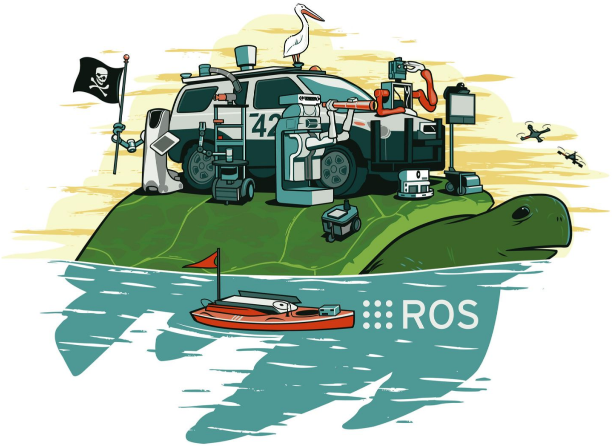
Building robots with ROS 2 and rmw_zenoh



Yadunund Vijay
ROSCon Japan 2025,
Nagoya

Yadunund “Yadu” Vijay

Staff Software Engineer @ Intrinsic

GitHub @yadunund

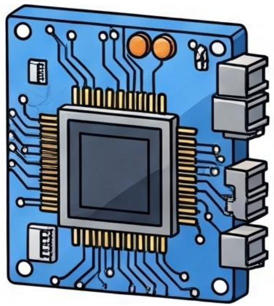
in @yadunundvijay

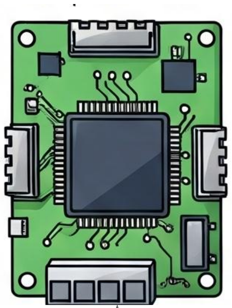
Imagine we're building a product

- A mobile manipulator for logistics and manufacturing.
  - Autonomous navigation
  - Pick & Place
- The platform will need to handle high-throughput sensor data.
- We must ensure the platform has real-time control.
- Interconnect several robots for Fleet Management.

ROS™

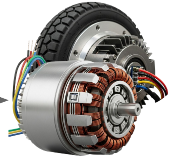
Typical hardware architecture





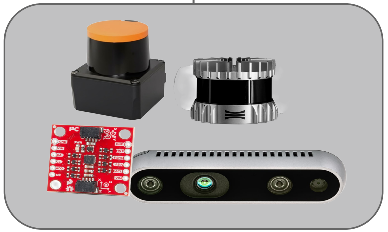
Motor Controller Unit

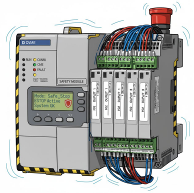
- Embedded platform with RTOS
- Control of motors in arm and base



Main Compute (MC)

- Sensor drivers
- Perception & localization
- Planning & trajectory generation
- AI inference & e2e models
- Cloud connectivity

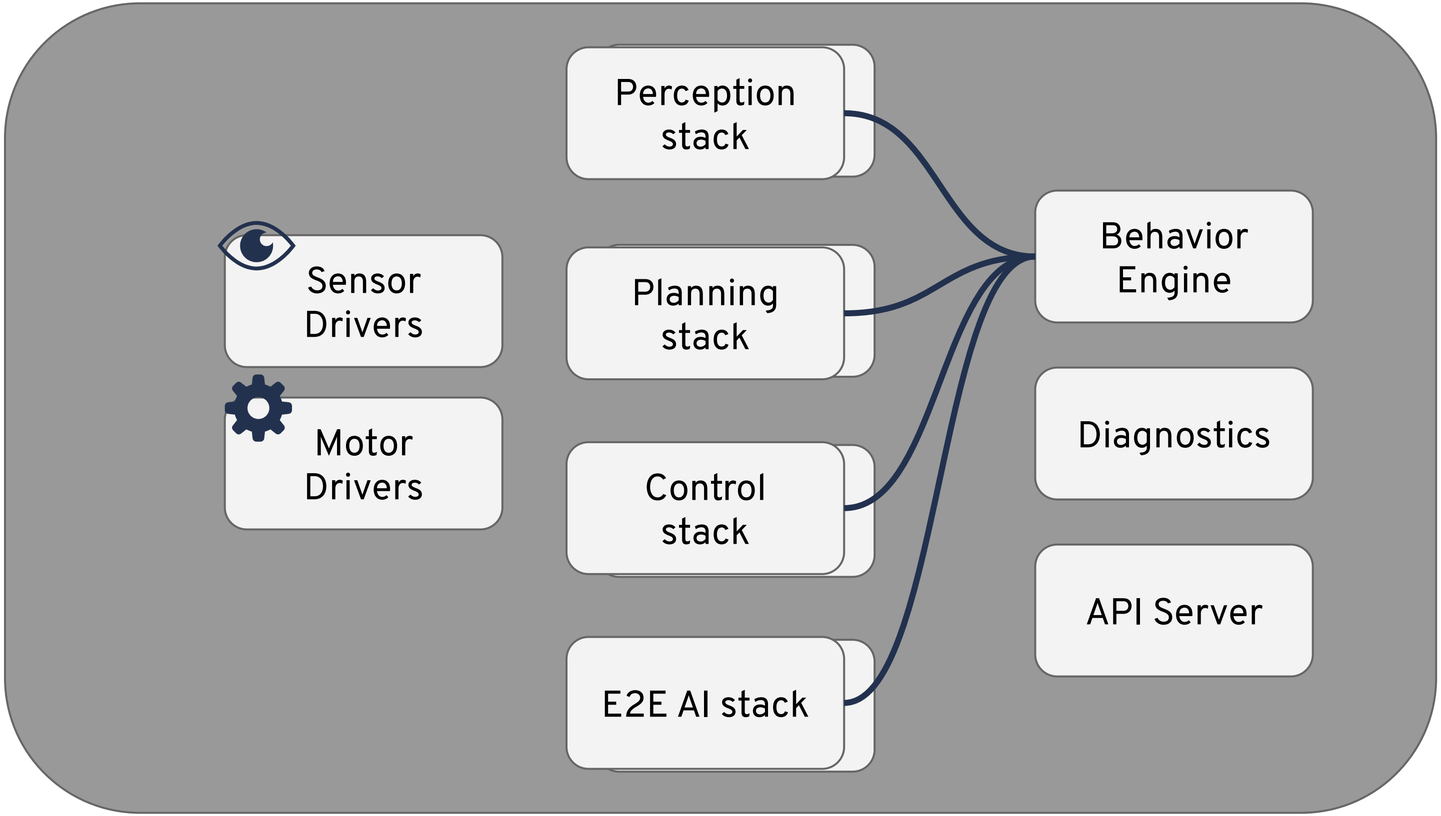


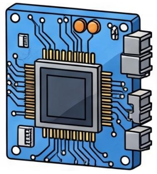


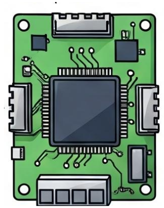
Safety Controller

ROS™

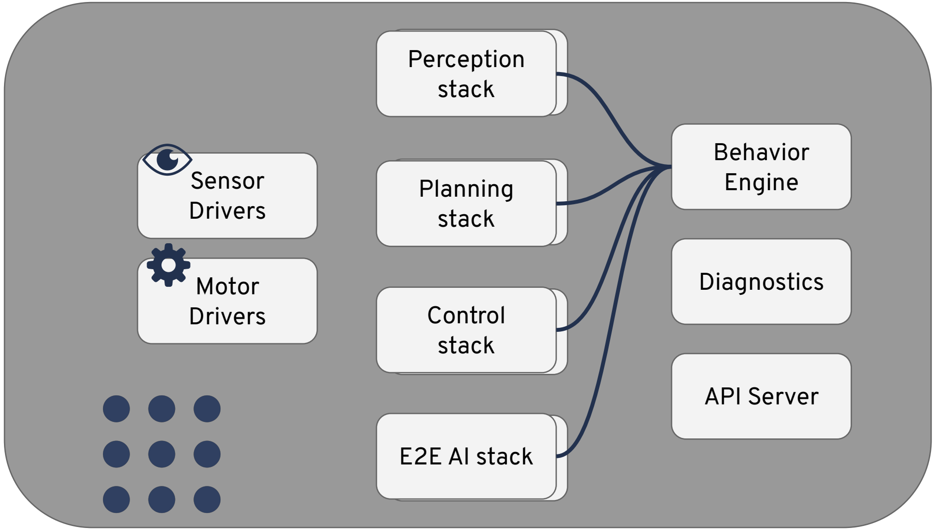
Typical software architecture

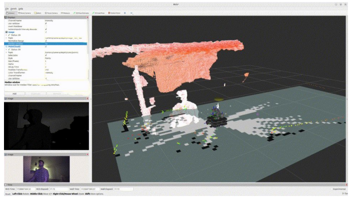


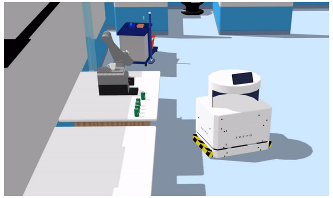


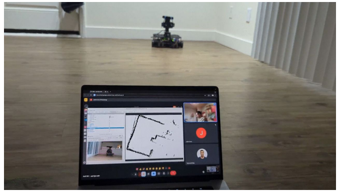


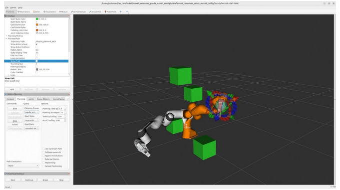
ROS & Friends have your back











NAV 2

MoveIt

open-RMF

ROS™

ROS Best Practices

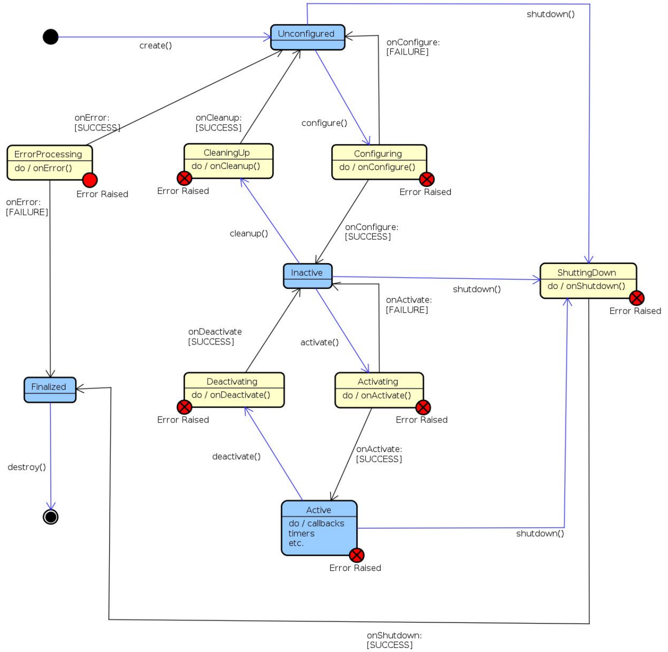
to significantly improve performance

ROS™

https://github.com/ros2/demos/blob/rolling/lifecycle/README.rst

Lifecycle Nodes

- Managed states for ROS 2 nodes
  - Unconfigured, Inactive, Active, Finalized
  - Custom states can exist between these states.
- Why
  - Controlled and deterministic startup.
  - Efficient resource management.
  - Improved fault tolerance and recovery.
  - System health monitoring

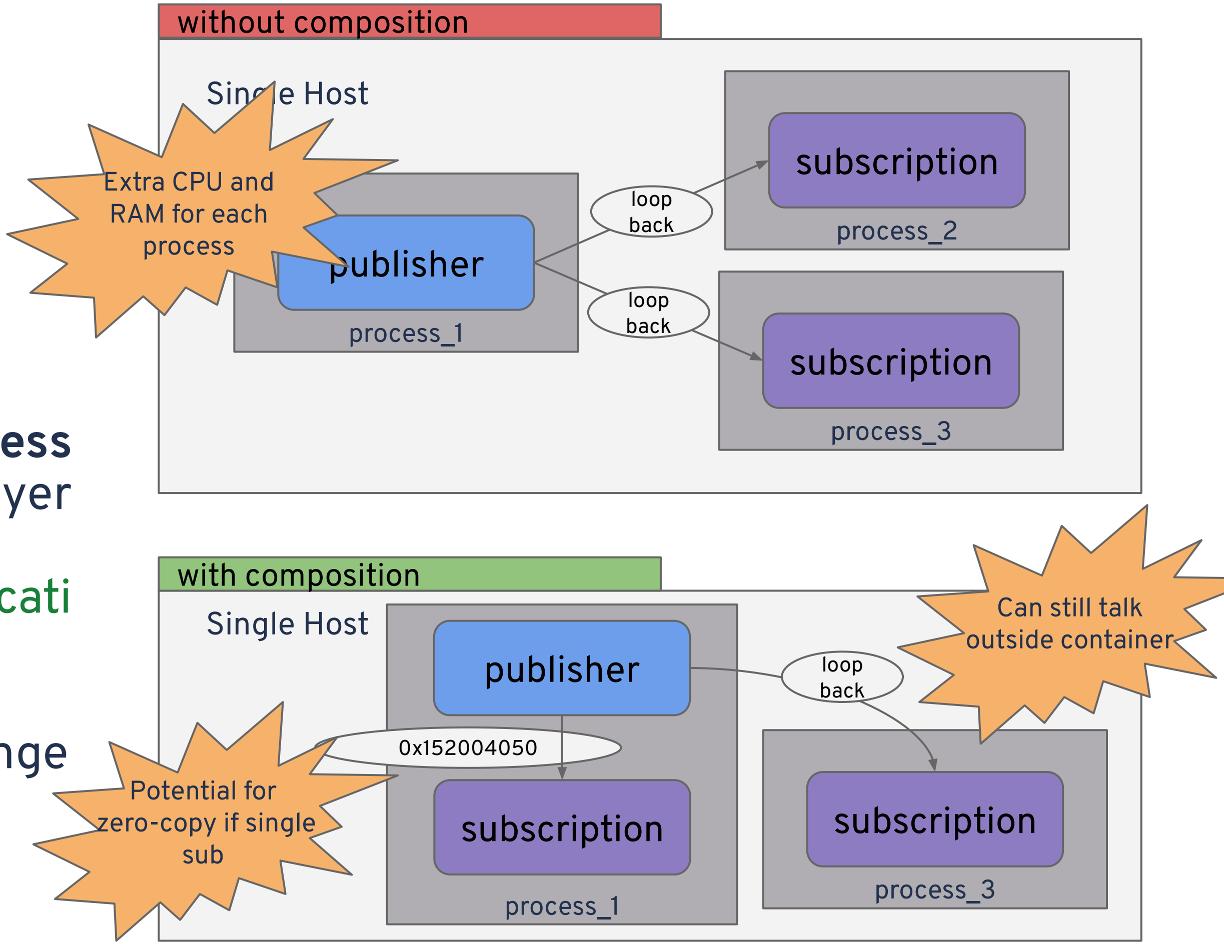


ROS™

Composition

- Enhanced performance from **running multiple nodes in the same process**
- Potential to skip RMW layer completely with intra_process_communication
    - Skip message serialization; exchange pointers
- Supported in rclcpp and pending PR in rclpy.

https://docs.ros.org/en/rolling/Concepts/Intermediate/About-Composition.html#composition



Composition

How much better is performance?

- **28% reduction in CPU** & **33% reduction in RAM** usage for nav2 demo
  - Potential for more in your application!
- Lower latency in the system.
- Lower system load_average from fewer kernel operations related to I/O
- **Consistent performance across middlewares.**

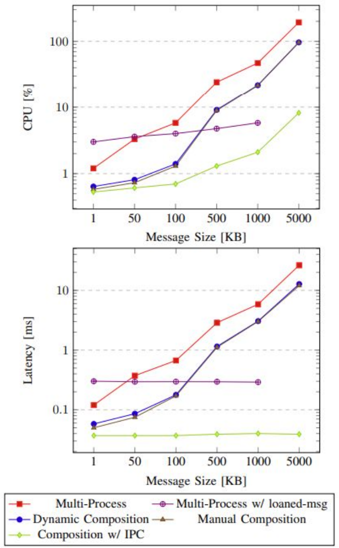


Fig. 3: CPU, latency for different message sizes.

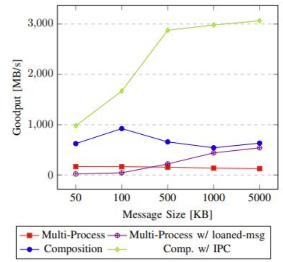


Fig. 4: Maximum goodput between one publisher and one subscription for different message sizes.

TABLE I: Nav2 resources used on ARM and x86 platforms.

ARM	PSS [MB]	CPU [%]
Multi-Process	116.63 ± 0.40	154.27 ± 3.91
Manual Composition	77.84 ± 0.47	110.50 ± 2.87
Dynamic Comp. (isolated)	78.63 ± 0.17	109.62 ± 2.43
Dynamic Comp. (multi-threaded)	75.52 ± 0.71	140.32 ± 7.05
x86	**PSS [MB]**	**CPU [%]**
Multi-Process	118.85 ± 0.36	48.60 ± 3.15
Manual Composition	67.13 ± 0.18	36.60 ± 4.22
Dynamic Comp. (isolated)	67.67 ± 0.14	36.09 ± 4.00
Dynamic Comp. (multi-threaded)	66.71 ± 0.27	46.27 ± 2.80

https://docs.ros.org/en/rolling/Concepts/Intermediate/About-Composition.html#composition

Composition

It's easy to implement!

- Compile **shared library** and **register component.**
- **Tips for fewer copies**
  - **Transfer ownership** when **publishing.**
  - Subscribe to **ConstSharedPtr** for **immutable** reference to message.
- Freedom to select the type of executor.

```
add_library(talker_component SHARED src/talker_component.cpp)
rclcpp_components_register_nodes(talker_component
"composition::Talker")
```

```
// talker_component.cpp

 auto msg = std::make_unique<std_msgs::msg::String>();
 pub_→publish(std::move(msg));

#include "rclcpp_components/register_node_macro.hpp"
RCLCPP_COMPONENTS_REGISTER_NODE(composition::Talker)
```

```
// listener_component.cpp

 auto callback =
  [this](std_msgs::msg::String::ConstSharedPtr msg) → void
  {
    ...
  };

#include "rclcpp_components/register_node_macro.hpp"
RCLCPP_COMPONENTS_REGISTER_NODE(composition::Listener)
```

https://docs.ros.org/en/rolling/Concepts/Intermediate/About-Composition.html#composition

Composition

It's easy to implement!

- Compose process at compile time or runtime!
- Set intra_process_comms=true to bypass middleware if possible.
  - **Zero-copy if single pub & single sub**
- Note: You can still start a node with *ros2 run rclcpp_component talker*

```xml
<launch>
 <node pkg="rclcpp_components"
    executable="component_container"
    name="my_container"
    namespace=""
    output="screen">

   <composable_node pkg="composition"
          plugin="composition::Talker"
          name="talker">
     <param name="use_intra_process_comms" value="true" />
   </composable_node>

   <composable_node pkg="composition"
          plugin="composition::Listener"
          name="listener">
     <param name="use_intra_process_comms" value="true" />
   </composable_node>

 </node>
</launch>
```

https://docs.ros.org/en/rolling/Concepts/Intermediate/About-Composition.html#composition

Type Adaptation

```
// image_publisher.cpp

  auto msg =
std::make_unique<sensor_msgs::msg::Image>();
  pub_→publish(std::move(msg));
```

```
// image_subscription.cpp

  auto callback =
   [this](sensor_msgs::msg::Image::ConstSharedPtr msg) → void
   {
      cv::Mat mat;
      memcpy(*msg→data, mat.data, size);
      // Inference logic.
   };
```

Subscription still copies to CV::Mat

https://docs.ros.org/en/rolling/Concepts/Intermediate/About-Composition.html#composition

Type Adaptation

```
// image_publisher.cpp

  auto msg =
std::make_unique<sensor_msgs::msg::Image>();
  pub_→publish(std::move(msg));
```

Skips copy

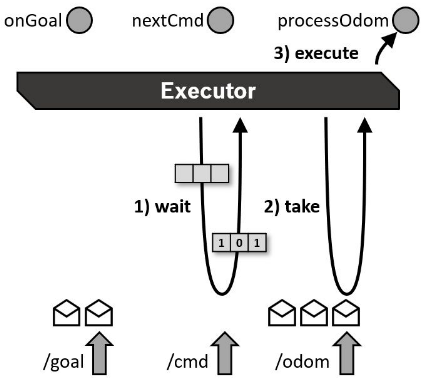
```
// cv_mat_image_type_adapter.hpp
#include "rclcpp/type_adapter.hpp"

class ROSCvMatContainer {
 . . .
};
template<>
struct rclcpp::TypeAdapter<ROSCvMatContainer,
  sensor_msgs::msg::Image> {
   void  convert_to_ros_message( . . . );
  void convert_to_custom( . . . );
};
```

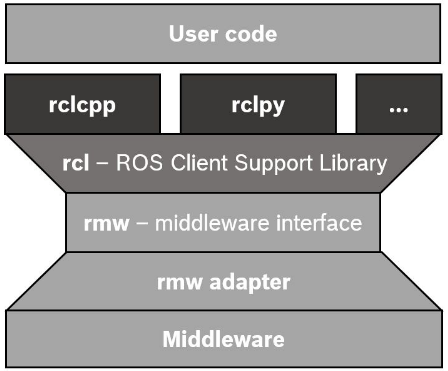
```
// image_subscription.cpp

  auto callback =
    [this](const ROSCvMatContainer & msg) → void
    {
      const cv::Mat & mat = msg.cv_mat();
      // Inference logic.
    };
```

https://docs.ros.org/en/rolling/Concepts/Intermediate/About-Executors.html

Executors

- Execution management in ROS 2 is handled by Executors
    - Callbacks for timers, subs, services, clients
- Default executor is SingleThreadedExecutor






```
// Avoid this → Executor choice is ambiguous.
rclcpp::spin_some(node);

// Be explicit with the executor choice.
rclcpp::executors::SingleThreadedExecutor executor;
executor.add_node(node);
executor.spin();
```

ROS™

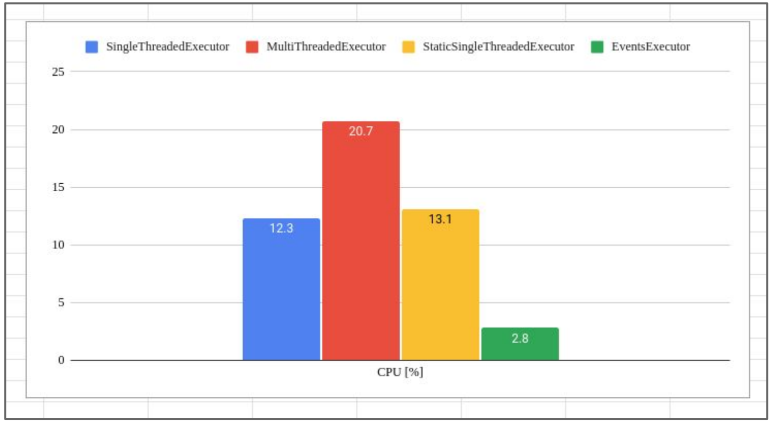
https://discourse.openrobotics.org/t/the-ros-2-c-executors/

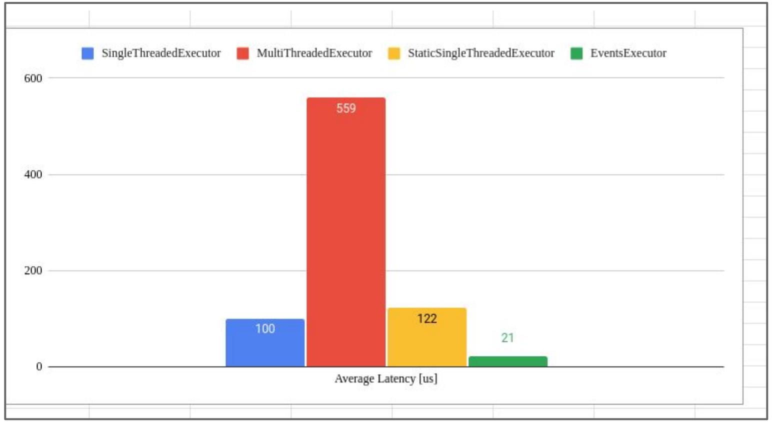
Executors

There are more performant executors!

- EventsExeuctor is experimental but already being adopted actively.
  - Not default yet due to simulation clock issue.
- Other executors like the cm_executor are actively being refined for L-turtle.

```
// Switch to events exeuctor.
rclcpp::experimental::executors::EventsExecutor executor;
executor.add_node(node);
executor.spin();
```





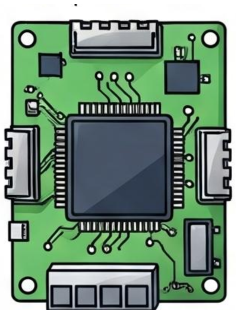
Robot stack

Combine all of these for a really performant stack!

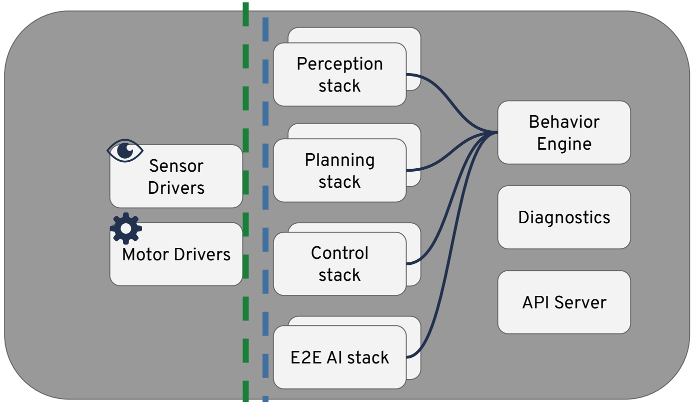
- In a previous case study we **reduced an entire CPU core** usage and **lowered load average** from **8 to 3~** on a **Jetson** platform.

Why do we need a middleware?

Low-level Motor Control Unit (MCU)

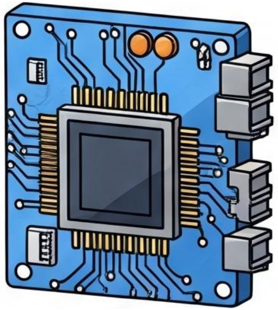


- Embedded platform
- Real-time control of motors in arm and base
- Safety controller



Two boards need to talk

Main Compute (MC)

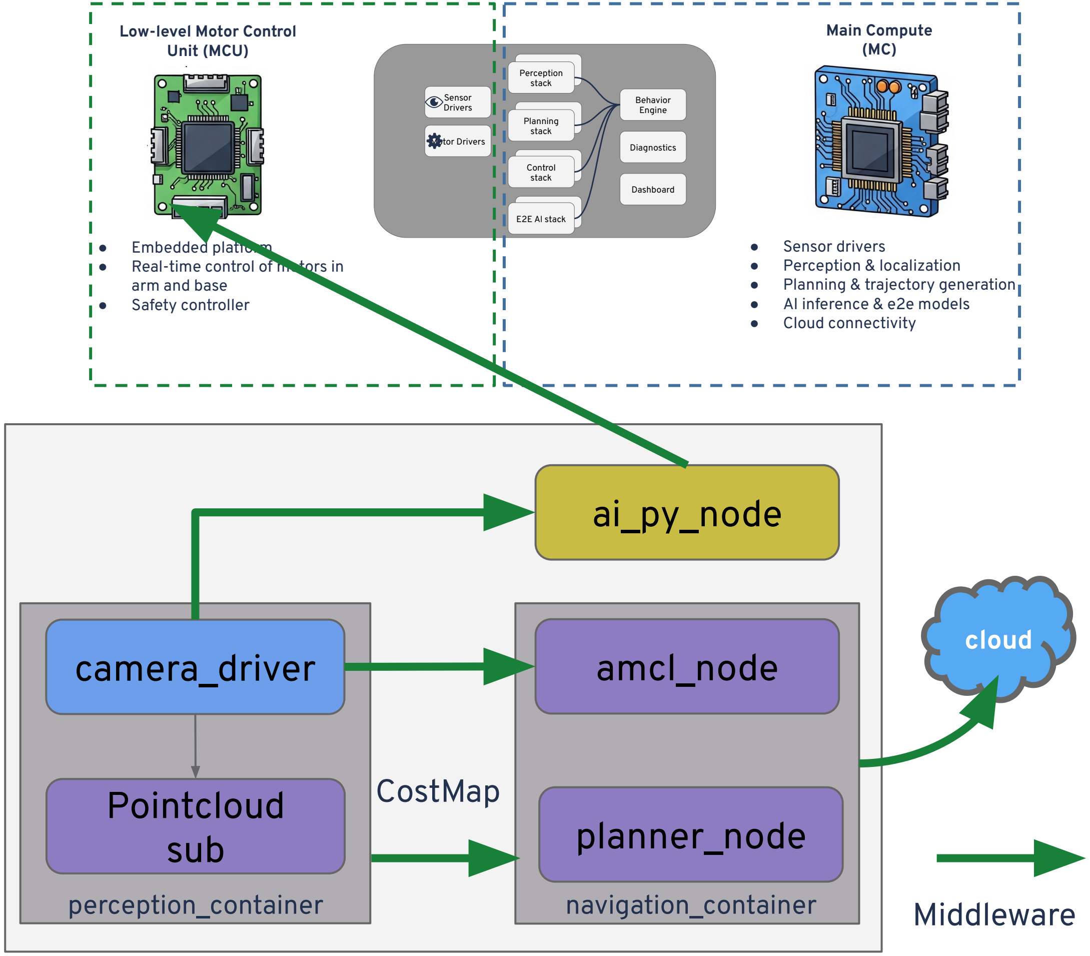


- Sensor drivers
- Perception & localization
- Planning & trajectory generation
- AI inference & e2e models
- Cloud connectivity

Why do we need a middleware?

Within central compute

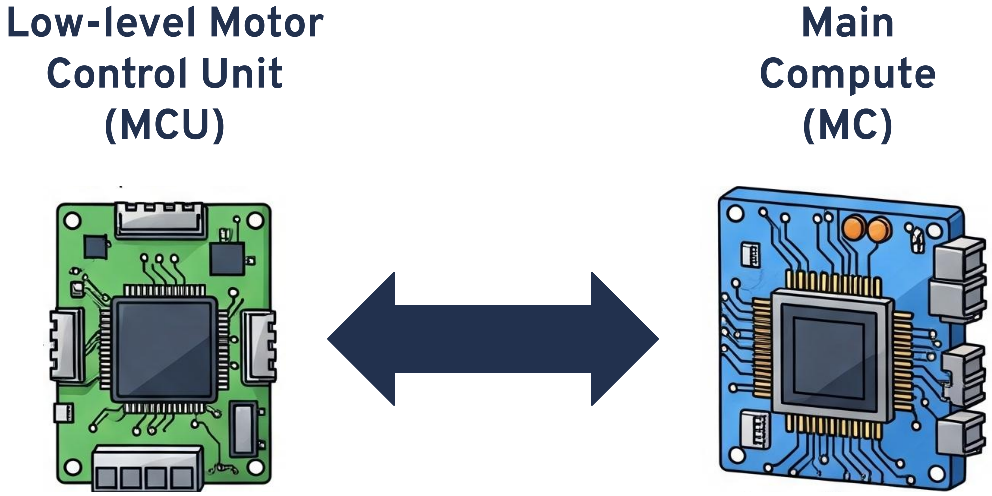
- Not all nodes can run in the same process
  - e.g., rclcpp & rclpy nodes.
- Not every topic has a single pub & sub.
- Some topics have to bridge component containers.
- Some topics have to bridge to the other systems.
  - Eg. Cloud computers



Back to our robot

ROS™

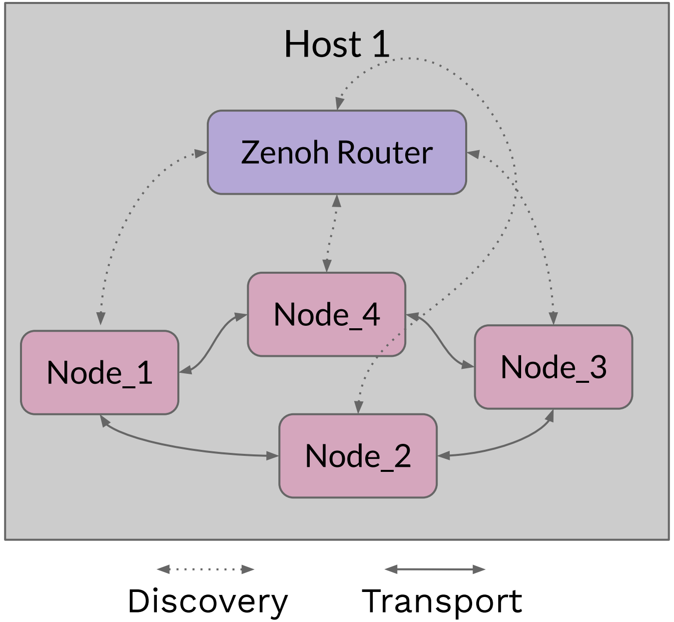
A Tale of Two Worlds: High-Level ROS 2 & Low-Level MCUs



Question: How do we bridge the world of ROS 2 nodes and the world of resource-constrained microcontrollers?

ROS 2 + Zenoh to the rescue

- **rmw_zenoh:** ROS 2 Middleware based on Zenoh protocol writen using zenoh-cpp.
- Core ROS 2 package and Tier-1 status since Kilted Kaiju.
- TCP for discovery and transport.
  - UDP, QUIC, etc can be configured
- Hop-to-hop reliability
- No QoS mismatches.
- Default topology
  - Discovery is brokered by the Zenoh router
  - Data transmission is P2P.
  - Discovery range is localhost only

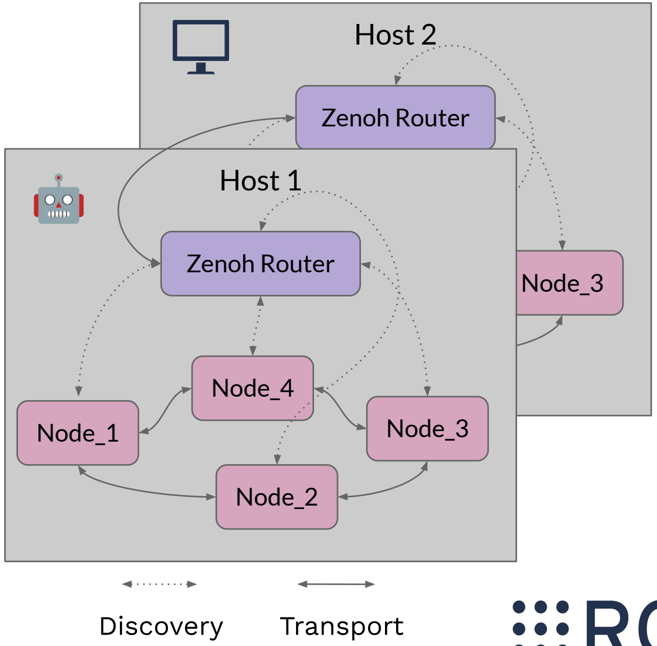


https://github.com/ros/rmw_zenoh

Let’s talk about the router

- **Is it similar ROS 1’s roscore?**

  Yes, but it does a lot **MORE** as we’ll see.
- **What if the router crashes?**
  No impact on running Nodes.
  ROS Daemon still present for graph cache.
  Just restart the router! **No need to re-launch your Nodes.**
- **Is the router mandatory?**
  No. You can configure Zenoh for UDP multicast discovery.



https://github.com/ros/rmw_zenoh

Configuration is easy with rmw_zenoh

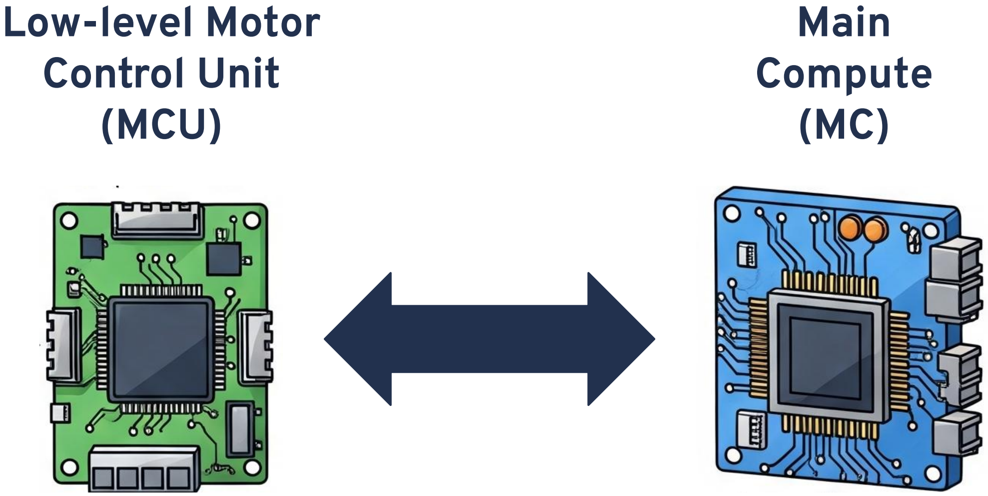
- Default configuration files for Router and Nodes **tuned for good out-of-box experience** for most use cases.
- New config files can be passed by setting ZENOH_ROUTER_CONFIG_URI and ZENOH_SESSION_CONFIG_URI envars
- Fields in the default config can be overwritten using ZENOH_CONFIG_OVERRIDE envar.
  - export ZENOH_CONFIG_OVERRIDE='connect/endpoints=["tcp/192.168.0.3:7447", "tcp/192.168.0.4:7447"]'

```
// default router config.
  mode: "router",
  listen: {
    endpoints: [
      "tcp/[::]:7447"
    ],
  },
  scouting: {
    multicast: {
      enabled: false,
    },
    gossip: {
      enabled: true,
    },
  },
```

```
// default session config.
  mode: "peer",
  connect: {
    endpoints: [
      "tcp/localhost:7447"
    ],
  },
  listen: {
    endpoints: [
      "tcp/localhost:0"
    ],
  },
  scouting: {
    multicast: {
      enabled: false,
    }
```

ROS 2 + Zenoh to the rescue

- **rmw_zenoh** running on Central Compute
- **pico-ros** (zenoh-pico wrapper) running on MCU
- **Zenoh router** running on Central Compute
  - MCU session connects to the router.

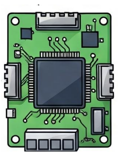


https://github.com/Pico-ROS/Pico-ROS-software

https://github.com/Yadunund/rmw_zenoh_examples/

Interoperability with microcontrollers

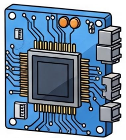
Low-level Motor Control Unit (MCU)



```c
#include <picoros.h>

#define MODE "client"
#define DEFAULT_LOCATOR "tcp/10.0.0.228:7447"

picoros_publisher_t pub_odo = {
   .topic = {
      .name = "odom",
      .type = ROSTYPE_NAME(ros_Odometry),
      .rihs_hash = ROSTYPE_HASH(ros_Odometry),
   },
};
picoros_node_t node = {
   .name = "odometry_node",
};
picoros_node_init(&node);
picoros_publisher_declare(&node, &pub_odo);

uint8_t pub_buf[1024];
ros_Odometry odom = { … }
size_t len = ps_serialize(pub_buf, &odom, 1024);
picoros_publish(&pub_odo, pub_buf, len);
```

Zenoh Router
listen:
“tcp/10.0.0.228:7447”

Main Compute (MC)

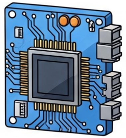


```cpp
subscription_ =
this→create_subscription<nav_msgs::msg::Odometry>(
     "odom",
     odom_qos,
     [this](nav_msgs::msg::Odometry::ConstSharedPtr msg)
     {
          …
     });
```

Congestion control

Zenoh Router
listen:
“tcp/10.0.0.228:7447”

Main Compute (MC)



- **System load is largest at startup** and there is a high probability for important messages to be dropped.
  - Eg. PointCloud, OccupancyGrid
- In rmw_zenoh, KEEP_ALL and RELIABLE QoS settings will force publisher to use **reliable** channels and **always block** packages.
  - **But more resources required.**
- **Solution**: We configure Zenoh to **control dropping** & **priority** policy **per topic**.
  - **blockfirst** makes congestion control more robust and fair.

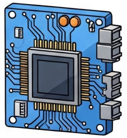
```
qos: {
  publication: [
    {
      key_exprs: ["*/map/*/*"],
      config: {
        congestion_control: "blockfirst",
        priority: "data_high",
        express: true,
        reliability: "reliable",
        allowed_destination: "remote",
      },
    },
  ],
}
```

Set congestion control to blockfirst for /map

https://github.com/Yadunund/rmw_zenoh_examples/

Zenoh Router
listen:
“tcp/10.0.0.228:7447”

Main Compute (MC)



Priority

- Zenoh can prioritize the delivery and processing of data
  - Z_PRIORITY_REAL_TIME: Priority for “realtime” messages.
  - Z_PRIORITY_DATA_HIGH: Highest priority for “data” messages.
  - (among others)

- While RMW API does not allow priority configuration, we can do so on a **per-topic basis in the Zenoh session config.**

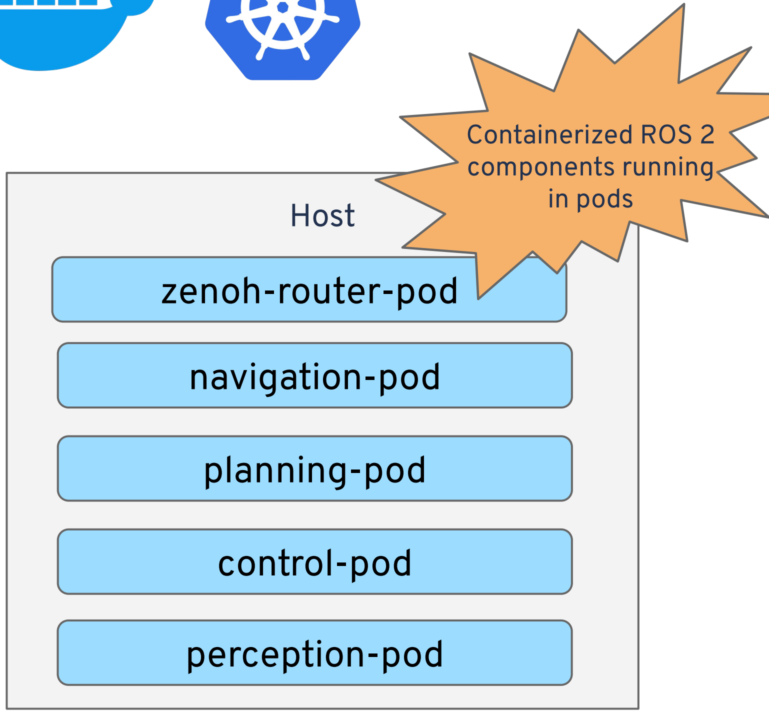
```
qos: {
  publication: [
    {
      key_exprs: ["*/map/*/*"],
      config: {
        congestion_control: "blockfirst",
        priority: "data_high",
        express: true,
        reliability: "reliable",
        allowed_destination: "remote",
      },
    },
  ],
}
```

Set priority to
data_high for /map

https://github.com/Yadunund/rmw_zenoh_examples/

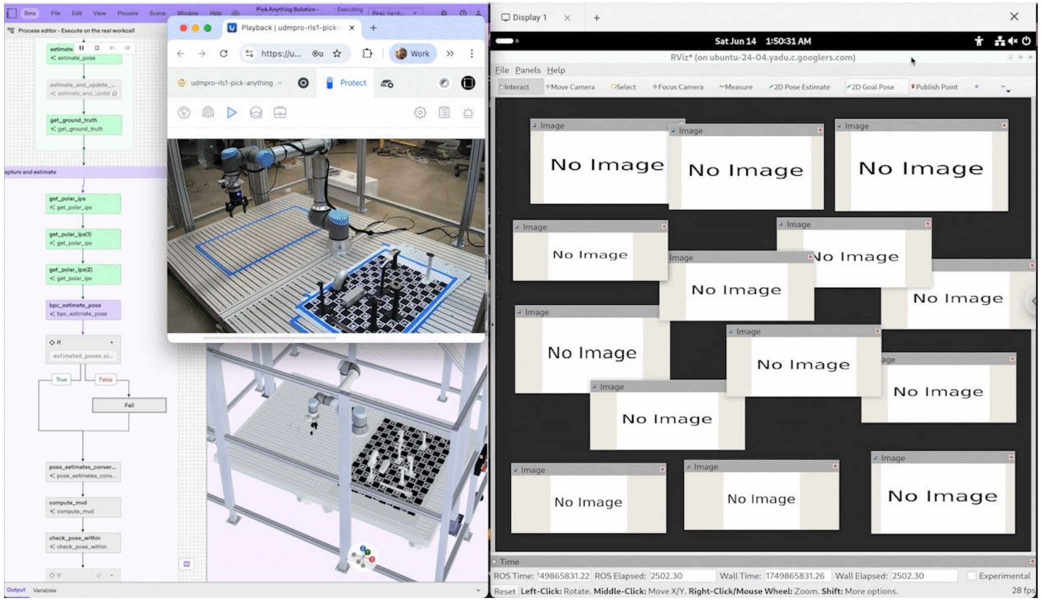
Containerization

- **Production systems often run processes in containers (isolation, OTA updates, etc)**
  - DDS discovery requires complex networking config (`--net=host`, custom bridges, multicast forwarding) to work across containers.
- Wtih `rmw_zenoh` **containers** only need to **connect** to the **`zenohd` router's** port. **No complex networking required.**
- Thee Zenoh router transparently "tunnels" ROS 2 traffic between containers. **It just works**.



Real-use case

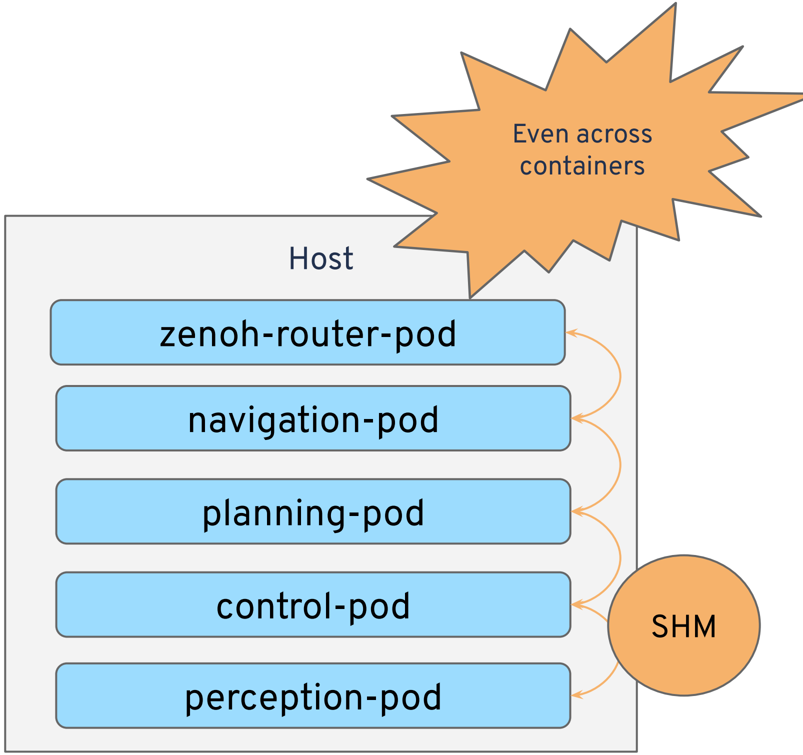
- Intrinsic co-organized Bin Picking Challenge with OpenCV.
- Participants submitted containerized pose estimators.
- Seamless tunneling across containers and WANs
  - **>200MB image payload** delivered **reliably** from **containers** on **edge** device to **containers** on **cloud** over ROS 2 & rmw_zenoh.



https://github.com/opencv/bpc

Shared memory

- **rmw_zenoh** now supports shared memory!
- Disabled by default but can be enabled by overriding config
  - export ZENOH_CONFIG_OVERRIDE='transport/shared_memory/enabled=true'
- Configurable SHM size (16MB default).
- Works seamlessly with remote & non SHM-enabled nodes
  - And across containers with **--ipc=host**



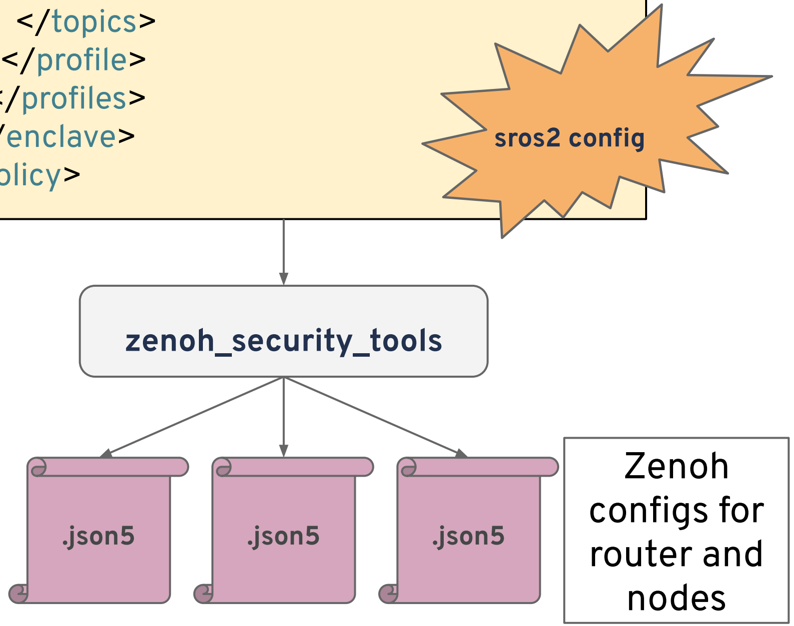
Security is Not an Afterthought

rmw_zenoh supports access control, authentication and encryption.

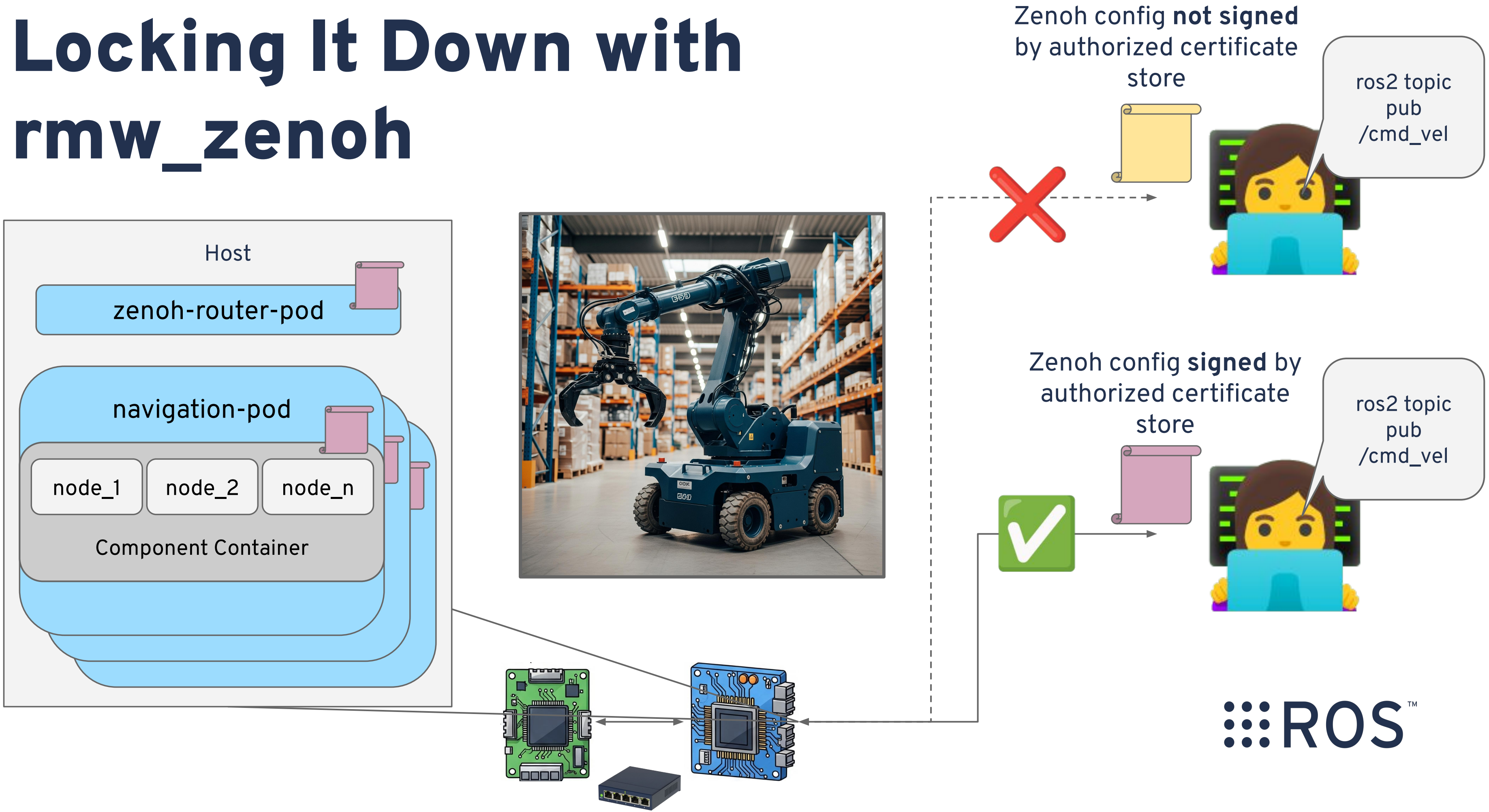
zenoh_security_tools package generate Zenoh configs based on **sros2 policies.**

https://github.com/ros2/rmw_zenoh/tree/rolling/zenoh_security_tools

```xml
<policy version="0.2.0"
 <enclaves>
  <enclave path="/my_robot/map_server">
   <profiles>
    <profile ns="/" node="map_server">
     <xi:include href="common/node.xml"
      xpointer="xpointer(/profile/*)"/>
     <topics publish="ALLOW" >
      <topic>map</topic>
     </topics>
    </profile>
   </profiles>
  </enclave>
</policy>
```



Locking It Down with rmw_zenoh



From One Robot to a Connected Fleet

- **The Business Need:**
  - Fleet management
  - Remote monitoring and telemetry.
  - Remote debugging and teleoperation.
  - Over-the-air updates.
- **The Technical Challenge:** How to bridge a local robot network (often behind NAT/firewalls) to a cloud service securely and efficiently?

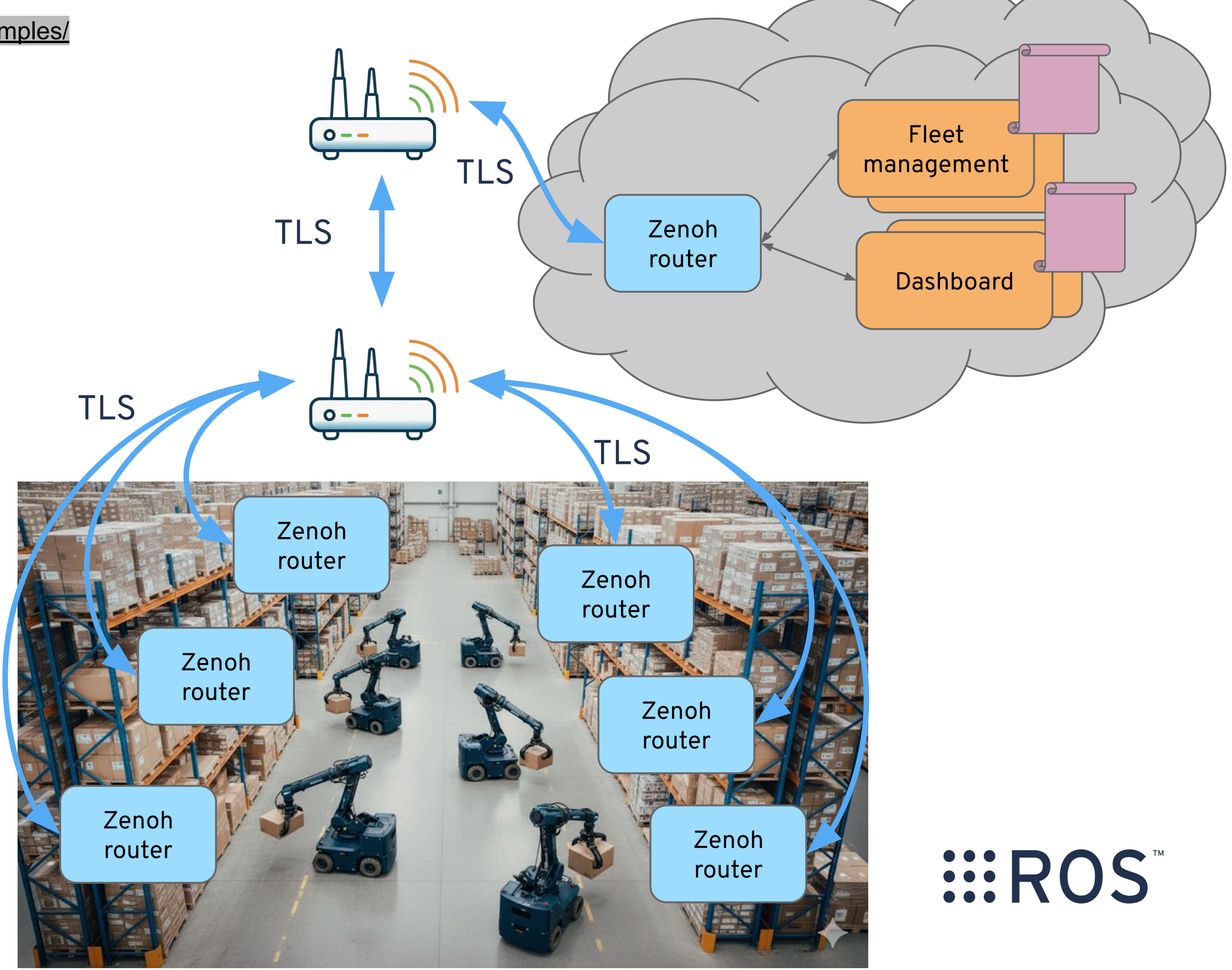
ROS™

https://github.com/Yadunund/rmw_zenoh_examples/

Connecting Worlds with Zenoh Routers

Zenoh routers can be linked together.

Messages can be **securely routed** to a **cloud** router which is authenticated.



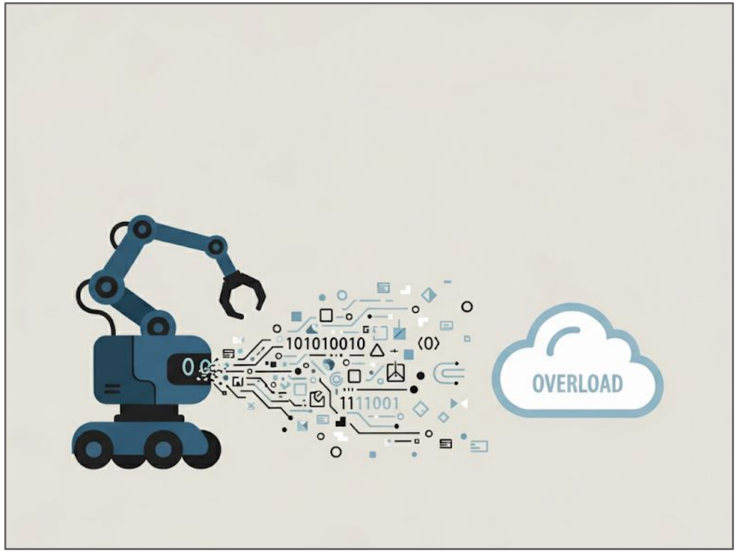
ROS™

The Data Overload Problem

We've connected a robot to other services in the cloud, but we can't send all this data. What do we do?

Solutions:

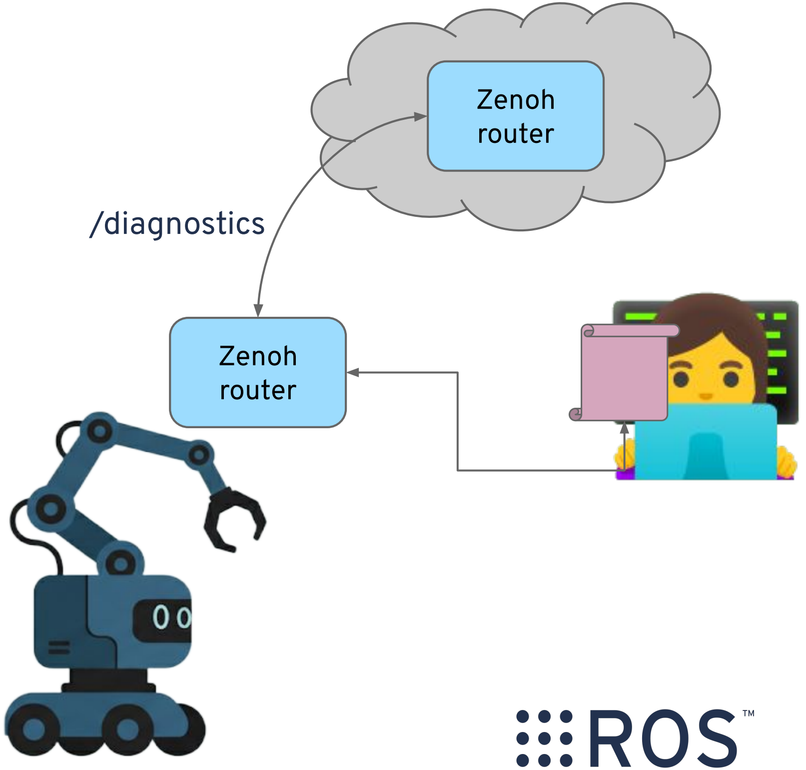
- **Selective Bridging:** Restrict what goes out of the robot.
- **Data Reduction:** Compress and Downsample data.



ROS™

Selective bridging

- **Cloud router** can be authenticated to access only **specific topics** on the robot.
- Robot's router will not forward topics topics not in the allow list.
- To **remotely debug** the robot, a developer can **authenticate** with the robot's router using a **config** with **broader permissions**.



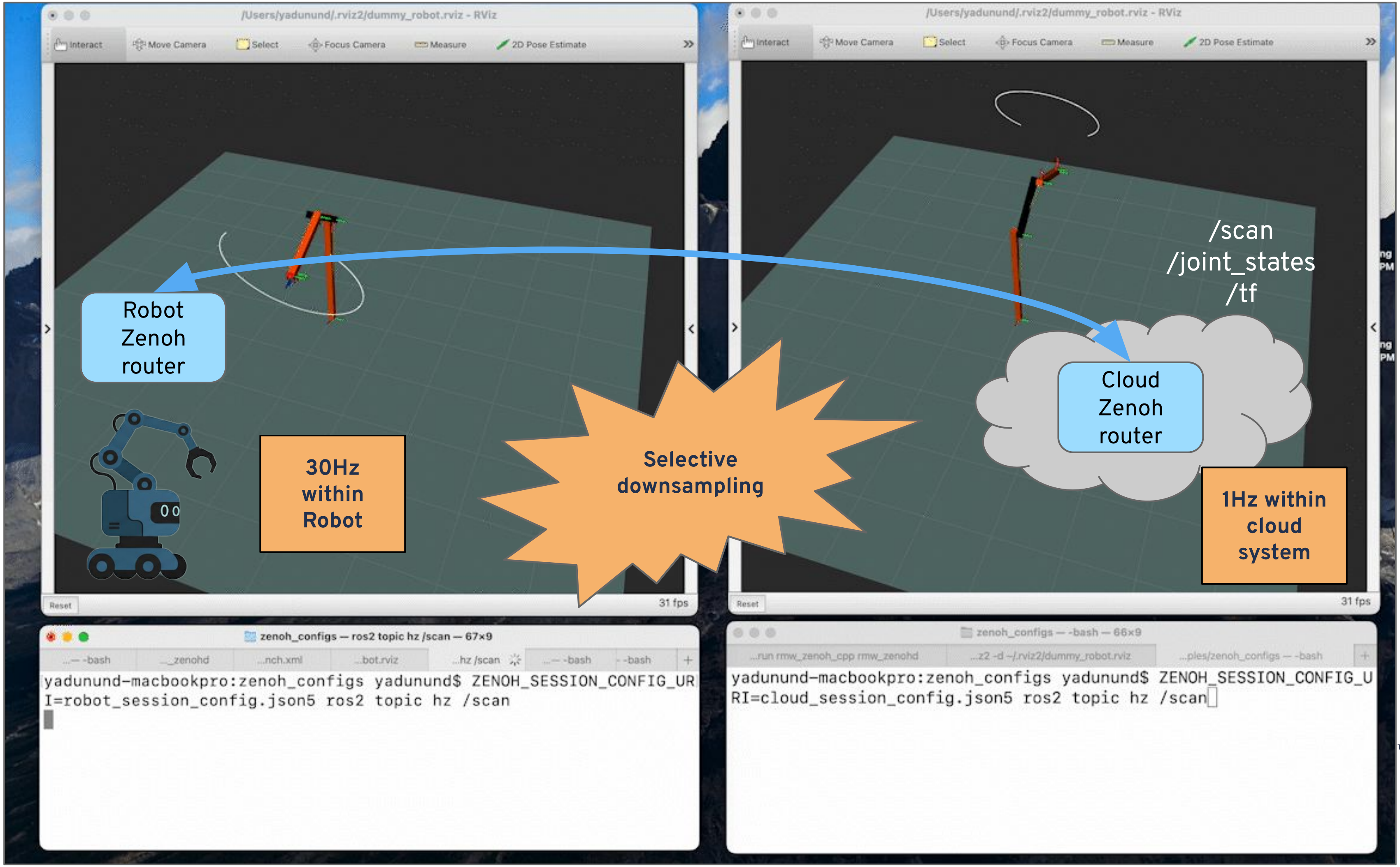
ROS™

Downsample high bandwidth topics

- We still want some topics but at a lower frequency.
- **Configure robot's Zenoh router to automatically downsample topics when sending to the cloud!**
- Also useful to control bandwidth and adapt traffic when crossing system boundaries (eg, from wired LAN to wireless or GSM/LTE network)

```
// Robot Router config.
downsampling: [
    {      rules: [
        {
          key_expr: "*/scan/*/*",
          freq: 1.0
        },
        // 1Hz for /tf topic.
        {
          key_expr: "*/tf/*/*",
          freq: 1.0
        },
      ],
    },
  ],
```

https://github.com/Yadunund/rmw_zenoh_examples/



/Users/yadunund/.rviz2/dummy_robot.rviz - RViz
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
Robot Zenoh router
30Hz within Robot
Selective downsampling
/scan
/joint_states
/tf
Cloud Zenoh router
1Hz within cloud system
Reset
31 fps
zenoh_configs — ros2 topic hz /scan — 67×9
yadunund-macbookpro:zenoh_configs yadunund$ ZENOH_SESSION_CONFIG_URI=robot_session_config.json5 ros2 topic hz /scan
zenoh_configs — -bash — 66×9
yadunund-macbookpro:zenoh_configs yadunund$ ZENOH_SESSION_CONFIG_URI=cloud_session_config.json5 ros2 topic hz /scan

QUIC transport outside robot

- **QUIC** is a modern alternative designed to be faster, more efficient, and more secure than TCP/TLS.
  - Faster connections, no head-of-line-blocking, built-in security (TLS 1.3)
- The connection is identified by a unique ID, not the IP address.
  - Switch networks without dropping the connection!
- Configure Zenoh routers to transmit data between robot and cloud routers over QUIC!
  - QUIC is supported for both best effort and reliable traffic

QUIC link used for all connections **OUTSIDE** the robot

```
{
  connect: {
    endpoints:
["quic/your.cloud.server:7447"] // Use
QUIC for robustness
  },
  // ... security config from before
}
```

TCP link used for all connections **WITHIN** the robot

HMIs and dashboards

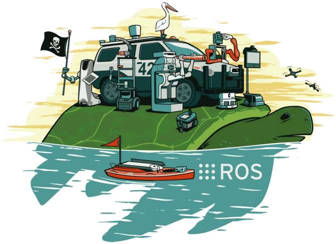
```
// In the browser HMI
import { Zenoh } from '@zenoh/zenoh-ts';

const session = await Zenoh.open();
// Get status of robot 42
const status = await
session.get('robot_42/status');
```

- Zenoh isn't just for backends. It can run directly in a web browser, allowing you to build rich, live HMIs with minimal effort.
  - Typescript implementation of Zenoh: zenoh-ts
- Directly subscribe and visualize data from robots in the fleet!
  - Interoperability with rmw_zenoh is not as seamless like pico-ros.
  - Community contributions welcome!

ROS™

https://github.com/eclipse-zenoh/zenoh-ts

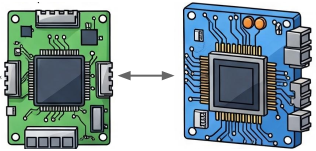
Putting it all together



ROS and it's ecosystem is full of capabilities to build most types of robotic applications

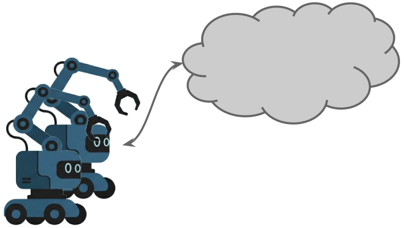
Leverage LifecycleNodes, Composition, Type Adapter & Events Exeuctor to maximize performance

rmw_zenoh is a robust and performant middleware for building ROS 2 applications



Seamless communication across resource constrained systems

Fine tune transport within robot for reliable performance



Seamless interconnection to remote systems with fine-grain control of transport

Questions

Congratulations on architecting a robust and scalable robot product!