State of ROS 2

Demos and the technology behind

Oct. 3rd 2015
Dirk Thomas, Esteve Fernandez, William Woodall
ROSCon 2015, Hamburg, Germany

Goals of ROS 2

Support multi-robot systems
involving unreliable networks

Remove the gap between
prototyping and final products

"Bare-metal"
micro controller

Support for
real-time control

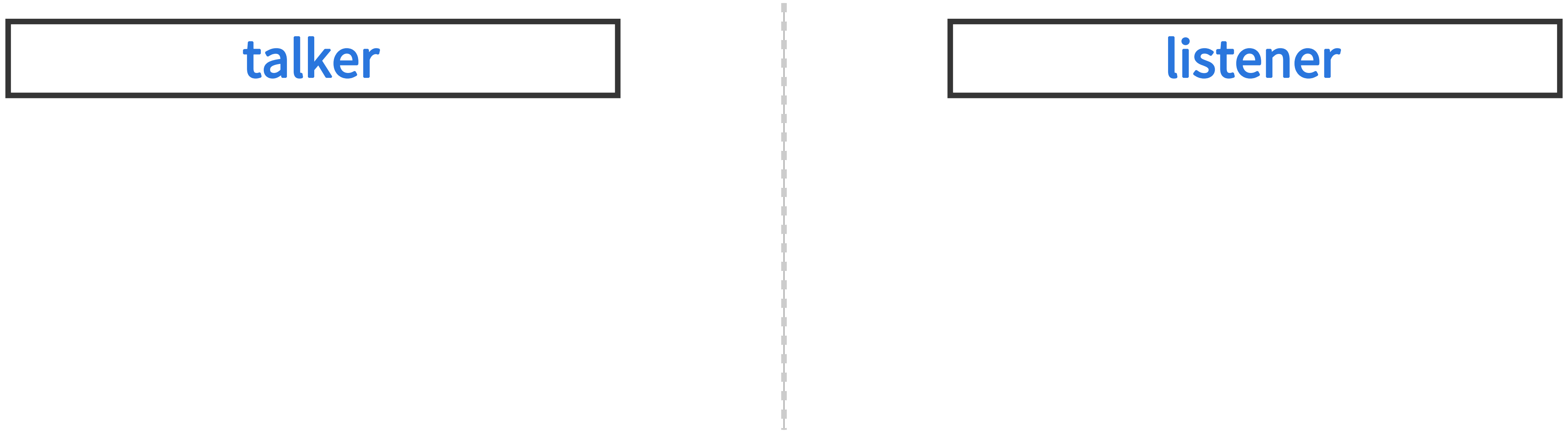
Cross-platform
support

Outline

- Walk through multiple demos
    - https://github.com/ros2/ros2/wiki/Tutorials
- Technical background information

Publish / Subscribe Demo

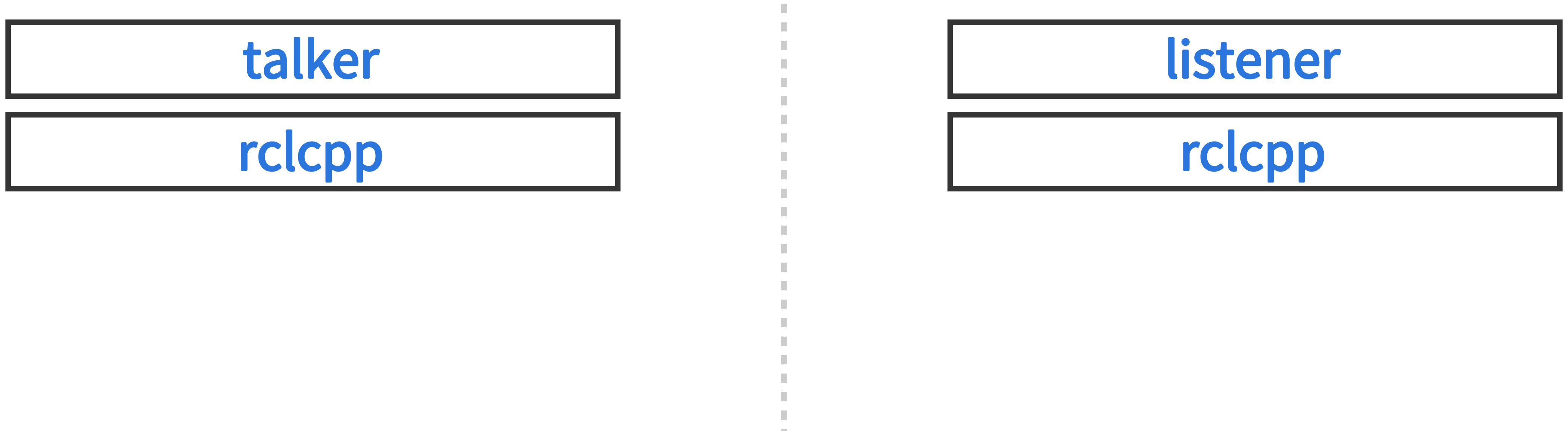
Publish / Subscribe



talker

listener

Publish / Subscribe



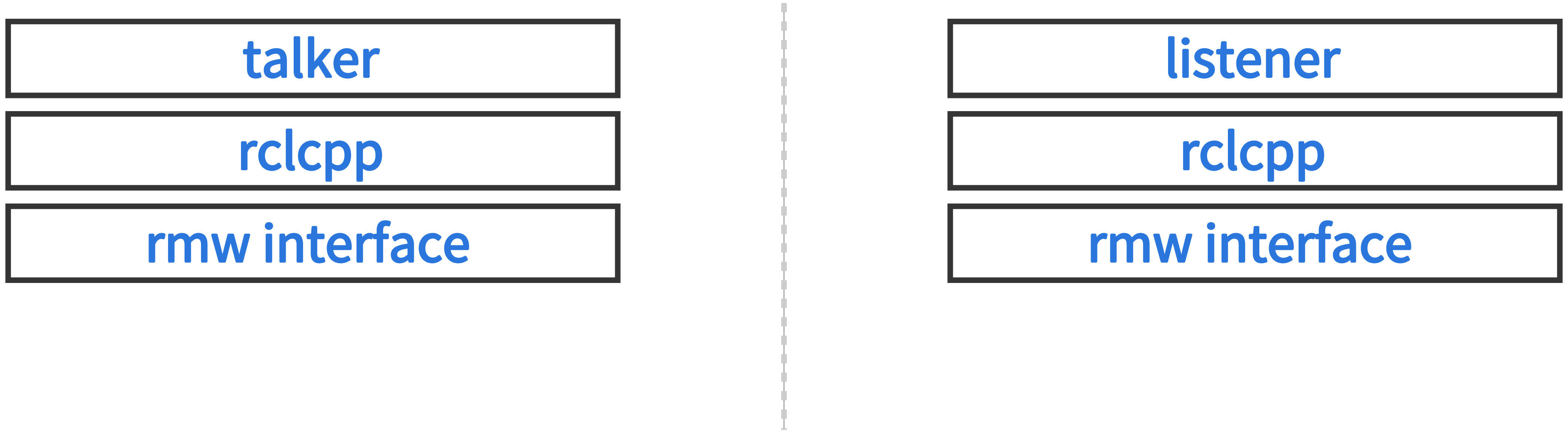
talker

rclcpp

listener

rclcpp

Publish / Subscribe



talker

rclcpp

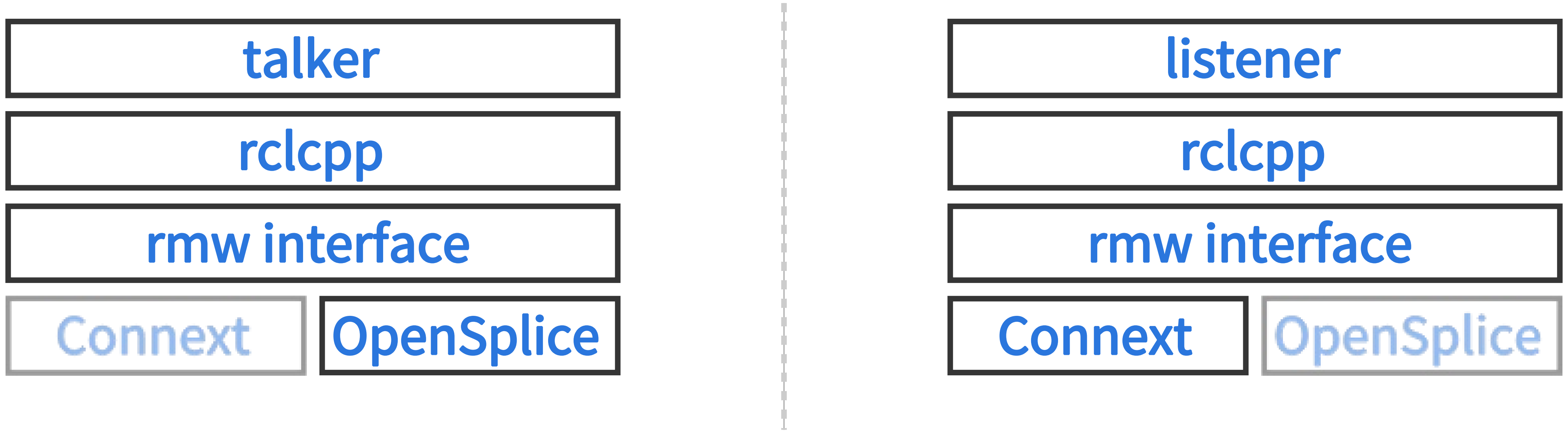
rmw interface

listener

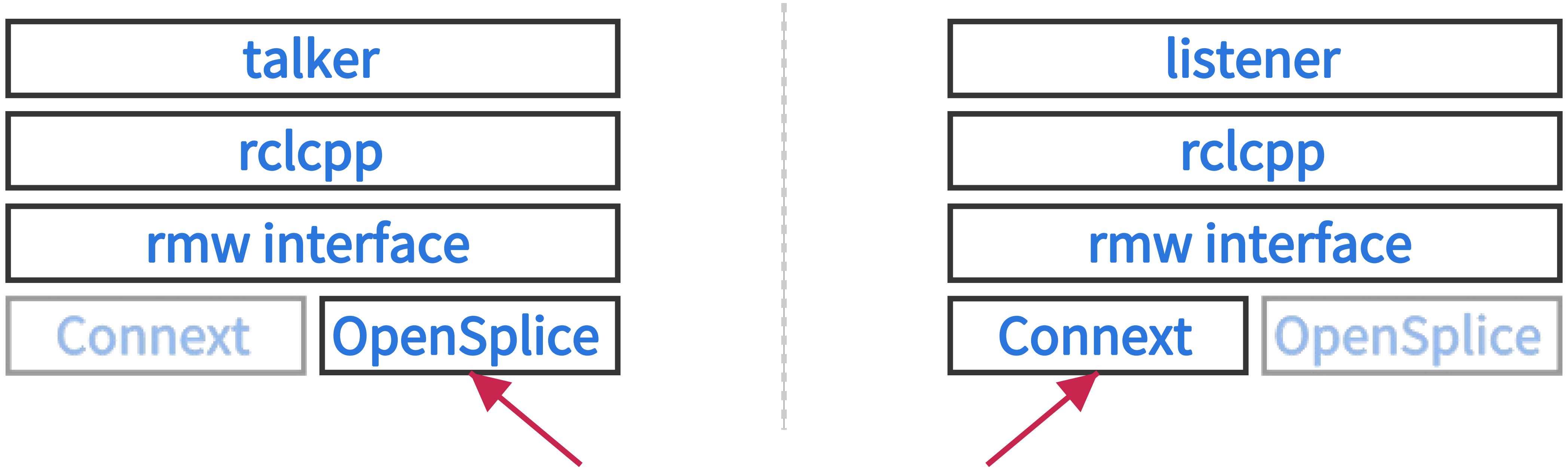
rclcpp

rmw interface

Publish / Subscribe

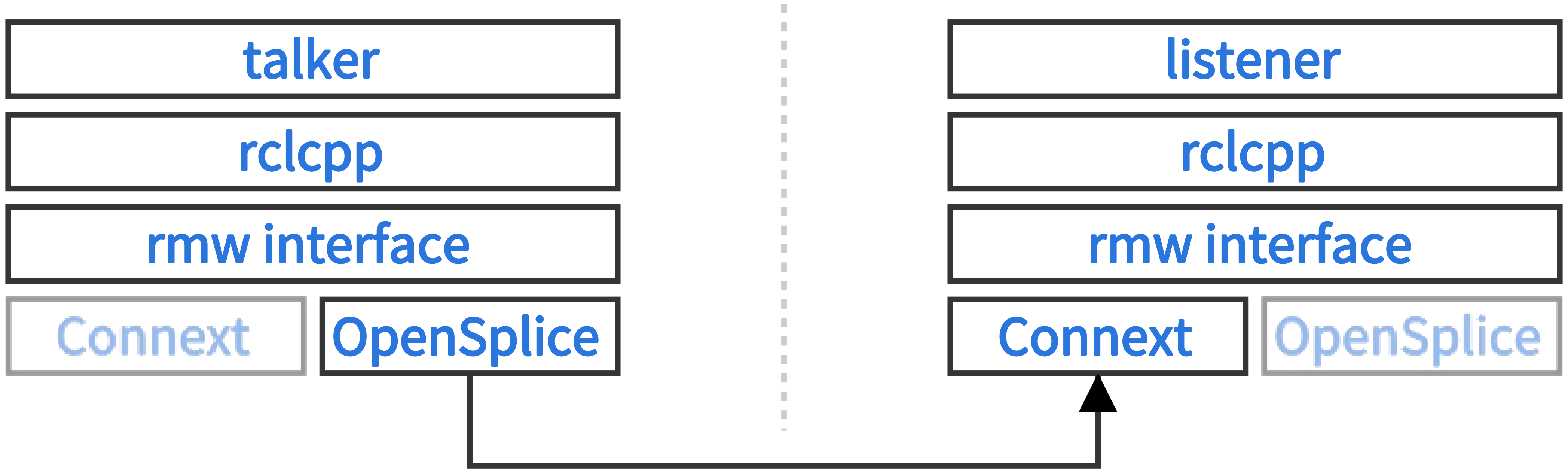


Publish / Subscribe



selected at link-time
of the executables

Publish / Subscribe



DDS Interoperability Wire Protocol

Source code of the *listener* (ROS 1)

```cpp
void callback(const std_msgs::String::ConstPtr & msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char * argv[])
{
  ros::init(argc, argv, "listener");

  ros::NodeHandle node;

  ros::Subscriber sub = node.subscribe("chatter", 10, callback);

  ros::spin();

  return 0;
}
```

Source code of the *listener* (ROS 1)

```
// void callback(const std_msgs::String::ConstPtr & msg)

{
  // ROS_INFO("I heard: [%s]", msg->data.c_str());

}

int main(int argc, char * argv[])
{
  // ros::init(argc, argv, "listener");


  // ros::NodeHandle node;


  // ros::Subscriber sub = node.subscribe("chatter", 10, callback);



  // ros::spin();


  return 0;
}
```

Source code of the *listener* (ROS 2)

```
// void callback(const std_msgs::String::ConstPtr & msg)
void callback(std_msgs::msg::String::ConstSharedPtr msg)
{
  // ROS_INFO("I heard: [%s]", msg->data.c_str());
  printf("I heard: [%s]\n", msg->data.c_str());
}

int main(int argc, char * argv[])
{
  // ros::init(argc, argv, "listener");
  rclcpp::init(argc, argv);

  // ros::NodeHandle node;
  auto node = rclcpp::Node::make_shared("listener");

  // ros::Subscriber sub = node.subscribe("chatter", 10, callback);
  auto sub = node->create_subscription<std_msgs::msg::String>(
    "chatter", rmw_qos_profile_default, callback);

  // ros::spin();
  rclcpp::spin(node);

  return 0;
}
```

C++11 wherever it makes it easier,
the callback can be a lambda.

Source code of the *listener (ROS 2)*

```
// void callback(const std_msgs::String::ConstPtr & msg)
void callback(std_msgs::msg::String::ConstSharedPtr msg)
{
  // ROS_INFO("I heard: [%s]", msg->data.c_str());
  printf("I heard: [%s]\n", msg->data.c_str());
}

int main(int argc, char * argv[])
{
  // ros::init(argc, argv, "listener");
  rclcpp::init(argc, argv);

  // ros::NodeHandle node;
  auto node = rclcpp::Node::make_shared("listener");

  // ros::Subscriber sub = node.subscribe("chatter", 10, callback);
  auto sub = node->create_subscription<std_msgs::msg::String>(
    "chatter", rmw_qos_profile_default, callback);

  // ros::spin();
  rclcpp::spin(node);

  return 0;
}
```

The node's name is passed to the node constructor, not the global init() function.

Source code of the *listener (ROS 2)*

```
// void callback(const std_msgs::String::ConstPtr & msg)
void callback(std_msgs::msg::String::ConstSharedPtr msg)
{
  // ROS_INFO("I heard: [%s]", msg->data.c_str());
  printf("I heard: [%s]\n", msg->data.c_str());
}

int main(int argc, char * argv[])
{
  // ros::init(argc, argv, "listener");
  rclcpp::init(argc, argv);

  // ros::NodeHandle node;
  auto node = rclcpp::Node::make_shared("listener");

  // ros::Subscriber sub = node.subscribe("chatter", 10, callback);
  auto sub = node->create_subscription<std_msgs::msg::String>(
    "chatter", rmw_qos_profile_default, callback);

  // ros::spin();
  rclcpp::spin(node);

  return 0;
}
```

The ~~subscriber is tem~~plated
on the message type.

Source code of the *listener* (ROS 2)

```
// void callback(const std_msgs::String::ConstPtr & msg)
void callback(std_msgs::msg::String::ConstSharedPtr msg)
{
  // ROS_INFO("I heard: [%s]", msg->data.c_str());
  printf("I heard: [%s]\n", msg->data.c_str());
}

int main(int argc, char * argv[])
{
  // ros::init(argc, argv, "listener");
  rclcpp::init(argc, argv);

  // ros::NodeHandle node;
  auto node = rclcpp::Node::make_shared("listener");

  // ros::Subscriber sub = node.subscribe("chatter", 10, callback);
  auto sub = node->create_subscription<std_msgs::msg::String>(
    "chatter", rmw_qos_profile_default, callback);

  // ros::spin();
  rclcpp::spin(node);

  return 0;
}
```

`spin()` is called *on* the node, not globally.

DDS vendors

Company and product name	License	RMW impl.	Comments
RTI Connext	commercial, research	✓	stat. & dyn. impl.
PrismTech OpenSplice	commercial, LGPL	✓	only version 6.4 is LGPL
TwinOaks CoreDX	commercial	–	
eProsima FastRTPS	LGPL	✓	no support for fragmentation yet
OSRF FreeRTPS	Apache 2	partial	small part of DDS only aiming for emb. devices

Transparent Intra-Process Communications

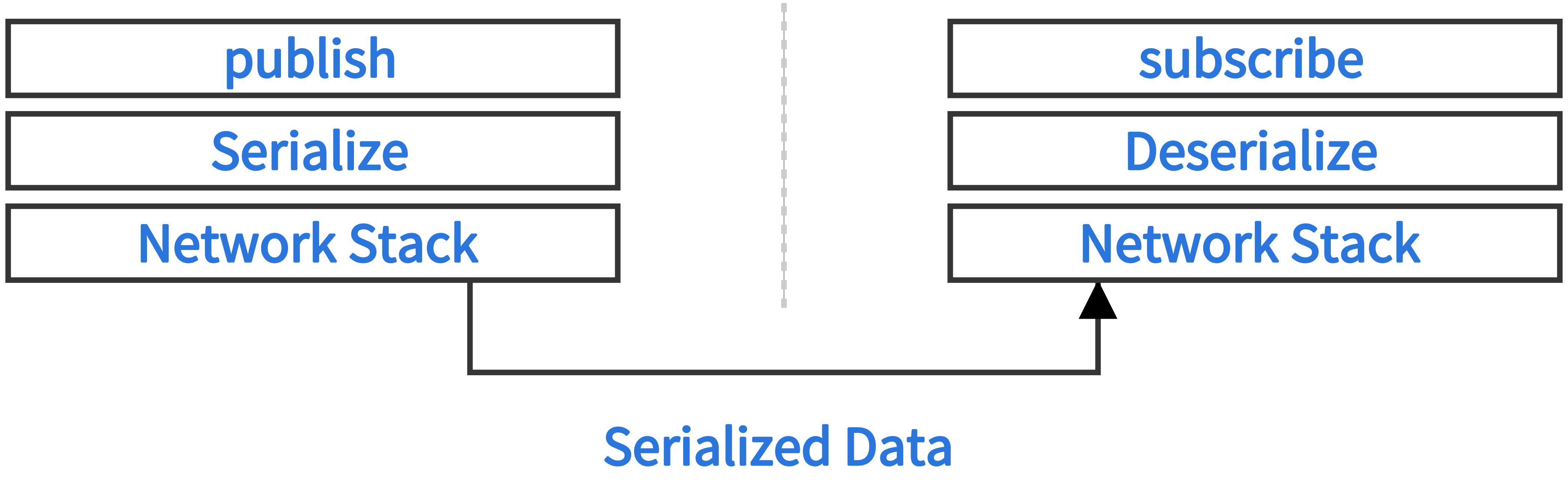
Why support transparent intra-process communications?

- Provide performance improvements for Nodes which:
  - communicate to themselves (pub/sub loop back).
  - communicate with other nodes in the same process.

Related, but previously covered topics:

- ROS 2 Node API will be capable of handling multiple nodes in a single process.

Inter Process Publish / Subscribe



publish

Serialize

Network Stack

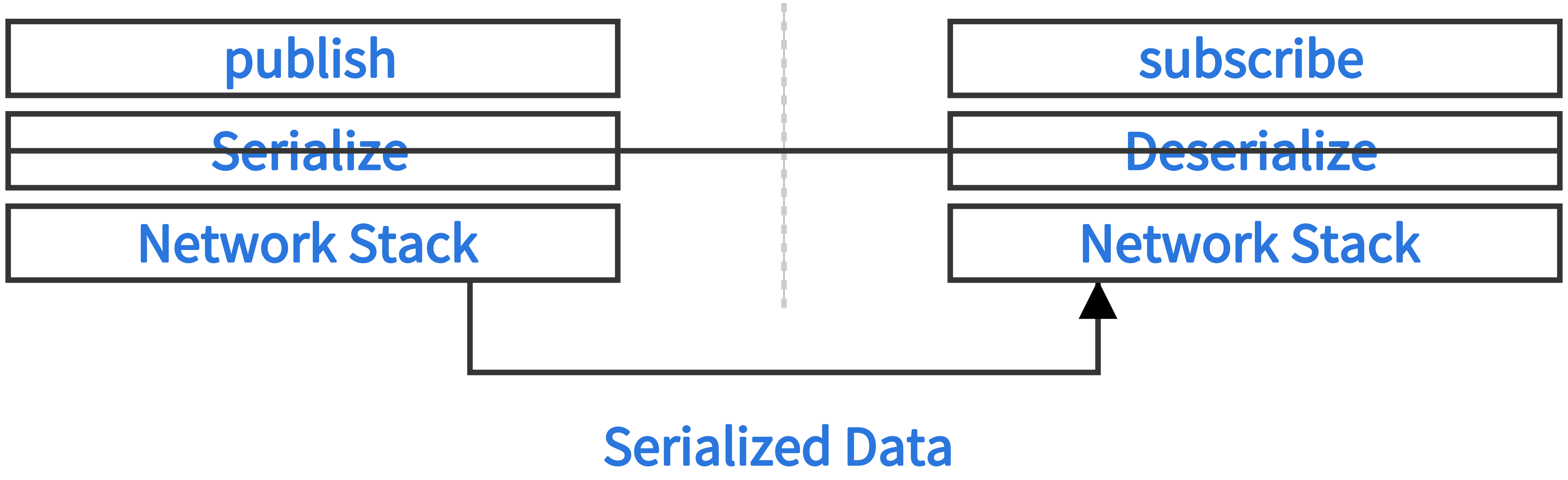
subscribe

Deserialize

Network Stack

Serialized Data

Inter Process Publish / Subscribe



publish

Serialize

Network Stack

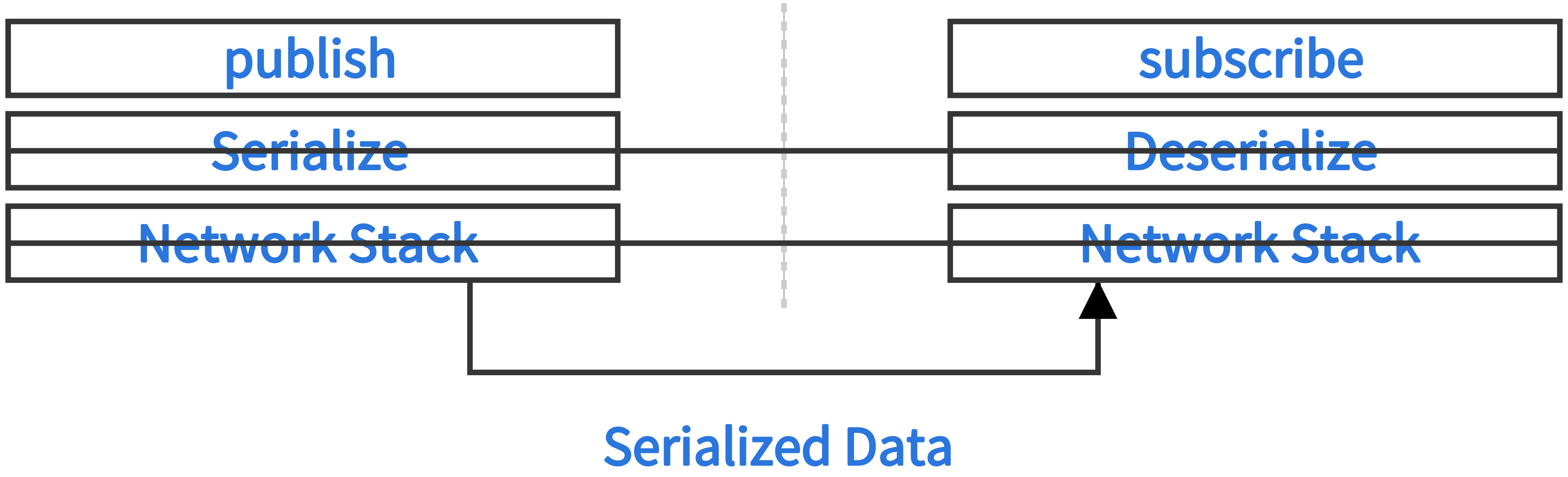
subscribe

Deserialize

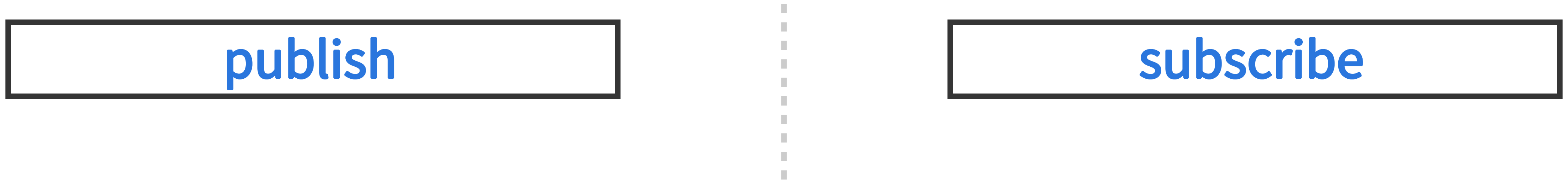
Network Stack

Serialized Data

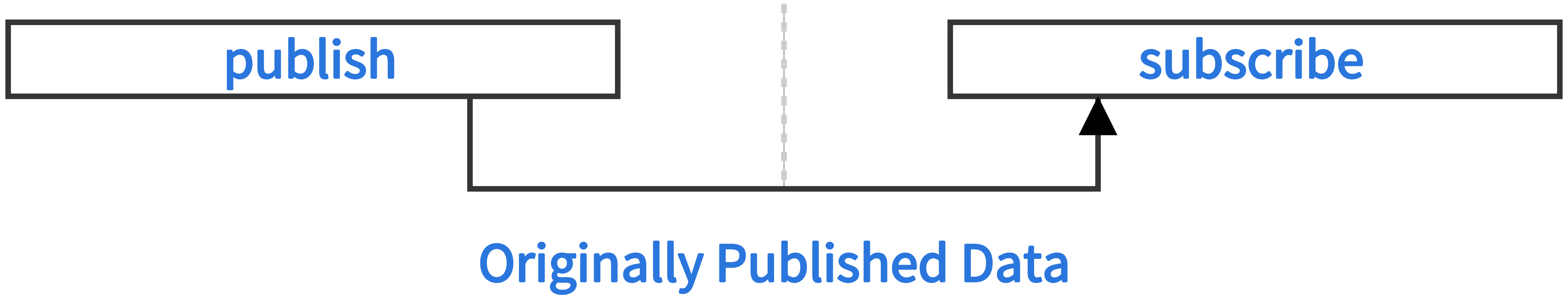
Inter Process Publish / Subscribe



Intra Process Publish / Subscribe



Intra Process Publish / Subscribe



ROS 1

Already Does a Pretty Good Job

Intra-process communication:

- Avoids serialization and deserialization.
- Avoids the network stack (TCP so no userspace packetization).
- Avoids copies, though in an unsafe way:

From http://wiki.ros.org/roscpp/Overview/Publishers%20and%20Subscribers#Intraprocess_Publishing:

- "Note that when publishing in this fashion, there is an implicit contract between you and roscpp: you may not modify the message you've sent after you send it, since that pointer will be passed directly to any intra-process subscribers. If you want to send another message, you must allocate a new one and send that."

Hidden Issues in ROS 1

Consider the case of publishing a `shared_ptr` of a message:

```
std::shared_ptr<std_msgs::msg::String> msg(new std_msgs::msg::String());

msg->data = "testing";
publisher->publish(msg);
// The user still has ownership at this point, could do something like this:
msg->data = "testing2";
```

Will the subscribing callbacks get `"testing"` or `"testing2"`?

- They will get `"testing"` because `publish(...)` actually calls the intra-process callbacks directly.
- Unless they store it and check it later, in which case it might be `"testing2"`.

What happens if the subscriber callbacks run long?

ROS 2

Same Functionality, Safer Patterns

Intra-process communication:

- Avoids serialization and deserialization.
- Avoids the network stack and packetizing of data.
- Safely avoids copies by providing `unique_ptr` based APIs.
- More consistency between intra-process and inter-process communications.

Do Intra-Process Safely

The issues with how ROS 1 does intra-process cause differences between intra-process and inter process behavior. How do we solve this?

- *By tracking ownership with ownership semantics, i.e.* `unique_ptr`

Now consider how a `unique_ptr` works:

```
std::unique_ptr<A> a, b;
a.reset(new A());
// a is valid.
// b is a nullptr.
b = a;  // Ownership of the pointer returned by `new A()` transfered.
// a is now nullptr.
// b is now valid.
```

So when assigning a `unique_ptr` the ownership is traded between them.

Applying unique_ptr to Publish in ROS 2

If applied to publishing:

```
std::unique_ptr<std_msgs::msg::String> msg(new std_msgs::msg::String());

msg->data = "testing";
publisher->publish(msg);  // This is non-blocking, the message goes into a queue.
// The user no longer has access to the message created above.
// Instead the middleware now owns it, and this is no longer valid:
// msg->data = "testing2";  // <-- access nullptr, will cause segmentation fault.
```

The benefit is that the middleware did not need to make a copy, but the user is not able to accidentally change the data they relinquished.

- But it is not always the optimal solution, e.g. if you are reusing messages intentionally.

The Subscribing Side

What about the subscribing side of the problem? A typical example first:

```
void callback(std_msgs::msg::String::ConstSharedPtr msg)
{
  // msg->data = "new value"; This is illegal; the user doesn't own it.
  std_msgs::msg::String msg_copy(*msg);  // Must make a copy that the user owns.
  msg_copy = "new value";
  outgoing_publisher->publish(msg_copy);
}
```

The middleware does not give the user ownership because it may need to give the same shared message to another callback.

- Result: the user needs to copy it explicitly in order to modify it.

Using unique_ptr on the Subscribe Side

If you use a `unique_ptr` in the callback signature, it looks like this:

```
void callback(std_msgs::msg::String::UniquePtr msg)
{
  msg->data = "new value";  // Edit directly; middleware relinquished ownership.
  outgoing_publisher->publish(msg);
}
```

The middleware will make a copy if there are other callbacks, so:

- This does not avoid any extra copies, but can simplify your code if you are going to copy it anyways.
- In one special case it can avoid a copy: if this is the only intra-process callback.

Demo Cyclic Pipeline

Demo Cyclic Pipeline

Full text: https://github.com/ros2/demos/blob/release-alpha1/intra_process_demo/src/cyclic_pipeline/cyclic_pipeline.cpp

```
struct IncrementerPipe : public rclcpp::Node
{
  IncrementerPipe(const std::string & name, const std::string & in, const std::st
ring & out)
  // ...
      [this](std_msgs::msg::Int32::UniquePtr & msg) {
        printf("Received message with value:         %d, and address: %p\n",
          msg->data, msg.get());
        printf("  sleeping for 1 second...\n");
        if (!rclcpp::sleep_for(1_s)) {
          return;  // Return if the sleep failed (e.g. on ctrl-c).
        }
        printf("  done.\n");
        msg->data++;  // Increment the message's data.
        printf("Incrementing and sending with value: %d, and address: %p\n",
          msg->data, msg.get());
        this->pub->publish(msg);  // Send the message along to the output topic.
      });
  }
  // ..
};
```

Demo Cyclic Pipeline

Running two instances:

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

  auto pipe1 = std::make_shared<IncrementerPipe>("pipe1", "topic1", "topic2");
  auto pipe2 = std::make_shared<IncrementerPipe>("pipe2", "topic2", "topic1");
  // ..
  // Publish the first message (kicking off the cycle).
  std::unique_ptr<std_msgs::msg::Int32> msg(new std_msgs::msg::Int32());
  msg->data = 42;
  printf("Published first message with value:  %d, and address: %p\n",
    msg->data, msg.get());
  pipe1->pub->publish(msg);

  executor.add_node(pipe1);
  executor.add_node(pipe2);
  executor.spin();
  return 0;
}
```

See https://github.com/ros2/ros2/wiki/Intra-Process-Communication#the-image-pipeline-demo

Using unique_ptr

So what can we say about these new ownership semantics:

- Can be used to create efficient pipelines, i.e. chains of 1 to 1 pub/sub.
- But cannot rely on the published pointer to be received by callback.
- Not always the preferred signature, since you may want to reuse published `shared_ptr`'s.

Domains where this matters:

- Using pub/sub within a high performance perception algorithm.
- Systems where every `memcpy` costs battery life or latency.

Consistent Behavior between Inter and Intra

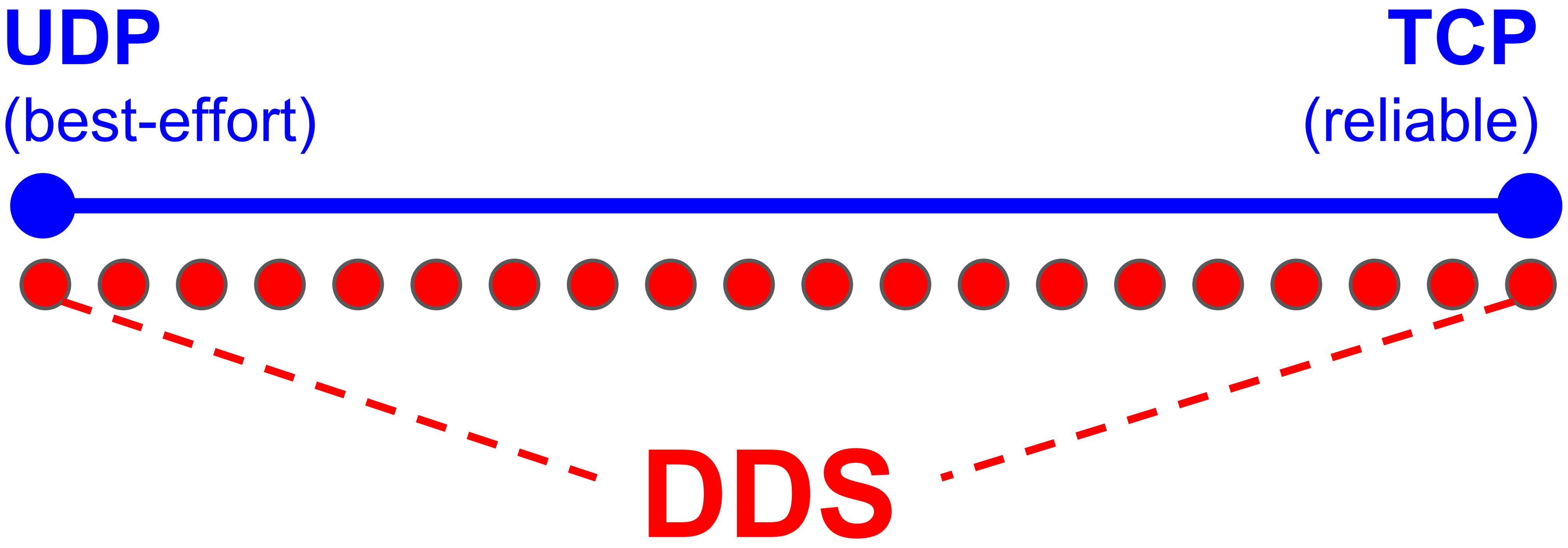
How does ROS 2 make publishing more similar in these two cases?

- Intra-process callbacks are handled outside of the user's callback.
- More middleware QoS and queueing behavior's are supported by intra-process.

What's Next?

- Make the intermediate intra-process storage smarter (intelligently convert when beneficial).
- Consider alternative implementations of the intra-process system (internally).
- Allow better control of memory allocation and test for real-time safety.
- Implement Type Masquerading.
- Building and Running a Node:
    - Remove the boilerplate, make it easy to write once then choose stand-alone versus shared process later.

The networking spectrum



UDP

(best-effort)

TCP

(reliable)

DDS

Some of the QoS settings

ROS1: **UDPROS/TCPROS** ROS2: **Reliability**

- `Best effort`: messages arrive “on time“ at the expense of losing some
- `Reliable`: all messages must reach the other end

ROS1: **Queueing** ROS2: **History**

- `Keep last`: only store *N* messages, configurable with queue depth option
- `Keep all`: store all messages

ROS1: **Latching** ROS2: **Durability**

- `Volatile`: no persistence
- `Transient local`: durable data is maintained by the writer

Much richer spectrum of QoS capabilities with ROS2

DDS provides QoS “for free”

- Industry-proven QoS strategies
    - Extensive DDS documentation
    - Shared knowledge
    - Frees us from implementing a complex custom solution

- Using UDP (instead of TCP) allows multicasting
    - Publisher won't have to transmit extra copies of a message to every subscriber

- Support unreliable networks, e.g. drones, IoT, high latency links

Quality of Service Demo

Quality of Service Demo

QoS profiles

```
typedef struct RMW_PUBLIC_TYPE rmw_qos_profile_t
{
  enum rmw_qos_history_policy_t history;
  size_t depth;
  enum rmw_qos_reliability_policy_t reliability;
  enum rmw_qos_durability_policy_t durability;
} rmw_qos_profile_t;
```

Predefined profiles

- sensor data
- services
- parameters

Integration with existing DDS deployments

- every policy has a “system default“ option
- optionally use DDS vendor tools to define QoS settings and profiles
- do not disrupt existing DDS deployments

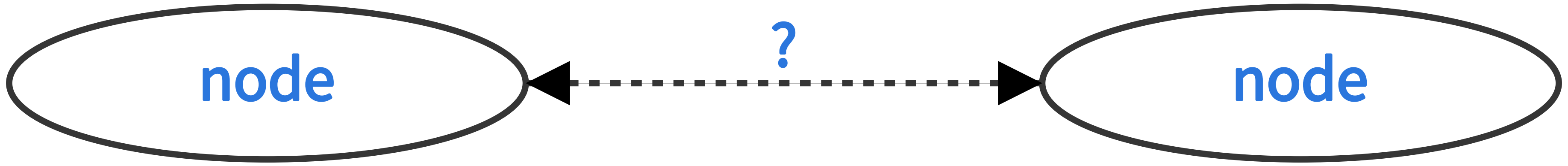
Bridging between ROS versions

ROS 2

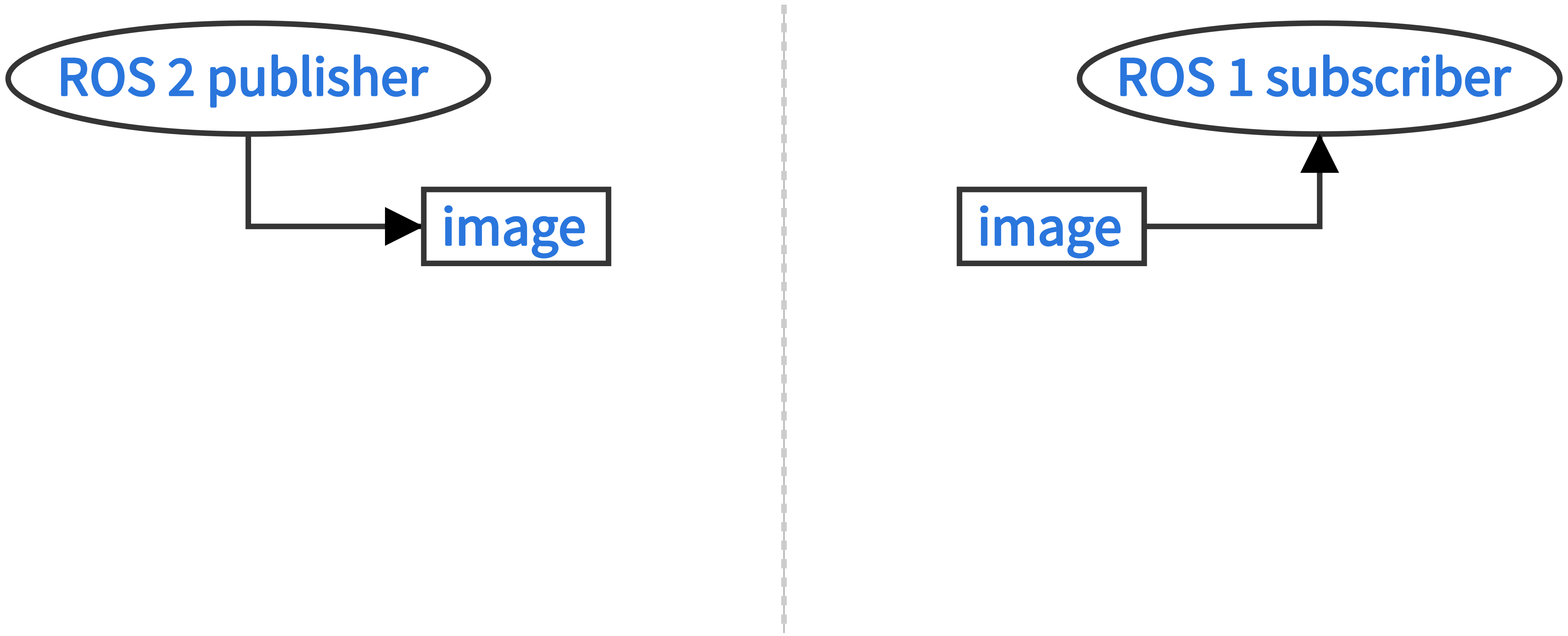
- New features
- Superior communication

ROS 1

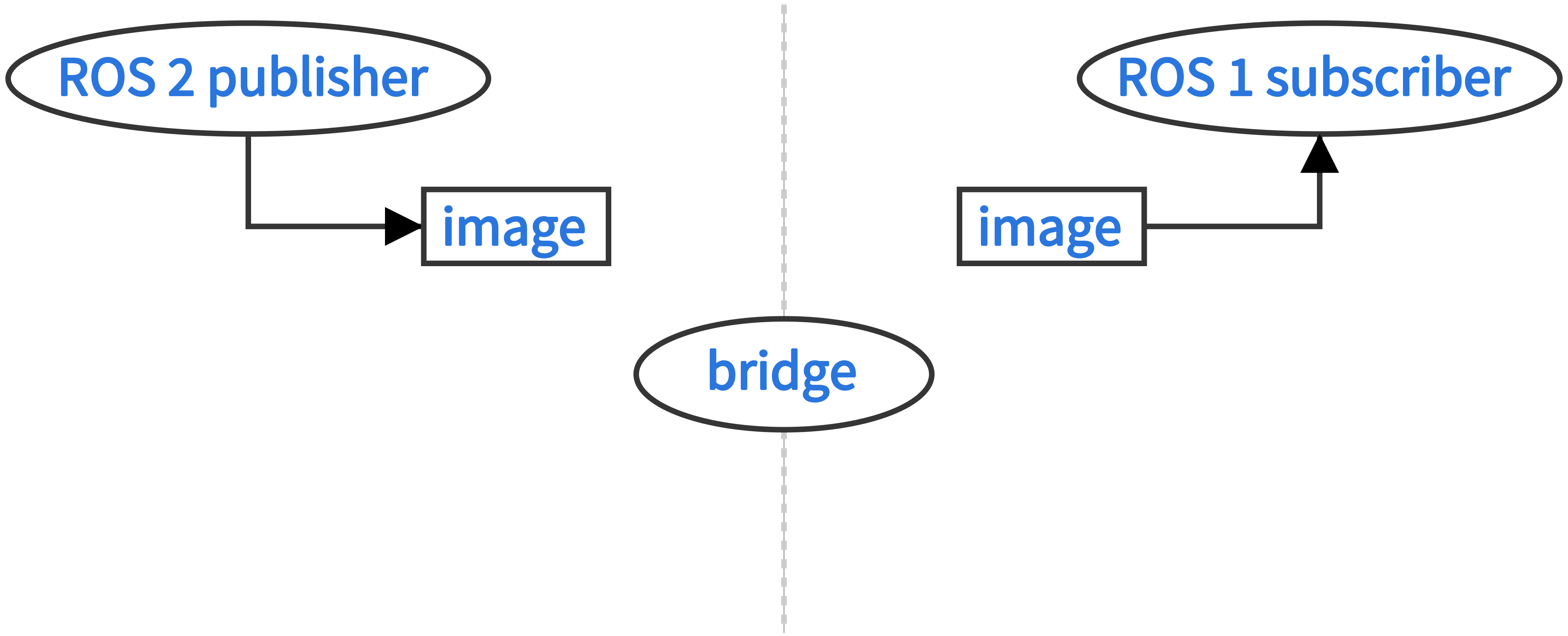
- Plenty of tools
- Existing funtionality



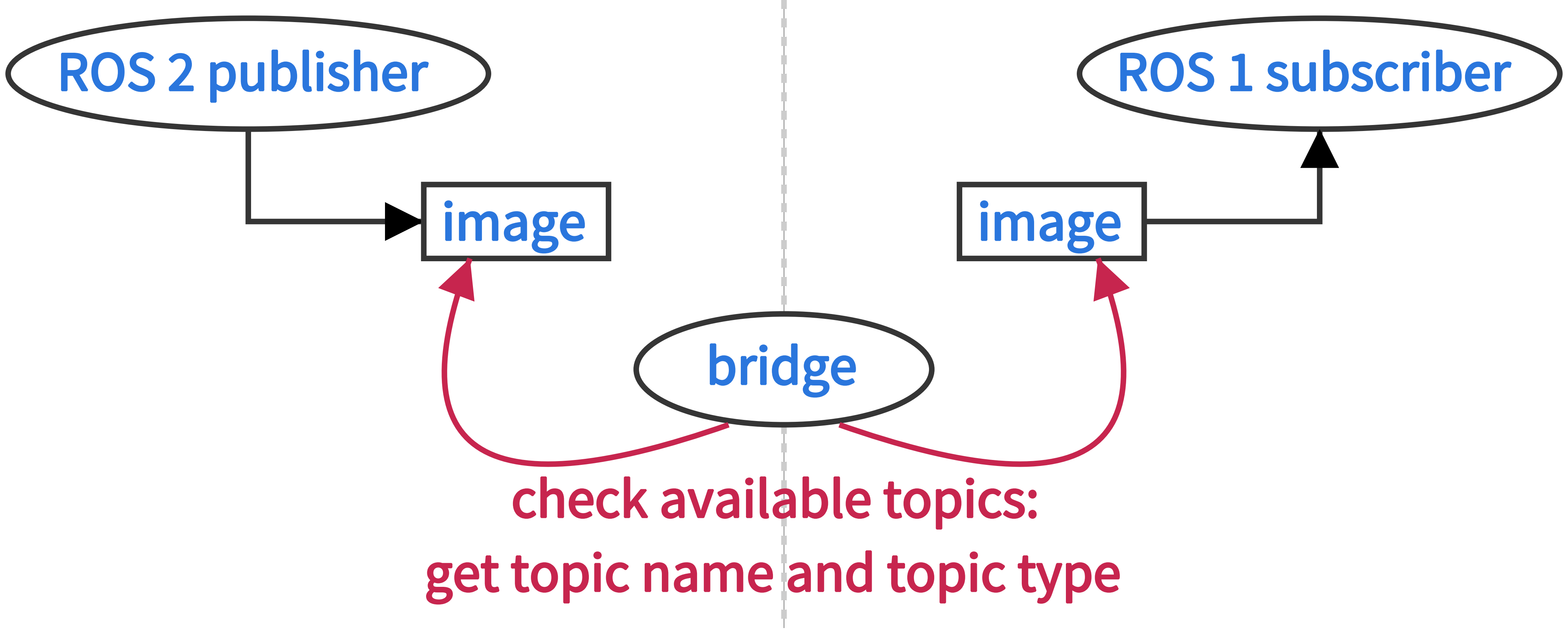
Dynamic Bridge



Dynamic Bridge

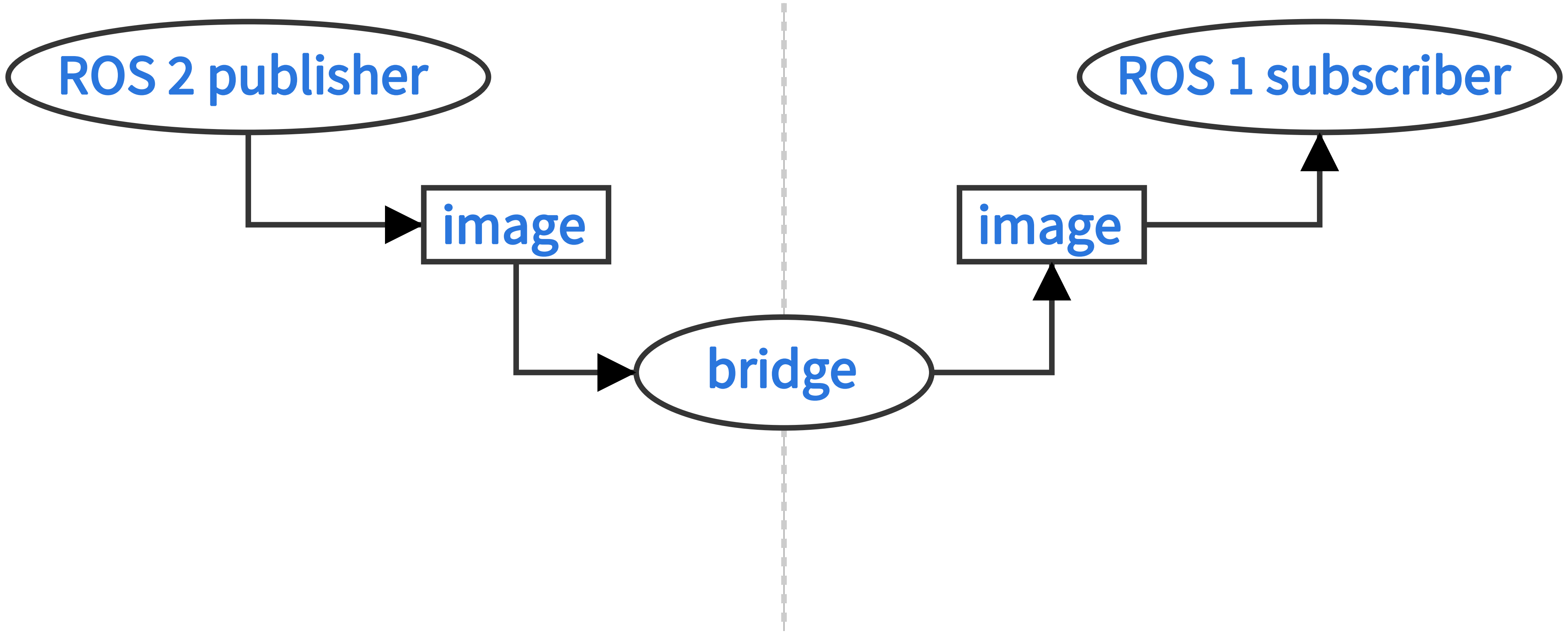


Dynamic Bridge



check available topics:
get topic name and topic type

Dynamic Bridge



Bridge Demo

Technical background

- Currently implemented in C++
- Find all message definitions
  - in ROS 1 using the `rosmsg` API (crawls the FS ☹)
  - in ROS 2 using the `ament resource index` (no crawling 😎)
    https://github.com/ament/ament_cmake/blob/master/ament_cmake_core/doc/resource_index.md
- Generate mappings between ROS 1 types and ROS 2 types
  - automatic rules
  - optionally: custom rules (specified in a `.yaml` file)
  - ∀ type pairs
    - register at a factory
    - generate conversion functions (ROS 1 ↔ ROS 2)
- Build the bridge
  - use `roscpp` found via `pkg-config`
  - use `rclcpp` found via `CMake find_package()`
- `Challenge`: all header files must be non-colliding (!)

Roadmap

- First release (`Alpha 1`) was on Sep. 1st
    - https://github.com/ros2/ros2/wiki/Alpha1-Overview

Roadmap

- First release (`Alpha 1`) was on Sep. 1st
  - https://github.com/ros2/ros2/wiki/Alpha1-Overview
- Upcoming features, grouped and ordered
  - https://github.com/ros2/ros2/wiki/Roadmap

Roadmap

- First release (`Alpha 1`) was on Sep. 1st
    - https://github.com/ros2/ros2/wiki/Alpha1-Overview
- Upcoming features, grouped and ordered
    - https://github.com/ros2/ros2/wiki/Roadmap
- Current work items for `Alpha 2`
    - Component life cycle
        - Introspection and orchestration APIs
        - Using `class_loader` / `pluginlib`
    - Launch system
        - Using life cycle and orchestration
    - Continue work on ROS client libraries
        - Solve technical challenges in C++
        - C as well as Python API

Related presentations

- **ROS 2 on "small" embedded systems**
  - already presented in the morning by *Morgan*
- **Real-time Performance in ROS 2**
  - upcoming presentation from *Jackie* and *Adolfo*

Questions...

For more information go to:
www.ros2.org