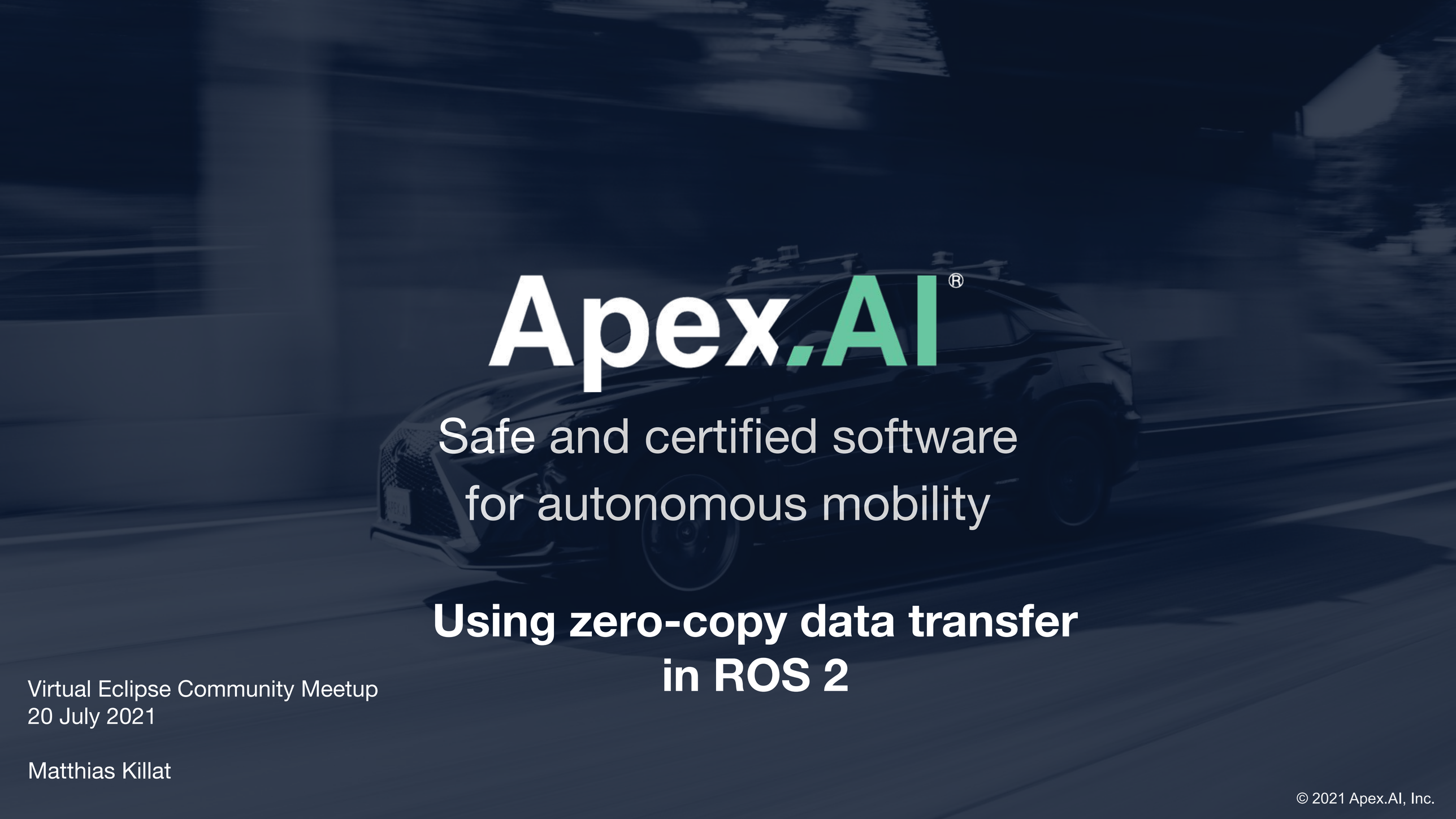


Apex.AI®

Safe and certified software
for autonomous mobility

Using zero-copy data transfer in ROS 2

Virtual Eclipse Community Meetup
20 July 2021

Matthias Killat

Using zero-copy data transfer in ROS 2

Agenda

- Why do we need zero-copy communication?
- Demonstrate how to use shared memory data transfer in ROS™ 2 Galactic
- Explain how Eclipse Cyclone DDS™ is used as ROS 2 middleware
- Show how Eclipse iceoryx™ is used for shared memory communication

Outline

Part 1

1. What is zero-copy data transfer and why do we need it?
2. ROS 2 communication
3. Eclipse Cyclone DDS
4. Eclipse iceoryx

Part 2

1. ROS 2 zero-copy example
2. Performance
3. Limitations
4. Future Work

1 Zero-Copy Communication in ROS 2

Zero-Copy Data Transfer

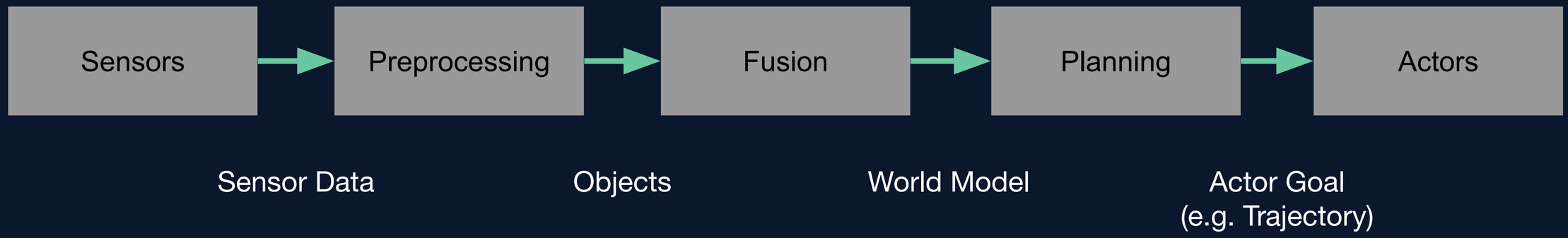
Modern robotics systems are distributed and consist of multiple applications

Modular design promotes reusability and extensibility

It also leads to the challenge of exchanging large volumes of data between modules ...

- data from the sensors themselves (video, radar, Lidar)
- intermediate computation results (objects, descriptors, world model, etc.)
- avoid copying data if possible

Perception Pipeline



Zero-Copy Data Transfer

Zero-copy
- data is generated in memory at its destination (e.g. by sensors or as a computation result)
- transmission between applications does not incur additional copies
- data transmission cost is independent of message size

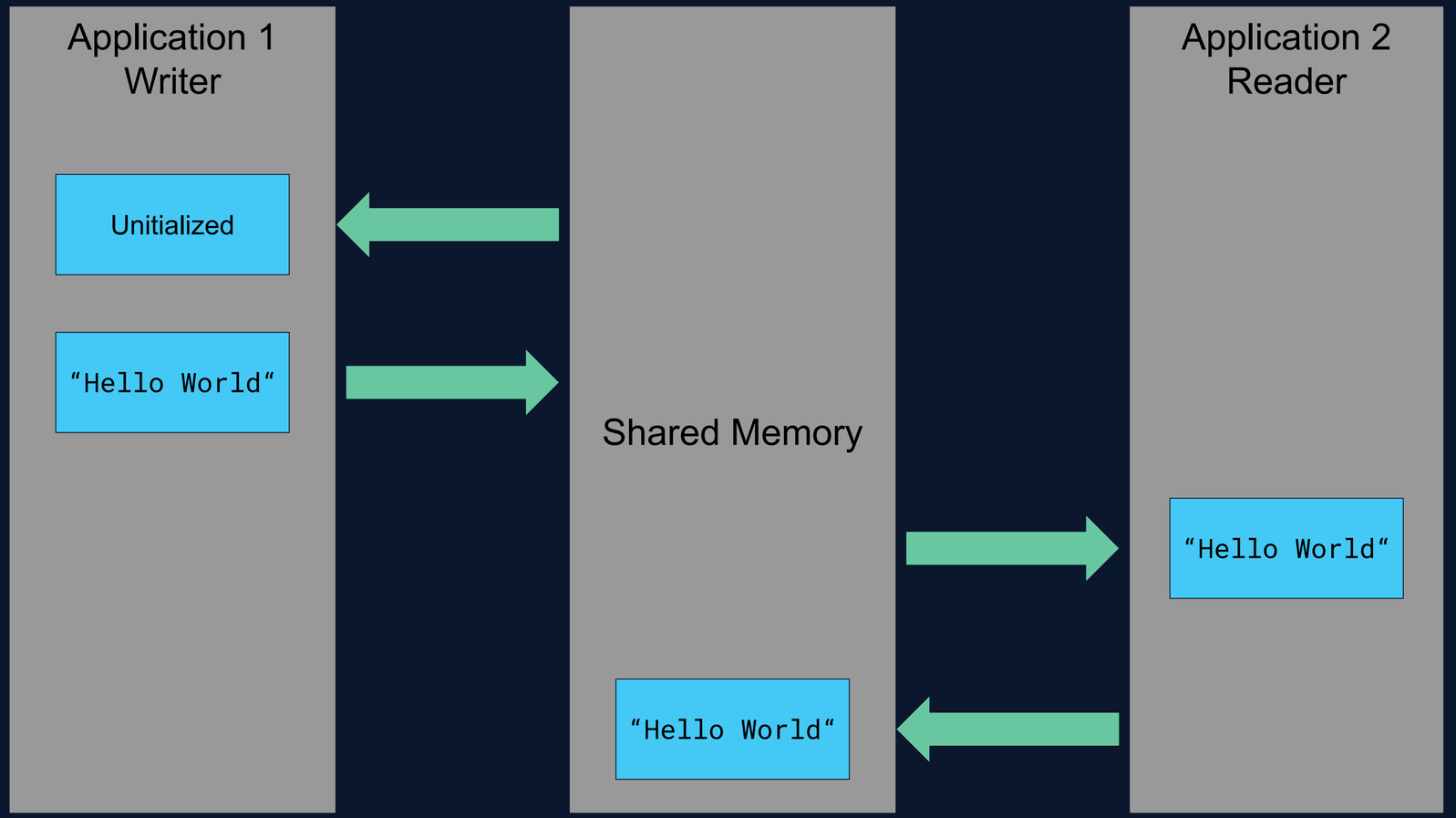
Shared memory
- requires applications to be able to share the same memory (i.e. run on the same hardware device)
- data is not transferred but access permission is passed around
- from a user interface perspective it is like sending the data

Multiple hardware devices
- if the data is required on a different device (ECU) no zero-copy data transfer is possible
- in this case some kind of network communication is usually used
- copies can and should still be minimized

We focus on the single device use case

Zero-Copy Data Transfer

1. request memory block
2. write data
3. pass data to the reader
4. read data
5. release memory block



No copies are required since we directly construct the data at a location where it can be read

ROS 2

ROS 2 is a framework to develop distributed robotics applications.

Latest release: Galactic Geochelone (May 2021)

- applications contain ROS nodes
- nodes perform computations on data received from other nodes
- results are usually published for other nodes to process

Publish subscribe communication

- messages belong to a specific topic
- data is passed between nodes using publishers
- nodes can be subscribed to specific topics
- subscription can be cancelled by the subscribing nodes

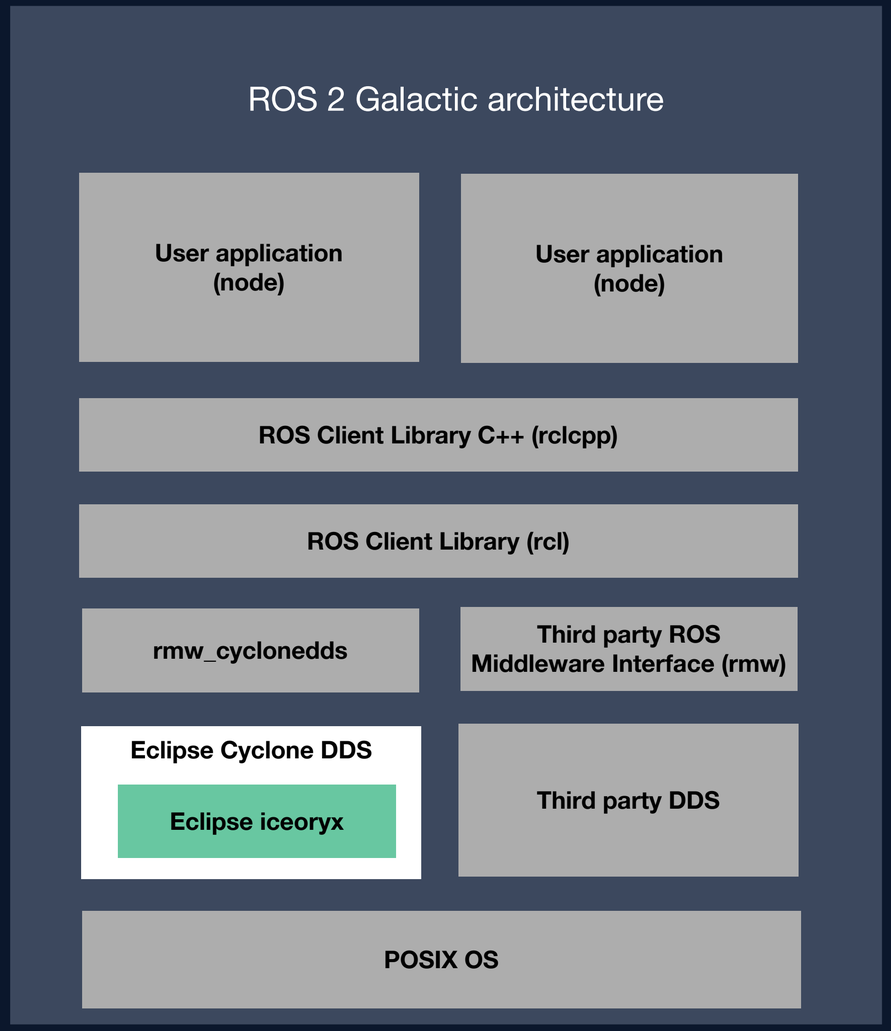
ROS 2 Architecture

ROS 2 middleware (RMW)

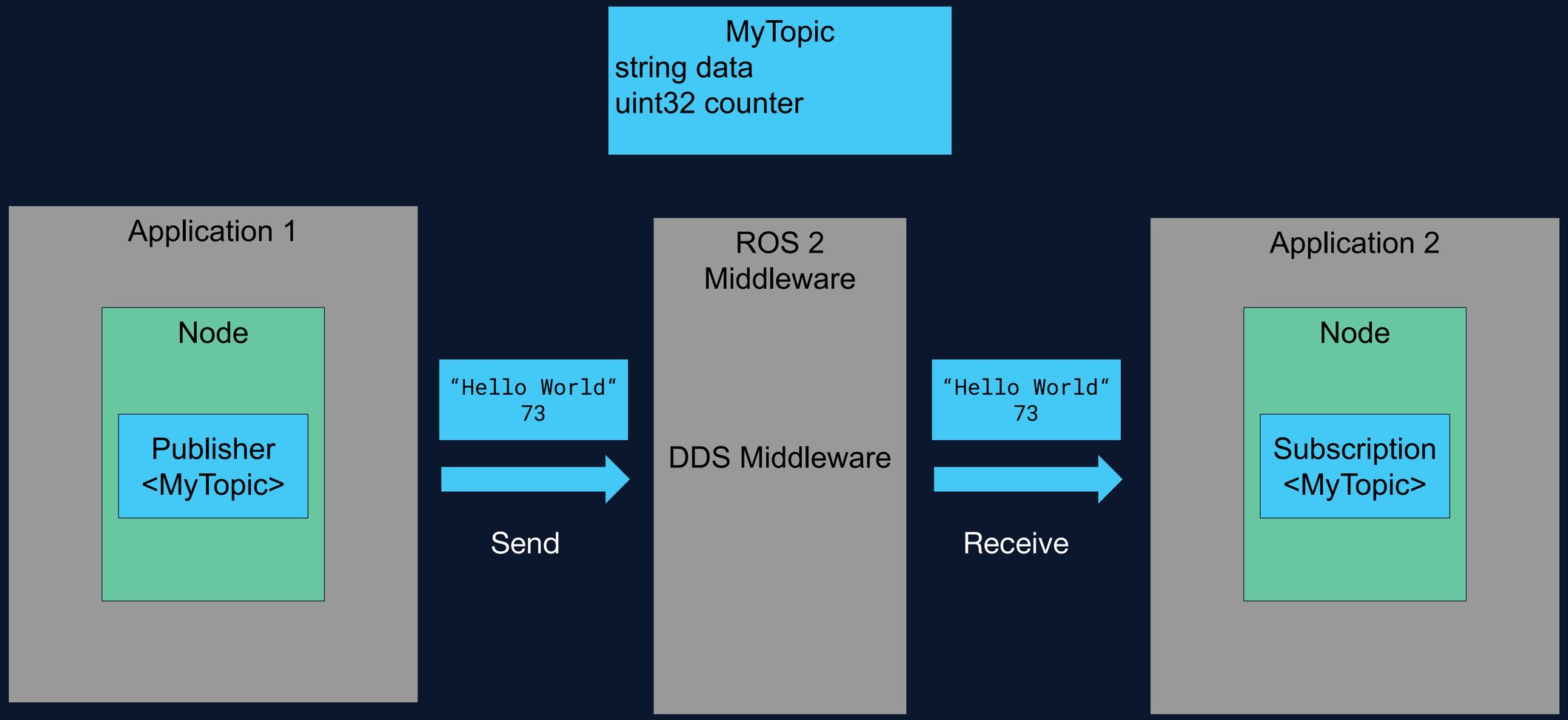
- applications use the rclcpp interface
- messages are passed using a DDS middleware implementation
- Eclipse Cyclone DDS is the default middleware implementation of ROS 2 Galactic
- middleware is exchangeable
- applications based on different RMW implementations can communicate with each other

Complex systems

- hundreds of nodes distributed over 10 and more applications
- hundreds of different message types and thousands of topics



ROS 2 Communication



Uses the network interface to send the data.
Applications may be on different machines.

ROS 2 Communication

Messages

- messages are defined in an interface definition language (IDL)
- these are compiled into language specific data types, e.g. C++
- the language specific types are used in the application

ROS message MyTopic.msg

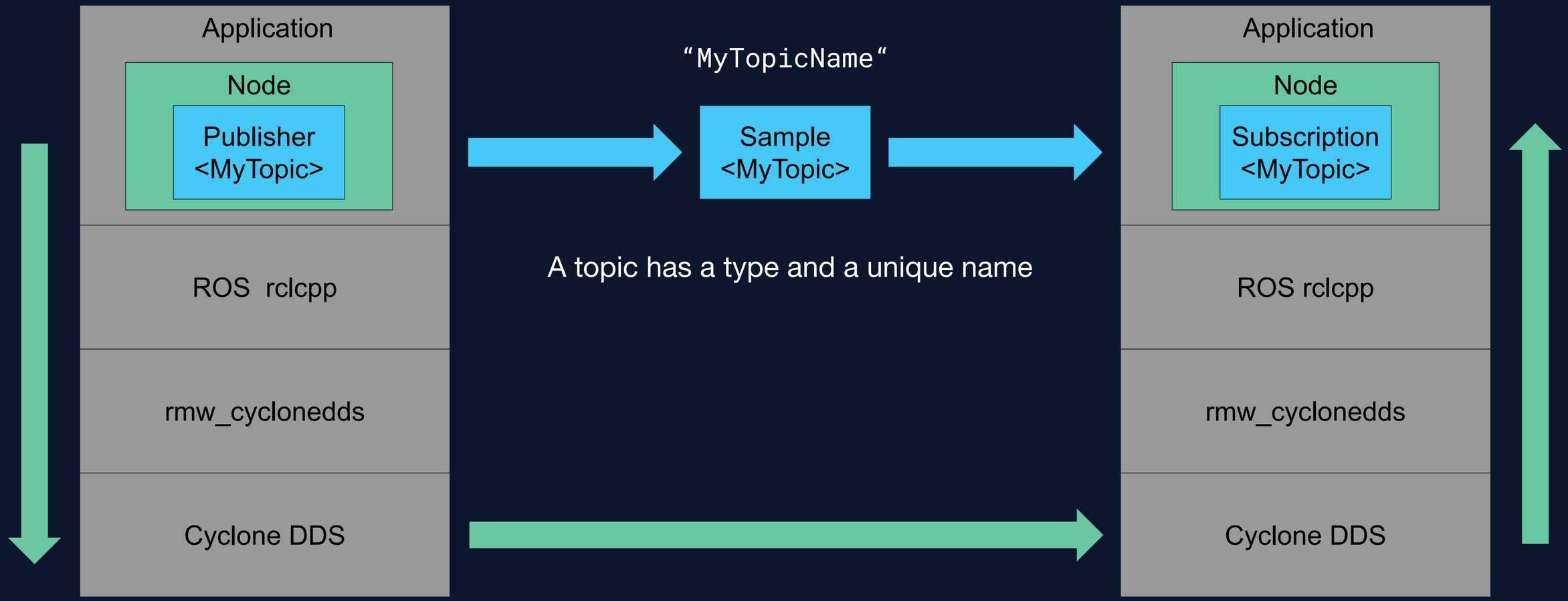
```
string data
uint64 counter
```

C++ Type MyTopic.hpp

```
class MyTopic {
  std::string data;
  uint64_t counter;
  // ROS IDL compiler generates member types from IDL representation
  // ...
  // type specific functions required by ROS 2
};
```

This means the application will at least partly rely on the specific implementation of the type.

ROS 2 Communication



Data is passed over the network stack by default

Data Distribution Service (DDS)

- middleware standard for inter-machine communication
- specified by the Object Management Group
- dependable high-performance communication for real-time systems
- multiple vendor implementations are interoperable

Features
- publish-subscribe communication
- code generation for user defined topic types with an interface definition language (IDL)
- various Quality of Service (QoS) settings
  - Reliability - is sent data guaranteed to arrive? (i.e. resend if required)
  - History - store previously sent data (up to some point)
  - Durability - defines storage of historical data (e.g. persistent vs. non-persistent)
  - others such as Deadline are less important
- abstraction of network communication (UDP multicast)

Used in robotics, autonomous vehicles, aerospace, defense, transportation and others

Eclipse Cyclone DDS

Open Source DDS implementation hosted by the Eclipse Foundation
Developed and maintained by ADLINK Technology Inc. and Apex.AI Inc.

- native implementation is in C https://github.com/eclipse-cyclonedds/cyclonedds
- C++ binding exists as well https://github.com/eclipse-cyclonedds/cyclonedds-cxx
- used by ROS 2 Galactic as default middleware https://github.com/ros2/rmw_cyclonedds

Latest version 0.8

- supports inter- and intra-machine communication
- with default settings data transfer via network interface
- uses loopback interface for intra-machine communication

Shared memory
- restricted to data exchange on the same machine
- uses the Eclipse iceoryx middleware
- must be enabled at runtime (via configuration)

Eclipse iceoryx

Open Source shared memory middleware hosted by the Eclipse Foundation
Developed and maintained by Robert Bosch GmbH, Apex.AI Inc. and community

- native implementation in C++ https://github.com/eclipse-iceoryx/iceoryx
- C language binding
- used by Eclipse Cyclone DDS but can also be used independently
- supports only intra-machine communication

Latest LTS version 1.0.1 (April 2021)

Features
- publish-subscribe communication
- typed API for user-defined C++ classes as topics (with restrictions)
- untyped API for raw memory transfer
- zero-copy data transfer
- lock-free for robustness (and performance)
- uses a static memory model required in high-safety applications
- the iceoryx hoofs library provides useful building blocks similar to the STL for high-safety applications

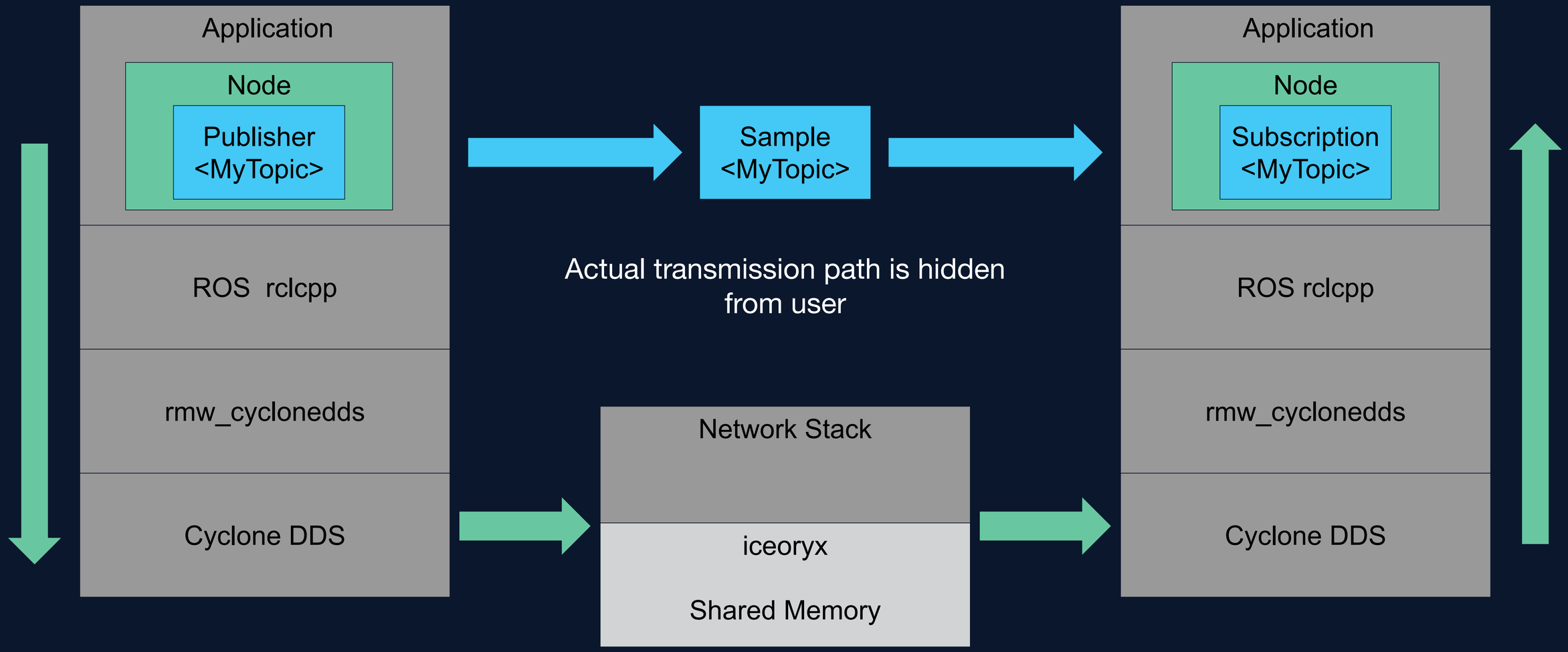
Putting It All Together

ROS 2
- is based on publish subscribe communication
- Is suited for intra machine and inter machine communication between ROS nodes
- uses Eclipse CycloneDDS as default middleware (Galactic Release)
- uses the network stack by default

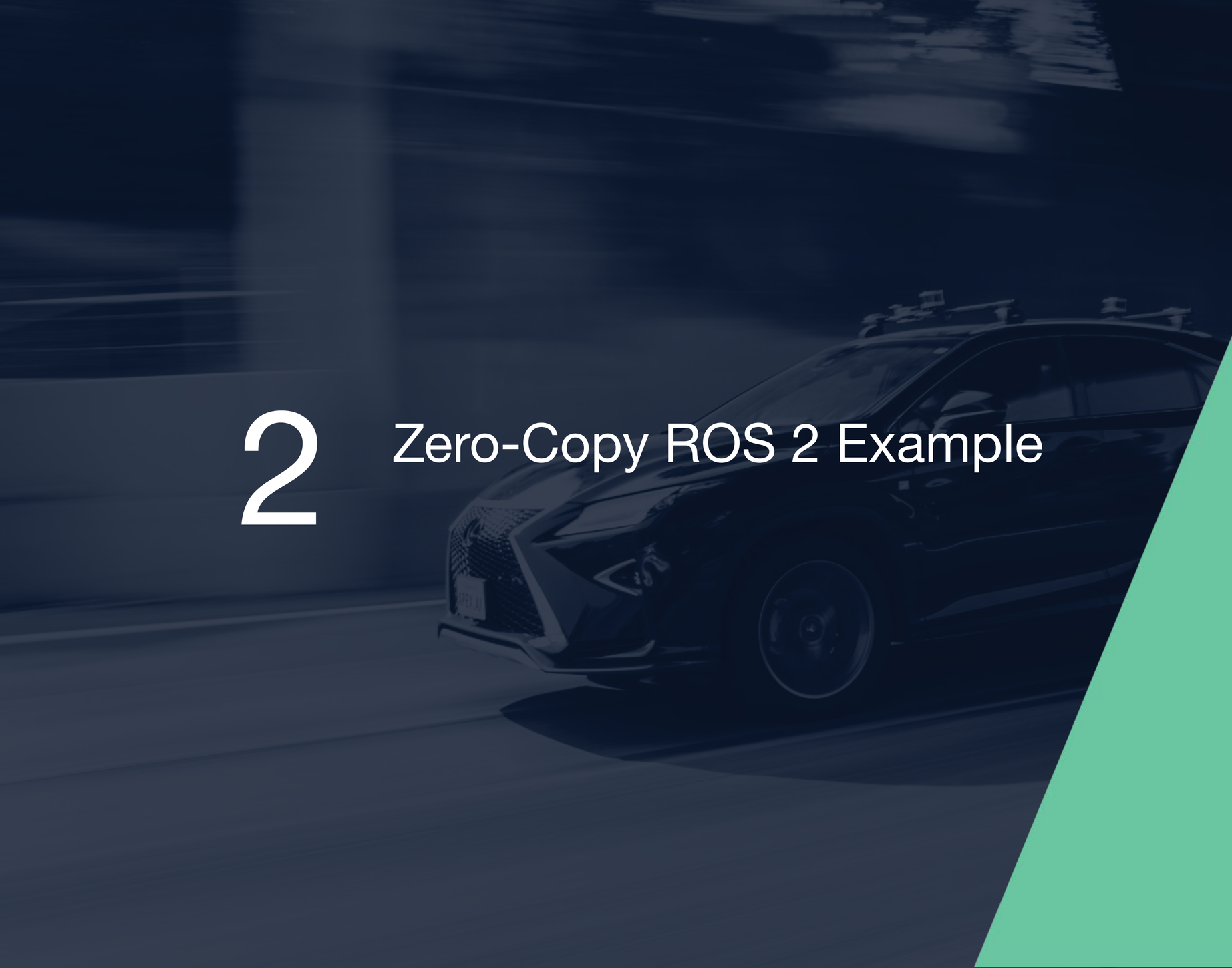
Shared Memory
- can be enabled with Eclipse Cyclone DDS
- uses Eclipse iceoryx middleware
- allows zero-copy data transfer in the intra machine case
- generally much faster than the loopback interface

Limitations
- shared memory transfer is only available for a subset of QoS settings
- only fixed-size ROS message types are currently supported
- if these requirements are not met, network communication is used

ROS 2 Shared Memory Communication



Shared Memory is only used for intra-machine communication



2 Zero-Copy ROS 2 Example

ROS 2 Galactic Installation

General

- install ROS 2 as described in https://docs.ros.org/en/galactic/Installation.html
- source build and installation from package are possible
- installs Eclipse Cyclone DDS and Eclipse iceoryx as well
- the ROS 2 Rolling distribution can also be used

Shared Memory support

- Eclipse Cyclone DDS is built with Shared Memory support by default
- disabled at runtime by default (communication uses the network stack)
- must be enabled in an additional configuration file before running applications
- requires the iceoryx middleware daemon RouDi to be running

Further information

https://github.com/ros2/rmw_cyclonedds/blob/master/shared_memory_support.md

ROS 2 Example - Define a Message

- transmit a string and a counter
- cannot use the string type as it is dynamic (no fixed size)

ShmTopic.msg

```
char[256] data
uint8 size
uint64 counter
uint8 MAX_SIZE=255
```

Fixed size type
All types used by the message have fixed size.

Further information can be found in the ROS 2 tutorials.
https://docs.ros.org/en/galactic/Tutorials/Single-Package-Define-And-Use-Interface.html

- IDL compiler generates C++ type
- read and write access to the members

shm_topic.hpp

```cpp
class ShmTopic {
public:
  std::array<uint8_t, 256> data;
  uint8_t size;
  uint64_t counter;

  static constexpr uint8_t MAX_SIZE = 255u;
  // ...
};
```

This is not the exact generated type but the member interface is usable like this.

ROS 2 Example - Talker

```
#include "rclcpp/rclcpp.hpp"
#include "ros2_shm_demo/msg/shm_topic.hpp"

class Talker : public rclcpp::Node {
public:
  explicit Talker(const rclccpp::NodeOptions
                  &options);

private:
  using Topic = ros2_shm_demo::msg::ShmTopic;

  uint64_t m_count = 1;
  rclcpp::Publisher<Topic>::SharedPtr m_publisher;
  rclcpp::TimerBase::SharedPtr m_timer;

  void populateLoanedMessage(
     rclcpp::LoanedMessage<Topic> &loanedMsg);
};
```

Note: some boilerplate code is omitted

- define the talker node
- topic class is generated by IDL compiler from ShmTopic.msg
- publisher will send topic samples on a 1s timer
- count the samples sent

ROS 2 Example - Publisher

```
Talker(const rclcpp::NodeOptions &options)
  : Node("shm_demo_talker", options) {

  auto publishMessage = [this]() -> void {
    auto loanedMsg =
    m_publisher->borrow_loaned_message();

    populateLoanedMessage(loanedMsg);
    m_publisher->publish(std::move(loanedMsg));
    m_count++;
  };

  rclcpp::QoS qos(rclcpp::KeepLast(10));
  m_publisher =
    this->create_publisher<Topic>("chatter", qos);
  m_timer = this->create_wall_timer(1s,
    publishMessage);
}
```

- create a publisher for the Topic with topic name “chatter”
- set QoS to keep last 10 messages for late subscribers
- define a function publishMessage to be executed periodically on a timer
- obtain a loanedMsg from ROS 2 (request is propagated to the middleware)
- populate loanedMessage with data to publish
- publish loanedMessage

ROS 2 Example - Populate the Message

```
void populateLoanedMessage(rclcpp::LoanedMessage<Topic>
                      &loanedMsg) {
  std::string payload{"Hello World"};

  Topic &msg = loanedMsg.get();
  msg.size = (uint8_t)std::min(payload.size(),
             (size_t)Topic::MAX_SIZE);

  msg.counter = m_count;

  std::memcpy(msg.data.data(), payload.data(),
              msg.size);

  RCLCPP_INFO(this->get_logger(), "Publishing
              %s %lu", payload.c_str(),
              msg.counter);
  }
```

- the payload will in general be variable
- the transmitted data is bounded by MAX_SIZE
- we store the number of bytes of the payload in the message
- since the underlying type of msg.data is an std::array we use memcpy to fill it

Zero-copy?

- ideally the data would be constructed in-place
- current ROS 2 API does not allow that (potential for future optimization)
- we still save copies compared to the network transmission path

ROS 2 Example - Listener

```
#include "rclcpp/rclcpp.hpp"
#include "ros2_shm_demo/msg/shm_topic.hpp"

class Listener : public rclcpp::Node {
public:

    explicit Listener(const rclcpp::NodeOptions
                          &options);

private:
  using Topic = ros2_shm_demo::msg::ShmTopic;

  rclcpp::Subscription<Topic>::SharedPtr
m_subscription;
  char m_lastData[Topic::MAX_SIZE];
};
```

- define the listener node
- node creates a subscription to topic
- arriving data will be received and displayed
- we keep the last data we receive in m_lastData

ROS 2 Example - Subscriber

```
Listener(const rclcpp::NodeOptions &options)
      : Node("shm_demo_listener", options) {

  auto callback = [this](const Topic::SharedPtr msg) {

    std::memcpy(m_lastData, msg->data.data(), msg->size);
    m_lastData[Topic::MAX_SIZE] = '\0';

    RCLCPP_INFO(this->get_logger(), "Received %s %lu",
      m_lastData, msg->counter);
  };

  rclcpp::QoS qos(rclcpp::KeepLast(10));
  m_subscription = create_subscription<Topic>("chatter",
                   qos, callback);
}
```

- define a subscriber callback
  - copy the data to a local buffer and display it
  - since we know the number of bytes used by the payload we only copy those
  - to display the data we would not need to copy
  - in general we can process the arrived message in place
- create a subscriber with compatible QoS which executes the callback when data arrives

Build the Example

The example can be found at https://github.com/ApexAI/ros2_shm_demo

After cloning the repository into the ROS workspace at

```
~/ros2_galactic_ws/src/ros2_shm_demo
```

```
~/ros2_galactic_ws$ colcon build
```

will build the package ros2_shm_demo.

Generated executables:

- talker
- listener
- iox_subscriber

The iox_subscriber is only used for validation of shared memory usage.

Run the Example

We need to enable shared memory support with a configuration file cyclonedds.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<CycloneDDS xmlns="https://cdds.io/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="https://cdds.io/config
https://raw.githubusercontent.com/eclipse-cyclonedds/cy
clonedds/iceoryx/etc/cyclonedds.xsd">
    <Domain id="any">
        <SharedMemory>
            <Enable>true</Enable>
            <SubQueueCapacity>256</SubQueueCapacity>
            <SubHistoryRequest>16</SubHistoryRequest>
            <PubHistoryCapacity>16</PubHistoryCapacity>
            <LogLevel>info</LogLevel>
        </SharedMemory>
    </Domain>
</CycloneDDS>
```

- before starting any ROS 2 applications we need to export the configuration file
- Cyclone DDS will parse the configuration when the ROS 2 application is launched
- configuration options control some aspects of the underlying iceoryx middleware
  - how much data a subscriber can keep
  - how much historical data a subscriber can request at most
  - how much historical data a publisher will keep

Run the Example

Assuming that we already built the talker and listener in the package ros2_shm_demo

In the ROS 2 workspace in all terminals

```
~/ros2_galactic_ws$ source install/setup.bash
```

In different terminals

1. Start the RouDi (Routing and Discovery) middleware daemon

```
~/ros2_galactic_ws$ iox-roudi
```

2. Start the talker

```
~/ros2_galactic_ws$ export CYCLONEDDS_URI=file://$PWD/cyclonedds.xml
~/ros2_galactic_ws$ ros2 run ros2_shm_demo talker
```

3. Start the listener

```
~/ros2_galactic_ws$ export CYCLONEDDS_URI=file://$PWD/cyclonedds.xml
~/ros2_galactic_ws$ ros2 run ros2_shm_demo listener
```

RouDi can be run with specific memory configurations optimized for the specific use case but this is not required here.

Run the Example

How can we verify shared memory is actually used?

1. Measure throughput and latency with and without shared memory enabled
   - if the data is large and sent with high frequency there will be a noticeable difference
2. Use the iceoryx introspection
   - not fully supported and not built with the 1.0 release
   - can be built manually https://github.com/ros2/rmw_cyclonedds/blob/master/shared_memory_support.md
   - will show Shared Memory blocks in use by iceoryx and active iceoryx connections
3. Create a native iceoryx application and subscribe to the ROS 2 topic
   - has to use the ROS 2 generated C++ data type as topic
   - allows to essentially eavesdrop on the ongoing communication
   - will only receive data if shared memory is used
   - provided in the example repository as iox_subscriber

Performance

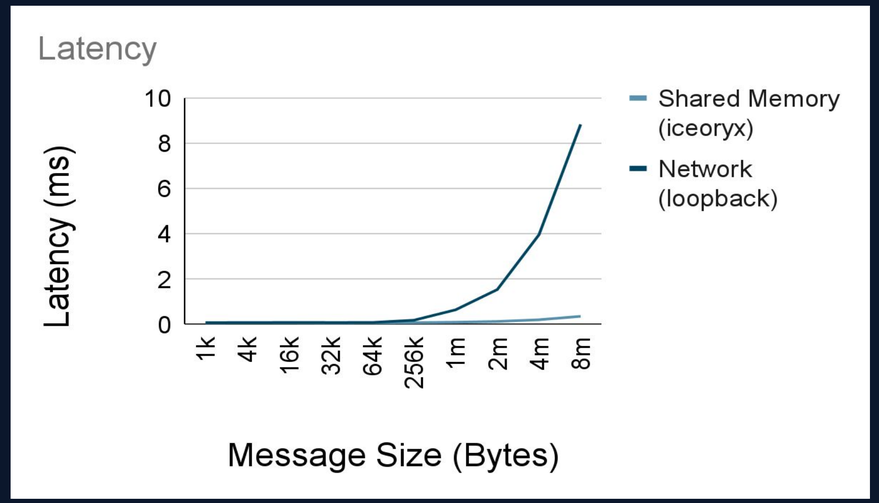
https://gitlab.com/ApexAI/performance_test/

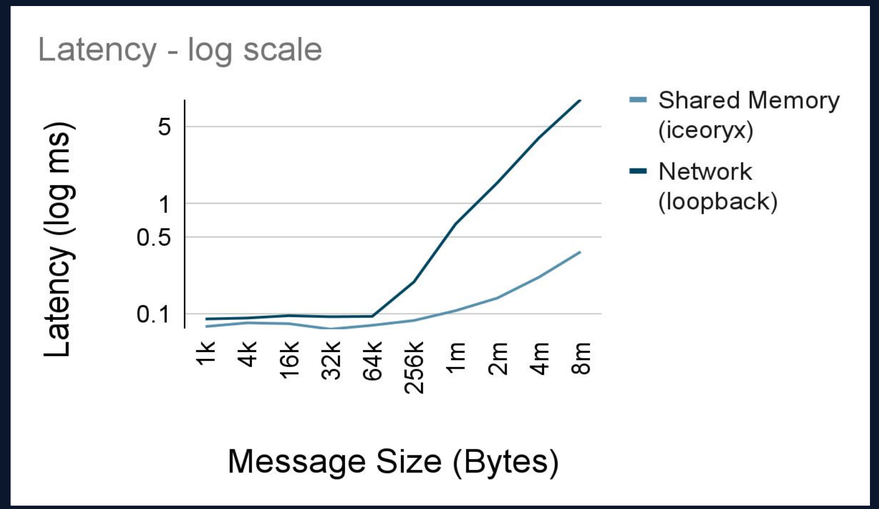
Latency Benchmark

- single publisher and subscriber
- publish in bursts with a given message size
- measure time difference before loaning message to reception of message
- shared memory benefit becomes apparent for sufficiently large message sizes (>64k here), but is never worse than loopback

Further Tests

- demonstrate the benefit even for smaller messages
- advantages for multiple subscribers (data sharing)
- CPU load is generally lower if shared memory is used
  - CPU can be used for other computation or increased throughput
- systematic throughput measurements





Limitations

Using shared memory with iceoryx has some restrictions on QoS and message types.

Reliability

- Reliable or Best-Effort
- due to iceoryx internal caching, data will be lost if the subscriber buffer overflows
  - depends on SubQueueCapacity in the cyclonedds.xml configuration
  - predictable loss of least recent data (FIFO) in the case of overflow

History

- KeepLast n
- `SubHistoryRequest` ≤ `PubHistoryCapacity` in the cyclonedds.xml configuration

Durability

- Volatile
- writer is not required to keep any data for late-joining readers

ROS 2 default settings Reliable, KeepLast 10, Volatile are supported.

Limitations

Using shared memory with iceoryx has some restrictions on QoS and message types.

ROS 2 Message Types
Only fixed-size data types

- primitive types, e.g. char, integers, floats
  - strings are excluded
- fixed size arrays, e.g. char[128]
  - maps to std::array
- structs of fixed-size types
  - which are then themselves fixed-size types and hence recursive nesting is possible

Not supported
- string
  - translated to to the dynamic std::string by the IDL compiler
- bounded arrays, e.g. int32[<=128]
- unbounded arrays, e.g. int32[ ]

Apex.OS

Apex.OS®
Derivative of ROS 2 for use in series development

- rmw and rclcpp layer certified according to ISO26262 ASIL-D
- almost complete subset of ROS 2 functionality
- interoperable with ROS 2 and its tool landscape (e.g. rviz)

Upcoming version 1.3 uses Apex.Middleware™ (Release August 2021)

Apex.Middleware
- middleware implementation based on Eclipse Cyclone DDS and Eclipse iceoryx
- rmw_apex_middleware as interface to Apex.OS
- zero-copy shared memory transfer

Apex.OS 1.3 provides the same benefits as ROS 2 Galactic regarding shared memory transfer.

Apex.OS applications can communicate with ROS 2 Galactic applications via shared memory or network.

Future Work

Supported Types

- support for bounded types
  - easy if memory is less of a concern (optimize for runtime)
- support for unbounded types
  - harder since we only know the size at runtime (need the serialized data size)
  - will in general require a copy/serialization
  - can avoid further copies with shared memory (minimum copy)

Shared Memory Allocators

- currently iceoryx has a (lock-free) pool allocator that requires much configuration for optimal use
- if used suboptimally, we will waste a lot of memory
- improve the allocator performance and make the allocator exchangeable
- support hardware accelerators (GPU, FPGA)

API

- true zero-copy API for ROS 2 and Apex.OS
- reduce overhead in `borrow_loaned_message()`
- add interfaces to other protocols to Apex.Middleware (e.g. SOME/IP)
  - leads to minimum copy interfaces from ROS 2 and Apex.OS to these protocols

References

ROS 2
https://docs.ros.org/en/galactic/Releases/Release-Galactic-Geochelone.html
https://github.com/ros2/rmw_cyclonedds

Shared Memory Usage Example
https://github.com/ApexAI/ros2_shm_demo

Cyclone DDS
https://github.com/eclipse-cyclonedds/cyclonedds

iceoryx
https://github.com/eclipse-iceoryx/iceoryx

Virtual Eclipse Community Meetup
Introducing Eclipse iceoryx Almond (v1.0.0)
https://www.youtube.com/watch?v=giLPYIRBQkw

Apex.OS
https://www.apex.ai/apex-os

Thanks for the Attention

Questions?

See you at EclipseCon
October 2021

Follow the way of the turtle ...

… write robotics applications!