An introduction to ROS

Michele Antonazzi
Dipartimento di Informatica
michele.antonazzi@unimi.it

ROS: the Robot Operating System

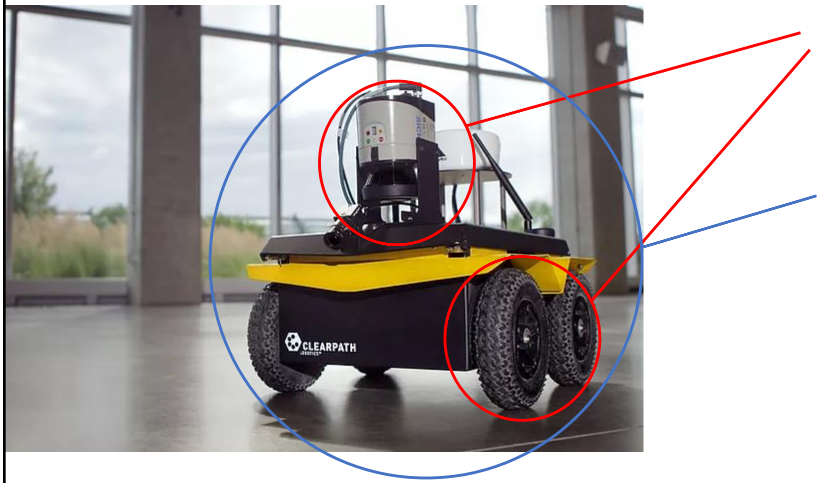
ROS is an *open-source*, **meta-operating** system for your robot. It provides the services you would expect from an operating system:

- including hardware abstraction
- low-level device control
- message-passing between processes
- package management
- tools and libraries for writing, building, and running code across multiple computers

Ros as Middleware

- **Modular** and **scalable** software architecture for both **low-level** and **high-level features**
- ROS is the de-facto standard for robot development

ROS as Middleware



Low level functionalities:

- as real-time motor controllers
- sensors drivers
- battery management

+

Core functionalities:

- mapping
- localization
- navigation

+

Reasoning mechanisms:

- path planning
- task allocation
- self management

ROS Features

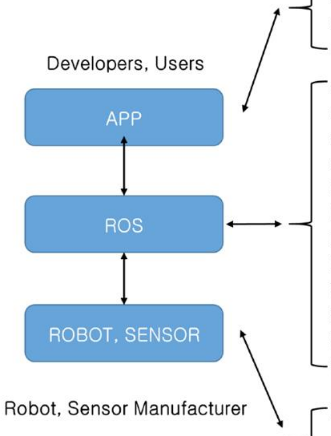
ROS is a *meta-operating* system that offers useful features:

- Scheduling, loading, monitoring, and debugging processes
- **Virtualization** layer which allows distributed computation
- Runs on top of (multiple) operating system(s)
- Enforce a **modular** software architecture

More ROS Features

- **Code reuse:** the modularity allows to use the same code for more functionalities
- **Thin:** ROS is designed to be as thin as possible
- **Integration:** with other frameworks and libraries
- **Language independent:** core languages are Python and C++ but you can use what you want
- **Scaling:** ROS tools can be distributed across different machines and is appropriate for large development process

Ros Community



- ❑ More than 5,000 packages
- ❑ 2,818 Official Packages (Indigo, March 2017)
- ❑ 13,441,711 .deb Packages Download (July 2017)
- ❑ 18,839 Wiki page (July 2017)

❑ Applications	fetch beer, elevator …
❑ Simulators	gazebo, player/stage, STDR Simulator …
❑ Intelligent Modules	navigation, action, grasping …
❑ Libraries	tf, PCL, OpenCV, OpenRave …
❑ Device Drivers	camera_drivers, urg_node …
❑ Debugging Tools	rviz, rqt_graph, rosbag, rostopic …
❑ Message Passing	rosmaster, rosmsg, rosservice …
❑ Execution Tools	rosrun, roslaunch …
❑ Compile Tools	catkin_make, rosbuild …
❑ File Systems	roscd, rosls …
❑ Installer	rosinstall …
❑ Programming Languages	C++, Python, Lisp, Java, Ruby, MATLAB, etc

- ❑ Supports more than 90 robots and 80 sensors

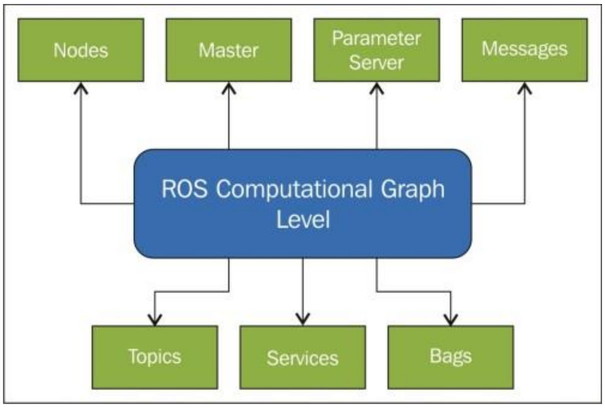
Core aspects of

ROS

ROS Software Architecture

ROS implements a **distributed architecture,** called **Computational Graph:**

- It is **peer-to-peer** network of ROS processes
- Each component **provide or use data** to the graph in different way



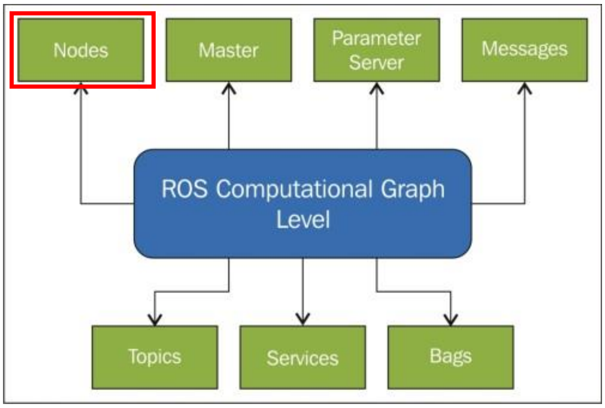
Nodes

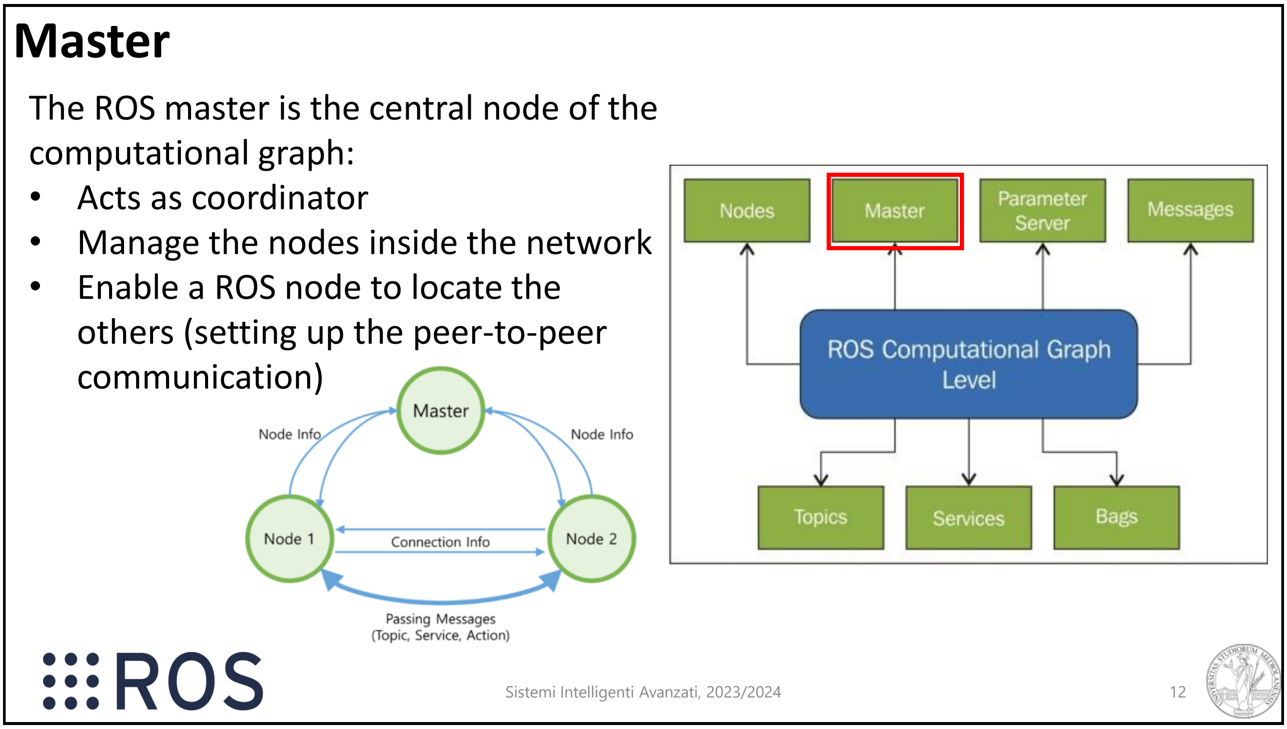
A *node* = process:

- Solves a precise sub-task
- Nodes collaborate with each others

Benefits:

- **Divide-et-impera:** the *code complexity* is reduced in comparison to monolithic systems
- **Encapsulation:** the code complexity is hidden inside nodes, that expose easy APIs
- **Fault tolerance:** the crashes are isolated
- **Substitutability:** changing implementations or language

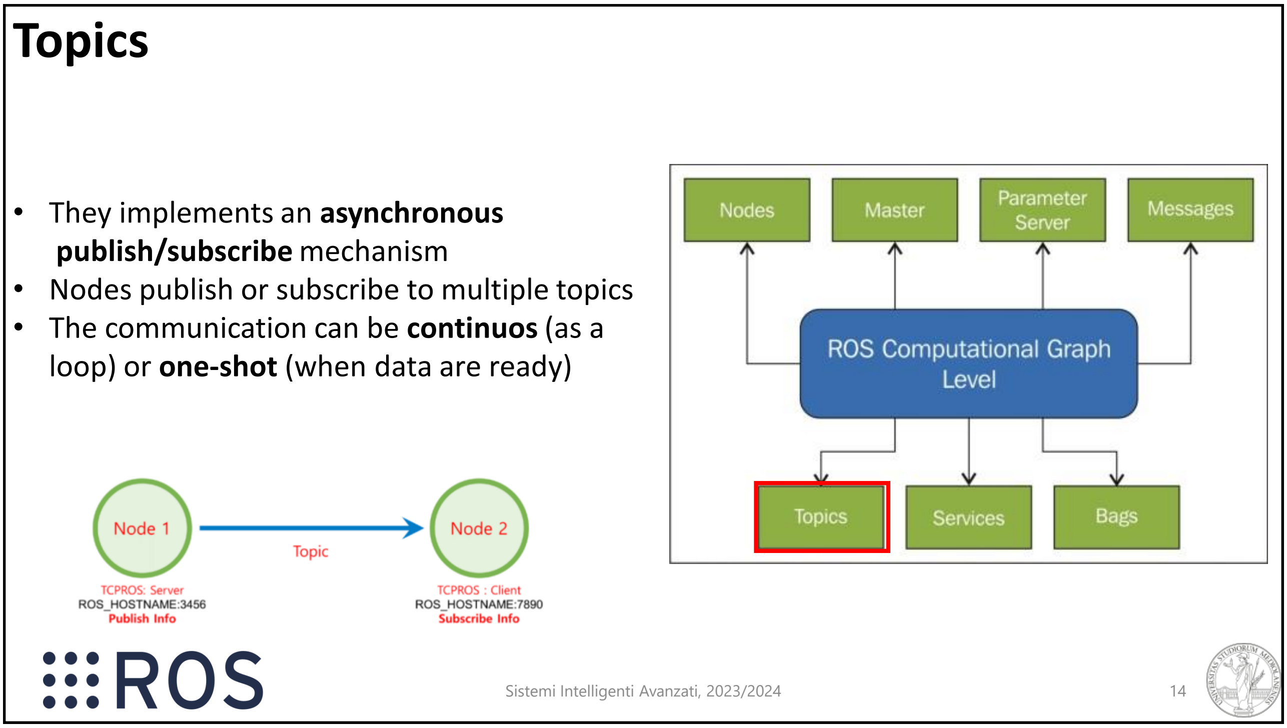




Master

The ROS master is the central node of the computational graph:

- Acts as coordinator
- Manage the nodes inside the network
- Enable a ROS node to locate the others (setting up the peer-to-peer communication)



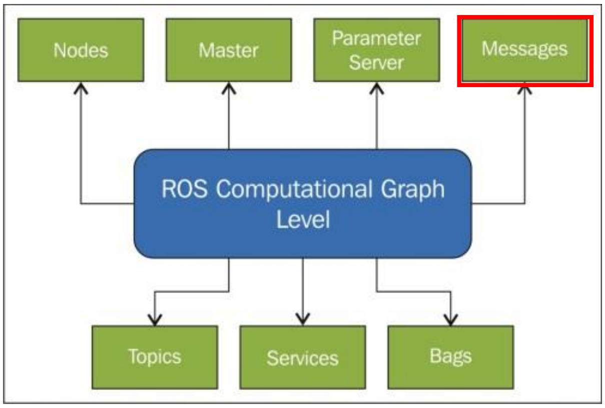
Topics

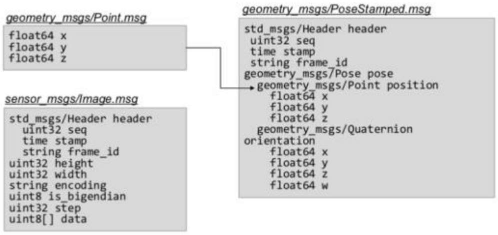
- They implements an **asynchronous publish/subscribe** mechanism
- Nodes publish or subscribe to multiple topics
- The communication can be **continuos** (as a loop) or **one-shot** (when data are ready)

Messages

A message is a data structure

- Standard types (integer, string, …) or array
- A message type can include arbitrarily nested structures of types, array, and other 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
```

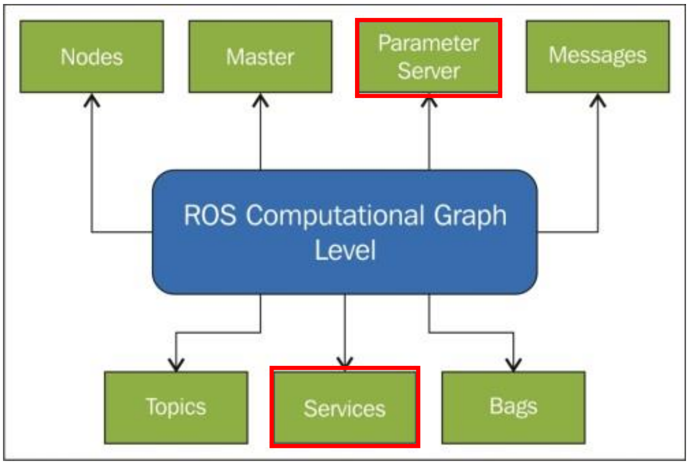
geometry_msgs/PoseStamped.msg

```
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
```

Other Communication Modalities

4 types of interactions

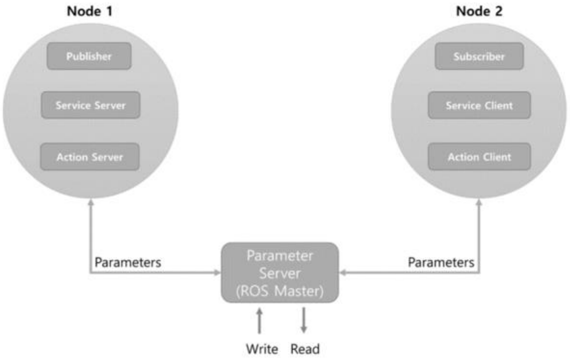
- Topics
- Parameters Server
- Services
- Actions



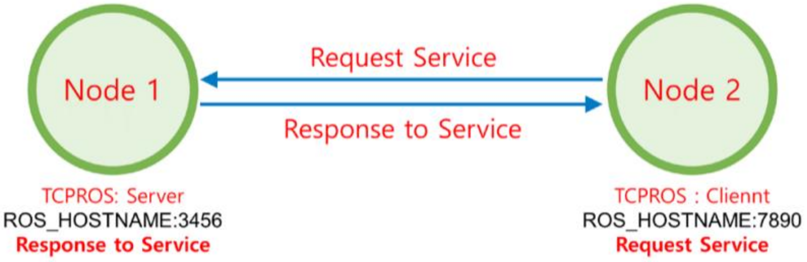
Communication with Parameter Server

Share data in a central location (master).

- Globally viewable, it is useful to inspect and modify the global configuration the system



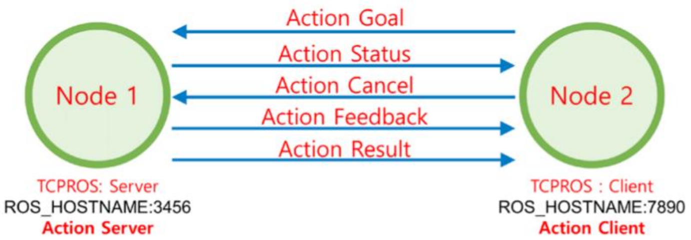
Communication with Services



A service implements a **request/reply** interaction:

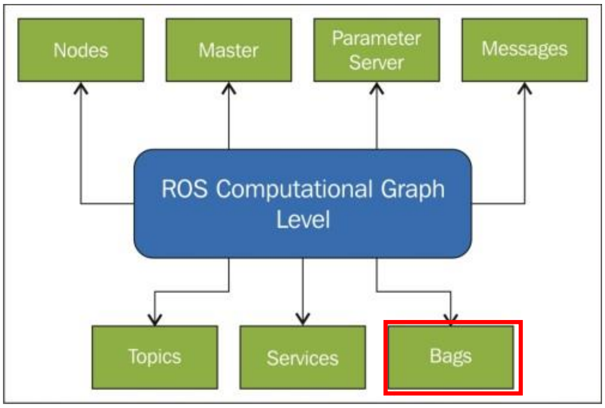
- **Synchronous** requests are sent to a server node from client node
- A service is defined by a pair of message structures, one for the request and one for the response

Communication with Actions

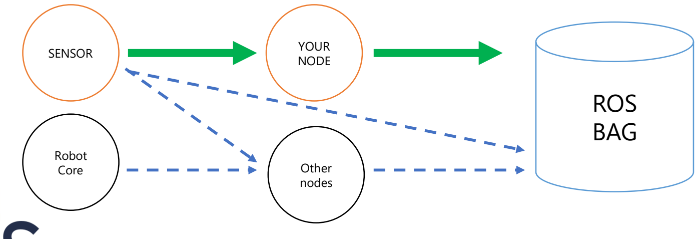


- ROS actions are **asynchronous** requests from one node to another
- If a service takes a long time to execute, the client might want the ability to cancel or get periodic feedbacks while a request is progressing

Bags



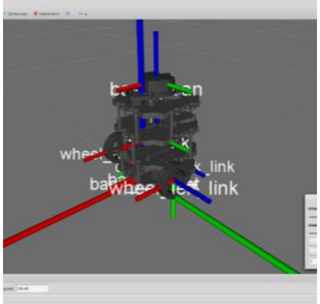
- Bags allow to record and save robot runs (in simulation or in the real world)
- A saved bag can be run in another moment for developing or testing new functionalities

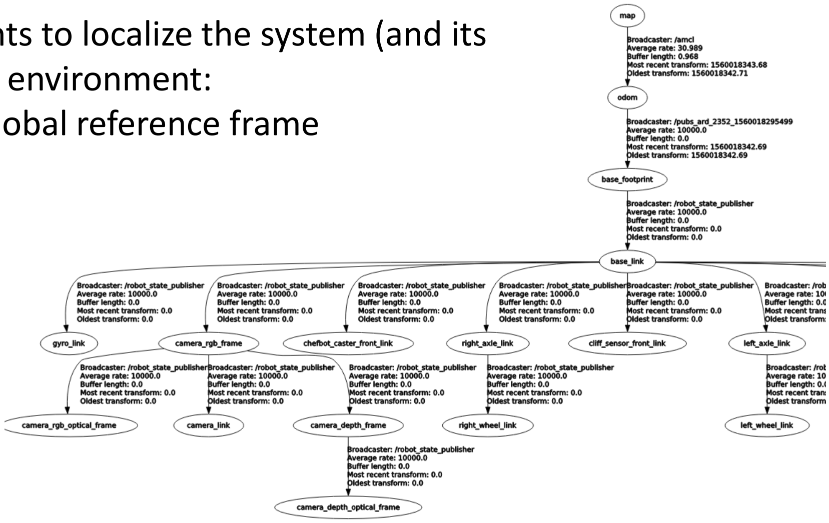


Transforms

Defines a chain of joints to localize the system (and its components) into the environment:

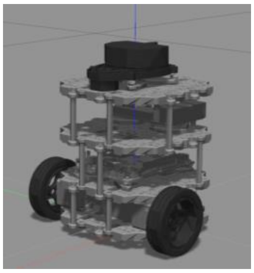
- Chain starts from global reference frame
- Static or dynamic





Support to Simulation

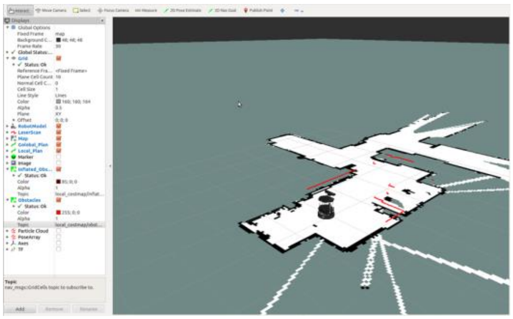
- One of the most powerful tool that ROS have is the possibility to use integrated 2D and 3D simulations
- ROS simulation nodes replaces sensor drivers and allows to test the same algorithm with real robot and simulations

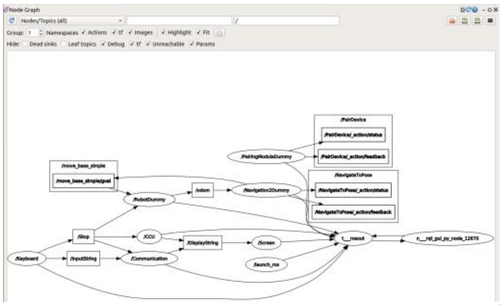


Debugging Tools

ROS offers some useful debugging tools for monitoring a complex robotic application:

- Command line tools:
  - **Rostopic** to control topics (echo, list, pub)
  - **Rosnode** to control nodes (start, stop, …)
- Visual tools:
  - **Rviz** provides a dynamic view of the status of the system (nodes, messages, etc)
  - **RQT** furnishes a static view of the system (computational graph, TF, etc)

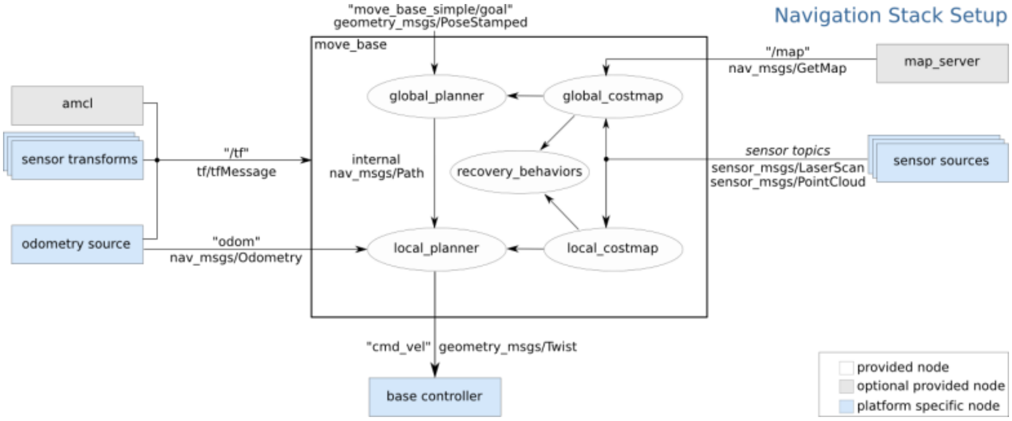




Autonomous navigation with ROS

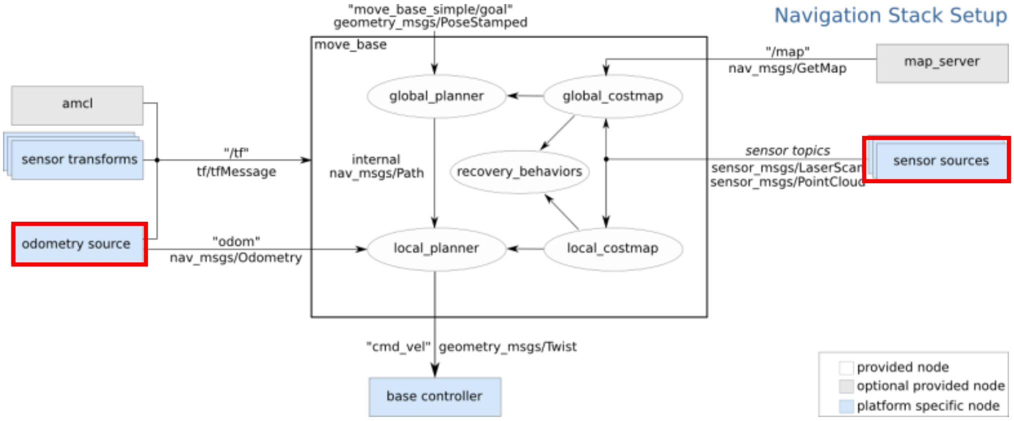
ROS Navigation Stack

ROS provides many nodes to perform navigation (sensors' controller, localization modalities, mapping strategies, planning algorithms, etc). In a robot, all these nodes are organized in a structure called Navigation Stack (move_base is the most famous)



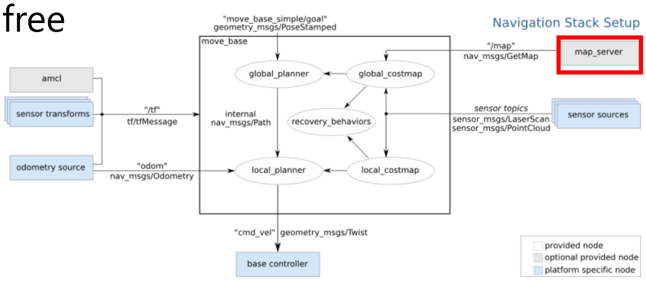
Robot Sensors

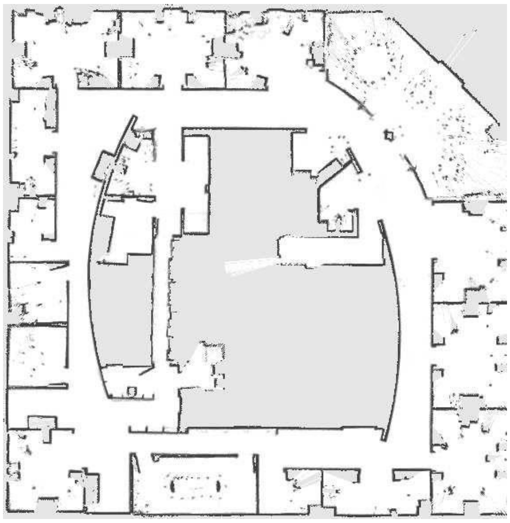
The robot proprioceptive and exteroceptive sensors are used to calculate the motion (odometry) and perception model.



Map

- The map is an abstract representation of the environment where the robot is operating
- Represented as a grid map: the environment is discretized in equal sub-portions that can be occupied or free

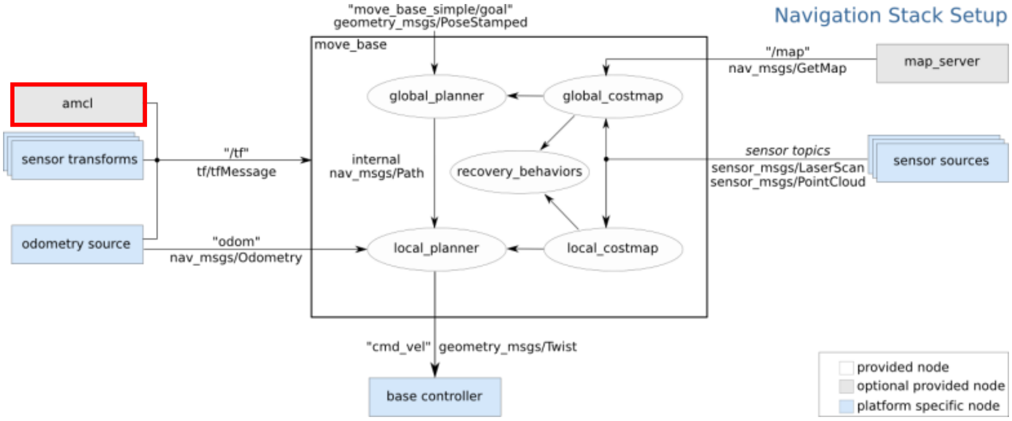




ROS

Localization

A widely used method for localization is **AMCL** (Adaptive Monte Carlo), which is already implemented in ROS.

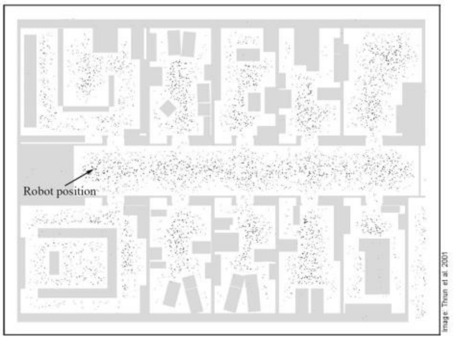


ROS

Adaptive Monte Carlo Localization (AMCL)

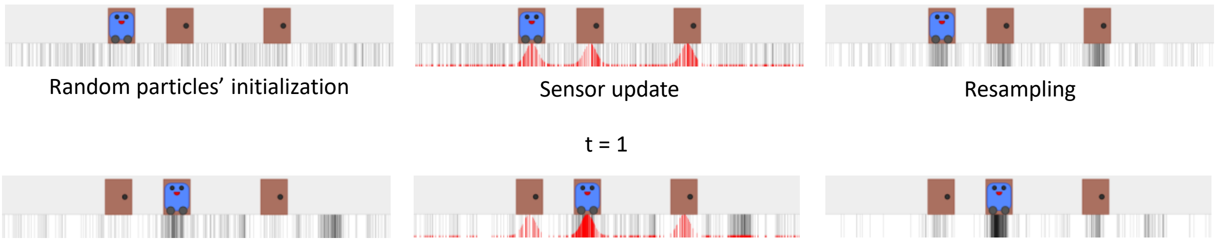
- It is a Markov Localization Algorithm implemented with a **particle filter.**
- Particle are updated with the motion and perception models
- The particles are **resampled** around the previous particles with high probability.

Monte Carlo Localization (MCL)



Adaptive Monte Carlo Localization Example

t = 0



Random particles' initialization

Sensor update

Resampling

t = 1

Motion update

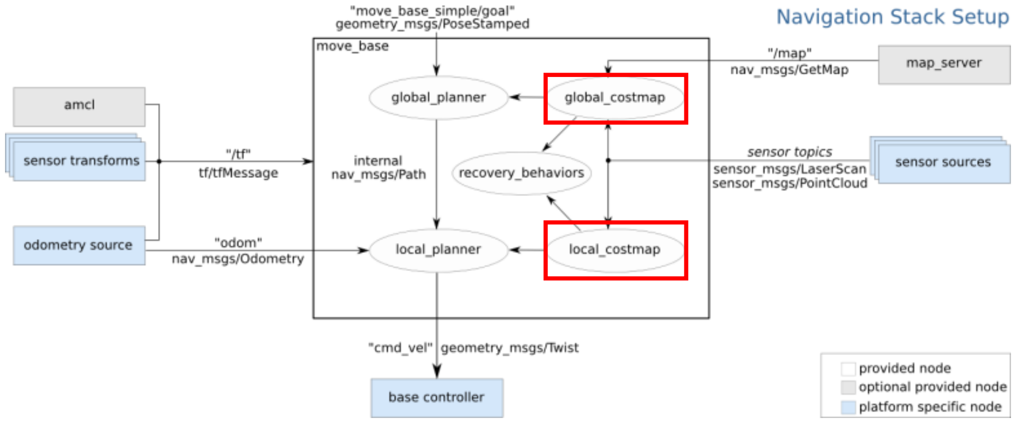
Sensor update

Resampling

t = 2

Costmap

To perform a safe navigation, the grid map of the environment is unflated with costmaps.

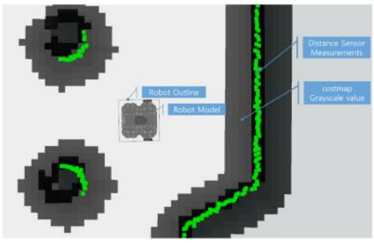


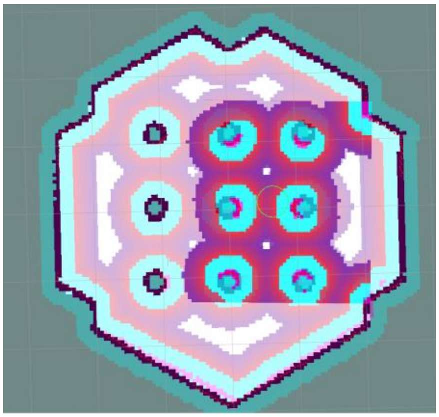
ROS

Costmap

A costmap is a grid map

- To each cell is assigned a specific value of cost: higher cost = smaller distance between the robot and an obstacle
- A costmap can be **local** (centered on the robot position and **frequently updated**) or **global** (centered on the map)
- Costmaps are used by the **planners** and to model the **obstacle uncertainty**

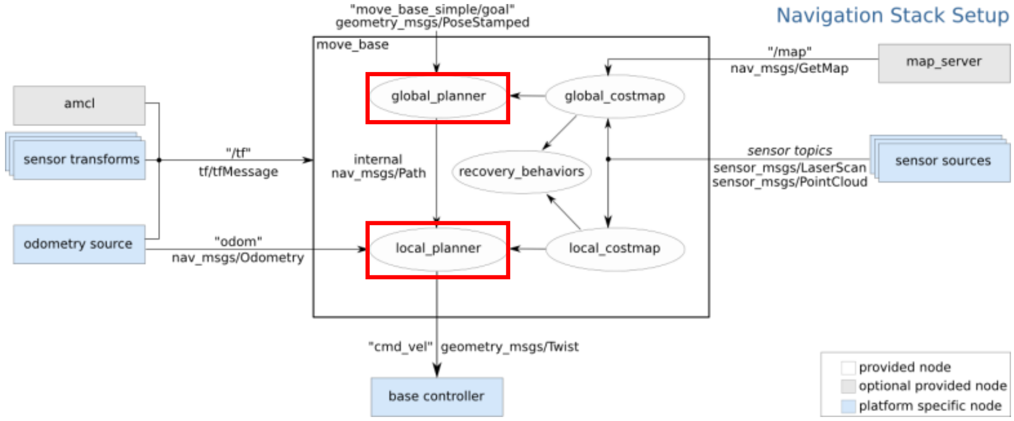




ROS

Planners

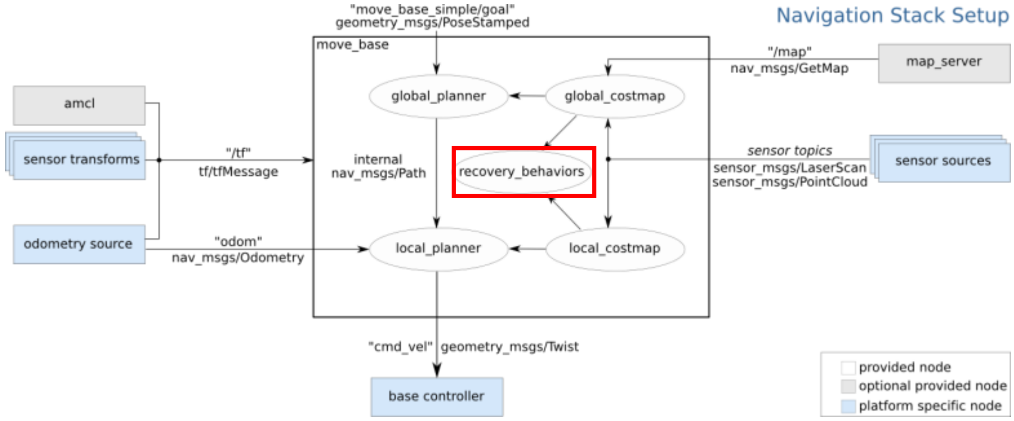
The global and local planners compute a sequence of sub-goal to reach a final goal. They minimize a cost function based on the (local and global) costmaps to avoid obstacles, performing a safe navigation



:::ROS

Recovery Bahaviors

Recovery behaviors are executed when the robot is stuck or cannot proceed to the goal (cannot execute the path or cannot compute the path) because the localization fails or it is too close to an obstacle.

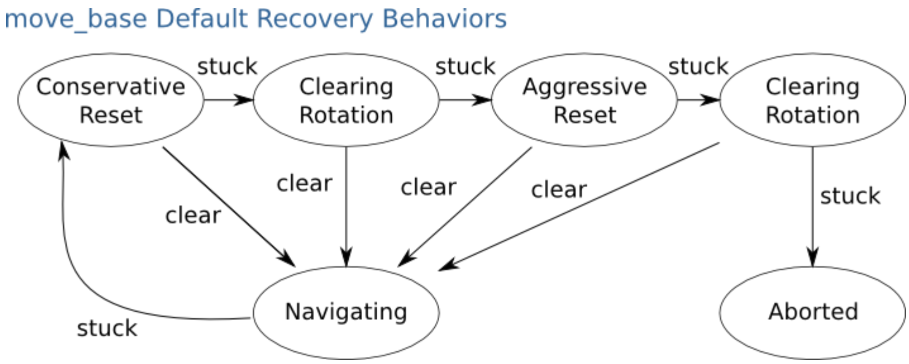


:::ROS

Recovery Behaviours

Standard recovery behaviors are:

- Conservative reset: reset the local costmap and try to recompute the local plan
- Clearing rotation: the robot rotates to relocalize itself
- Aggressive reset: reset the entire navigation stack



Ros Tutorial

Installation:

- Ubuntu 16.04 → ROS Kinetic
- Ubuntu 17.04 → ROS Lunar
- Ubuntu 18.04 → ROS Melodic
- Ubuntu 20.04 → ROS Noetic https://wiki.ros.org/noetic/Installation

ROS tutorials link:

- Create (link) and build (link) a ROS package
- Turtlesim tutorials (nodes, topics, rosservices)
- Create Publisher and Subscriber nodes in c++ or python
- Create a Service and Client in c++ or python

Turtlebot 3 tutorials link:

- SLAM
- Navigation