Robot operációs rendszerek és fejlesztői ökoszisztémák

Modern Robotirányítás ROS 2 alapokon

Svastits Áron

2025. október 20.

KUKA

MŰEGYETEM 1782

iit

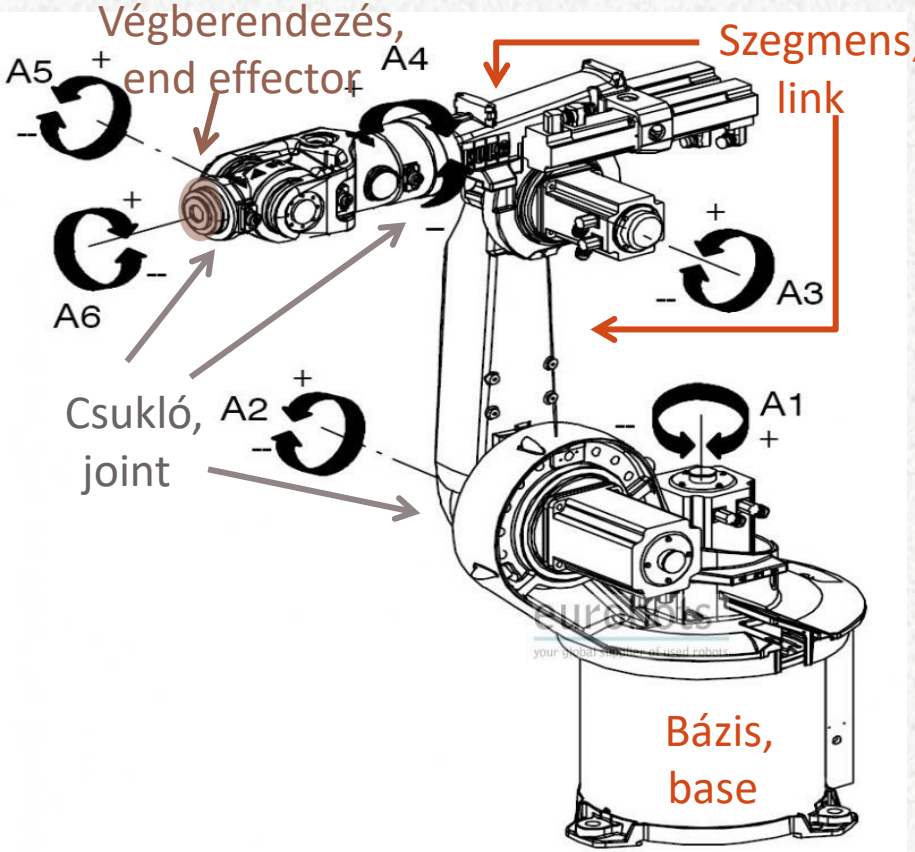
Visszaemlékezés…

Principles of SoA

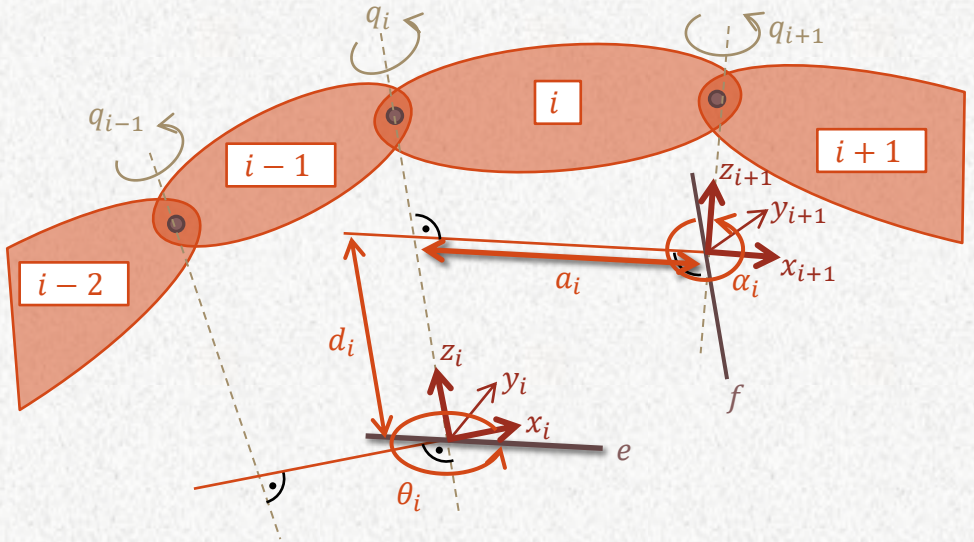
According to Amazon, the basic principles of SoA are:

- Interoperability
  - Any client system can run the service, interact with it
  - Functionalities and capabilities are well defined
- Loose coupling
  - Stateless services
  - Little to no dependency to external resources
- Abstraction
  - Consumers don't need the knowledge of implementation details -> black box
- Granularity
  - One discrete business function per service

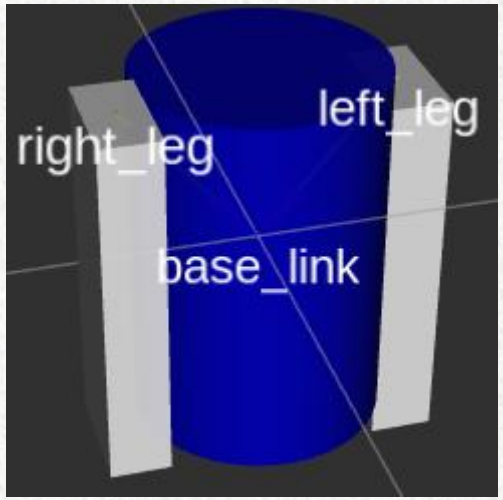
Robotics basics



Modeling with DH parameters



URDF - Example




```xml
<?xml version="1.0"?>
<robot name="materials">

  <material name="blue">
    <color rgba="0 0 0.8 1"/>
  </material>

  <material name="white">
    <color rgba="1 1 1 1"/>
  </material>


  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
      <material name="blue"/>
    </visual>
  </link>

  <link name="right_leg">
    <visual>
      <geometry>
        <box size="0.6 0.1 0.2"/>
      </geometry>
      <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
      <material name="white"/>
    </visual>
  </link>

  <joint name="base_to_right_leg" type="fixed">
    <parent link="base_link"/>
    <child link="right_leg"/>
    <origin xyz="0 -0.22 0.25"/>
  </joint>

  <link name="left_leg">
    <visual>
      <geometry>
        <box size="0.6 0.1 0.2"/>
      </geometry>
      <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
      <material name="white"/>
    </visual>
  </link>

  <joint name="base_to_left_leg" type="fixed">
    <parent link="base_link"/>
    <child link="left_leg"/>
    <origin xyz="0 0.22 0.25"/>
  </joint>

</robot>
```

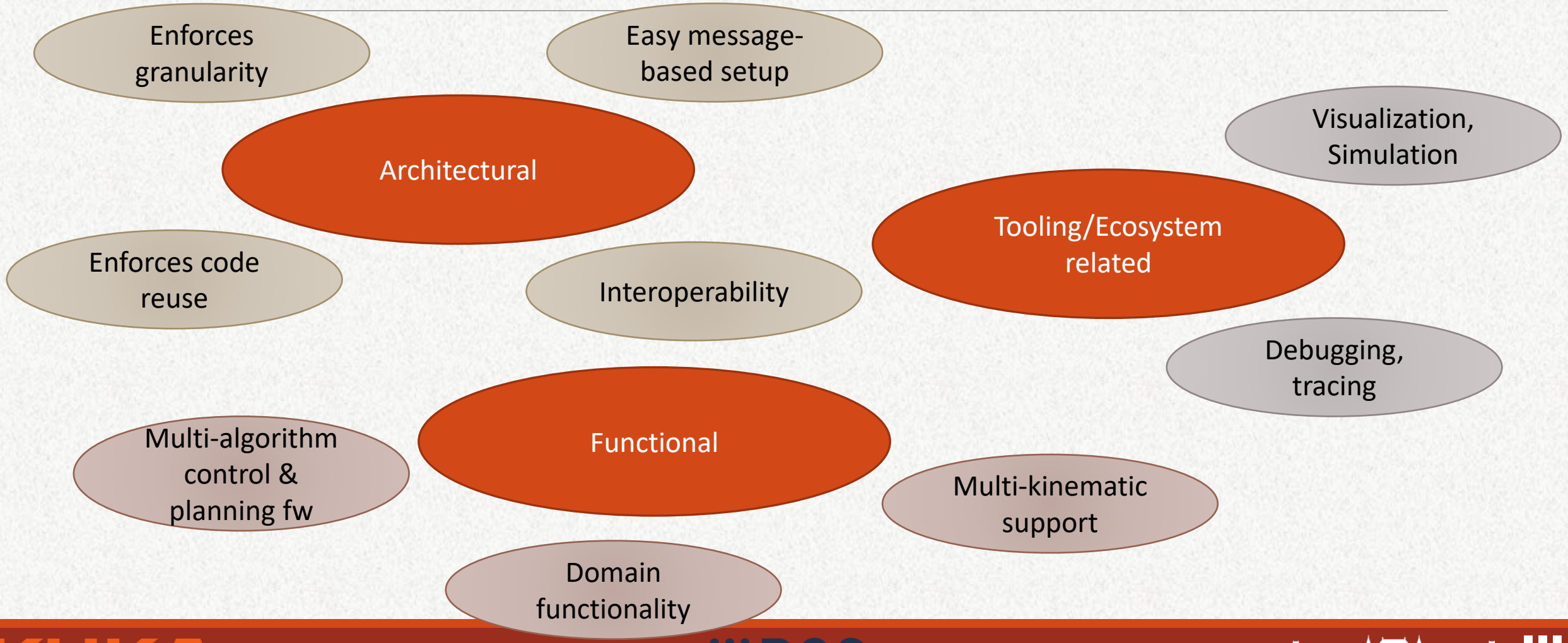
`urdf_tutorial/urdf/04-materials.urdf`

Road to improvement

ROS („1”)

ROS Groovy Galapagos	December 31, 2012			ROS Noetic Ninjemys	May 23rd, 2020			May, 2025 (Focal EOL)
ROS Fuerte Turtle	April 23, 2012			ROS Melodic Morenia	May 23rd, 2018			June 27, 2023 (Bionic EOL)
ROS Electric Emys	August 30, 2011			ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Diamondback	March 2, 2011			ROS Kinetic Kame	May 23rd, 2016			April, 2021 (Xenial EOL)
ROS C Turtle	August 2, 2010			ROS Indigo Igloo	July 22nd, 2014			April, 2019 (Trusty EOL)
ROS Box Turtle	March 2, 2010			ROS Hydro Medusa	September 4th, 2013			May, 2015

ROS Advantages



ROS Disadvantages

- Highly coupled to (deprecated) technologies
  - Python2
  - C++03
  - Cmake (even for python packages, a custom cmake script is run)
  - Ubuntu (others community supported only)
- Proprietary in-system communication
  - Custom message serialization
  - Custom transport protocol
  - Custom discovery mechanism
- No realtime capability
  - Thus, lack of deterministic control
  - Communication based on TCP by default

ROS Disadvantages

- Rosmaster: central DNS (*Domain Name Server*) for all communications, to which nodes connect for discovery
  - Needs to be run first before all nodes
  - Central parameter server (also run by rosmaster)
  - Lack of capability to act as a distributed system
    - Different functionalities require different resources
- Not compatible with safety-critical applications
  - No realtime, no fault-tolerance mechanisms

An Ideal Robot OS vs ROS

- Scalable in the resources & capabilities dimension 😐
- Scalable in the machine portfolio dimension 🙂
- Offers standardized capabilities & interfaces 😐
- Enables deterministic programming ☹️
- Easily testable 🙂
- Cloud & simulation support 😐

ROS 2

Key design principles

Realtime computing — Need to be able to perform computation in realtime reliably since runtime efficiency is crucial in robotics

Diverse networks — Need to be able to run and communicate across vast networks from LAN to multi-satellite hops to accommodate the variety of environments where robots could operate and need to communicate.

Embedded Systems — ROS2 aims to support a wider range of platforms and use cases, including embedded systems

Cross-platform support— ROS 2 should be more portable and run on various operating systems

Security — It needs to be safe with proper encryption where needed

Do not change what was good in ROS1, but backwards compatibility is not required

Technologies

- Targets C++17, heavily uses C++11
- Python 3
- Build system
  - Tools (Usually Cmake for C++, and setuptools for Python)
  - Meta-build tool: colcon
    - Depedency management: topologically order a group of packages, and build or test them in the correct dependency order
    - Able to build packags in parallel
  - Helper: Ament (So that metainformation is handled correctly)
    - Linting, Static code check, copyright check
- Officially not coupled to Ubuntu only
- Support for microcontrollers (micro-ROS)

Supported Platforms

Jazzy Jalisco is primarily supported on the following platforms:

Tier 1 platforms:

- Ubuntu 24.04 (Noble): `amd64` and `arm64`
- Windows 10 (Visual Studio 2019): `amd64`

Tier 2 platforms:

- RHEL 9: `amd64`

Tier 3 platforms:

- macOS: `amd64`
- Debian Bookworm: `amd64`

Nodes

- In ROS 1, there were Nodes and Nodelets
  - Node: executable, using the ROS client library for communication
  - Nodelet: shared libraries (derived from *nodelet::Nodelet*)
- In ROS 2, things got simplified…
  - Derived from ***rclcpp::Node***
  - Which follows the concept of Nodelets with more user-friendly interfaces
  - Using ROS2 nodes, it is possible to…
    - Run multiple nodes in separate processes with the benefits of process/fault isolation as well as easier debugging of individual nodes and
    - Run multiple nodes in a single process with the lower overhead and optionally more efficient communication
- Single-process execution allows intra-process publish/subscribe connections resulting in zero-copy transport of messages when publishing and subscribing

Example: publisher (ROS 1)

```cpp
int main(int argc, char ** argv) {
    ros::init(argc, argv, "talker");

    ros::NodeHandle n;
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
    ros::Rate loop_rate(10);
    int count = 0;

    while (ros::ok()) {
        std_msgs::String msg;
        std::stringstream ss;
        ss << "hello world " << count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());

        chatter_pub.publish(msg);

        ros::spinOnce();
        loop_rate.sleep();
        ++count;
    }
    return 0;
}
```

Example: publisher (ROS 2)

```
class MinimalPublisher : public rclcpp::Node
{
  public:
    MinimalPublisher() : Node("minimal_publisher"), count_(0)
    {
        publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
        timer_ = this->create_wall_timer(500ms,
        std::bind(&MinimalPublisher::timer_callback, this));
    }

  private:
    void timer_callback()
    {
        auto message = std_msgs::msg::String();
        message.data = "Hello, world! " + std::to_string(count_++);
        RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
        publisher_->publish(message);
    }
    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
    size_t count_;
};

int main(int argc, char * argv[])
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<MinimalPublisher>());
    rclcpp::shutdown();
    return 0;
}
```

Example: intra-process communication

```cpp
int main(int argc, char * argv[])
{
    rclcpp::init(argc, argv);
    rclcpp::executors::SingleThreadedExecutor executor;

    rclcpp::NodeOptions options;
    // Enable intra-process communication
    options.use_intra_process_comms(true);

    auto producer = std::make_shared<Producer>("producer", options);
    auto consumer = std::make_shared<Consumer>("consumer", options);

    executor.add_node(producer);
    executor.add_node(consumer);
    executor.spin();

    rclcpp::shutdown();

    return 0;
}
```

Launch files

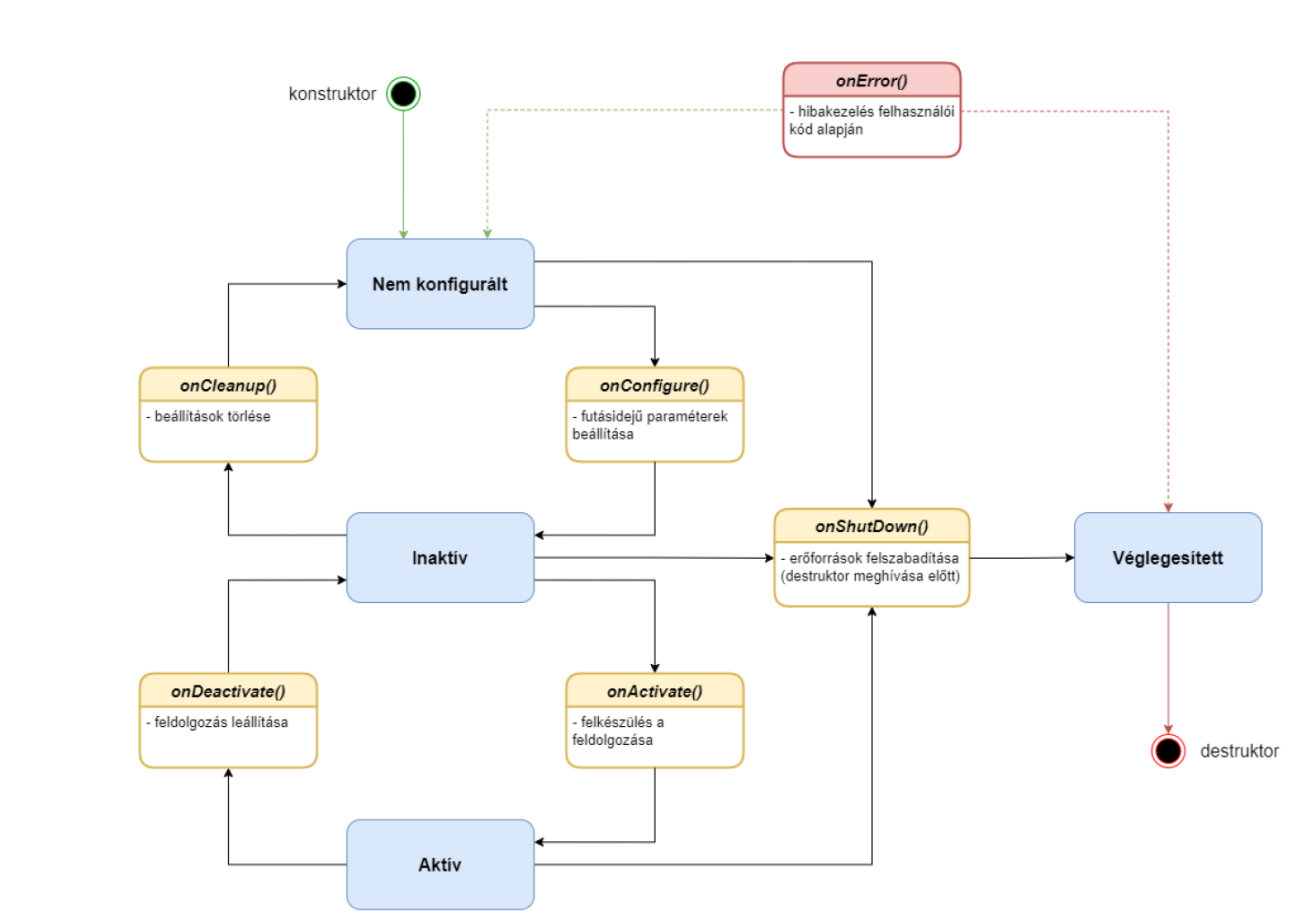
- **Flexibility**: ROS2 launch files written in Python can include conditional logic, loops, etc., making them more flexible than ROS1's static XML files

- **New features**
  - Event handlers: user-defined reactions for launch-related events (e.g. *OnShutDown*)
  - Actions (execute process, set environment variable, create log entry, timer action, …)

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            namespace='turtlesim1',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='turtlesim',
            namespace='turtlesim2',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='turtlesim',
            executable='mimic',
            name='mimic',
            remappings=[
                ('/input/pose', '/turtlesim1/turtle1/pose'),
                ('/output/cmd_vel', '/turtlesim2/turtle1/cmd_vel'),
            ]
        )
    ])
```

Managed nodes



Going deeper

Messaging middleware

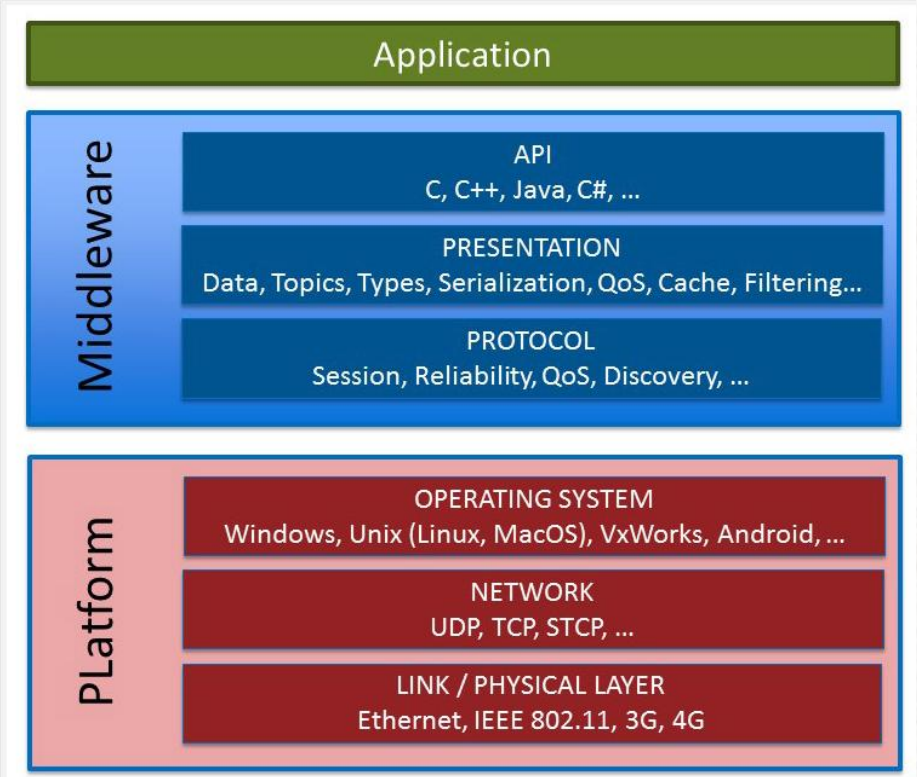
- To avoid getting to the same dead-end, in-house communication was decoupled to a **messaging middleware**, called DDS

- What is a messaging middleware?
  - Messaging middleware, also known as message-oriented middleware (MOM), is
    - a software or hardware infrastructure…
    - …that facilitates the sending and receiving of messages between distributed systems.
    - It acts as an intermediary layer that allows different applications, often running on different platforms, to communicate with each other in a reliable and scalable manner.
  - Should have the ability to
    - Buffer messages
    - Transform messages
    - Perform asynchronous communication

In-system communication: DDS

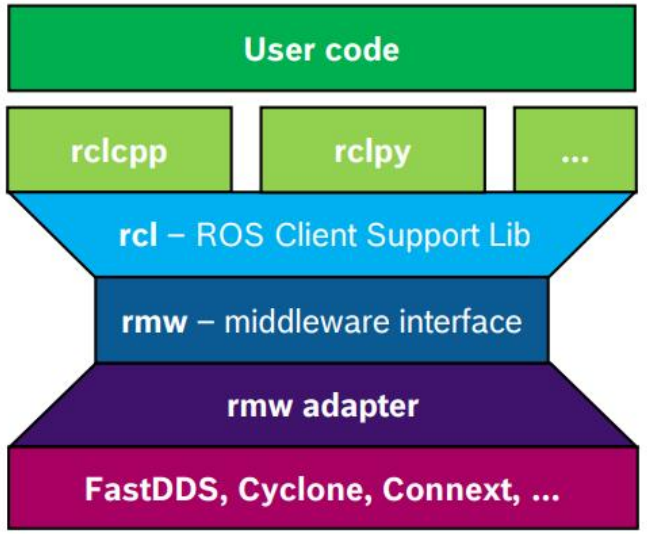
- DDS is an industry standard protocol specifying API and behaviour
  - Provides low-latency, reliable and scalable architecture
  - RTPS (a.k.a. DDSI-RTPS) is the wire protocol used by DDS to communicate over the network.
  - it is implemented by a range of vendors, such as RTI's Connext DDS, eProsima's Fast DDS, Eclipse's Cyclone DDS, or GurumNetworks's GurumDDS.

- Key requirements to consider when choosing DDS implementation
  - Licensing
  - Platform and language availability
  - Computation footprint

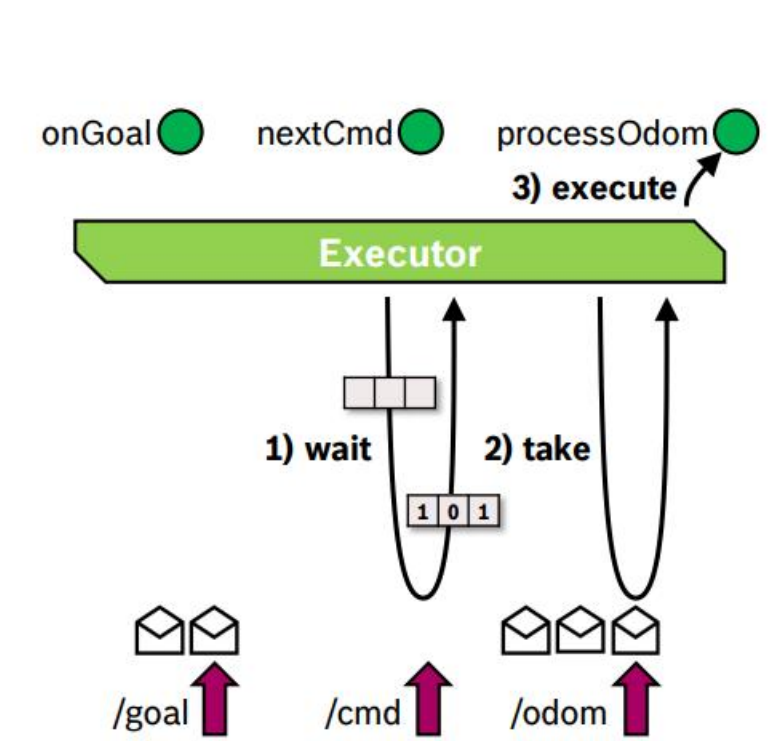


DDS in ROS 2

- ROS 2 aims to support multiple DDS implementations - despite the fact that each of them differ slightly in their exact API
  - an abstract interface - **R**OS **m**iddle**w**are interface (rmw) - is introduced …
  - … which defines the API between the ROS client library and any specific implementation.
  - The interface can be implemented for different DDS implementations resulting in the DDS adapters
- For many cases you will find that nodes using different RMW implementations are able to communicate, however this is not true under all circumstances.
- In ROS 2, the messaging middleware is responsible for:
  - discovery,
  - publish and subscribe mechanism,
  - request-reply mechanism for services,
  - serialization of message types

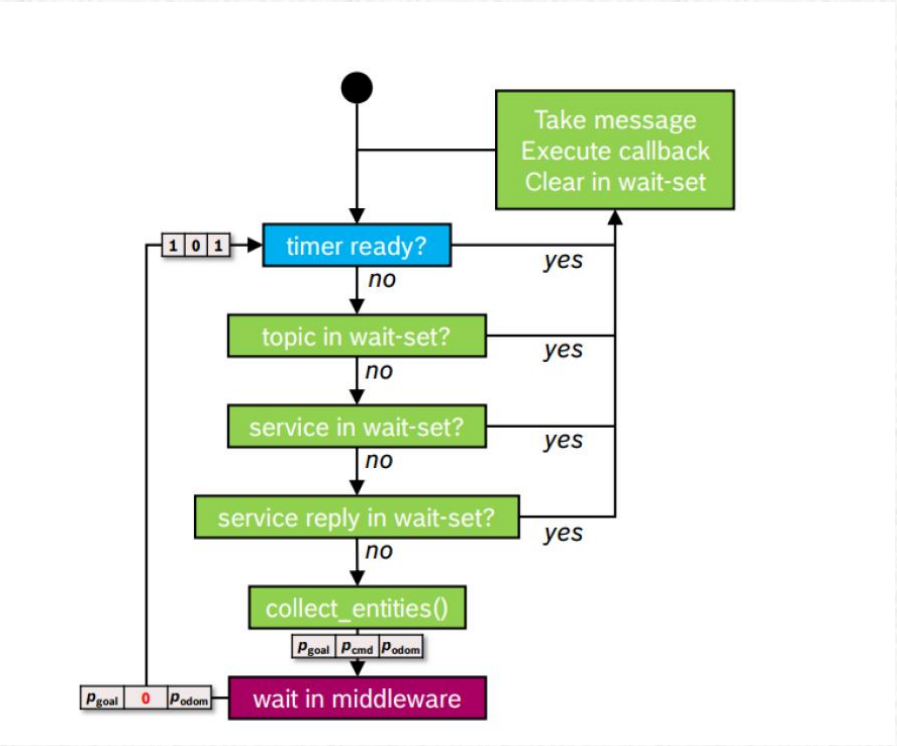


Event processing



- ROS 1: common queue for all messages, FIFO processing
- ROS 2: messages are queued in middleware layer separately for every topic
  - Executor has to poll middleware whether a message has been received
    - binary output for every topic
  - Executor takes message based on client library logic and runs appropriate callback

Event processing

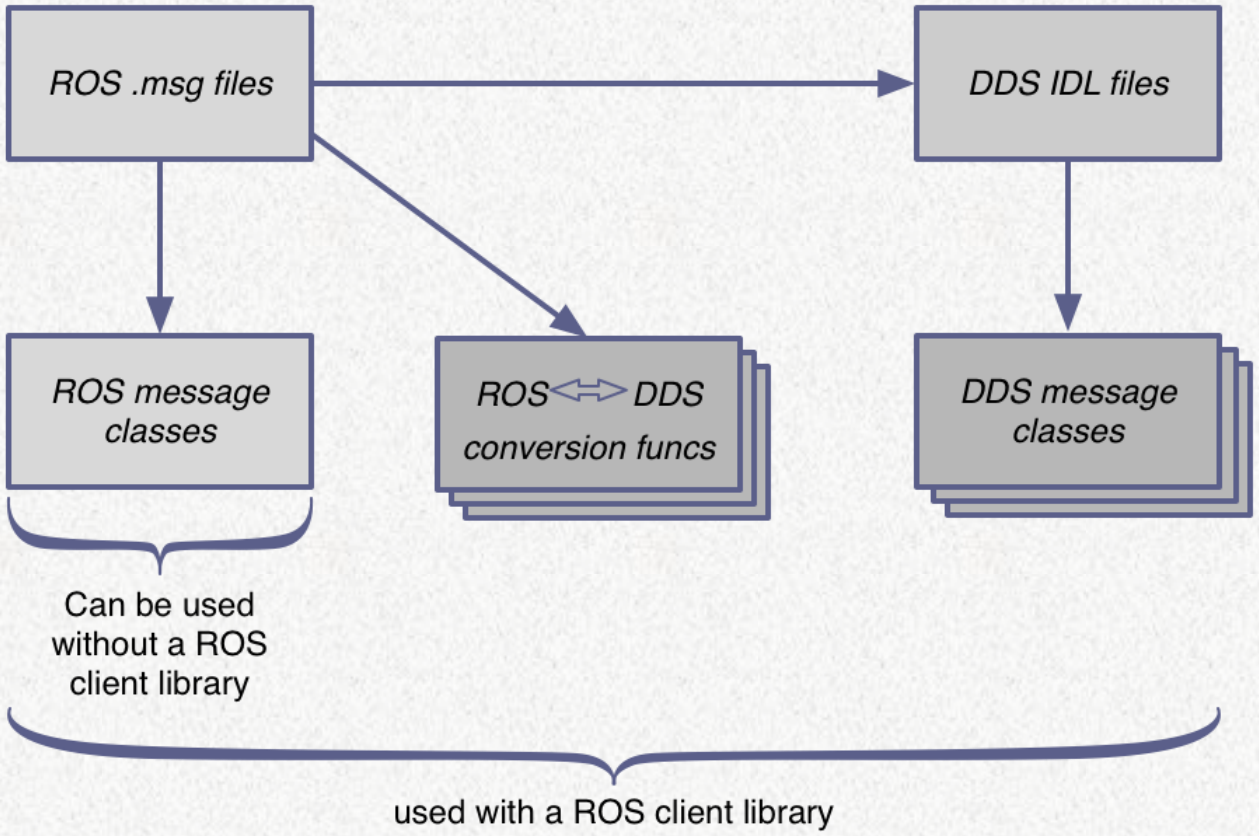


- Request available messages from middleware for specific node
- „timer ready” comes from client library
- „Non-preemptive priority + round-robin”
  - Timers have higher priority but callbacks cannot be preempted
  - Only one message from a channel can be processed in one cycle
- Multi-threaded executor is also supported, where more commands can be processed at the same time
  - Callback groups to avoid concurrency issues
  - By default all callbacks are mutually exclusive

Internal Interfaces

- **rmw**
  - The minimal set of primitive middleware capabilities needed to build ROS on top. Providers of different middleware (DDS) implementations must implement this interface in order to support the entire ROS stack on top.
- **rcl**
  - Used by client libraries (rclc, rclcpp, rclpy, etc.) to be smaller and more consistent with each other by avoiding the duplication of functions and logic. However, some language idiomatic methods are not included (like different thread implementations of languages).
- **rosidl**
  - Consists of a few message related static functions and types along with a definition of what code should be generated by messages in different languages. The generated message code specified in the API will be language specific and will contain things like the message data structure, functions for construction, destruction, etc.

rosidl



- Design decision: ROS 2 should preserve the ROS 1 message definitions and in-memory representation
  - Conversion needed to DDS IDL
- Language specific types generated for both ROS2 and DDS
  - Conversion methods needed between message structs

QoS – Quality of Service

- The description or measurement of the overall performance of a service
- To quantitatively measure quality of service, several related aspects of the network service are often considered
  - Packet loss
  - Packet delay
  - Throughput
  - Bitrate
  - Jitter
  - Availability
  - …

QoS for ROS 2 communication

- To be able to connect to different layers of the system
  - Communication constraints must be set and supervised
- With the right set of QoS policies, ROS 2 communication can be
  - as reliable as TCP
  - as best-effort as UDP
  - or somewhere between
- In ROS 2, there are QoS profiles available, made up of QoS policies
  - Presets available
  - Customs could be made
- Profile compatibility is checked for each policy
  - Subs: requested *min* quality, Pubs: offered *max* quality

Publisher	Subscription	Compatible
Best effort	Best effort	Yes
Best effort	Reliable	No
Reliable	Best effort	Yes
Reliable	Reliable	Yes

QoS for ROS 2 communication

- QoS policies:
  - History (keep all/ keep last *N*)
  - Depth (queue size for keep last)
  - Reliability (reliable or best effort)
  - Durability (handling of late-joiner subscriptions)
  - Deadline (max time between subsequent messages)
  - Lifespan (time for which a message is considered valid)
  - …

  - QoS anomalies can be detected via QoS events (available via callbacks)
    - e.g. publisher missed deadlie specified

Default QoS settings

- **Publishers and subscriptions**
  - In order to make the transition from ROS 1 to ROS 2 easier, exercising a similar network behavior is desirable. By default, publishers and subscriptions in ROS 2 have **keep last** for history with a queue size of 10, **reliable** for reliability**,** and **volatile** for durability.
- **Services**
  - In the same vein as publishers and subscriptions, services are **reliable**. It is especially important for services to use **volatile** durability, as otherwise service servers that re-start may receive outdated requests. While the client is protected from receiving multiple responses, the server is not protected from side-effects of receiving the outdated requests.
- **Sensor data**
  - For sensor data, in most cases it's more important to receive readings in a timely fashion, rather than ensuring that all of them arrive. That is, developers want the latest samples as soon as they are captured, at the expense of maybe losing some. For that reason the sensor data profile uses **best effort** reliability and a smaller queue size.
- **Parameters**
  - Parameters in ROS 2 are based on services, and as such have a similar profile. The difference is that parameters use a much larger queue depth so that requests do not get lost when, for example, the parameter client is unable to reach the parameter service server.

Security

- Standards updated in 2025
  - ISO 10218: **cybersecurity is integral to safety** in modern robotic systems
  - IEC 62443: cybersecurity requirements

- Defined by the underlying Middleware (DDS)
  - Support for security plugin needed, defined in DDS Security Specification
- ROS2 has the ability to secure communications among nodes within the ROS 2 computational graph
  - No additional software needed, just configuration files for each communication participant
- Configuration enables **encryption** and **authentication**
  - policies can be defined both for **individual** nodes and for the **overall** ROS graph
  - Ensures the integrity of data being sent, and enables domain-wide access controls
- Security Enclaves can be set for different subsystems with different policies

Key Takeaways

- Realtime capabilities & subsystems
- Security concept
- Distributed system capabilities
- Optimizable messaging
- Extended build tooling
- Modernized technologies & extended support for platforms

An Ideal Robot OS vs ROS2

- Scalable in the resources & capabilities dimension 
- Scalable in the machine portfolio dimension 
- Offers standardized capabilities & interfaces 
- Enables deterministic behaviour 
- Easily testable 
- Cloud & simulation support 


ALMOST
THERE

Is there something missing?

Standardized messages

Code quality of open-source components

Generic architectures for robotic apps are not defined

Documentation guidelines not set

No upcoming schedule for features

ROS-Industrial

ROS-Industrial

- ROS-Industrial is built up from 3 Consortiums (ROS-In Americas, Europe and Asia-Pacific)
- ROS-Industrial is an open-source project that extends the advanced capabilities of ROS software to industrial relevant hardware and applications
- Key mission is:
  - Managing a **roadmap** to identify and prioritize ROS-Industrial capabilities for industrial robotics and automation
  - Instituting and enforcing **code quality standards** appropriate for an industrial software product. These include rating/tracking code quality metrics, multi-level testing and documentation
  - Providing a wide range of user services, including **technical support and training**, to facilitate the continued adoption of ROS-Industrial by industry

ROS-Industrial Standardization

- ROS-In has selected the followings as standards for a robot control app:
  - ros2_control (hardware interface, controllers)
    - Out-of-the-box moveit compatibility
  - URDF (xacro) with strict naming conventions
    - Names for all resources are included
- ROS-In CI pipeline available
- Standardized messages defined
  - e.g. RobotStatus
- Find the one that conforms to these for KUKA robots at **kroshu**
  - https://github.com/kroshu
  - *Also, who uses the kroshu driver for the homework will have the possibility to try things out on real robots at KUKA Budapest!*

Questions? ☺