Robot Operating System 2 (ROS 2)

Why ROS

Robots are becoming more affordable, more capable, and more useful in many real life scenarios → need to share spaces and work together

Robot Operating System 2 (ROS 2) and **Robot Middleware Framework** (RMF) tries to simplify the creation and operation of complex multi-robot systems

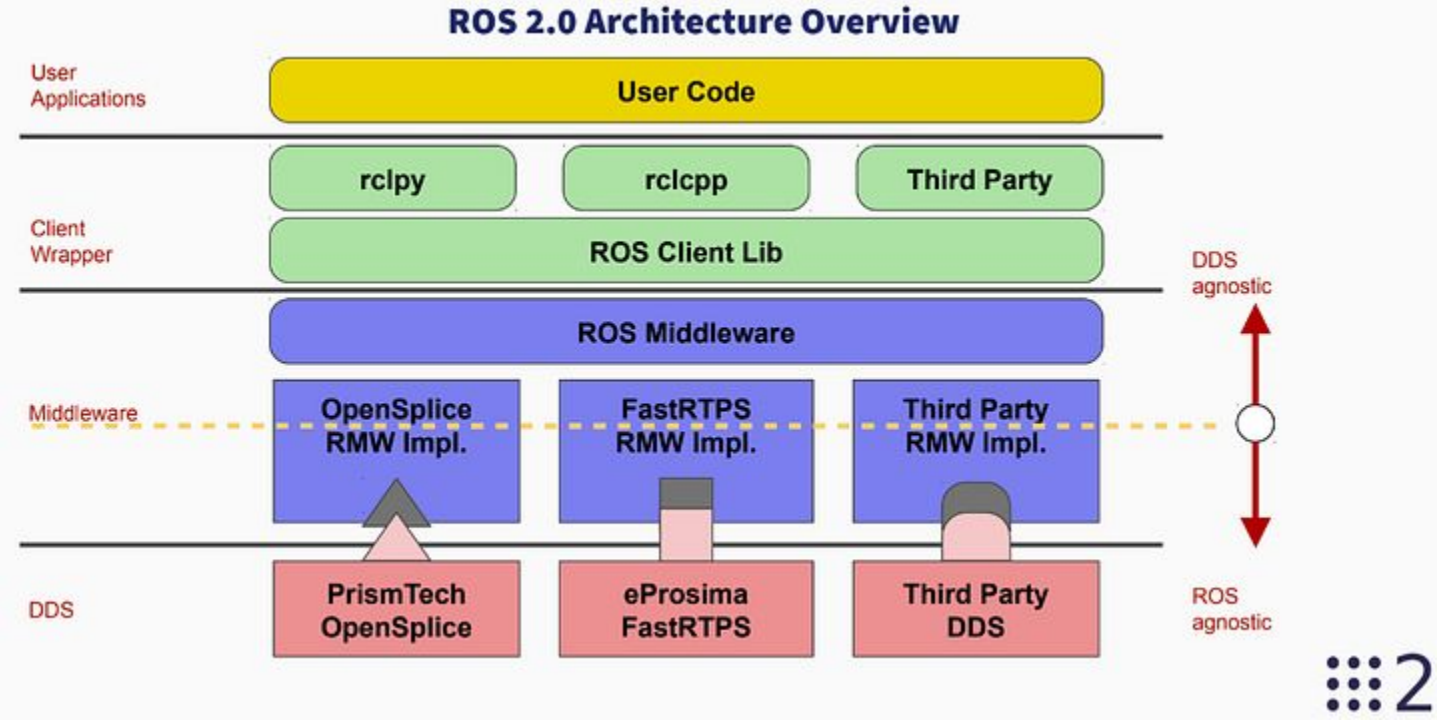
From ROS 1 to ROS 2

The Robot Operating System (ROS) is a set of software libraries and tools for building robot applications.

ROS 1 → robots, wheeled robots of all sizes, legged humanoids, industrial arms, outdoor ground vehicles

ROS 2 → “simply” supports new cases but in an **interoperable** way

ROS Architecture

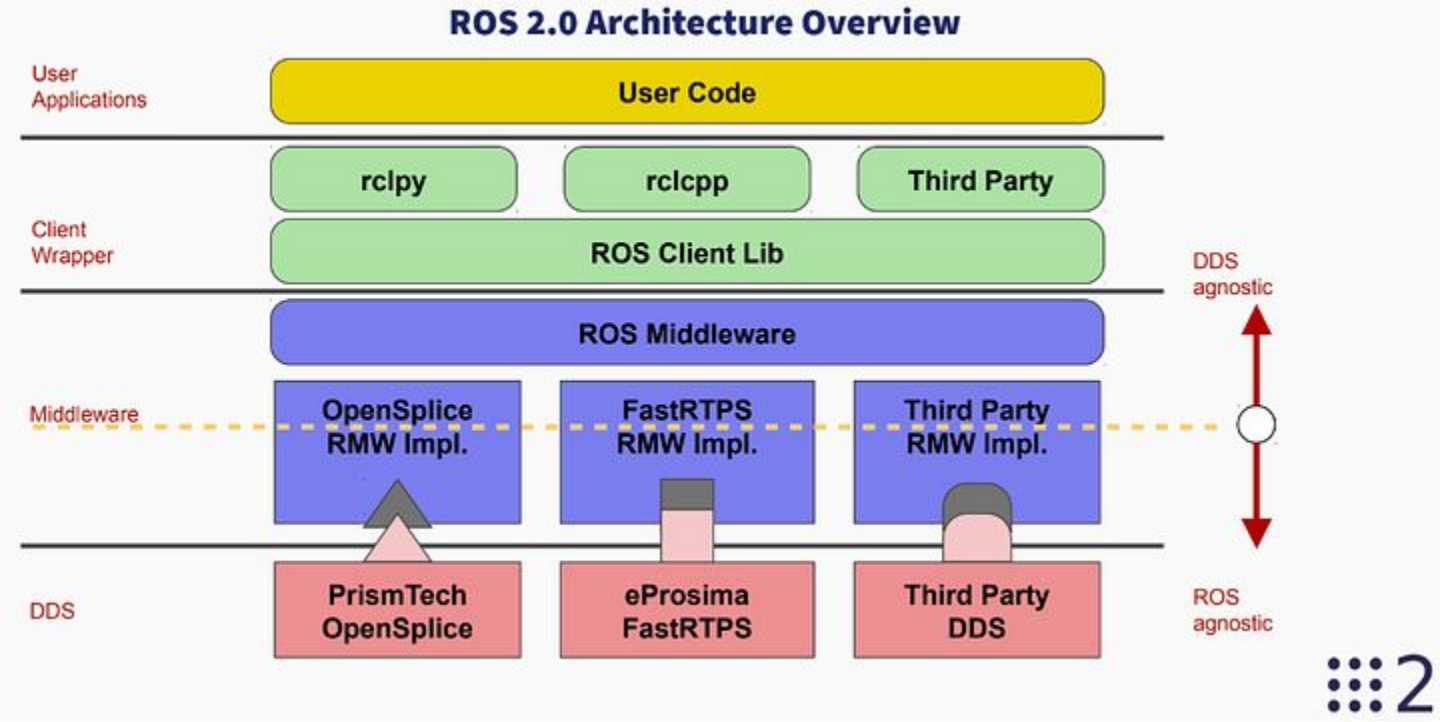


Source: https://mab-robotics.medium.com/legged-robots-ros2-6051f9c907cd

ROS Architecture

ROS client layer (RCL): user-facing interface that provides high-level functionalities

ROS middleware layer (RMW): provides real-time publish/subscribe protocol

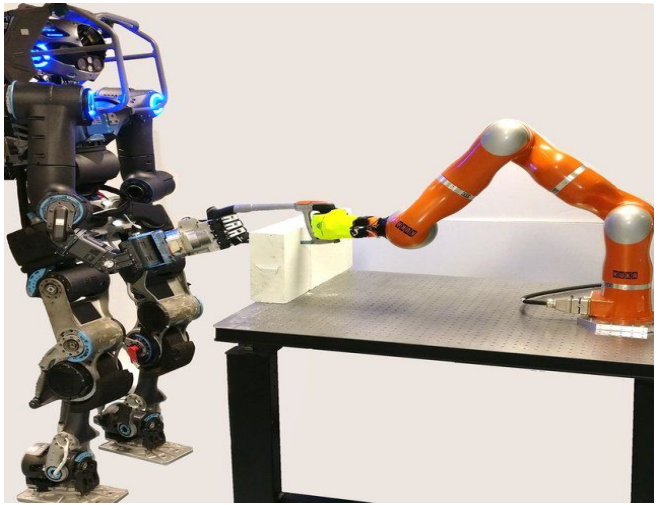


Robotics Middleware Framework (RMF)

Robots start to perform those tasks **lack of abstract planning**, reasoning and informal communication

Also, **multi-robot systems from multi-vendors** remain a challenge

RMF → provides a set of conventions, tools, and software implementations to allow multiple robots to **interoperate with each other and with shared information**



RMF in Practice

RMF is a collection of reusable, scalable libraries and tools building on top of ROS 2

→ interoperability of heterogeneous robotic systems

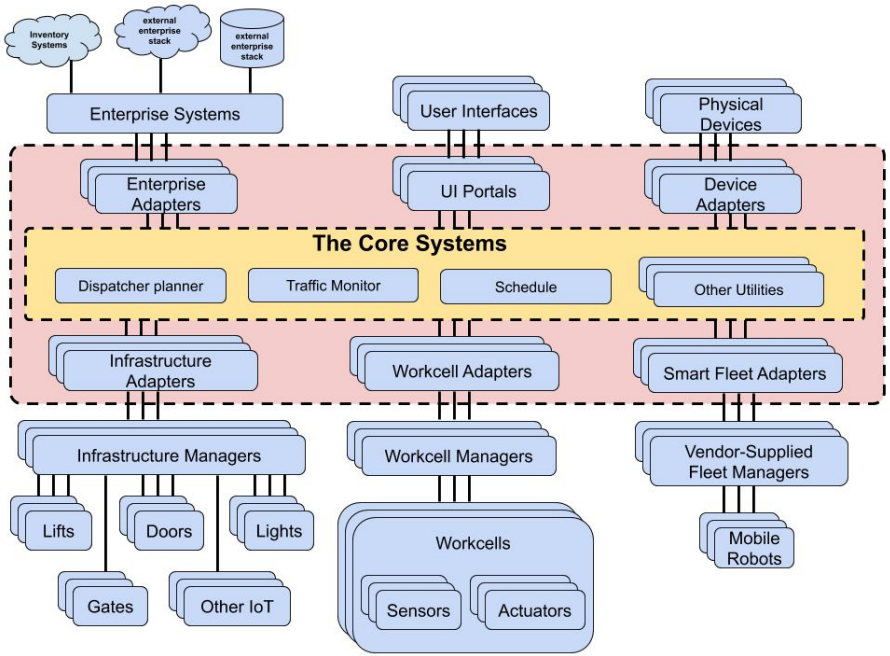
It adds intelligence to the system through **resource** allocation and by preventing **conflicts** over shared resources

RMF Overview

RMF operates over virtually any communications layer and integrates with any number of devices

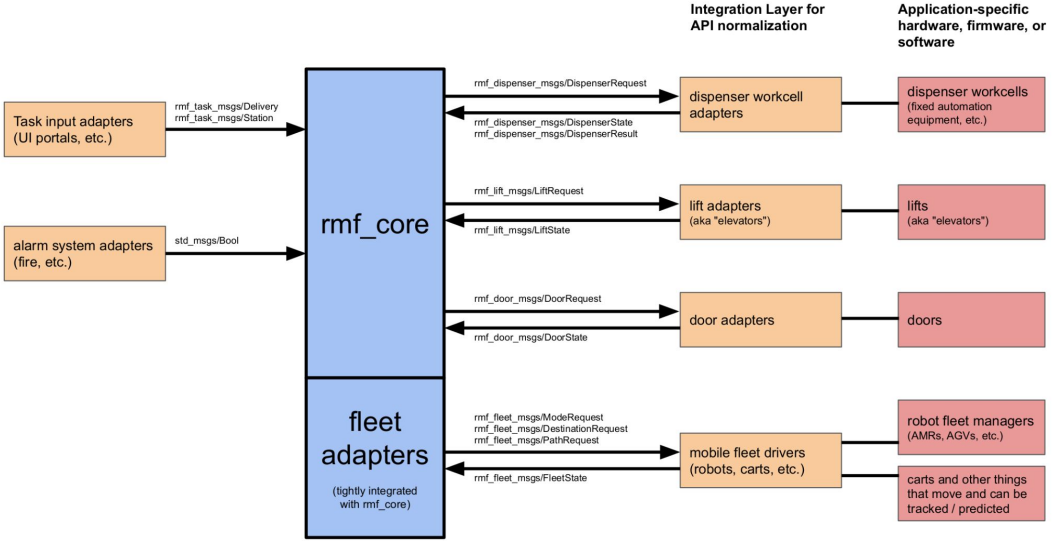
RMF architecture allows for scalability as the level of automation in an environment increases

RMF also saves costs by sharing resources and integrations to be minimized



Message Passing

Another RMF goal is to simplify and standardize messaging as much as possible

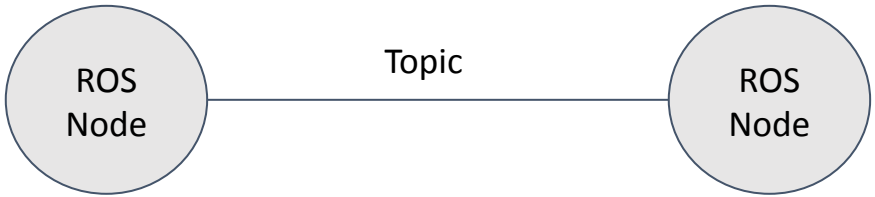


ROS Concepts and Design Patterns

A robot can be seen as a **distributed system**: each part plays a well-defined role, communicating as needed with other parts

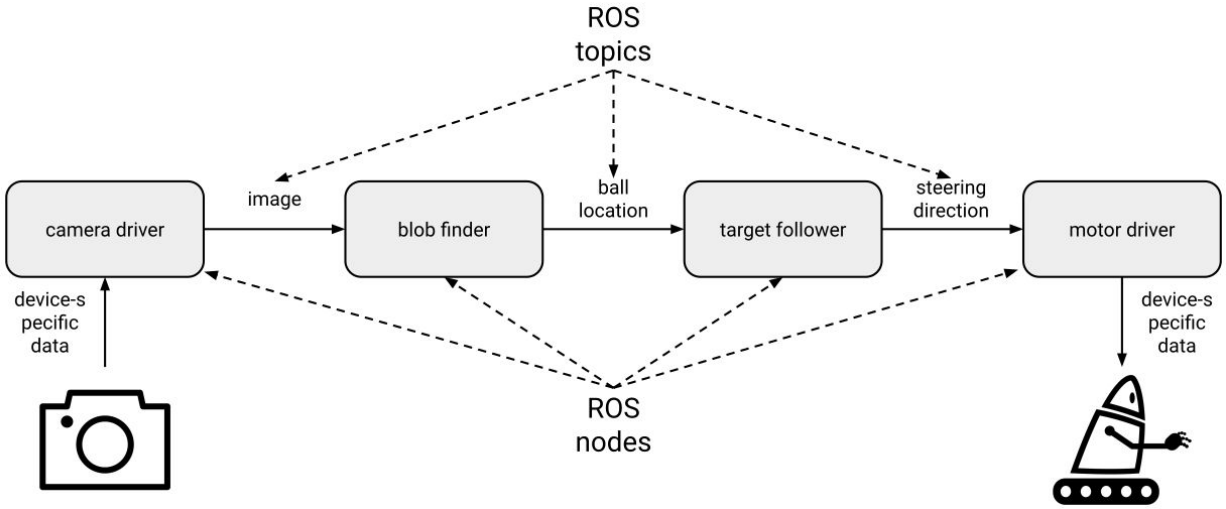
ROS separates the functions of a complex system into individual **parts** that **interact** with each other to produce the desired behavior of that system.

Parts → nodes **Interactions → topics**



ROS Communication Graph

Wheeled robot that chases a red ball with a camera to see the ball, a vision system to process the camera images, a control system to decide what direction to move, and some motors to move the wheels to allow it to move



Publish-Subscribe Messaging: Topics and Types

ROS uses **publish-subscribe messaging** where data is sent as messages from **publishers** to **subscribers**

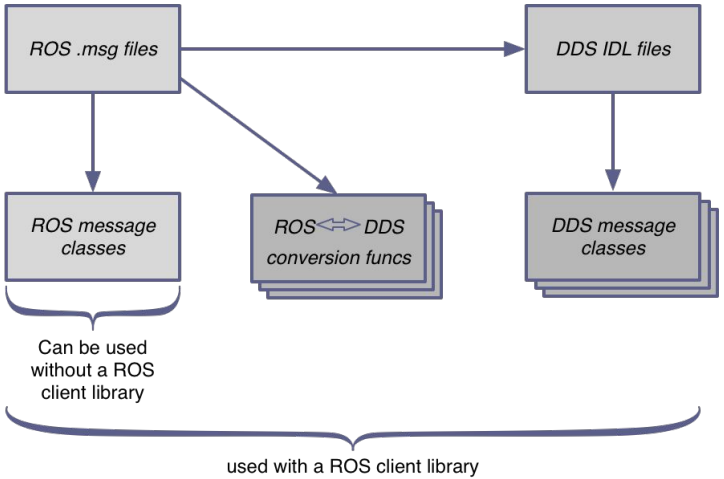
A publisher may have zero, one, or **multiple subscribers listening** to its published messages

Messages may be published at any time, making the system **asynchronous**

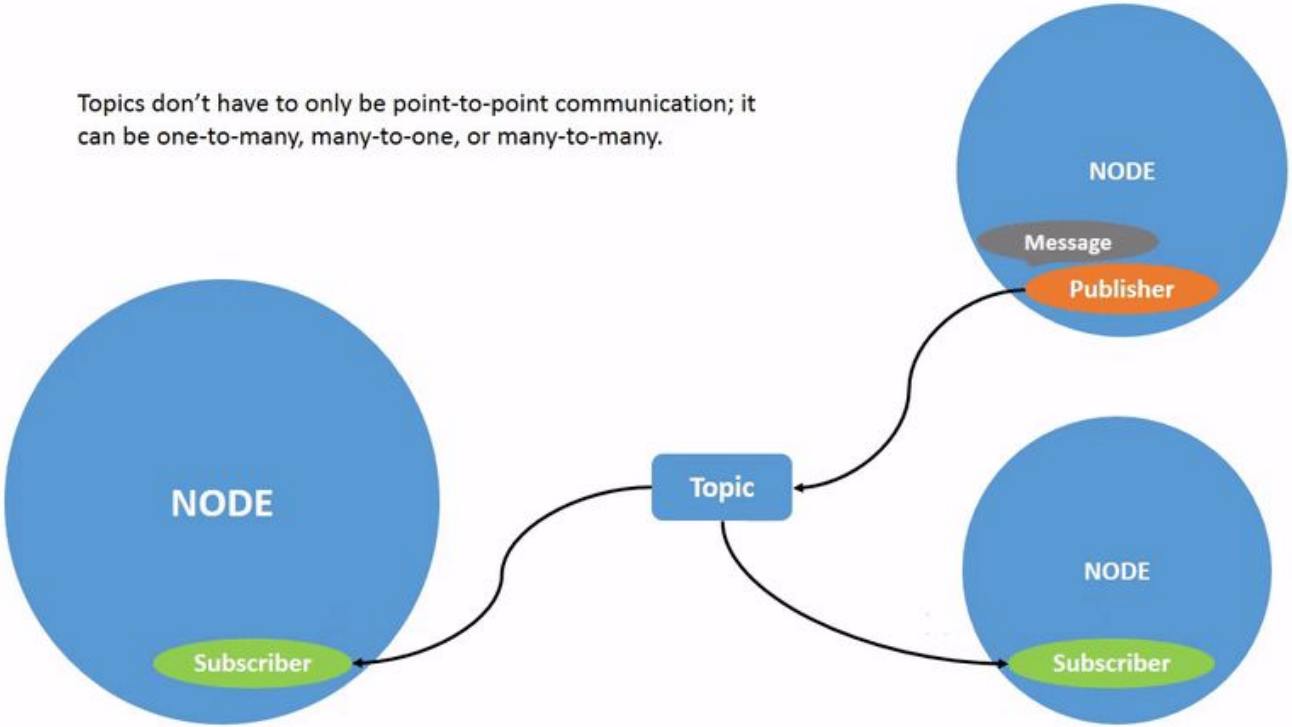
ROS2 Messages

.msg files are used in ROS2 and they are converted into .idl files so that they could be used with the DDS transport

The ROS 2 API would work exclusively with the .msg style message objects in memory and would convert them to .idl objects before publishing



Topics don't have to only be point-to-point communication; it can be one-to-many, many-to-one, or many-to-many.



Source: https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html

Pub-Sub ROS benefits

- **Substitution**: upgrade a component only requires to modify the ROS node
- **Reuse**: a node can be used in many different components
- **Collaboration**: separating functionalities on different nodes that work together
- **Introspection**: nodes are explicitly communicating with each other via topics, so it is possible to visualize, log, and play back node's inputs and outputs
- **Fault tolerance**: run nodes in separate processes allows for fault tolerance, or run them together in a single process, which can provide higher performance
- **Language independence**: running those nodes in separate processes permits to use different technologies

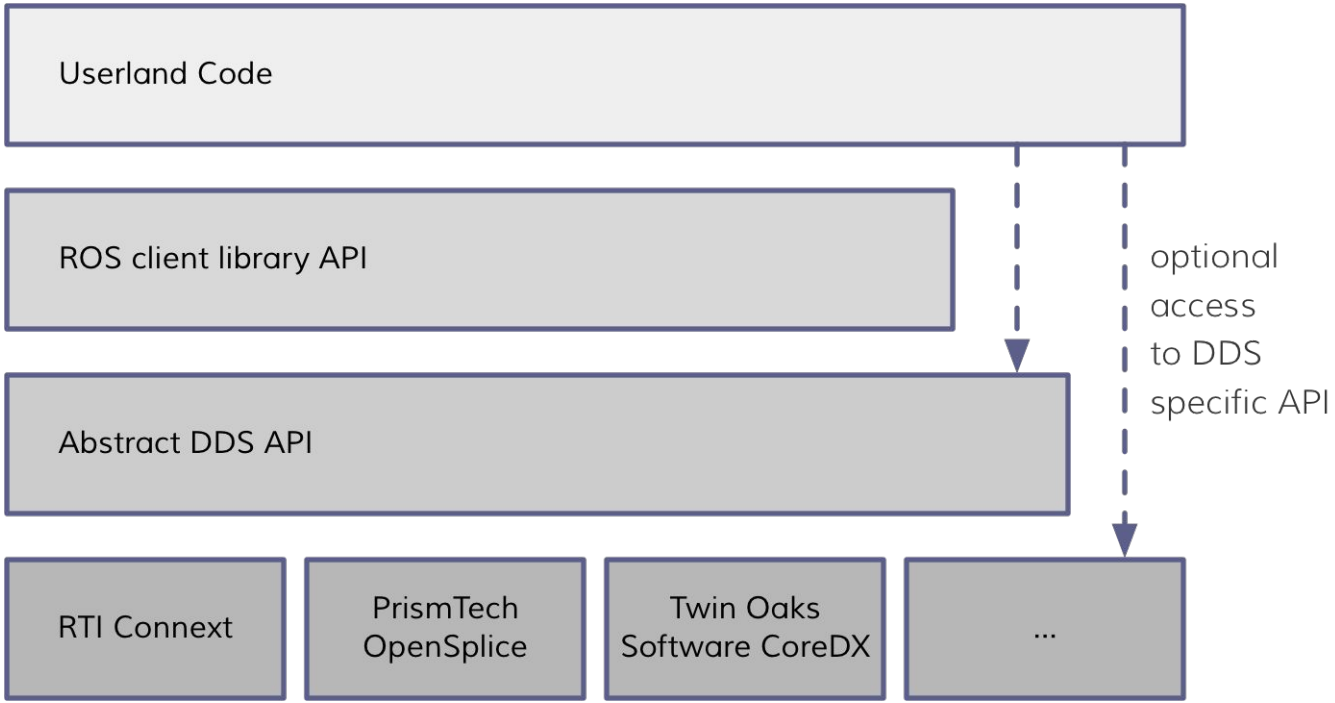
ROS Built on DDS

DDS is an implementation detail of ROS2 → DDS APIs and message definitions are hidden

ROS2 provides an **interface** on top of DDS which **hides much of the complexity** of DDS for the majority of ROS users.

It also **separately provides access to the underlying DDS implementation** for users that have extreme use cases or need to integrate with other, existing DDS systems

ROS Built on DDS



Source: https://design.ros2.org/articles/ros_on_dds.html

Middleware Interface

ROS2 relies on DDS communication protocol

how to deal with the different implementations? → a **middleware abstract interface** is introduced and it can be implemented for different DDS implementations

This middleware interface defines the API between the ROS client library and any specific implementation

ROS2 to MI to DDS

1. ROS data objects → 2. Middleware data format → 3. DDS implementation

1. DDS implementation → 2. Middleware data format → 3. ROS data objects

A defined mapping defines how to convert the primitive data types of ROS to middleware specific ones

Mapping of primitive types

ROS type	DDS type
bool	boolean
byte	octet
char	char
float32	float
float64	double
int8	octet
uint8	octet
int16	short
uint16	unsigned short
int32	long
uint32	unsigned long
int64	long long
uint64	unsigned long long
string	string

Mapping of arrays and bounded strings

ROS type	DDS type
static array	T[N]
unbounded dynamic array	sequence
bounded dynamic array	sequence<T, N>
bounded string	string

ROS Node to DDS Participant

ROS Node (Publishers, Subscriptions, Servers, Clients) participates in a Context (encapsulating sharing states)

DDS Participant (Publishers, Subscribers, Data Writters, Data Readers) discovery, tracking, thread creation makes it a heavyweight entity

one-to-one mapping associates a node to a participant → overhead with multiple participants, extra information needed to discover them

Services

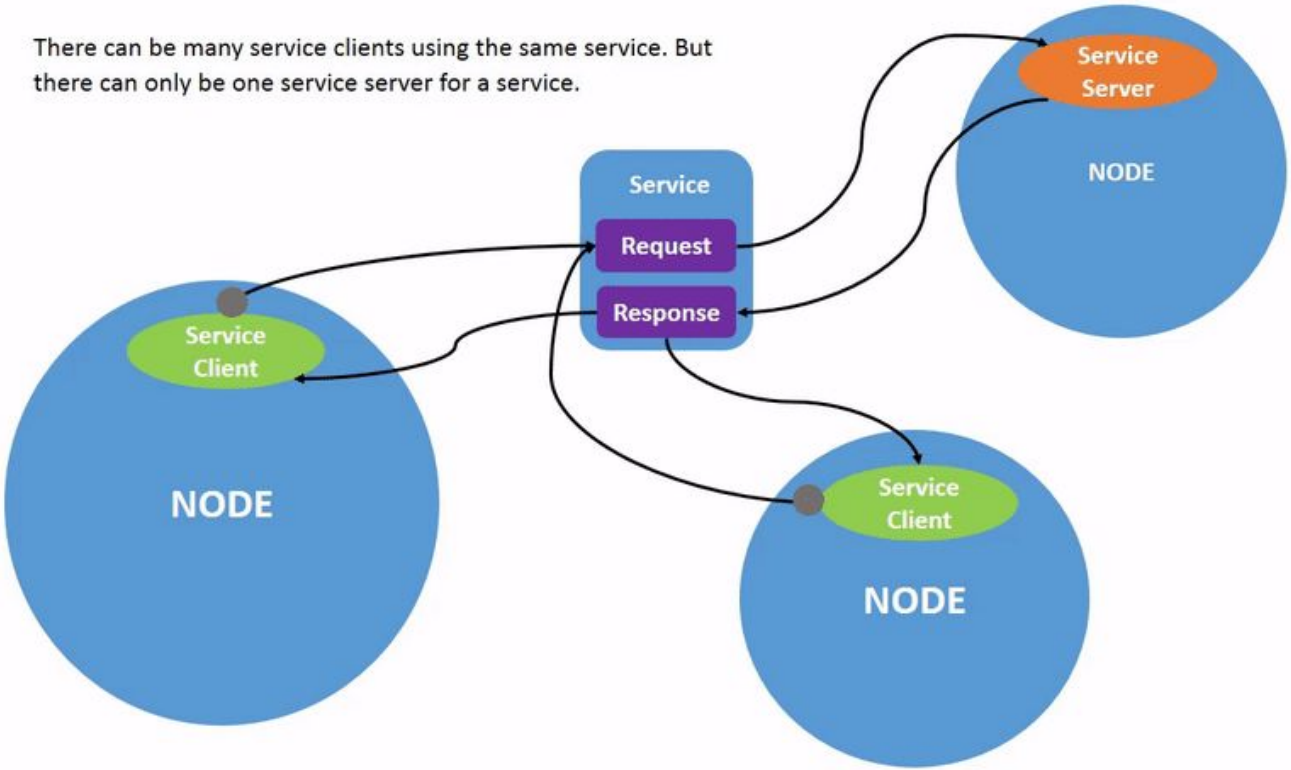
An alternative pattern **request-reply**: a ROS service is a form of **remote procedure call (RPC)**, the call may be dispatched to another process or even another machine on the network

Request and a reply

Node calling the service populates the request message and sends it to the node implementing the service, where the request is processed, resulting in a reply message that is sent back

Those two entities are called Service client and Service server

There can be many service clients using the same service. But there can only be one service server for a service.



Actions

Interactions request-reply but with **high latency** → goal-oriented, time-extended tasks, cancelable

Action composed of 3 messages:

1. **Goal**: sent once and indicates what action is trying to achieve
2. **Result**: sent once indicates what happened
3. **Feedback**: sent periodically updates the caller on how things are going

We can distinguish between Action client and Action server

Action Server

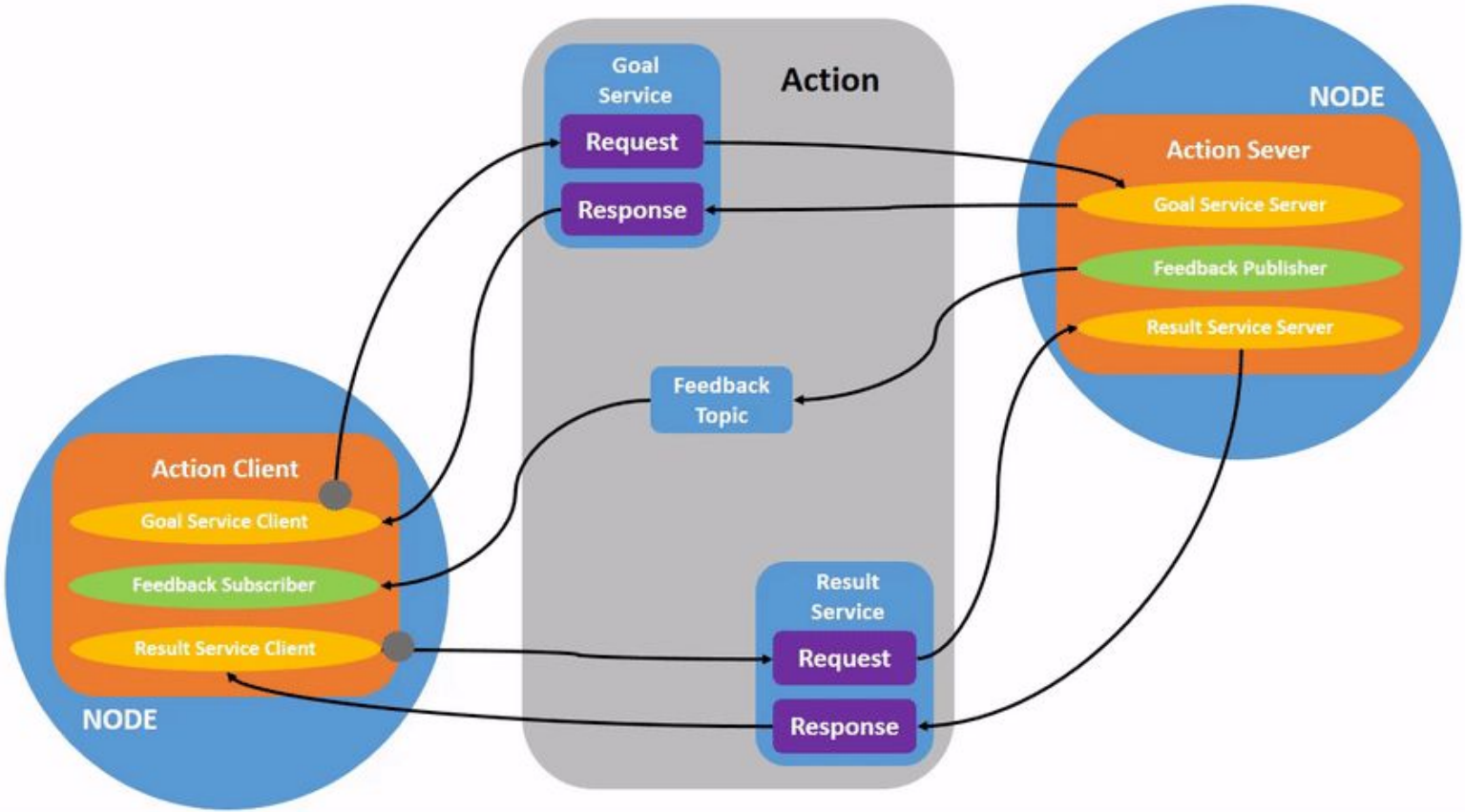
An action server **provides an action having a name and a type** and It is responsible for:

1. advertising the action to other ROS entities
2. accepting or rejecting goals from one or more action clients
3. executing the action when a goal is received and accepted
4. optionally providing feedback about the progress of all executing actions
5. optionally handling requests to cancel one or more actions
6. sending the result of a completed action, including whether it succeeded, failed, or was canceled, to a client that makes a result request.

Action Client

An action client sends one or more goals (an action to be performed) and monitors their progress. It is responsible for:

1. sending goals to the action server
2. optionally monitoring the user-defined feedback for goals from the action server
3. optionally monitoring the current state of accepted goals from the action server
4. optionally requesting that the action server cancel an active goal
5. optionally checking the result for a goal received from the action serve


Goal
Service
Request
Response
Action
NODE
Action Sever
Goal Service Server
Feedback Publisher
Result Service Server
Feedback
Topic
Action Client
Goal Service Client
Feedback Subscriber
Result Service Client
NODE
Result
Service
Request
Response

Parameters

How to specify information when starting robot nodes?

ROS parameter is what you might expect: a named, typed, place to store a piece of data

When it starts up, the node would use the value of that parameter to know which device to open to get to the motor system

Parameters

Ros parameters can be set as follows:

1. **Defaults**: A ROS node that uses a parameter must embed in its code some default value for that parameter
2. **Command-line**: There is standard syntax for setting parameter values on the command-line when launching a node
3. **Launch files**: When launching nodes via the launch tool instead of manually via the command-line, you can set parameter values in the launch file
4. **Service calls**: ROS parameters are dynamically reconfigurable via a standard ROS service interface, allowing them to be changed on the fly, if the node hosting the parameters allows it

Callbacks

When subscribing to a **topic**, you supply a **callback** →

function that will be invoked each time your node receives a message on that topic

Similar also in services, actions and parameters

Why? ROS is an event-based pattern

ROS node structure

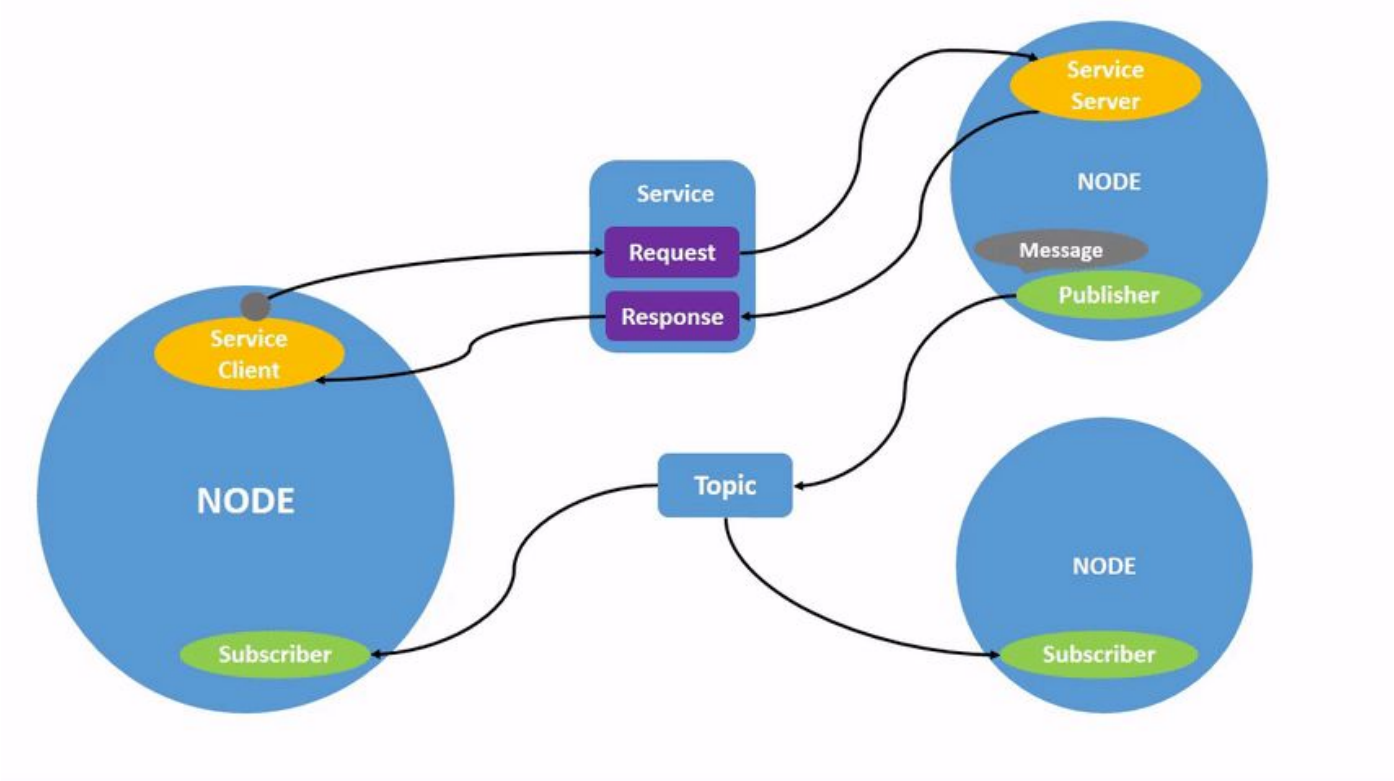
- **Get parameter values**: Retrieve the node's configuration, considering defaults and what may have been passed in from outside.
- **Configure**: Do whatever is necessary to configure the node, like establish connections to hardware devices.
- S**et up ROS interfaces**: Advertise topics, services, and/or actions, and subscribe to services. Each of these steps supplies a callback function that is registered by ROS for later invocation.
- **Spin**: Now that everything is configured and ready to go, hand control over to ROS. As messages flow in and out, ROS will invoke the callbacks you registered

ROS Nodes

A node is a participant using a client library to communicate with other nodes. Nodes can communicate with other nodes within the same process, in a different process, or on a different machine

Nodes are often a complex combination of publishers, subscribers, service servers, service clients, action servers, and action clients, all at the same time.

Connections between nodes are established through a **distributed discovery process**



Source:
https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html

ROS Discovery

ROS discovery can be summarized as follows:

1. When a **node** is started, it **advertises** its presence to other nodes on the network with the same ROS domain (ROS_DOMAIN_ID environment variable). **Nodes respond** to this advertisement with information about themselves
2. **Nodes periodically advertise their presence** so that connections can be made with new-found entities, even after the initial discovery period.
3. **Nodes advertise to other nodes when they go offline**

Nodes will only establish connections with other nodes if they have compatible Quality of Service settings

ROS QoS

ROS provides native support for few DDS QoS via a configuration struct when creating Publishers, Subscribers, etc.

1. History
2. Depth
3. Reliability
4. Durability

ROS QoS policies

The base QoS profile currently includes settings for the following policies:

1. History
   a. **Keep last**: only store up to N samples, configurable via the queue depth option
   b. **Keep all**: store all samples, subject to the configured resource limits of the underlying middleware
2. Depth
   a. **Queue size**: only honored if the “history” policy was set to “keep last”
3. Reliability
   a. **Best effort**: attempt to deliver samples, but may lose them if the network is not robust
   b. **Reliable**: guarantee that samples are delivered, may retry multiple times

ROS QoS policies

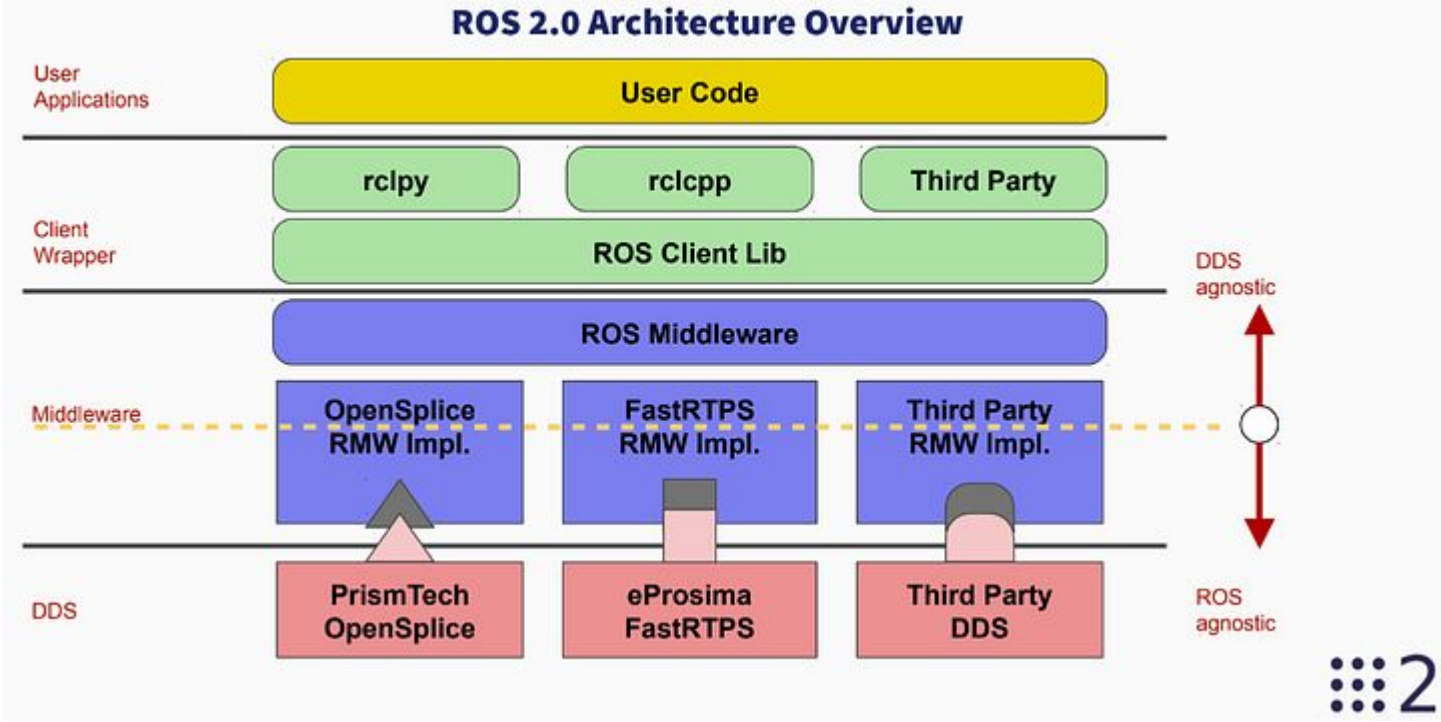
1. Durability
   a. **Transient local**: the publisher becomes responsible for persisting samples for “late-joining” subscriptions
   b. **Volatile**: no attempt is made to persist samples
2. Deadline
   a. **Duration**: the expected maximum amount of time between subsequent messages being published to a topic
3. Lifespan
   a. **Duration**: the maximum amount of time between the publishing and the reception of a message without the message being considered stale or expired (expired messages are silently dropped and are effectively never received)
4. Liveliness
   a. **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
   b. **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)
5. Lease Duration
   a. **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

QoS profile defines a set of policies that are expected to go well together for a particular use case → no worries about possible settings

1. **Default QoS settings for publishers and subscriptions**: by default, pubs and subs have “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”
2. **Services**: volatile durability, as otherwise service servers that re-start may receive outdated requests. The server is not protected from side-effects of receiving the outdated requests.
3. **Sensor data**: latest samples as soon as they are captured, at the expense of maybe losing some. Sensor data profile uses best effort reliability and a smaller queue size
4. **Parameters**: use a larger queue depth so that requests do not get lost when, for example, the parameter client is unable to reach the parameter service server
5. **System default**: RMW implementation’s default values for all of the policies. Different RMW implementations may have different defaults

ROS Architecture



Exercise

Design an architecture for a self-driving car

1. requirements
2. components
3. missions

An example

Requirements:

1. Real-time performance
2. Reliability
3. Adaptability and Interoperability
4. Flexibility and Variability
5. Re-Usability and Extensibility
6. Open community

Some missions:

1. looking for parking space
2. drive vehicle in the parking space
3. secure vehicle

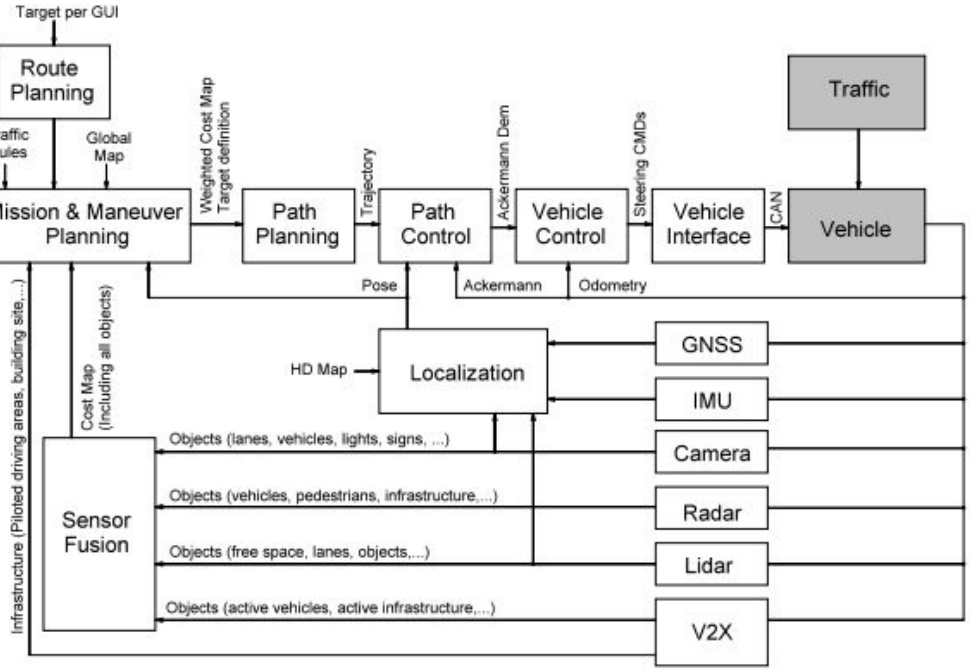


Image from: *Reke, Michael, et al. "A self-driving car architecture in ROS2." 2020 International SAUPEC/RobMech/PRASA Conference. IEEE, 2020.*

An example

Fig. 2. One of the KIA Niros, which has been automated at FH Aachen.

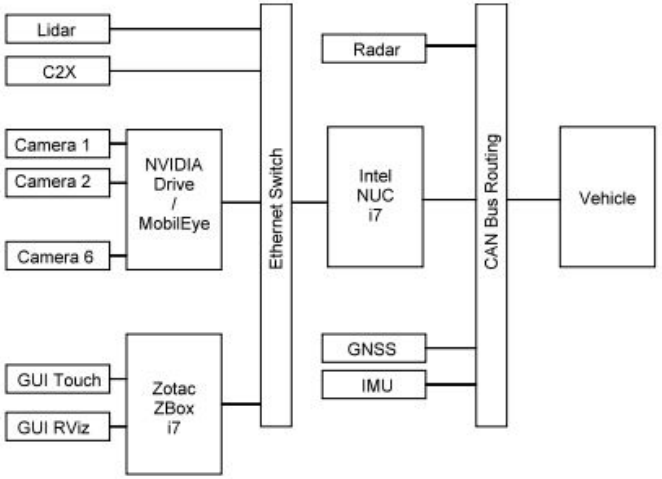


Image from: *Reke, Michael, et al. "A self-driving car architecture in ROS2." 2020 International SAUPEC/RobMech/PRASA Conference. IEEE, 2020.*