The F1/10 platform

Real-Time Embedded System - The F1tenth autonomous racing

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

High Performance Real Time Lab

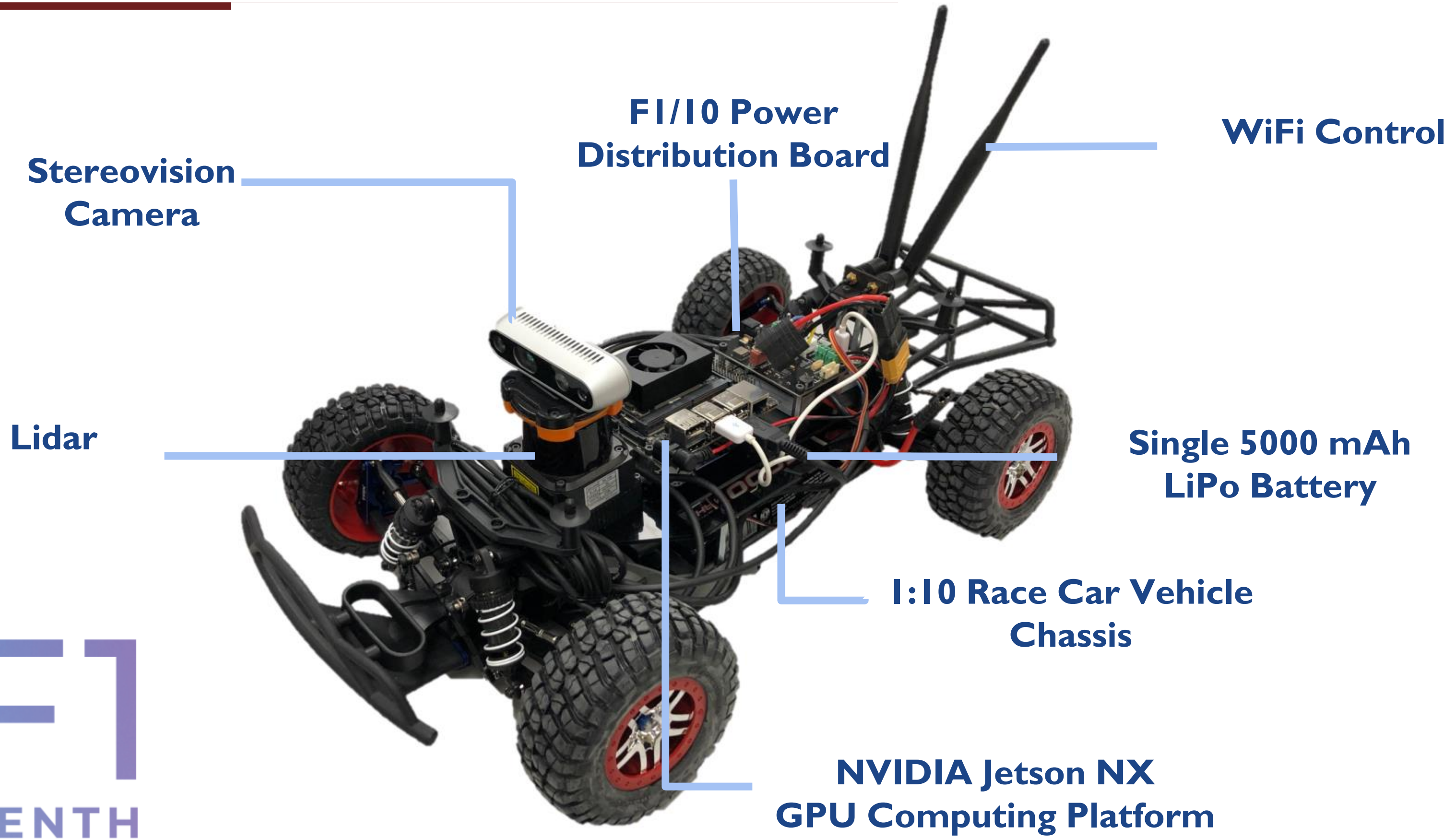
Course outline

- Intro course + basics of AD
- Hardware platform
- ROS2: Installation and profiling
  - Ex: ROS2 to HiL, open a bag
- Navigation: FTG, FTW, Pure pursuit
  - EX: navigation HiL
- Perception: scan matching, PF, LIO?
  - Ex: perception (PF with PThreads)
- Build the car

I do <u>not</u> cover all aspects of AD!!!

- Systems and control theory => Prof. Falcone
- Platforms and algorithms for autonomous systems => Prof. Sanudo & Prof. Falcone
- High-Performance Computing => Prof. Marongiu (FIM)
- Machine Learning => Cucchiara's

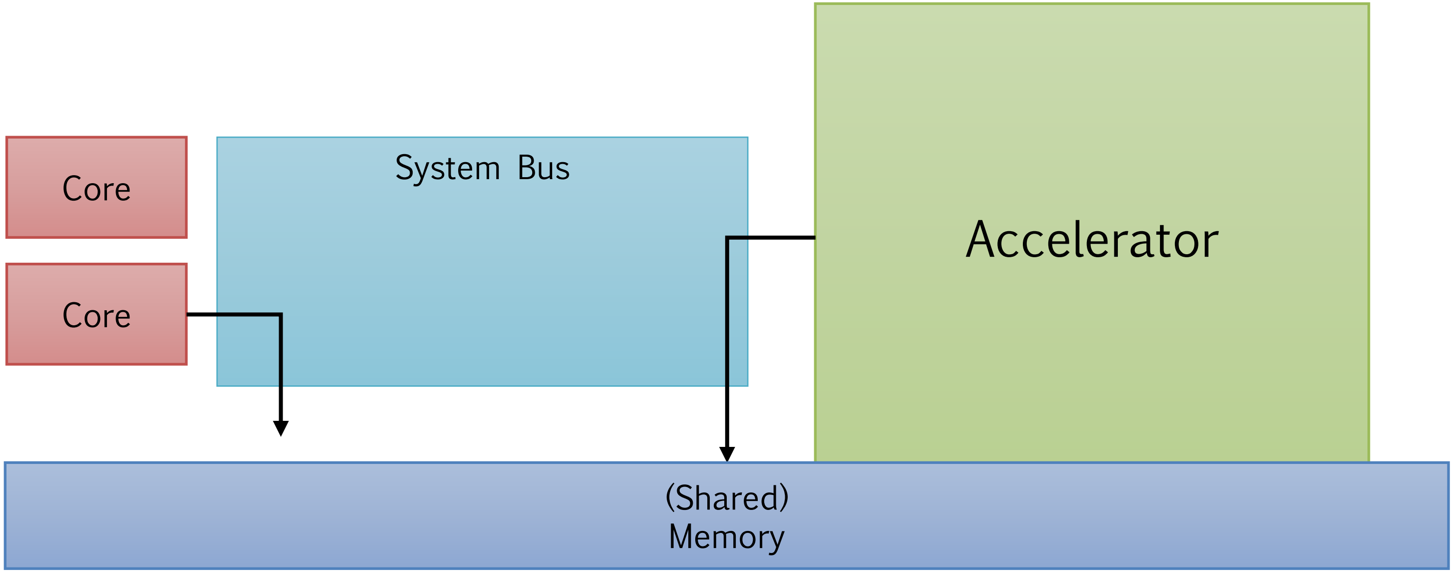
The F1/10 platform



F1 TENTH

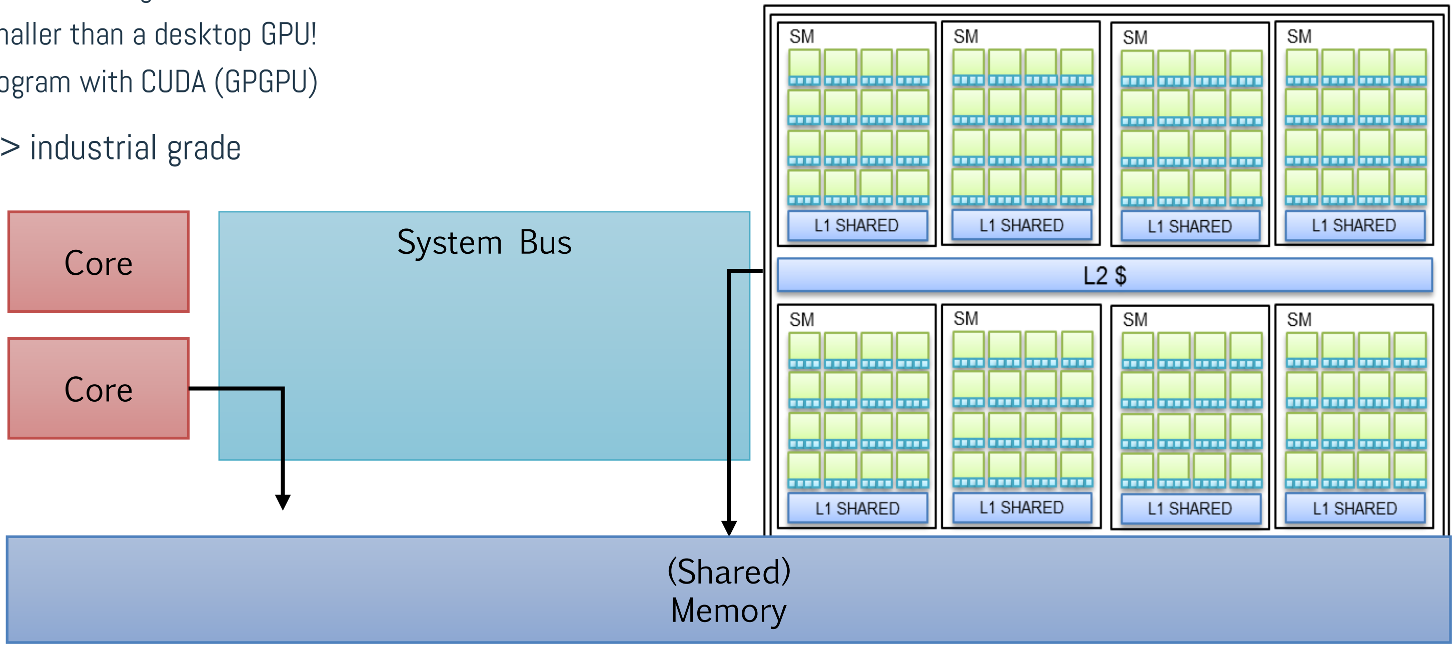
On-board computer - template

- Aka: Domain controller, aka: ECU, aka: …
- Embedded heterogeneous platform
  - Multi-core + data cruncher accelerator
  - Shall employ safety core (Aurix?) to enable automotive ISO26262/ASIL-D compliancy



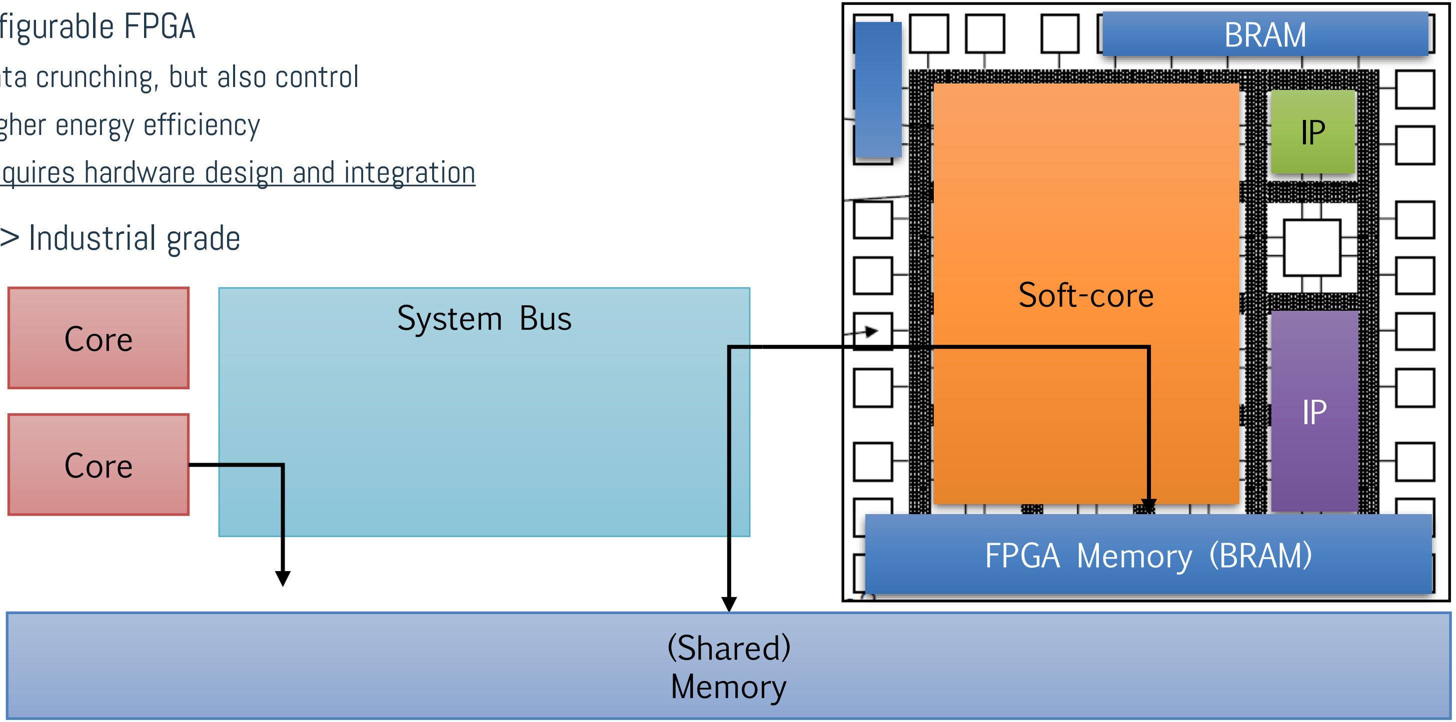
NVIDIA Jetson Orin/NX/Nano

- 6/8 core ARM host
  - For control-based workload
- Embedded GPU
  - Data crunching
  - Smaller than a desktop GPU!
  - Program with CUDA (GPGPU)
- Orin => industrial grade



Xilinx UC+ / Kria

- 6 core ARM host
  - For control-based workload
- Reconfigurable FPGA
  - Data crunching, but also control
  - Higher energy efficiency
  - Requires hardware design and integration
- Kria => Industrial grade



Our F1/10 cars

With GPGPUs

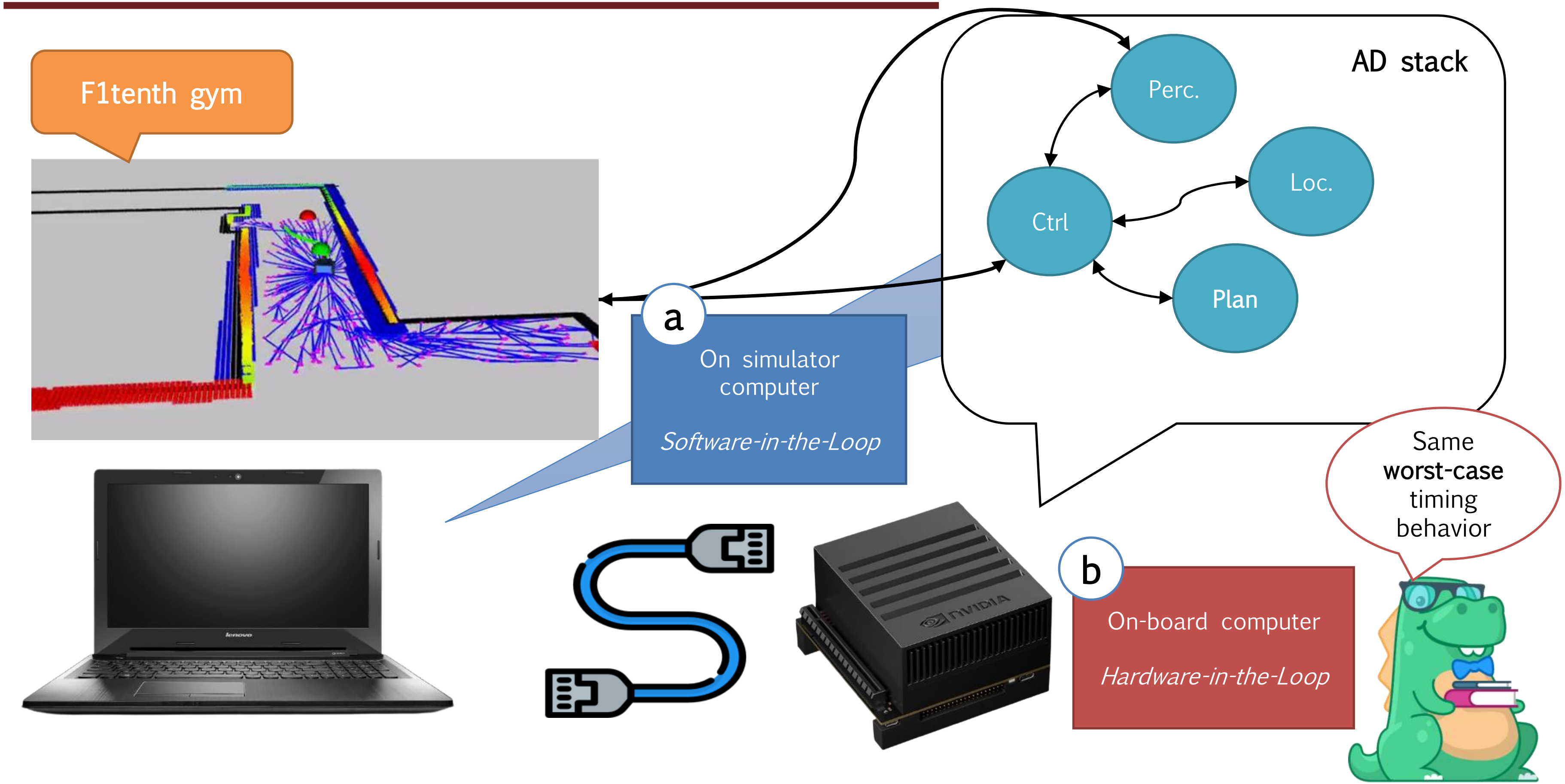
- **Bud** => NVIDIA Orin
- **Kitt** => NVIDIA Xavier NX

With FPGA

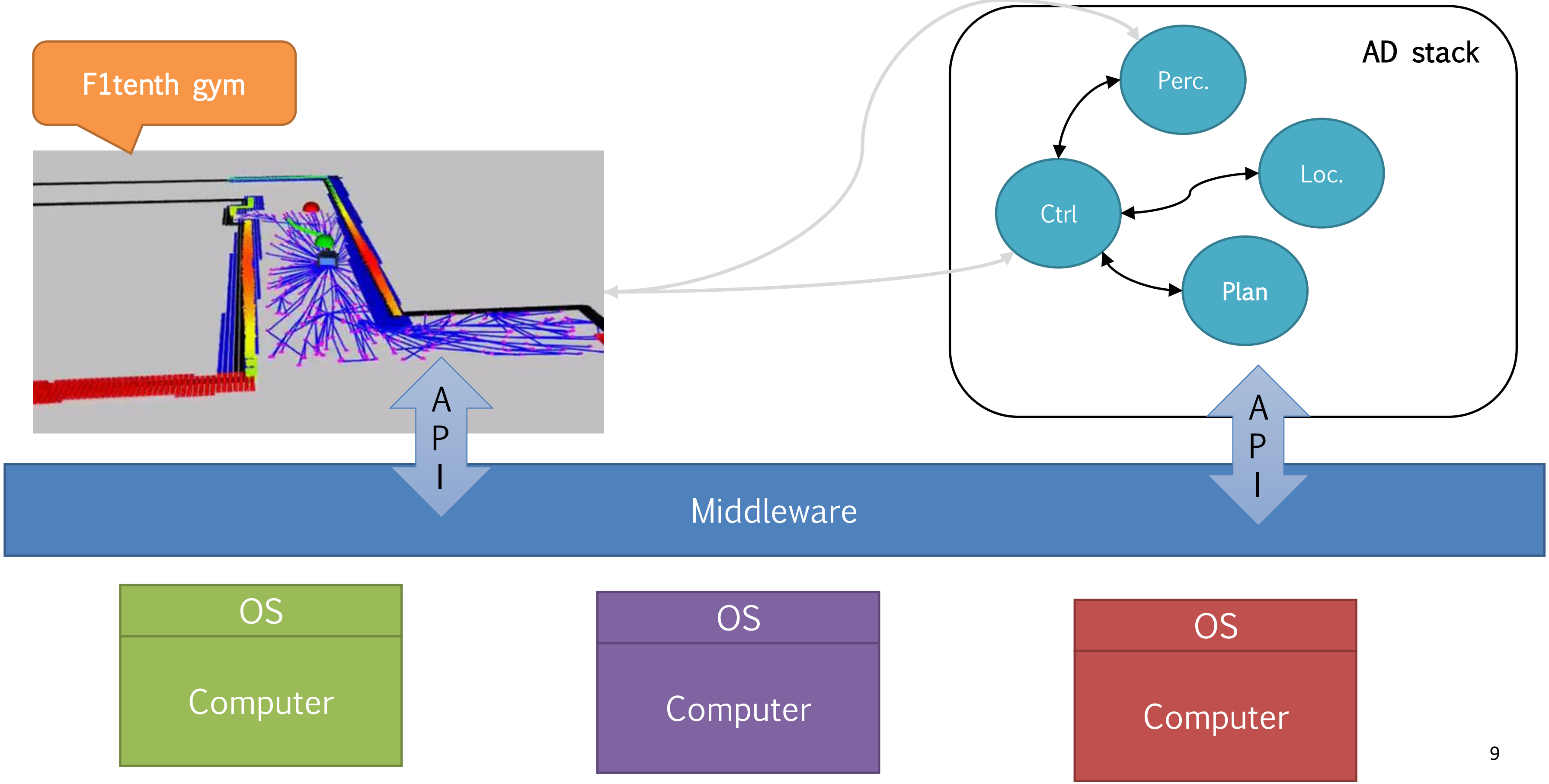
- **Thundershot** F1/10 => Xilinx Kria

- **Frankenstein**/Frankie
  - Whatever we want to test
  - Currently, cybersecurity

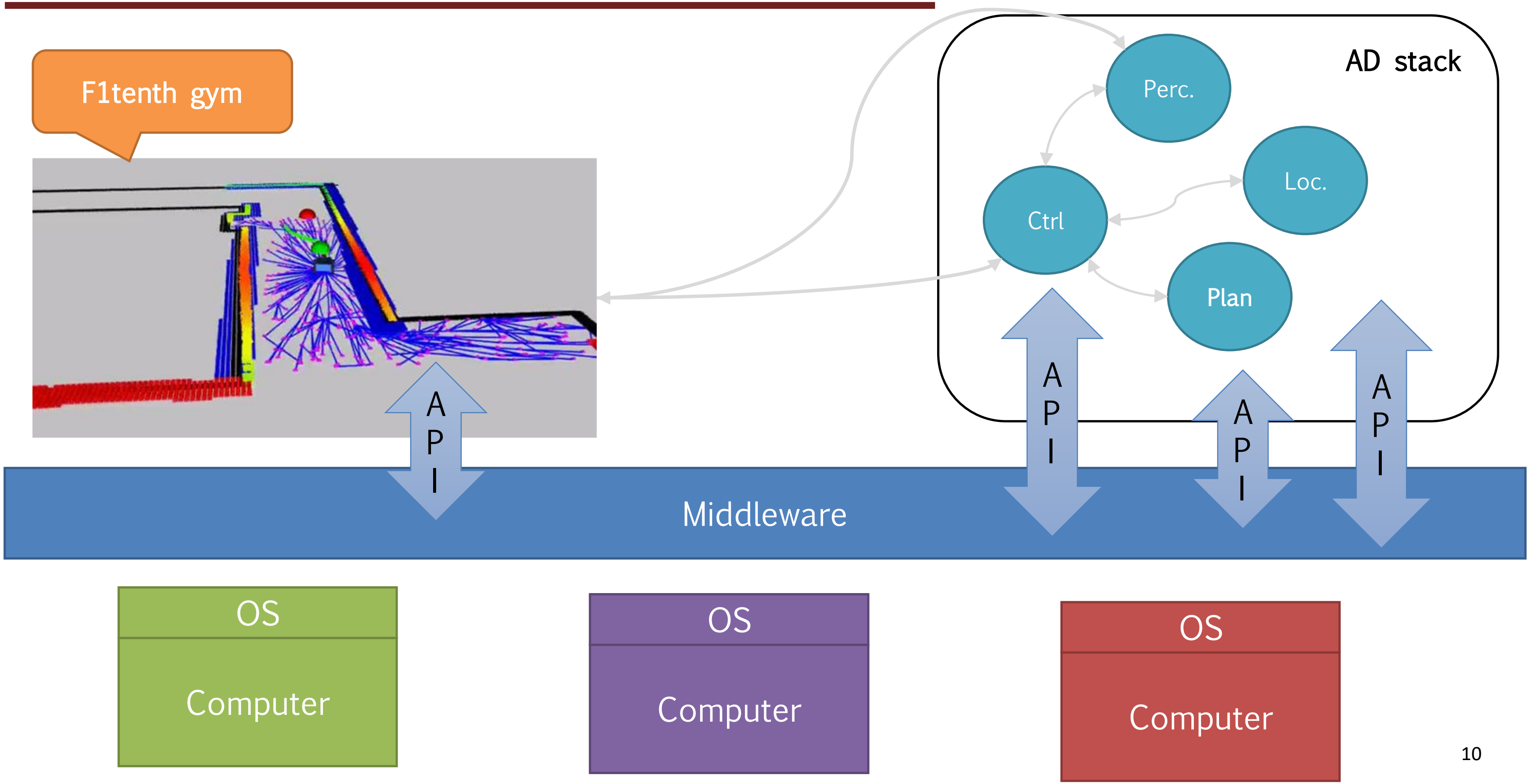
Software ecosystem – Sil vs. HiL



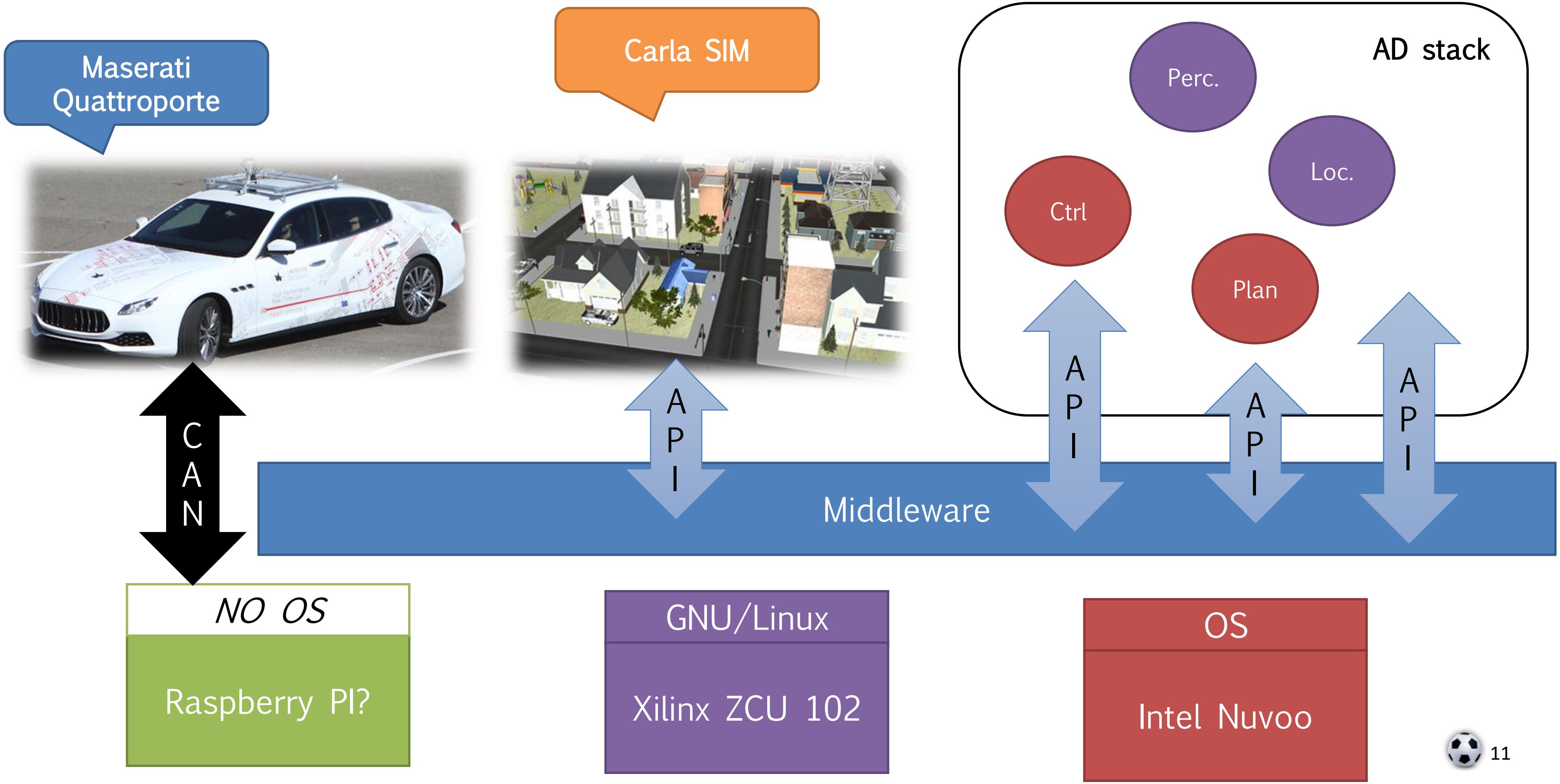
Communication middleware



Communication middleware (rev'd)



The Thundershot project – A real example



Robot Operating System

- Peer to peer
- Distributed
- Multi-lingual
- Light-weight
- Free and open-source

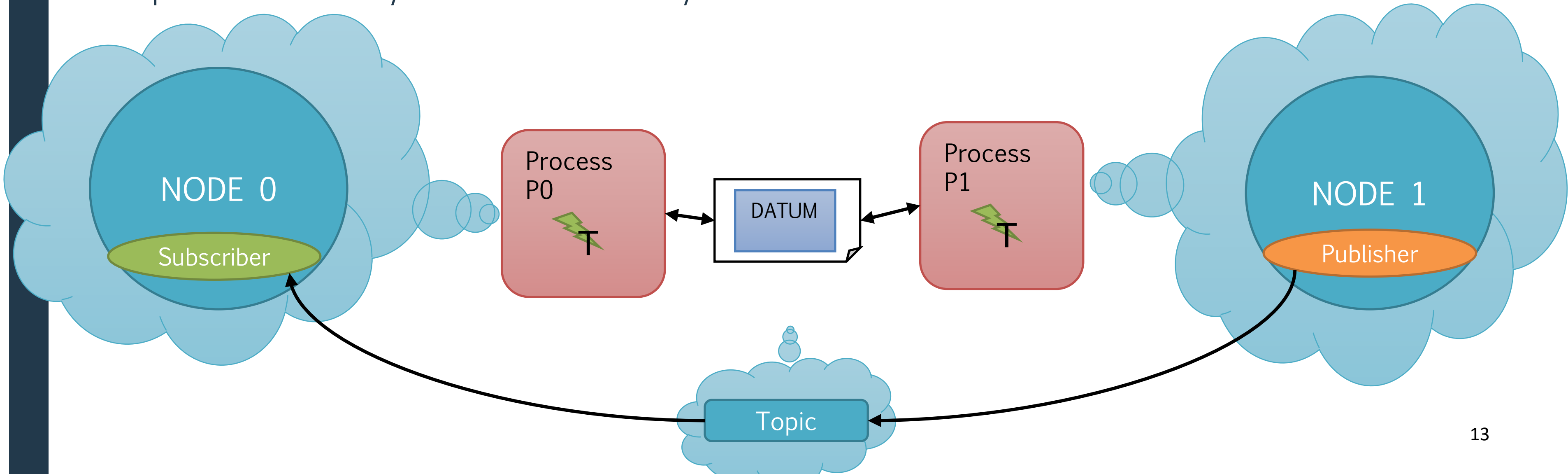
Recap: message-passing vs. shared memory

ROS is a pub-sub middleware

› ROS nodes are processes

› Communication happens via messages called topics

› Implements a *competitive* concurrency



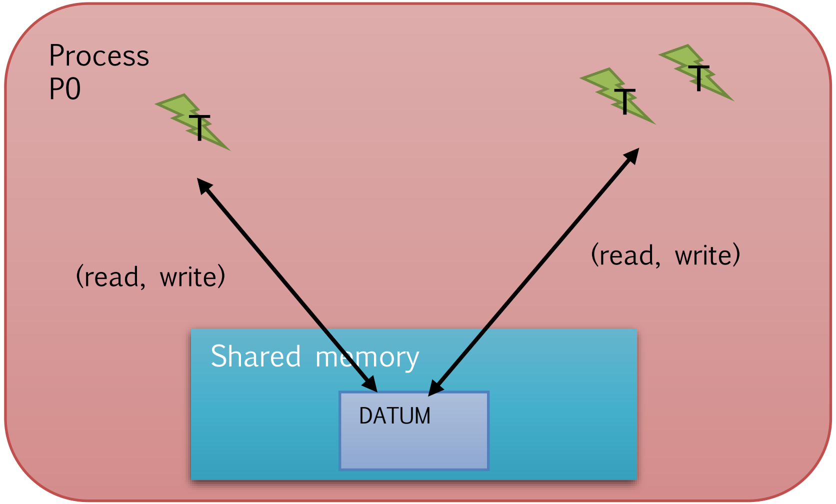
Recap: message-passing vs. shared memory

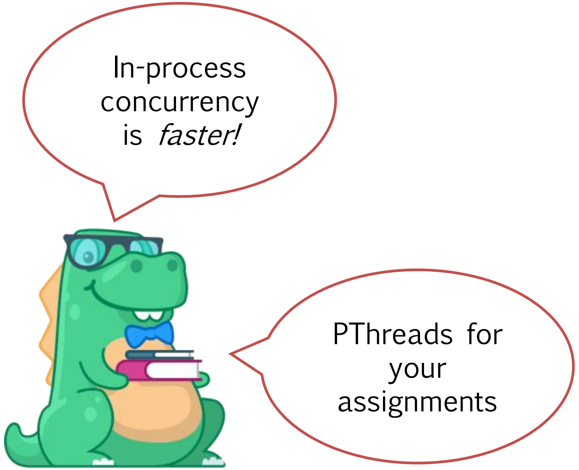
PThreads (OpenMP, etc..) are shared memory APIs

› Identify tasks, assigned to threads

› Communication happens via shared memory (issues with **data races**…)

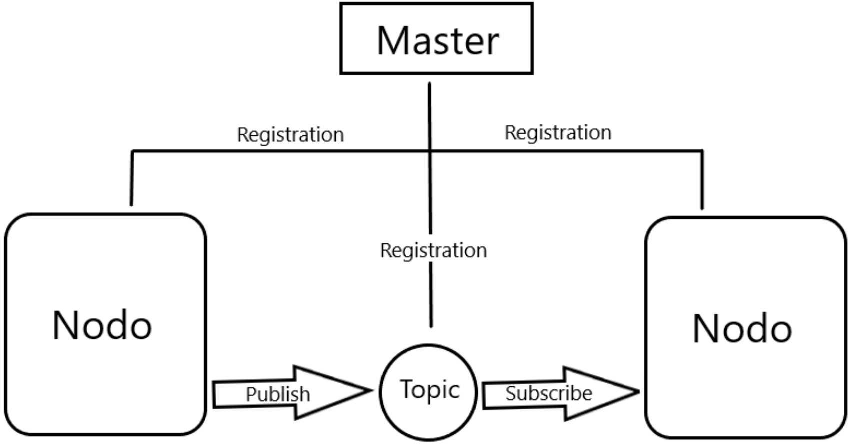
› Implements *cooperative* concurrency





ROS 1 structure

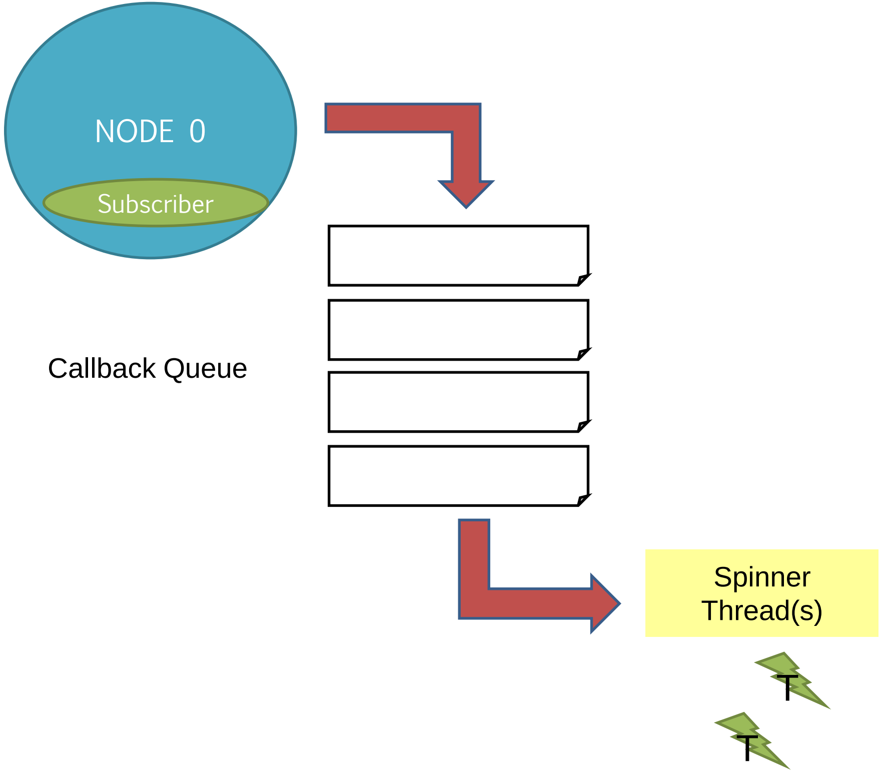
- Modular, based on the concept of <u>nodes</u>
- Message passing (<u>topics</u>)
  - Asynchronous, callbacks
- Package management + rich ROS Client Libraries (RCL)



Communication in ROS

ROS1 is NOT REAL TIME!!!

- No scheduling, no resource mgmt.
- "Simple" FIFO Pub-sub



ROS2 re-engineering process

Libreries

- ROS Client Libraries (RCLs)
- Rigid scheduling

DDS - Data Distribution Services

✓Data-Centric Publish-Subsctibe

✓Real-Time Publish-Subscribe for Transport

✓Distributed (*no master node*)

Quality of Service (QoS)

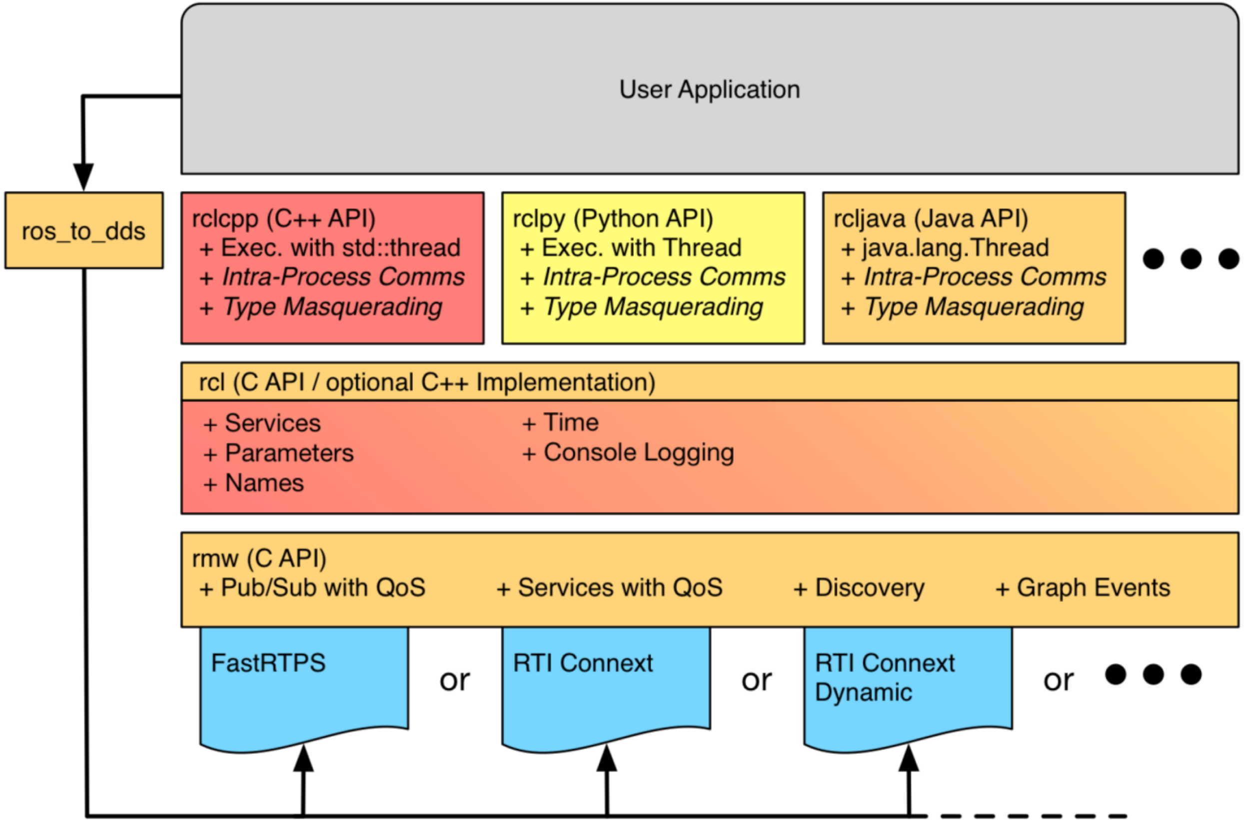
✓At the single entity

✓Suitable for Real-Time systems ☺

Recap: ROS1 v.s. ROS2

- Language standards: at least C++11 and Python3 for ROS2
  - Tighter Python integration
- Using off the shelf middleware. Now supports discovery, transport and serialization over DDS (Data Distribution Services)
  - Enables communication with non-ROS SW!!!!
- Distributed vs. centralized architecture
  - ROS 1 has a “core” node
- Real time capabilities!!!!
- **API change**

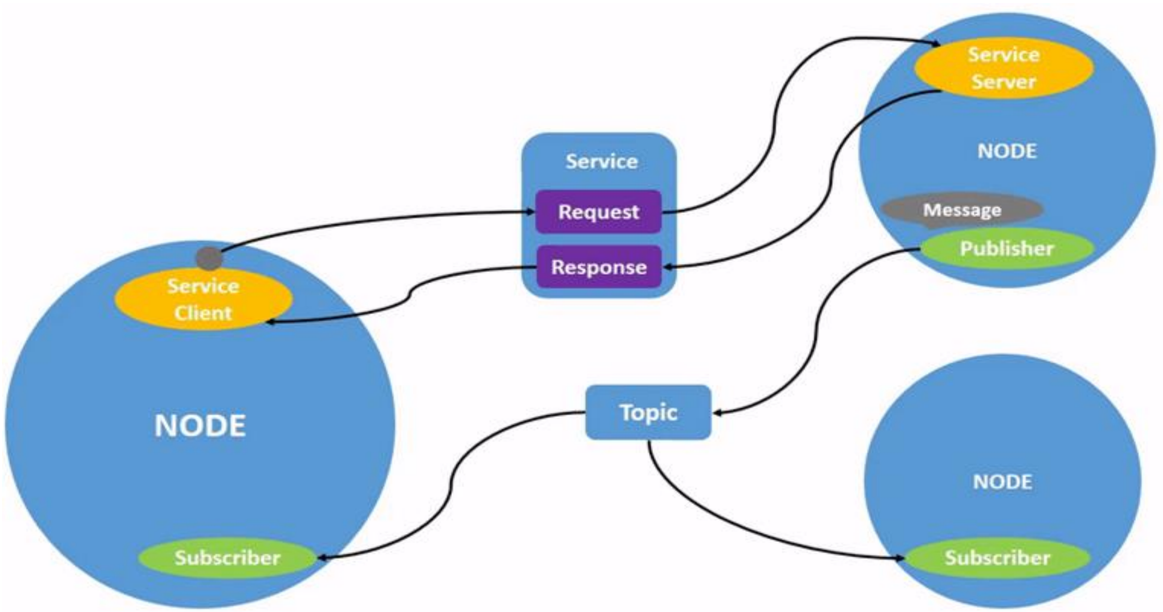
ROS2 stack



* *Intra-Process Comms* and *Type Masquerading* could be implemented in the client library, but may not currently exist.

ROS graph

The ROS graph is a network of ROS 2 elements processing data together at one time. It encompasses all executables and the connections between them if you were to map them all out and visualize them.



ROS Nodes

Each node in ROS should be responsible for a single, module purpose (e.g. one node for controlling wheel motors, one node for controlling a laser range-finder, etc)

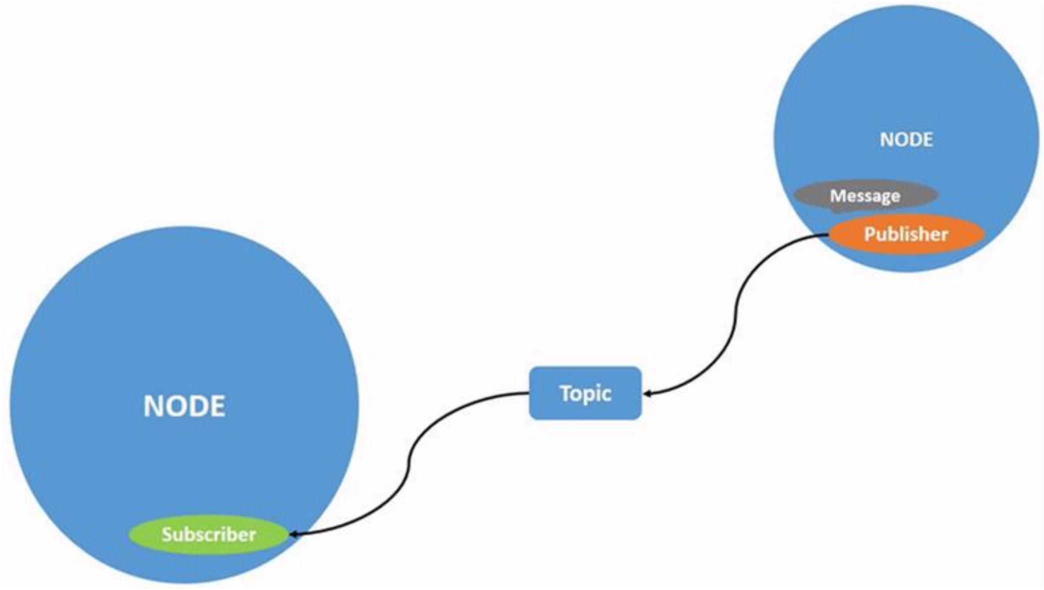
Each node can send and receive data to other nodes via topics, services, actions, or parameters.

Related command line commands

- `ros2 run <package_name> <executable_name>`
- `ros2 node list`
- `ros2 node info <node_name>`

Topics

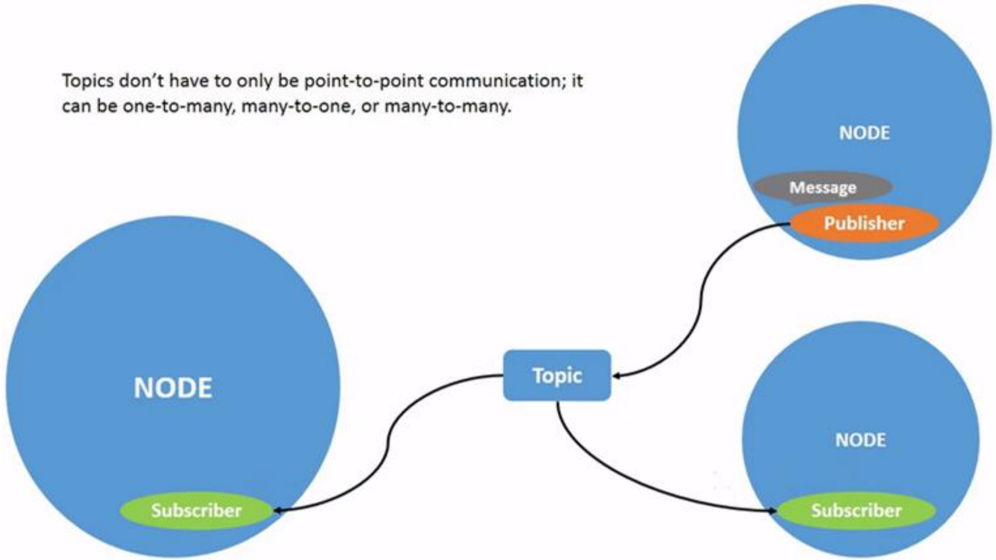
ROS 2 breaks complex systems down into many modular nodes. Topics are a vital element of the ROS graph that act as a bus for nodes to exchange messages.



Topics

A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics.

Topics are one of the main ways in which data is moved between nodes and therefore between different parts of the system.



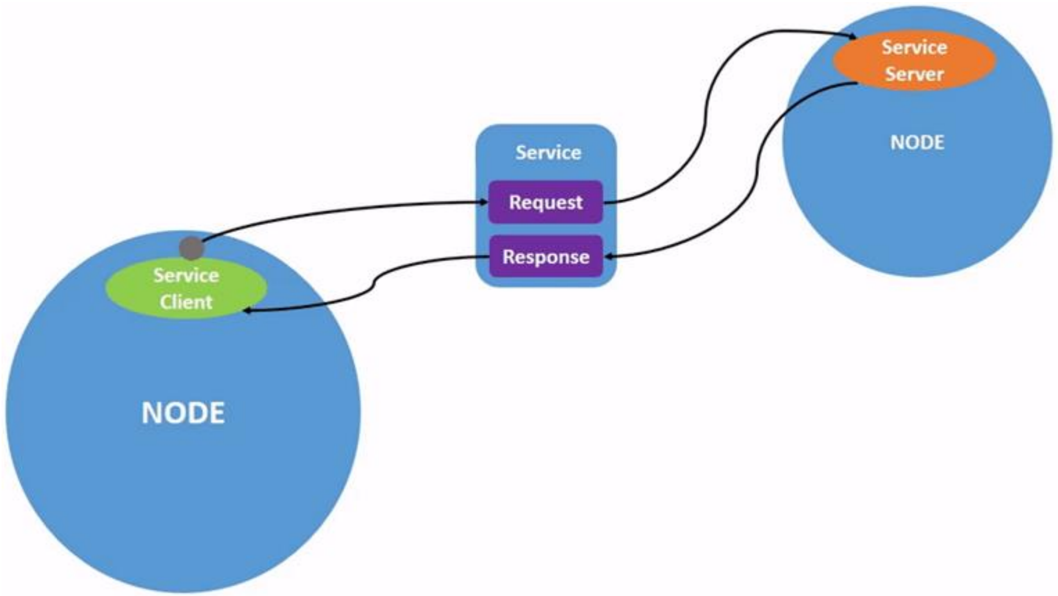
Topics

Related command line commands

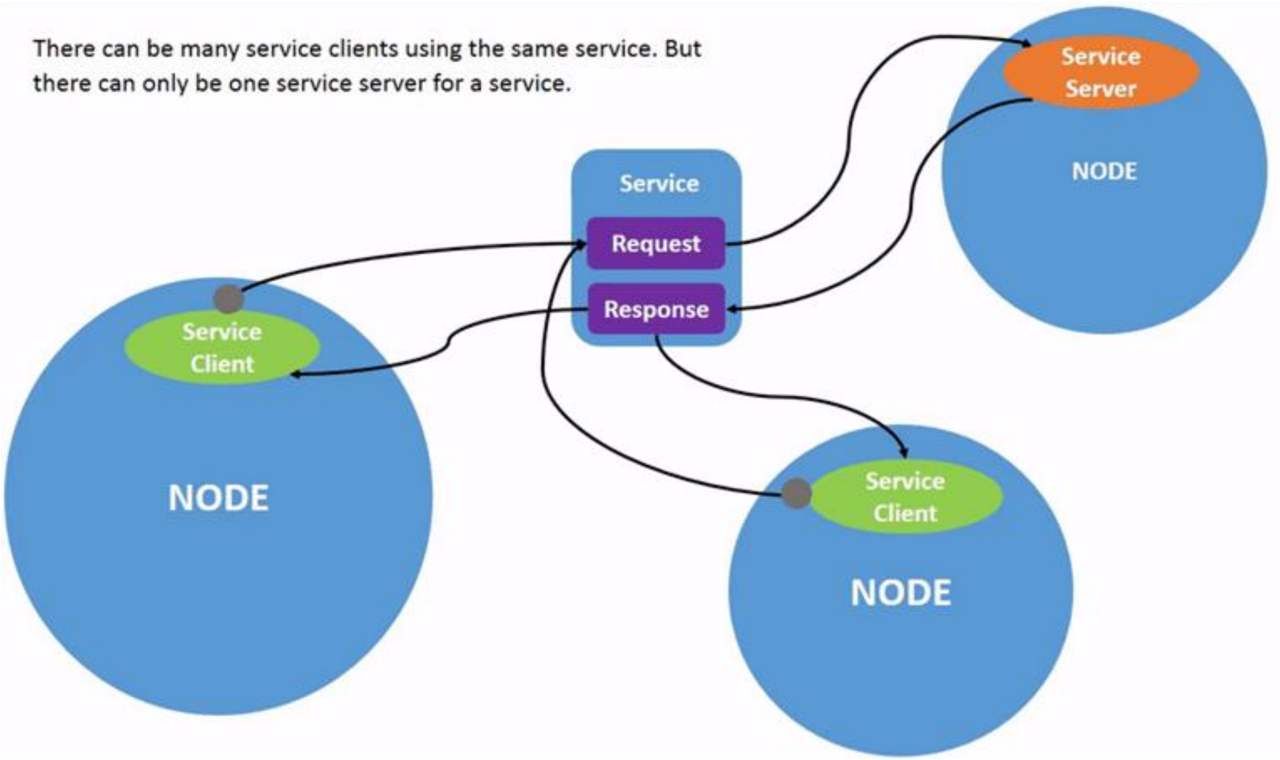
- `rqt_graph`
- `ros2 topic list`
- `ros2 topic list -t`
- `ros2 topic echo <topic_name>`
- `ros2 topic info <topic_name>`
- `ros2 interface show <msg_type>`
- `ros2 topic pub <topic_name> <msg_type> ‘<args>’`
- `ros2 topic hz <topic_name>`

Services

Services are another method of communication for nodes in the ROS graph. Services are based on a call-and-response model, versus topics' publisher-subscriber model. While topics allow nodes to subscribe to data streams and get continual updates, services only provide data when they are specifically called by a client.



Services


There can be many service clients using the same service. But there can only be one service server for a service.
Service
Server
NODE
Service
Request
Response
Service
Client
NODE
Service
Client
NODE

Services

Related command line commands

- `ros2 service list`
- `ros2 service type <service_name>`
- `ros2 service list -t`
- `ros2 service find <type_name>`
- `ros2 interface show <type_name>.srv`
- `ros2 service call <service_name> <service_type> <arguments>`

Parameters

- A parameter is a configuration value of a node
- Can be integers, floats, booleans, strings and lists
- All parameters are dynamically reconfigurable, and built off of ROS 2 services.

Related command line commands

- `ros2 param list`
- `ros2 param get <node_name> <parameter_name>`
- `ros2 param set <node_name> <parameter_name> <value>`

Could also use an yaml file to define the parameters.

Parameters

- Parameters could also be set in either a launch file or a yaml file.
- Useful to have all parameters in one place while tuning.
- Set parameters for multiple nodes in one file.
- For a full length tutorial: https://roboticsbackend.com/ros2-yaml-params/

Parameters

Getting ROS parameters programmatically:

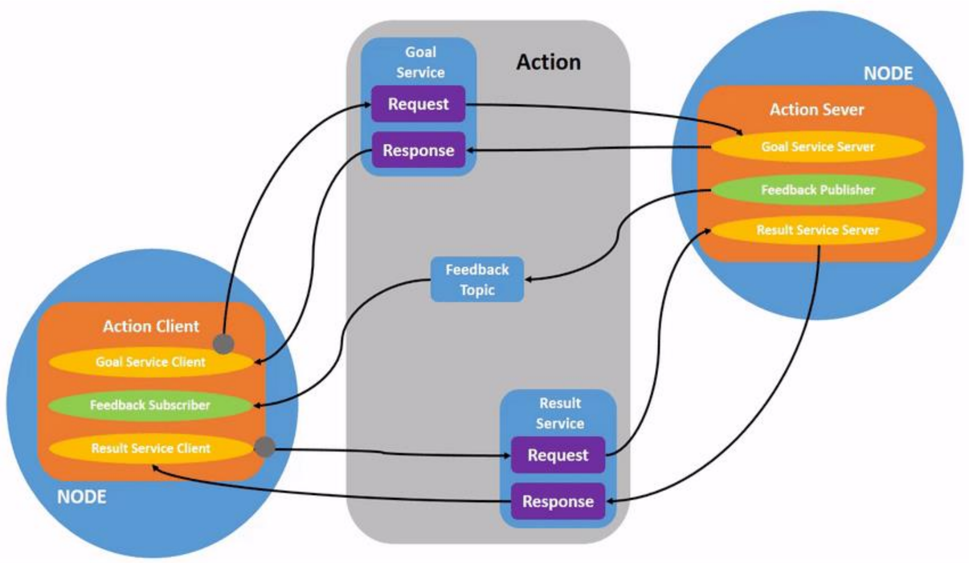
- In a node, declare a parameter first
  - `self.declare_parameter('my_param')`
- Then getting a parameter
  - `self.get_parameter('my_param')`
- You could also get multiple parameters at once
- Similar in C++
- For full tutorials on parameters see:
  - https://roboticsbackend.com/rclpy-params-tutorial-get-set-ros2-params-with-python/
  - https://roboticsbackend.com/rclcpp-params-tutorial-get-set-ros2-params-with-cpp/

Actions

Actions are one of the communication types in ROS 2 and are intended for long running tasks. They consist of three parts: a goal, feedback, and a result.

Actions are built on topics and services. Their functionality is similar to services, except actions are preemptable (you can cancel them while executing). They also provide steady feedback, as opposed to services which return a single response.

Actions use a client-server model, similar to the publisher-subscriber model (described in the topics tutorial). An "action client" node sends a goal to an "action server" node that acknowledges the goal and returns a stream of feedback and a result.



Quality of Service in ROS

ROS 1 uses TCP as the underlying transport

- “No good” (cit.) for lossy networks such as wireless links

ROS 2 uses DDS => UDP

- Better control over the level of reliability a node can expect and act accordingly
- Specify for: Topic, DataReader, DataWriter, Publisher and Subscriber
- Request vs Offerer Model. Publications and Subscriptions will match if the QoS are compatible

QoS policies in ROS2

- History
  - *Keep last* (only store up to N samples) vs. *Keep all*
- Depth
  - *Queue size* (if "history" was set to *Keep last*)
- Reliability
  - *Best effort* (lose samples if the network is not robust) vs. *Reliable* (might retry)
- Durability
  - *Transient local*: the publisher becomes responsible for persisting samples for "late-joining" subscriptions.
  - *Volatile*: no attempt is made to persist samples.

QoS policies in ROS2 – cont'd

- Deadline
  - *Duration*: the expected maximum amount of time between subsequent messages being published to a topic
- Lifespan
  - *Duration*: the maximum amount of time between the publishing and the reception of a message without the message being considered stale or expired (then dropped)
- Liveliness
  - *Automatic*: the system will consider all of the node's publishers to be alive for another "lease duration" when any one of its publishers has published a message.
  - *Manual by topic*: the system will consider the publisher to be alive for another "lease duration" if it manually asserts that it is still alive (via a call to the publisher API).
- Lease Duration
  - *Duration*: the maximum period of time a publisher has to indicate that it is alive before the system considers it to have lost liveliness (losing liveliness could be an indication of a failure).

QoS profiles in ROS2

Defines a set of policies that are expected to go well together for a particular use case.

› Remember that they must be compatible at both ends!

Default QoS settings for publishers and subscriptions

› *Keep last* for history with a *queue size* of 10, *reliable* for reliability, *volatile* for durability, and *system default* for liveliness. Deadline, lifespan, and lease durations are also all set to "*default*"

Services

› Services shall use *volatile* durability, as otherwise service servers that re-start may receive outdated requests

Sensor data

› More important to receive readings in a timely fashion, rather than ensuring that all of them arrive

› *Best effort* reliability and a smaller queue size

Parameters

› Parameters in ROS 2 are based on services, but with a larger queue depth

System default

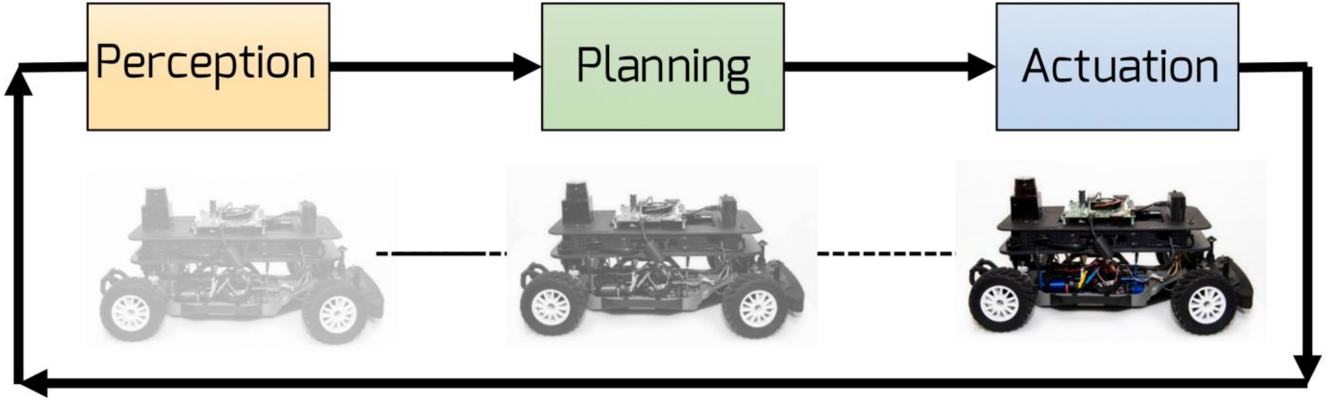
› This uses the RMW (*Ros MiddleWare* runtime) implementation's default values

ROS2 and F1/10

Real-Time Embedded System - The F1tenth autonomous racing

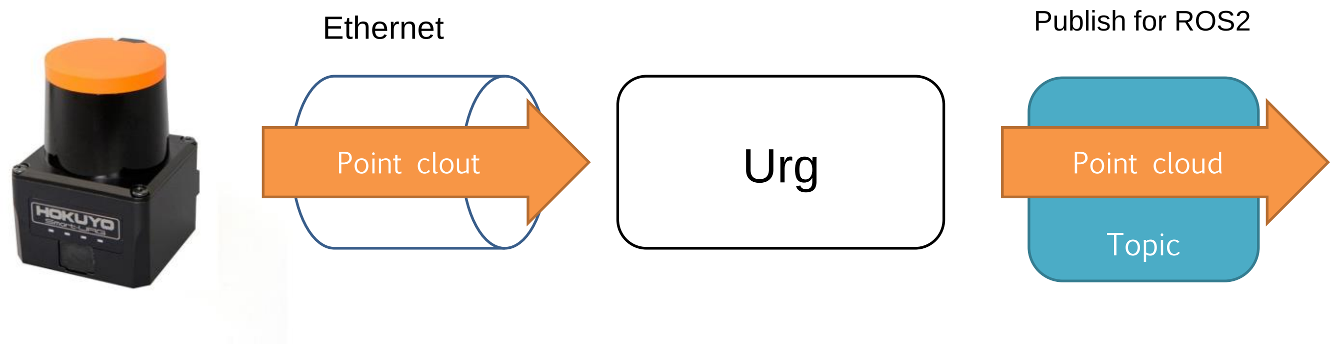
UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA
High Performance Real Time Lab

Our simplest stack



Urg → Localize (?) → Follow The Gap → Teensy

Urg – Data from LiDAR



Perception/Localization

Is this really necessary?

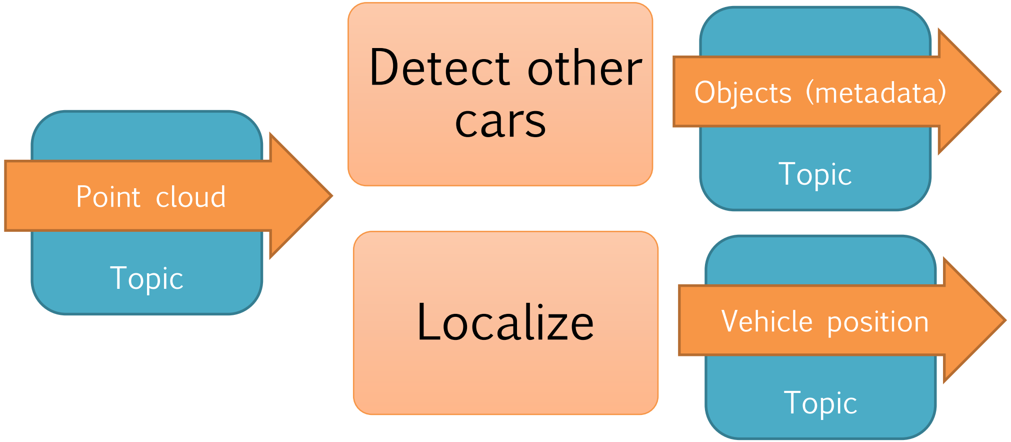
- YES! But not always...
- Remember, you have limited computing power...

In a time attack race, no overtaking/obstacle avoidance

- We use only (faster) local planners

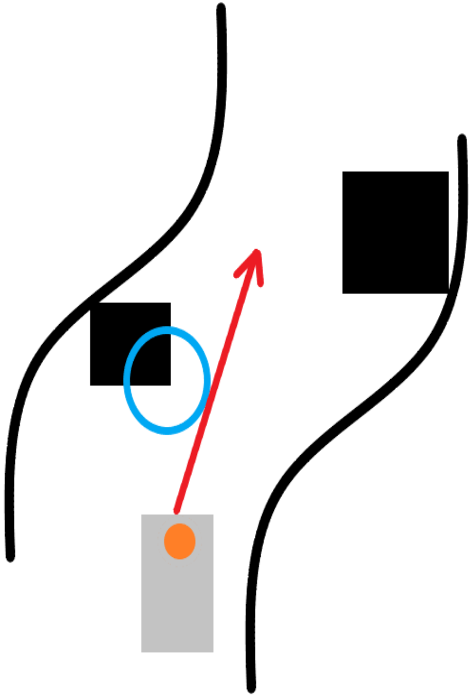
Within a simulator, you can directly fetch vehicle position

- Why is this useful?



Follow The Gap

- Local planner
- Fast, reactive
- Non-optimal trajectories
- Suitable for overtaking!



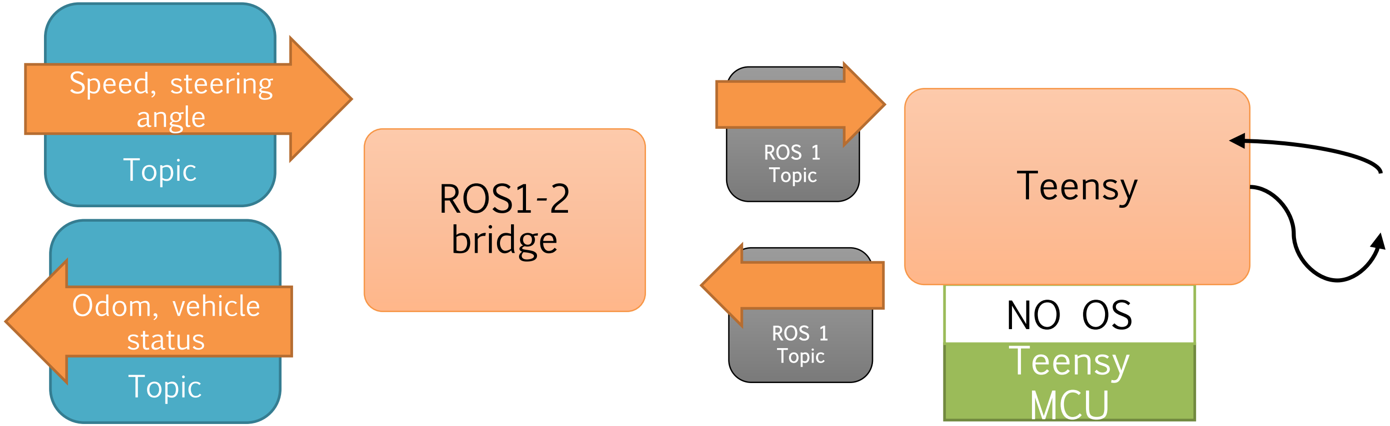
«Teensy» ROS node

Teensy is the microcontroller that controls the brushless engine

- Typical scenario, also real cars have <u>legacy</u> actuator ECUs!

Note written in ROS1 (teensy has no OS!)

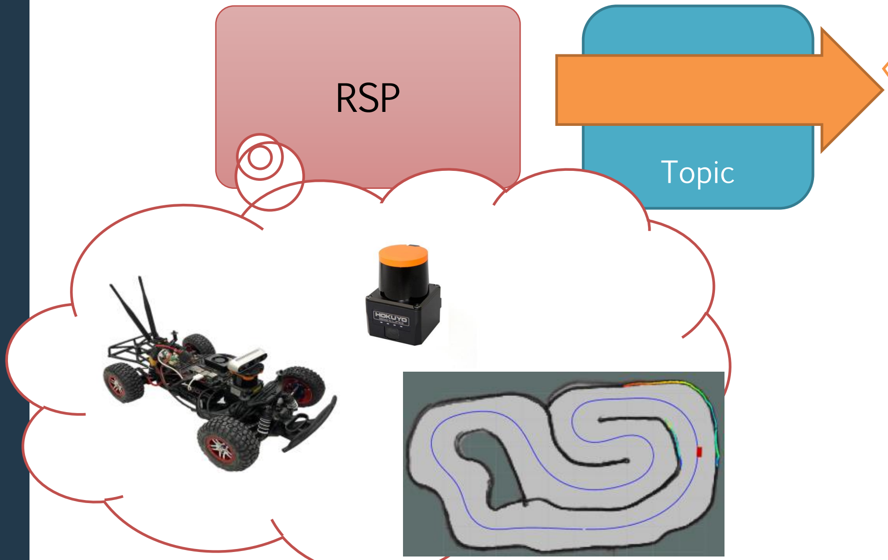
- ROS1-to-ROS2 bridge

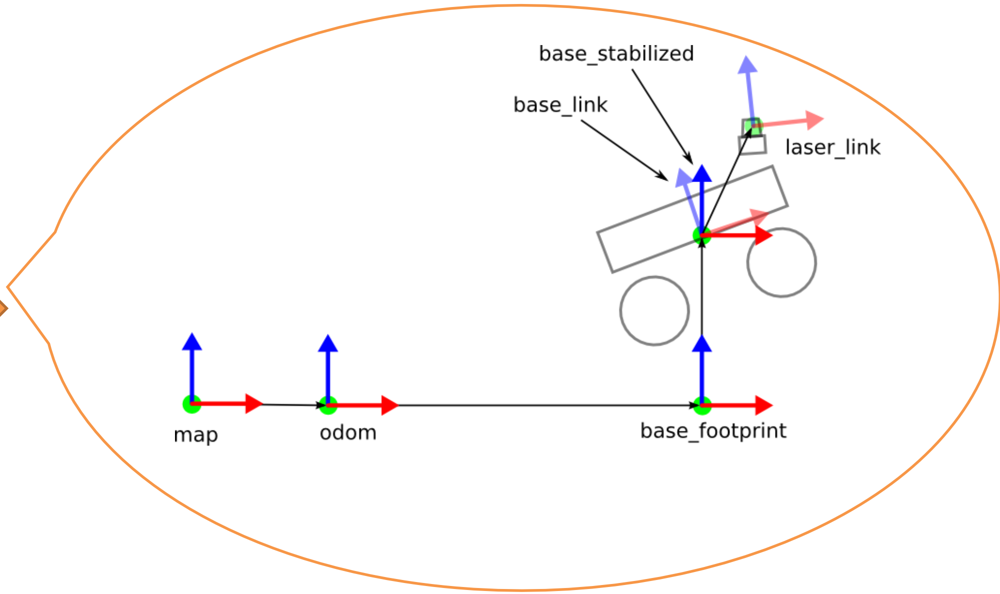


Notable nodes - Robot State Pubblisher

Required for ROS2 components to work – stores "static" information

- Knows the reference systems of various sensors
- Knows (possibly) the pre-built map
- Racing strategy?





Engineering a ROS program

Real-Time Embedded System - The F1tenth autonomous racing

UNIMORE UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

High Performance Real Time Lab

Code&build: ROS Workspaces

A workspace is a directory containing ROS 2 packages. Before using ROS 2, it's necessary to source your ROS 2 installation workspace in the terminal you plan to work in. This makes ROS 2's packages available for you to use in that terminal.

You also have the option of sourcing an "overlay" – a secondary workspace where you can add new packages without interfering with the existing ROS 2 workspace that you're extending, or "underlay".

Your underlay must contain the dependencies of all the packages in your overlay. Packages in your overlay will override packages in the underlay. It's also possible to have several layers of underlays and overlays, with each successive overlay using the packages of its parent underlays.

Code&build: ROS Workspaces

- Defines context for the current workspace

- Creating a new workspace
  - `$ mkdir -p <your_workspace>/src`
  - Then you can put your desired ROS2 packages inside src

- Resolving dependencies
  - `$ cd <your_workspace>`
  - `$ rosdep install -i --from-path src --rosdistro foxy -y`

Code&build: Overlays and Underlays

We refer to the main ROS 2 environment as the **underlay**. This has all the necessary setup to run ROS 2. Your workspaces are referred to as **overlays**. By sourcing your overlays you get access to your packages on top of the base ROS 2 environment.

After building, in a new terminal, source the underlay by:

- `$ source /opt/ros/foxy/setup.bash`

And in the root of your desired workspace:

- `$ cd <your_workspace>`
- `$ source install/local_setup.bash`
- Note that there's also a `install/setup.bash` in your workspace. You can also source this instead. This is equivalent to sourcing both your workspace overlay and the underlay your workspace was created/built in.

You might be tempted to leave these sourcing commands in your bashrc. I highly recommend **against** it. I've seen too many times that a teammate working on a package wondering why their code isn't working. And it ended up being `bashrc` sourcing the wrong overlay.

Colcon: the ROS build system

Build the workspace with `colcon`:

- From the root of your workspace: `$ colcon build`
- Useful arguments when building:
  - `--packages-up-to`: builds the package you want, plus all its dependencies, but not the whole workspace. This will save some time if you don't need all the packages in the workspace.
  - `--symlink-install`: saves you from having to rebuild every time you tweak Python scripts
  - `--event-handlers console_direct+`: shows console output while building (can otherwise be found in the `log` directory)

Once build finishes, you'll see `build` `install` `log` `src` directories in your workspace. The `install` directory is where your workspace's setup files are.

Release: ROS 2 Packages

- A package can be considered a container for your ROS 2 code. If you want to be able to install your code or share it with others, then you'll need it organized in a package. With packages, you can release your ROS 2 work and allow others to build and use it easily.

- Package creation in ROS 2 uses ament as its build system and colcon as its build tool. You can create a package using either CMake or Python, which are officially supported, though other build types do exist.

ROS 2 Packages

- Python and CMake packages each have their own minimum requirements.
- CMake packages:
  - `package.xml`: file containing meta info about the package
  - `CMakeLists.txt:` file describing how to build the code within the package
- Python packages:
  - `package.xml`: file containing meta info about the package
  - `setup.py`: contains instructions for how to install the package
  - `setup.cfg`: required when a package has executables, so ros2 run can find them
  - `/<package_name>:` a directory with the same name as your package, used by ROS 2 tools to find your package, contains `__init__.py`

ROS 2 Packages

The simplest package may have a file structure that looks like:

- CMake:
  - ```
    my_package/
         CMakeLists.txt
         package.xml
    ```
- Python:
  - ```
    my_package/
         setup.py
         package.xml
         resource/my_package
    ```

ROS 2 Packages

- A single workspace can contain as many packages as you want, each in their own folder. You can also have packages of different build types in one workspace (CMake, Python, etc.). You cannot have nested packages.

- Best practice is to have a src folder within your workspace, and to create your packages in there. This keeps the top level of the workspace “clean”.

ROS 2 Packages

A workspace with multiple packages might look like this:

```
workspace_folder/
    src/
      package_1/
          CMakeLists.txt
          package.xml

      package_2/
          setup.py
          package.xml
          resource/package_2
      ...
      package_n/
          CMakeLists.txt
          package.xml
```

ROS 2 Packages

Creating a package:

- `$ cd <your_workspace>/src`
- (Python packages): `$ ros2 pkg create --build-type ament_python <package_name>`
- (CMake packages): `$ ros2 pkg create --build-type ament_cmake <package_name>`

Package contents

- CMake packages
  - `CMakeLists.txt, include, package.xml, src`
  - Node source files (.cpp) are in src, and headers files (.h) in include
- Python packages
  - `my_package, package.xml, resource, setup.cfg, setup.py, test`
  - Node source files (.py) are inside the `my_package` directory

Customizing package.xml

- Fill in name and email on `maintainer` line, edit description to `summarize` the package, update the `license` line.
- Fill in your dependencies under the `_depend` tags. For documentation on what types of depend tags, see: https://www.ros.org/reps/rep-0149.html#build-depend-multiple

Customizing setup.py

- For Python packages, you'll also need to fill in `setup.py`
- Fill in the same description, maintainer, and license fields as in `package.xml`. You'll need to match these exactly. You'll also need to match the `package_name` and `version`.

Publisher (Python)

```python
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'topic', 10)
        timer_period = 0.5  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback
        self.i = 0
    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1
```

```python
def main(args=None):
    rclpy.init(args=args)
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
if __name__ == '__main__':
    main()
```

https://docs.ros.org/en/foxy/Tutorials/Writing-A-Simple-Py-Publisher-And-Subscriber.html

Adding Dependencies and Entrypoint

- Filling in the dependencies in package.xml for our example node:
  - `<exec_depend>rclpy</exec_depend>`
  - `<exec_depend>std_msgs</exec_depend>`
  - This declares the package needs `rclpy` and `std_msgs` when code is executed.
- Add an entrypoint in setup.py:

```
entry_points={
        'console_scripts': [
                'talker = py_pubsub.publisher_member_function:main',
        ],
},
```

Rosdep: installing dependencies

- ROS 2 uses `rosdep` to install package dependencies.
- https://index.ros.org/ maintains repos and packages you can use as dependencies and has recipes for installation for `rosdep`.
- To install dependencies for a workspace, in the workspace directory:
  - `rosdep install -i --from-path src --rosdistro foxy -y`
  - This will install dependencies declared in `package.xml` from all packages in the `src` directory for ROS 2 foxy

Subscriber (Python)

```python
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalSubscriber(Node):
    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String,
            'topic',
            self.listener_callback,
            10)
        self.subscription  # prevent unused variable warning
    def listener_callback(self, msg):
        self.get_logger().info('I heard: "%s"' % msg.data)
```

```python
def main(args=None):
    rclpy.init(args=args)

    minimal_subscriber = MinimalSubscriber()

    rclpy.spin(minimal_subscriber)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

Adding Entrypoint

- Since we've already added the dependencies to `package.xml`, we'll only need to add the entrypoint for the subscriber here:
-
```
entry_points={
        'console_scripts': [
                'talker = py_pubsub.publisher_member_function:main',
                'listener = py_pubsub.subscriber_member_function:main',
        ],
},
```

Launch files

- Launch files allow you to start up and configure a number of executables containing ROS 2 nodes simultaneously.

- Running a single launch file with the `ros2 launch` command will start up your entire system - all nodes and their configurations - at once.

- In your package, create a new directory for launch files: `$ mkdir launch`
- Then create your launch file: `$ touch <your_launch>.py`
- Note that launch files are written in Python now in ROS 2 instead of xml in ROS 1. This allows for access to Python libraries.

Example Launch file

```python
from launch import LaunchDescription
from launch_ros.actions import Node
from launch.substitutions import Command
from ament_index_python.packages import get_package_share_directory
import os
import yaml

def generate_launch_description():
    ld = LaunchDescription()
    config = os.path.join(
        get_package_share_directory('f1tenth_gym_ros'),
        'config',
        'sim.yaml'
        )
    config_dict = yaml.safe_load(open(config, 'r'))

    bridge_node = Node(
        package='f1tenth_gym_ros',
        executable='gym_bridge',
        name='bridge',
        parameters=[config]
    )
    rviz_node = Node(
        package='rviz2',
        executable='rviz2',
        name='rviz',
        arguments=['-d', os.path.join(get_package_share_directory('f1tenth_gym_ros'), 'launch', 'gym_bridge.rviz')]
    )
```

Example Launch file

```
    map_server_node = Node(
        package='nav2_map_server',
        executable='map_server',
        parameters=[{'yaml_filename': config_dict['bridge']['ros__parameters']['map_path'] + '.yaml'},
                    {'topic': 'map'},
                    {'frame_id': 'map'},
                    {'output': 'screen'},
                    {'use_sim_time': True}]
    )
    nav_lifecycle_node = Node(
        package='nav2_lifecycle_manager',
        executable='lifecycle_manager',
        name='lifecycle_manager_localization',
        output='screen',
        parameters=[{'use_sim_time': True},
                    {'autostart': True},
                    {'node_names': ['map_server']}]
    )
    ego_robot_publisher = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='ego_robot_state_publisher',
        parameters=[{'robot_description': Command(['xacro ', os.path.join(get_package_share_directory('f1tenth_gym_ros'), 'launch', 
'ego_racecar.xacro')])}],
        remappings=[('/robot_description', 'ego_robot_description')]
    )
```

Example Launch file

```
# finalize
    ld.add_action(rviz_node)
    ld.add_action(bridge_node)
    ld.add_action(nav_lifecycle_node)
    ld.add_action(map_server_node)
    ld.add_action(ego_robot_publisher)

    return ld
```

How to run the examples

Let's
code!

› Find them in `Code/` folder from the course website

For building ROS2 workspaces

```
$ colcon build
```

Run

```
$ ros2 launch <LAUNCH-FILE>
```

References

Course website

- http://personale.unimore.it/rubrica/contenutiad/markober/2023/71846/N0/N0/10005
- https://github.com/HiPeRT/F1tenth-RTES
  - Online resources/preview

My contacts

- paolo.burgio@unimore.it
- http://hipert.mat.unimore.it/people/paolob/

Resources

- https://docs.ros.org/en/foxy/Tutorials.html
- https://roboticsbackend.com/category/ros2/
- https://docs.ros.org/en/foxy/Tutorials/Writing-A-Simple-Py-Publisher-And-Subscriber.html
- https://it.mathworks.com/help/ros/ug/manage-quality-of-service-policies-in-ros2.html
- A "small blog"
  - http://www.google.com