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

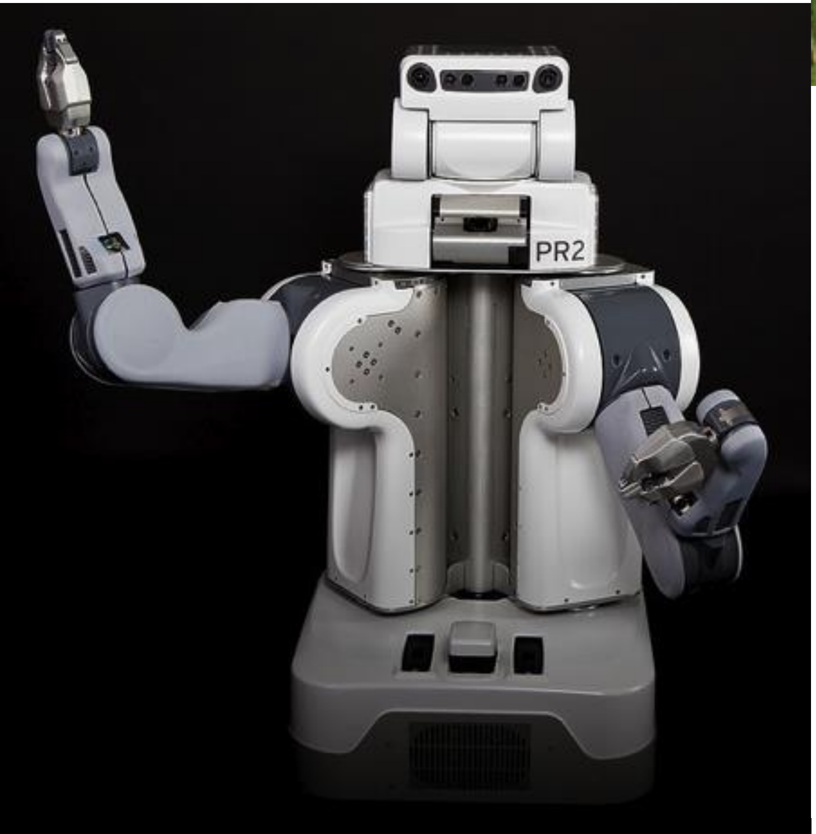
The development of (even a single) robots (functionality) requires both low-level hardware related and high-level AI-based mechanism

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

ROS provide this

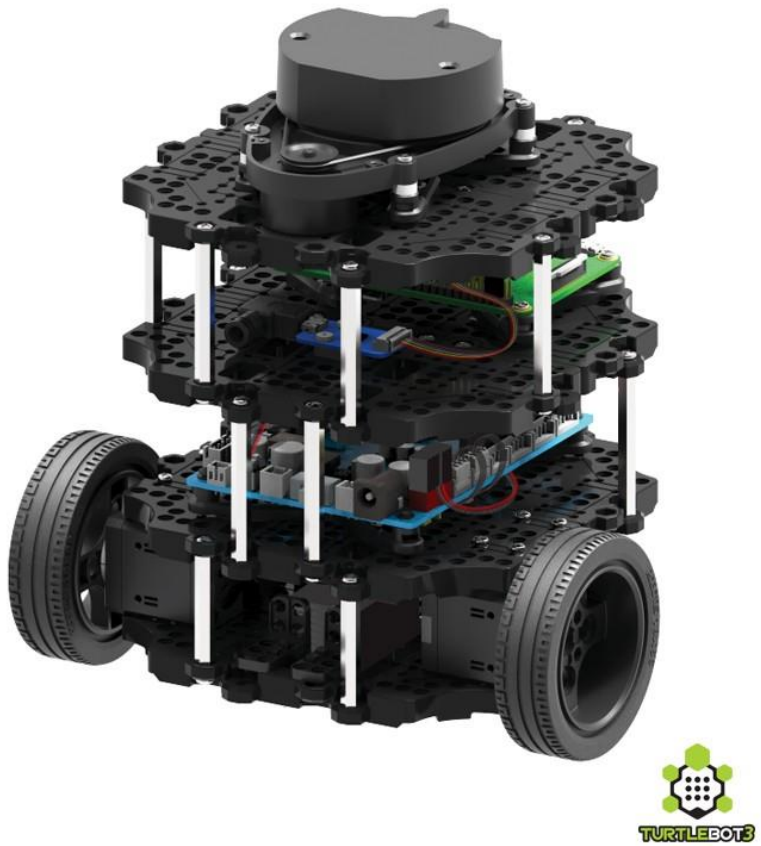
ROS has established itself as the de-facto standard for robot development

ROS



More at:
robots.ros.org

Our ROS robots



ROS

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

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.

ROS

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

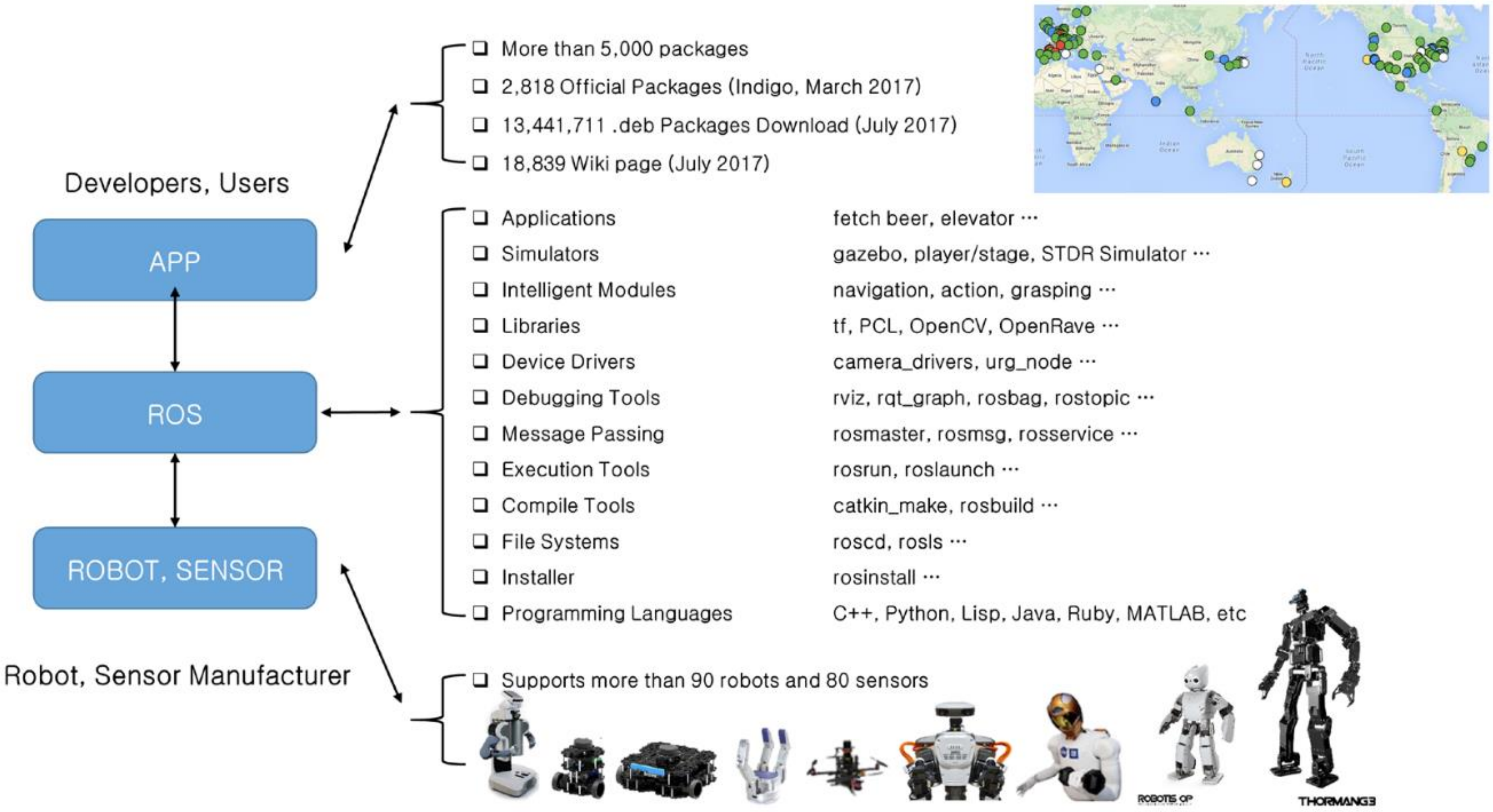
The core idea behind all of this is: <u>code reuse + modularity</u>

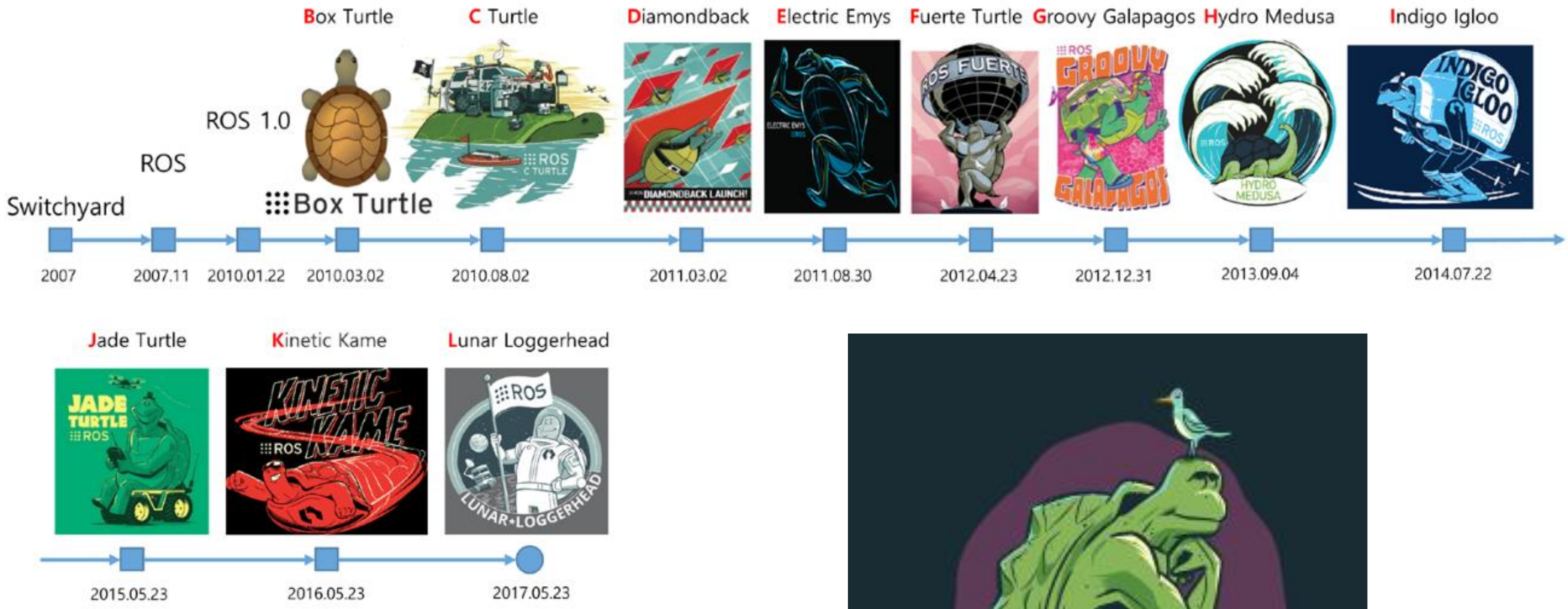
What ROS provides

- 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

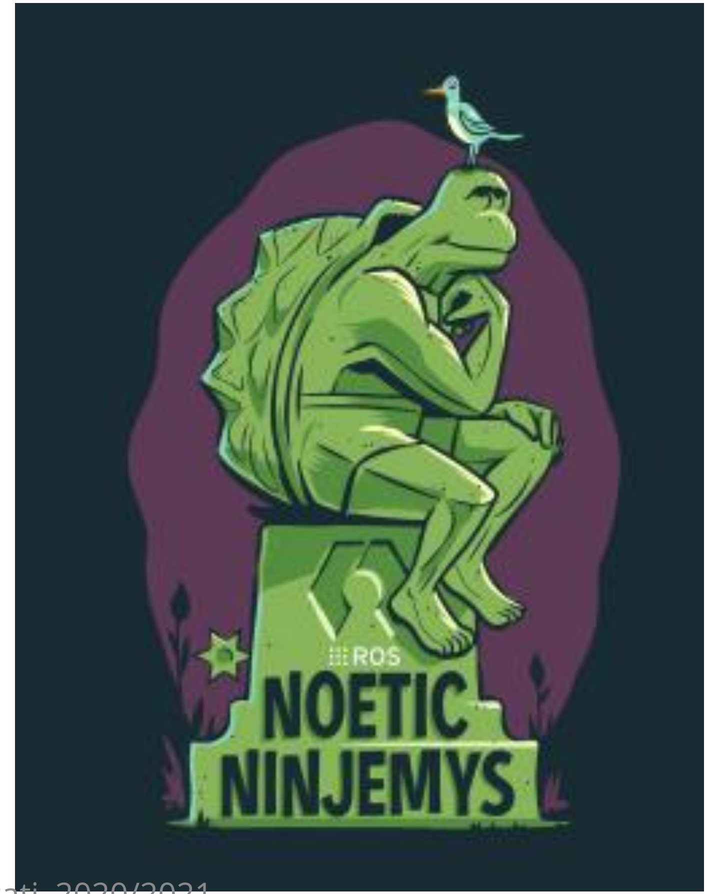
All free and ready to use
Support from the community

ROS-community





- more than 10y of ROS now
- last version (ROS1): ROS Noetic (2020)
- EOL 2025
- next mayor release: ROS 2
  - first version already released
  - migration at the beginning



ROS

Core aspects of ROS

ROS aspects

- nodes
- topics
- messages

Building blocks of ROS

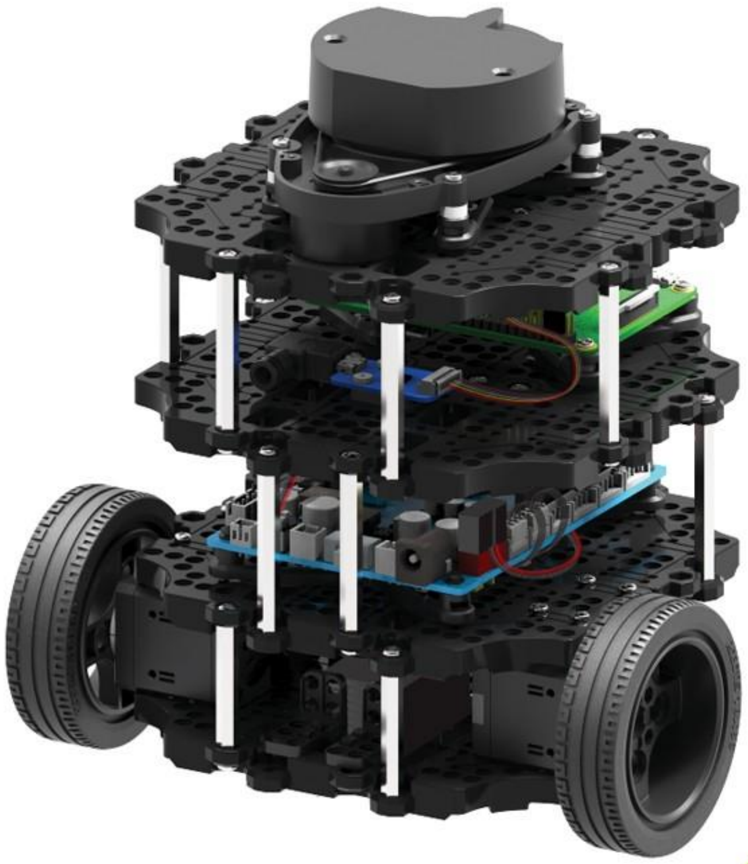
- services
- actions
- transforms

Communication / SW architecture

- debugging Tools
- simulations
- bags

Developers tools

ROS *nodes*

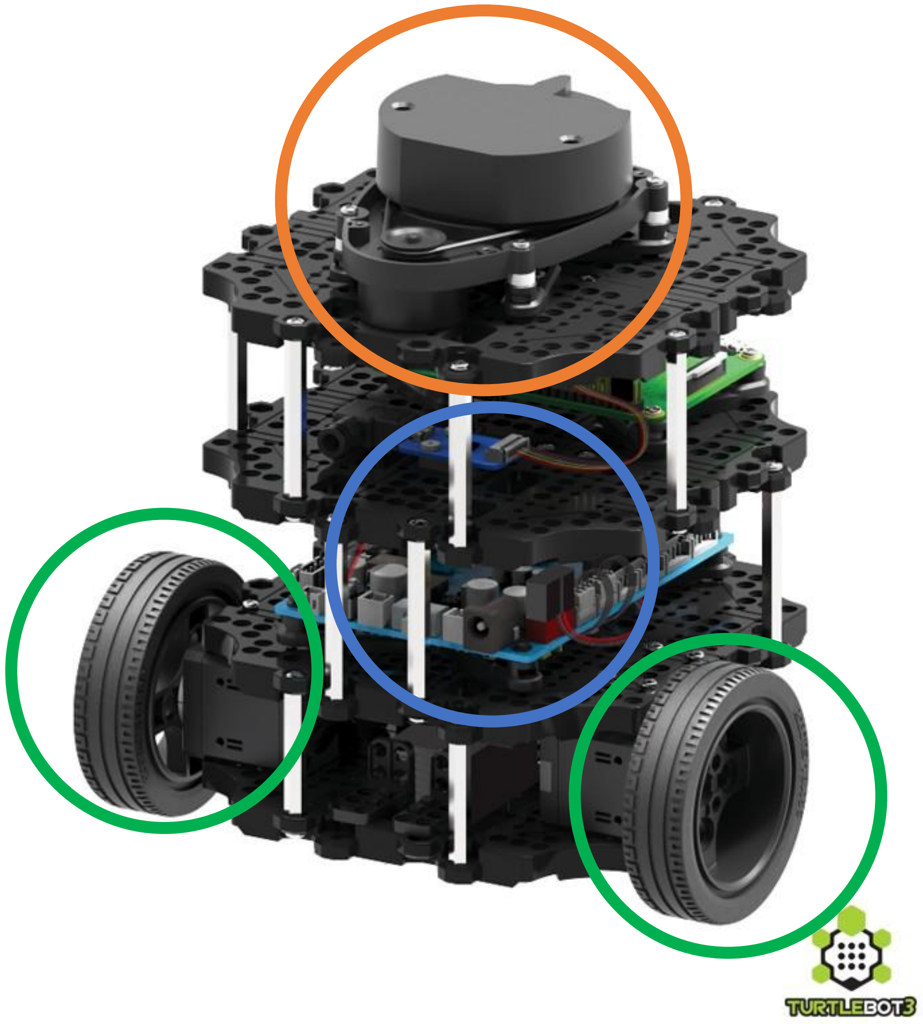


A *node* is a process that performs computation:

- 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

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.

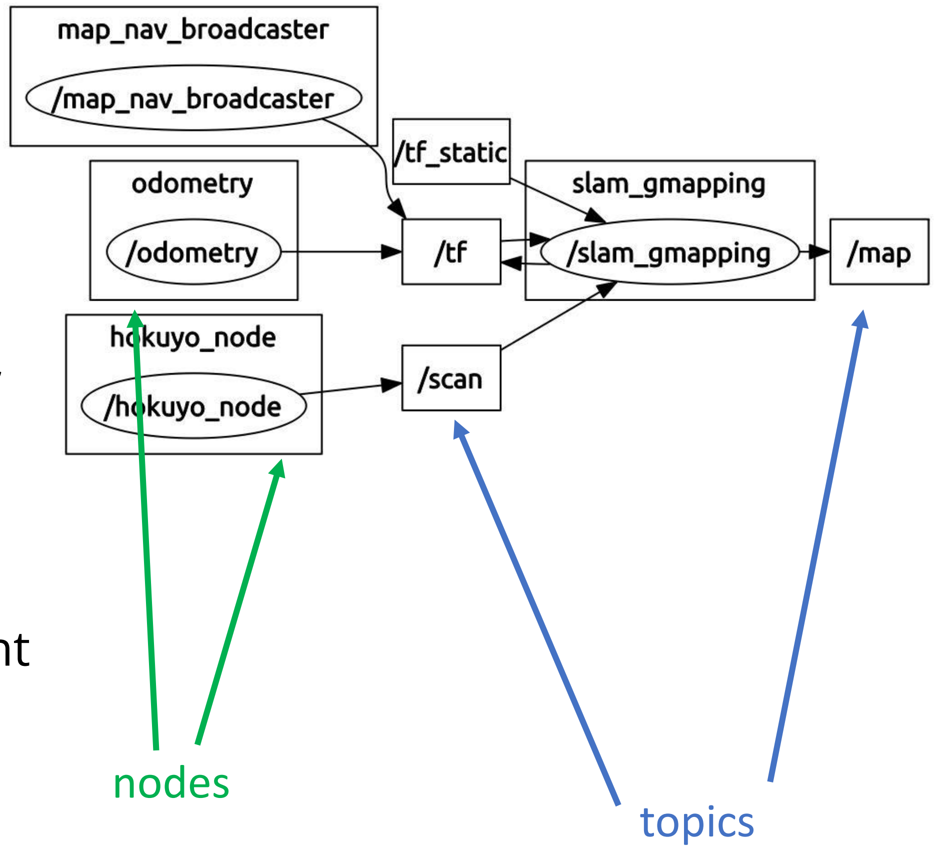
ROS

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

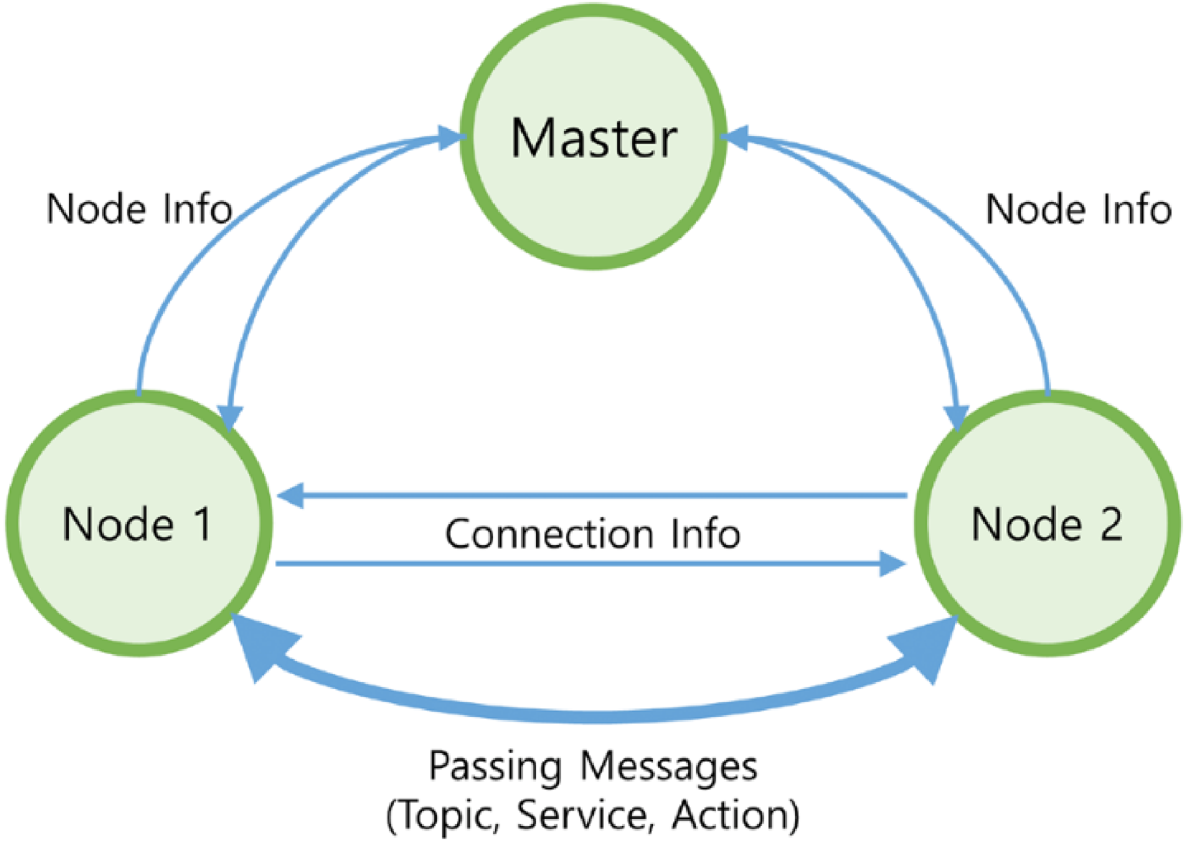
```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

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
```

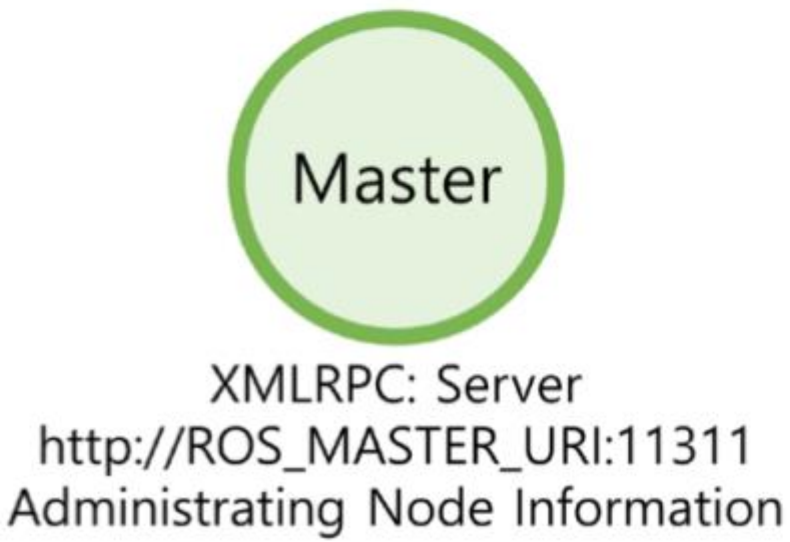
- each topic is strongly typed by the ROS message type used to publish to it
- nodes can only receive messages with a matching type.
- type consistency is not enforced among the publishers, but subscribers will not establish message transport unless the types match.
- all ROS clients check to make sure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

ROS master



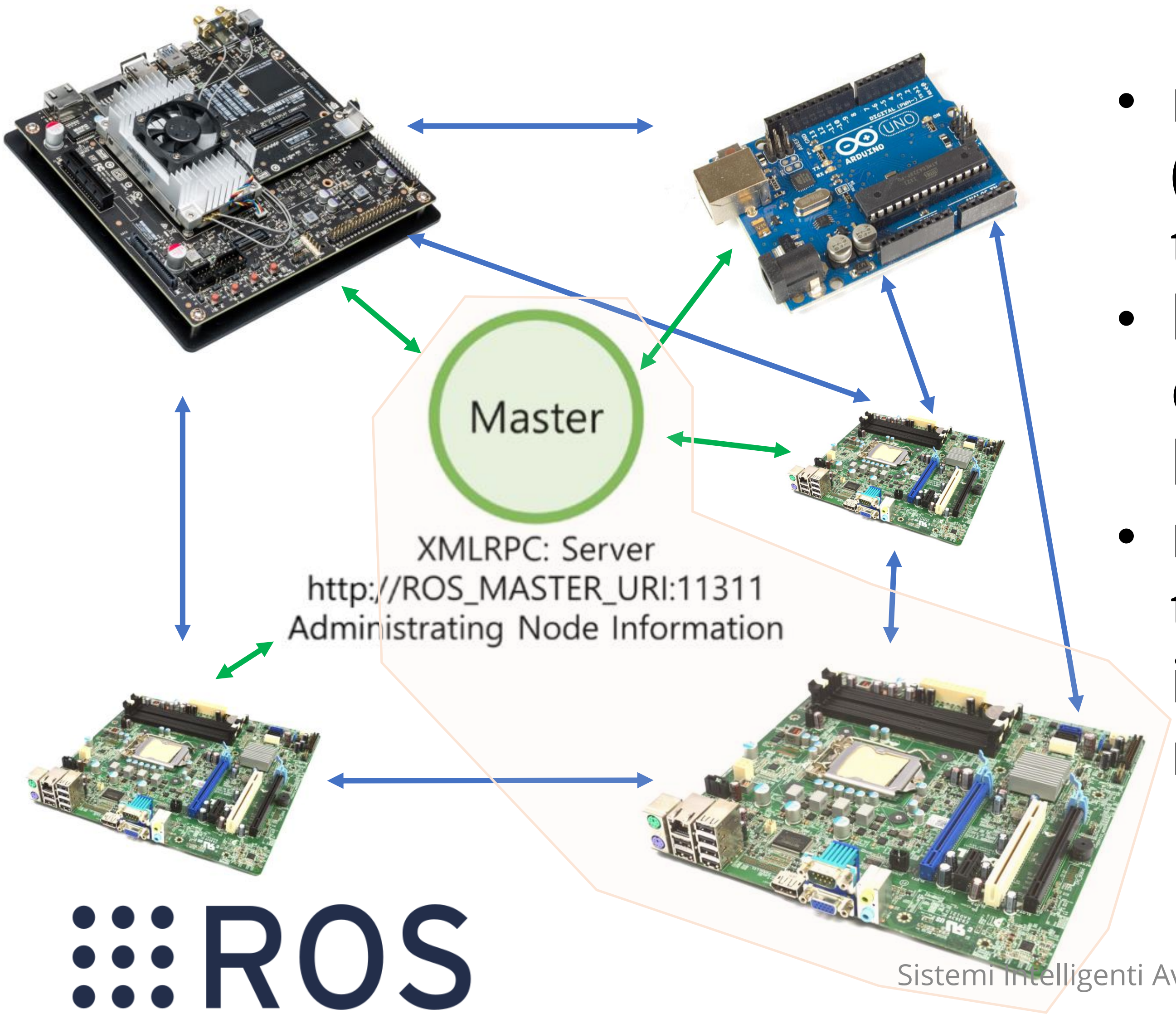
- the ROS Master provides naming and registration services to the rest of the nodes in the ROS system.
- it tracks publishers and subscribers to topics.
- it enables individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer.

ROS master and nodes



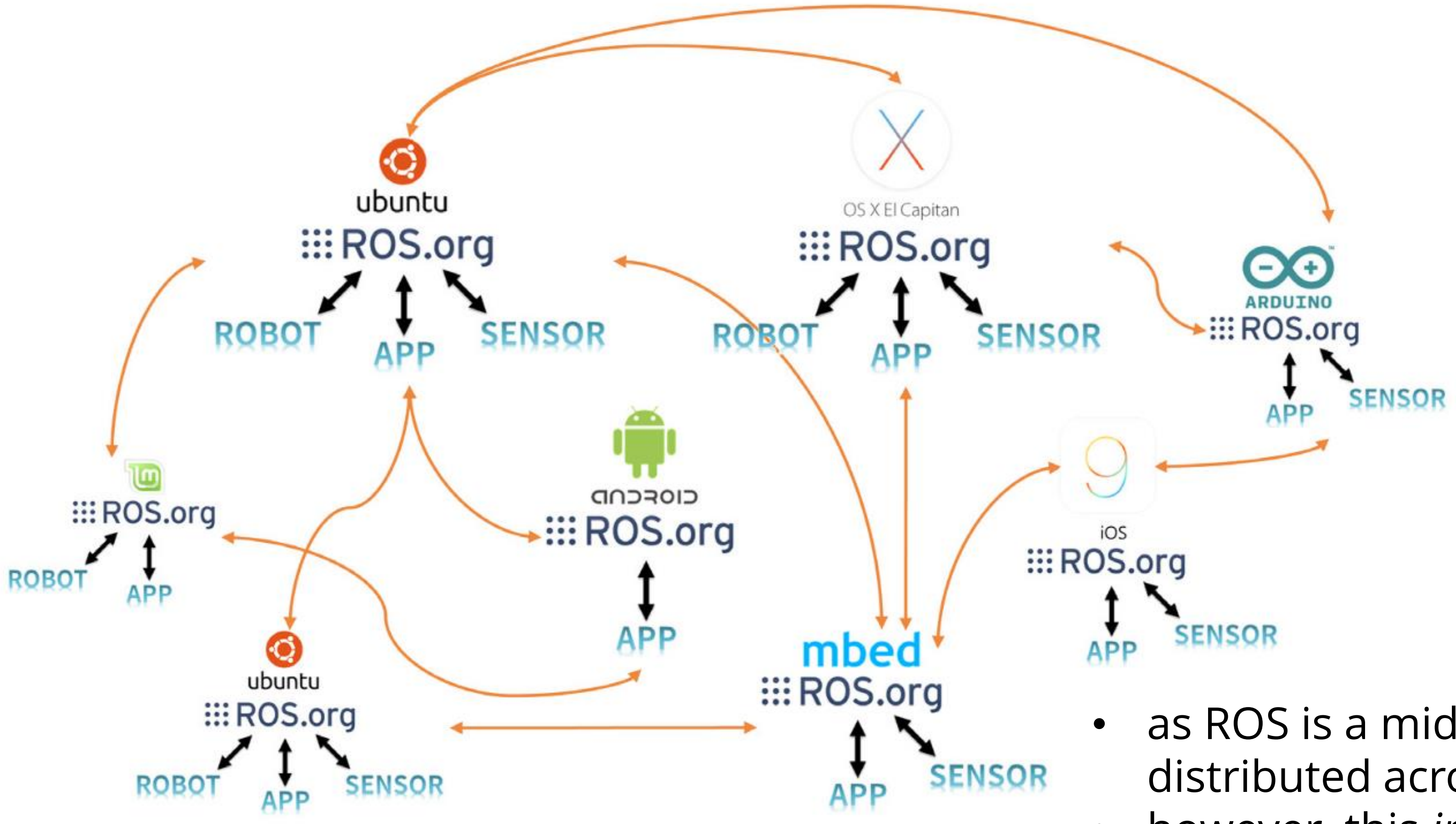
- the ROS master is a process and it is defined by its IP/port shared across all nodes
- acts as coordinator and manages the communication among nods
- this allows nodes to be distributed on different machines
  (in the same network)
- this mechanism allows to decouple the execution of a process from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

ROS master and nodes



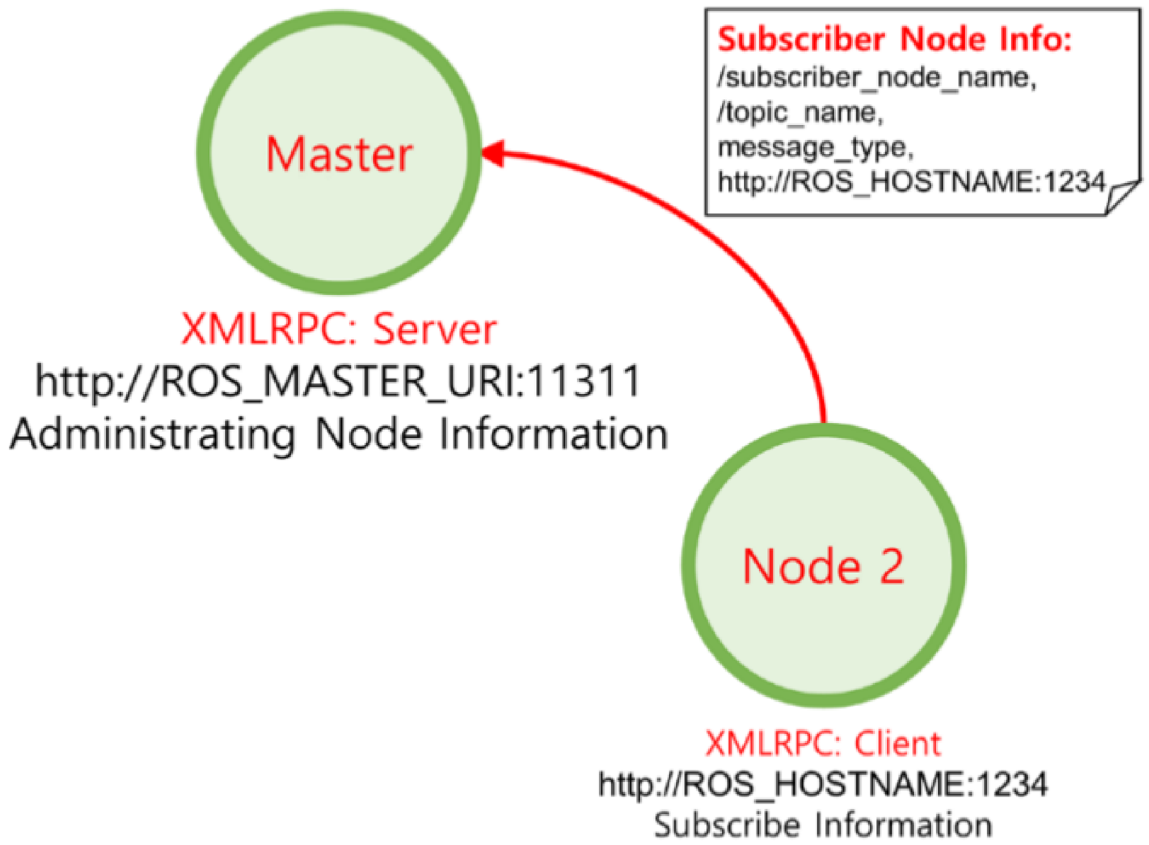
- robots may have to perform several (computationally intensive) tasks together
- hardware decoupling allows to distribute such tasks on dedicated hardware (e.g., Nvidia Jetson for GPUs)
- moreover, robots are <u>hardware</u> and this architecture allows to easily interface control boards for sensors,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

ROS on multiple platforms



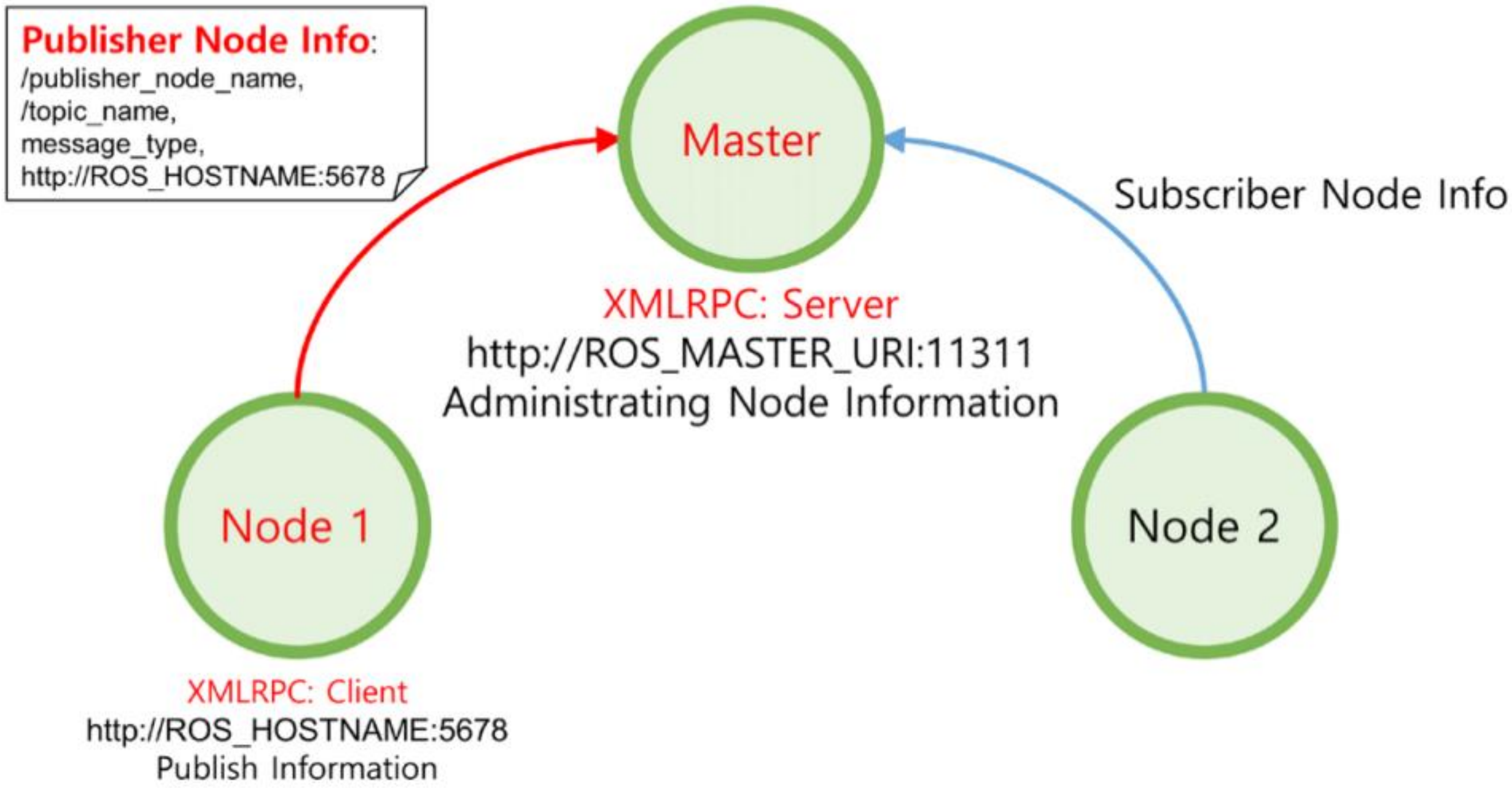
- as ROS is a middleware, computation can be distributed across different OS
- however, this *in practice* is far than ideal
- OS independence is de-facto provided for linux-based and embedded systems.
- rule-of-thumb: use Ubuntu for non-embedded systems not all versions either, but this will improve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

Set up a ROS topic publisher/subscriber



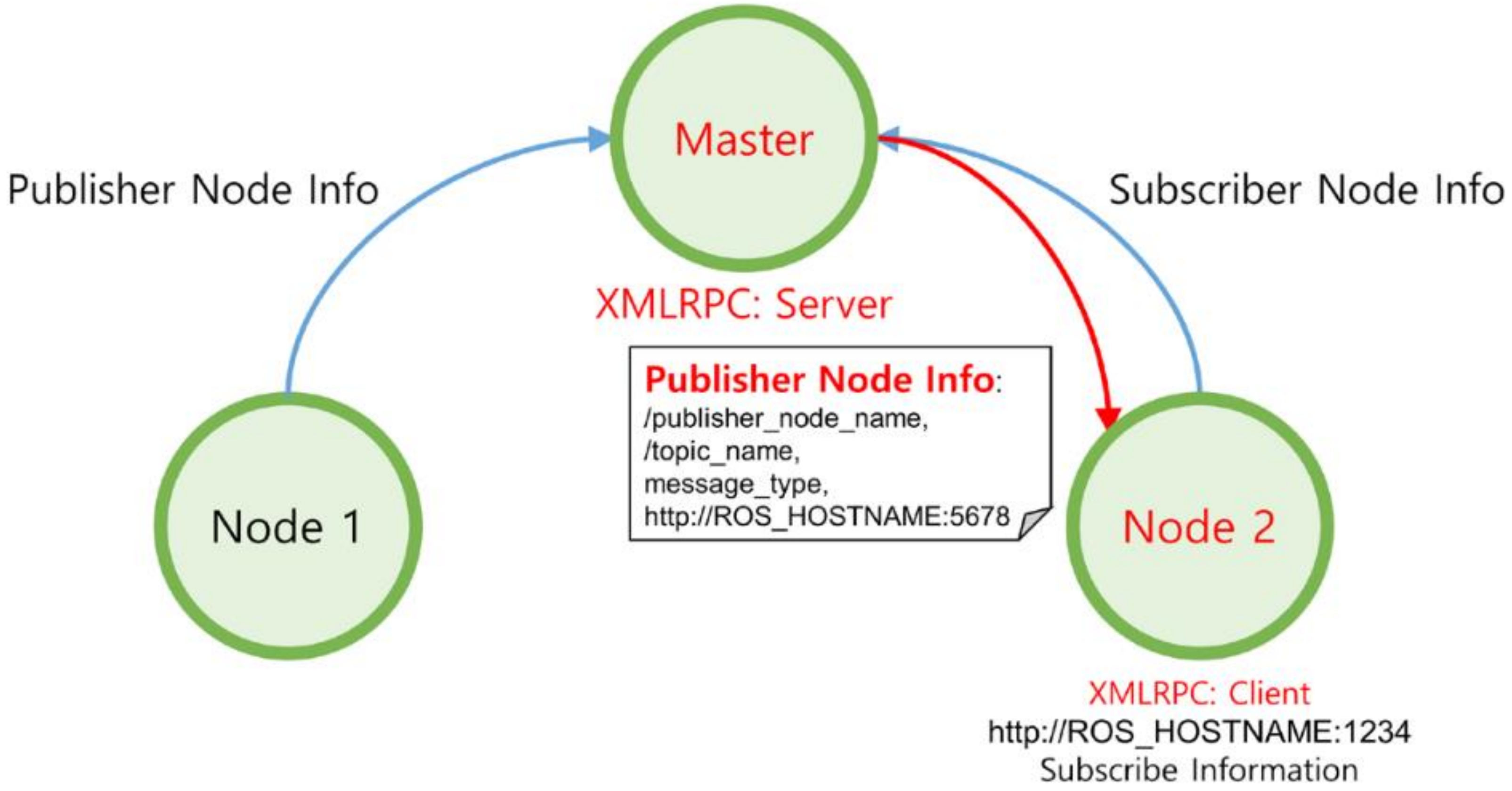
- a subscriber node registers to the ROS MASTER
- and announces its
  - Name
  - Topic name
  - Message Type
- communication is performed using XMLRPC

Set up a ROS topic publisher/subscriber



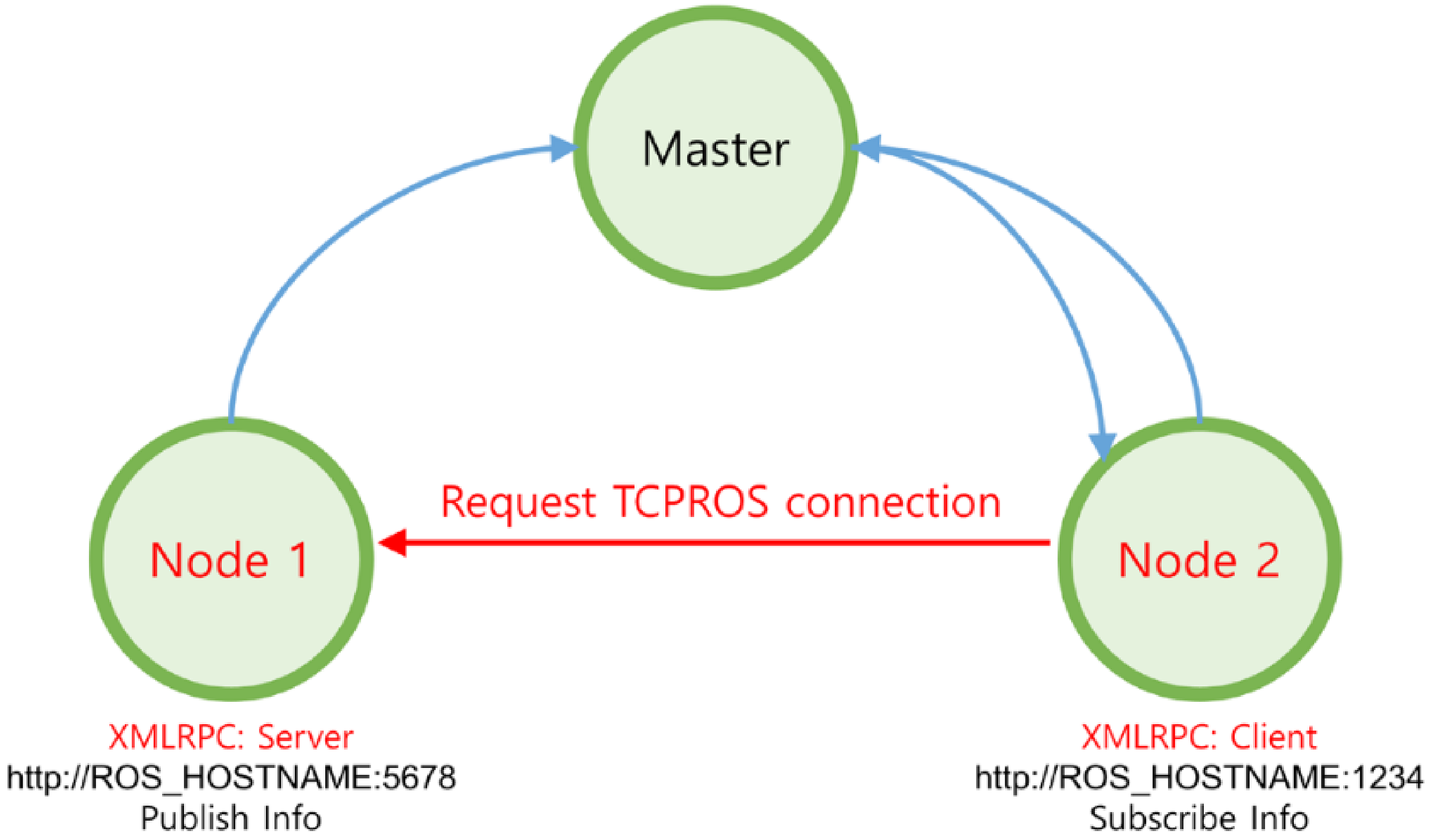
A publisher node now registers to the ROS MASTER

Set up a ROS topic publisher/subscriber



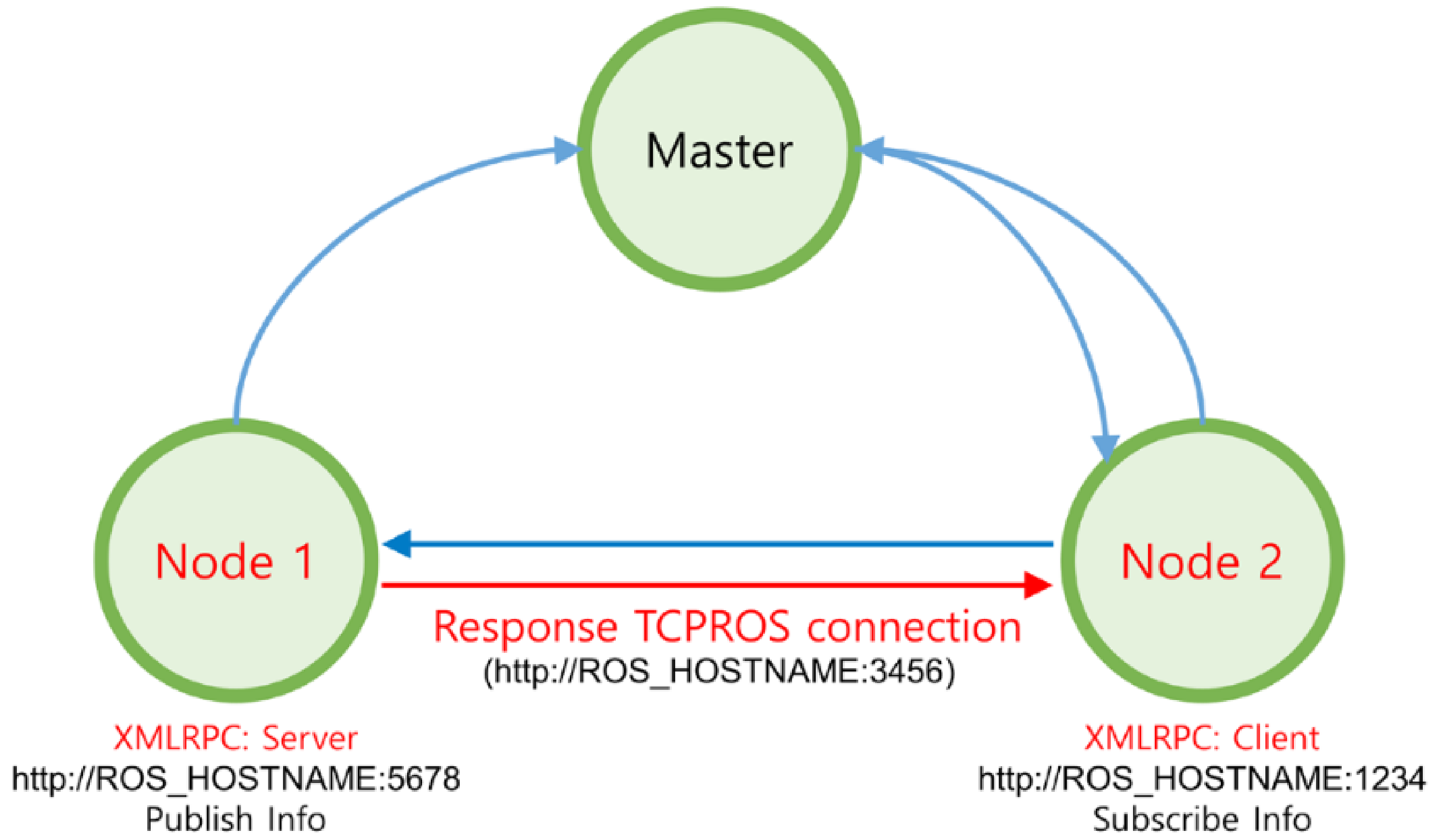
The ROS MASTER distributes info as all subscribers that want to connect to the topic and to the publisher node

Set up a ROS topic publisher/subscriber



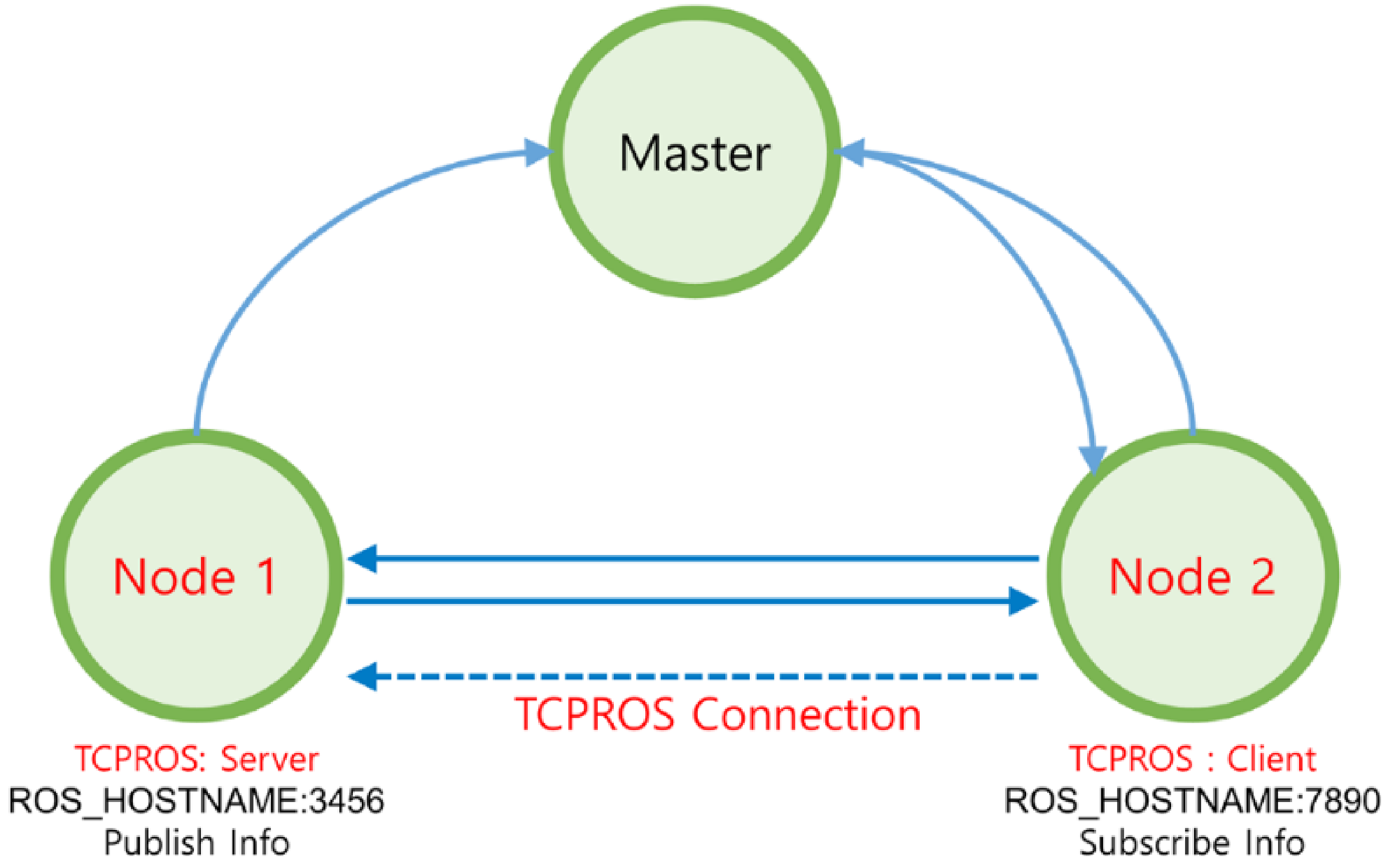
The subscriber node requests a direct connection to the published node and transmits its information to the publisher node

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

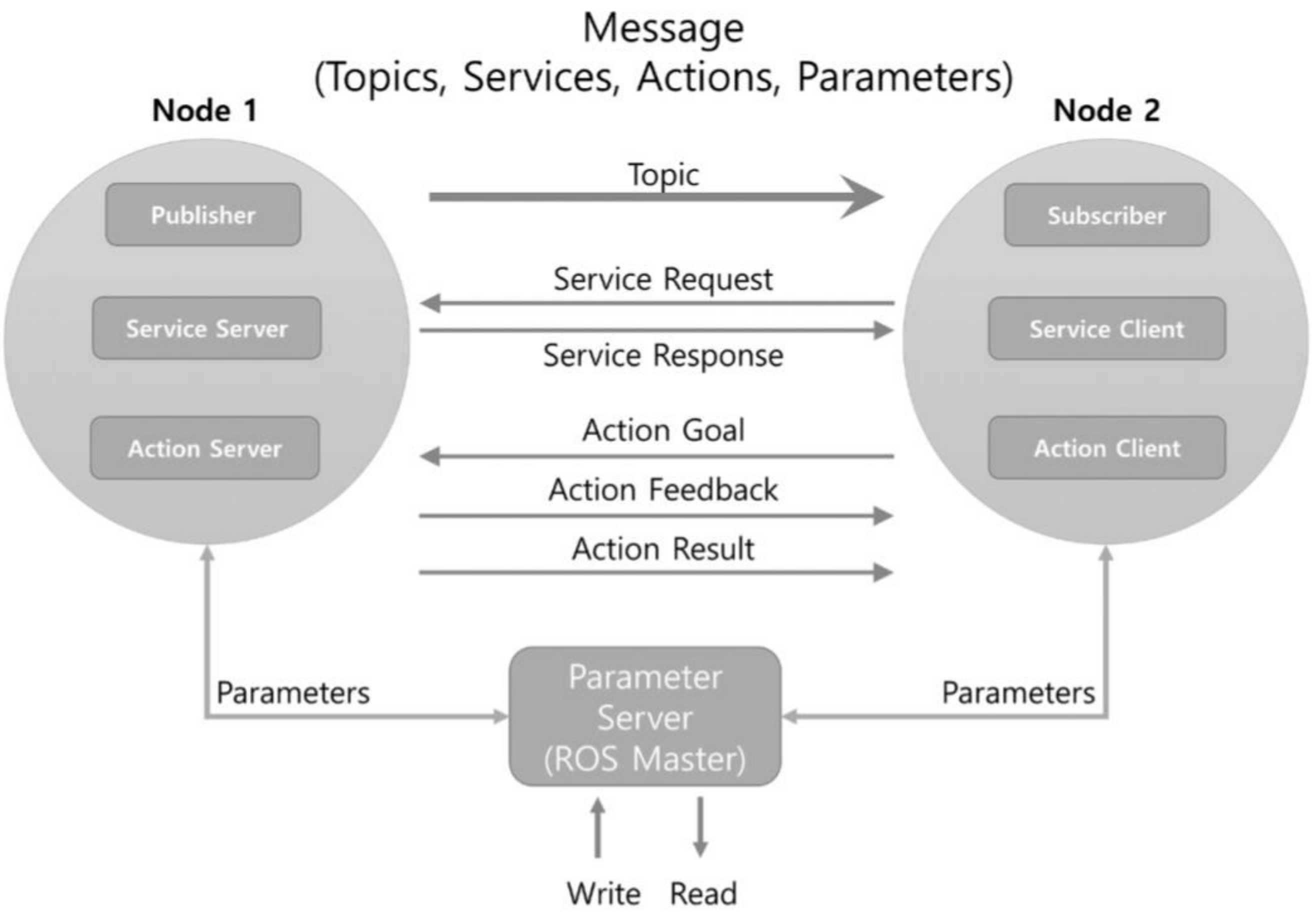


At this point a direct connection between publisher and subscriber node is established using TCPROS (TCP/IP based protocol)

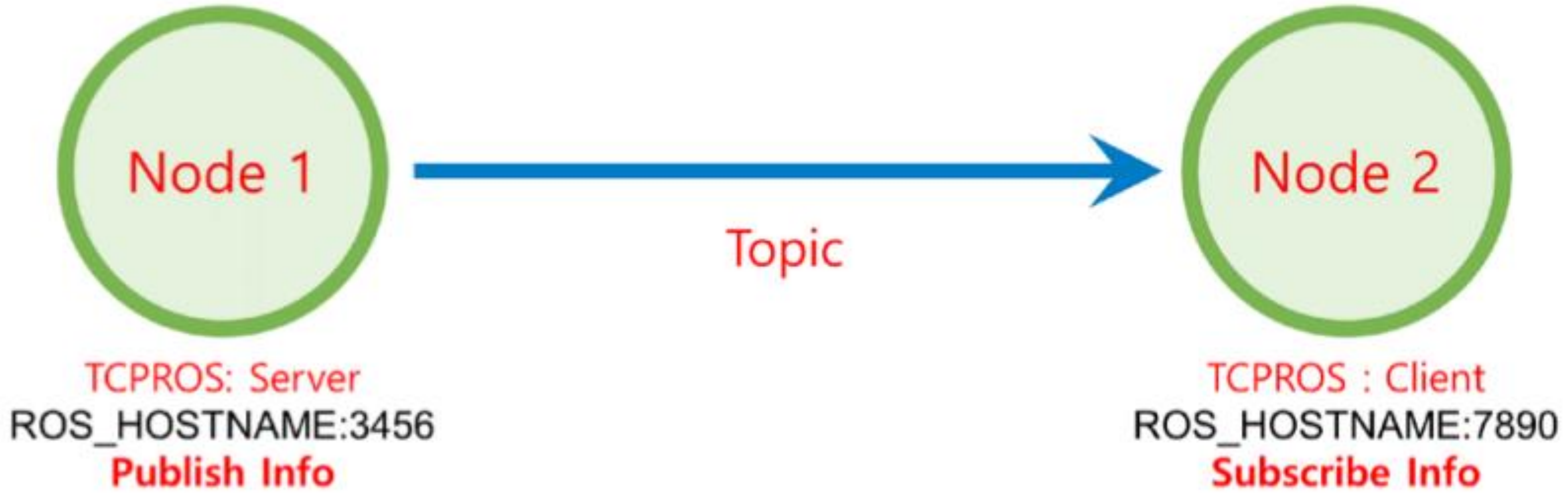
Communication among nodes

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

- Topics
- Services
- Actions

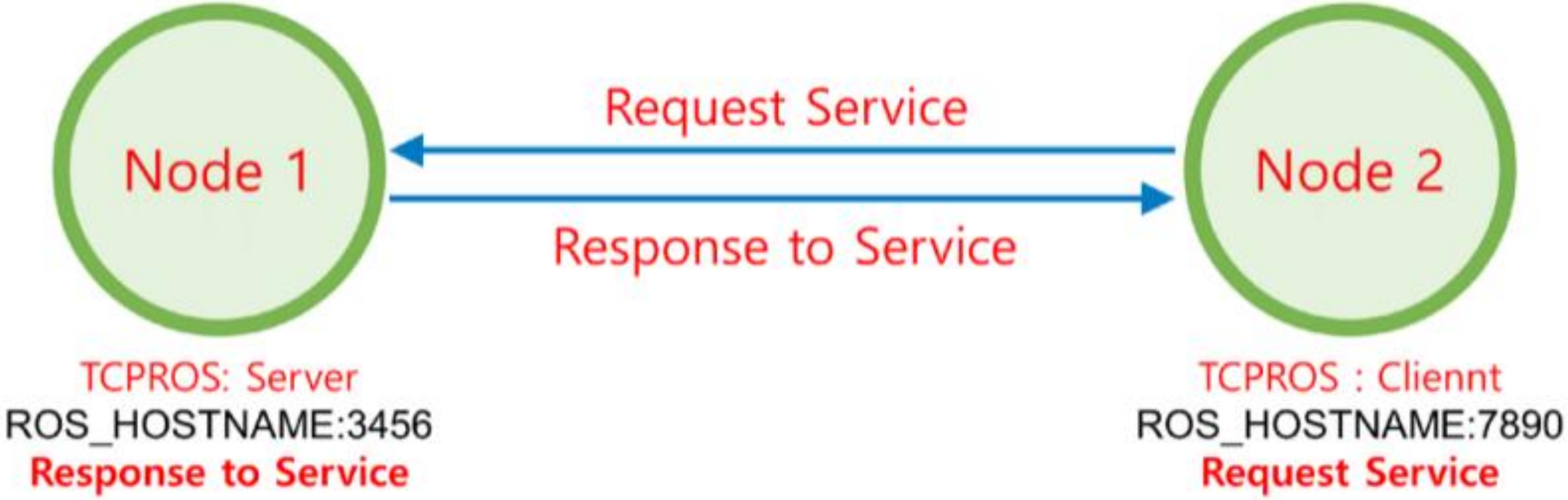


Communication among nodes



- 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)

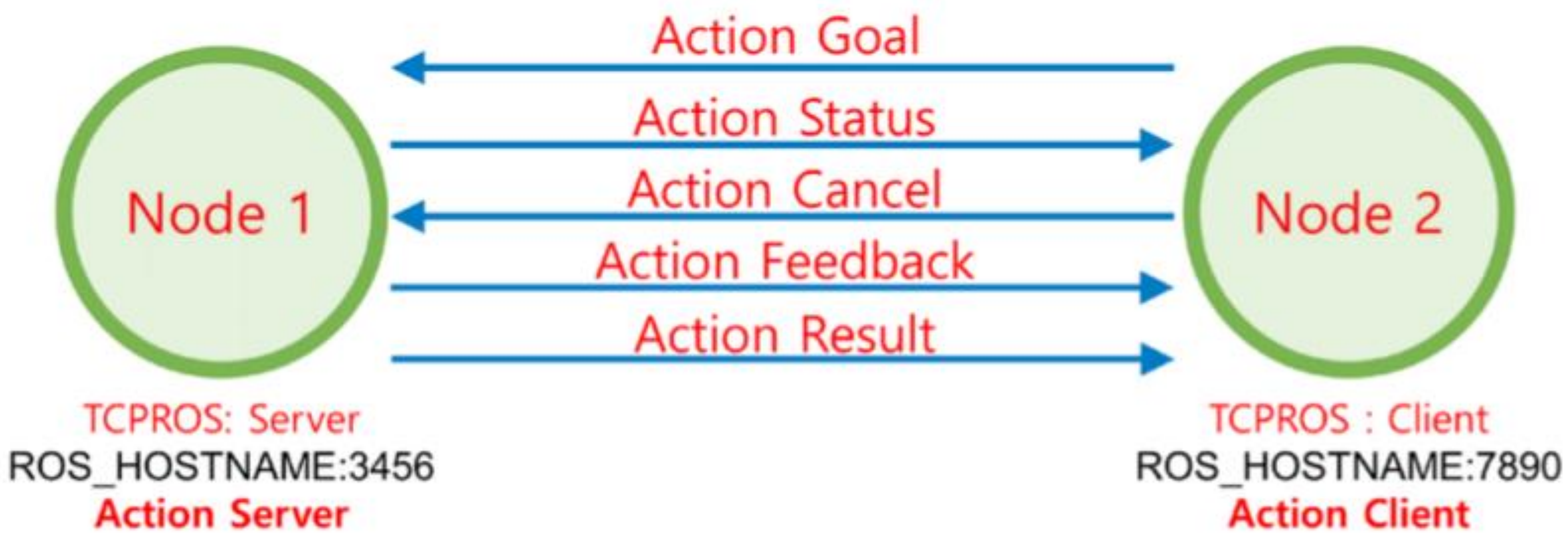
ROS *Services*



- 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*



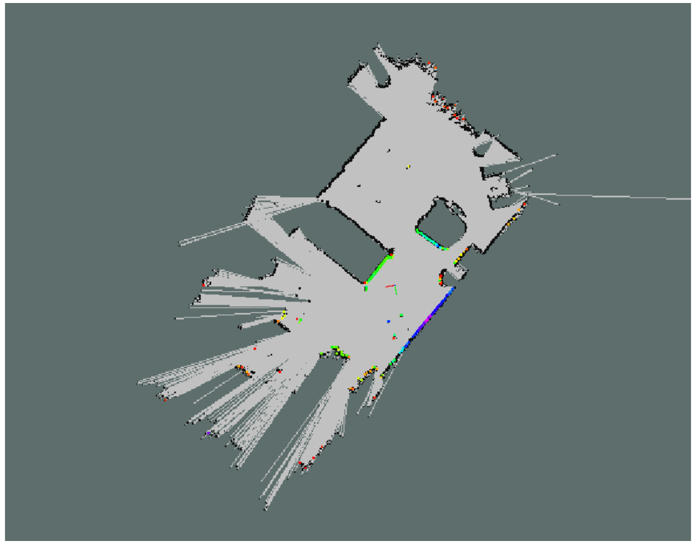
If the service takes a long time to execute, the user might want the ability to cancel the request during execution or get periodic feedback about how the request is progressing.
Action Services are for these tasks.

- 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.

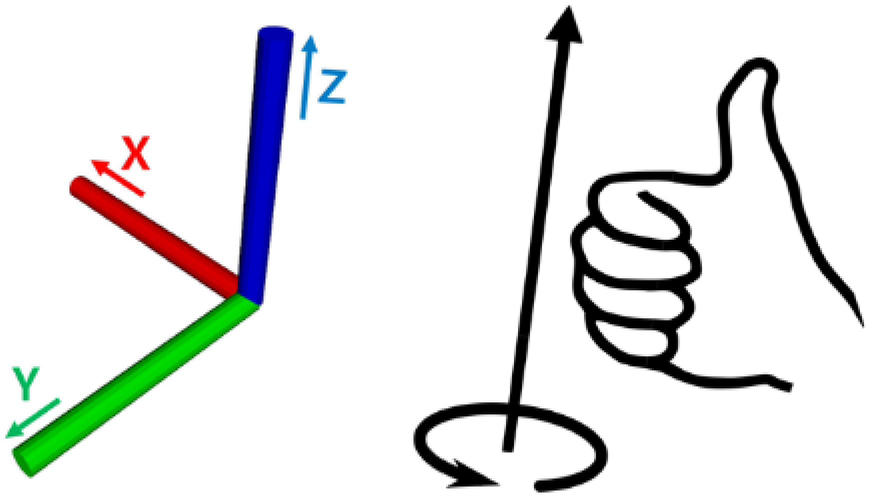
- nodes use this server to store and retrieve parameters at runtime.
- used for static, non-binary data such as configuration parameters.
- globally viewable so that tools can easily inspect the configuration state of the system and modify if necessary.



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

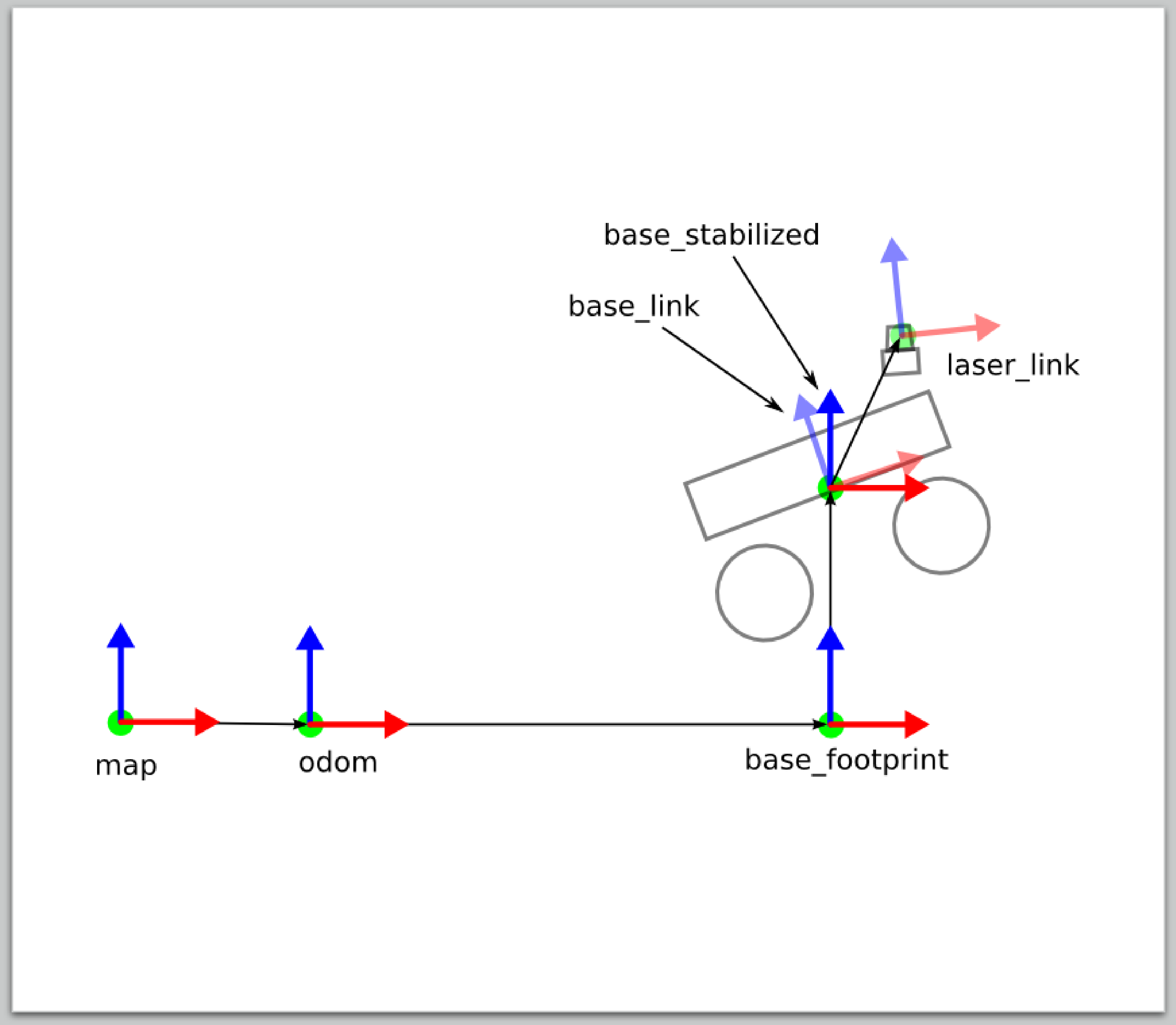
ROS *Transforms*

- 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



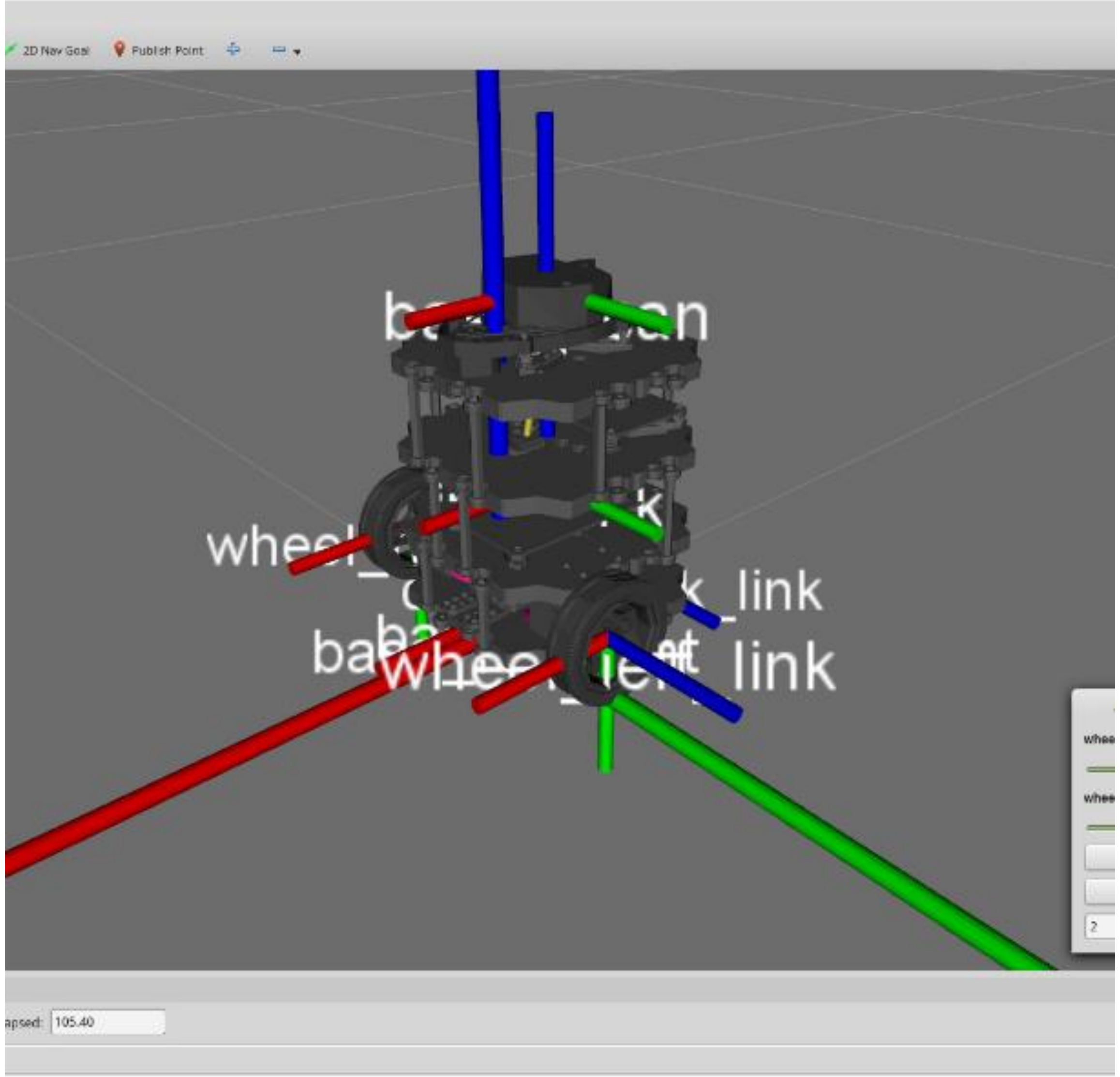
ROS *Transforms*

- 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



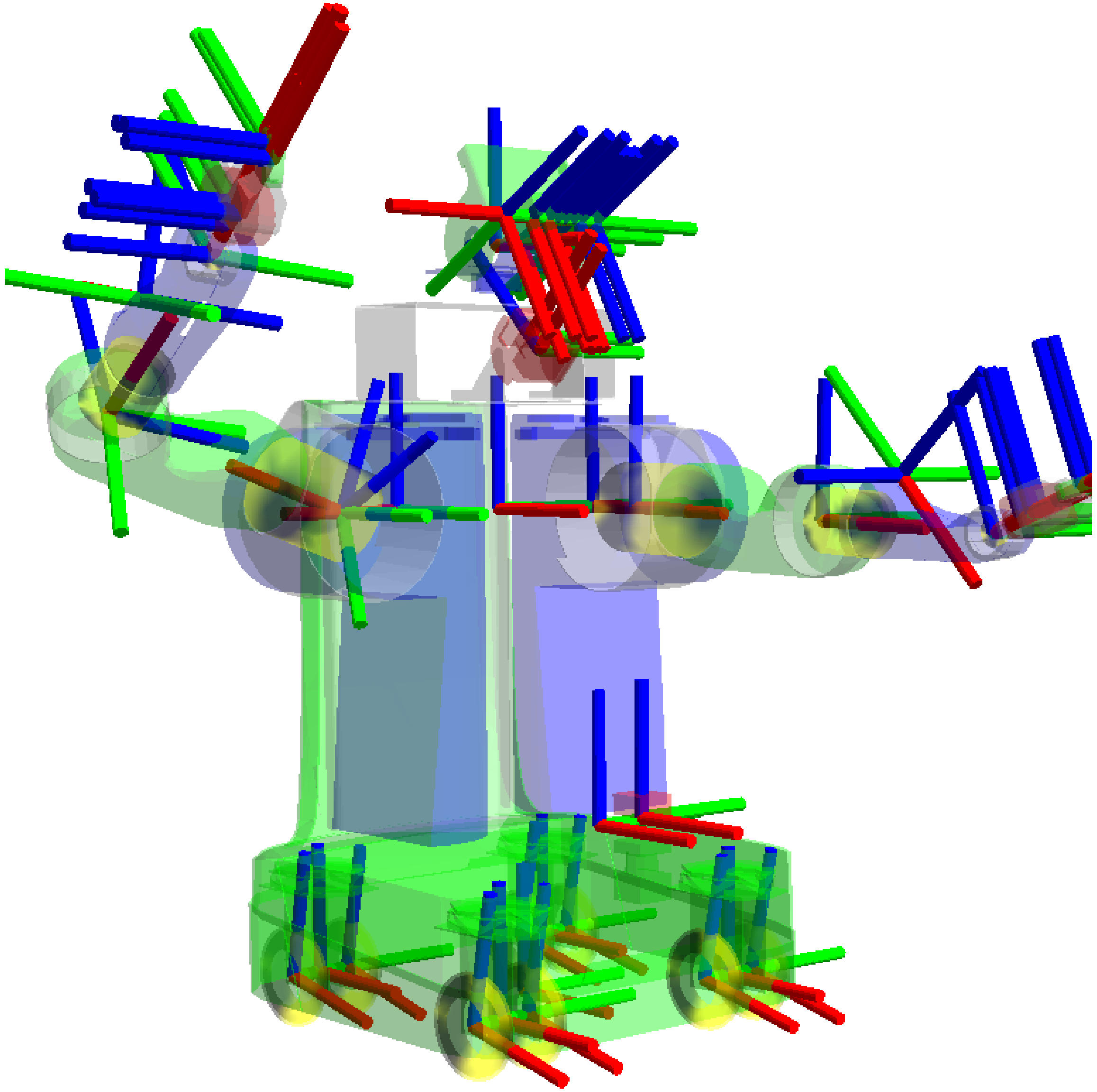
ROS *Transforms*

- 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 *Transforms*

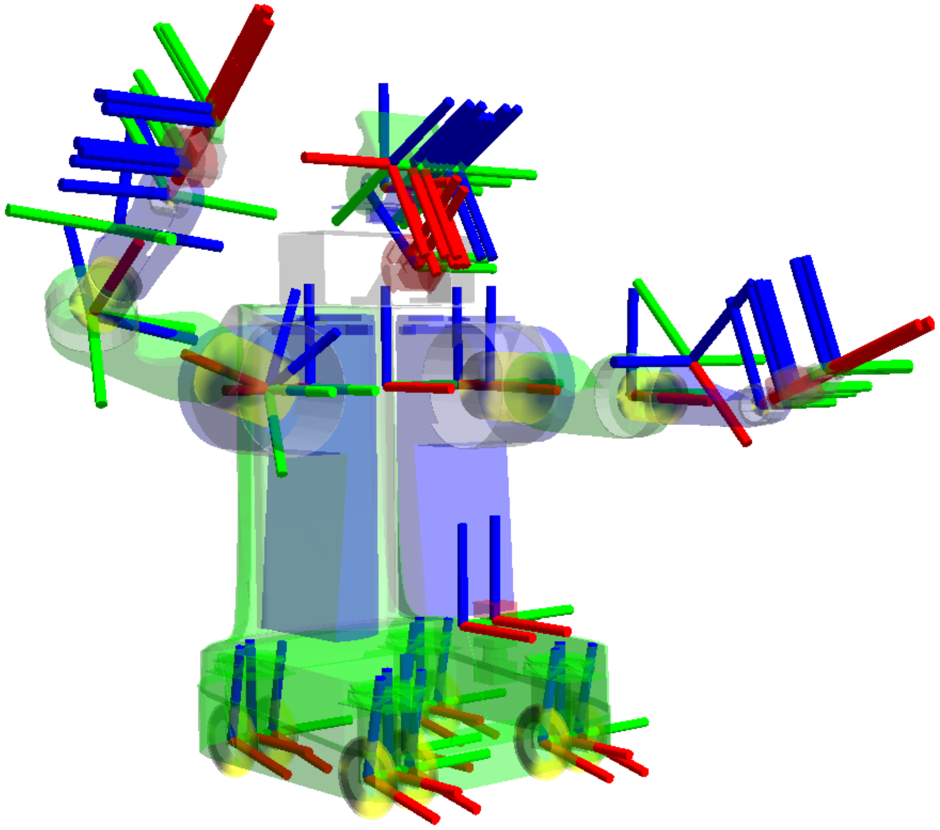
- TF can become complex, especially for robot with a lot of Degrees Of Freedom (DOF) as grippers
- ROS provides visualization tools for controlling such aspects



Developing toos

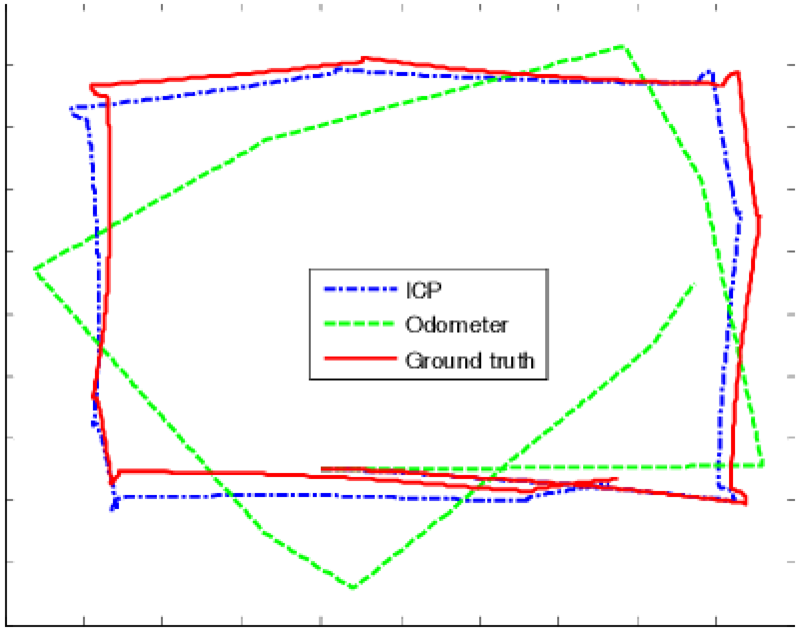
Developing a robot in ROS

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



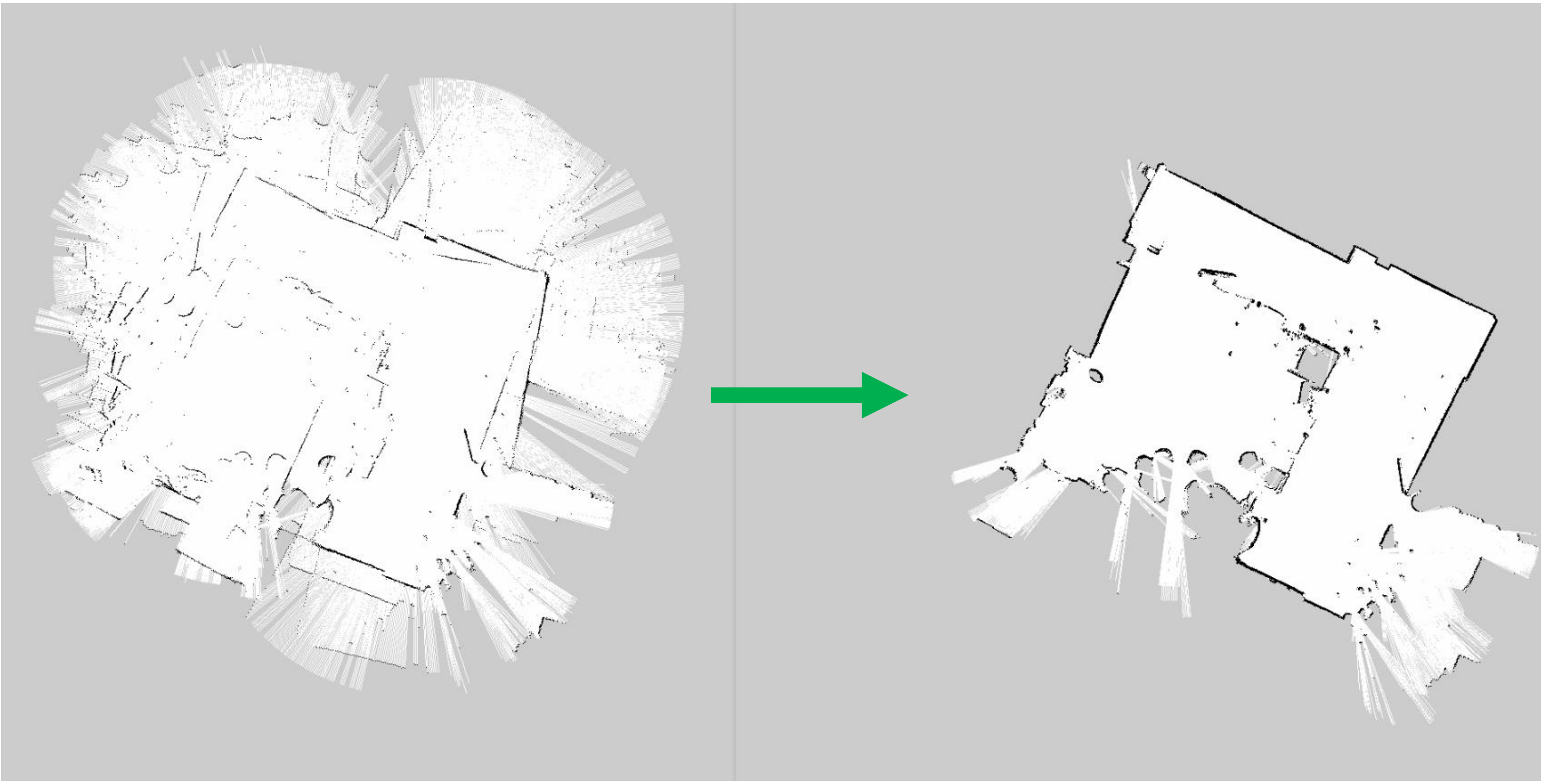
Environmental inaccuracies

- 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



Reducing environmental inaccuracies

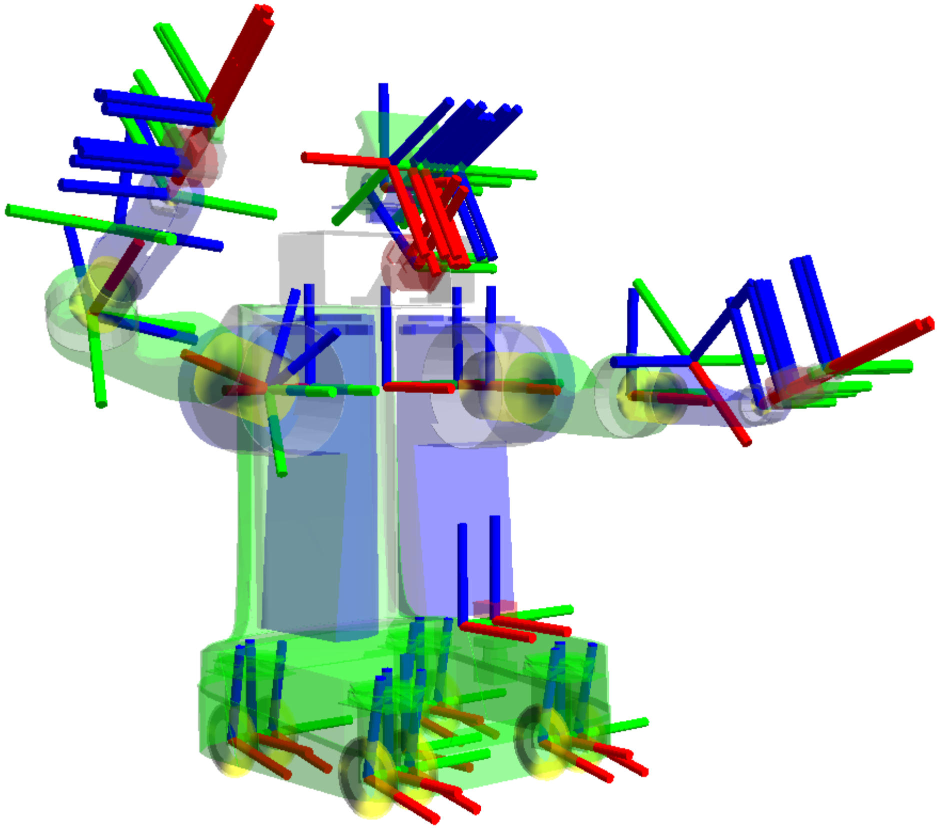
Even if assuming that there are no unexpected failures in the robot modules, some of the robot modules are designed to cope and reduces known sources of uncertainty and to integrate data together

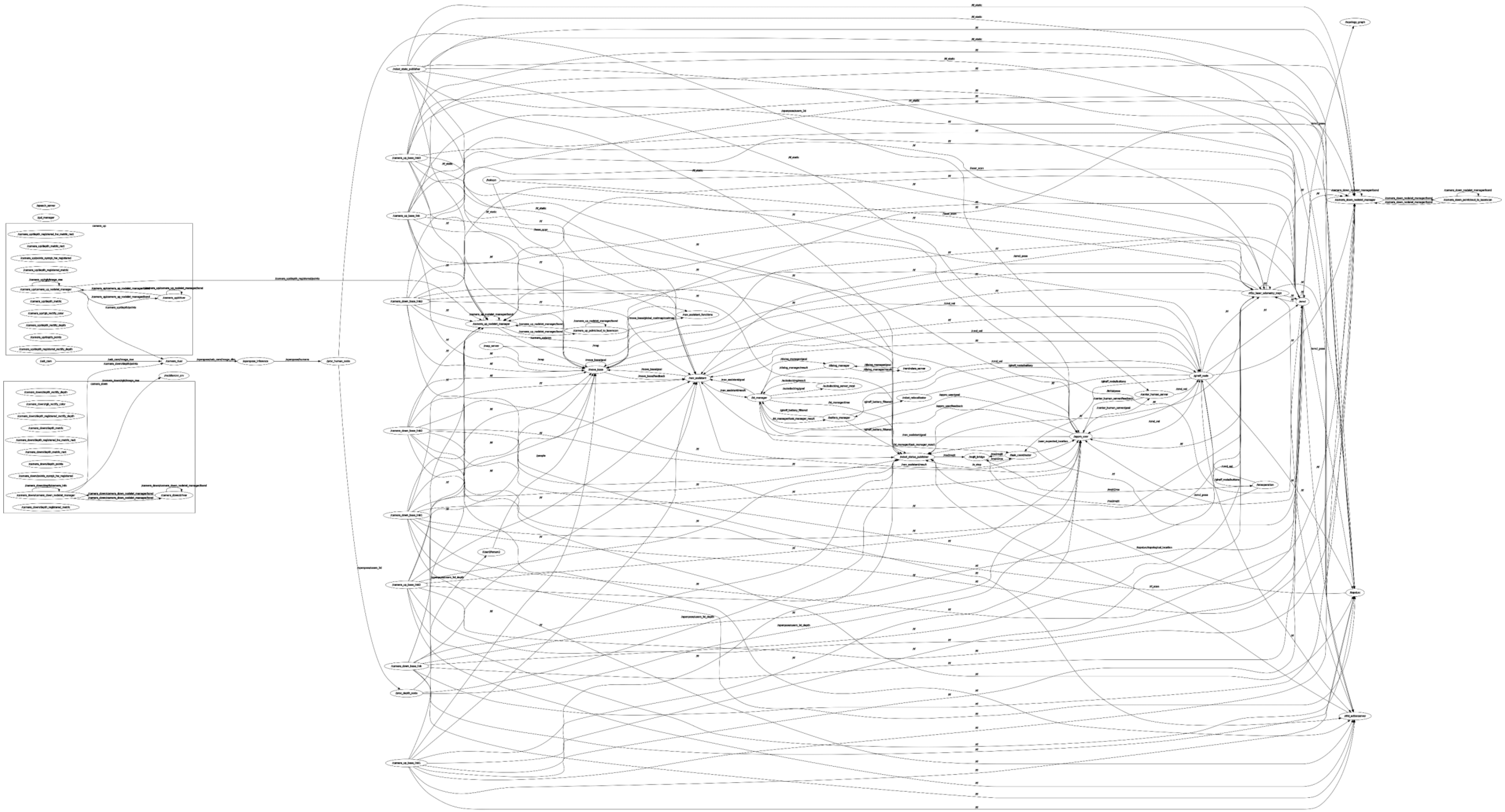


Mapping integrates sensor readings (e.g., laser range scanner) together reducing odometry error thus obtaining a valid map of the environment

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





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

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

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