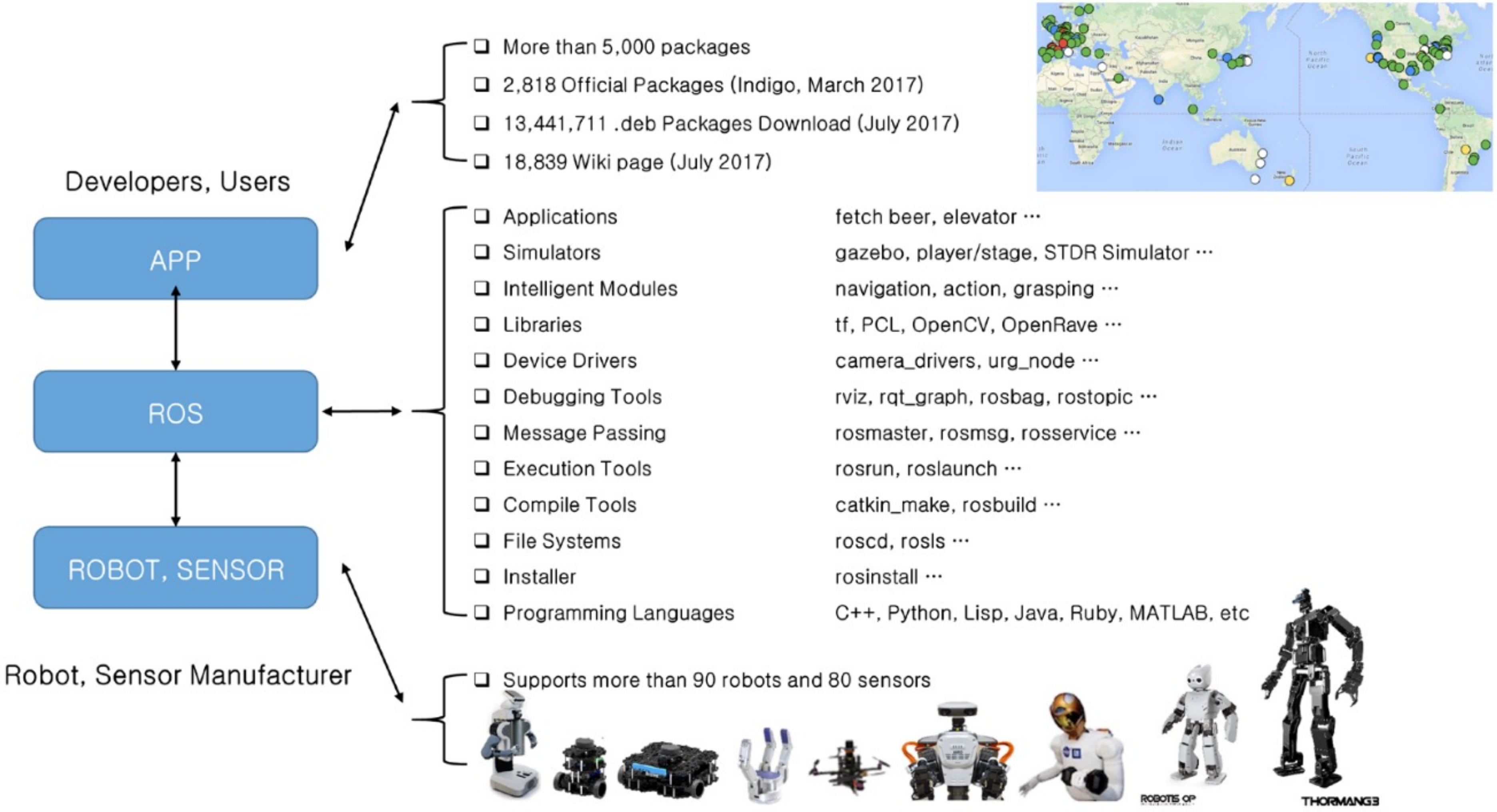
Client Layer	roscpp	rospy	roslisp	rosjava	roslibjs		
Robotics Application	MoveIt!	navigatioin	executive smach	descartes	rospeex		
	teleop pkgs	rocon	mapviz	people	ar track		
Robotics Application Framework	dynamic reconfigure	robot localization	robot pose ekf	Industrial core	robot web tools	ros realtime	mavros
	tf	robot state publisher	robot model	ros control	calibration	octomap mapping	
	vision opencv	image pipeline	laser pipeline	perception pcl	laser filters	ecto	
Communication Layer	common msgs	rosbag	actionlib	pluginlib	rostopic	rosservice	
	rosnode	roslaunch	rosparam	rosmaster	rosout	ros console	
Hardware Interface Layer	camera drivers	GPS/IMU drivers	joystick drivers	range finder drivers	3d sensor drivers	diagnostics	
	audio common	force/torque sensor drivers	power supply drivers	rosserial	ethercat drivers	ros canopen	
Software Development Tools	RViz	rqt	wstool	rospack	catkin	rosdep	
Simulation	gazebo ros pkgs	stage ros					

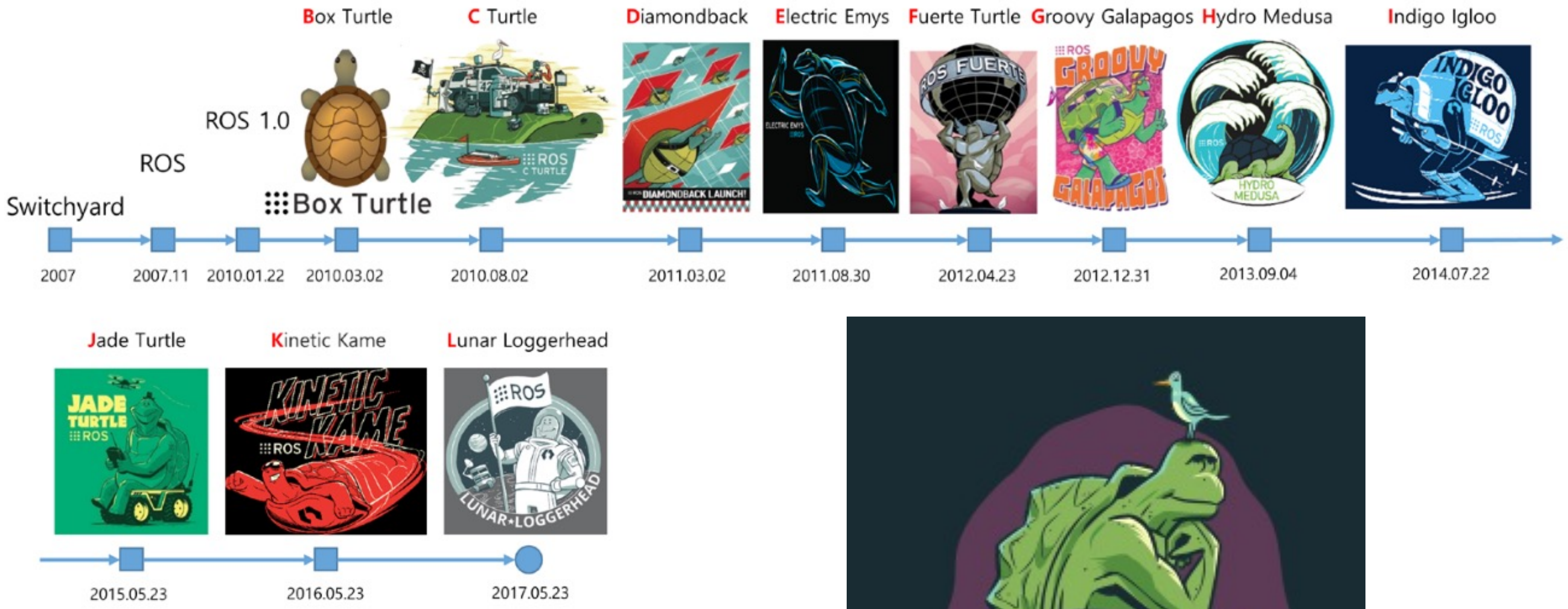
- core and advanced robot functionalities (mapping, localization, navigation, obstacle avoidance)
- drivers and integration with sensors
- integration with multiple robot architectures
  UAV – manipulators –wheeled robots
- integration with libraries (OpenPose, OpenCV, deep learning fw)
- simulation tools

All free and ready to use
Support from the community

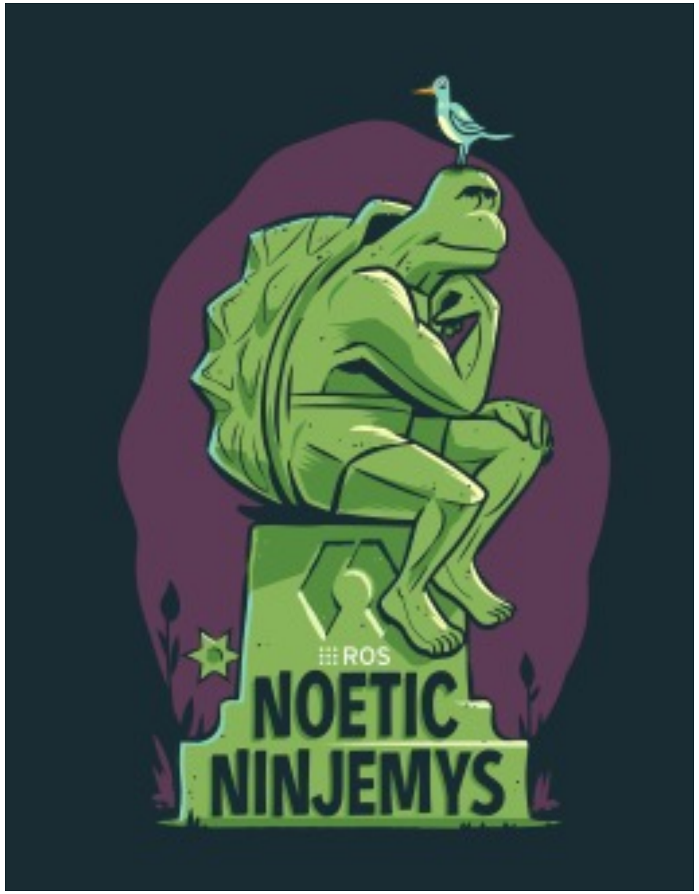
ROS

ROS-community





- more than 10y of ROS now
- last version (ROS1): ROS Noetic (2020)
- **EOL 2023**
- next mayor release: ROS 2
  - already released
  - migration in the way



ROS

Core aspects of ROS

ROS aspects

- nodes
- topics
- messages

Building blocks of ROS

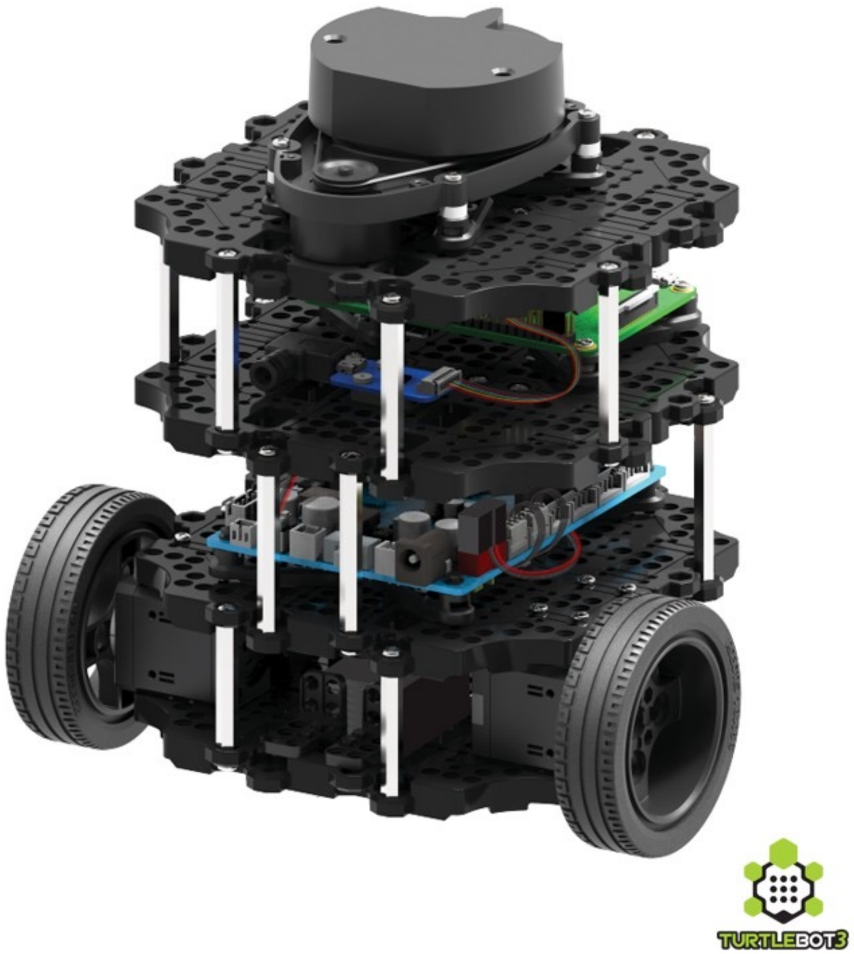
- services
- actions
- transforms

Communication / SW architecture

- debugging Tools
- simulations
- bags

Developers tools

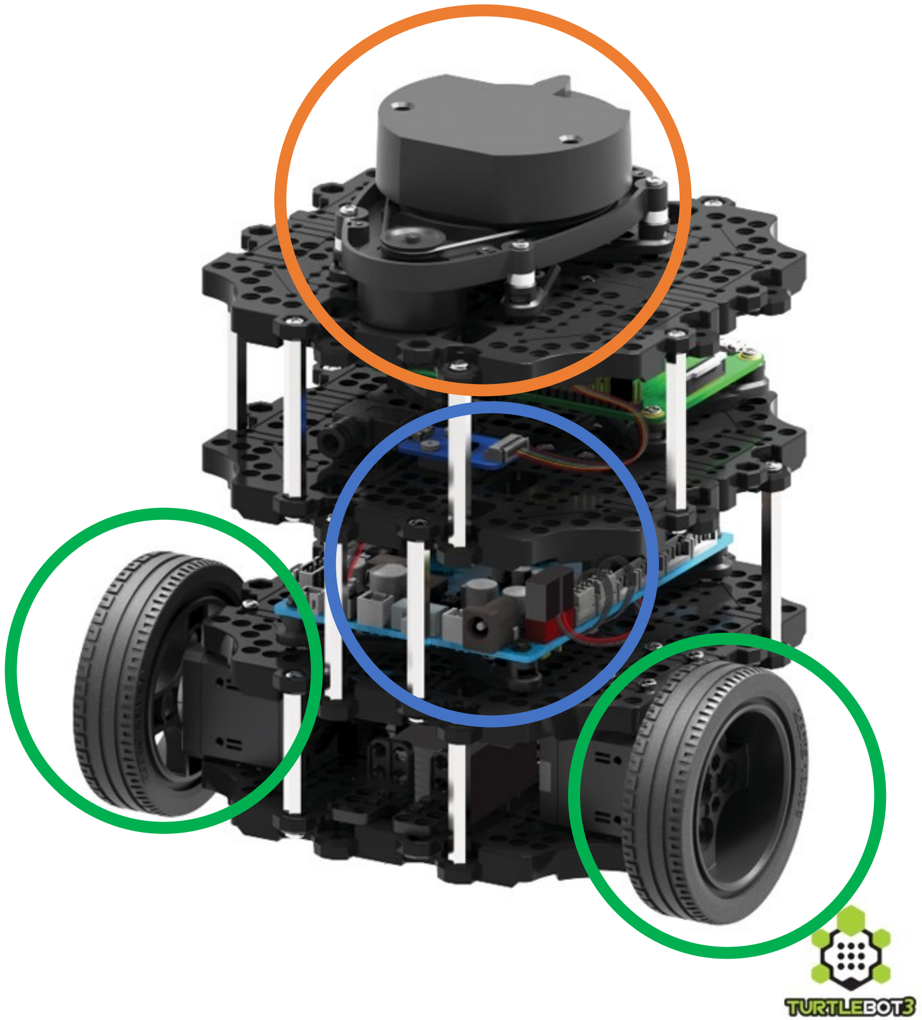
ROS *nodes*



A *node* is a process that performs computation:

- nodes are combined together into a graph and communicate with one another using streaming topics, services, and parameters,
- are meant to operate at a fine-grained scale,
- a robot control system will usually comprise many nodes.

ROS *nodes*



For example, one node controls a laser range-finder, one Node controls the robot's wheels motors, one node performs mapping, one localization, one node performs path planning, one node gives velocity commands to the wheels, one node provides a graphical view of the system, and so on.

ROS *nodes*

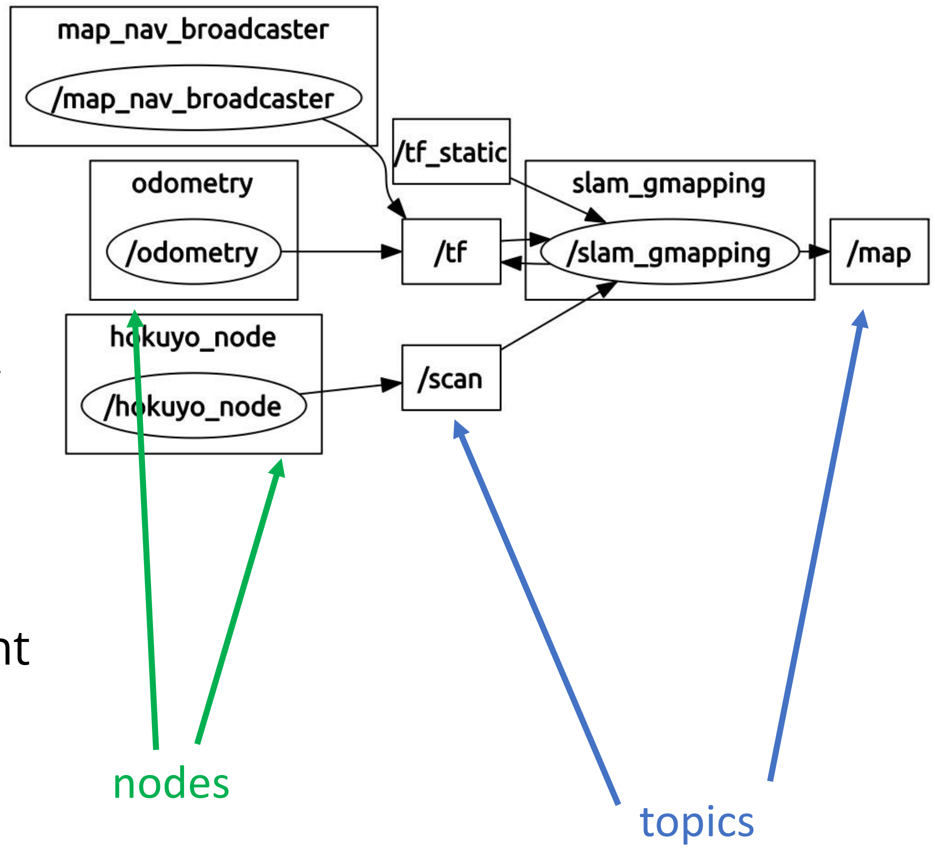
The use of nodes in ROS provides several benefits to the overall system.

- *fault tolerance* as crashes are isolated to individual nodes.
- *code complexity* is reduced in comparison to monolithic systems. Implementation details are also well hidden - nodes expose a minimal API –
- alternate implementations, even in other programming languages, can easily be substituted.

Nodes and *topics*

Topics are named buses over which nodes exchange messages.

- topics have **anonymous publish/subscribe semantics**, which decouples the production of information from its consumption.
- nodes are not aware of who they are communicating with.
- nodes that are interested in data *subscribe* to the relevant topic; nodes that generate data *publish* to the relevant topic.
- there can be multiple publishers and subscribers to a topic.



ROS *topics* and *messages*

geometry_msgs/Point.msg

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

sensor_msgs/Image.msg

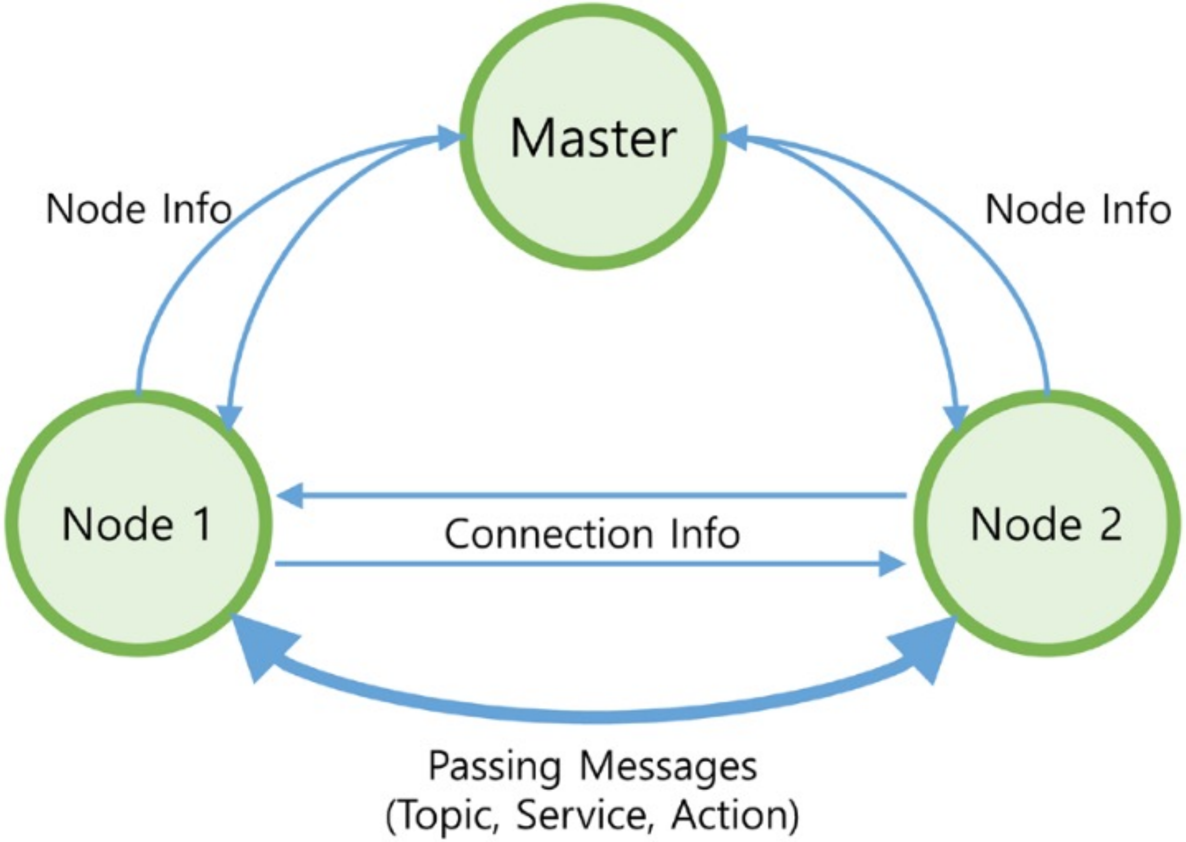
```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

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
```

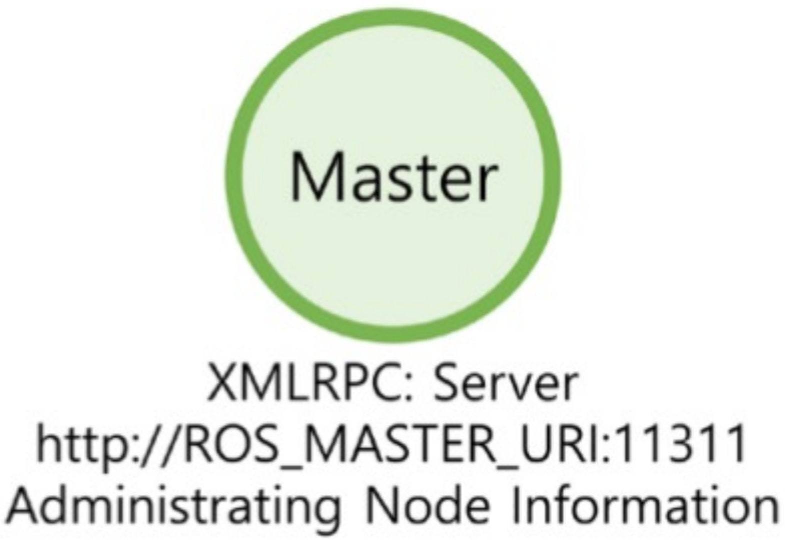
- each topic is strongly typed by the ROS message type used to publish to it
- nodes can only receive messages with a matching type.
- type consistency is not enforced among the publishers, but subscribers will not establish message transport unless the types match.
- all ROS clients check to make sure that an MD5 computed from the message format match.

ROS master



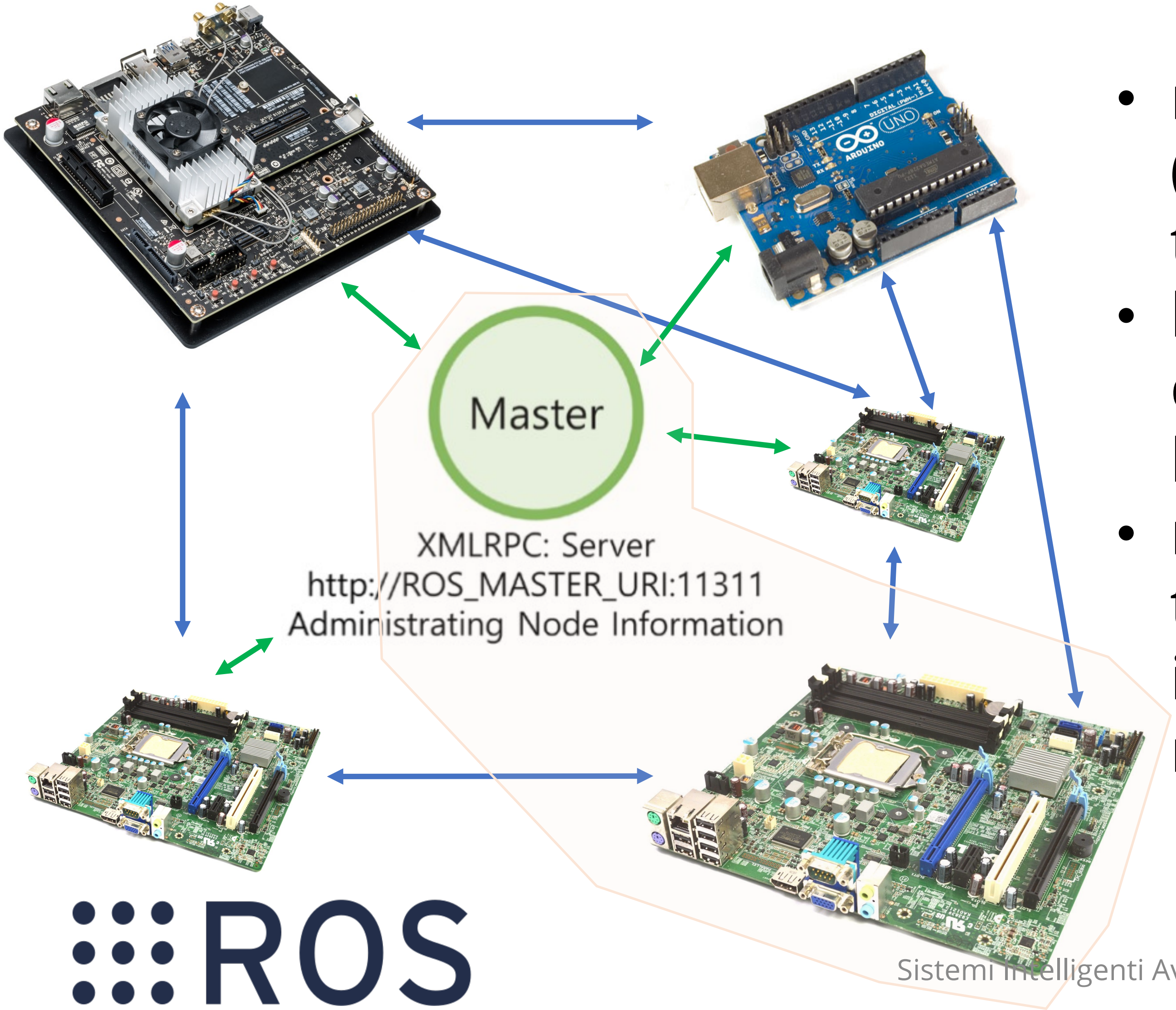
- the ROS Master provides naming and registration services to the rest of the nodes in the ROS system.
- it tracks publishers and subscribers to topics.
- it enables individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer.

ROS master and nodes



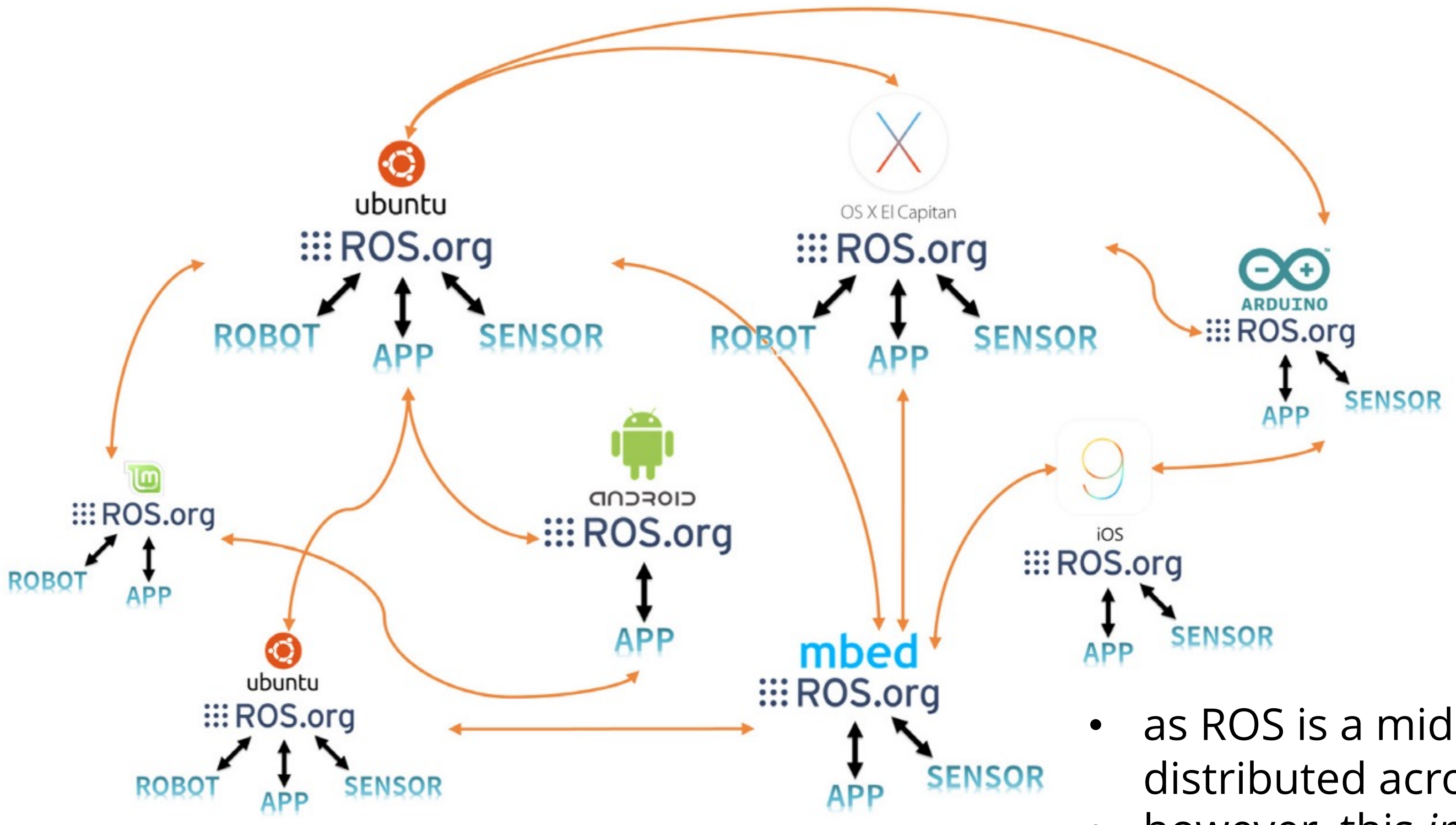
- the ROS master is a process and it is defined by its IP/port shared across all nodes
- acts as coordinator and manages the communication among nods
- this allows nodes to be distributed on different machines
  (in the same network)
- this mechanism allows to decouple the execution of a process from the machine where the process is distributed

ROS master and nodes



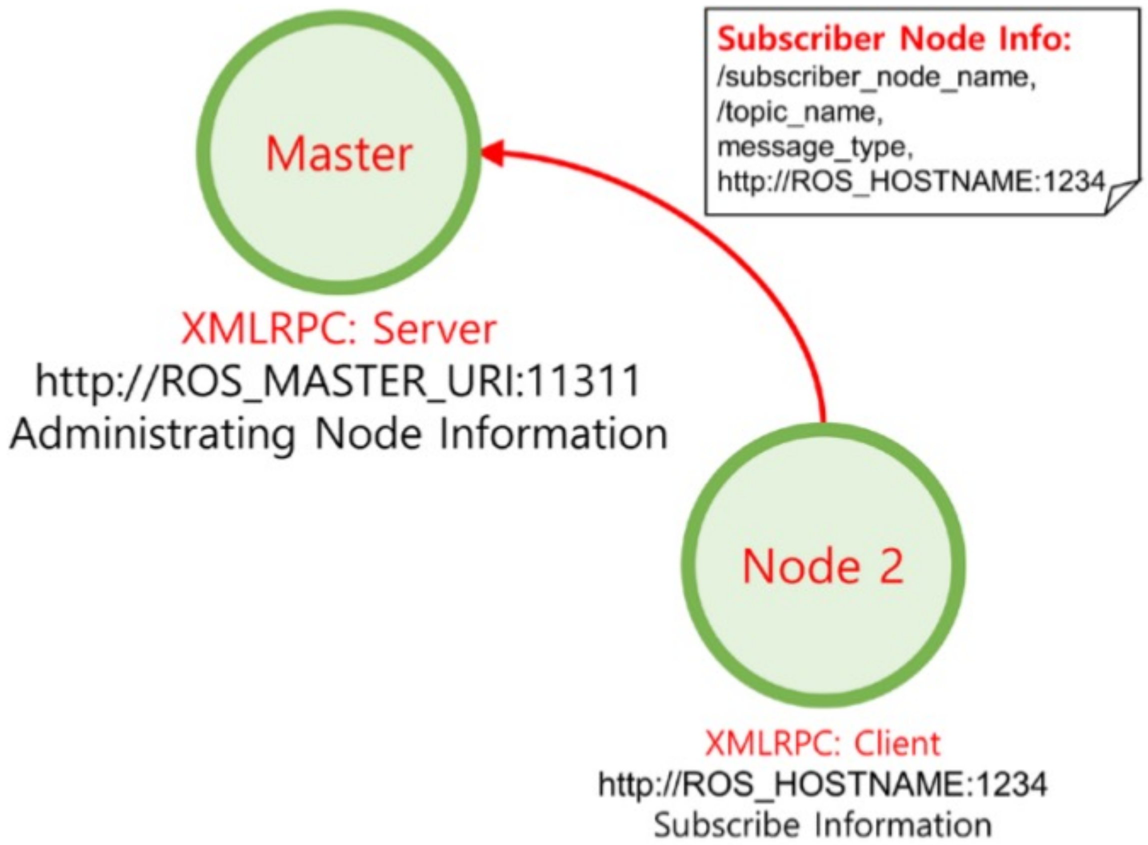
- robots may have to perform several (computationally intensive) tasks together
- hardware decoupling allows to distribute such tasks on dedicated hardware (e.g., Nvidia Jetson for GPUs)
- moreover, robots are <u>hardware</u> and this architecture allows to easily interface control boards for sensors, motors, etc.. (e.g., Arduino)

ROS on multiple platforms



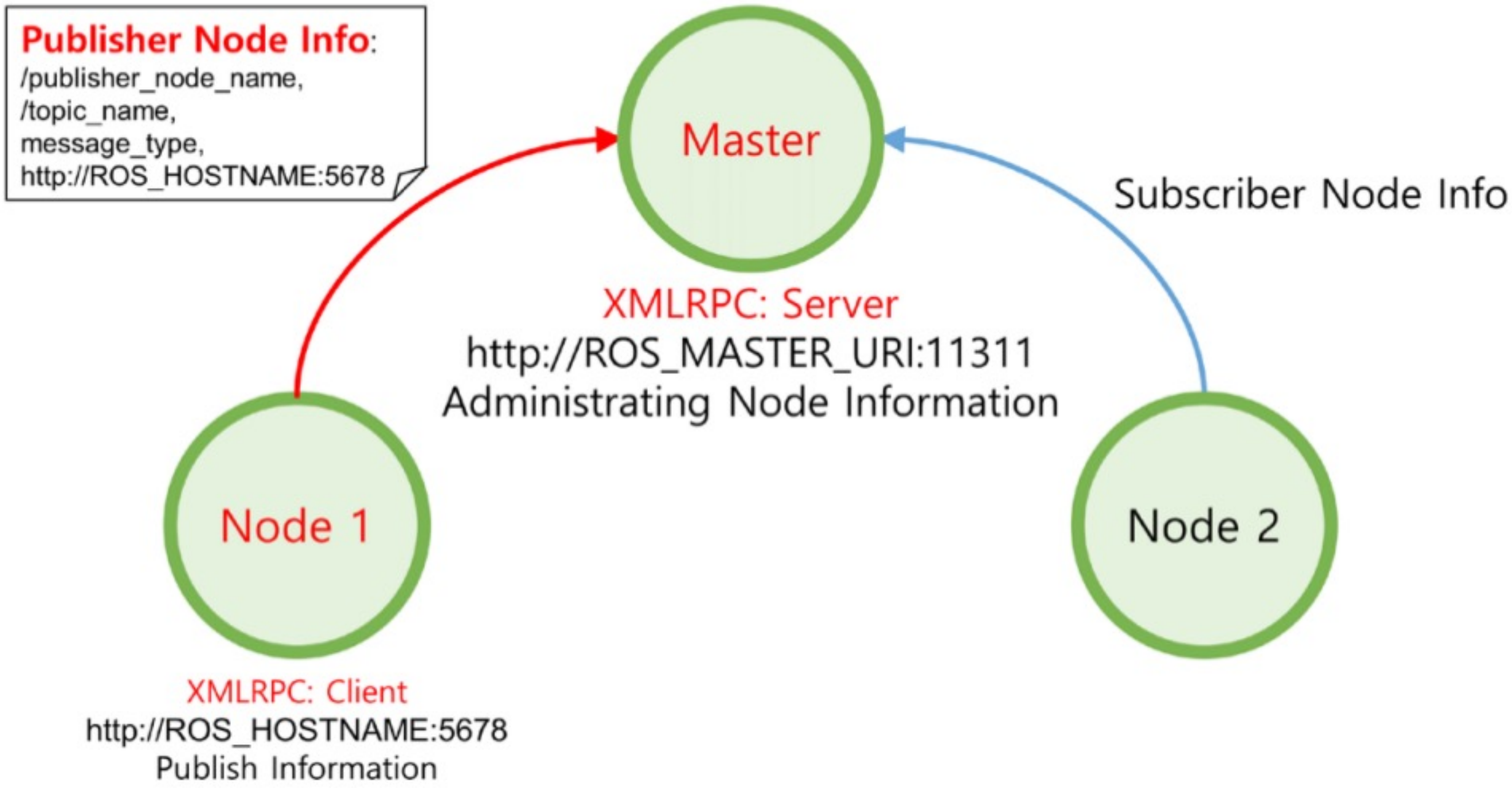
- as ROS is a middleware, computation can be distributed across different OS
- however, this *in practice* is far than ideal
- OS independence is de-facto provided for linux-based and embedded systems.
- rule-of-thumb: use Ubuntu for non-embedded systems not all versions either, but this is improved with ROS2

Set up a ROS topic publisher/subscriber



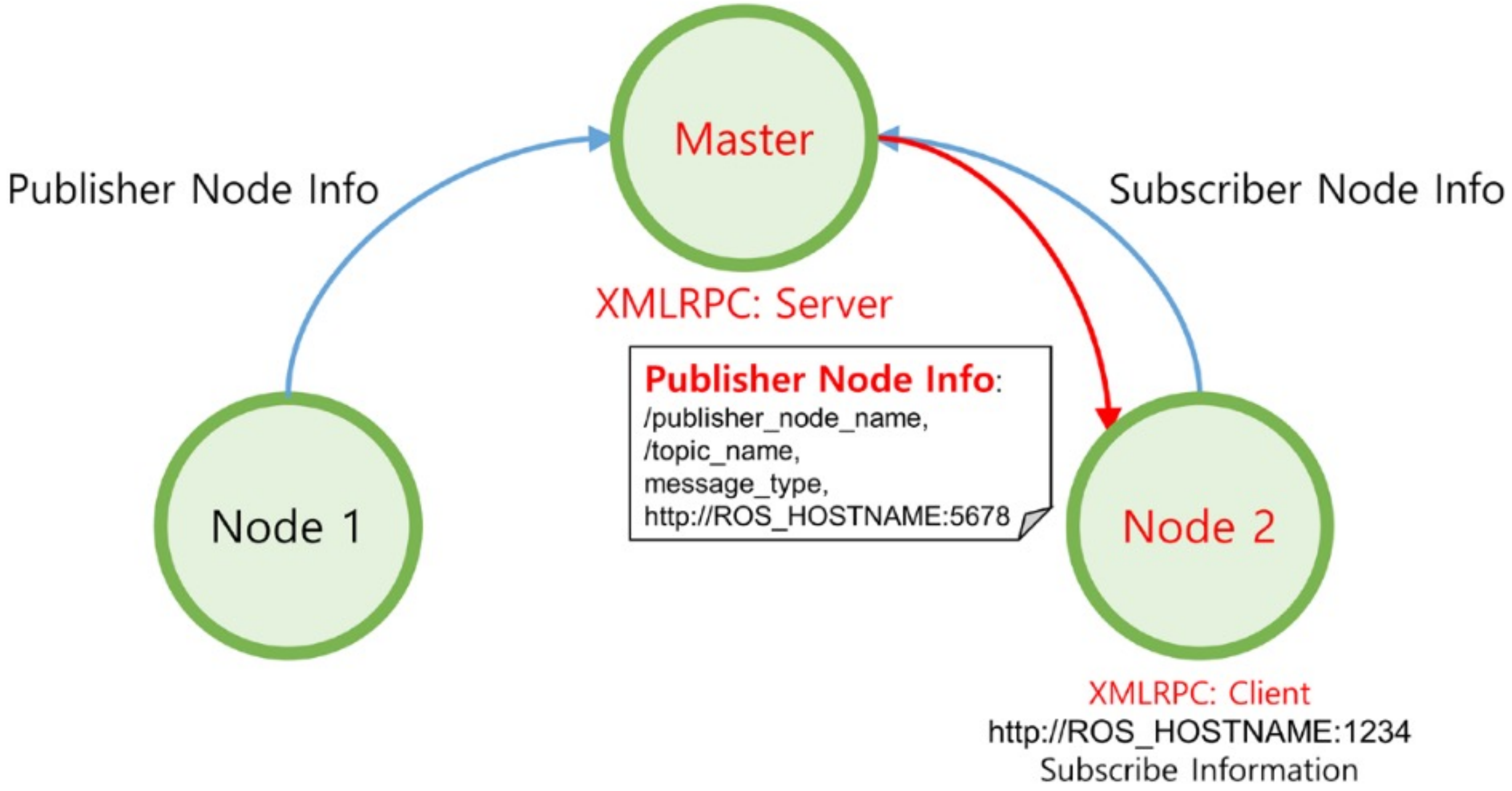
- a subscriber node registers to the ROS MASTER
- and announces its
  - Name
  - Topic name
  - Message Type
- communication is performed using XMLRPC

Set up a ROS topic publisher/subscriber



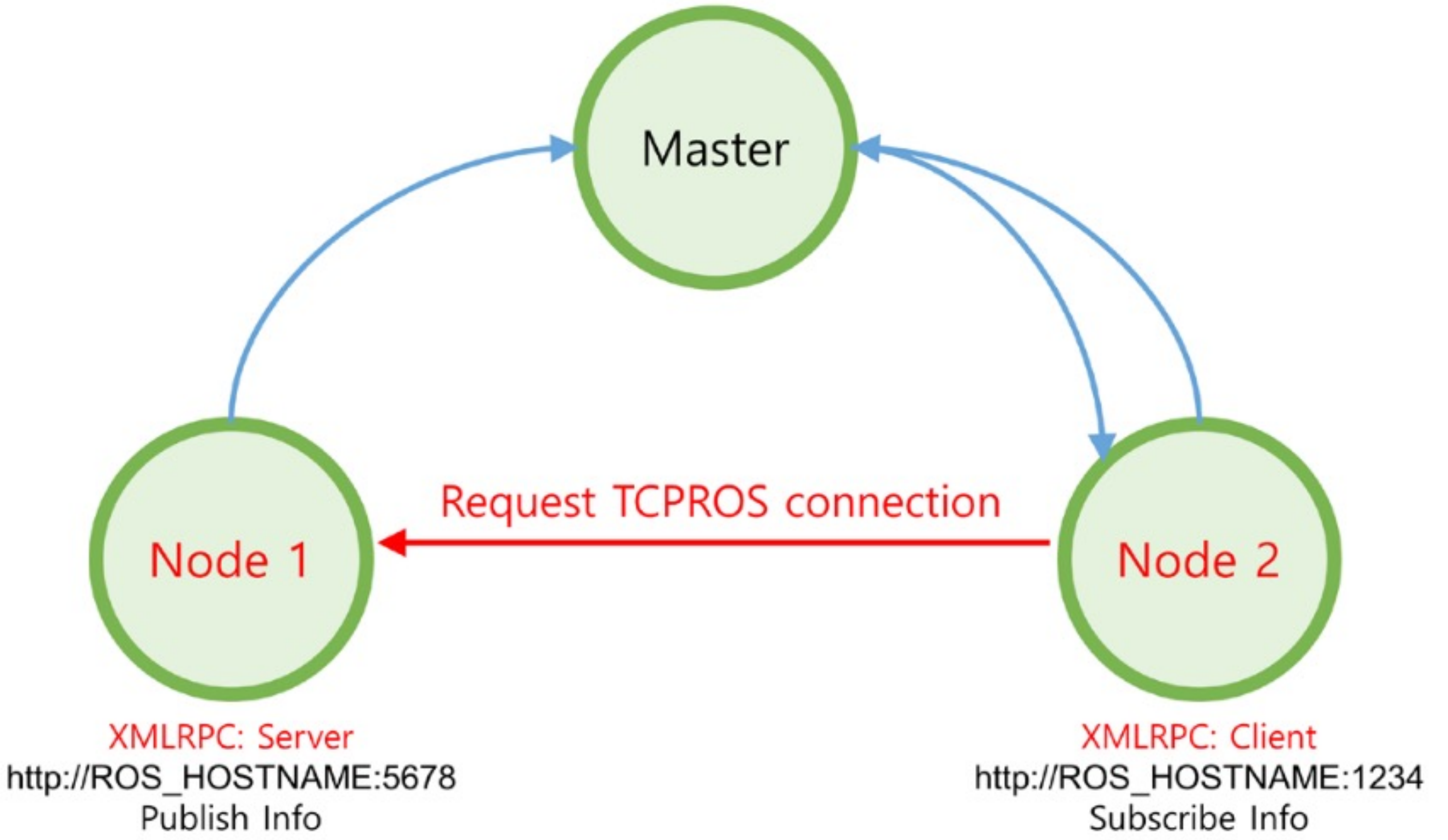
A publisher node now registers to the ROS MASTER

Set up a ROS topic publisher/subscriber



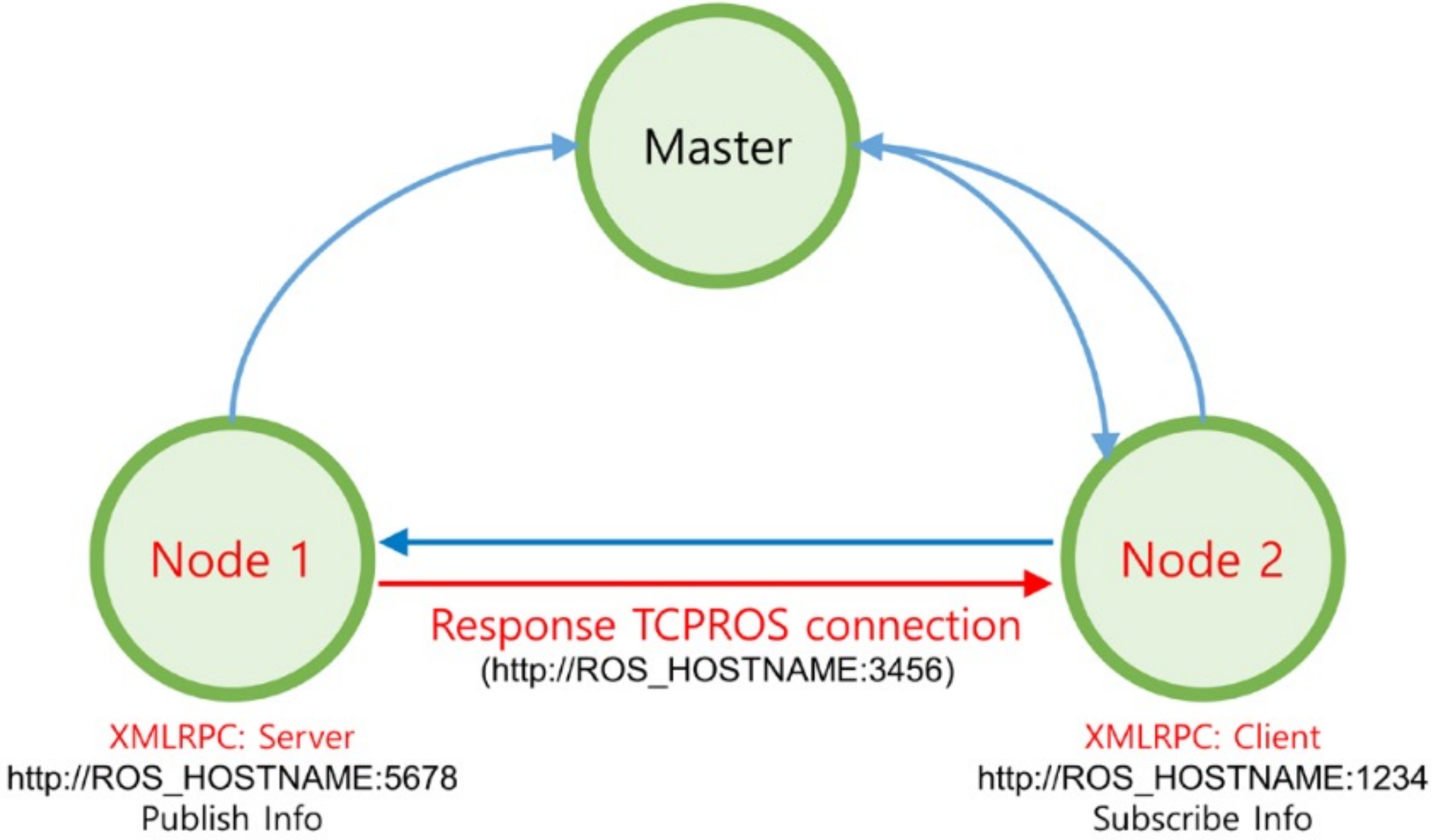
The ROS MASTER distributes info as all subscribers that want to connect to the topic and to the publisher node

Set up a ROS topic publisher/subscriber



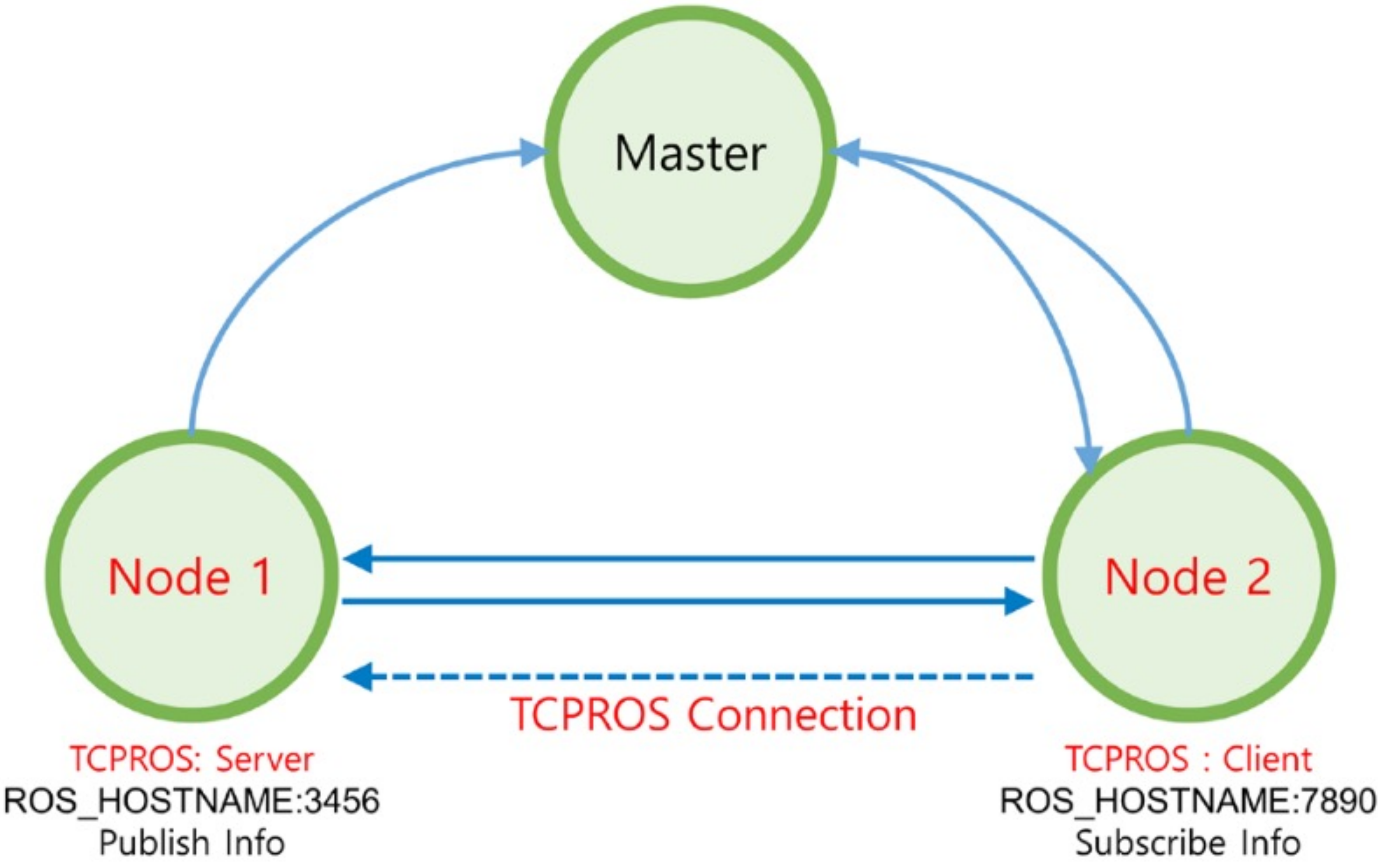
The subscriber node requests a direct connection to the published node and transmits its information to the publisher node

Set up a ROS topic publisher/subscriber



The publisher node sends the URI address and port number of its TCP server in response to the connection request.

Set up a ROS topic publisher/subscriber

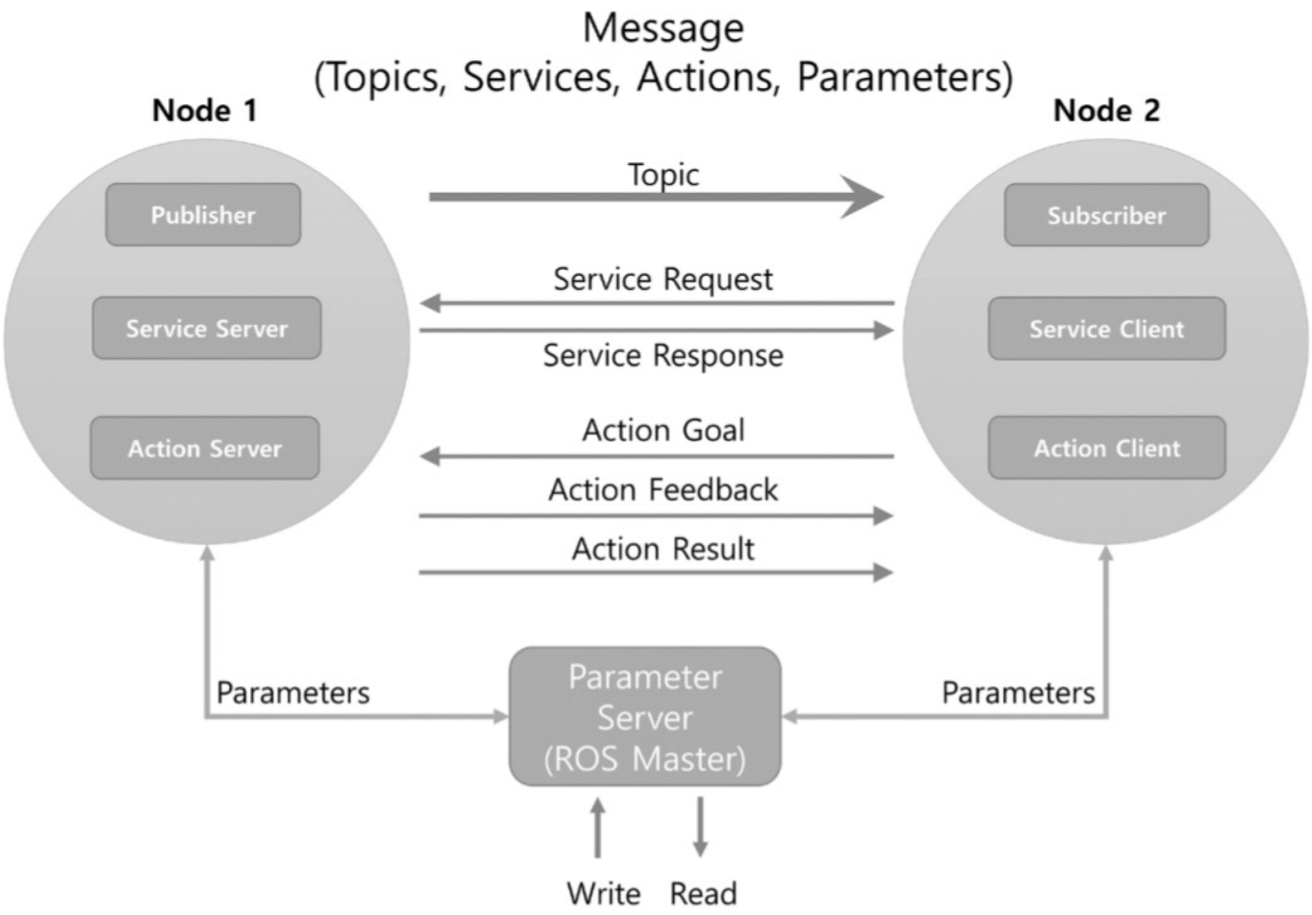


At this point a direct connection between publisher and subscriber node is established using TCPROS (TCP/IP based protocol)

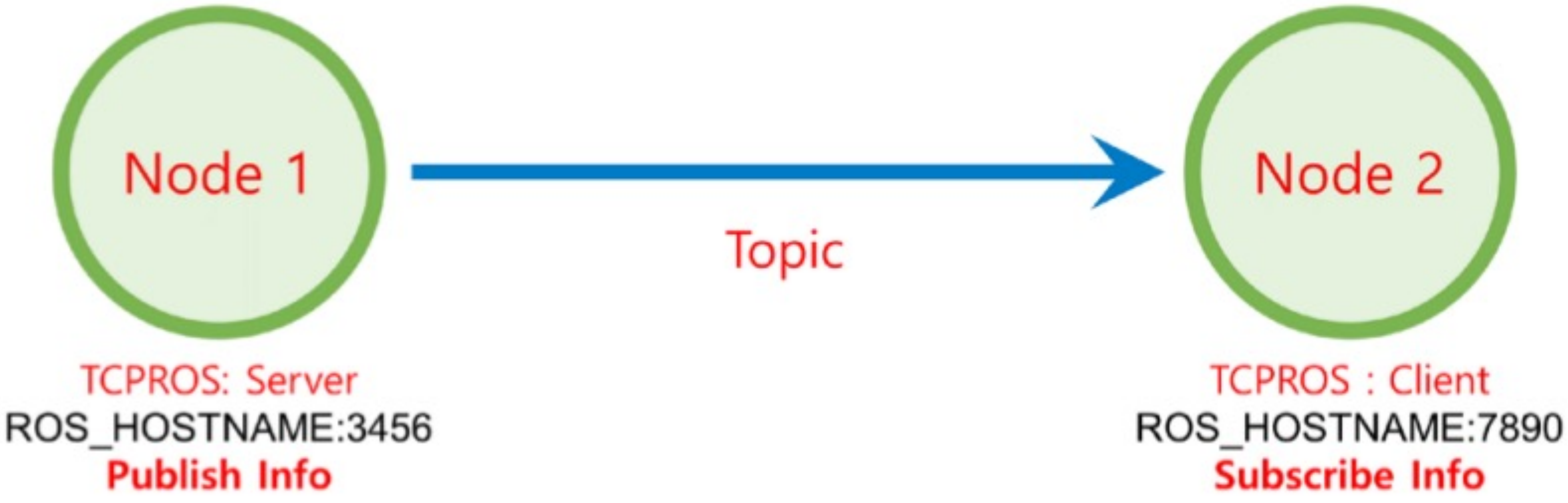
Communication among nodes

After communication between nodes is established, ROS provides 3 types of interactions

- Topics
- Services
- Actions



Communication among nodes



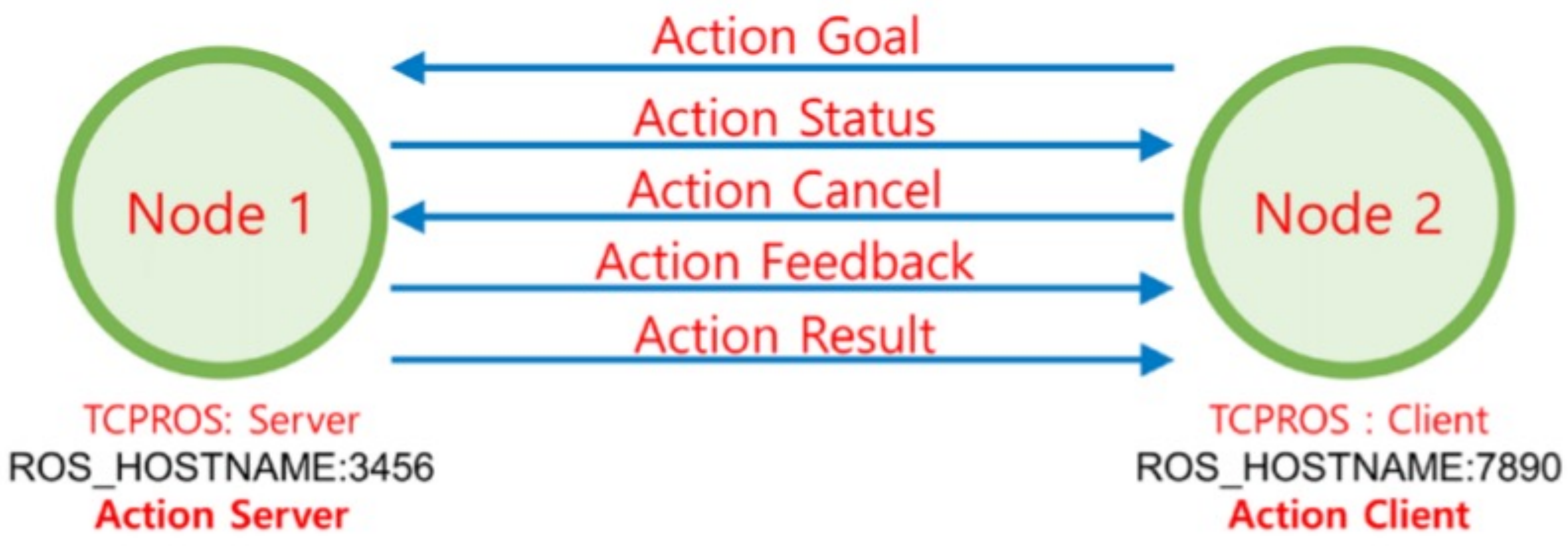
- The standard communication mechanism is using ROS topics.
- Nodes can have multiple topics
- Nodes can even use topics for internal communication
- Continuos -loop()- or one-shot (e.g. when data are ready)

ROS *Services*

- ROS services are synchronous request from one node to another.
- Request/Reply mechanism.

A client can make a persistent connection to a service, which enables higher performance at the cost of less robustness to service provider changes.

ROS *Actions*



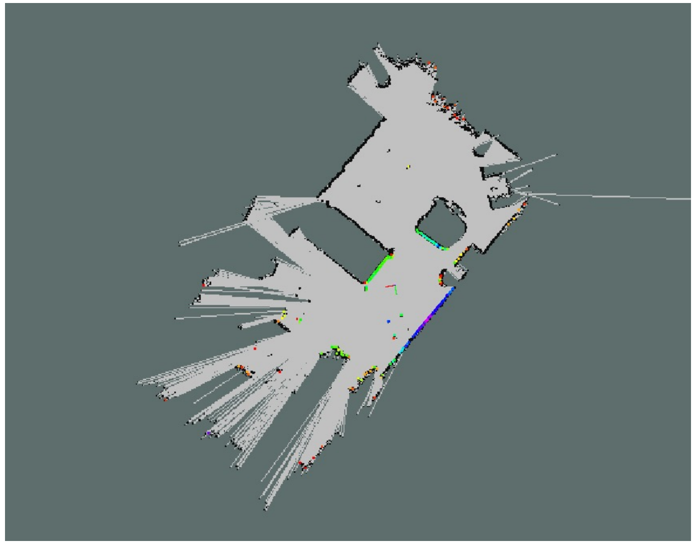
If the service takes a long time to execute, the user might want the ability to cancel the request during execution or get periodic feedback about how the request is progressing.
Action Services are for these tasks.

- ROS services are asynchronous request from one node to another.
- Request/Reply mechanism, with feedbacks and the possibility to cancel the request.

ROS *parameter server*

The parameter server is a shared, multi-variate dictionary that is accessible via network APIs.

- nodes use this server to store and retrieve parameters at runtime.
- used for static, non-binary data such as configuration parameters.
- globally viewable so that tools can easily inspect the configuration state of the system and modify if necessary.

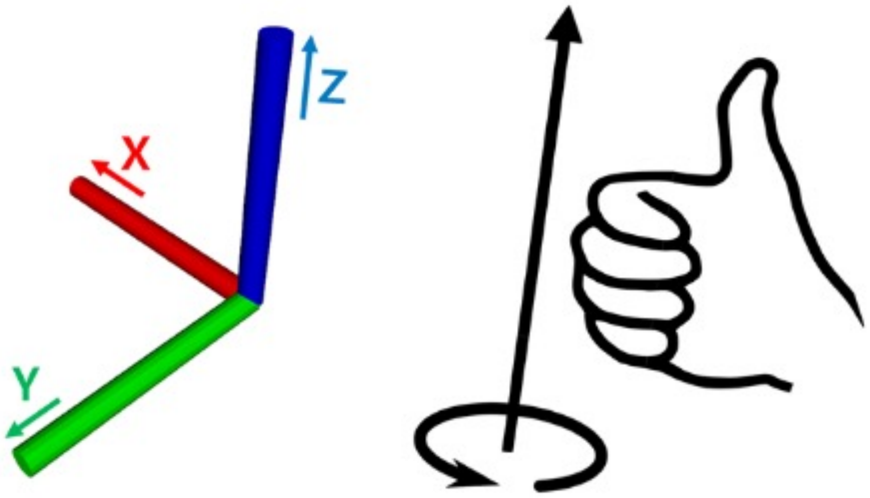


Example of params are map size/resolution and sensor configuration/settings.

ROS

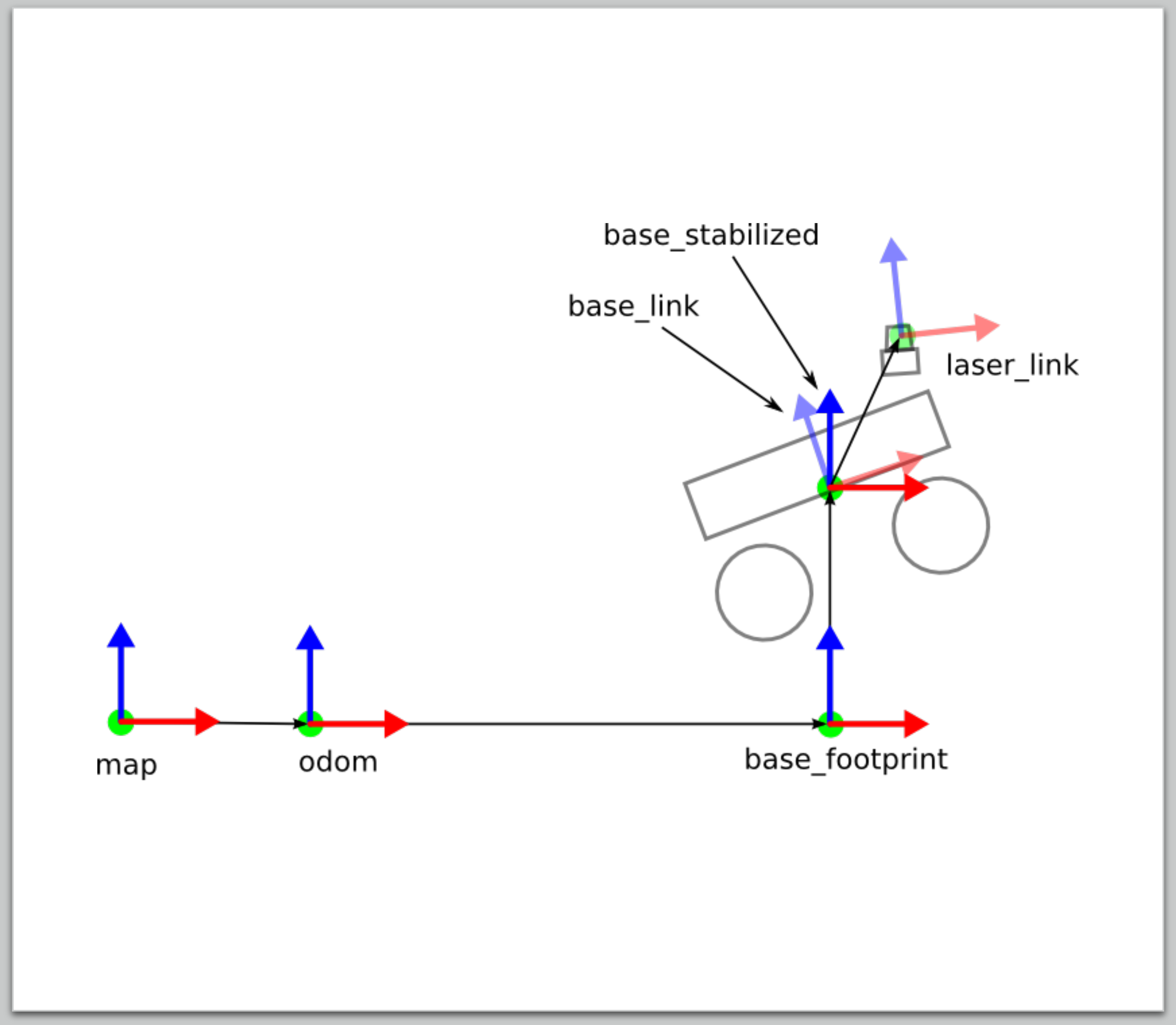
ROS *Transforms*

- in robotics programming, the robot's joints, or wheels with rotating axes, and the position of each robot through coordinate transformation are very important
- in ROS, this is represented by TF (transforms)
- TF are published with a mechanism similar to (and parallel) the one used for ROS Topics



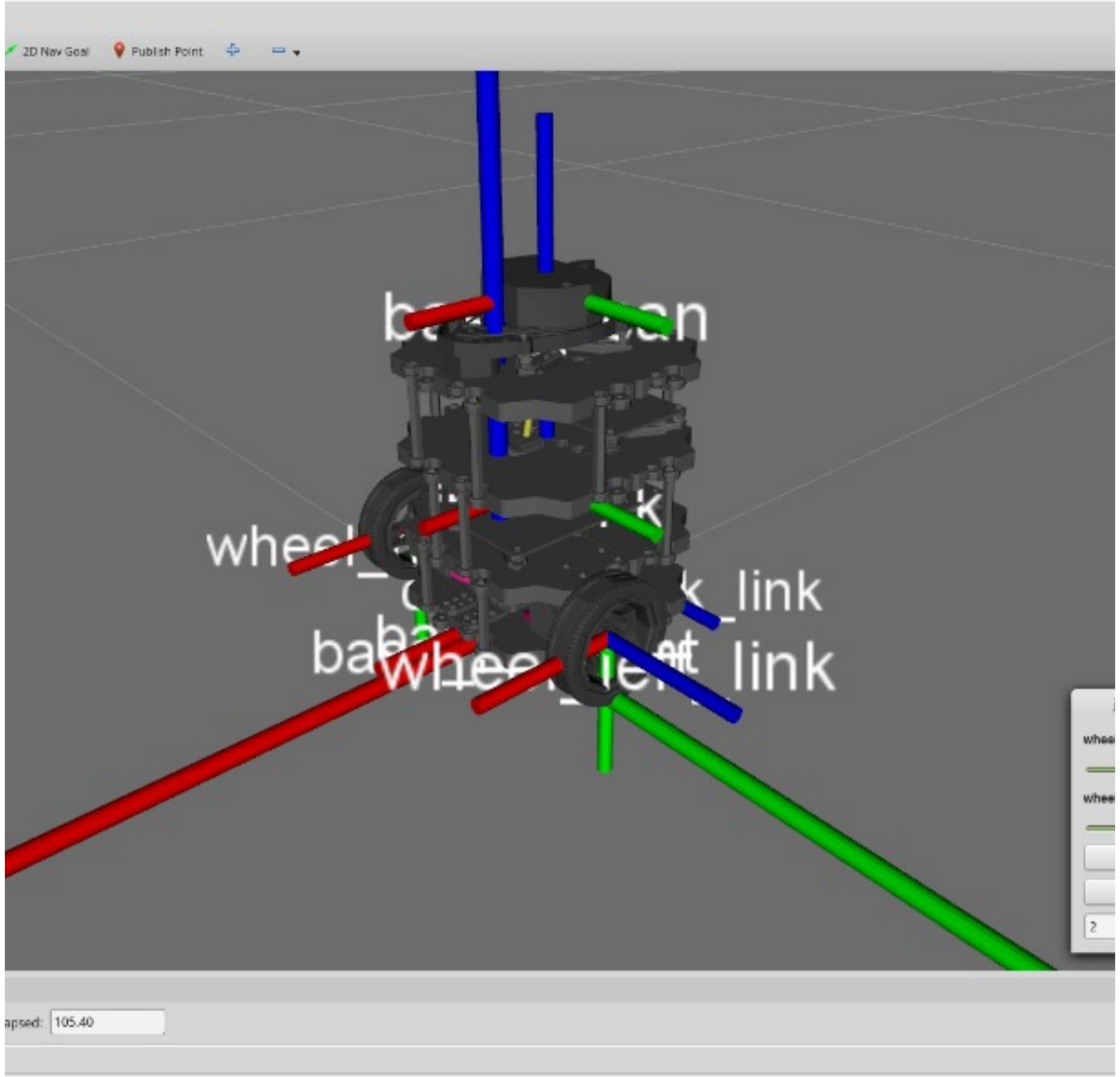
ROS *Transforms*

- all components of the robots should be connected through a chain of TF to a global reference frame (*world* or *map*)
- this is particularly important, as TFs allow the robot to project sensors onto a global reference frames



ROS *Transforms*

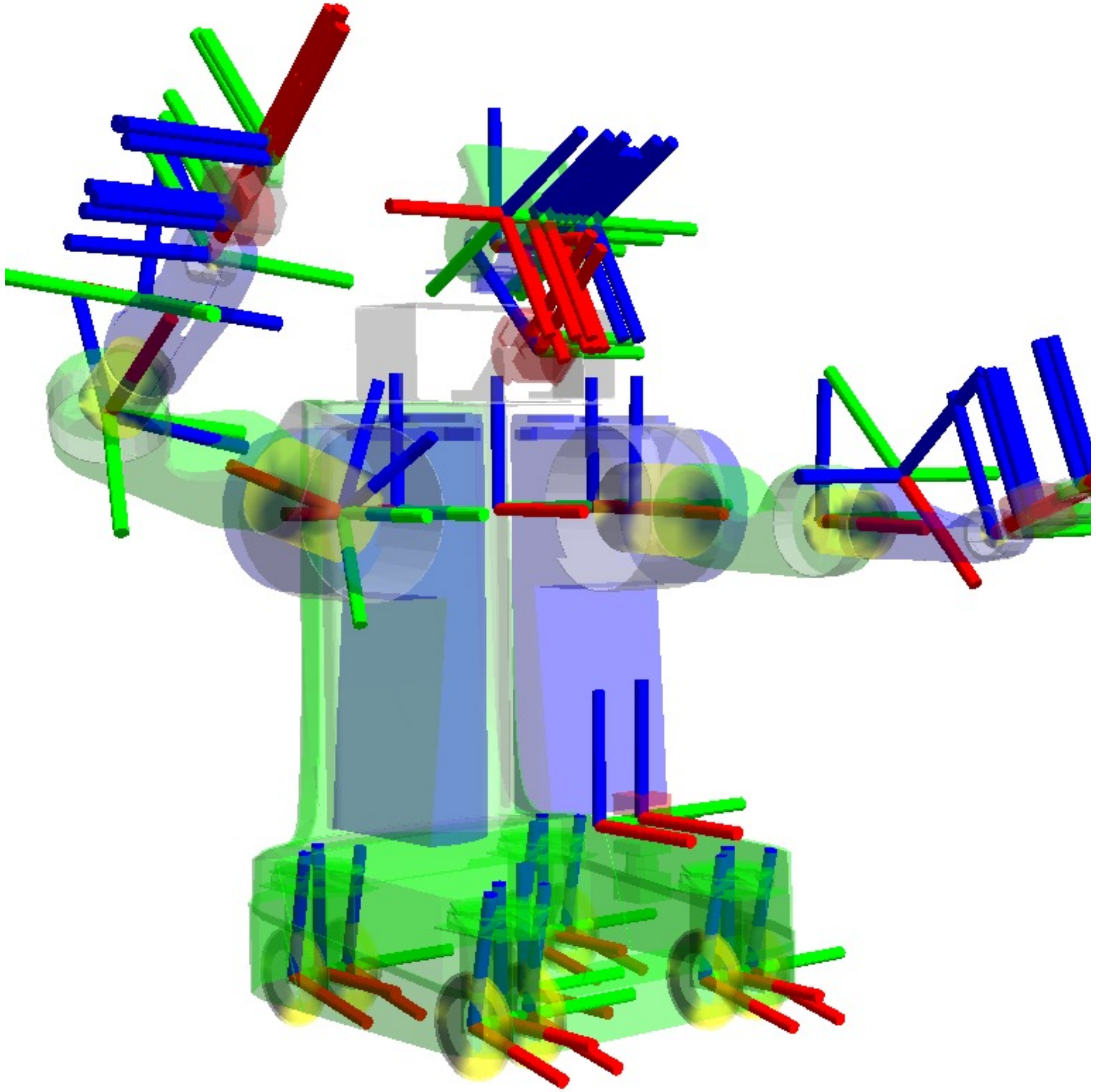
- some TF are static (e.g., the position of sensors w.r.t. The robot reference frame)
- some TF are dynamic and are computed real-time by nodes
  (e.g. the position of the robot in the map, the position of joints in a hand gripper)



ROS

ROS *Transforms*

- TF can become complex, especially for robot with a lot of Degrees Of Freedom (DOF) as grippers
- ROS provides visualization tools for controlling such aspects

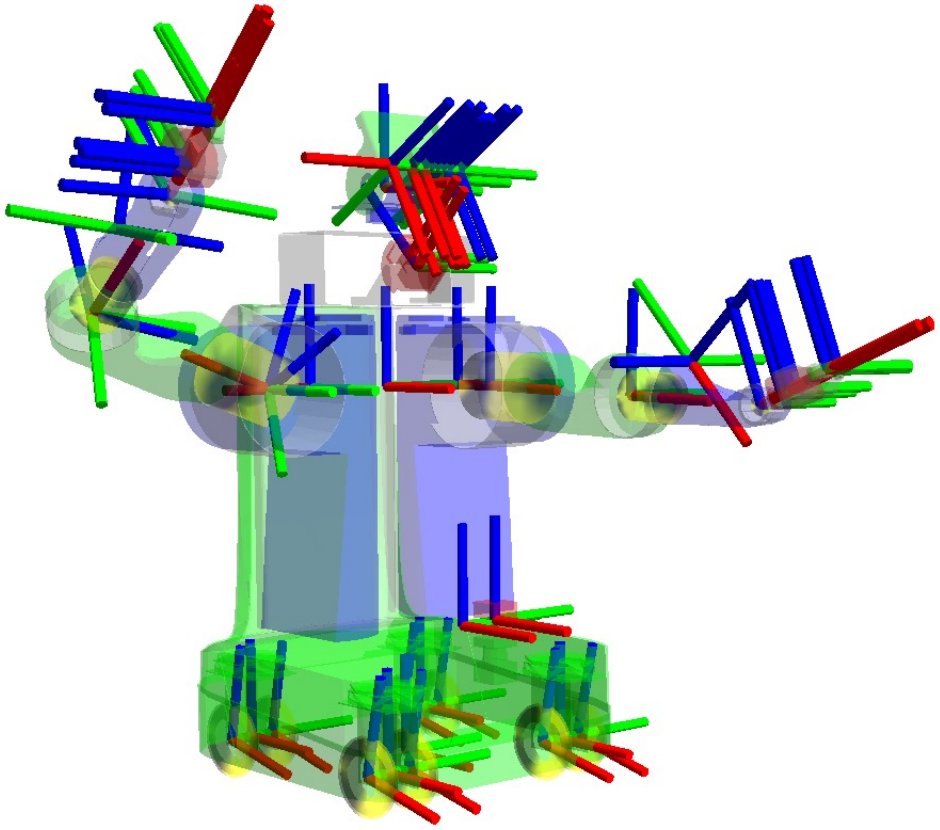


ROS

Developing toos

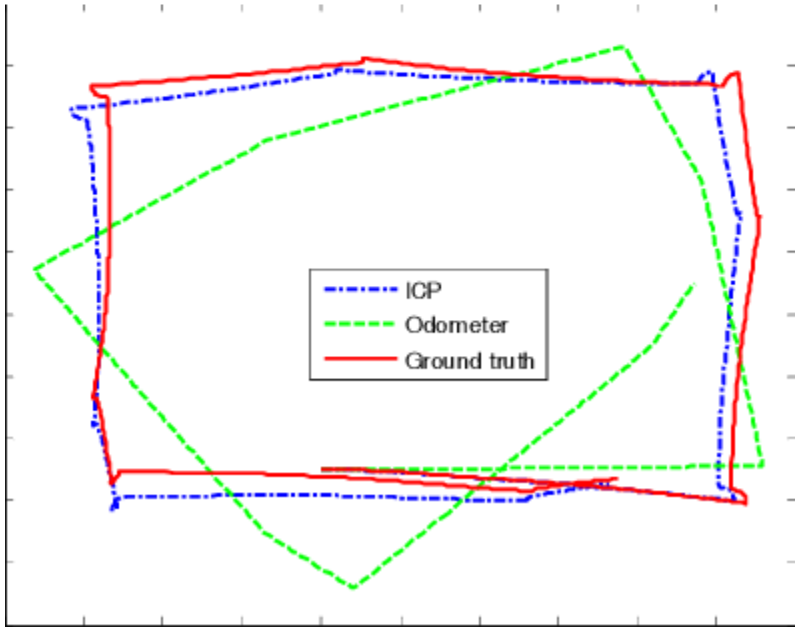
Developing a robot in ROS

- mobile robots easily became very complex objects
- issues can emerge with single components, hardware failures, integration, …
- impossible to control all possible sources of uncertainty



Environmental inaccuracies

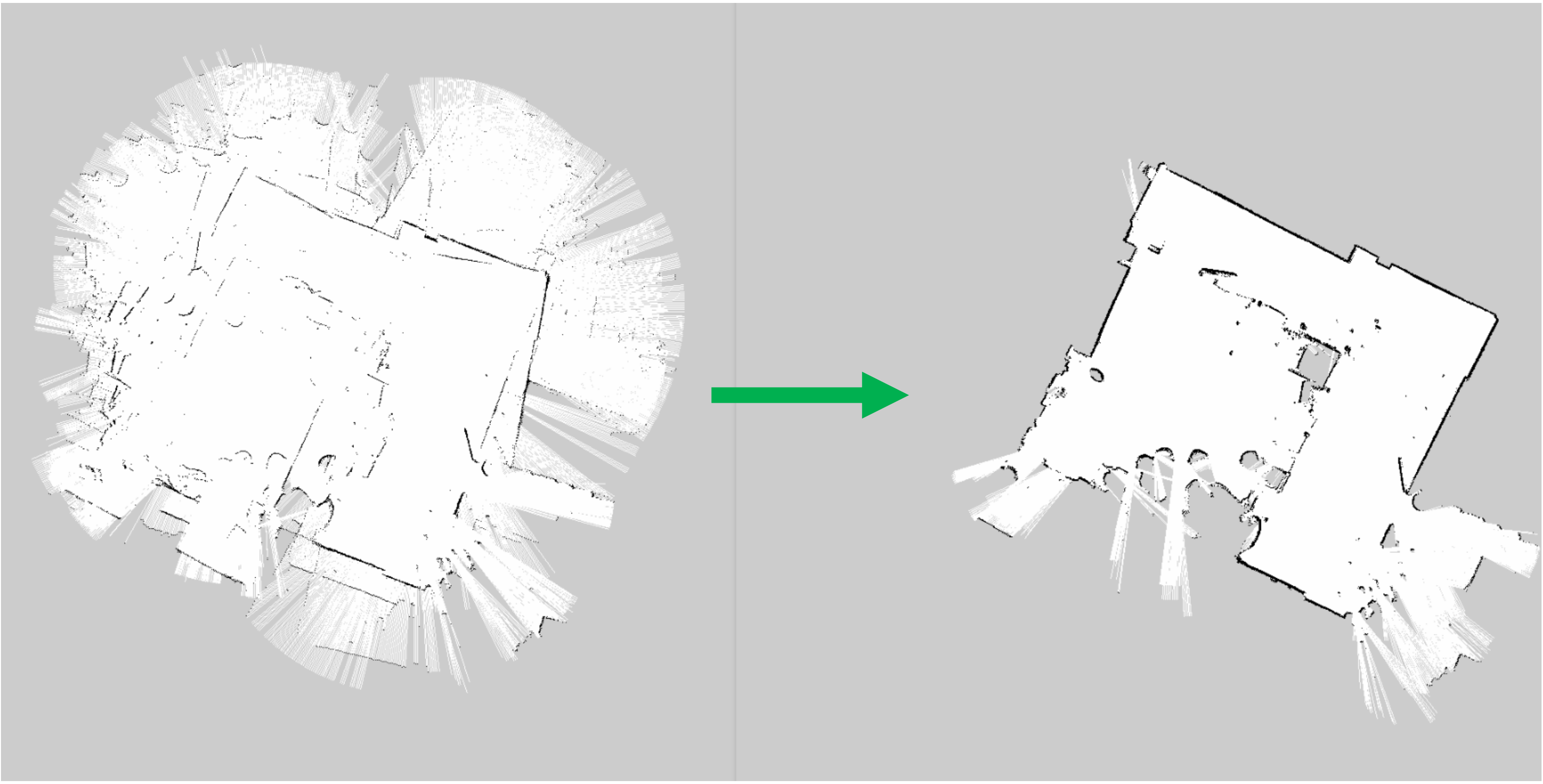
- All of the robot available knowledge is based on sensors but...
- ...sensors itself are (very) noisy
- odometry is the estimation of the robot motion from internal sensors (e.g. IMU or velocity)
- odometry itself is very noisy and unreliable



ROS

Reducing environmental inaccuracies

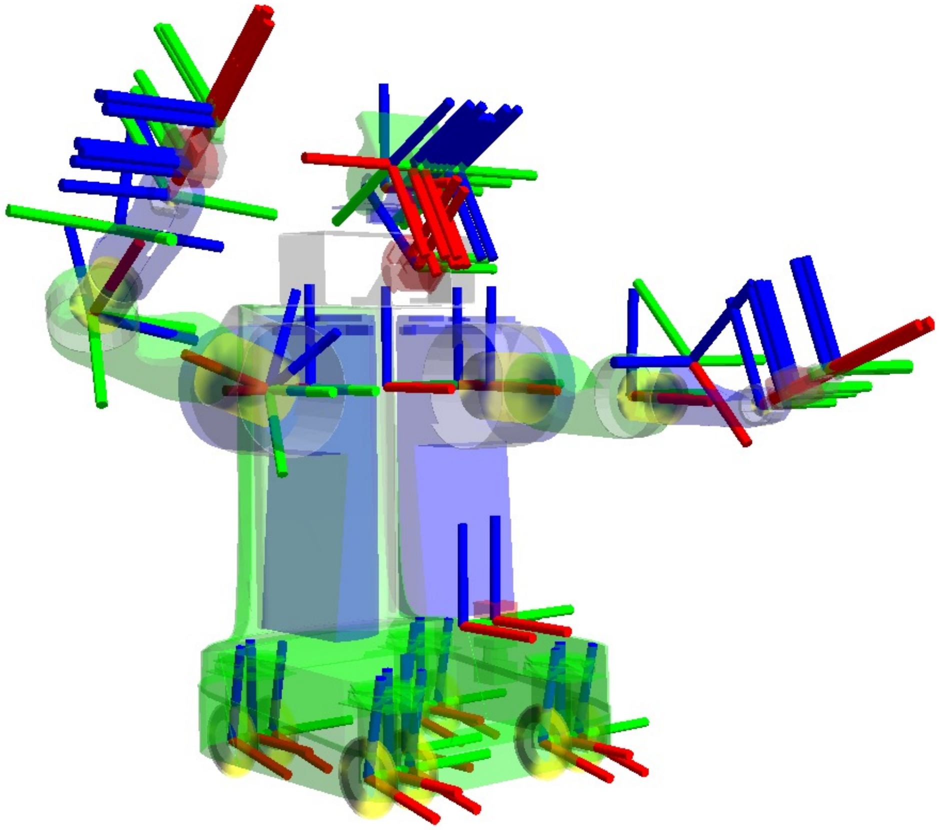
Even if assuming that there are no unexpected failures in the robot modules, some of the robot modules are designed to cope and reduces known sources of uncertainty and to integrate data together



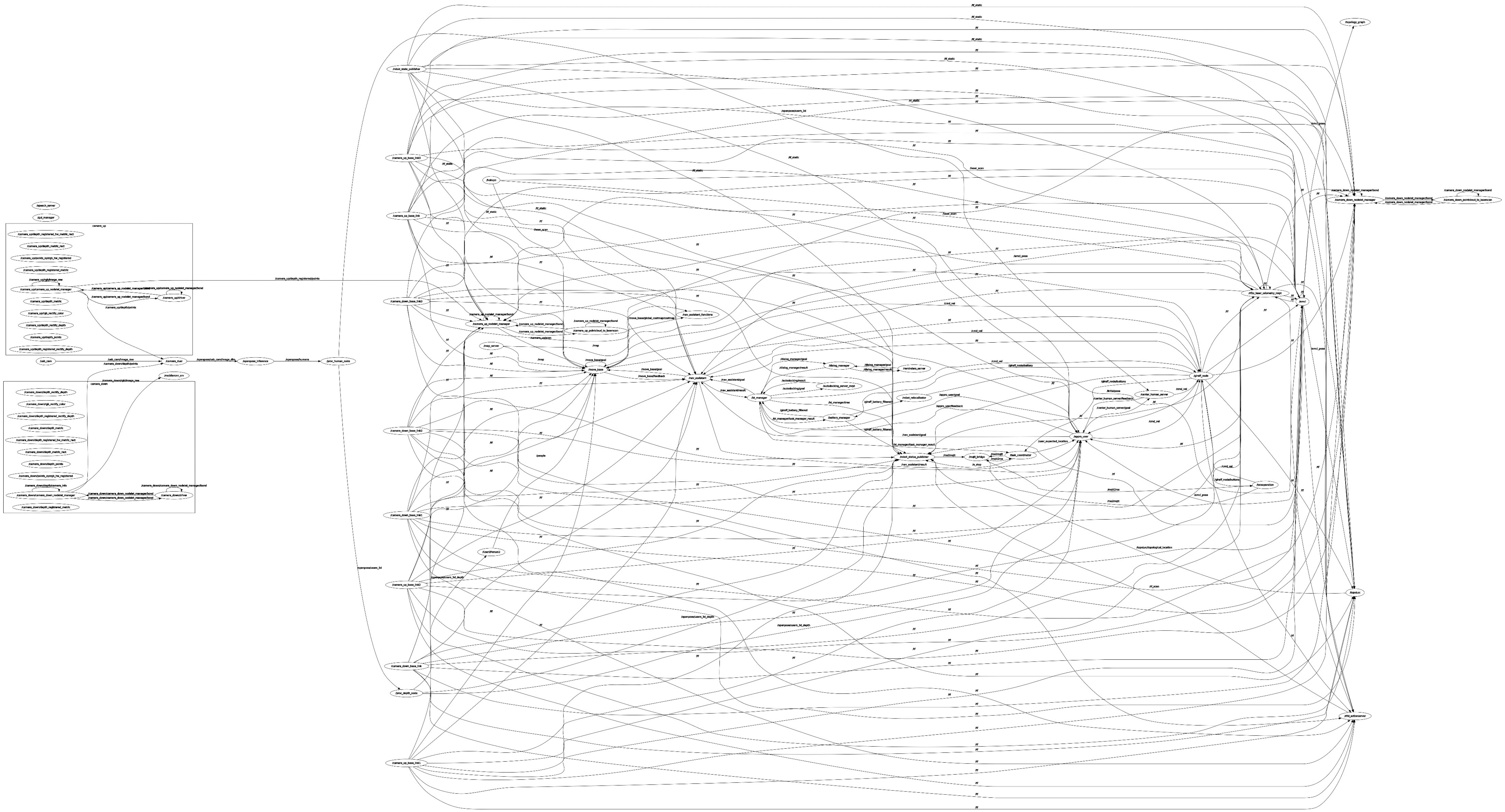
Mapping integrates sensor readings (e.g., laser range scanner) together reducing odometry error thus obtaining a valid map of the environment

Developing a robot in ROS

- Modularity and scalability of nodes and topics help in developing complex integrated system but…
- …still the resulting ROS computational graph is impossible to be analyzed at glance



ROS



The graph of ROS nodes and topics of a real robot

How to program robots then?

- A lot of components and modules integrated among them
- Sensors and robot hardware are noisy and can fail
- Impossible to control all possible sources of uncertainty

Making even a simple run with a robot can be very time consuming

ROS

How to program robots then?

- A lot of components and modules integrated among them
- Sensors and robot hardware are noisy and can fail
- Impossible to control all possible sources of uncertainty

Developing and integrating a new functionality into a pre-existing robot can be difficult too

Why ROS is useful

- A lot of components and modules integrated among them
- Sensors and robot hardware are noisy and can fail
- Impossible to control all possible sources of uncertainty

- Use packages provided by the community
- Split computation into nodes
- Test in advance in simulations
- Use pre-recorded sensor inputs
- Visual inspection tool for monitoring all of the robot aspects

ROS

Why ROS is useful

- A lot of components and modules integrated among them
- Sensors and robot hardware are noisy and can fail
- Impossible to control all possible sources of uncertainty

- Use packages provided by the community
- Split computation into nodes
- Test in advance in simulations
- Use pre-recorded sensor inputs
- Visual inspection tool for monitoring all of the robot aspects

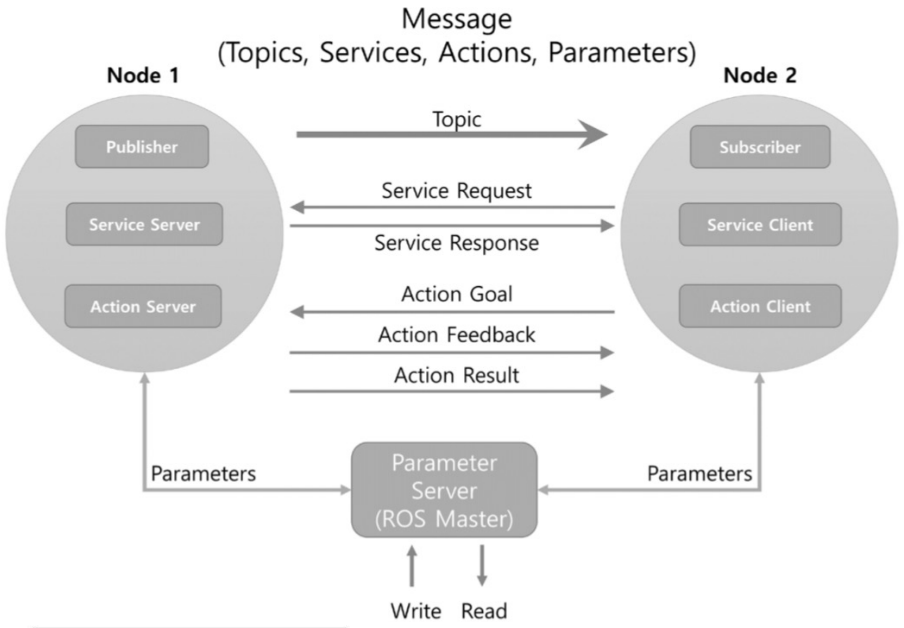
Why ROS is useful

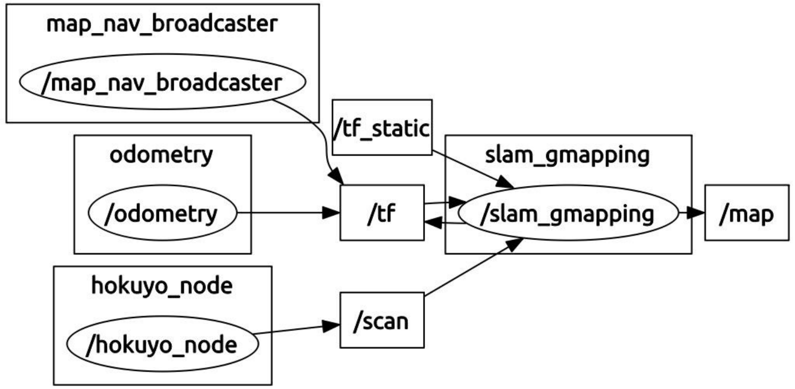
- A lot of components and module integrated among them
- Sensors and robot hardware are noisy and can fail
- Impossible to control all possible sources of uncertainty

- Use packages provided by the community
- Split computation into nodes
- Test in advance in simulations
- Use pre-recorded sensor inputs
- Visual inspection tool for monitoring all of the robot aspects

ROS core concepts

- Support for all robot stack (low level motor controls to AI)
- Modularity and scalability
- Cross platform - distributed
- Repositories and code reusability
- Divide-et-impera
- Computational graph using *Nodes* and *Topics*





Why ROS is useful

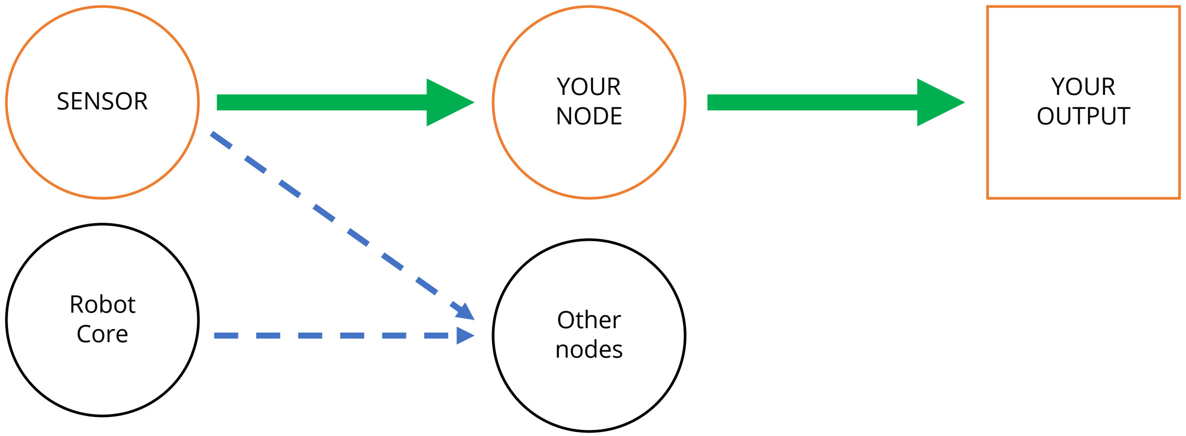
Problems

- A lot of components and module integrated among them
- Sensors and robot hardware are noisy and can fail
- Impossible to control all possible sources of uncertainty

Solutions

- Use packages provided by the community
- Split computation into nodes
- Test in advance in simulations
- Use pre-recorded sensor inputs
- Visual inspection tool for monitoring all of the robot aspects

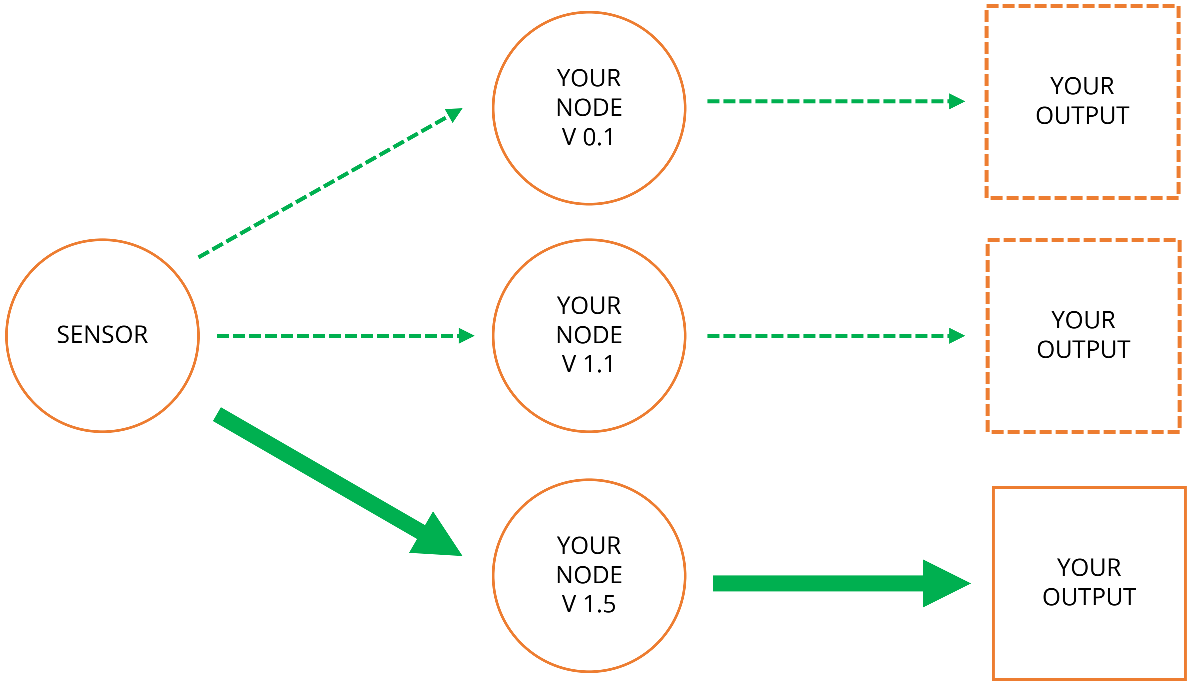
An example: writing your own *Node*



Assume that you have to implement an algorithm for a robot, e.g. a module that <u>detects narrow passages</u> that are challenging for the robot navigation.

ROS allows you to develop just your node while using a pre-built robot set up (from the community) and to use pre-existing robot functionalities (remote commands, mapping, odometry, sensors parsing, mapping, localization)

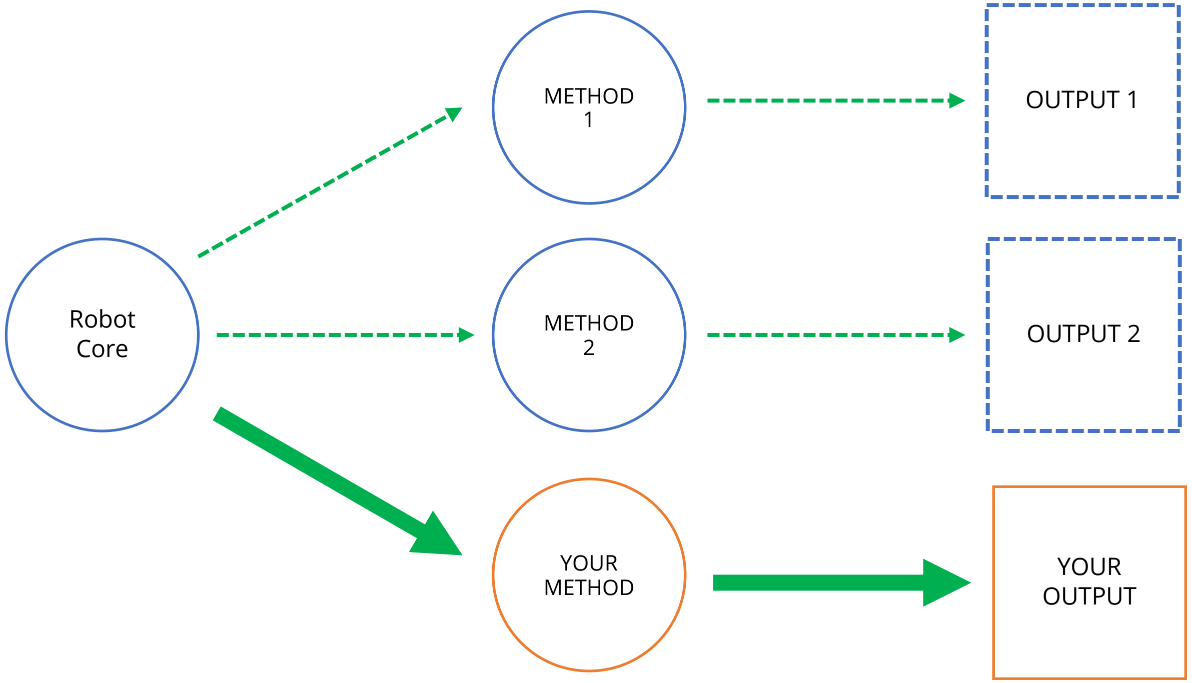
An example: writing your own *Node*



Probably you will develop several version of your node. The first one will have bugs and wont work, then a new version is produced with improvements, ...

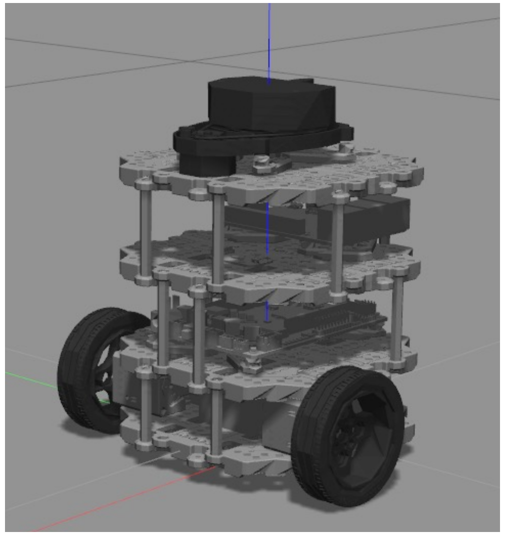
... and testing the result of different version together could be a good idea

An example: writing your own *Node*



Using *nodes* and *topics* it is also straightforward to test several methods (to see what is more useful for your robot) or to compare the results of your method with the one available to the community (and release it).

ROS and Simulations

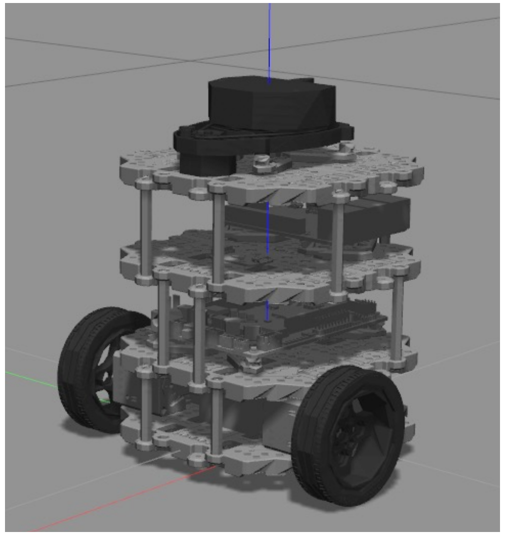


One of the most powerful tool that ROS have is the possibility to use integrated 2D and 3D ROS simulations.

ROS simulation nodes replaces sensor drivers and allows to test the same algorithm with real robot and simulations

ROS

ROS and Simulations



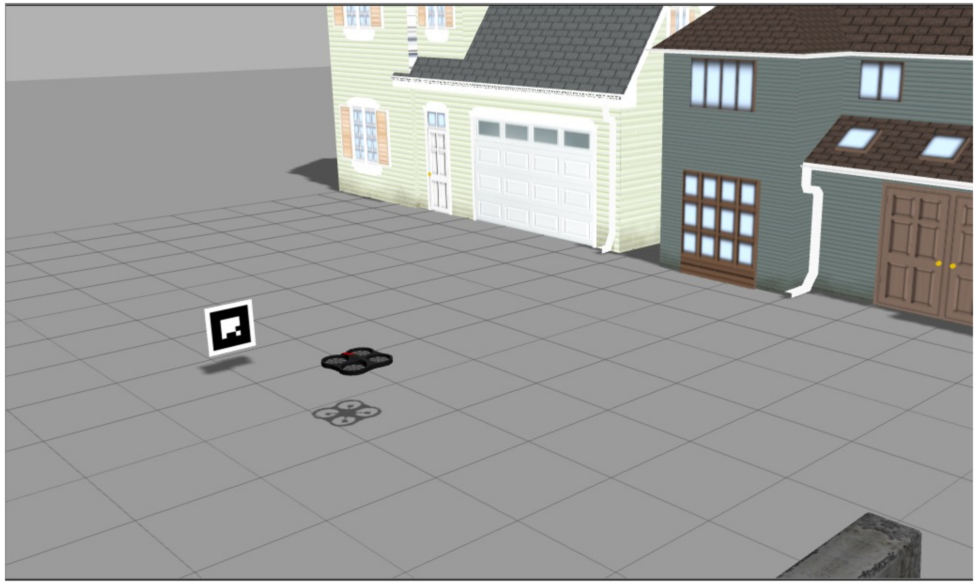
(iterative)
development of
a new method in simulations

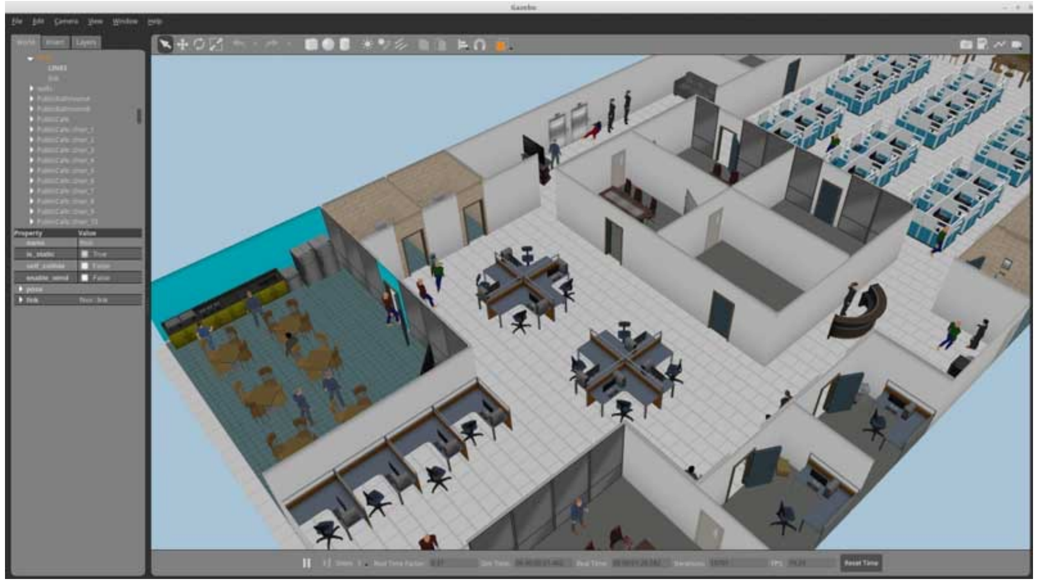
Test and validation with real robots

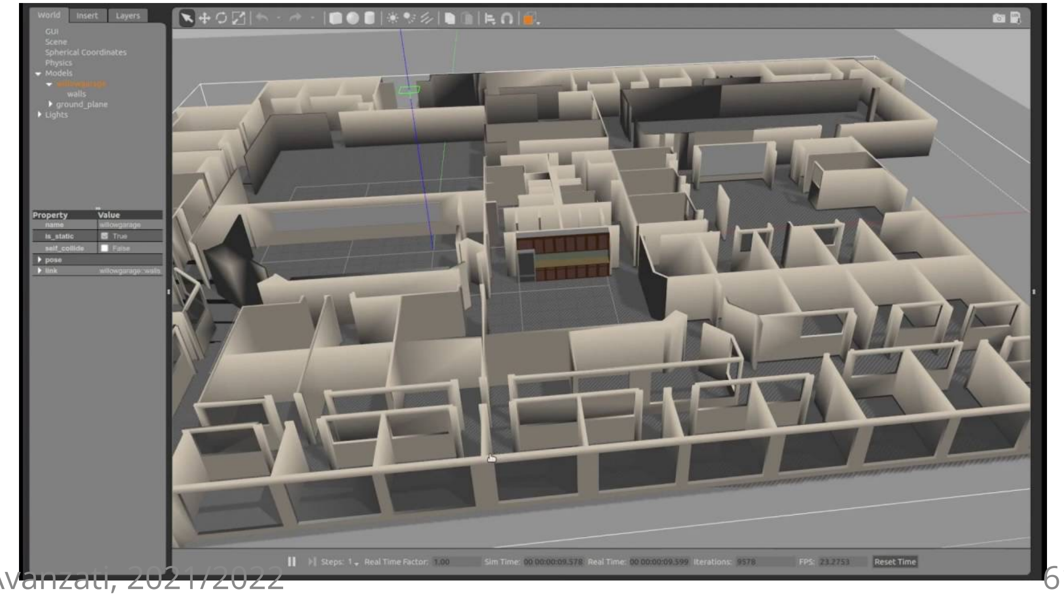
- Robots in ROS simulations are modeled starting from their real counterpart.
- This allow a fast transitioning from tests performed with simulation and with real robot just changing a few lines of code.

Simulations with Gazebo

Gazebo (3D) is the most popular and used ROS simulation tool, and it allows to simulate mobile robots, UAVs, manipulators, indoor and outdoor environments, …

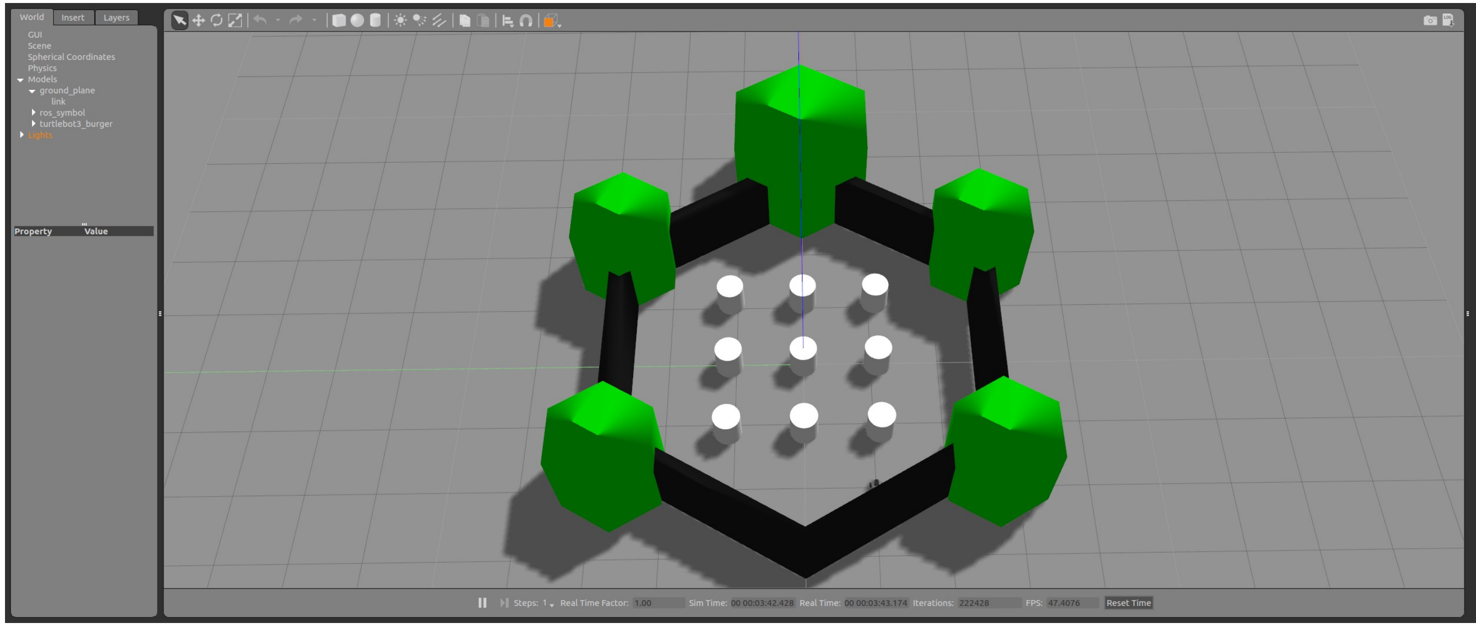


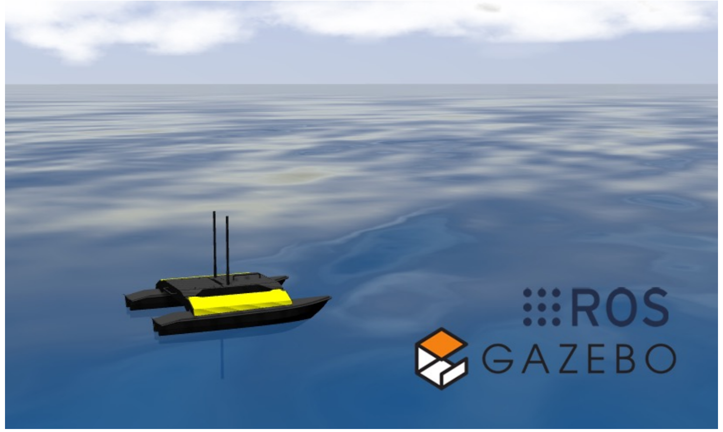


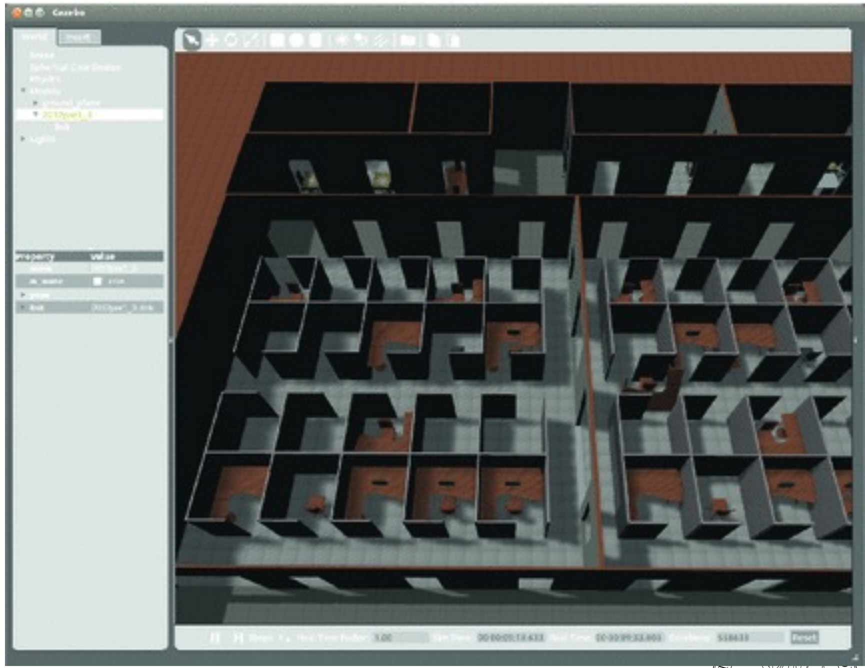


ROS Gazebo

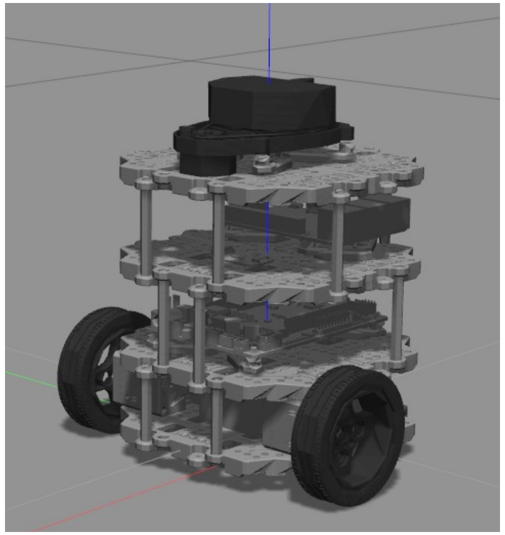
- ROS-Gazebo is installed with ROS
- You can use publicly available *worlds* or create your own with a GUI / editor (and importing 3D models)
- You can simulate toy block-world environments and complex large-scale ones, with dynamic component
- You can simulate both mobile robot and grippers







ROS Gazebo



(iterative)
development of
a new method in simulations

Test and validation with real robots

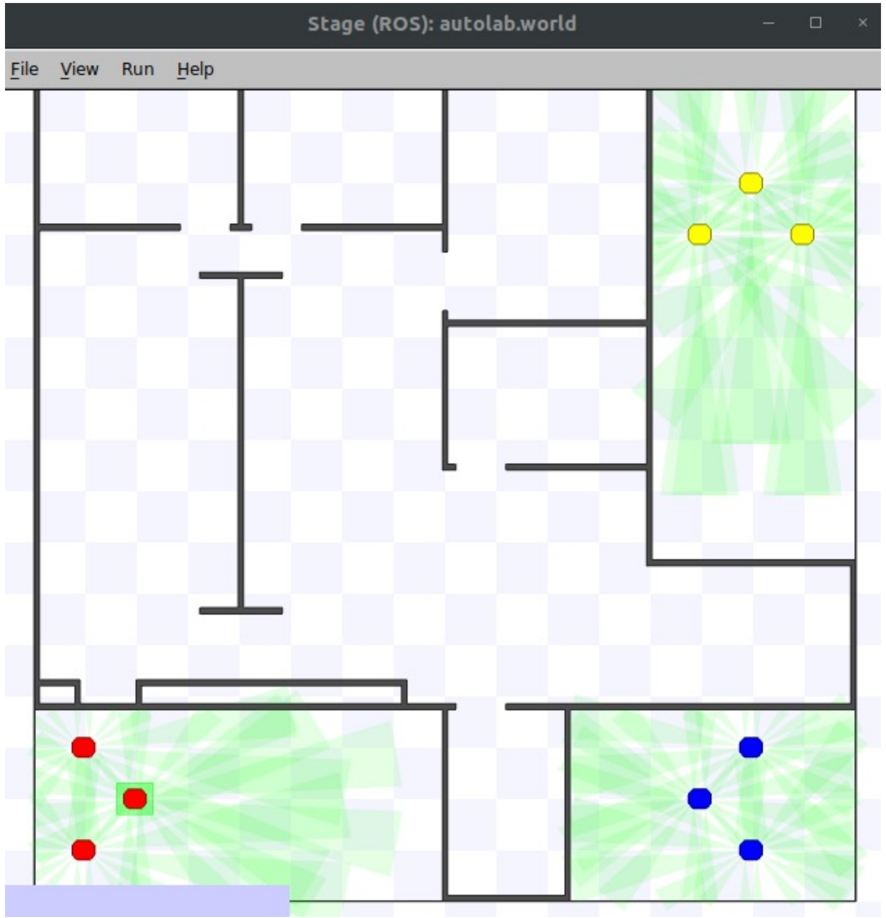
- Most robot provide a pre-configured ROS Gazebo model
- You can experiment using simulation without the need to configure the robot and the required packages and later transition to the real robot (if needed)
- They also provide documentation and tutorials

ROS

Simulations with STAGE

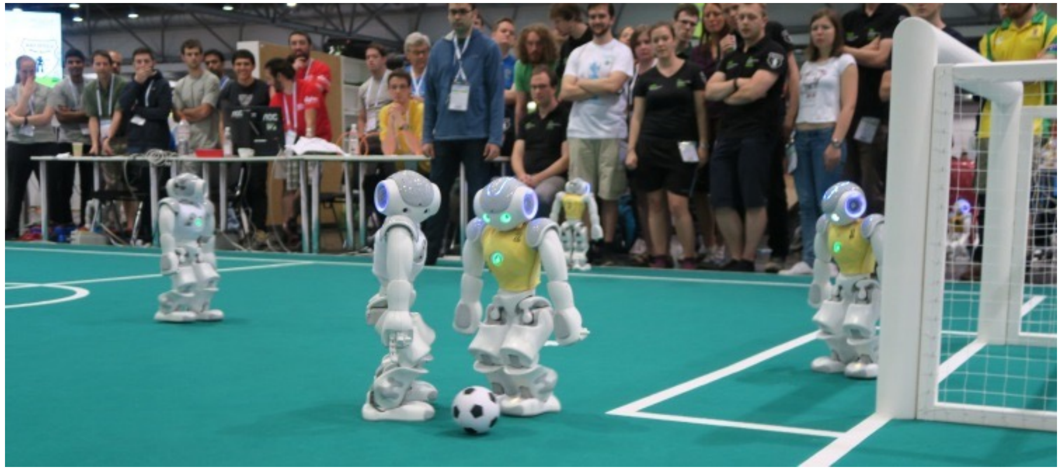
Stage is a simple 2D robot simulator.
It is useful for testing multi-robot systems, swarm robotics (even 10k robots) and for testing robotic tasks that require a higher level of abstraction.

Besides Gazebo and Stage, ROS can work with many commercial and open-source robotic simulation tools.



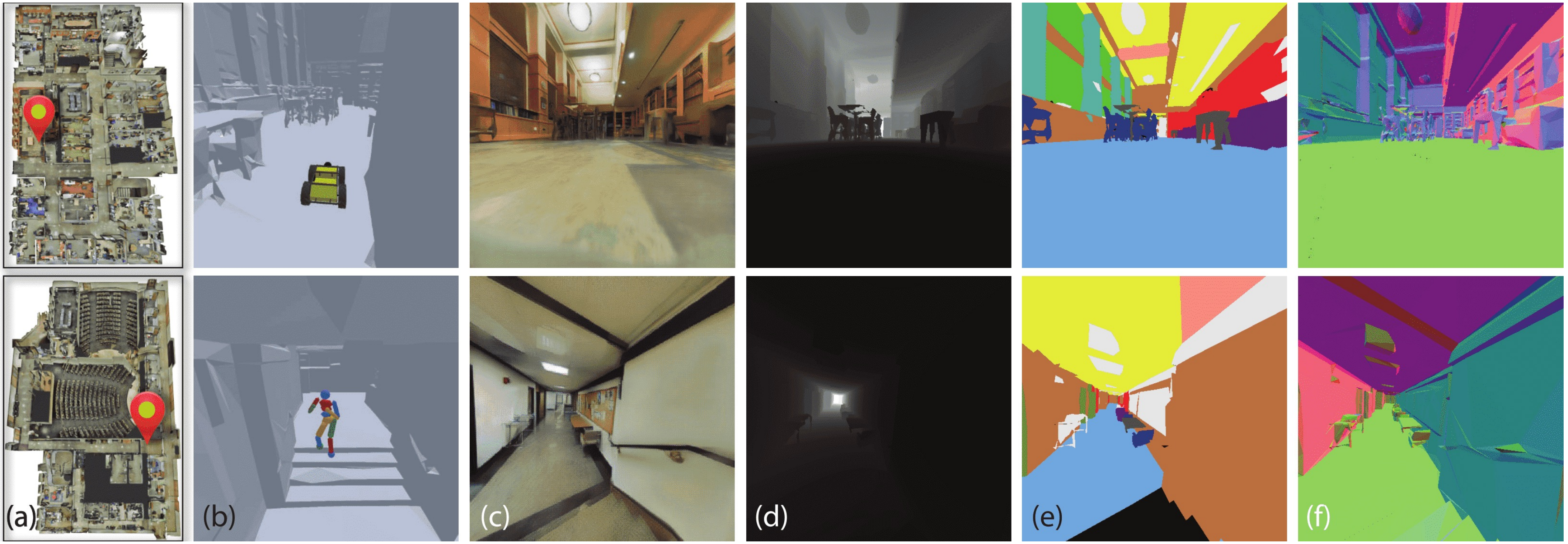
From simulations to reality

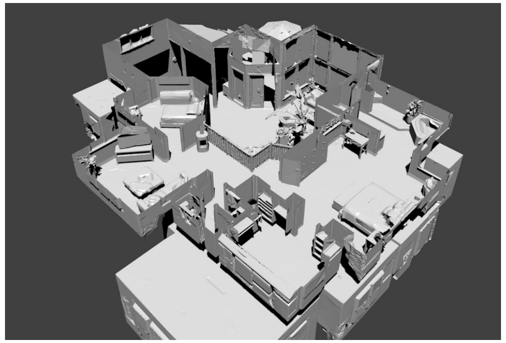
- Simulations are currently used to test complex robot behaviours
- The more robots have physical limitations (e.g. humanoid), the more simulations are used
- However, how to transfer capabilities trained in simulations to real-world-robots is still an open question



Realism in simulation

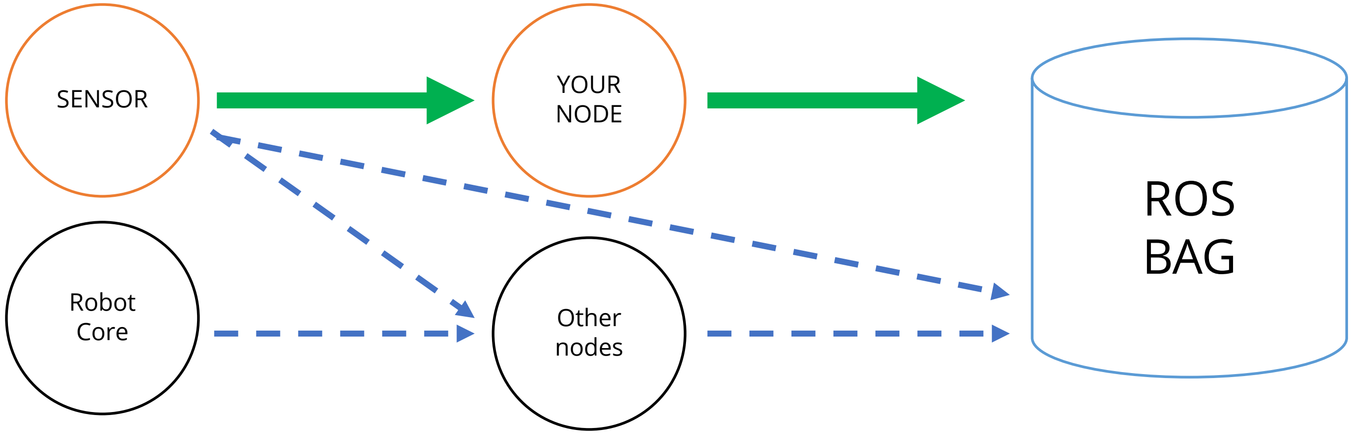
- Simulated environments look different from real ones
- Can we train/test vision-based methods in simulation? (online)
- Solution: use visually realistic simulation from real-world model





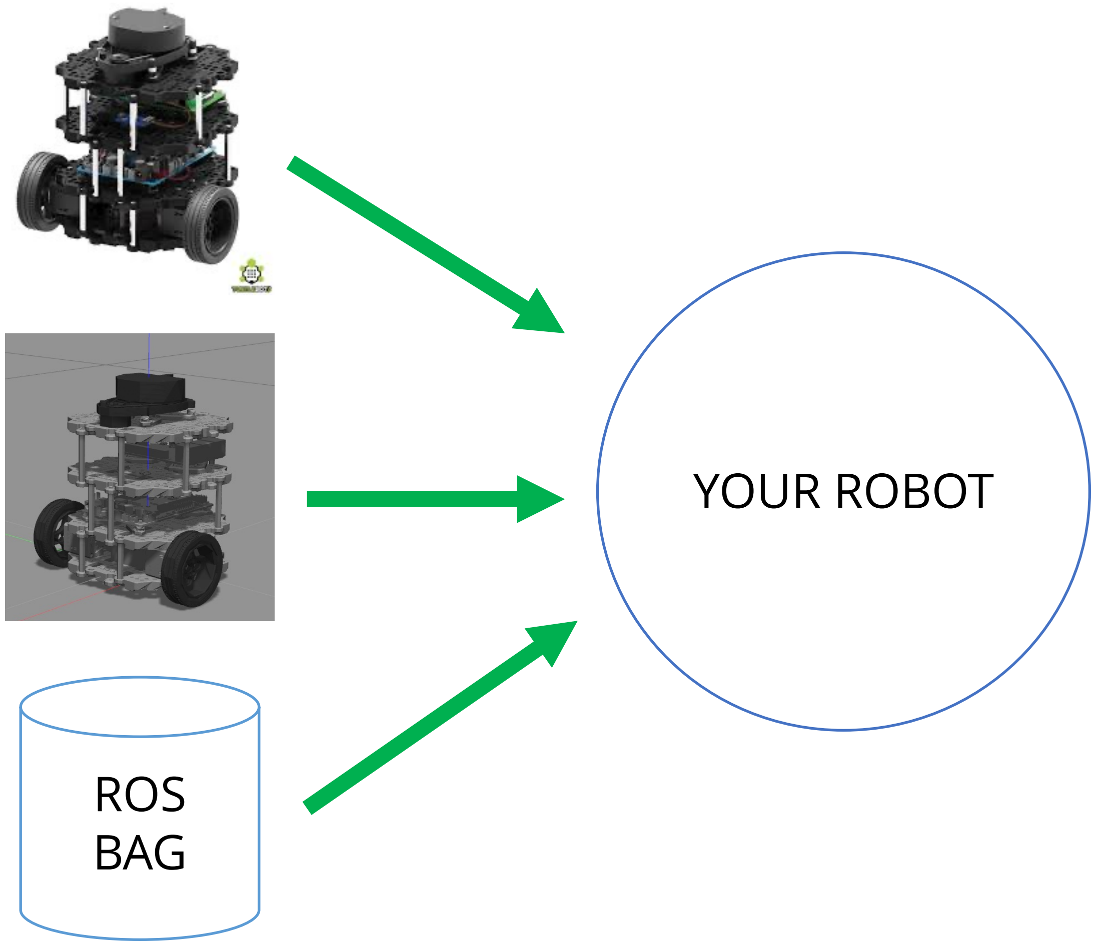
http://gibsonenv.stanford.edu/ with ROS

Another solution – use datasets



- Another important tool embedded in ROS is the possibility to record robot runs (in simulations or with real robots and to replay them).
- **ROS bags** store a time-stamped serialized version of all selected topics (sensors, nodes outputs, …)
- Different algorithms can be tested with the same input to test improvements
- Bags can be reproduced at 2x, 4x, 5x → speed up of development times
- Publicly available datasets are shared among the community

Wrapping up



With the only constraint of using the same topics and the same msg format, you can switch between:

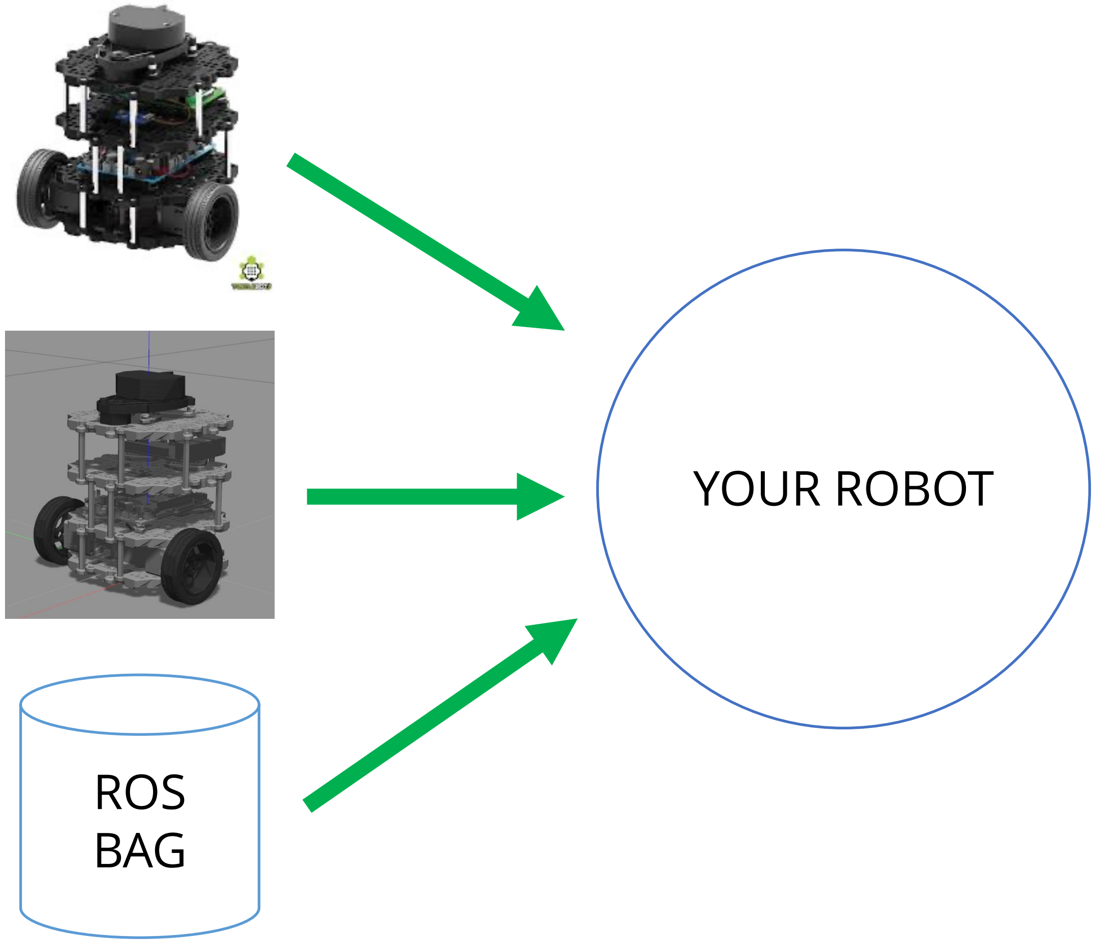
- real robots
- simulations
- pre-recorded data streams

without changing the other part of the robot code.

The rest of the robot code structure and of the nodes used remain the same.

ROS

Wrapping up

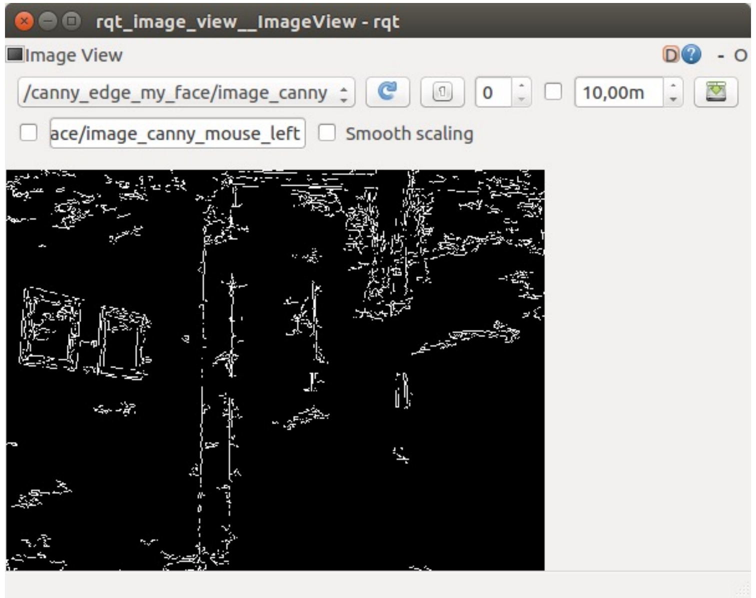


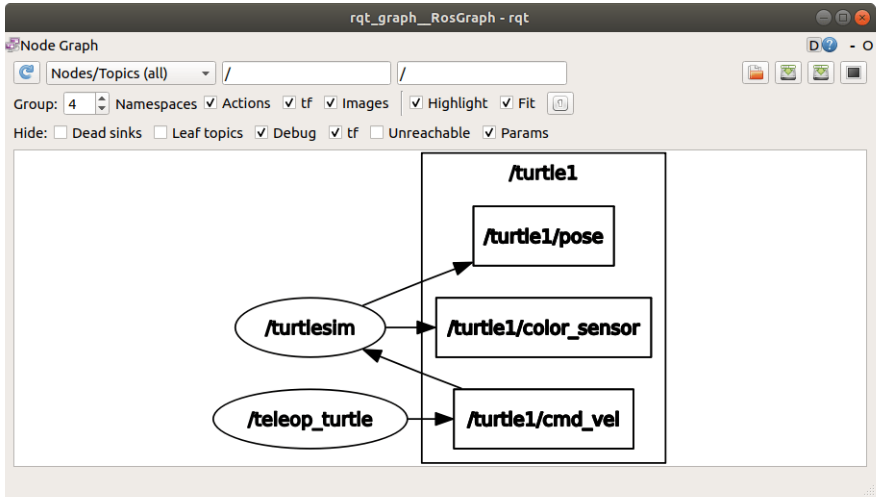
This has multiple (positive) side effects:

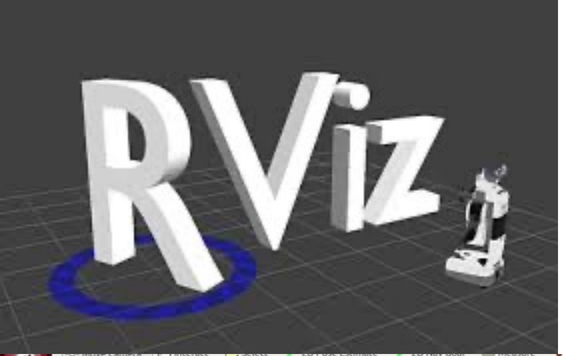
- You can focus on the specific task you want to develop
- You can develop a robot even without having a robot – use simulations
- You are not even forced to acquire no runs – just use datasets/rosbags

Debugging tool with ROS

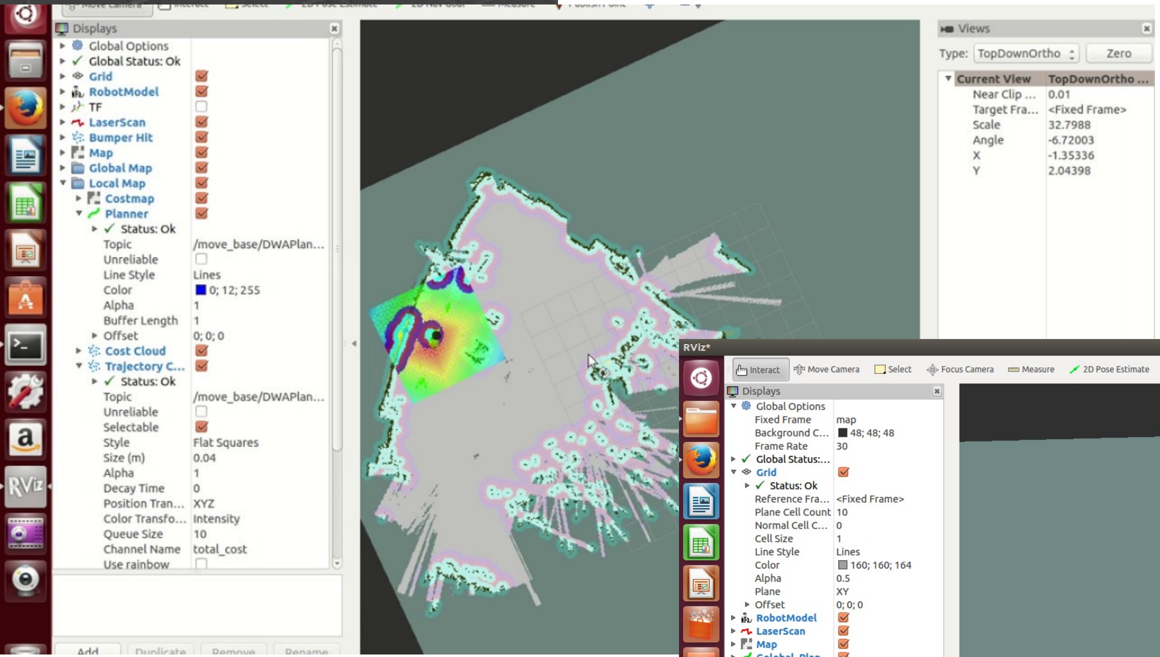
Even using simulations or ROS bags, robots are still complex. ROS offers several visualization tools that are useful for debugging of a complex system.

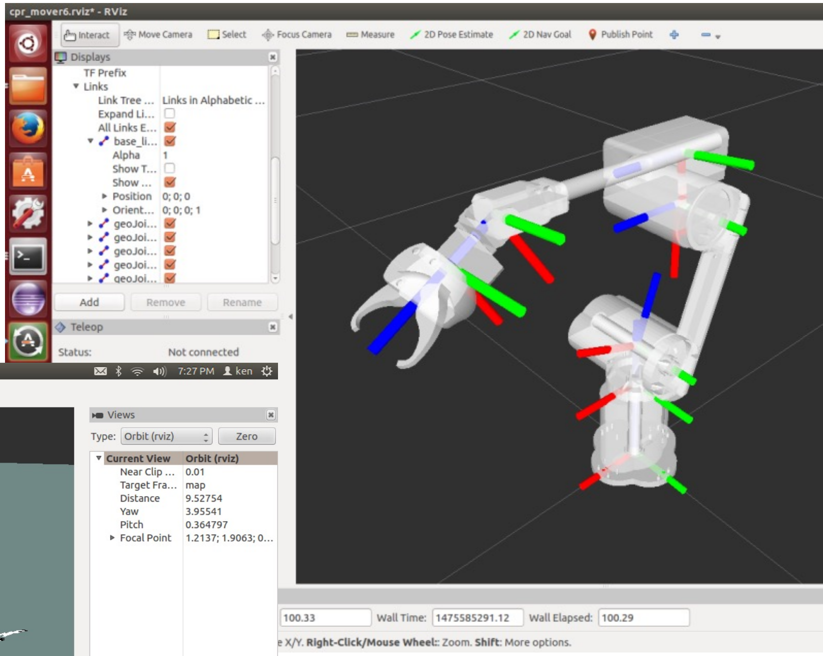


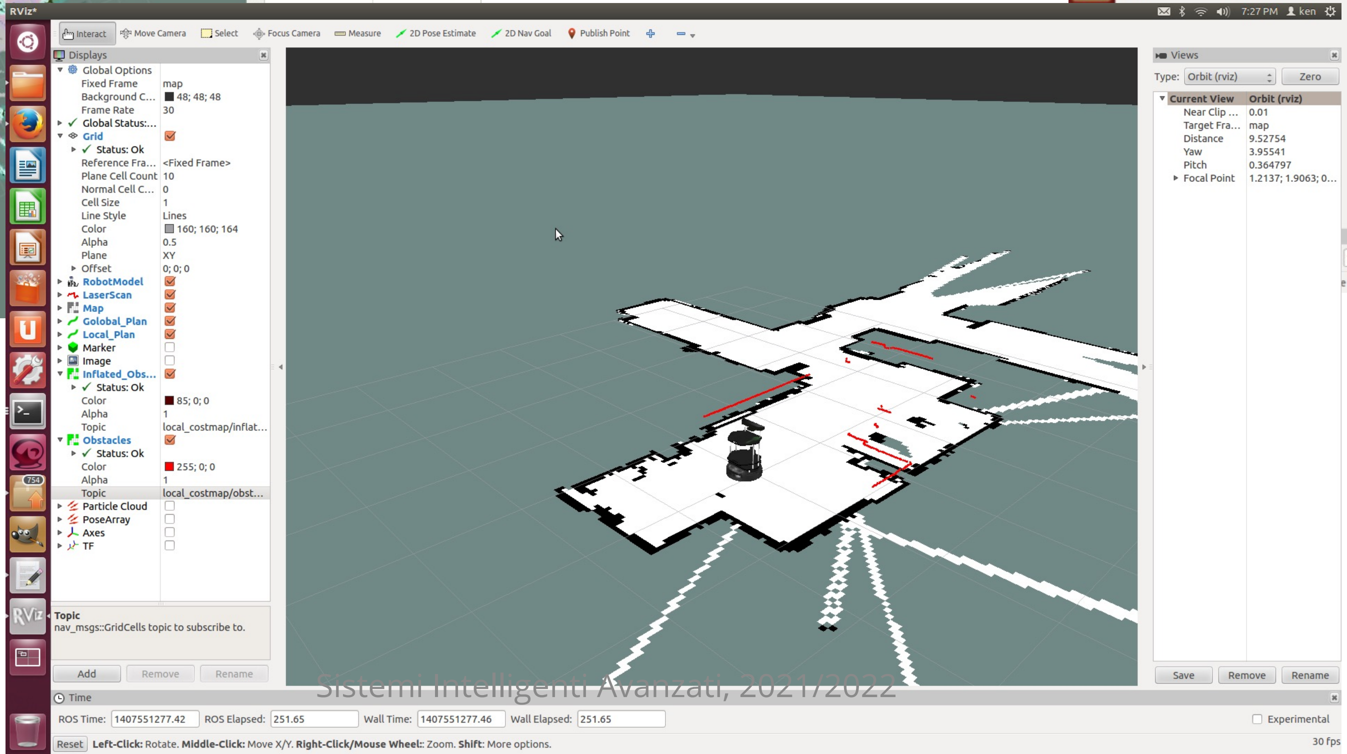


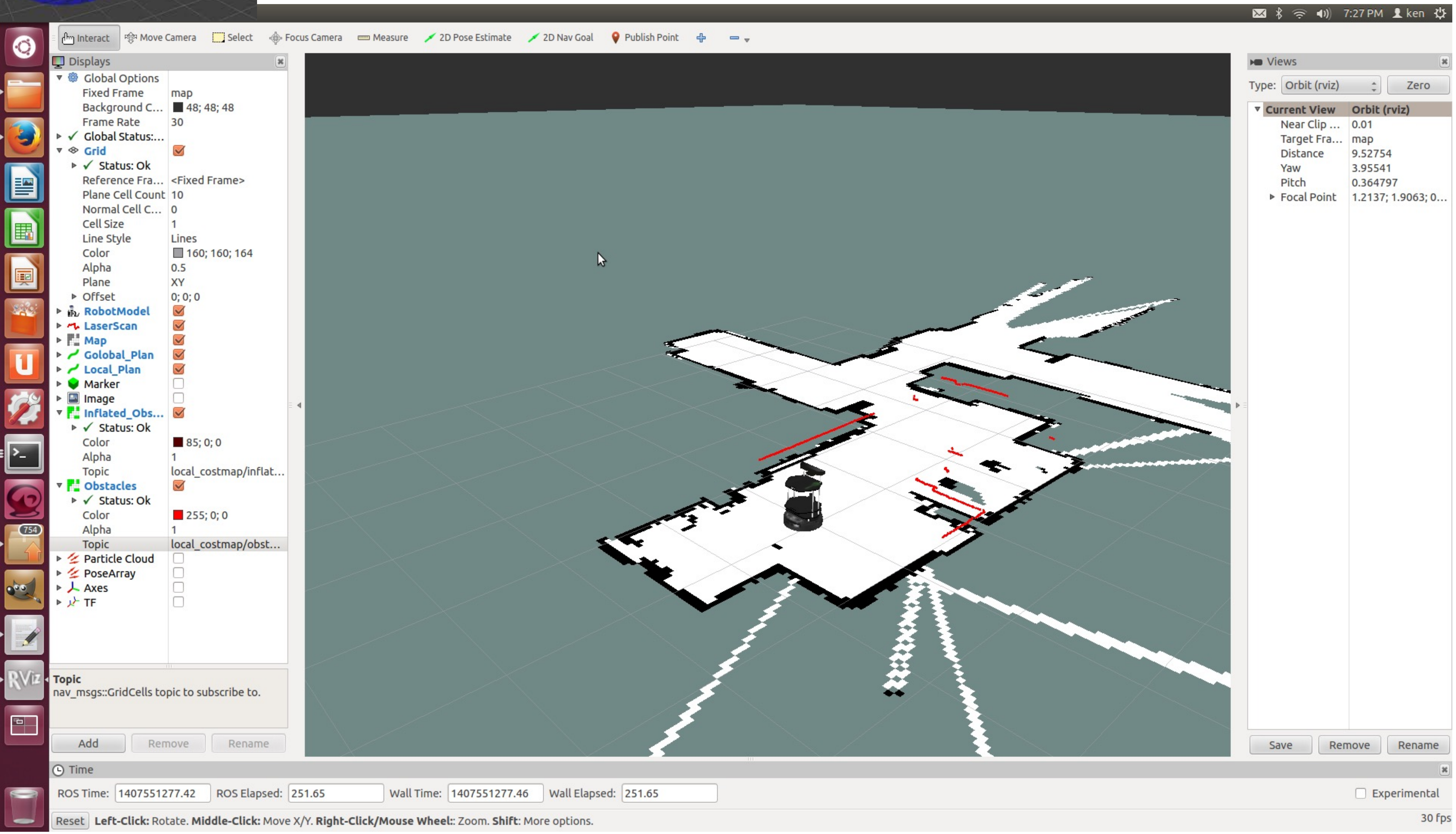


RViz is the main visualization tool of ROS.
It is used to display sensor readings, maps, cost maps, joints, TF, and, in general, to have an overview of the internal status of the robot and of all its sensors and nodes.









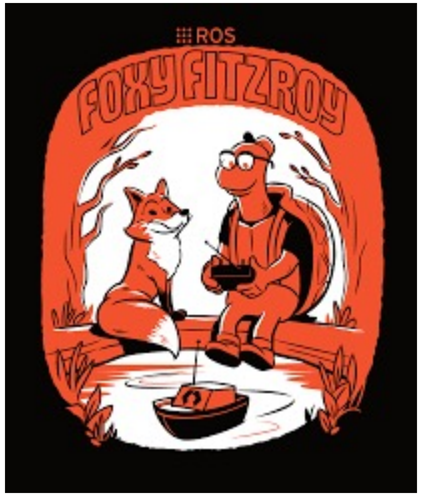
Rviz is "de-facto" a debugging tool for ROS.

- visualize sensor readings
- visualize maps
- check topic naming
- send robot to location
- see robot status (localization)
- check TF
- …

How to start with ROS

ROS installation

- ROS can be installed with multiple OS, but the simplest way for starting is using Ubuntu
- Each version of Ubuntu has its own ROS distro:
  - Ubuntu 16.04 → ROS Kinetic
  - Ubuntu 17.04 → ROS Lunar
  - Ubuntu 18.04 → ROS Melodic
  - Ubunto 20.04 → ROS Noetic
- ROS 2.0 is here– ROS2 Foxy Fitzroy available
- Stick with a LTS (ROS Noetic or ROS Melodic)
- A VM - docker is fine too for starting
- Whatever you do with simulated robots you can do with real ones later



ROS installation

- Follow guide at http://wiki.ros.org/ROS/Installation
- basically installation on Ubuntu is:
  sudo apt install ros-noetic-desktop-full
  and wait
- then follow the basic tutorials (2/3h tops)
  http://wiki.ros.org/ROS/StartGuide
  http://wiki.ros.org/ROS/Tutorials → learn how to use ROS tool for compiling, adding packages,
- and you are ready to go, e.g. with simulated robots.

ROS command line tools

- read and publish messages on topic
- have a list of all active services/nodes/topics/params
- find packages and folders
- compilation tools
- edit tools
- check tf
- check frequency of nodes
- …

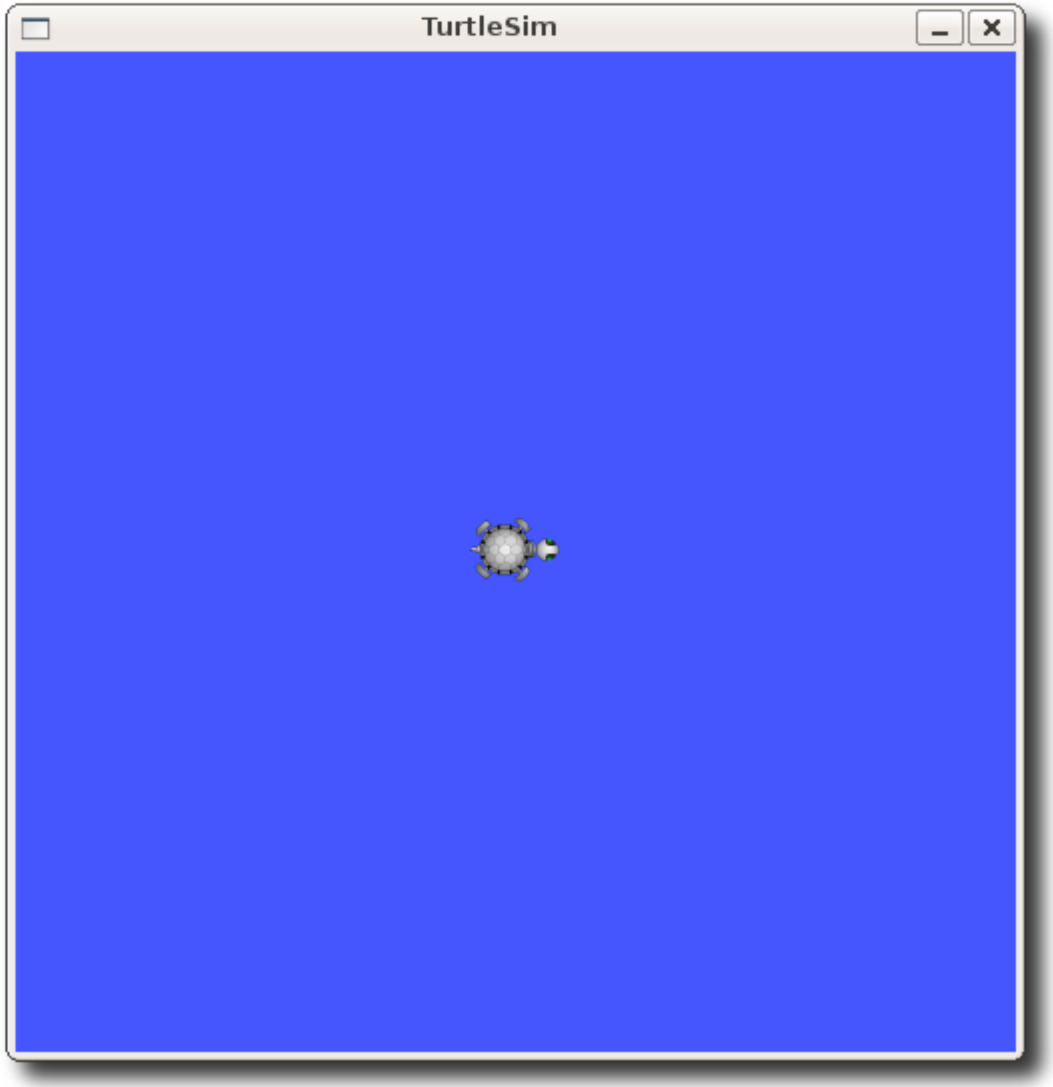
A simple example: turtlesim

TurtleSIM

Simulation tutorial of ROS:

- 2D world simulation
- a “turtle” “robot”
- can receive commands and move around

Easy to understand ROS topics and messages + command line tools



TurtleSIM

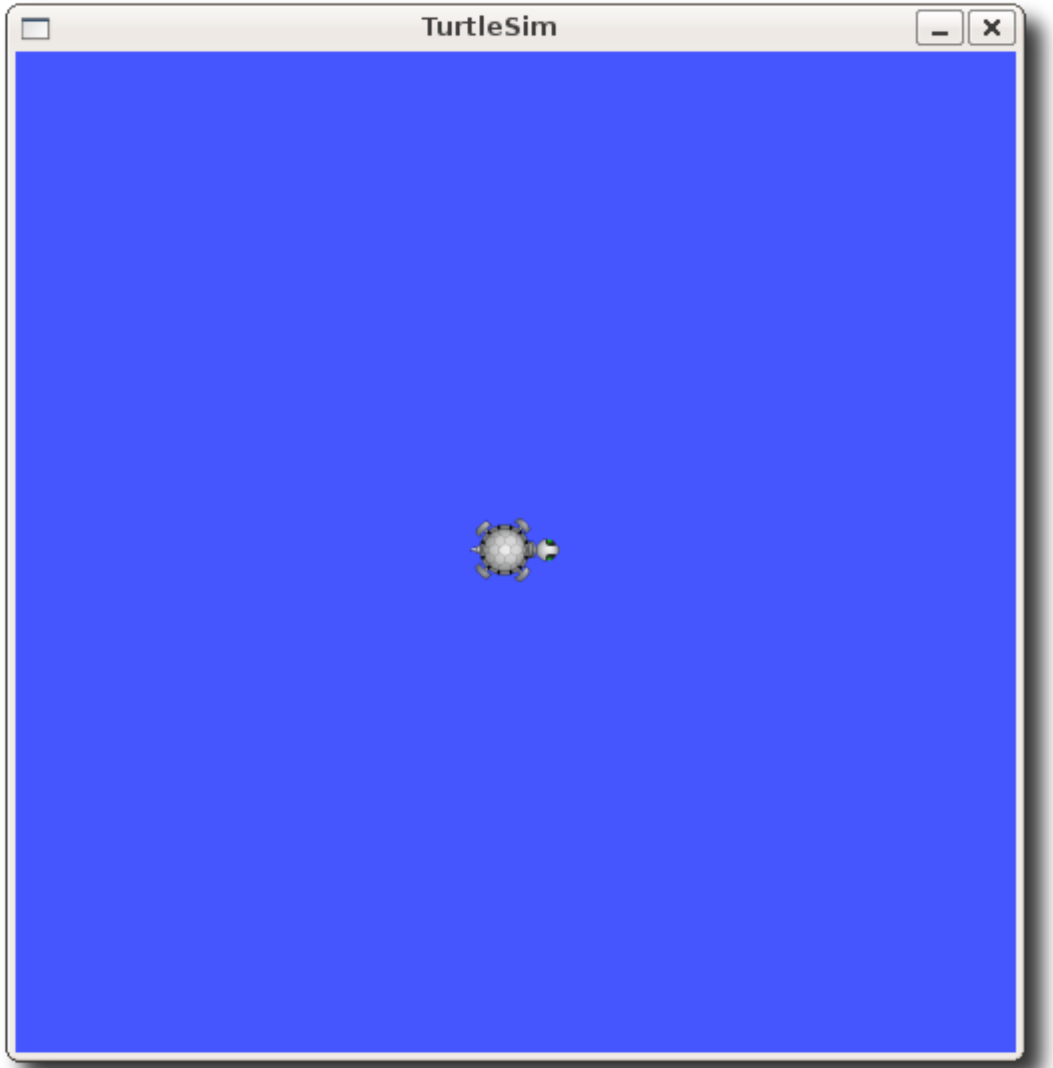
1. install ROS
2. install turtlesim

```
$ sudo apt-get install ros-$(rosversion -d)-turtlesim
```

3. launch roscore on a terminal
4. run turtlesim node
   on another terminal

```
$ rosrun turtlesim turtlesim_node
```

This will open the simulator.



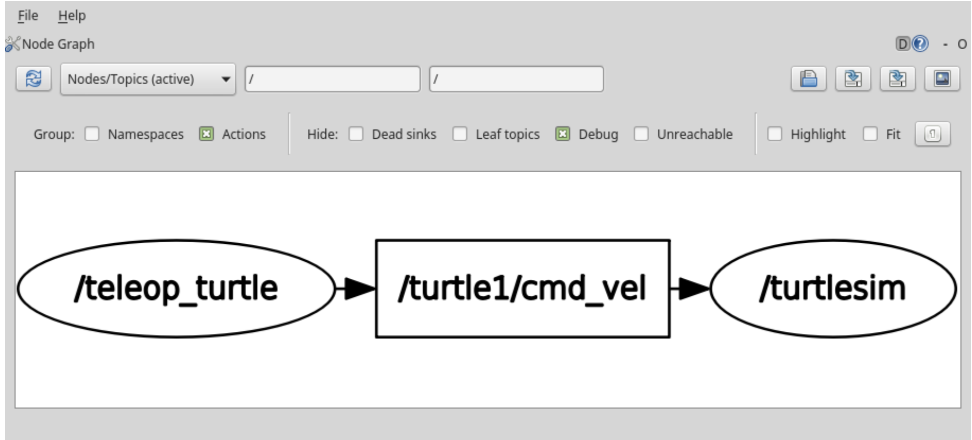
Turtlesim

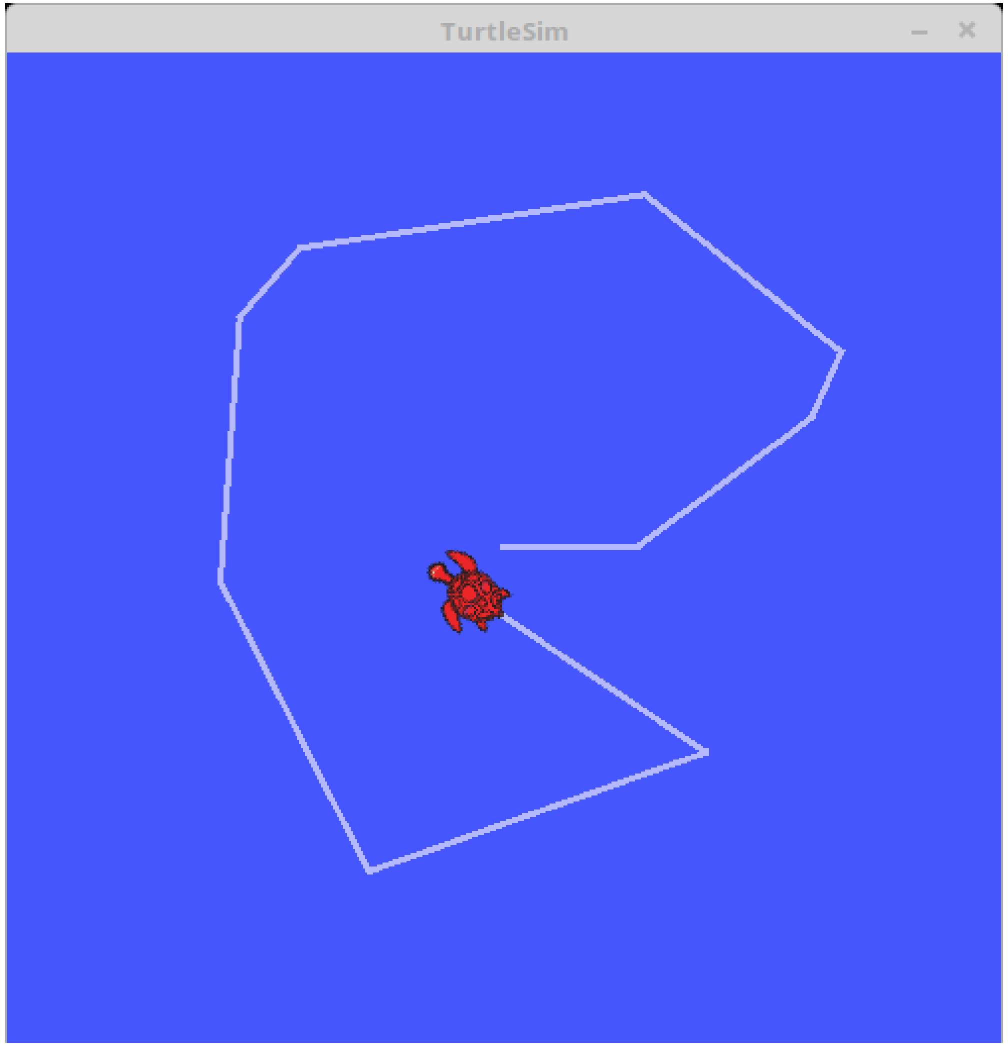
4. launch teleoperation node
   on another terminal

```
$ rosrun turtlesim turtle_teleop_key
```

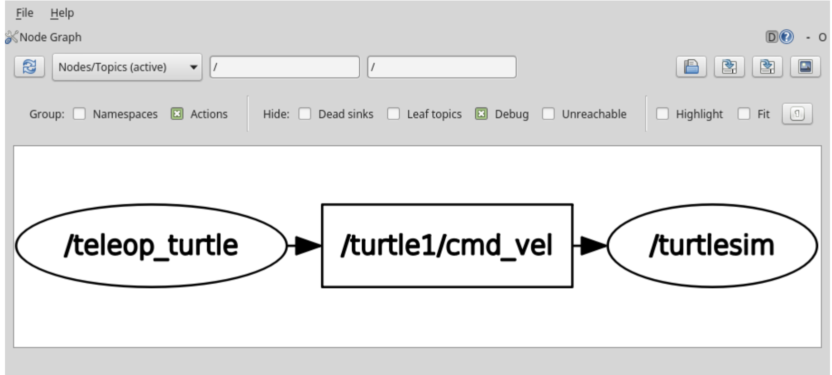
5. move around the turtle with keys

6. open rqt_graph
   on another terminal

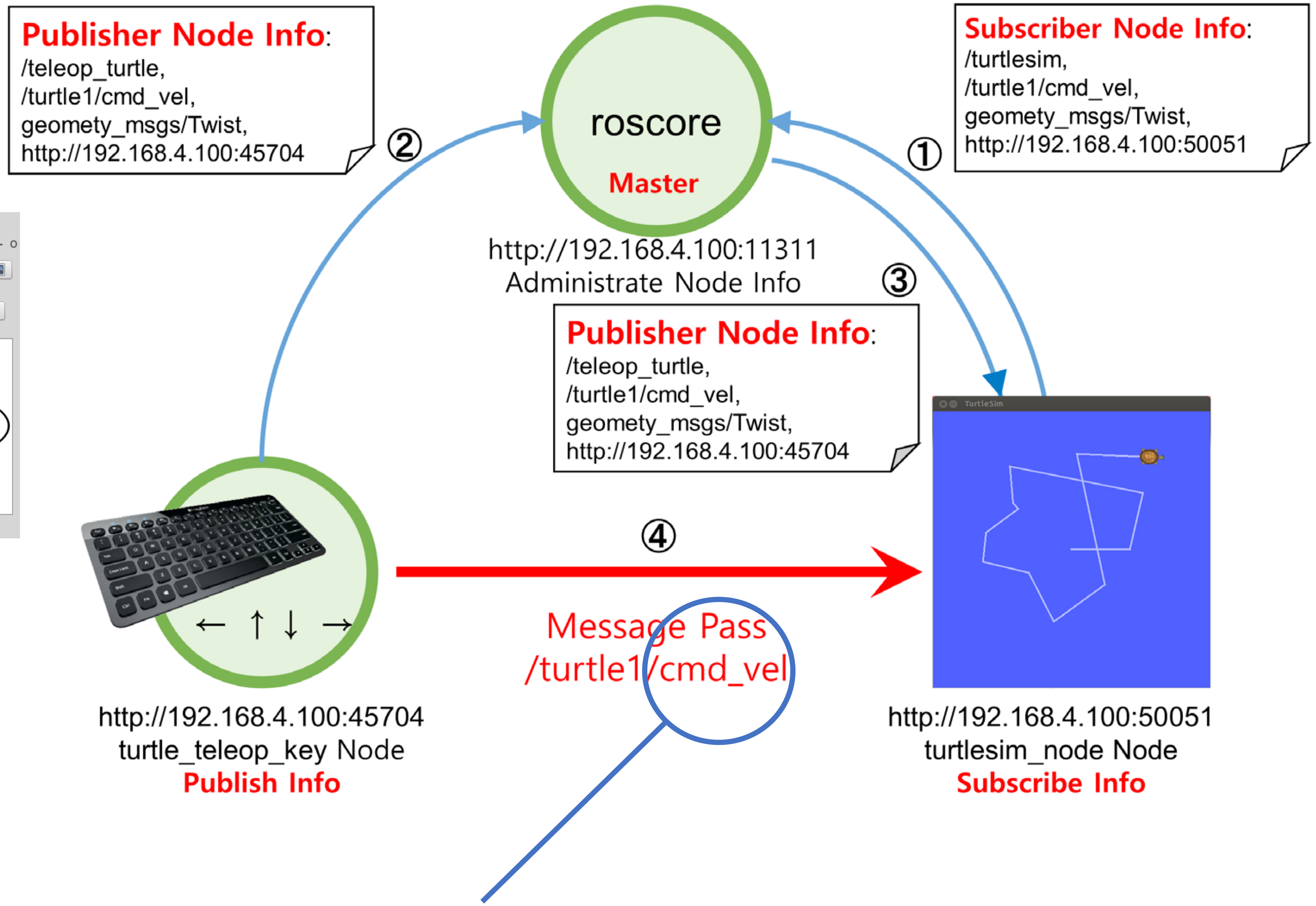




Turtlesim



RQT_graph shows the computational graph, Nodes and topics



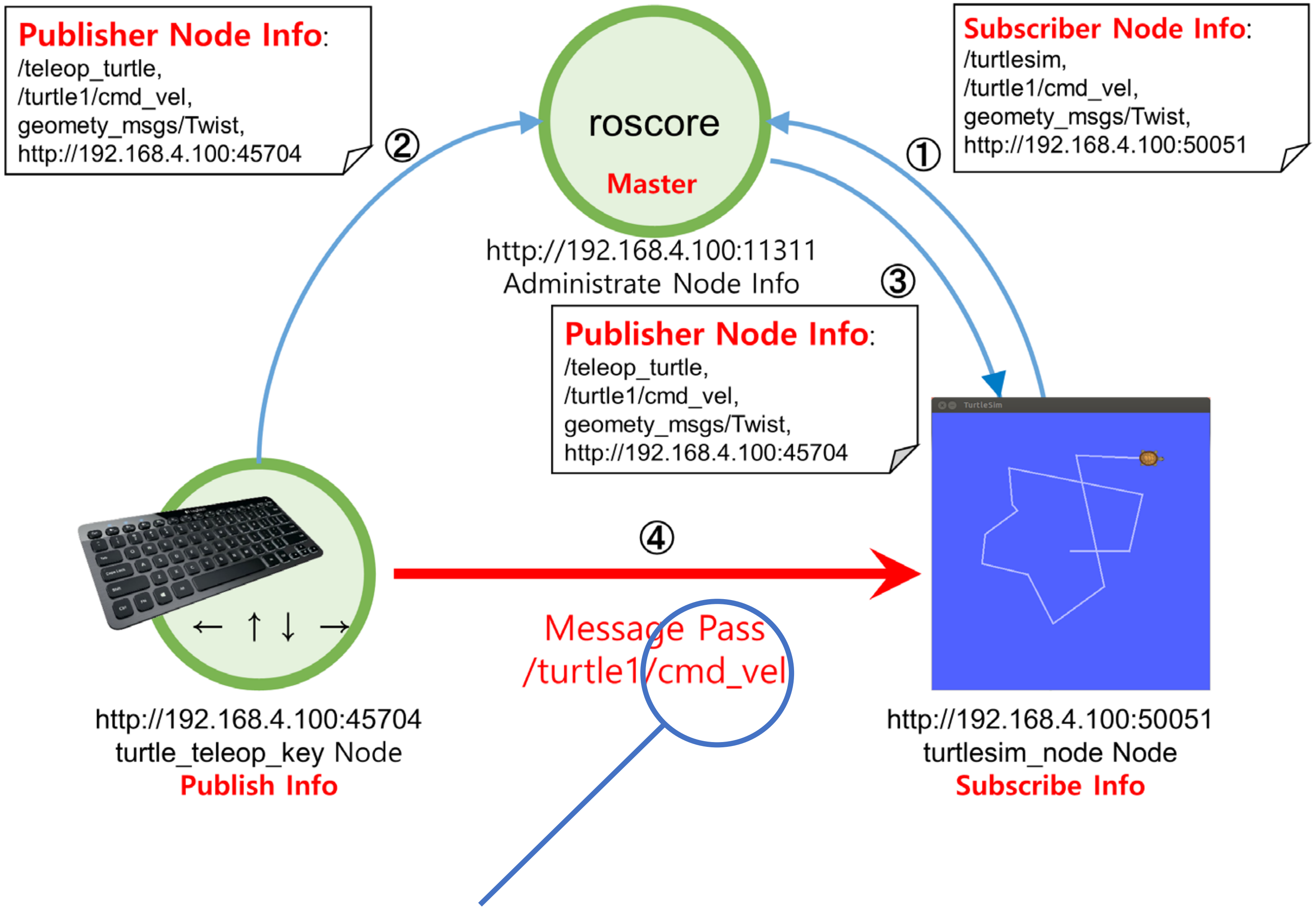
cmd_vel is the standard topic for sending velocity commands

Command Line Tools

```
$ rostopic list
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

Rostopic list shows all active topics.

- cmd_vel is the velocity
- pose is the (x,y) position



cmd_vel is the standard topic for sending velocity commands

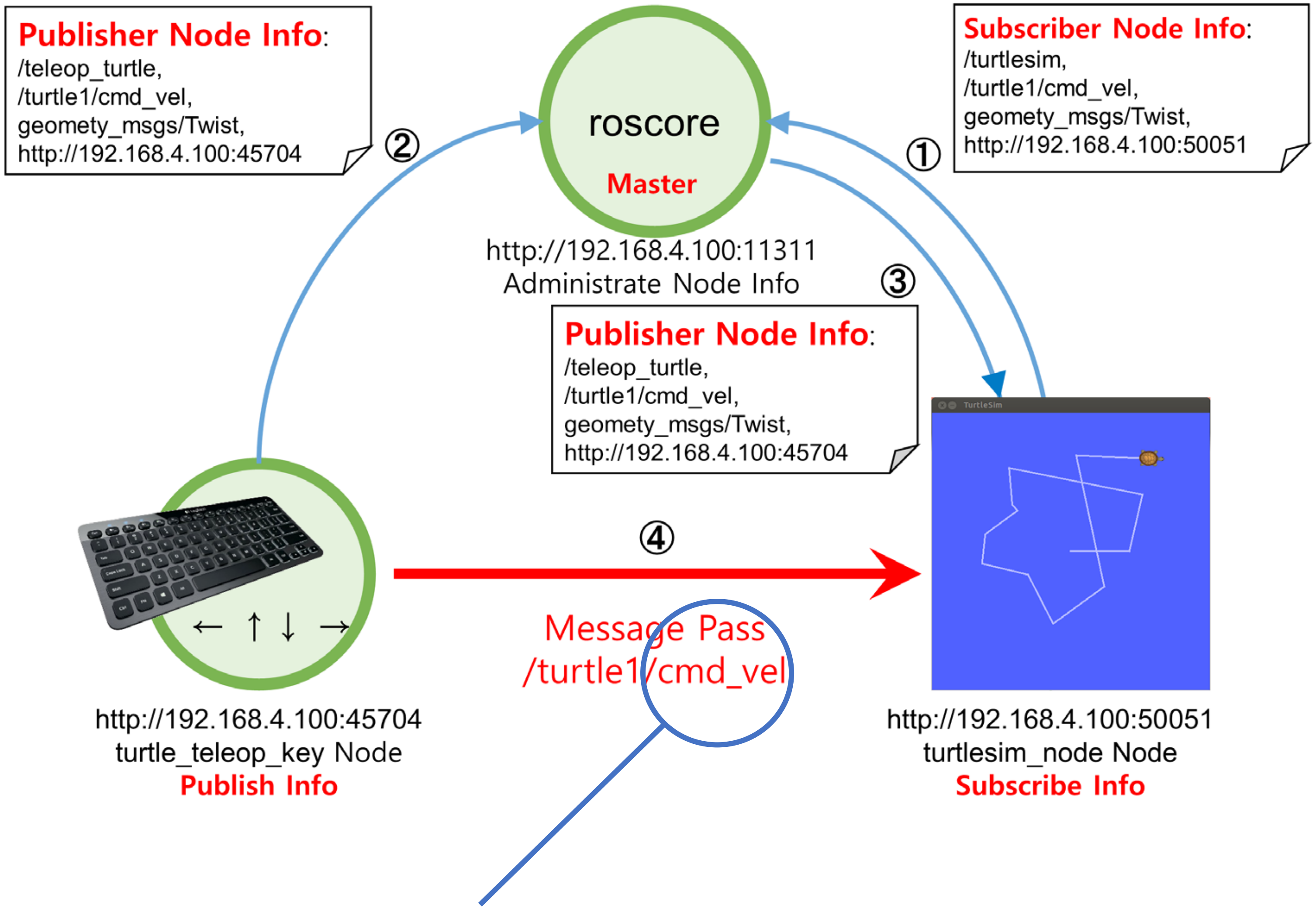
Command Line Tools

```
$ rostopic echo
/turtle1/pose
x: 5.35244464874
y: 5.544444561
theta: 0.0
linear_velocity: 0.0
angular_velocity: 0.0
~ omitted ~
```

Rostopic echo listens and streams all messages of a given topic:

```
$ rostopic echo
/turtle1/cmd_vel
```

publishes the result of teleop

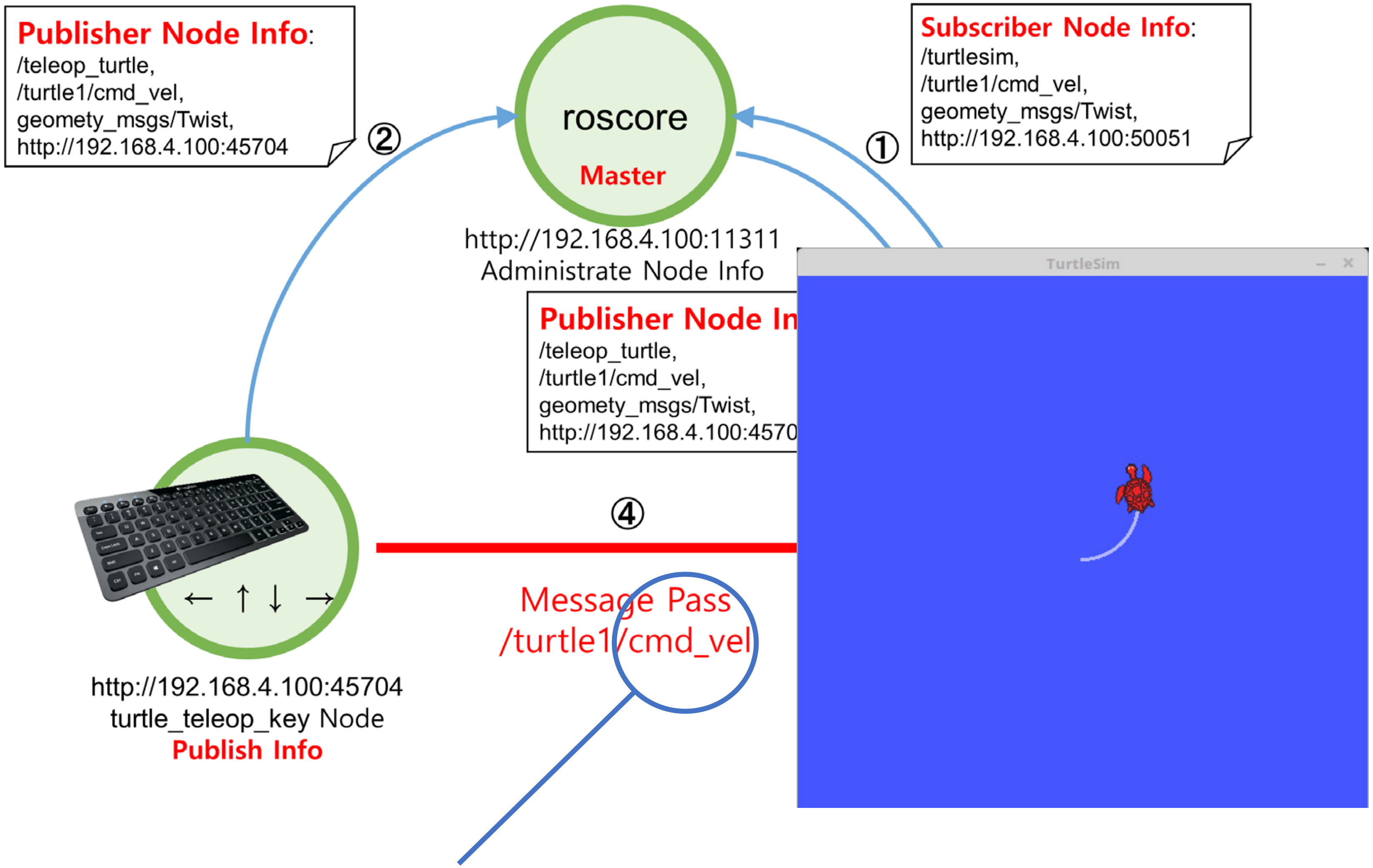


cmd_vel is the standard topic for sending velocity commands

Command Line Tools

```
$ rostopic pub -1
/turtle1/cmd_vel
geometry_msgs/Twist --
'[2.0, 0.0, 0.0]' '[0.0,
0.0, 1.8]'
publishing and latching
message for 3.0 seconds
```

rostopic pub allows to publish a message on a topic manually



cmd_vel is the standard topic for sending velocity commands

```
$ rostopic list
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

```
$ rostopic pub -1
/turtle1/cmd_vel
geometry_msgs/Twist --
'[2.0, 0.0, 0.0]' '[0.0,
0.0, 1.8]'
```

```
$ rostopic echo
/turtle1/pose
x: 5.35244464874
y: 5.544444561
theta: 0.0
linear_velocity: 0.0
angular_velocity: 0.0
~ omitted ~
```

One of the (most) useful feature of ROS are command line tools.

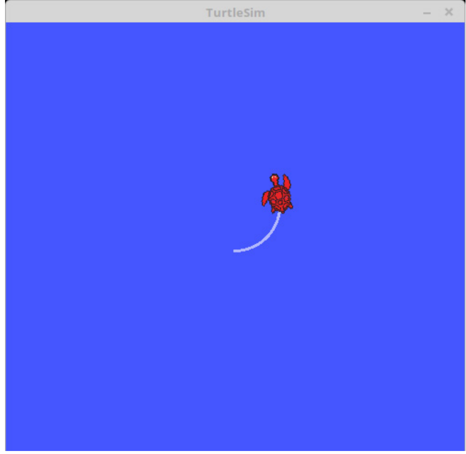
Command line tools allows the inspection of communication between nodes, to see if naming of topics and nodes is properly set, and to manually trigger messages.

ROS command line tools are designed not to write all commands directly but to let ROS autocomplete them with the Tab button.

Structure of messages, packages and topic names are automatically autocompleted by ROS – facilitating development and preventing errors.

Moreover, if ROS cannot autocomplete a message it usually indicates that something is going wrong with such topic/node/message

Turtlesim and services




```
$ rosservice call /turtle1/set_pen 255 0 0 5 0
```

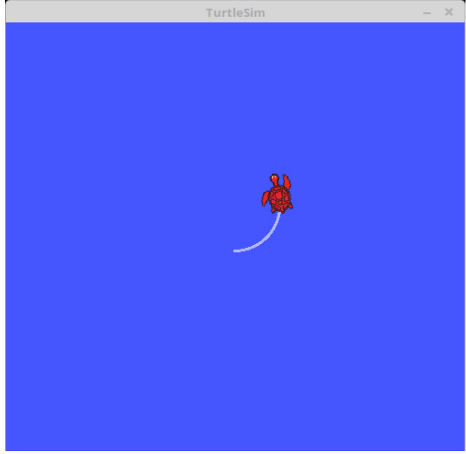
Turtlesim offers also some services – e.g. in this case changing the color of the robot trajectory

Rosservice call is the command used for triggering manually a ROS service
(also here, let ROS to autocomplete the call to service for you)

```
$ rosservice list
```

```
$ rosservice args /turtle1/set_pen
r g b width off
```

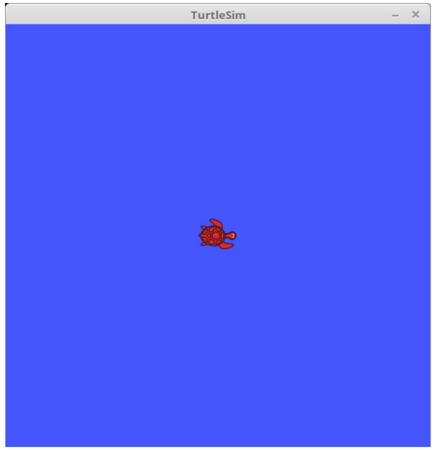
Turtlesim and params




```
$ rosparam list
/background_b
/background_g
/background_r
/rosdistro
/roslaunch/uris/host_192_16
8_1_100__39536
/rosversion
/run_id
```

Rosparam is the command line tool for accessing all param of the parameter service.
Turtlesim has background colors as params

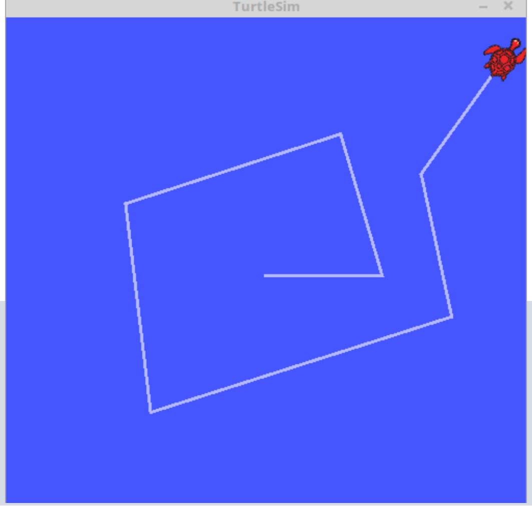
Try to run “rosparam get” and “rosparam set” and let ROS to autocomplete (tab)




```
$ rosparam set background_b 0
$ rosservice call clear
```

Turtlesim and rosbags

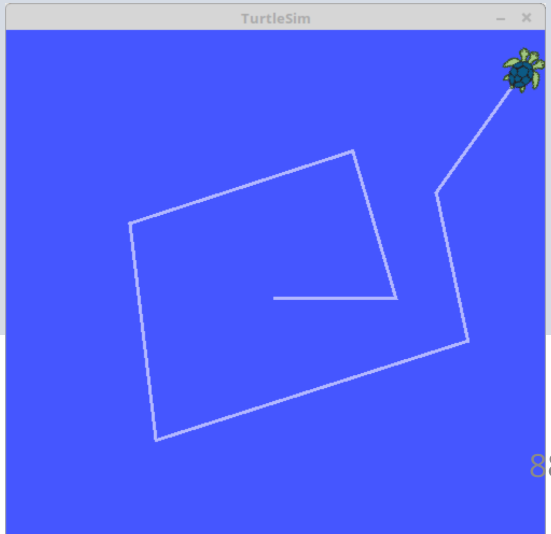
```
$ rosbag record /turtle1/cmd_vel
[INFO] [1499663788.499650818]: Subscribing to /turtle1/cmd_vel
[INFO] [1499663788.502937962]: Recording to 2017-07-10-14-16-28.bag.
```



saves messages of /turtle1/cmd_vel into a bag

Replays the same bag and publish on /turtle1/cmd_vel

```
$ rosbag play 2017-07-10-14-16-28.bag
[INFO] [1499664453.406867251]: Opening 2017-07-10-14-16-28.bag
Waiting 0.2 seconds after advertising topics... done.
Hit space to toggle paused, or 's' to step.
[RUNNING] Bag Time: 1499663790.357031 Duration: 0.000000 / 17.419737
~ omitted ~.
```



ROS

Autonomous navigation with ROS

Robot Navigation

Navigation is the ability to determine the position of the robot and to plan a path towards a goal location.

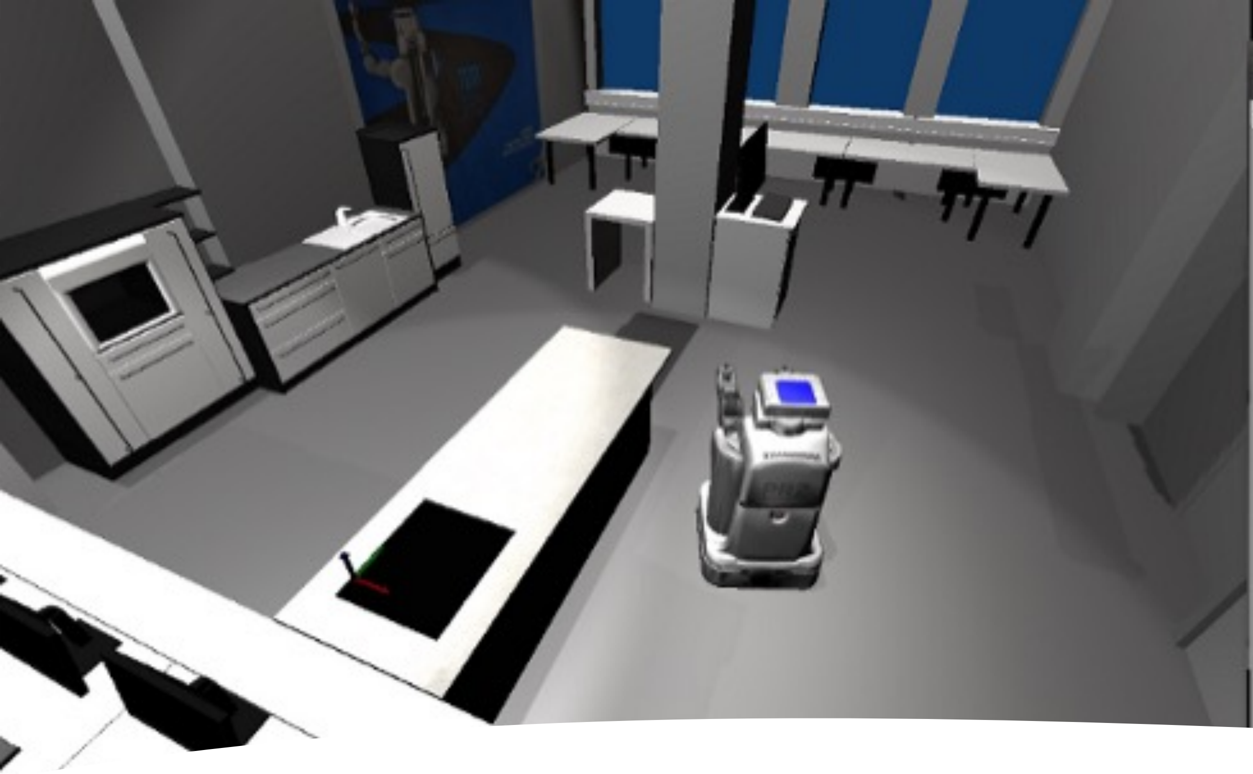
Navigation is both a core capability of autonomous mobile robots and a complex one.

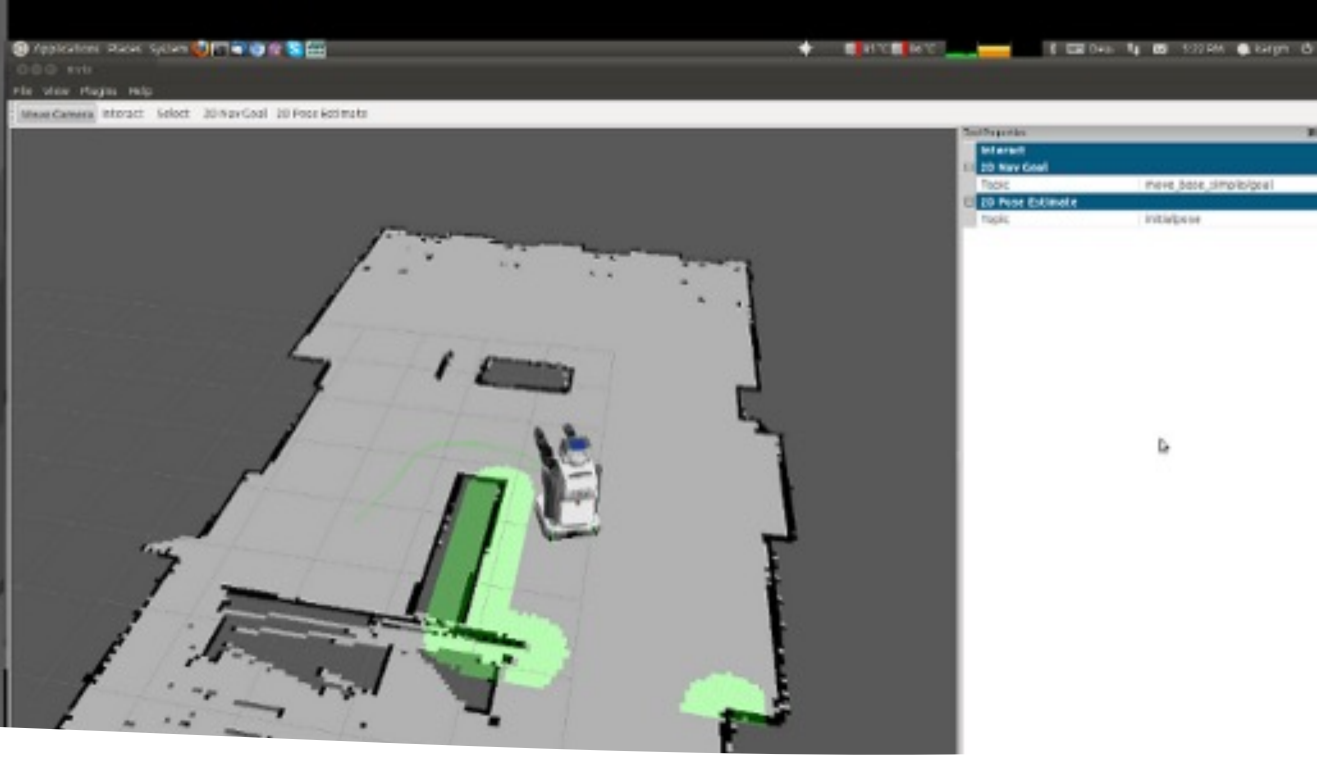
ROS and navigation

- how navigation is done is based on sensors mounted on the robot
- navigation sub-tasks depend on which is the goal of the robot (USAR, service robot, teaching, …)
  (e.g. a robot working in close contact with humans, should take that into account)

ROS provides support in both directions:

- integration and drivers for sensors
- ready to use modules





Outline

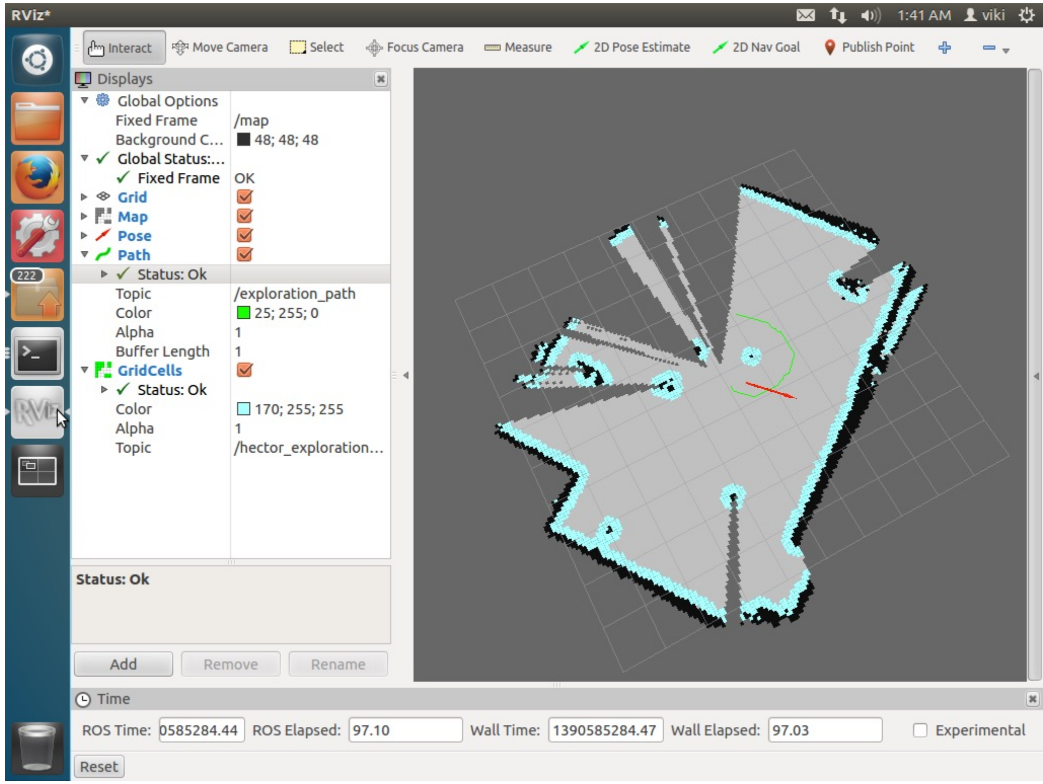
- Overview of robot sensors and how to integrate them
- Overview of what are the sub-tasks that the robot needed for navigation, and how ROS implement them
- How to use all of this in ROS with the ROS Navigation Stack – wheeled ground robots

ROS

Robot and sensors

Robot Sensors

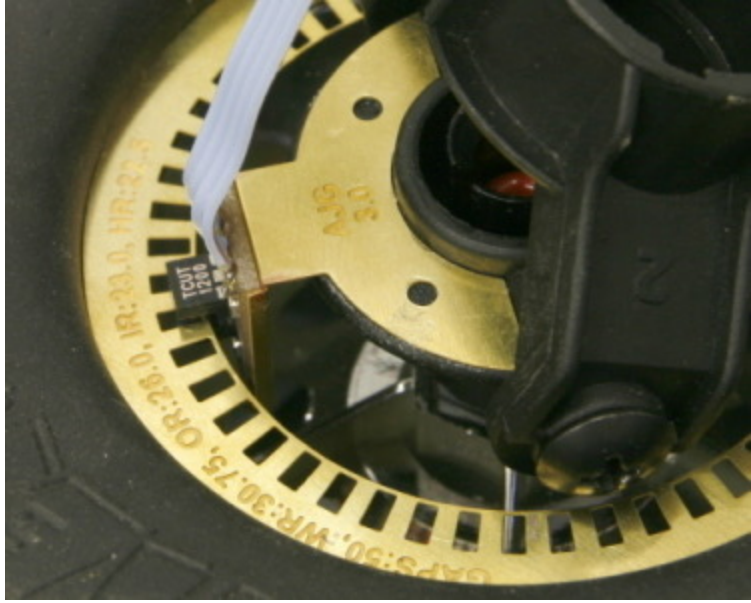
- sensors are the main and primarily source of knowledge of a mobile robot
- sensors are noisy, inaccurate
- some of these inaccuracies can be modeled, some not
- sensors are *raw data* and do not have any semantic
  - humans can interpret the data (we have semantic knowledge)
  - robot can't (unless you implement it)



Sensors

- proprioceptive – info about the robot internal state
  - speed
  - Encoders
  - battery
- exteroceptive – info about the world
  - vision
  - 2D/3D lidars
  - olfaction
  - chemicals
- active and passive (e.g. sonars and microphones)

Proprioceptive sensors



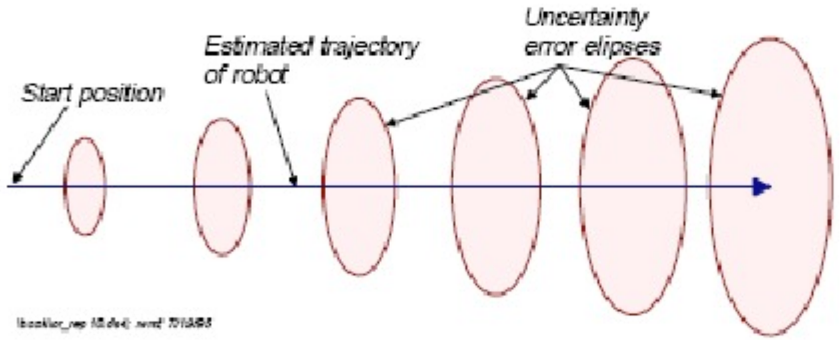
- wheel encoders
- IMU
    - accellerometers
    - gyros

Open-loop estimation of the robot state.

Errors in sensors are integrated too so the estimate state gets less and less accurate with operational time

Odometry

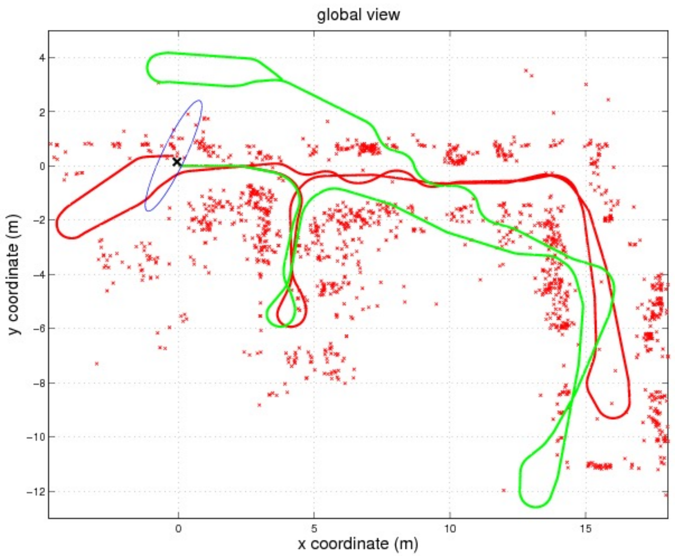
Odometry is the use of data from motion sensors to estimate change in position over time, to estimate the robot position relative to a starting location.



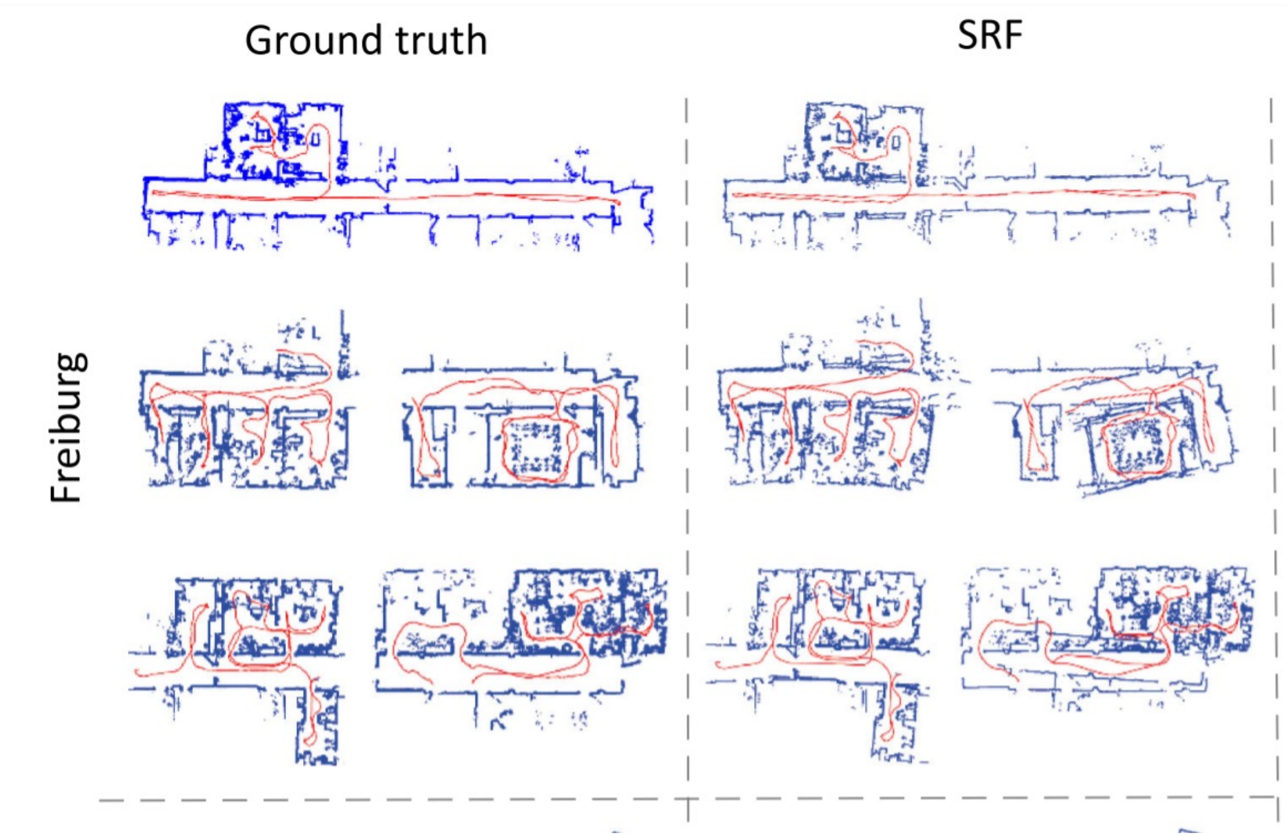
It could be used open-loop to estimate the robot position, but needs integration.

Rotational movements are fare more difficult to measure than translational ones.

Odometry



Sometimes odometry is so noisy that is reconstructed from exteroceptive sensors



From https://doi.org/10.1109/TRO.2018.2861911

Exteroceptive sensors (for navigation)

- laser rangescanner (LIDAR)
  2D and 3D velodyne
- camera (stereo – mono)
- RGBD camera
- sonars
- …

2D lidar

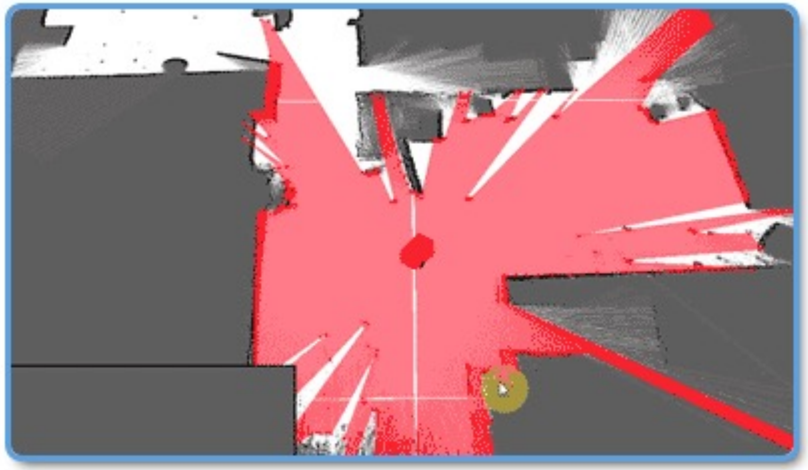
- (probably the most used sensors in robotics)
- 2D representation of the environment
- highly reliable
- widely adopted and used also on commercial platforms
- measure time of flight of laser beam

ROS

2D lidar specs

- usable range 3-5m (indoor) 30-50m (outdoor)
- 1hz – 50hz
- relatively cheap (0.5k – 15k)
  (1k for a good indoor one,10k for a «good one»)
- Wide FOV (180° → 360°)
- Security for collision detection
- not subject to environmental changes (e.g. day/light)

2D LIDAR PROs

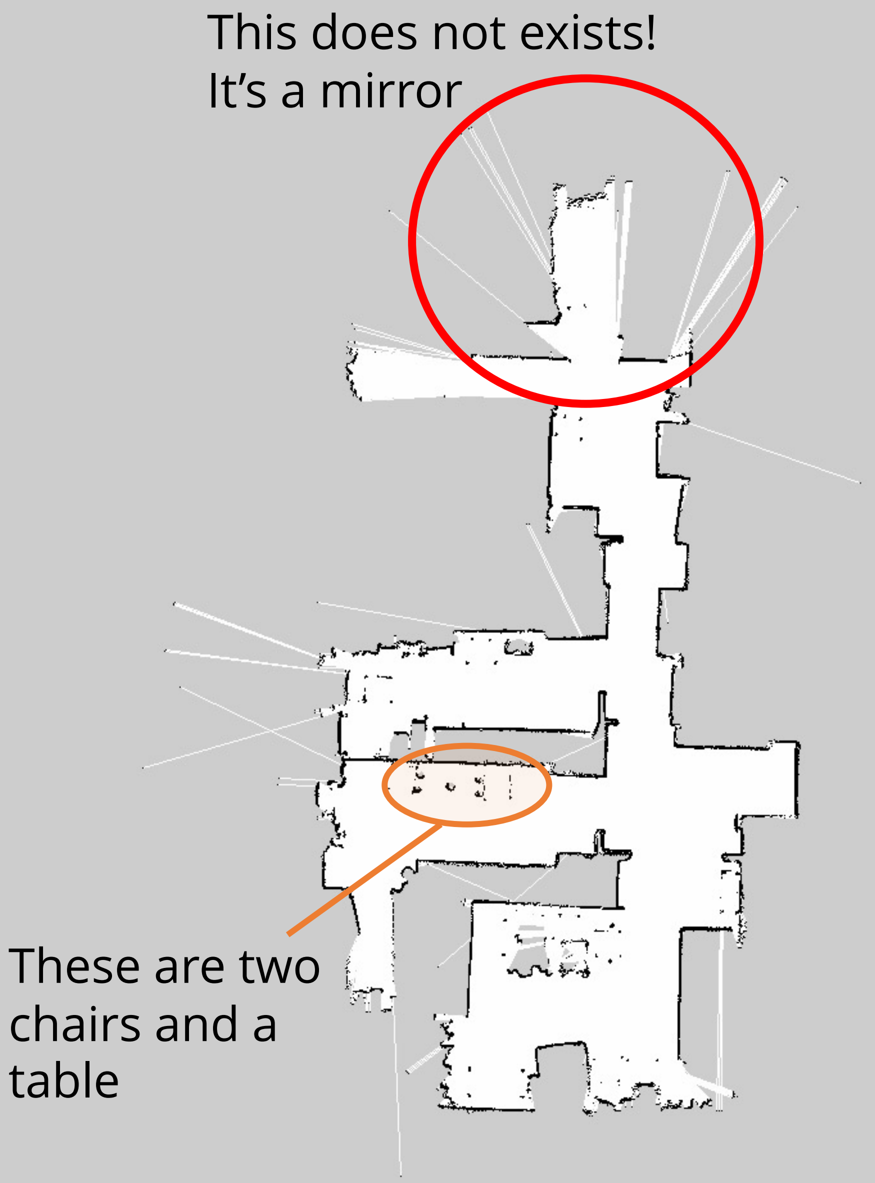


Realtime ICP-SLAM based on RPLIDAR

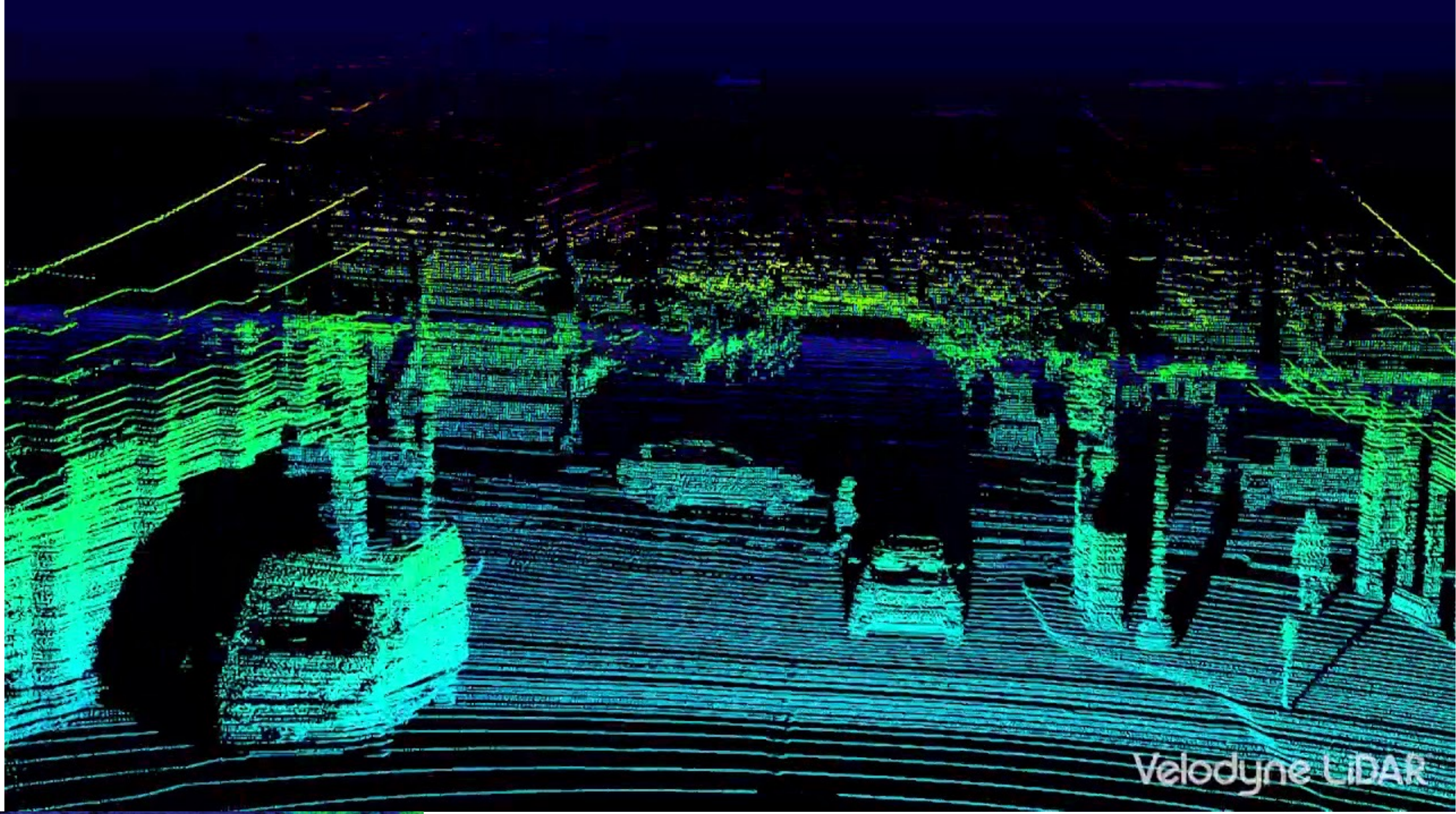
- reliable
- cheap
- easy to use and to process
- robust to changes in the environment and to light changes
- large FOV
- long range
- outdoor and indoor
- useful for mapping, localization and path planning

2D LIDAR CONs

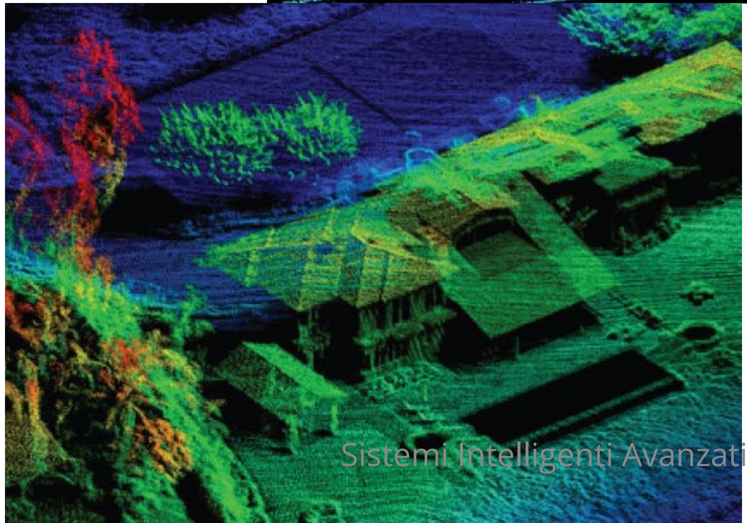
- not that much info (2D)
- difficult to add semantic knowledge (only occupancy)
- reflections / mirrors are a problem
- planar – only object at a given height can be perceived (e.g. chairs, tables are not visible)
- not suited for UAV (drones) or in general to non-wheeled mobile robots



- multi-layer lidar
- highly reliable
- 3D representation of the environment
- wide range of application (autonomous cars)
- semantic knowledge can be added
- used mostly outdoor



3D LIDAR - velodyne



ROS

3D LIDARs

- Usually 360° and higher range than 2D LIDAR
- thicker representation close to the source and coarser w.r.t the distance
  (all planar scans start from the same point)
- not subject to lighting condition changes (e.g. night)
- very expensive (10k-100k)
- most indoor application are still based on a 2D map

Currently, 2D lidars are preferred to 3D ones only for their price

Cameras (monocular)

- perceive lots of data
- “easy” to add semantic knowledge
  (e.g. object recognition, people detection, classification…)
- cheap (10$ → 10k)
- no depth info
- limited range + distortion
- difficult to be used to build a map of the environment
- subject to light changes / day-night changes, …

Camera (stereo)

- 2x monocular camera
- allow triangulation –
  can be used to do 3d reconstruction
- estimation error grows with distance
- sparse 3D representation
- same PROs and CONs of monocular camera
  (more PROs, but also you need calibration)

RGBD Camera

- camera + depth information using an active sensor
- easy to reconstruct 3D image of the environment
- good for a lot of sensing tasks (e.g. human detection)
- widely used and useful, especially indoor
- limited range (useless at 3/5m, some even before)
- distortion
- very cheap (100$→1000$)

(b) Asus Xtion Sensor

What sensors for navigation?

- 2D lidars provide cheap, reliable, long-range knowledge of the environment but…
- …are planar and little to none semantic knowledge
- cameras (RBGD) have limited range and are noisy and subject to light changes, but…
- … provide a lot of data

Why not use both together?

Strands Scitos G5

2x RBGD CAMERA

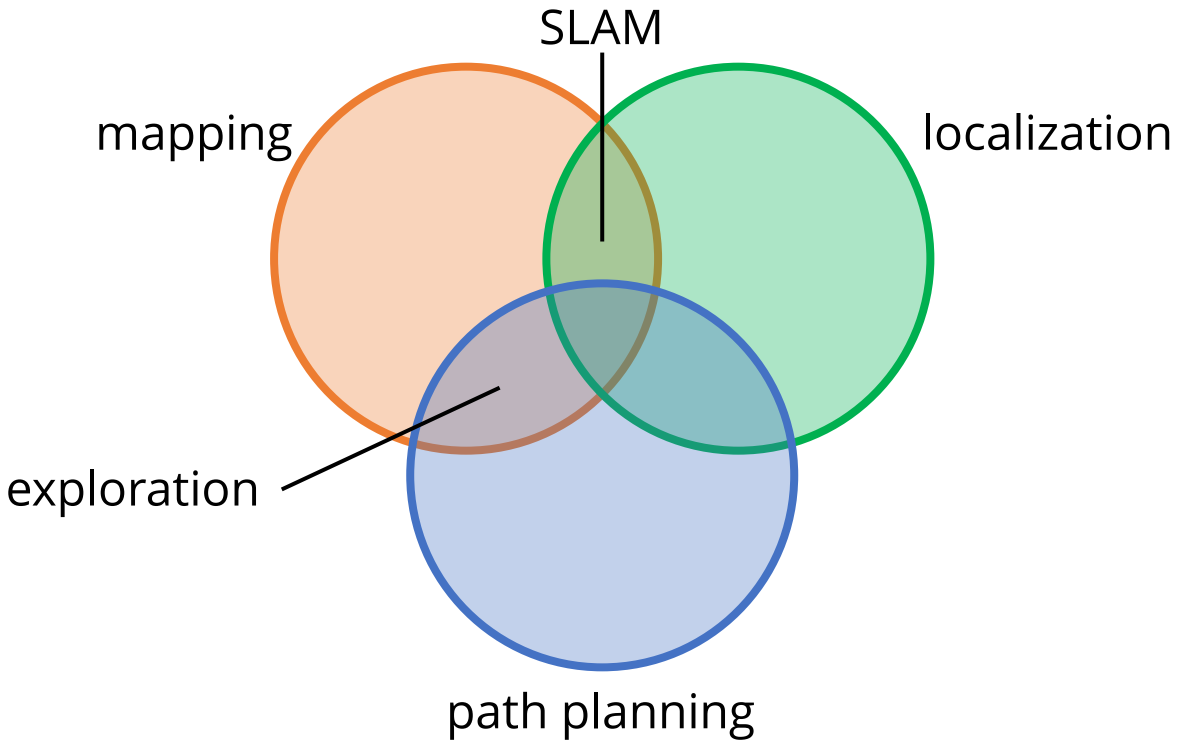
2D LIDAR (SICK LMS)

Our Giraff robot

- 2D lidar at the bottom for mapping and navigation
- an RBGD camera (Orbbec ASTRA) at the top, pointing downwards for detecting obstacles (tables) and help navigation
- another RGBD camera on the neck pointing upwards for people/object detection

ROS

From sensors to navigation



From [Grisetti, Burgard, Stachniss]

for moving autonomously the robot should be able to understand the environment from its sensor measurement

What is needed for navigation?

- sensors measurement → what I can see?
- map → what is the environment?
- localization → where am I?
- path planning → how I go there?

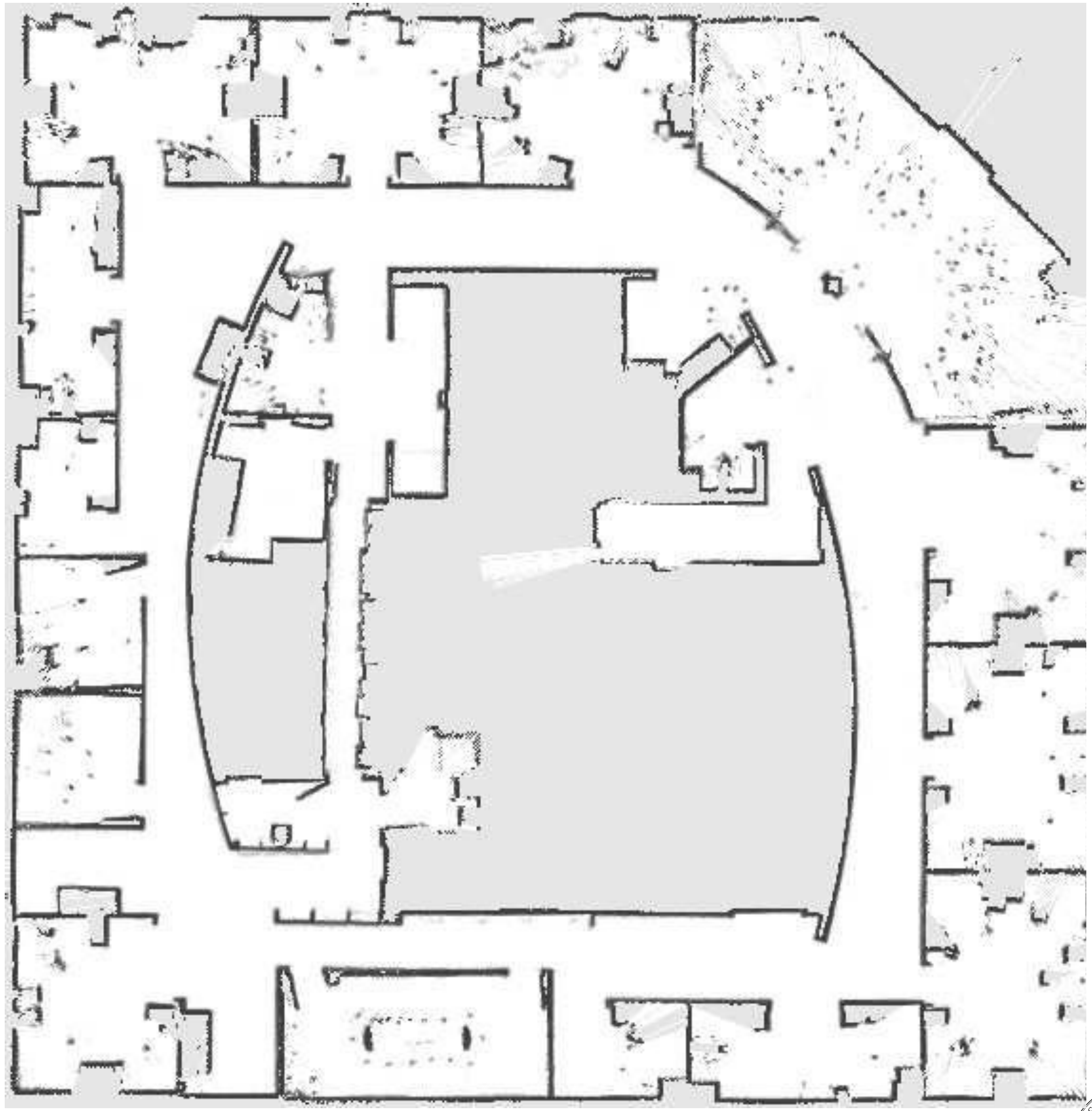
Besides this, there are many other subtasks:
mapping (creation of the map) is the most important one.

Note that we are not considering who is deciding where and how the robot should go (reasoning)

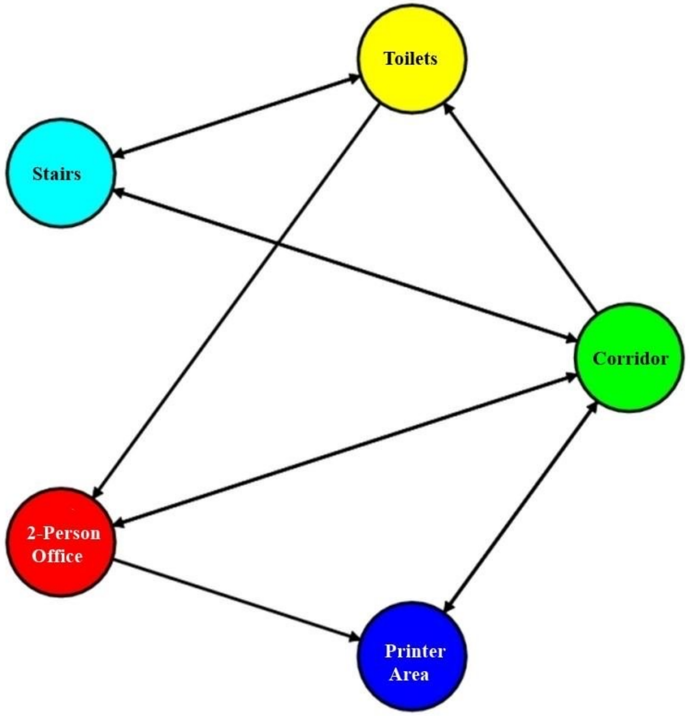
Map

- a map is a representation of the environment where the robot is operating
- Metric map
  - 2D or 3D
  - grid map
  - feature based
  - landmark-base
- Topological map
- hybrid maps

More maps could be used at the same time by a robot

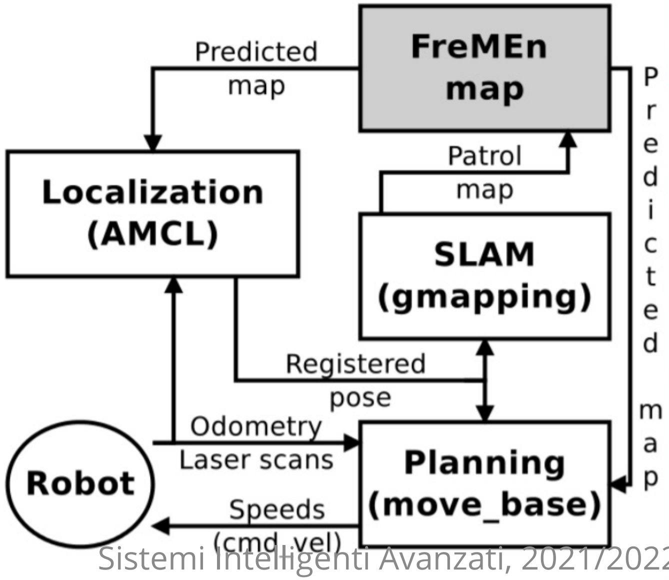


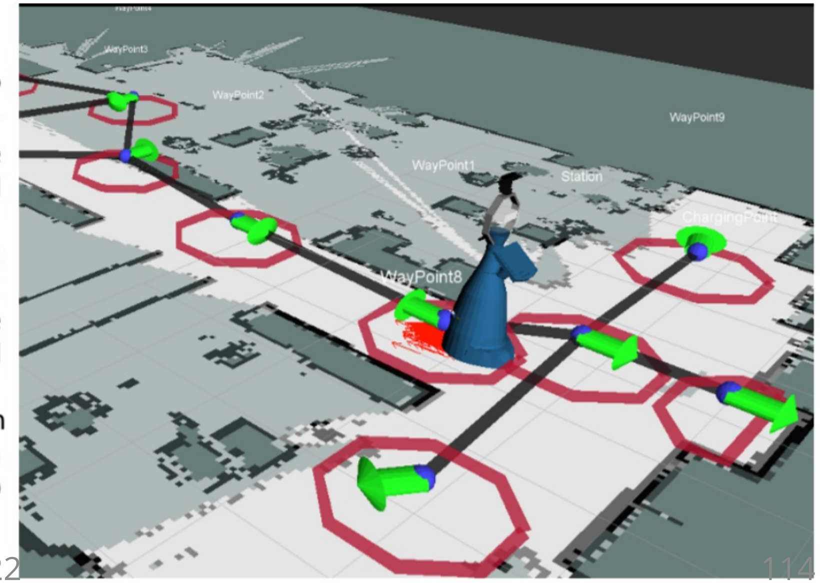
Topological and hybrid maps



Topological maps are an abstract graph representation of the environment, which could be used jointly with the metric map.

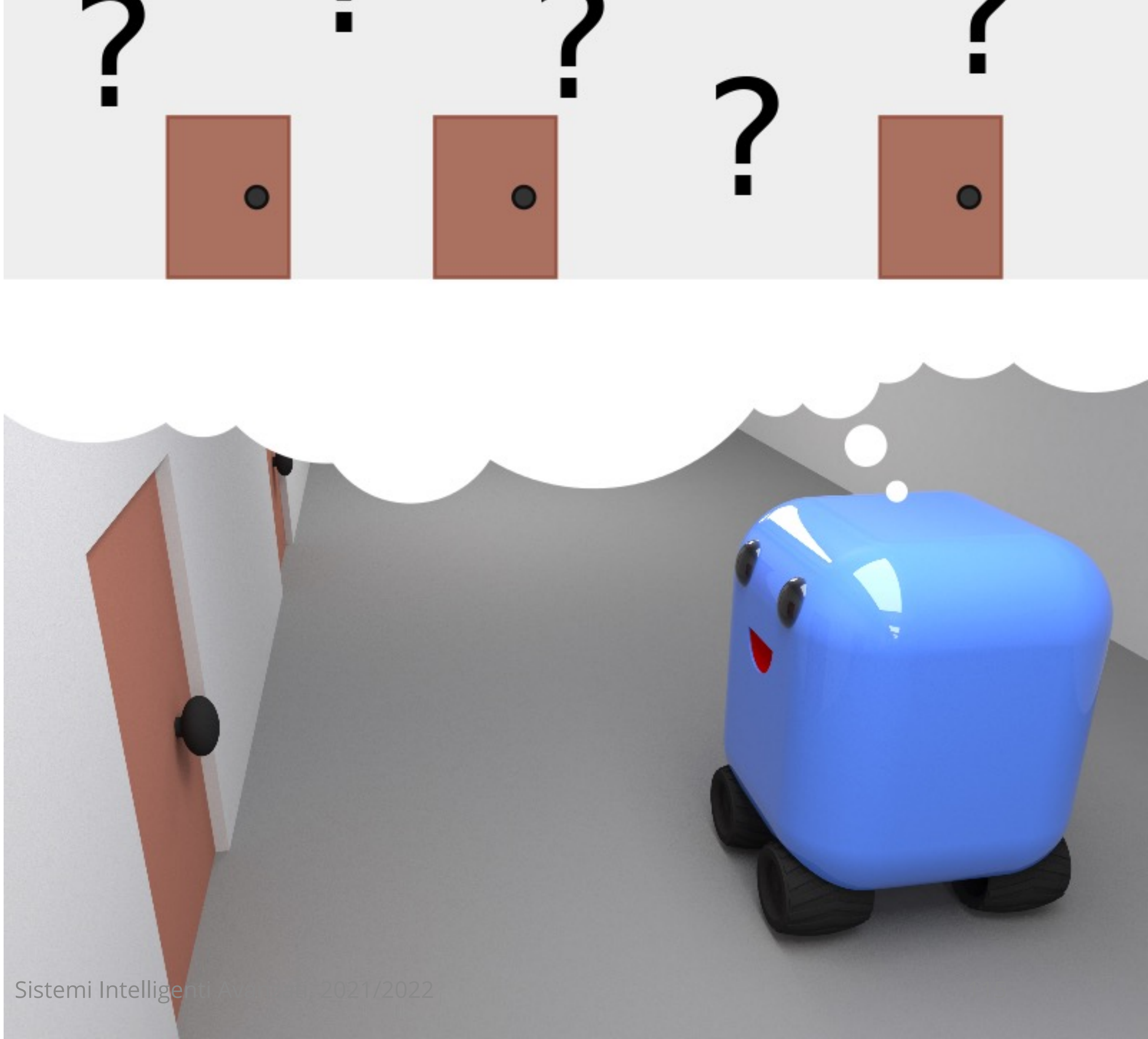
From Krajnik et al, T-RO, 2017





Localization

- the robot should know its position
(a *pose*, in 2D is a <x,y,theta> vector)
in the map (reference frame)
- when the robot move the position is updated according to the measurements performed from start till the current (latest) sensor measurement



AMCL

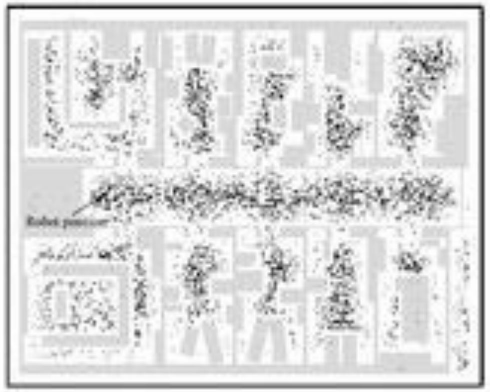


Fig. 2: Global localization: Initialization.

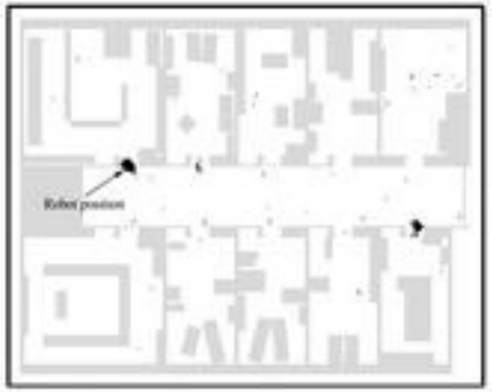


Fig. 3: Ambiguity due to symmetry.

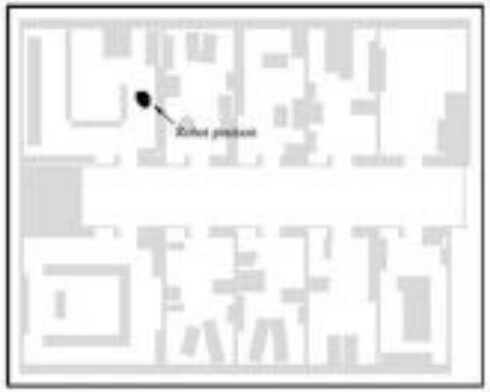


Fig. 4: Successful localization.

- a method widely used is Adaptive Monte Carlo Localization (AMCL)
- available in ROS and default localization method in the navigation stack
(more later)
- particle-based (several estimated location are maintained and updated together, the more the “cloud” of particles is thick, the more precise is the localization

Monte Carlo Localization (MCL)

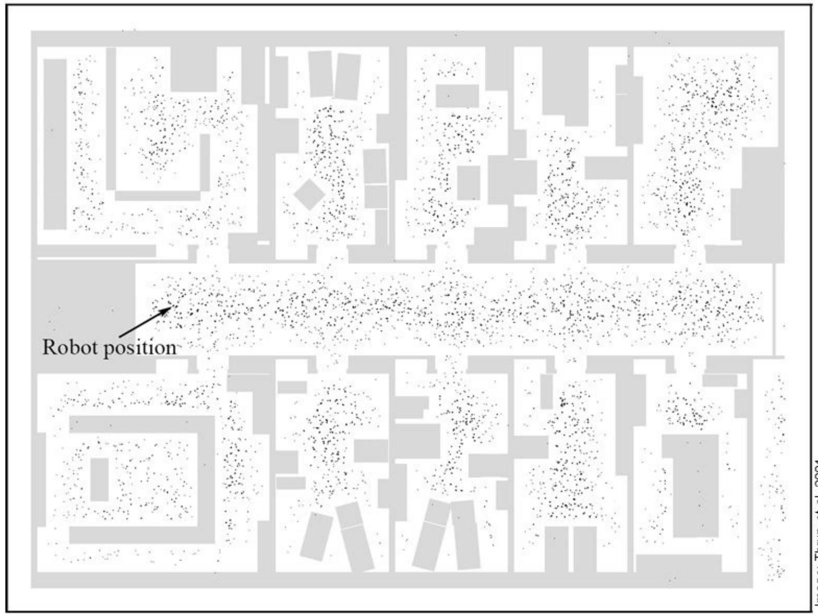


Image: Thrun et al. 2001

10

Mapping

Given the robot position, a sequence of measurements (and the position from which those measurements have been performed), build the map of the environment

How to know the robot position in the map, if I have no map?

We need to solve a bigger problem

SLAM: Simoultaneous Localization and Mapping

Estimate:

- the map of the environment
- the trajectory of a moving device

using a sequence of sensor measurements

SLAM is one of the core problems in robotics, widely studied and hundreds of solutions proposed during 20+ years

SLAM: Simoultaneous Localization and Mapping

SLAM approaches can be different w.r.t the type of the robot:

- indoor
- outdoor
- marine (water-surface or submarine)
- underground
- …

The type of map built (2D/3D) and the type of sensors used for mapping (2D/3D lidars, vision, sonars, …)

2D SLAM

- used for indoor environments
- 2D grid maps
- robust
- available and ready-to-use solutions
- 2D lidar as source (cheap and reliable)
- most algorithms (e.g. planning) assume are designed for working with such representation
- most methods are based on Filters
  Kalman filter, EKF, particle filter, …

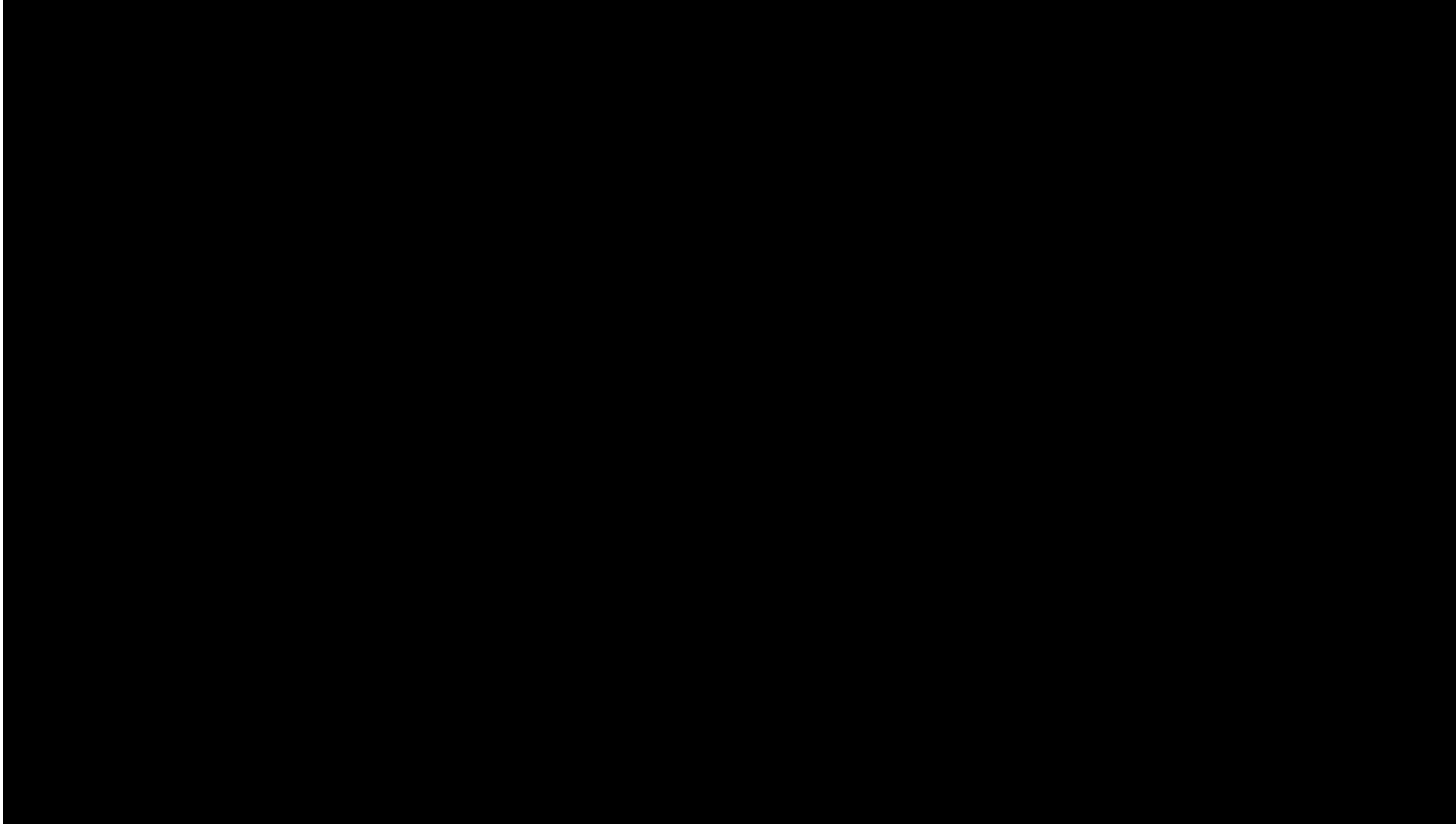
ROS

2D SLAM in ROS

- several available methods – widely used, tested and robust
- need parameter configuration, but it is not that hard + docs
- Gmapping [Grisetti et al, T-RO, 2009], Hector SLAM, and Cartographer are the most popular ones
- work reasonably well with a lot of different robot platform/settings, are robust to changes and clutter (noise in sensors and furniture, …), complex and large-scale environments, …

http://wiki.ros.org/gmapping
http://wiki.ros.org/hector_slam
http://wiki.ros.org/cartographer

Example: Cartographer SLAM



2D SLAM

- SLAM is the process used for building the map
- no-knowledge of the environment
- the map is built incrementally
- save and use the map later for future uses
- assumption: the environment is static (open/closed doors)
- dynamic changes (e.g. people) can be filtered out (while mapping) and are not present when a static map is used…
- …low-freq dynamic changing (e.g. doors) and static changing could jeopardize robot localization and navigation → redo mapping or use dynamic mapping mechanisms

Solution for mapping and navigation

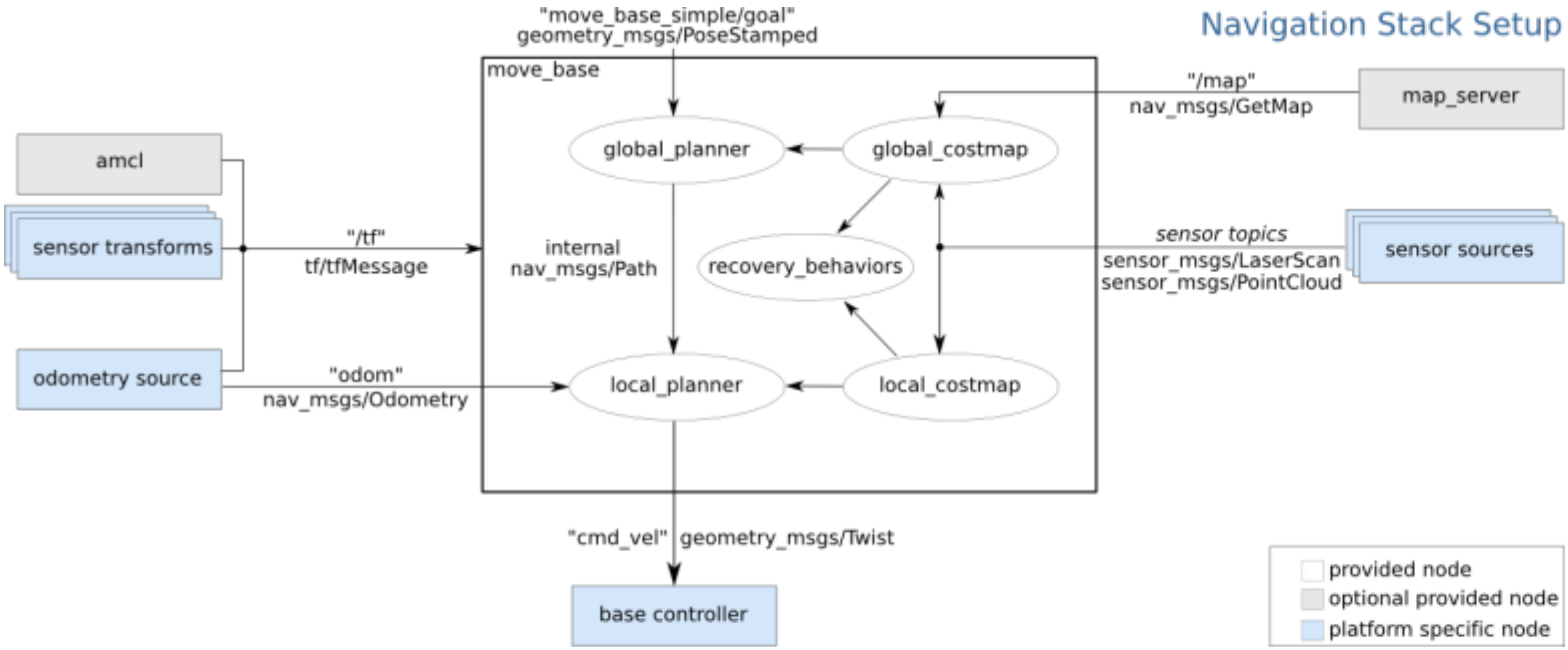
- New environment:
  - do SLAM and create a map while the robot moving
- Common setting:
  - Start a SLAM method
  - Move the robot around and create a map (manually or automatically)
  - Save the map
  - Use the map for future robot runs

ROS Navigation stack

- assumptions: you have selected a set of sensors, the robot architecture, and you have chosen your favorite localization algorithm (AMCL default) and have a map
- map can be given (so you use a previously acquired map) or being built incrementally (SLAM)
- ROS navigation stack handles this setting and allows path planning: finding if exists a list of robot positions that, if followed, allows the robot to reach its goal
- also, it allows the execution of this path
- the core of the method is called move_base

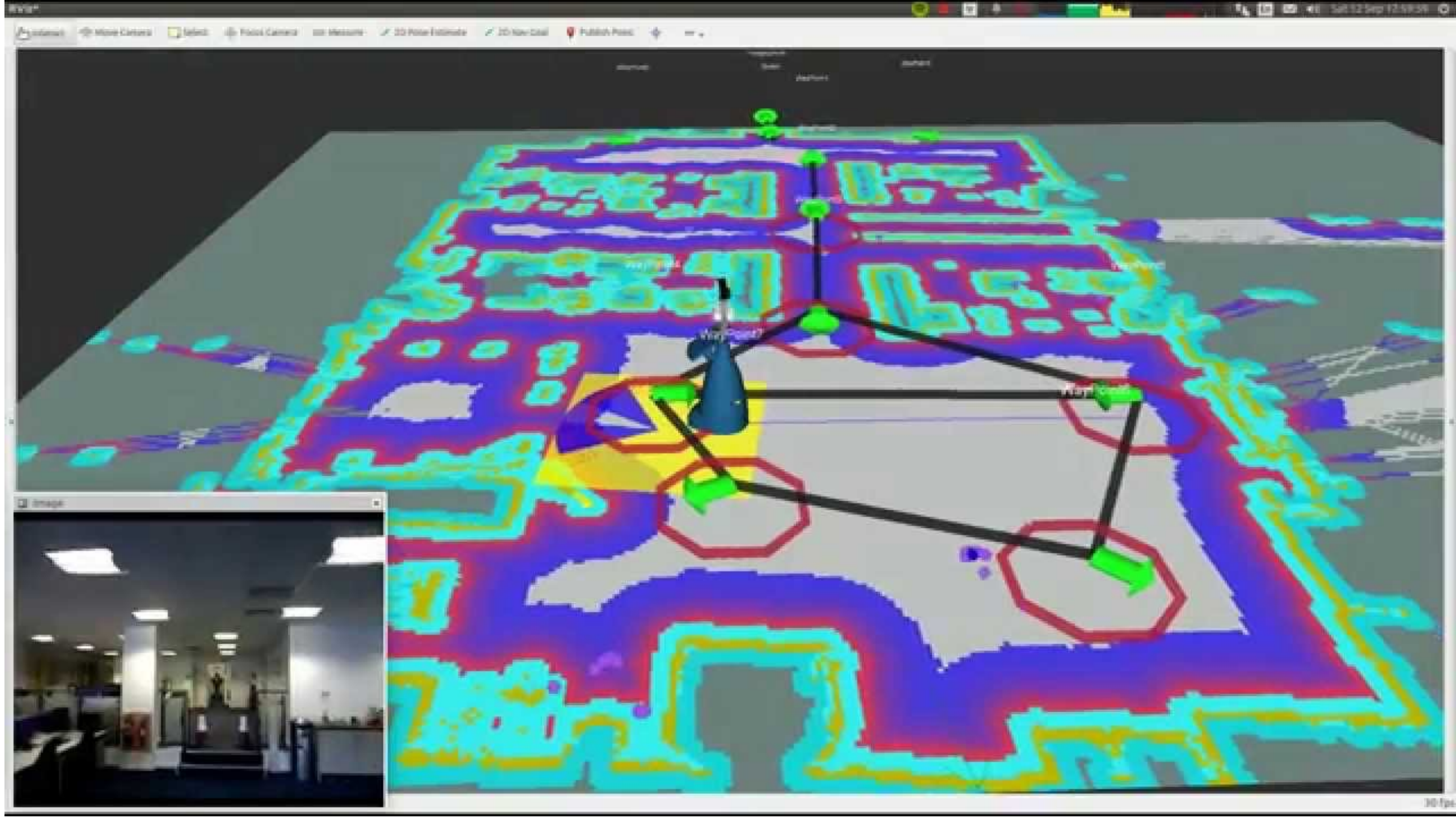
ROS

ROS Navigation Stack



Costmaps

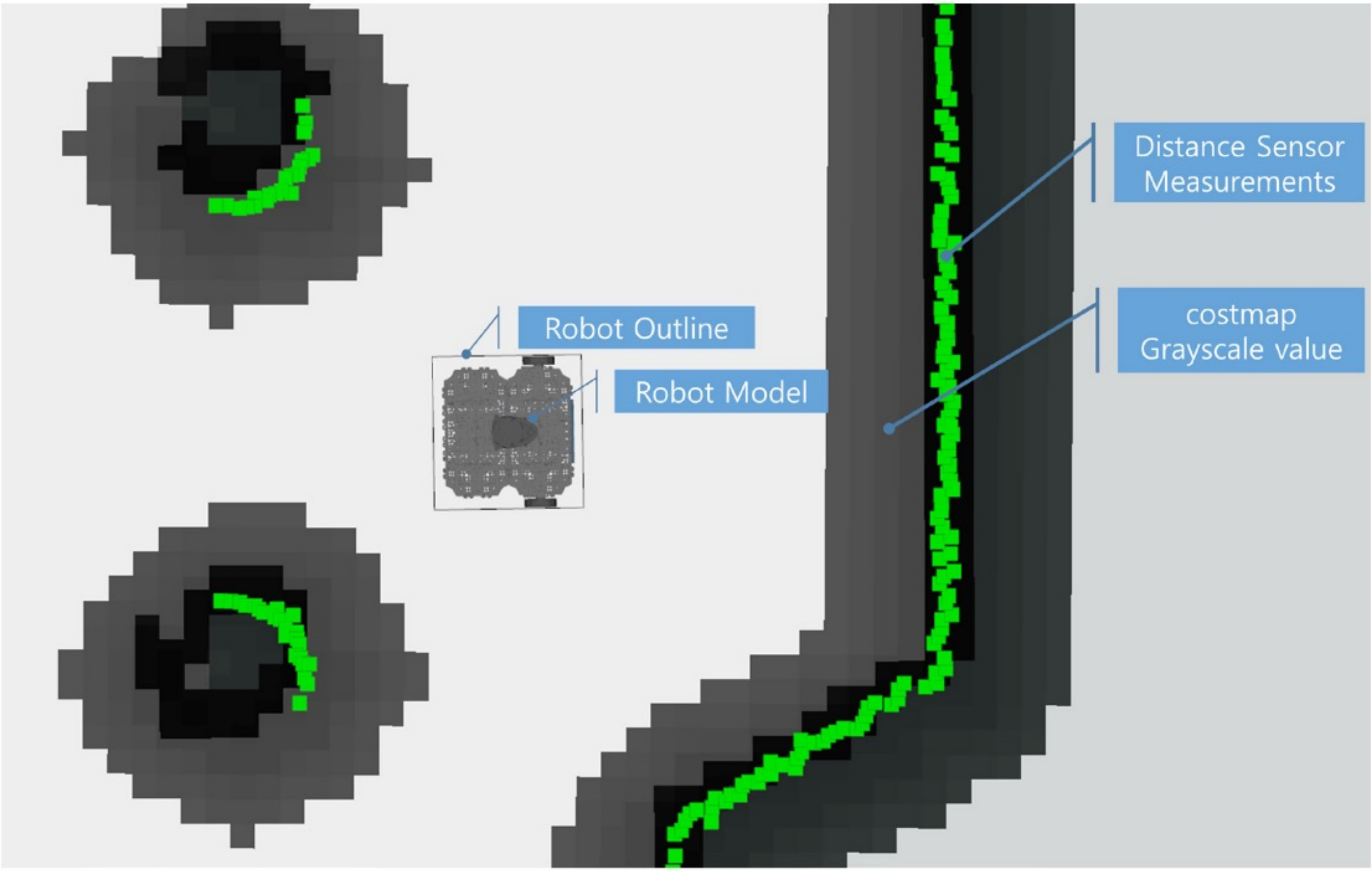
- the metric map is inflated according to the robot structure so the robot can perform a safe navigation



- in this way simpler paths in open areas are preferred to “costly” paths (close to obstacle or doors) where the robot may get stuck
- several methods to do so (e.g. robot footprint, inflate obstacles)

Costmaps

- the metric map is inflated according to the robot structure so the robot can perform a safe navigation



- in this way simpler paths in open areas are preferred to “costly” paths (close to obstacle or doors) where the robot may get stuck
- several methods to do so (e.g. robot footprint, inflate obstacles)

ROS

Deal with uncertainties and dynamics

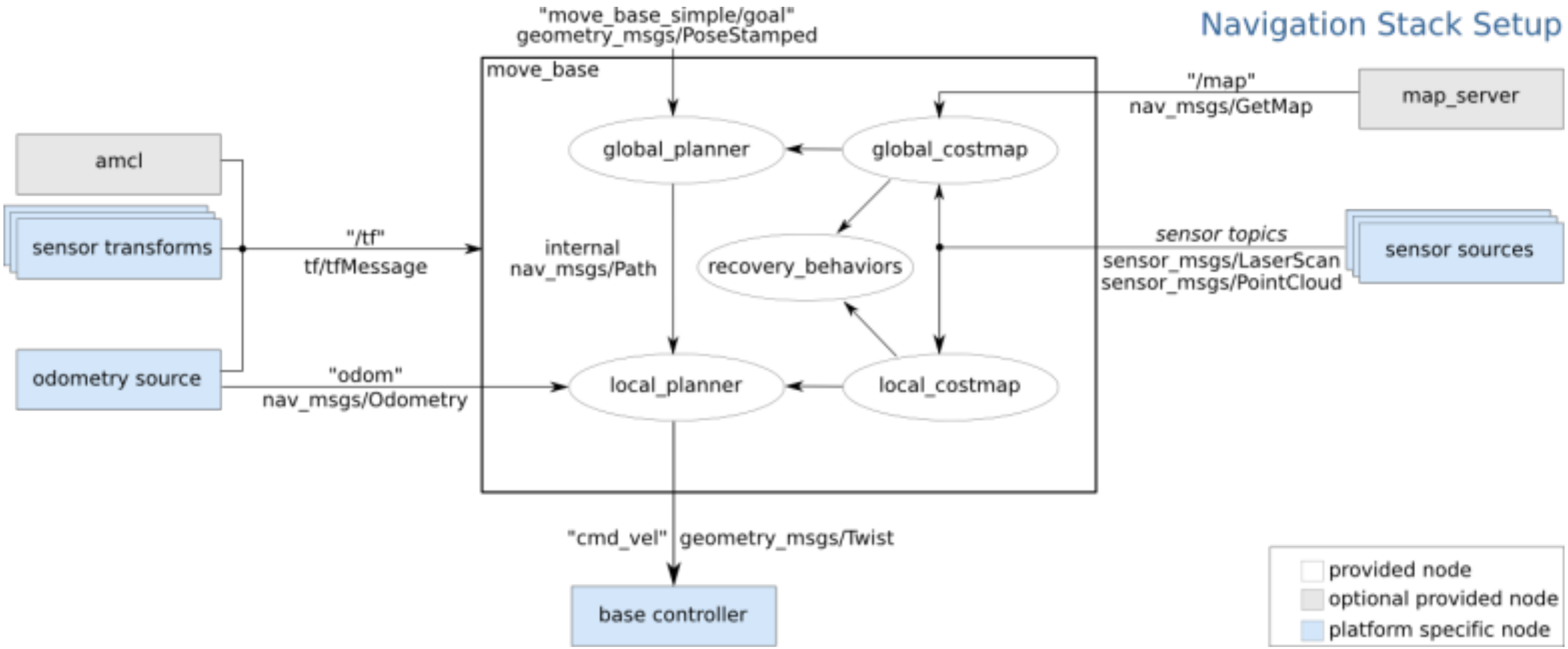
The robot plans its path in the static map but

- changes usually happens (doors open/close)
- new obstacles may appear (people, animals, children)
- the robot movements execution are very different w.r.t. the initial goal
- …

Solution: complement the (ideal) map with local information coming for sensors that address such issues

- obstacle avoidance
- local map refinement based on recent sensor readings

ROS Navigation Stack



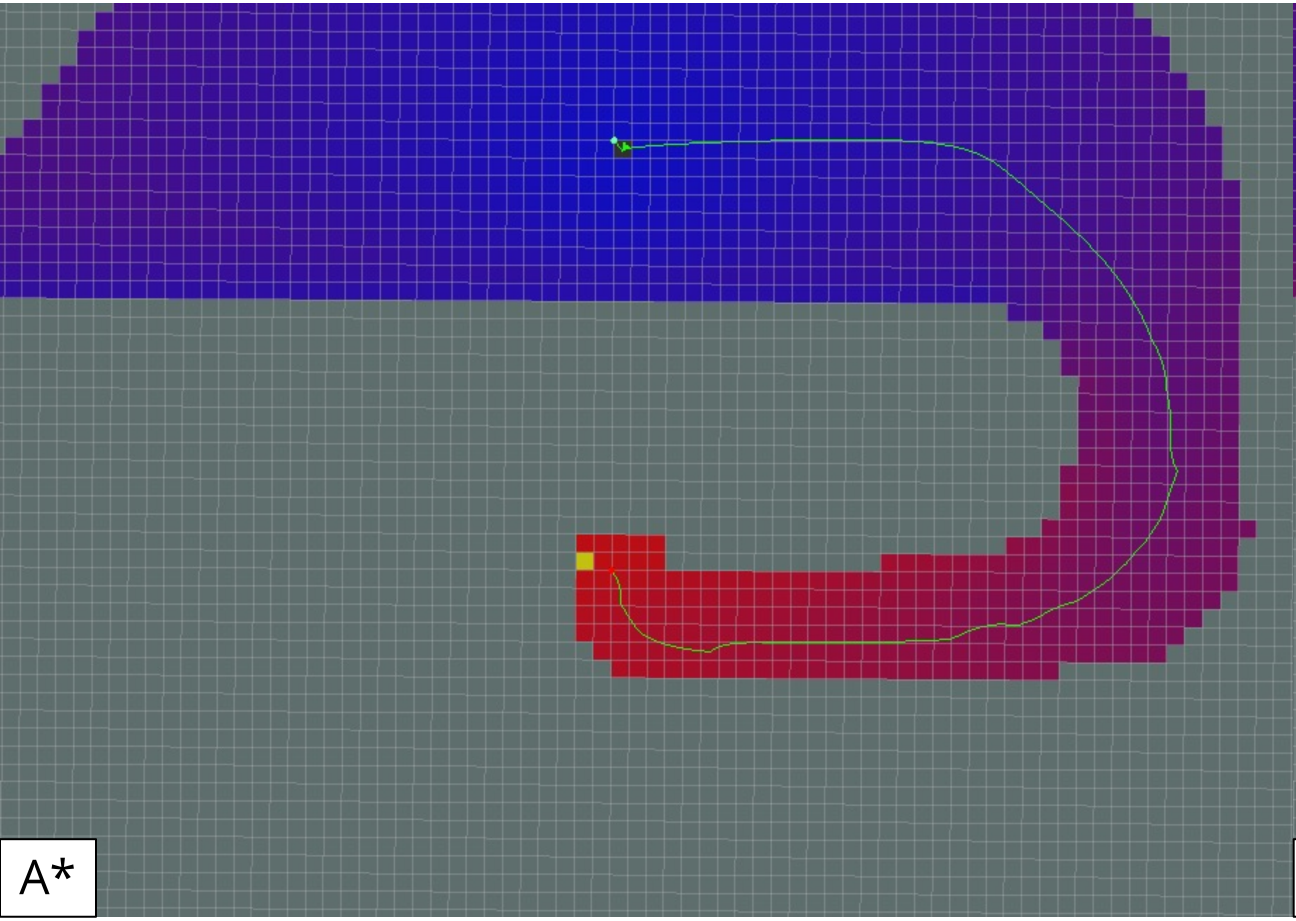
Global and local planner

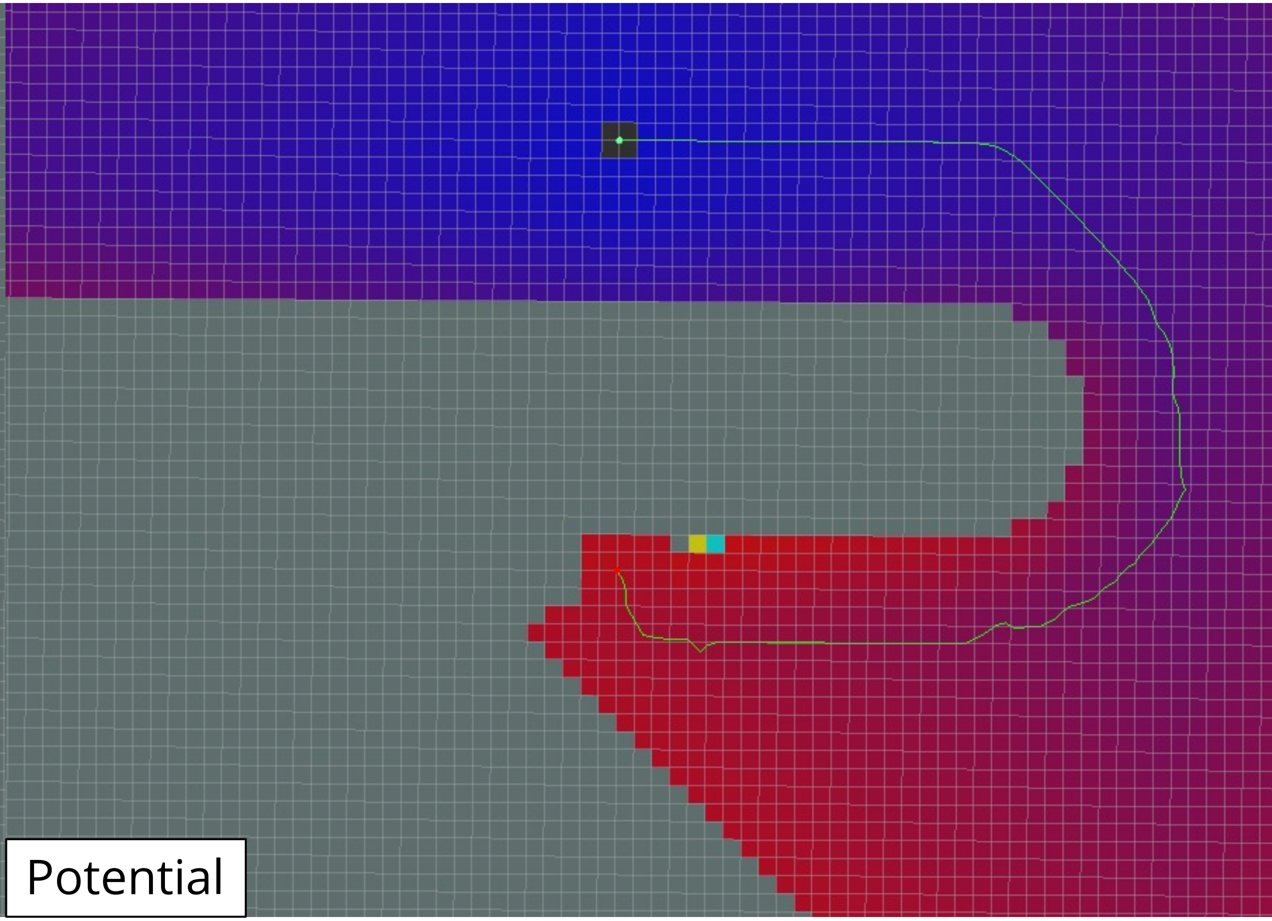
- global plan → identifies the long-term path that eventually will lead the robot to the goal
  works at low frequency, using 2D lidar data
- global costmap → used for path planning, based on the static metric map
- local plan → identify the next moves that the robot has to perform in order to follow the global path
  works at high frequency
- local costmap → centered on the robot, integrates all of the sensors of the robot (2D lidar, RGBD data, bumpers) that are needed to constantly adapt the local plan

Global planner and costmaps

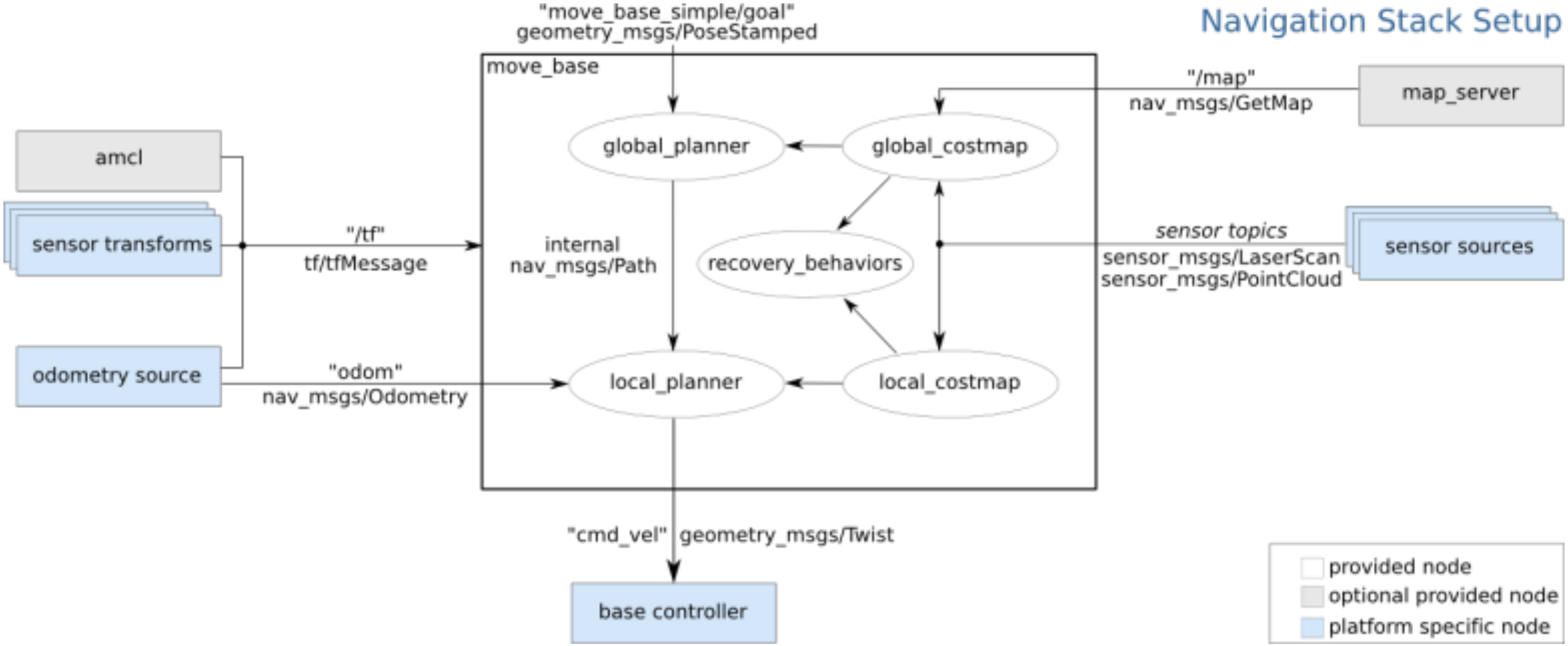
- implements several planning algorithms, use the one you want and that is most suited for your application
- costmaps also can be tuned in several ways according to your robot configuration
- you can visualize with RViz the path decided by the robot
- the global path could become outdated – replanning is also used

Global Planner





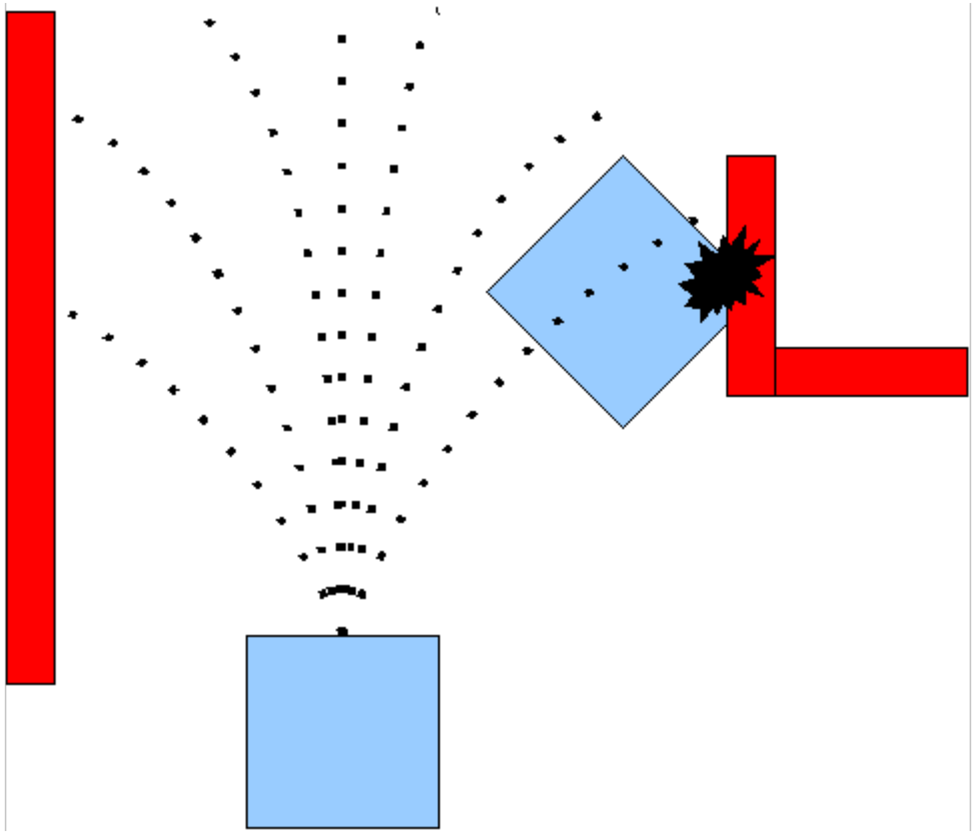
ROS Navigation Stack



ROS

DWA Local planner

The local planner package provides a controller that drives a mobile base in the plane and connect the path planner to the robot.
Using a map, the planner creates a kinematic trajectory for the robot to get from a start to a goal location. Along the way, the planner creates, at least locally around the robot, a value function, represented as a grid map. This value function encodes the costs of traversing through the grid cells. The controller's job is to use this value function to determine dx,dy,dtheta velocities to send to the robot.



DWA Local Planner

Dynamic Window Approach to local robot navigation on a plane. Given a global plan to follow and a costmap, the local planner produces velocity commands to send to a mobile base.

The basic idea of the Dynamic Window Approach (DWA) algorithm is as follows:

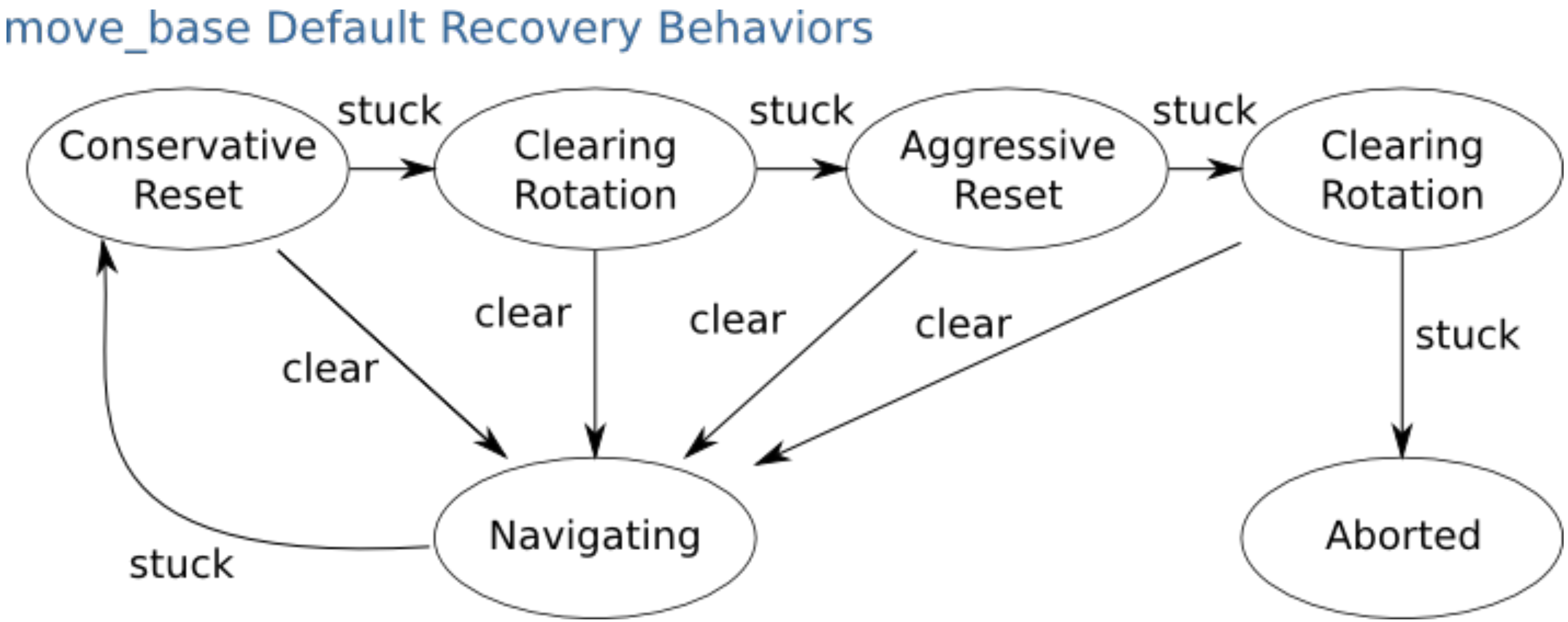
- discretely sample in the robot's control space (dx,dy,dtheta)
- for each sampled velocity, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time.
- evaluate (score) each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path, and speed. Discard illegal trajectories (those that collide with obstacles).
- pick the highest-scoring trajectory and send the associated velocity to the mobile base.
- rinse and repeat.

Handling failures

- despite the integration of local and global plan execution, the robot may get stuck → is not able to move and continue to execute its path
- this happens often in narrow passages (doorways), when a lot of rotations are involved, or with dynamic obstacles (people, is too close to an obstacle to safely move)
- the robot should be provided with mechanism to solve autonomously such issue and to restart following its path
- otherwise, a human intervention is needed

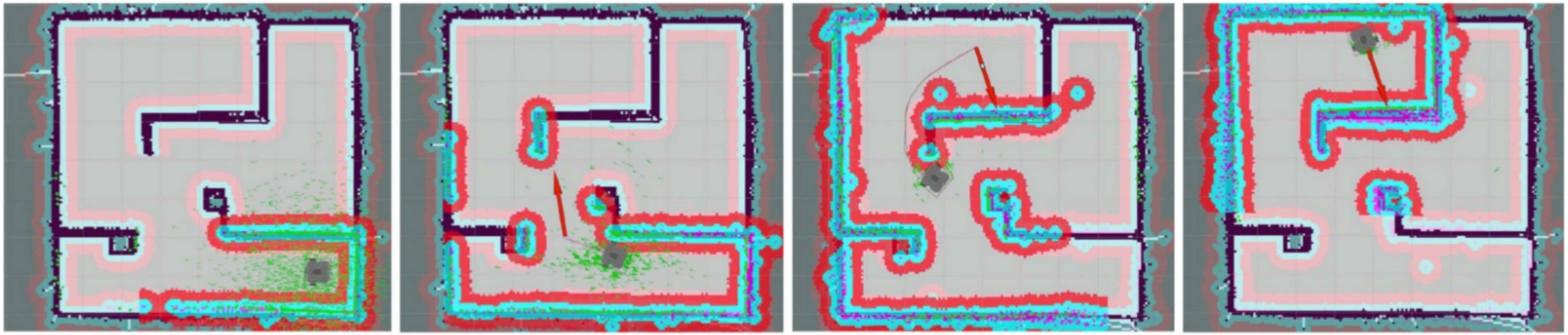
The navigation stack gives you a set of behavior for this

Recovery Behaviors



Recovery behaviors are executed when the robot is stuck or cannot proceed to the goal (cannot execute the path or cannot compute the path).
They try to free the robot from a dangerous position (e.g. too close to an obstacle) or to "clear" the costmaps (e.g. a noisy reading, a user was in front of the robot, an obstacle that was there and it is not there anymore)

An example of global and local costmaps + AMCL



(t1) (t2) (t3) (t4)

ROS

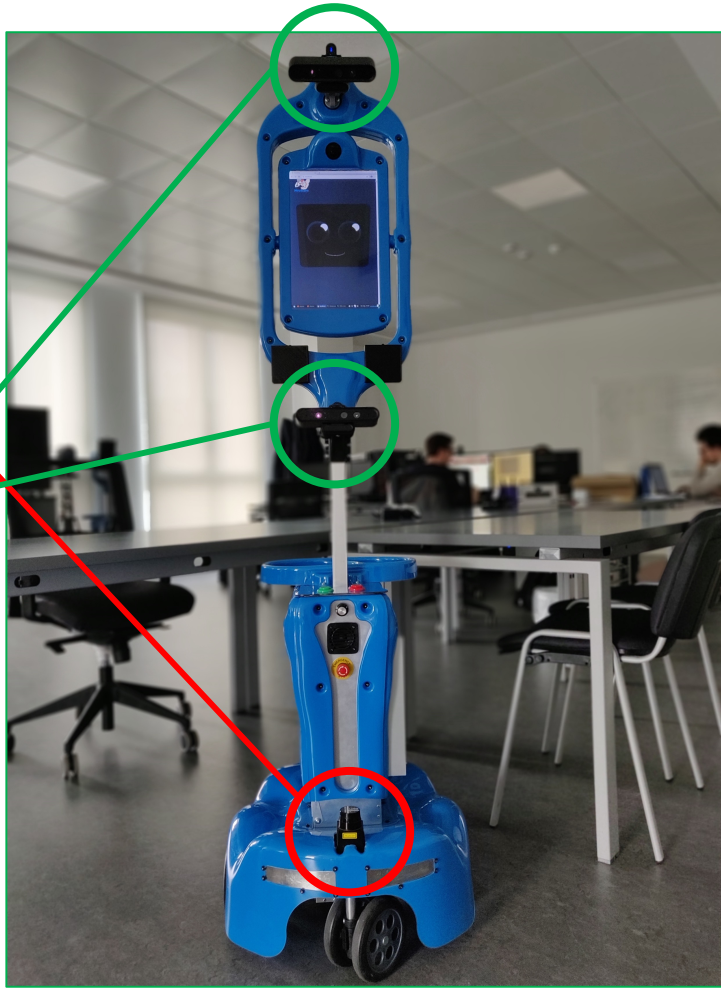
An example of navigation stack in use

Our Giraff robot intended to work inside houses, so with dynamic environment, people, and clutter

- gmapping for creating a (static) map
- AMCL for localization
- navigation Stack + custom-built packages for more robust navigation

An example of navigation stack in use

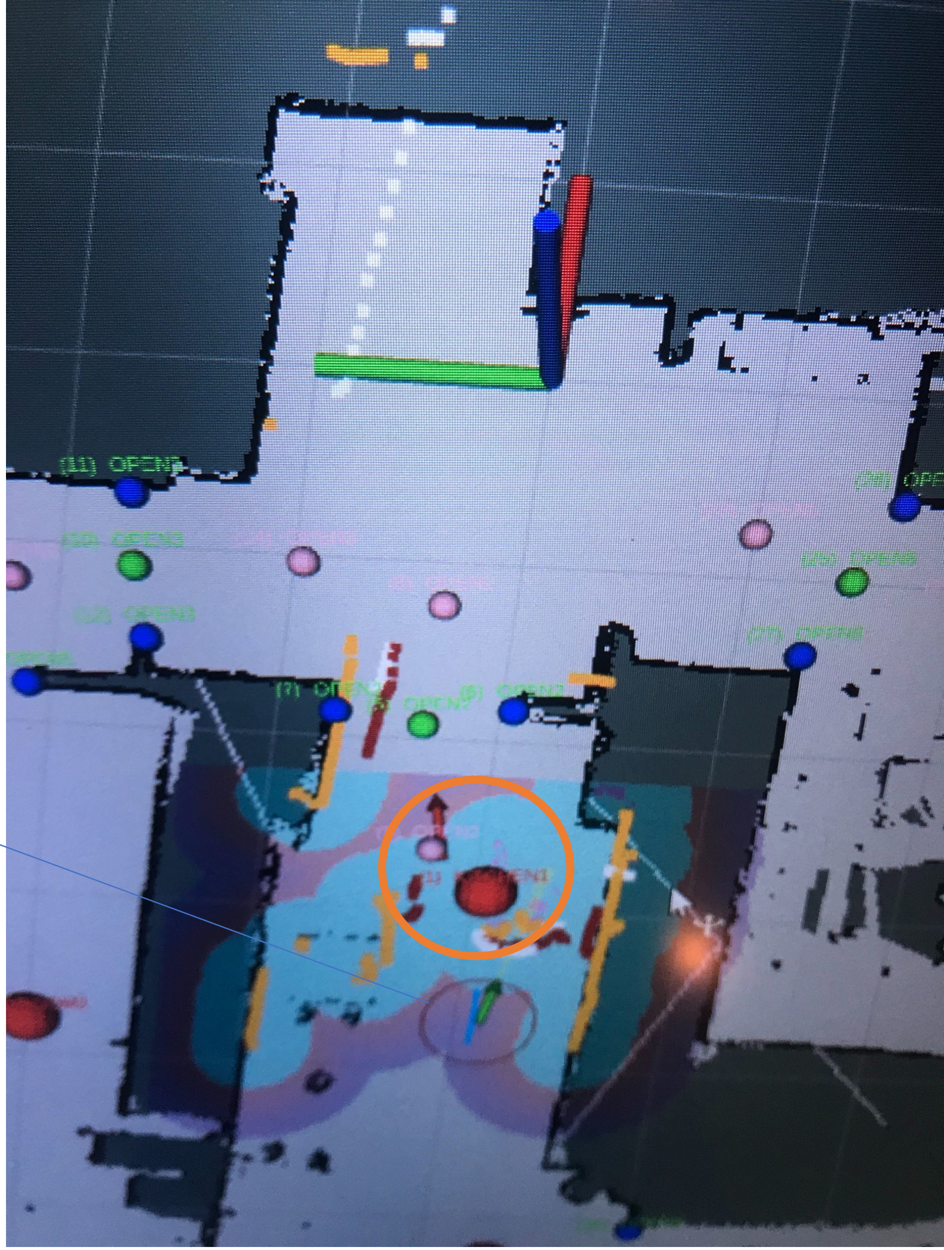
- the global costmap (and the map) is built using a 2D lidar 20cm from the ground
- the local costmap is integrated with depth sensor info projected at the 2D plane from 2 RBGD cameras, one pointing on the ground, the other one front-facing
- In this way the robot can detect people, tables, chairs, …



An example of navigation stack in use

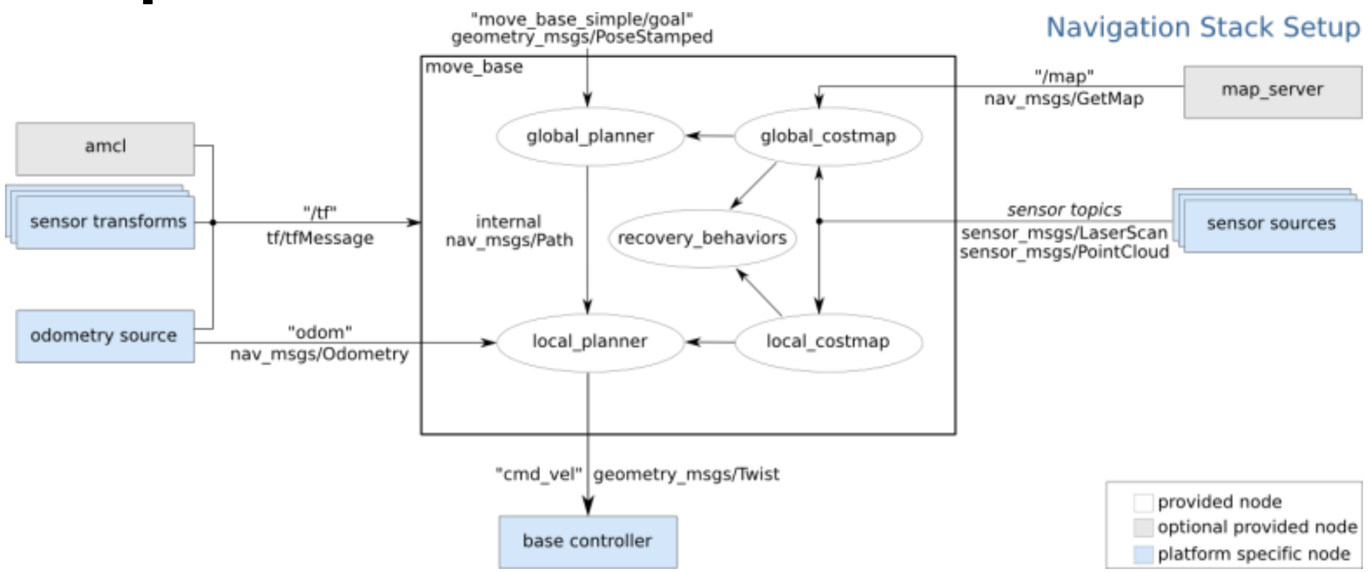
The local costmap is centered around the robot pose and integrates the sensors coming from the **2D lidar** and from the **RBGD cameras**.

A **person** is in front of the robot that, consequently, cannot move.



ROS Navigation wrap up

All the required modules for having a robot moving autonomously are available and ready-to use in ROS.

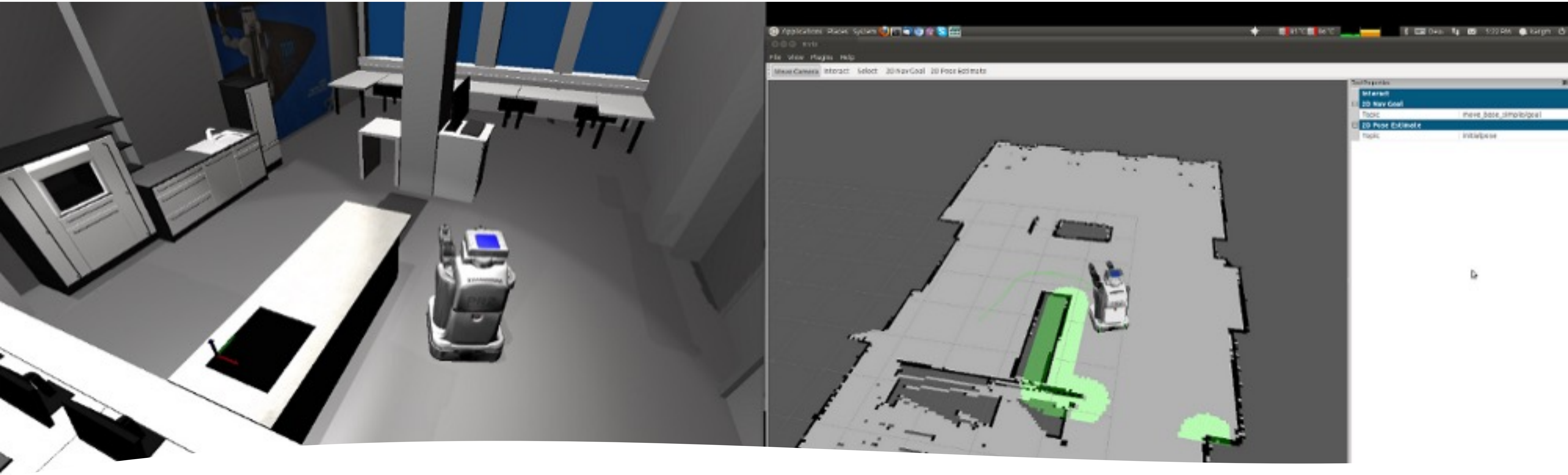


You just need to:

- select sensors and a robot
- pick up a SLAM algorithm and make a map
- use a localization mechanism
- use the navigation stack

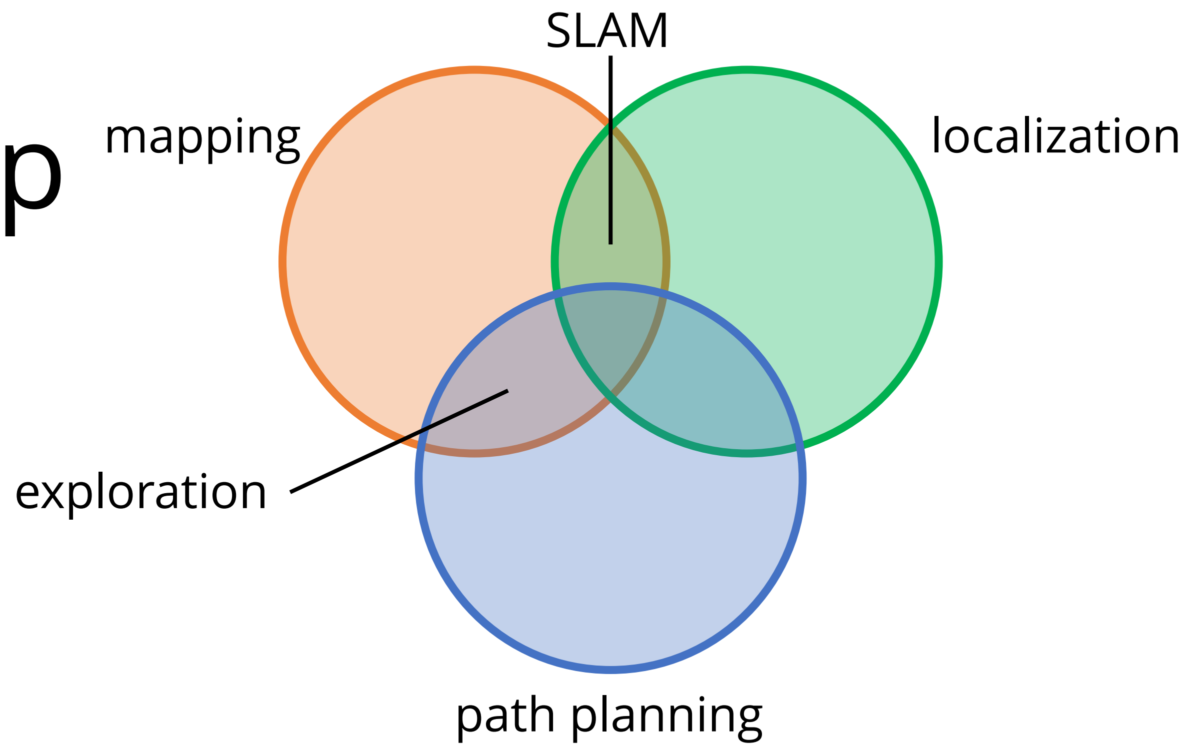
ROS Navigation wrap up

You can use simulations and test everything you've seen by yourself in a couple of afternoons



RViz is very useful to test and understand what happens.

ROS navigation wrap up



From [Grisetti, Burgard, Stachniss]

Remember that navigation is a core behavior of the robot, on top of which more complex ones are built.

E.g. exploration is the task of building a map of an unknown environment, making decision about the next position that has to be reached (next path planning goal).

Sources - References

- Wiki.ros.org
- ROS Robot Programming
  A Handbook Written by Turtlebot3 Developers
  (available at http://www.robotis.com/service/download.php?no=719)
- Robotis Turtlebot3 documentation
  http://emanual.robotis.com/docs/en/platform/turtlebot3/getting_started/
- Jason O'Kane, A Gentle Introduction to ROS
  https://cse.sc.edu/~jokane/agitr/agitr-letter.pdf

Sources - References on navigation

ROS wiki navigation stack home page, with a lot of tutorials and documentation

- http://wiki.ros.org/navigation
- http://wiki.ros.org/navigation#Tutorials
- http://wiki.ros.org/navigation/Tutorials/Using%20rviz%20with%20the%20Navigation%20Stack