ROS2: Intestines & Advanced Features

Course: Robotic Programming Environments

Michał Drwięga
michal.drwiega@pwr.edu.pl
www.mdrwiega.com/edu/rpe

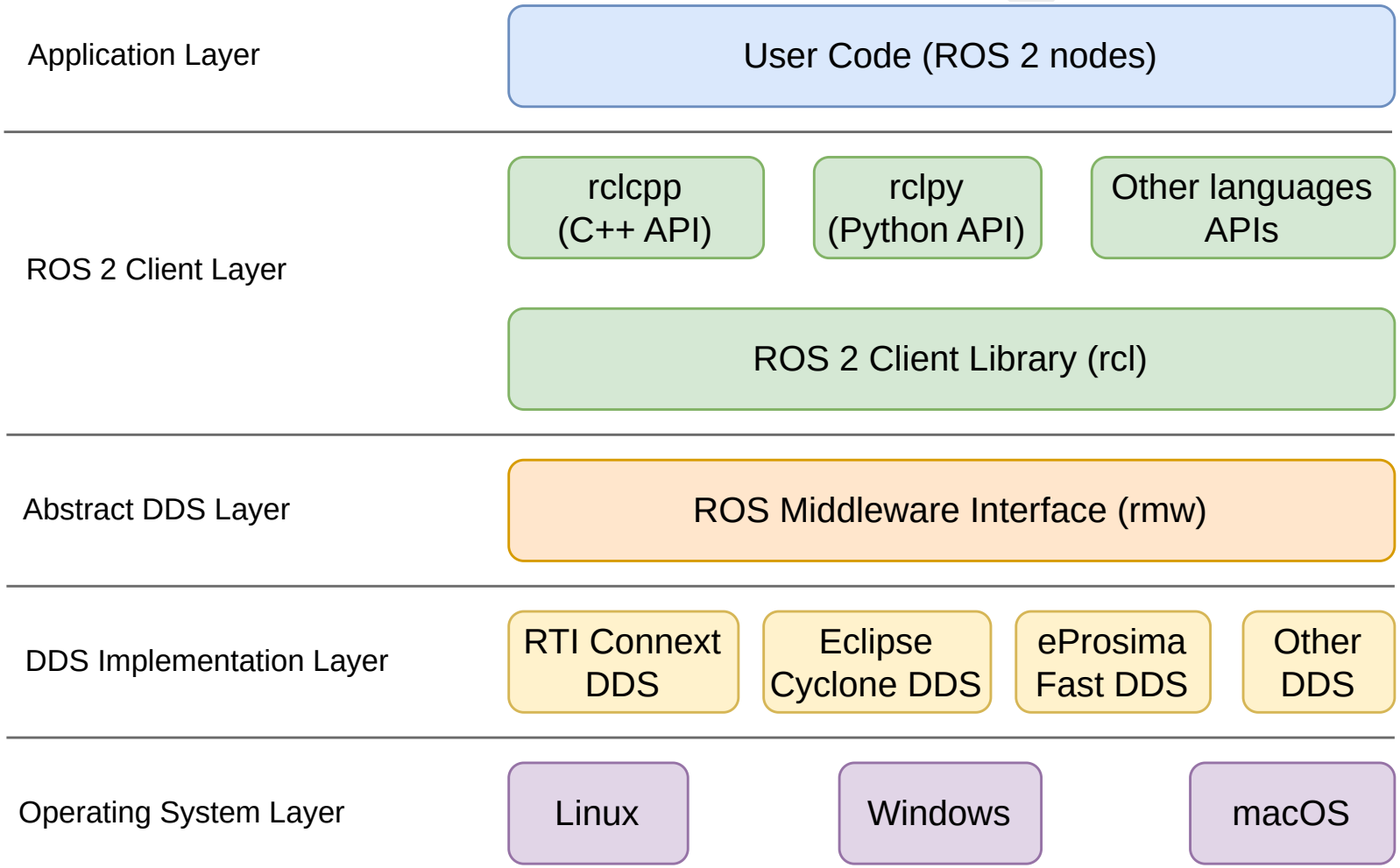
Department of Cybernetics and Robotics
Wrocław Univeristy of Science and Technology

Wrocław University
of Science and Technology

Content

ROS 2 Architecture

ROS 2: Layered Architecture



ROS Client Library (RCL)

The ROS Client Library (RCL) is a **language-agnostic API** in ROS 2 that provides the core functionality for creating nodes, publishers, subscribers, services, and timers.

Key Features:

- Node creation and management.
- Publishers and subscribers for message passing.
- Services and clients for request/response communication.
- Timers and callbacks for periodic tasks.
- Abstracts underlying middleware (RMW) from language-specific implementations.

Usage:

- Available in multiple languages through `rclpy` (Python), `rclcpp` (C++).

ROS Middleware (RMW)

The ROS Middleware (RMW) is an **abstraction layer** in ROS 2 that connects the ROS Client Library (RCL) with the underlying communication middleware (DDS, ROS 1 bridge, etc.).

Key Features:

- Abstracts underlying communication protocol from RCL.
- Handles message transport between nodes.
- Supports multiple middleware implementations (e.g., Fast DDS, Cyclone DDS, RTI Connext).
- Enables cross-language communication between nodes.

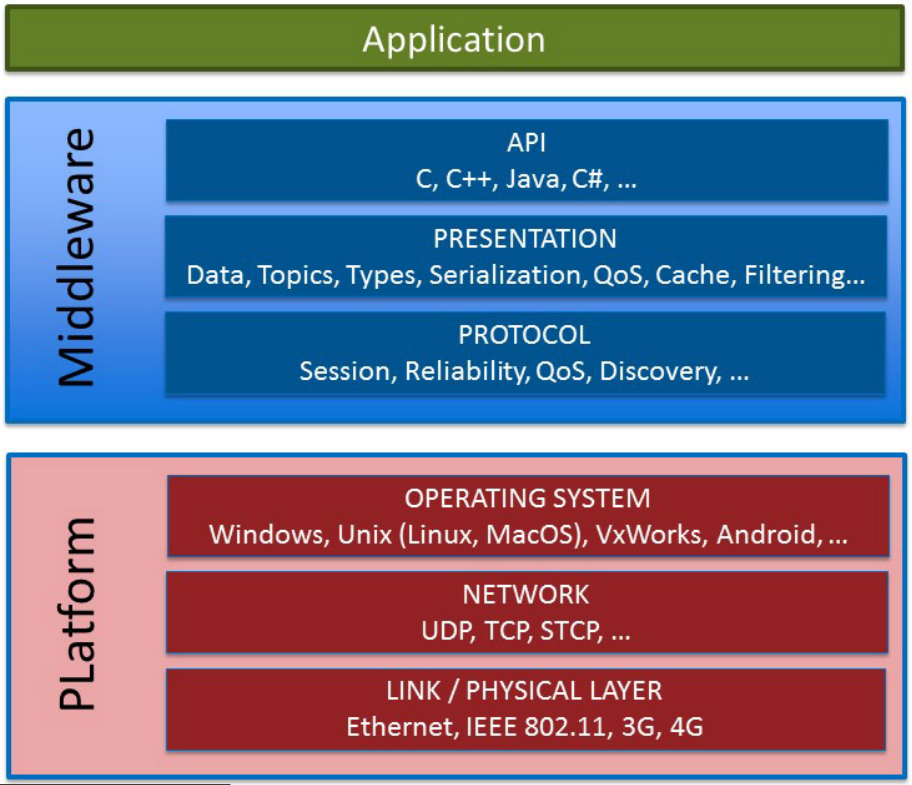
Usage:

- RCL calls are translated into RMW calls, which interface with DDS or other middlewares.
- Developers can switch middleware implementations without changing application code.

DDS

Data Distribution Service (DDS)

- **Middleware for real-time systems**
  - The DDS Middleware is a software layer that abstracts the Application from the details of the operating system, network transport, and low-level data formats.
  - High-performance, scalable communication layer
  - Used as the core middleware in ROS 2

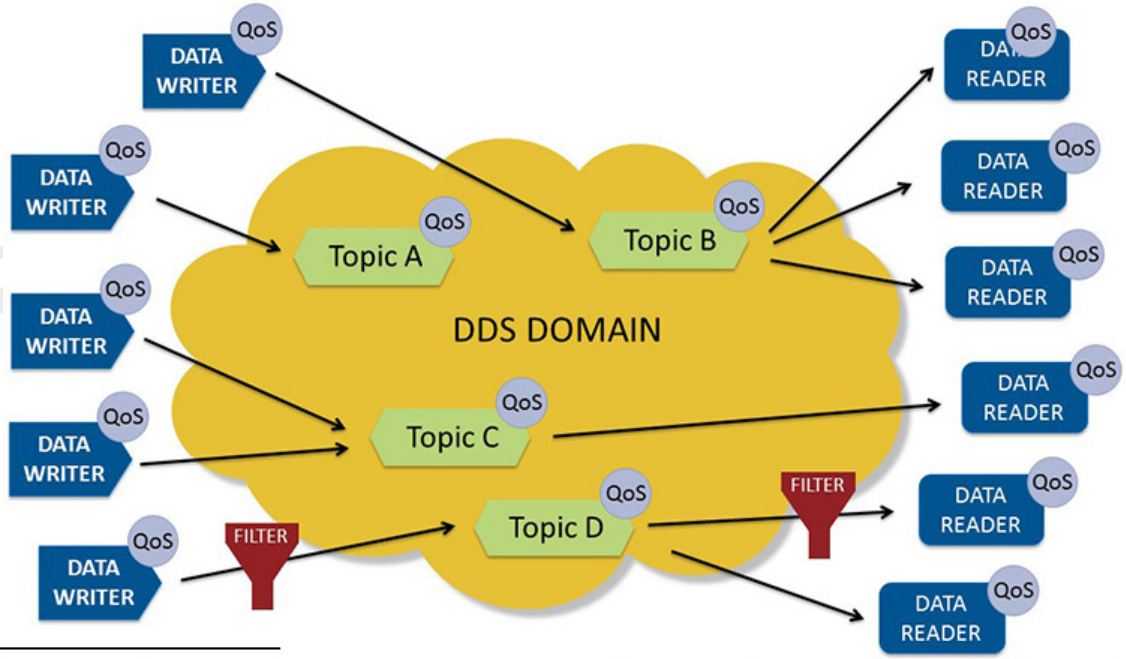


[1]

[1] https://www.dds-foundation.org/

Data Distribution Service (DDS)

- **Data definition**
  - Utilizes **Interface Description Language (IDL)** for defining message structures.
  - Enables strongly-typed message interfaces
- **Security and control**
  - Access control and data flow policies
  - Built-in encryption for secure communication
  - Supports flexible **Quality of Service (QoS)** for reliability, liveliness, and security.



Data Distribution Service (DDS)

- Implements a publish-subscribe communication pattern.
- DDS is an industry standard, implemented by vendors like RTI, eProsima, and Eclipse.[3]
- Provides dynamic discovery

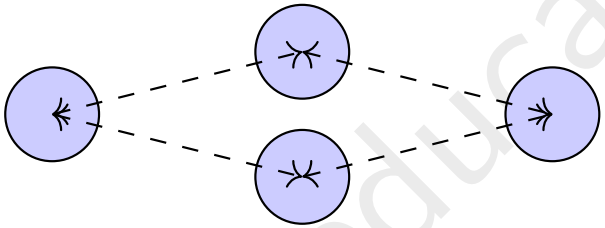
[3] https://docs.ros.org/

ROS 2 - DDS Discovery

What is Discovery?

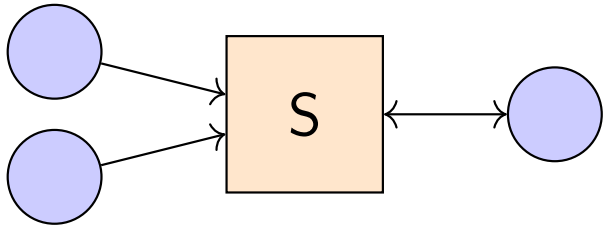
Process where ROS 2 nodes **automatically find each other** and establish communication without manual configuration. [4]

Simple Discovery (Default)



Peer-to-peer

Discovery Server (Alternative)

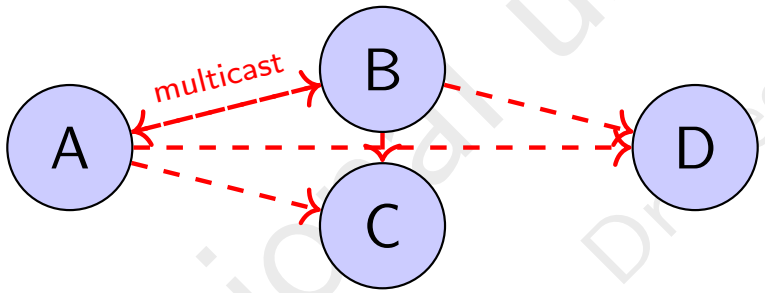


Client-server

[4] https://docs.ros.org/en/foxy/Tutorials/Advanced/Discovery-Server/Discovery-Server.html

ROS 2 - Standard DDS Discovery

The **Simple Discovery Protocol** (SDP) is the default mechanism defined in the DDS standard.



How it works:

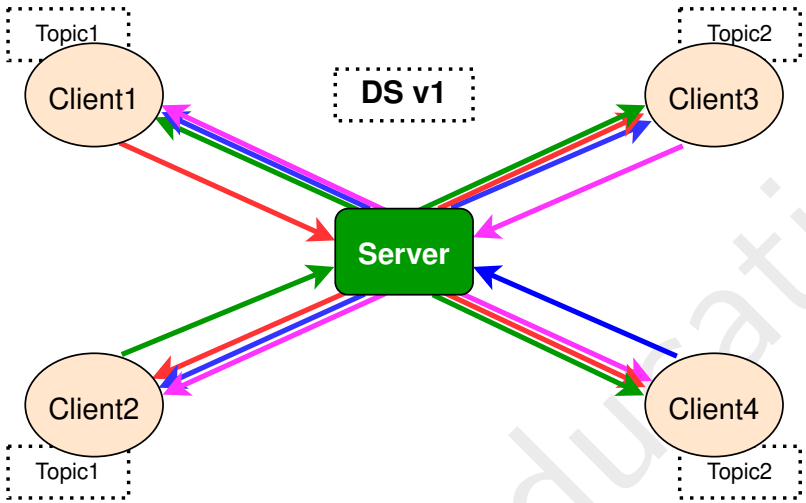
Participants send periodic multicast announcements
All nodes share discovery info with ALL other nodes

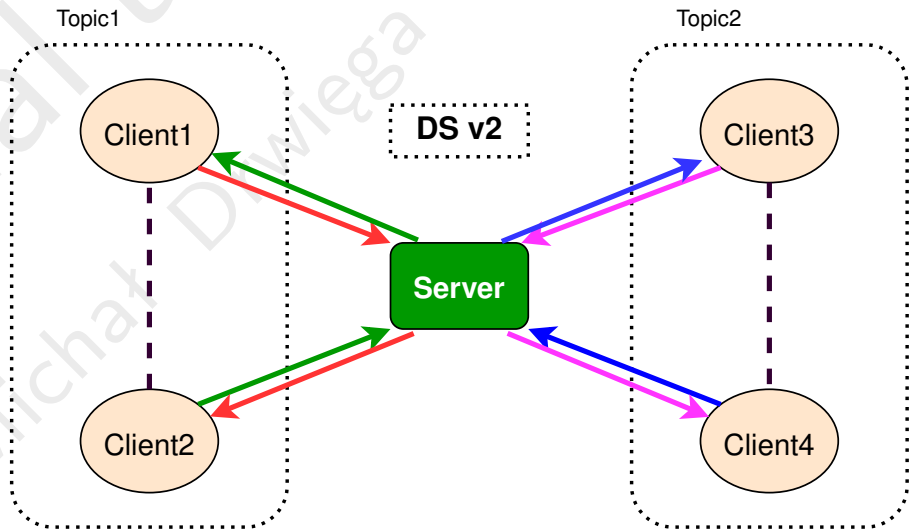
Limitations

- **Scaling issue:** Packet count increases *exponentially* with nodes
- **Multicast required:** May not work on WiFi, corporate/school networks

ROS 2 - Centralized Discovery Server

A **Discovery Server** (Fast DDS) acts as a rendezvous point where nodes register and discover each other.





- **Clients** connect to server(s)
- Share only **necessary** discovery info

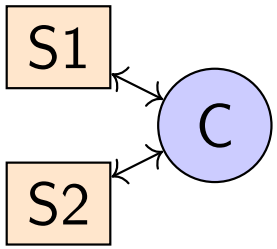
[5]

- No multicast required
- Works over WiFi, cloud, WAN

[5] https://docs.ros.org/en/foxy/Tutorials/Advanced/Discovery-Server/Discovery-Server.html

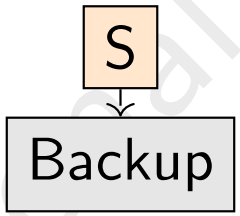
ROS 2 - Advanced Discovery Server

Redundancy



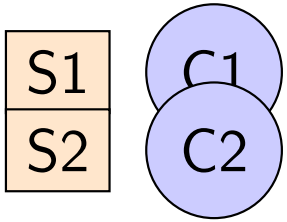
Connect to multiple servers for fault tolerance

Backup Server



Persists discovery database to file (`--backup`)

Partitioning



Virtual partitions via server selection

Discovery Server v2 Filtering

Uses **topic information** to decide if nodes need to discover each other, dramatically reducing network traffic [citation:4].

ROS 2 - Using Discovery Server

1. Start Discovery Server

```
# Terminal 1
fastdds discovery --server-id 0 \
  --ip-address 0.0.0.0 --port 11811
```

Default port: 11811

2. Configure Nodes

```
# Terminal 2 (listener)
export ROS_DISCOVERY_SERVER="127.0.0.1:11811
    "
ros2 run demo_nodes_cpp listener

# Terminal 3 (talker)
export ROS_DISCOVERY_SERVER="127.0.0.1:11811
    "
ros2 run demo_nodes_cpp talker
```

Multiple Servers (Redundancy)

```
# Start two servers
fastdds discovery --server-id 0 -p 11811
fastdds discovery --server-id 1 -p 11888

# Connect to both
export ROS_DISCOVERY_SERVER=\
"127.0.0.1:11811;127.0.0.1:11888"
```

If one server fails, discovery still works! [citation:4]

ROS 2 Daemon

For CLI tools (`ros2 topic list`) with Discovery Server:

```
export ROS_SUPER_CLIENT=True
```

Super client receives all discovery info

ROS 2 - DDS Security

Built-in Security

ROS 2 leverages DDS Security plugins to provide **authentication**, **encryption**, and **access control** without additional software.

Authentication
Verify participant identity
PKI-based (x.509 certs)

Access Control
Define permissions
Topic-level restrictions

Encryption
AES-GCM authenticated
Protect data in transit

ROS 2 - DDS Vendors Comparison

Feature	Fast DDS	Cyclone DDS	RTI Connext
Vendor	eProsima	Eclipse Foundation	RTI
License	Apache 2	Eclipse Public License v2.0	Commercial / Research
ROS 2 Status	Default RMW	Full support	Full support
RMW package	rmw_fastrtps_cpp	rmw_cyclonedds_cpp	rmw_connextdds

ROS 2 - DDS Performance Comparison

Performance characteristics vary by use case

Metric	Fast DDS	Cyclone DDS	RTI Connext
Latency (small messages)	Medium	**Lowest**	Low
Throughput (large data)	**High**	Medium	High
CPU utilization	Medium	**Lowest**	Higher
Memory footprint	Medium	**Smallest**	Largest
Discovery time (many nodes)	2s	**1s**	1.5s
Worst-case latency bound	Higher	Medium	**More predictable**

Fast DDS

Best for: Complex systems, feature-rich apps

Cyclone DDS

Best for: Real-time control, resource-constrained

RTI Connext

Best for: Mission-critical, enterprise

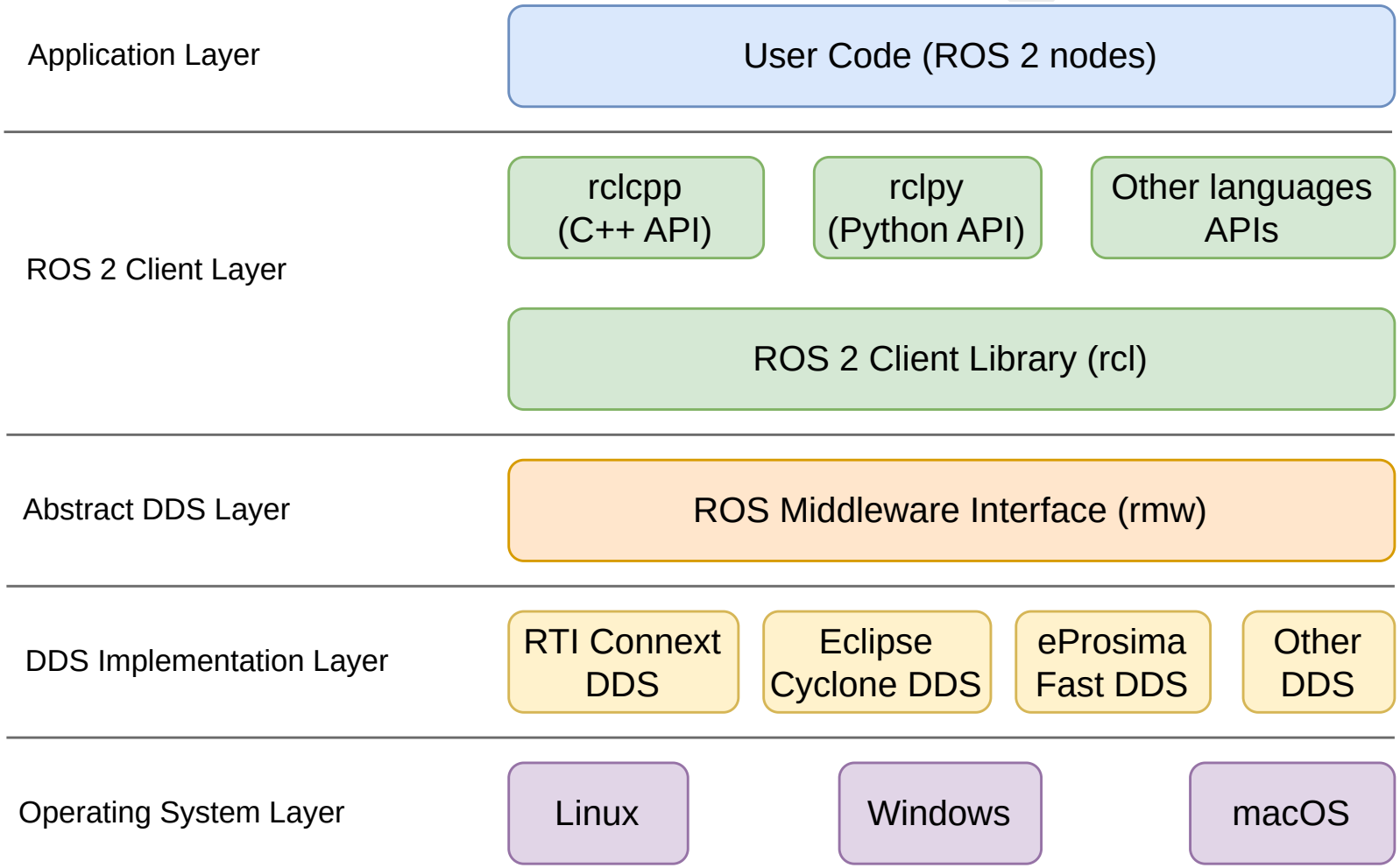
Limitations of DDS in ROS 2

- **Network Constraints**
  - Designed primarily for LAN environments
  - Issues with Wi-Fi, NAT traversal, and WAN/cloud communication
- **Discovery Overhead**
  - Heavy reliance on multicast-based discovery
  - Increased traffic and slower startup in large systems
- **Resource Usage**
  - Higher CPU and memory consumption
  - Less suitable for embedded or resource-constrained devices
- **Configuration Complexity**
  - Requires careful QoS tuning
  - Vendor-specific behavior (e.g., Fast DDS vs Cyclone DDS)
- **Debugging Difficulty**
  - Non-deterministic issues due to QoS and discovery
  - Hard to diagnose communication failures

Motivation: These challenges drive interest in alternative middleware.

Zenoh

ROS 2: Layered Architecture



Zenoh: Overview

What is Zenoh? Zenoh is a data-centric communication middleware designed for distributed systems spanning embedded devices, robots, edge, and cloud.

Core Concepts:

- Unified **publish/subscribe**, **query**, and **storage**
- Data addressed via **key expressions** (not fixed topics)
- Transport-agnostic: UDP, TCP, QUIC, shared memory

Design Goals:

- Low latency and low overhead
- Efficient operation on constrained devices
- Seamless LAN, WAN, and cloud communication

Zenoh vs DDS in Robotics

Aspect	**DDS**	**Zenoh**
Communication model	Data-centric pub/sub	Data-centric pub/sub + query + storage
Scope	Primarily LAN / local networks	LAN, WAN, cloud, embedded
Transport	UDP multicast / TCP unicast	UDP, TCP, QUIC, shared memory
Discovery	Automatic (multicast-based)	Flexible (supports WAN / NAT)
Latency	Low, deterministic in LAN	Low, but optimized for WAN and intermittent links
Resource usage	Moderate (requires DDS middleware)	Lightweight, low-overhead for embedded devices
Fault tolerance	Local network only	WAN-aware, handles temporary disconnections

PicoZenoh: Lightweight Data Communication

What is PicoZenoh?

- Minimal client implementation of Zenoh protocol
- Designed for **resource-constrained devices** (MCUs, embedded systems)
- Part of the **Zenoh ecosystem**

Key Features

- Very small footprint (tens of kB)
- Written in C for portability
- Supports publish/subscribe and query mechanisms
- Works over multiple transports (UDP, serial, etc.)

Example Use Cases

- Sensor nodes publishing data
- Microcontrollers in robotic systems

ROS 2: Environmental Variables

ROS 2 - Domain ID

What is Domain ID?

A **logical partition** that isolates ROS 2 networks. Nodes with different Domain IDs cannot communicate .

Why Use Domain ID?

- Multiple robots in same network
- Separate development/testing
- Avoid interference between teams

Usage

```
# Set before running nodes
export ROS_DOMAIN_ID=42
ros2 run demo_nodes_cpp talker

# Range: 0-232 (inclusive)
# 0 = default
# Avoid common IDs (e.g., 1, 2)
```

Important

Domain ID determines UDP port numbers: `7400 + DOMAIN_ID` (e.g., ID=42 $\rightarrow$ port 7442)

ROS 2 - Essential Environment Variables

Core Communication

ROS_DOMAIN_ID

0-232 (default: 0)
Isolates ROS 2 networks

ROS_DISCOVERY_SERVER
IP:port of discovery server
Default: (none)

ROS_SUPER_CLIENT
True/False (CLI tools)
Default: False

DDS/RMW Configuration

RMW_IMPLEMENTATION

rmw_fastrtps_cpp
rmw_cyclonedds_cpp
rmw_connextdds

FASTRTPS_DEFAULT_PROFILES_FILE
XML config file path

CYCLONEDDS_URI
Cyclone DDS config file

Security

ROS_SECURITY_ENABLE

true/false

ROS_SECURITY_STRATEGY
Enforce or Permissive
Default: Permissive

ROS_SECURITY_KEYSTORE
Path to keystore directory

Examples

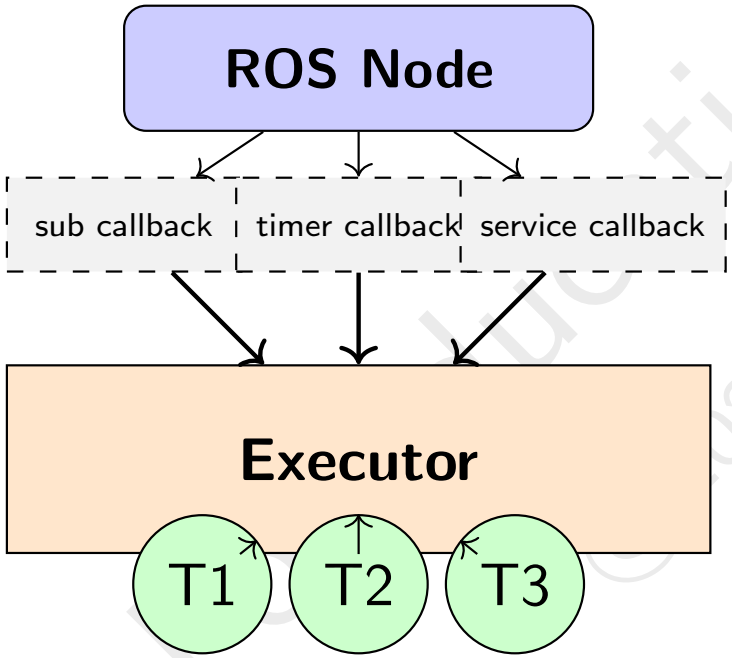
```
1 # Isolate robot fleet
2 export ROS_DOMAIN_ID=42
3
4 # Switch DDS vendor
5 export RMW_IMPLEMENTATION=
      rmw_cyclonedds_cpp
6
7 # Enable security
8 export ROS_SECURITY_ENABLE=true
```

View all: ros2 env — Documentation: https://docs.ros.org

Nodes execution

ROS 2 - What is an Executor?

Execution manager that uses one or more threads to invoke callbacks when messages or events arrive



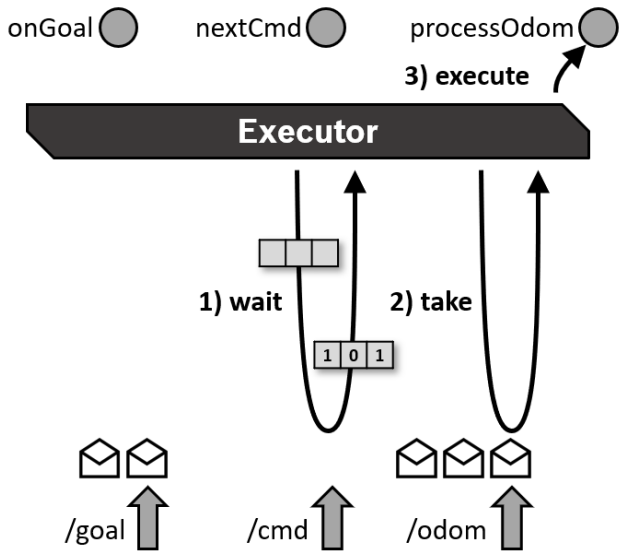
What it does

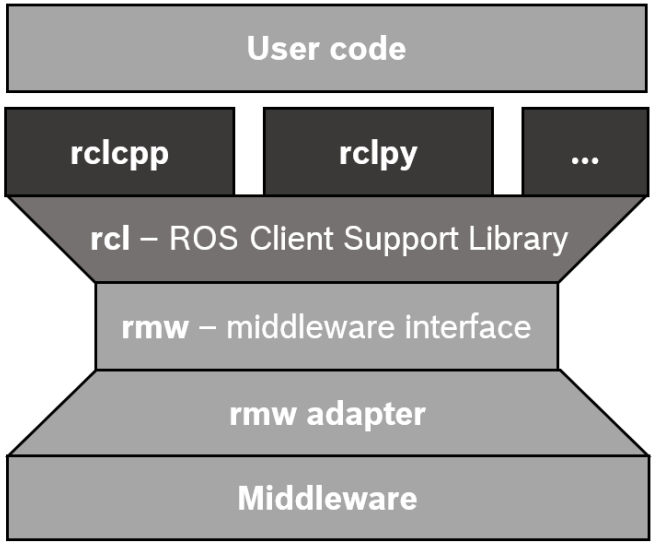
- Waits for incoming messages/events
- Routes them to appropriate callbacks
- Manages thread(s) for execution

Callback Types

- Subscription callbacks
- Timer callbacks
- Service/action callbacks

Executors

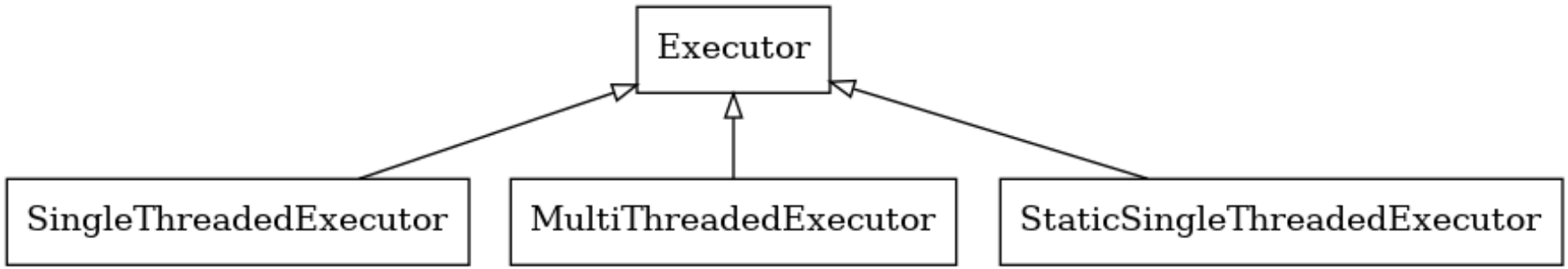




When the spin() is called, the current thread starts querying the rcl and middleware layers for incoming messages and other events and calls the corresponding callback functions until the node shuts down. An incoming message is not stored in a queue on the Client Library layer but kept in the middleware until it is taken for processing by a callback function. A wait set is used to inform the Executor about available messages on the middleware layer, with one binary flag per queue. [6]

[6] https://docs.ros.org/en/iron/Concepts/Intermediate/About-Executors.html

Executor types



The **Multi-Threaded Executor** creates a configurable number of threads to allow for processing multiple messages or events in parallel.

The **Static Single-Threaded Executor** optimizes the runtime costs for scanning the structure of a node in terms of subscriptions, timers, service servers, action servers, etc. It performs this scan only once when the node is added, while the other two executors regularly scan for such changes.[7]

[7] https://docs.ros.org/en/iron/Concepts/Intermediate/About-Executors.html

ROS 2 - Executor Variants

Feature	Static Executor	Dynamic Executor
Callback collection	At creation	Continuously
Performance	Higher	Medium
Flexibility	Lower	Higher
Use case	Fixed node graph	Dynamic changes

Static Executor

- Collects callbacks **once** at start
- No runtime overhead for collection
- Best for fixed-pipeline robots

Dynamic Executor

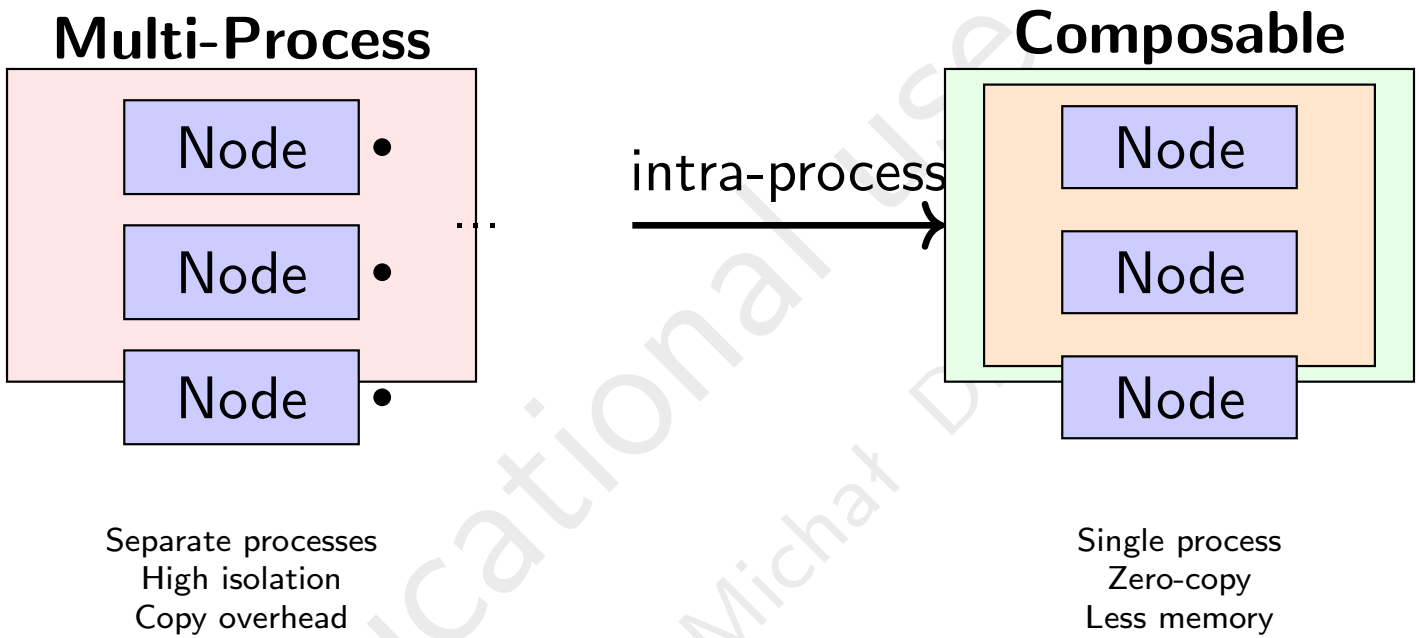
- Continuously checks for new callbacks
- Can add/remove nodes at runtime
- Better for reconfigurable systems

Executors - code example

```
rclpy.init()
try:
  node = Node()
  executor = SingleThreadedExecutor()
  executor.add_node(node)
  try:
    executor.spin()
  finally:
    executor.shutdown()
    node.destroy_node()
finally:
  rclpy.shutdown()
```

IPC & Composable Nodes

ROS 2 - Composable Nodes



Key Takeaway

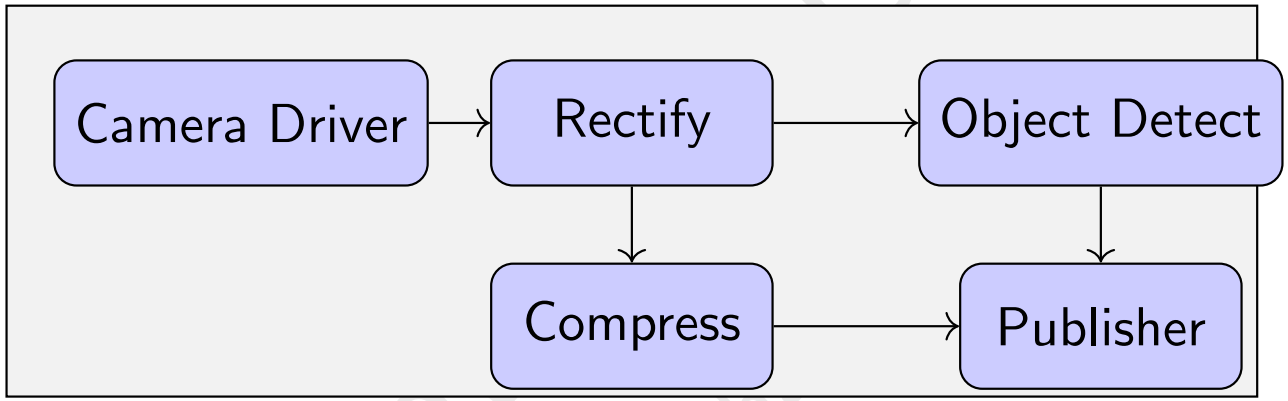
Same node code - only deployment changes!

Nodes are compiled as shared libraries, loaded into container

ROS 2 - Composable Nodes Example

Camera Processing Pipeline - All in One Process

Process: camera_pipeline



Intra-process Communication

- Messages passed via **shared memory**
- **Zero-copy** for large data
- No serialization overhead

Traditional Multi-process

- Serialize → copy → deserialize
- Higher latency
- Higher memory usage

ROS 2 - Component Container

Component Container - special node that **loads multiple components** (nodes) at runtime

Command Line

```
# Start container
ros2 run rclcpp_components
      component_container

# Load components into container
ros2 component load /container \
  composition composition::Talker

ros2 component load /container \
  composition composition::Listener
```

C++ Component

```
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_components/
      register_node_macro.hpp"

class MyComponent : public rclcpp::Node {
public:
  MyComponent(const rclcpp::NodeOptions&
        options)
  : Node("my_component", options) {
    // node initialization
  }
};

RCLCPP_COMPONENTS_REGISTER_NODE(
      MyComponent)
```

Load/unload nodes dynamically without restarting the process!

ROS 2 - Multi-Process vs Composable

Aspect	Multi-Process	Composable
Isolation	High (crash isolated)	Low (crash kills all)
Latency	Higher (serialization)	Lower (zero-copy)
Memory	Higher (per-process)	Lower (shared)
CPU overhead	Higher	Lower
Data passing	Copy	Shared memory
Debugging	Easier	Harder
Startup time	Slower	Faster

When to use composable nodes

- **Large data** - Images, point clouds
- **Low latency** - Real-time control
- **Limited resources** - Embedded systems
- **Tightly coupled** - Pipeline stages

Summary

The end

Thank you!