Open Source Robotics Foundation

# Why You Want to Use ROS 2

Sep 12th, 2014

Esteve Fernandez, Tully Foote, Dirk Thomas, *William Woodall*

ROSCon Chicago


# Why You Want to Use ROS 2

http://goo.gl/NgG1kK

# What is ROS 2?

ROS 2 is...

- Not going to break ROS 1, and will not be rolled out into ROS 1.
- Breaking API with ROS 1, but conceptually very similar.
- A chance to reevaluate design decisions.
- A chance to use more modern dependencies and tools.
- A chance to change fundamental parts of the system.
- Going to interoperate with ROS 1 (probably with a bridge).
- Going to continue using the ROS 1 .msg IDL format (genmsg).
- Embracing C++11 and Python3.
- Going to have a full feature C API.
- Will use DDS as the middleware.
- Aiming for real time, embedded, and Windows.
- Built on standards, like DDS's RTPS and X-Types.
- Gonna be awesome.

# What's the current status of ROS 2?

Decisions Made:

- DDS will be used as the middleware.
- Switching DDS vendors will be supported by default.
- ROS 1's .msg file IDL will continue to be used to define interfaces.

Existing Capabilities:

- Node's, Publisher's, Subscription's, WallTimer's, and WallRate's.
- Switch DDS vendors at link time (ABI compatible implementations).
- Nodes in separate processes communicate using different vendors.
  - i.e. talker with Connext sends to listener with OpenSplice
- Multiple Nodes in one process.
- Control over callback concurrency.

Examples:

- https://github.com/ros2/examples

# Overview

- Component Style API
  - Example: Talker node
- ROS 1 like API
  - Example: Listener node
- Improvements to “nodelet” style development
  - Unifying “node” and “nodelet” API
  - Improve developer workflow and convenience
  - New tools for managing concurrent code
- Dynamic Graph Features
  - Topic remapping and aliasing
  - Changing graph layout during runtime
- Node lifecycle management
  - Verifiable systems
  - Live relocatable nodes

# Jumping right in: What does it look like?

```
class Talker : public Node {
public:
  Talker() : Node("talker"), count_(0)
  {
    pub_ = this->create_publisher<std_msgs::String>("chatter", 10);
    timer_ = this->create_wall_timer(10_hz, &Talker::on_timer, this);
  }
  void on_timer()
  {
    auto msg = pub_->create_shared_message();
    msg->data = "Hello World: " + std::to_string(count_++);
    pub_->publish(msg);
  }
private:
  size_t count_;
  Publisher::SharedPtr pub_;
  WallTimer::SharedPtr timer_;
};
```

# Jumping right in: What does it look like?

```
class Talker : public Node {
public:
  Talker() : Node("talker"), count_(0)
  {
    pub_ = this->create_publisher<std_msgs::String>("chatter", 10);
    timer_ = this->create_wall_timer(10_hz, &Talker::on_timer, this);
  }
  void on_timer()
  {
    auto msg = pub_->create_shared_message();
    msg->data = "Hello World: " + std::to_string(count_++);
    pub_->publish(msg);
  }
private:
  size_t count_;
  Publisher::SharedPtr pub_;
  WallTimer::SharedPtr timer_;
};
```

Inherits from Node

Delegate name to Node constructor

# Jumping right in: What does it look like?

```
class Talker : public Node {
public:
  Talker() : Node("talker"), count_(0)
  {
    pub_ = this->create_publisher<std_msgs::String>("chatter", 10);
    timer_ = this->create_wall_timer(10_hz, &Talker::on_timer, this);
  }
  void on_timer()
  {
    auto msg = pub_->create_shared_message();
    msg->data = "Hello World: " + std::to_string(count_++);
    pub_->publish(msg);
  }
private:
  size_t count_;
  Publisher::SharedPtr pub_;
  WallTimer::SharedPtr timer_;
};
```

Inherits from Node

Delegate name to Node constructor

C++11: Using auto

C++11: User defined literals

C++11: Usages of std::shared_ptr's

# What if I want to write my own main?

*ROS 1 version in comments.*

```
// void chatterCallback(const std_msgs::String::ConstPtr& msg)
void chatterCallback(const std_msgs::String::ConstPtr &msg)
{
  // ROS_INFO("I heard: [%s]", msg->data.c_str());
  std::cout << "I heard: [" << msg->data << "]" << std::endl;
}

int main(int argc, char *argv[])
{
  // ros::init(argc, argv, "listener");
  rclcpp::init(argc, argv);
  // ros::NodeHandle n;
  auto node = rclcpp::Node::make_shared("listener");

  // ros::Subscriber sub = n.subscribe("chatter", 7, chatterCallback);
  auto sub = node->create_subscription<std_msgs::String>("chatter", 7, chatterCallback);

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

  return 0;
}
```

# What if I want to write my own main?

```
// void chatterCallback(const std_msgs::String::ConstPtr& msg)
void chatterCallback(const std_msgs::String::ConstPtr &msg)
{
  // ROS_INFO("I heard: [%s]", msg->data.c_str());
  std::cout << "I heard: [" << msg->data << "]" << std::endl;
}

int main(int argc, char *argv[])
{
  // ros::init(argc, argv, "listener");
  rclcpp::init(argc, argv);
  // ros::NodeHandle n;
  auto node = rclcpp::Node::make_shared("listener");

  // ros::Subscriber sub = n.subscribe("chatter", 7, chatterCallback);
  auto sub = node->create_subscription<std_msgs::String>("chatter", 7, chatterCallback);

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

  return 0;
}
```

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

# What if I want to write my own main?

```
// void chatterCallback(const std_msgs::String::ConstPtr& msg)
void chatterCallback(const std_msgs::String::ConstPtr &msg)
{
  // ROS_INFO("I heard: [%s]", msg->data.c_str());
  std::cout << "I heard: [" << msg->data << "]" << std::endl;
}

int main(int argc, char *argv[])
{
  // ros::init(argc, argv, "listener");
  rclcpp::init(argc, argv);
  // ros::NodeHandle n;
  auto node = rclcpp::Node::make_shared("listener");

  // ros::Subscriber sub = n.subscribe("chatter", 7, chatterCallback);
  auto sub = node->create_subscription<std_msgs::String>("chatter", 7, chatterCallback);

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

  return 0;
}
```

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

# What if I want to write my own main?

```
// void chatterCallback(const std_msgs::String::ConstPtr& msg)
void chatterCallback(const std_msgs::String::ConstPtr &msg)
{
  // ROS_INFO("I heard: [%s]", msg->data.c_str());
  std::cout << "I heard: [" << msg->data << "]" << std::endl;
}

int main(int argc, char *argv[])
{
  // ros::init(argc, argv, "listener");
  rclcpp::init(argc, argv);
  // ros::NodeHandle n;
  auto node = rclcpp::Node::make_shared("listener");

  // ros::Subscriber sub = n.subscribe("chatter", 7, chatterCallback);
  auto sub = node->create_subscription<std_msgs::String>("chatter", 7, chatterCallback);

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

  return 0;
}
```

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

spin is called ***on*** the node, not globally.

# Why do non-global nodes matter?

This subtle shift in the API addresses a fundamental issue with the way ROS 1 handles nodes, which primarily affects how the ROS 1 concept of nodelets will be manifested in ROS 2.

In ROS 2:

- There will be no global state tracking what nodes are in a process.
  - This is why the node was passed to *spin*.
- There will be no distinction between “nodes” and “nodelets”.
  - Actually “nodelets” won’t even be a concept.
- There can be zero to many nodes in a single process.
- Nodes in a single process can communicate through shared pointers.

# How does this make things better?

Once the concept of multiple nodes sharing a single process is a first class concept in ROS, we can start to identify ways to make our code much more efficient.

Nodelets in ROS 1 exposed some of these ideas, like shared pointer passing over intra-process topics and sharing threads amongst many nodelets.

But notably nodelets had some issues, here are few:

- The API was different between nodes and nodelets
- Creating and running Nodelets was complicated
- Sharing threads lead to starvation due to user locking

# How does ROS 2 unify the API's?

**The API was different between nodes and nodelets:**

This issue is addressed directly by making nodes not global, and something that must be registered with the system in order to be executed (the ROS 2 nodes share this with nodelets in ROS 1).

Because of this, whether you are inheriting from the Node class and calling functions like `this->create_publisher<...>(...)` in order to create a publisher, or you are operating directly on a node you created by calling functions like `node->create_publisher<...>(...)`, the API is the same.

In this way the procedural, or "don't wrap my main", style API and the component, or "nodelet", style API are no longer at odds.

# How does ROS 2 make using Nodes easier?

**Creating and running Nodelets was complicated:**
In order to improve this process, new CMake macros have been created, like rclcpp_create_node(my_node src/my_node.cpp ...), which:

- creates a **shared library**, *libmy_node.so*
  - this contains the implementation of your node
- creates an **executable**, *my_node*
  - By default runs your node in its own process
  - Optionally, runs your node in a process with other nodes
- registers your node, so that other tools can find it

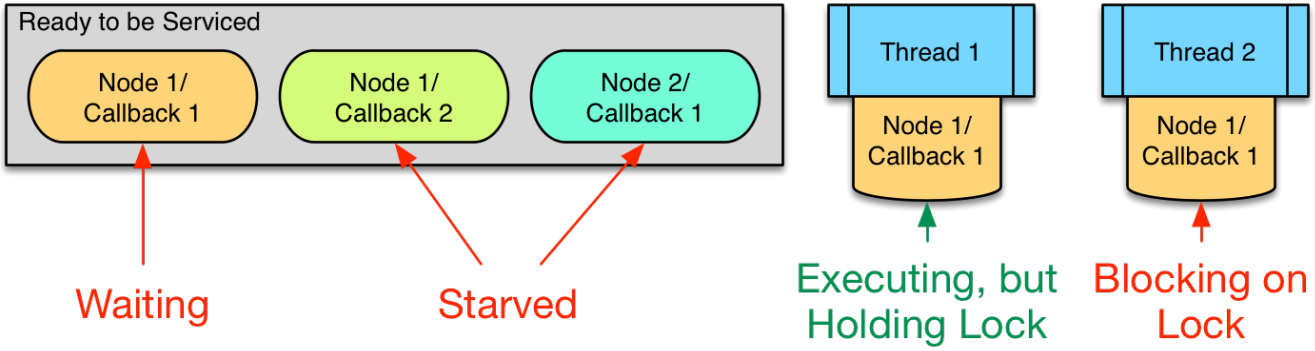
The **executable** will act as a proxy process while it is running in the other process. In this way the **executable** is the single point of entry for your node.

# What about thread-pool starvation?

**Sharing threads lead to starvation due to user locking:**

Nodelet's make it difficult to use a shared thread-pool efficiently:

- Multi-Threaded nodelet + User locks = Blocked thread-pool threads




- Single-Threaded nodelet = One (non thread-pool) thread per nodelet
  - This is not a very efficient use of resources, having many threads increases the memory footprint and decreases the performance of context-switching.

# How does ROS 2 do this differently?

ROS 2 addresses this by introducing **Callback Groups,** which:

- Group of things with callbacks, like *Subscriptions* and *Timers*
- Are completely optional constructs, hidden to users by default
- Works with both component style and the custom main style nodes.

Callbacks in *Reentrant* **Callback Groups** must be able to:

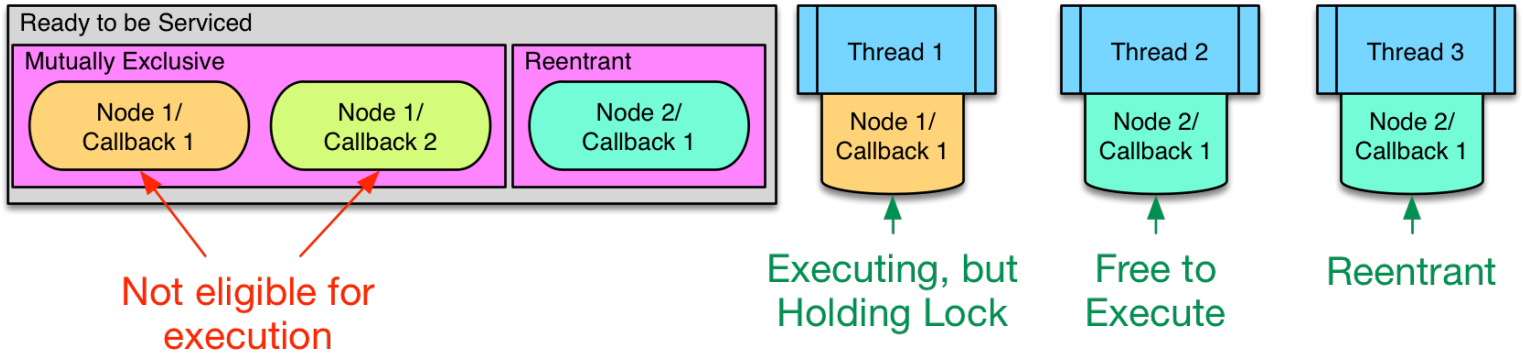
- run at the same time as themselves (reentrant)
- run at the same time as other callbacks in their group
- run at the same time as other callbacks in other groups

Whereas, callbacks in *Mutually Exclusive* **Callback Groups**:

- will not be run multiple times simultaneously (non-reentrant)
- will not be run at the same time as other callbacks in their group
- but must run at the same time as callbacks in other groups

# How could I use the callback groups?

Users can use *Mutually Exclusive* **Callback Groups** in order to ensure callbacks which operate on shared resources do not run at the same time, without using locks.




Apply this grouping to the previous example and now the ROS 2 system is smart enough to know not to try and execute multiple instances of "node 1's callback 1" at the same time. This keeps other threads free in the pool to handle other callbacks.

# So nodelet's are better, so what?

Making the "nodelet" style of writing nodes easier and providing tools like **Callback Groups**, allows users to take advantage of features like **thread-pooling** and **smart pointer passing** to write more efficient code without sacrificing convenience and simplicity.

Other concepts become possible too, for example *topic pipelines*:

```
std::unique_ptr<pkg::MsgType> callback(std::unique_ptr<pkg::MsgType> msg)
{
  // Modify msg, which came from topic_in
  return msg;  // this is published to topic_out for you
}
...
auto pipeline = node->create_pipeline<pkg::MsgType>("topic_in", "topic_out", callback, 10);
```

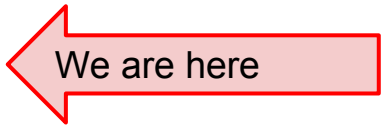
In this way the msg is never copied, but simply moved into and then back out of the callback which can be useful for writing "transfer functions".

# Checkpoint

- Component Style API
  - Example: Talker node
- ROS 1 like API
  - Example: Listener node
- Improvements to “nodelet” style development
  - Unifying “node” and “nodelet” API
  - Improve developer workflow and convenience
  - New tools for managing concurrent code
- Dynamic Graph Features
  - Topic remapping and aliasing
  - Changing graph layout during runtime
- Node lifecycle management
  - Verifiable systems
  - Live relocatable nodes

# Checkpoint

- Component Style API
  - Example: Talker node
- ROS 1 like API
  - Example: Listener node
- Improvements to “nodelet” style development
  - Unifying “node” and “nodelet” API
  - Improve developer workflow and convenience
  - New tools for managing concurrent code
- Dynamic Graph Features
  - Topic remapping and aliasing
  - Changing graph layout during runtime
- Node lifecycle management
  - Verifiable systems
  - Live relocatable nodes



# Topic Remapping and Aliasing

**Topic Remapping:**

Is "I want */camera* to now be */left/camera*" *(during runtime)*:

- Applies to Publishers and Subscriptions.
- If applied to a Publisher, Subscribers to */camera* will disconnect.
- If applied to a Subscriber, Publishers of */camera* will disconnect, and connections to Publishers of */left/camera* will be established.

**Topic Aliasing:**

Is "I want */camera* to also be */left/camera*" *(during runtime)*:

- Applies to Publishers and Subscriptions.
- If applied to a Publisher, connections are additionally established to Subscribers of */left/camera*.
- If applied to Subscribers, connections are additionally established to Publishers of */left/camera*.

# How is this achieved at runtime?

Nodes with either Publishers or Subscriptions automatically create some built-in Services for:

- Querying remappings and aliases
- Remapping a topic by *Name* (The ROS 1 concept of a *Name*)
- Adding an alias by *Name*
- Removing an alias by *Name*

For now we are thinking about Publishers and Subscriptions, but may also apply to Services, Parameters, and even changing the node's entire namespace.

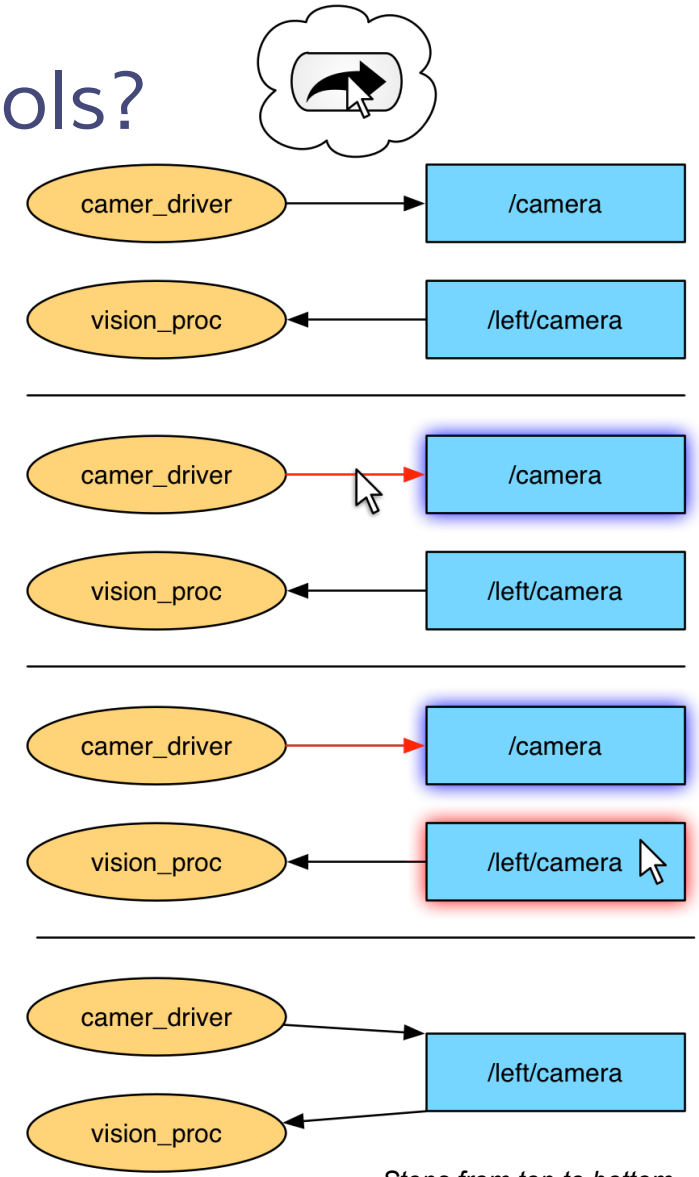
This should enable more powerful runtime developer tools as well as support dynamic situations like multi-robot networks and graph visibility.

# What kind of developer tools?

With dynamic remapping it is easy to imagine a more powerful version of *rqt_graph*.

This version of *rqt_graph* would be able to remap and alias topics at run time, then dump the settings you would need to change to make that change permanent in your launch configurations.

This would allow a workflow where you launch your system, introspect it, fix any missing connections, and then save the result.




*Steps from top to bottom*

# Node Lifecycle Management

In ROS 1:

- Launch files execute nodes, but no feedback on node state
- Can use *required=true* to prevent partial systems
- Using *respawn=true* is a common practice

**Case Study: Capabilities**

The new concept of *capabilities* in ROS 1 (wiki.ros.org/capabilities) works by defining capabilities of a robot, e.g. Navigation, Mobile Base, Pick and Place, as an interface of ROS primitives and a launch file to implement them. Capabilities can be started remotely through a capability_server, which executes the launch files. however, because there is no lifecycle to nodes there exists race conditions for people using the *capability_server*. This pattern could be greatly improved if the state of nodes could be aggregated into a capability state.

# Verifiable Systems

A common practice in ROS 1 is that if a node fails or is misbehaving you just restart the whole launch file, and usually the test for whether or not everything is running is to look at a leaf (or output) topic. This isn't a very robust way to monitor the execution of nodes.

For ROS 2 we are looking at existing lifecycle systems in Orocos RTT, OMG's RTC (implemented in OpenRTM), and others for inspiration.

Without listing the lifecycle states directly here are some of the goals:

- Reliable and Deterministic node startup and shutdown
- Introspectable state (all connections established, responsive, etc...)
- Execution agnostic (in a custom main or in an off-the-shelf container)
- Migratable (store state, stop, respawn elsewhere, restore state)

# Why should I want to use ROS 2?

So, why should you be excited for ROS 2?

- Modern API, minimal dependencies, and better portability
- Benefits of underlying DDS middleware
  - Reliability QoS settings
  - UDP Multicast, shared memory, TLS over TCP/IP
  - Real-Time capable
  - Master-less discovery
  - Minimal dependencies (Current DDS vendors have none)
- Easier to work with multiple nodes in one process
- More dynamic run-time features like topic remapping and aliasing
- Lifecycle management and verifiable systems
- And so many other things we don't have time to cover here...
  - Dynamic parameters
  - Synchronous, scheduled execution of nodes (the ecto problem)
  - More efficient package resource management

# Questions?

http://goo.gl/NgG1kK

Thanks!