Lesson 2: ROS API and Build Tools

- This lesson is intended to be an hour long crash course in the ROS 2 Dashing API and build tools.
- After this lesson you should be able to code and build a rudimentary ROS 2 application.
- It should be noted, that since this is a "crash course" we are giving you a pre-configured Docker environment call ADE.
  - We will not cover setting up the ROS environment.
  - This will be left as an exercise to the reader.
  - As much as possible we have used the tooling available *ROS 2 Dashing desktop full* installation.
- This crash course was written for ROS 2 Dashing.
  - ROS 2 Foxy is about to be released.
  - This release is the first major, stable, release of ROS 2.
- The next lesson, lesson 3, will show you how to use the ROS 2 command line interface.

Before We Begin...

"We choose to go to the moon, not because it is easy, but because it is hard."

ROS is hard. There are many topics you must learn to become a skilled robotocist. We try to make it easy, but be aware the path ahead is difficult. You will get stuck. You will get frustrated. You will need help

The good news is, all of this is OK and completely normal; there are resources out there to help!

Getting Help!

ROS has a ton of resources to help you be succeed. You should be aware of these before you begin.

- We have our own QA website like Stack Overflow called ROS Answers.
  - http://answers.ros.org
  - ROS answers actually predates the rise of Stack Overflow!
- Check out ROS Discourse. It is the community hub for news and discussion.
  - https://discourse.ros.org/
  - *DO NOT* ask questions on discourse.
- We have a very large ROS wiki. It is mostly ROS 1.
  - http://wiki.ros.org/
  - Most of the content is still highly useful.
- Most of this talk comes from the ROS 2 documentation.
  - https://index.ros.org/doc/ros2/
  - This is probably where you should look.

Other and/or Unofficial Resources

- The ROS / Robotics Sub Reddits are Great!
- There is an "unofficial" ROS Discord.
  - Please try using ROS Answers first.
- We have a yearly ROS developers conference ROSCon.
  - Most of old talks are free on the web.
- We're not big on social media but we're busy on the twitter.
  - @OpenRoboticsOrg is a bit more active.
  - @ROSOrg "Official" ROS announcements.
- Open Robotics is the non-profit that administers ROS and Ignition Gazebo.
  - We take donations and take contract work from time to time.

History of ROS

PR2 Image

- Let's go back to the early 2000's.
- Open Source is growing, but Windows dominates.
- What about Robots?:
  - Robots are expensive and mainly for mass manufacturing and R&D.
  - Mostly is "real-time" control systems. Just make arms move the same way over and over.
  - Not a lot of Open Source.
- ~2006, Former Google VPs decide to work on Robots.
  - Create a company called Willow Garage.
  - From this org we get OpenCV, PCL, ROS, PR2 Robot, and many spin outs.
- ~2012 Willow Garage folds, Open Robotics emerges.
  - 2017 ROS 2 Begins to move ROS out of the lab (it was already out of the lab).
  - Address security and robustness concerns.
  - Add RTOS support and support other OS's.

Concepts that Motivate ROS

ROS's design was informed by *design patterns* that were successfully used in prior robotic systems. We can't cover each of these in detail, but reading about them will help you better understand ROS.

- **Processes / Threads ==> ROS Nodes** -- A ROS Node is a self contained execution process, like a program. ROS is really a lot of tooling for running a bunch of programs in parallel.
- **Buses / PubSub ==> ROS Topics** -- The backbone of ROS is a publish/subscribe bus. If you have ever used ZeroMQ, RabbitMQ, or ModBus, ROS topics are very similar.
- **Serialization ==> ROS Messages / ROS Bags** -- ROS uses a predefined messages to move data over topics. This allows data to be serialized between nodes in different programming languages. An analog would be Google Protocol Buffers. ROS can be written to file, called a bag. A good analogy is a python pickle file.
- **Black Board Pattern ==> ROS Params** -- A blackboard is a way to create global variables between nodes/ programs. A good analogy would be Redis.
- **Synchronous Remote Procedure Call (RPC) ==> ROS Services** -- A ROS service is a program that can be called by another program. The caller is blocked until the callee returns. This is formerly called a remote procedure call.
- **Asynchronous Remote Procedure Call (RPC) ==> ROS Actions** -- A ROS action is a program that can be called by another program. The caller is **not** blocked until the callee returns.
- **State Machines ==> ROS Life cycles** -- State machines are a tool to move between states, or modes. State machines are a useful way to model machine behavior.
- **Matrix Math for 3D Operations ==> URDF and TF** -- TF, short for transform, and URDF (universal robot description format) are tools for automatically calculating robot geometry using matrix math .

Jumping in the Deep End

Let's start ADE and install / update deps

- First things first, let's make sure everything is ready to go.
- Now is a good time to hit pause on the video make sure you have installed the requirements.
- Install ADE as per Autoware Instructions.
- Now were going to update the system, install ROS dashing, and a couple tools.

```
ade start
ade enter
source /opt/ros/dashing/setup.bash
sudo apt update
sudo apt install ros-dashing-turtlesim
sudo apt install ros-dashing-rqt-*
sudo apt-install byobu
```

You should now be ready for the class!

Some Nomenclature as we Begin

- **Package** -- A collection of code.
- **Workspace** -- A workspace is a collection of source code / ROS packages that will run on a robot. It has a uniform directory structure. A good analogy is python virtual env, or "project" in most IDEs.
- **Overlay** -- A second workspace with more/different/new packages. If there are multiple versions of a package/code then the one at the bottom is used.
- **Underlay** -- The workspace, underneath an overlay, we're aware this is confusing.
- **Colcon** -- The ROS 2 build tool. Think of it as a layer above CMake/Make/SetupTools that helps these tools work together smoothly.

This is a bit confusing. You may ask yourself why we have our own build tool. The short of it is that the ROS ecosystem consists of tens of thousands of developers, working on thousands of packages, across a handful of platforms, using multiple languages. We needed a flexible system to build code and one didn't exist at the time, and still doesn't exist.

Let's Get Started

As we're diving headfirst into ROS our first job is to checkout a repository of examples and build it. Roughly the steps to do this are as follows.

- Fire up a terminal manager inside the container. I use byobu. You can use whatever you want. You can also fire up 3 real terminals and call *ade enter* on them.
- Source the ROS setup.bash file so we have the right version of ROS in our path.
- Make a workspace called *ros2_example_ws*. We usually use *_ws* to indicate a workspace.
- Clone an example repository and change to the dashing branch.
    - Generally ROS repos have a branch per release.
- Use Colcon to build the source.

```
source /opt/ros/dashing/setup.bash
mkdir -p ~/ros2_example_ws/src
cd ~/ros2_example_ws
git clone https://github.com/ros2/examples src/examples
cd ~/ros2_example_ws/src/examples/
git checkout dashing
cd ~/ros2_example_ws
colcon build --symlink-install
```

Nodes and Publishers

- The core of ROS is the ROS pub/sub bus. In ROS parlance this is called *topic*.
  - A topic has a *message type* that is published on the bus. These messages are defined in a yaml file and define the serialization/deserialization format for ROS messages.
  - ROS has a lot of built in message types. There are lots of pre-defined messages for controlling a robot, distributing sensor data, and understanding the geometry of your robot.
  - ROS publishers produce messages and slowly or as quickly as they need to.
  - A ROS subscriber, *subscribes* to a *topic* and then does things with the information.
- ROS has lots of built-in tools for managing topics. You can list them, echo (watch) them, rename them (called remap), and store them to file (called bagging).
- ROS *Nodes* are basically programs, or processes that run concurrently on ROS.
  - A ROS node can publish to one or more topics.
  - That same node can subscribe to other topics.
  - Many nodes subscribe to topics, process the data, and publish the results.
  - ROS has tooling to start and stop multiple nodes at the same time.

Preparing to Run a ROS Node

- Open a new terminal, in Byobu you can do this by pressing *F2*.
- First we need to source the *setup.bash* file for our *workspace*. This will help ROS find the programs we built.
  - *source ./ros2_example_ws/install/setup.bash*
  - Protip: you can find any file using *find ./ -name <file name>*
- **ROS Best Practice** *ALWAYS* build and execute in different terminals.
  - The build terminal should source the global ROS setup.bash file (i.e. /opt/ros/dashing/setup.bash).
  - The execution terminal should source the *setup.bash* of your workspace
  - This is a common failure mode for new users. If something seems weird or funky. Create a new terminal and source the correct bash file.

Let's Run a Simple C++ Publisher Node.

- ROS has an advanced, and fairly complex CLI interface. We'll cover it in depth in our next lesson.
- We are going to ask ros to run the EXECUTABLE *publisher_lambda* in our WORKSPACE named *examples_rclcpp_minimal_publisher*.
- The syntax for doing this is *ros2 run* <*WORKSPACE*> <*EXECUTABLE*>
- To run our publishing node, let's run the following command in our execution terminal: *ros2 run examples_rclcpp_minimal_publisher publisher_lambda*
- If everything works you should see something like this:

```
kscottz@ade:~$ ros2 run examples_rclcpp_minimal_publisher publisher_lambda
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 0'
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 1'
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 2'
[INFO] [minimal_publisher]: Publishing: 'Hello, world! 3'
...
```

- To exit the program press *CTRL-C*

What just happened?

- We just executed a ROS node that publishes a simple string message to a topic called */topic* twice a second.
- I'll show you how I know this with some tools. We'll cover these tools in detail next time.

```
kscottz@ade:~$ ros2 topic list
/parameter_events
/rosout
/topic
kscottz@ade:~$ ros2 topic echo /topic
data: Hello, lambda world! 63
---
data: Hello, lambda world! 64
---
data: Hello, lambda world! 65
---
kscottz@ade:~$ ros2 topic hz /topic
average rate: 2.000
min: 0.500s max: 0.500s std dev: 0.00011s window: 4
kscottz@ade:~$
```

Digging into the Code

- Let's take a look at the code. Like a lot of software there is more than one way to skin a cat. Let's look at the member function approach.
- Using your favorite editor open the following source file, *./ros2_example_ws/src/examples/rclcpp/minimal_publisher/member_function.cpp*
- **rclcpp** is an abbreviation of "ROS Client Library C++", its the ROS C++ API

```
 1 #include <chrono>
 2 #include <memory>
 3
 4 #include "rclcpp/rclcpp.hpp" // THIS the header file for ROS 2 C++ API
 5 #include "std_msgs/msg/string.hpp" // This is header for the messages we
 6                                    // want to user
 7                                    // These are usually auto generated.
 8
 9 using namespace std::chrono_literals;
10
11 /* This example creates a subclass of Node and uses std::bind() to register a
12 * member function as a callback from the timer. */
13   // Make a class called Minimal Publisher
14   class MinimalPublisher : public rclcpp::Node
15   // Have it inherit from the ROS Node Class
```

Let's Build our Node's Constructor

- The *MinimalPublisher* constructor inherits from the RCLCPP Base Class, gives the name a node, and sets our counter.
- The next line creates a publisher object that publishes *std_msgs::msg*.
- The constructor then creates a callback to the function *timer_callback* that gets called every 500ms.

```
class MinimalPublisher : public rclcpp::Node // Inherit from ROS Node
{
  public:
    MinimalPublisher()
    : Node("minimal_publisher"), count_(0) // Set the node name
    {   //Create a publisher that pushes std_msgs::msg to the topic "topic"
      publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
      timer_ = this->create_wall_timer( // Call timer_callback every 500ms
      500ms, std::bind(&MinimalPublisher::timer_callback, this));
    }
```

Now to Handle the Callback

- In the callback function we do the following:
  - Create the ROS *std_msgs::msg::String()* to send to our topic.
  - Construct the message that will be pushed to the ROS Topic
  - Log the results.
  - Actually publish the newly constructed message.

```
private:
void timer_callback()
{
  auto message = std_msgs::msg::String(); // create message
  message.data = "Hello, world! " + std::to_string(count_++); // Fill it up
  RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str()); // Log it
  publisher_->publish(message); // Publish
}
// Create our private member variables.
rclcpp::TimerBase::SharedPtr timer_;
rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
size_t count_;
```

Finally, Let's Create the Main for our Node

- This last little bit creates the main node entry point.
- Initializes *rcpcpp* with the values from the command line.
- Run's the MinimalPublisher, until a terminate is given
- Finally the node cleans up everything and exits.

```
int main (int argc, char * argv[])
{
  rclcpp::init(argc, argv); // Init RCL
  rclcpp::spin(std::make_shared<MinimalPublisher>());// Run the minimal publish
  rclcpp::shutdown(); // Cleanup on shut down.
  return 0;
}
```

Exercise: Modify and Build this Node

- Let's try to make a few modification to our node for practice.
  - Make it run at 10Hz (100ms) instead of 500.
  - Change the topic name from "topic" to "greetings."
  - Change the message "Hello Open Road."
  - Change the node name from *minimal_publisher*, *revenge_of_minimal_publisher*
- Once you make these changes
  - Save the file.
  - Toggle over to your execution window run
  - Run *colcon build*
  - In your execution window run *ros2 run examples_rclcpp_minimal_publisher publisher_member_function*

Let's Try Subscribing.

- The pattern here is similar to publishing.
- We basically inherit from the Node class, and define the topic and message we want.
- Whenever that topic is published we hit a callback.
- If everything is correctly configured the file is at
    - /ros2_example_ws/src/examples/rclcpp/minimal_subscriber/member_function.cpp

```
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
using std::placeholders::_1;
// Again we inherit the public interface of a ROS node.
class MinimalSubscriber : public rclcpp::Node
{
  public:
  MinimalSubscriber() // Construct our node, calling it minimal_subscriber
  : Node("minimal_subscriber")
  { // Create a subscription, to messages of the format stdmsg:msg:String
    subscription_ = this->create_subscription<std_msgs::msg::String>(
    // Subscribe to the topic, "topic" and set a callback for when things are pub'd
    "topic", 10, std::bind(&MinimalSubscriber::topic_callback, this, _1));
  }
  ...
```

More Subscriber

- The subscriber node looks fairly similar to our publisher but instead of publishing on a regular callback, we get a callback when a new message hits our topic.

```
private:
  // Whenever we get a new messaged published on our topic
  // this callback will be executed.
  void topic_callback(const std_msgs::msg::String::SharedPtr msg) const
  {
    // Log the message that we are subscribed to
    RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg->data.c_str());
  }
  rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
};

// This is effectively the same boiler plate from last time.
int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<MinimalSubscriber>());
  rclcpp::shutdown();
  return 0;
}
```

Let's Modify the Subscriber

- In the publisher we changed the name of our publisher topic to *greetings*.
- Let's change the subscribed topic to *greetings*.
- Note that there are a lot of ways to change topic names, modifying source is just one approach. Often we just *remap* topics instead of changing source.
- Once you have modified the subscriber run *colocon build* (it will build everything)
- Open another terminal, source the bash file, and start the publisher.
    - *ros2 run examples_rclcpp_minimal_publisher publisher_member_function*
- Now run our subscriber.
    - *ros2 run examples_rclcpp_minimal_subscriber subscriber_member_function*

The Result

If everything went well you should have two screens. The first screen with the publisher should be spitting out the following

```
[INFO] [revenge_of_minimal_publisher]: Publishing: 'Hello, Open Road! 1000'
[INFO] [revenge_of_minimal_publisher]: Publishing: 'Hello, Open Road! 1001'
[INFO] [revenge_of_minimal_publisher]: Publishing: 'Hello, Open Road! 1002'
[INFO] [revenge_of_minimal_publisher]: Publishing: 'Hello, Open Road! 1003'
[INFO] [revenge_of_minimal_publisher]: Publishing: 'Hello, Open Road! 1004'
```

The subscriber screen should be pushing out:

```
[INFO] [minimal_subscriber]: I heard: 'Hello, Open Road! 1000'
[INFO] [minimal_subscriber]: I heard: 'Hello, Open Road! 1001'
[INFO] [minimal_subscriber]: I heard: 'Hello, Open Road! 1002'
[INFO] [minimal_subscriber]: I heard: 'Hello, Open Road! 1003'
[INFO] [minimal_subscriber]: I heard: 'Hello, Open Road! 1004'
```

You can terminate both of these programs with `CTRL-C`

Congratulations, you now know the three most important ROS components, nodes, publishers, and subscribers.

Making Things Happen with Services

- Publishing and subscribing nodes are the bread and butter of ROS. This pattern is great for moving around a lot of data, and processing it quickly.
- However, we often want our robots to respond to data. To construct simple behaviors in ROS we use *services*.
- A service is a robotic task that can be performed *synchronously*, which is just a fancy word for, "while you wait".
- A good analogy for services would be a regular old function call. In most programs when you call a function, the code making the call waits for the function to return before proceeding.
- A few toy examples of services for autonomous driving would be:
  - Turning Lights Off/On.
  - Checking a sensor and returning the results.
  - Lock / Unlock a door or window.
  - Beeping a horn.
- Services can be called via the command line or through an API call within another node.
- In ROS services are hosted within a ROS Node, and they can co-exist with other services as well as publishers and subscribers.

C++ Service Example

- As a toy example of a ROS service we are going to make a node that offers an "AddTwoInts" service.
- What will happen is the service has two inputs, and returns a single output.
- There is a full tutorial about the process here. It goes into more detail and it is worth looking at.

Let's start by looking at a prebuilt *srv* file for this tutorial. If you were writing this service from scratch you would need to build this *srv* file yourself, but for this example there is one ready for us already. We'll use less to peek into the srv file.

Run the following: *less /opt/ros/dashing/share/example_interfaces/srv/AddTwoInts.srv*

The file should have the following:

```
int64 a     # <== An input, of type int64, called a
int64 b     # <== An input, of type int64, called b
---
int64 sum   # <== An output, of type int64, called sum
```

Defining A Service

Essentially our service is a remote procedure call of a function that looks like this in pseudocode: *int64 sum = AddTwoInts(int64 a, int64b);*.

Let's take a look at the C++ code that defines the service. Use your favorite text editor to open the following file: *./ros2_example_ws/src/examples/rclcpp/minimal_service/main.cpp*.

```
 // This hpp file is autogenerated from the srv file.
 #include "example_interfaces/srv/add_two_ints.hpp"
 #include "rclcpp/rclcpp.hpp"// ROS header.
 // Scope resolution to our services.
 using AddTwoInts = example_interfaces::srv::AddTwoInts;
 // shared pointer to logger
 rclcpp::Node::SharedPtr g_node = nullptr;
 // Perform the service call
 void handle_service(
   const std::shared_ptr<rmw_request_id_t> request_header,// Header with timestamp etc
   const std::shared_ptr<AddTwoInts::Request> request,    // This is the input, two int64 a,
   const std::shared_ptr<AddTwoInts::Response> response)  // This response is int64 sum
 {
   (void)request_header;
   RCLCPP_INFO( // Logger message.
     g_node->get_logger(),
     "request: %" PRId64 " + %" PRId64, request->a, request->b);
  response->sum = request->a + request->b; // the actual function.
}
```

ROS 2 Service Main

```
 1  int main(int argc, char ** argv)
 2  {
 3    rclcpp::init(argc, argv);
 4    // get global ROS pointer
 5    g_node = rclcpp::Node::make_shared("minimal_service");
 6    // Create a service, of type AddTwoInts, named add_two_ints, that points to handle_service
 7    auto server = g_node->create_service<AddTwoInts>("add_two_ints", handle_service);
 8    rclcpp::spin(g_node); // run until shutdown
 9    rclcpp::shutdown();
10    g_node = nullptr;
11    return 0;
12  }
```

The main entry point is pretty simple. It does the following.

- Initialize the program.
- Get a shared pointer to the ROS node interface.
- Create the service, of type AddTwoInts, named add_two_ints, pointing to the function *handle_service*.
- Run the node until shutdown.

Let's Build and Run our Service

First we will fire up our service! The syntax for this is *ros2 run* <*pkg*> <*program*>.

```
kscottz@ade:~$ ros2 run examples_rclcpp_minimal_service service_main
```

At this point nothing should happen. We need to *call* the service. To do that we'll use a command line tool that's a little... long.

We'll talk about this more in the next lesson, but the syntax is roughly, *ros2 service call* <*service_name*> <*service_call_format*> <*actual_data*>.

In this case our service name is */add_two_ints* and the data type can be found in *example_interfaces/AddTwoInts*, and the input is yaml encased in quotation marks. Move over to a new terminal and enter the following:

```
kscottz@ade:~/ros2_example_ws$ ros2 service call /add_two_ints example_interfaces/AddTwoInts
waiting for service to become available...
requester: making request: example_interfaces.srv.AddTwoInts_Request(a=1, b=1)

response:
example_interfaces.srv.AddTwoInts_Response(sum=2)
```

Now switch back to your original terminal, you should see something like this:

```
kscottz@ade:~$ ros2 run examples_rclcpp_minimal_client client_main
3[INFO] [minimal_service]: Incoming request
a: 1 b: 1
```

Congratulations, you just made your first service call!

Using a Service in Code

We just called our service from the command line to test it, but more often than not we would want to do this in source code.

Let's look at an example of how to do that. In your editor or using less take a look at the following file: */home/kscottz/ros2_example_ws/src/examples/rclcpp/minimal_client/main.cpp*

```
// snipped
#include "example_interfaces/srv/add_two_ints.hpp" // include the service header file.
#include "rclcpp/rclcpp.hpp"
// Scope resolution on underlying call signature.
using AddTwoInts = example_interfaces::srv::AddTwoInts;

int main(int argc, char * argv[])
{
   rclcpp::init(argc, argv); //  init ROS C++ interface.
   auto node = rclcpp::Node::make_shared("minimal_client"); // shared node memory.
   auto client = node->create_client<AddTwoInts>("add_two_ints"); // create client interface
   while (!client->wait_for_service(std::chrono::seconds(1))) {// poll for service to come o
     if (!rclcpp::ok()) {  // if service doesn't come online, exit gracefully
       RCLCPP_ERROR(node->get_logger(),
           "client interrupted while waiting for service to appear.");
       return 1;
   }
   RCLCPP_INFO(node->get_logger(), "waiting for service to appear...");
}
```

C++ Service Client Part Deux

```
 // shared memory to request
 auto request = std::make_shared<AddTwoInts::Request>();
 request->a = 41;  // set the input values
 request->b = 1;   // set the input values
 auto result_future = client->async_send_request(request); // Send the request
 if (rclcpp::spin_until_future_complete(node, result_future) != // spin until result
    rclcpp::executor::FutureReturnCode::SUCCESS)
 {
    RCLCPP_ERROR(node->get_logger(), "service call failed :(");
    return 1;
 }
 auto result = result_future.get(); // Get the result
 RCLCPP_INFO(node->get_logger(), "result of %" PRId64 " + %" PRId64 " = %" PRId64,
    request->a, request->b, result->sum); // print the result
rclcpp::shutdown(); // shutdown
return 0;
}
```

Let's Run Our Client

- Now we're going to run our service and then call it from the client.
- You'll need two terminals to do this. Remember *F2*/*F3* let you open and switch to a new terminal in ADE.

First fire up your service if it isn't already running.

```
$ ros2 run examples_rclcpp_minimal_service service_main
```

Now start the client in a second terminal.

```
$ ros2 run examples_rclcpp_minimal_client client_main
[INFO] [minimal_client]: Result of add_two_ints: for 41 + 1 = 42
```

The client should fire off a request right away. You can see the result.

Finally, toggle back to the service.

```
$ ~/ros2_example_ws$ ros2 run examples_rclcpp_minimal_service service_main
[INFO] [minimal_service]: Incoming request
a: 41 b: 1
```

You can see the debug input has been printed to the terminal.

ROS C++ Actions

- Actions are ROS / ROS 2's answers to asynchronous remote procedure calls.
- Notice how quickly how fast our service call happened. It was more or less instant.
- Actions are the preferred approach for things that may not happen instantaneously.
- The canonical example of a ROS Action would be sending the robot a command to navigate to a way point.
- The process of navigation is going to take a bit of time, what we want to do is to kick off the process, wait for updates, and then once things are complete we get a result.
- Just like services there are two parts of an action. The action server and the action client. *Note that there can be more than one client*.
- Actions become fairly complex as they can serve multiple clients. This means the action may need to keep track of multiple concurrent connections.
    - Since action servers can get overwhelmed by requests, they need to *accept* every request before proceeding to process it.
    - The clients can also *cancel* at any time, so that needs to be handled.

Parts of an Action

- Find the action. An action server may be down!
- The Action Request -- the service *can* decline to take an action.
- The Action being accepted.
- The Action being canceled. Sometimes the client changes its mind.
- The action "feedback", sending back info from server to client.
- Send the result to the client -- the result could be the thing happened succesfully, or not!

Fibonacci Action

For our action server we're going to create a toy example, this example will calculate the *Nth number* in the Fibonacci series. So, what will happen when we call this toy action?

- We will call the action with a single integer indicating the *sequence number* of the Fibonacci number we want.
- The action will update us as it calculates the sequence of numbers and update it us as it calculates a new one.
- When the action gets to our desired number in the sequence, it will return the results.
- For example, if we called action with the input 7, we would get the seventh Fibonacci number. Which means, given the series <0, 1, 1, 2, 3, 5, 8>, would be the number 8.
- The action should update us along the way in the calculation. It should return the series of numbers every time it calculates a new number.

Action Definition Files

- Actions use a definition file to build all of the ROS boiler plate like cross language header/definition files for use in multiple programming languages.
- These action files are written in YAML and use the *.*action* suffix.
- The ROS meta build system colcon will use these action files to auto-magically generate all of the header files.

Let's take a look at an action file.

```
/opt/ros/dashing/share/example_interfaces/action/Fibonacci.action
# Goal -- the input, the order we want like 7
int32 order
---
# Result -- the *final result*, here the list of values 0,1,1,2,3,5,8....
int32[] sequence
---
# Feedback -- the *intermediate result* so <0>,<0,1>,<0,1,1>,<0,1,1,2> ...
int32[] sequence
Fibonacci.action (END)
```

Really quick, let's look under the hood!

As we said previously, the **.action* is used to auto-generate a bunch of other files. We can see this if we go down one directory to msg.

What we'll see is that the **.action* file is used to generate a bunch of ROS topic messages mapping to states in our action.

Essentially a ROS action is built upon ROS nodes and ROS topics

```
kscottz@ade:/opt/ros/dashing/share/example_interfaces/action/msg$ cd ~/
kscottz@ade:~$ cd /opt/ros/dashing/share/example_interfaces/action/msg/
kscottz@ade:/opt/ros/dashing/share/example_interfaces/action/msg$ ls
FibonacciActionFeedback.msg  FibonacciAction.msg        FibonacciFeedback.msg  FibonacciResu
FibonacciActionGoal.msg      FibonacciActionResult.msg  FibonacciGoal.msg
kscottz@ade:/opt/ros/dashing/share/example_interfaces/action/msg$ less FibonacciActionGoal.m
# This file is automatically generated by rosidl-generator
std_msgs/Header header
actionlib_msgs/GoalID goal_id
FibonacciGoal goal
FibonacciActionGoal.msg (END)
kscottz@ade:/opt/ros/dashing/share/example_interfaces/action/msg$ cd ~
```

Let's take a look at Action Server

- Let's take a look at how our Fibonacci action server.
- Use your favorite text editor to open: */home/kscottz/ros2_example_ws/src/examples/rclcpp/minimal_action_server/member_functions.cpp*

```
<headers cut>
class MinimalActionServer : public rclcpp::Node
{
public:  // Pre-defined interface files.
   using Fibonacci = example_interfaces::action::Fibonacci;
   using GoalHandleFibonacci = rclcpp_action::ServerGoalHandle<Fibonacci>;
   explicit MinimalActionServer(const rclcpp::NodeOptions & options = rclcpp::NodeOptions())
   : Node("minimal_action_server", options)
   {
     using namespace std::placeholders;
     // SCARY call to define that this is a Fib. action and what functions
     // attach to what events in the action lifecycle.
     this->action_server_ = rclcpp_action::create_server<Fibonacci>(
     this->get_node_base_interface(),      // The action server is basically
     this->get_node_clock_interface(),     // a node and we need return pointers
     this->get_node_logging_interface(),   // to all of standard interaces.
     this->get_node_waitables_interface(),
     "fibonacci",  // and bind our member functions to topic events.
     std::bind(&MinimalActionServer::handle_goal, this, _1, _2),
     std::bind(&MinimalActionServer::handle_cancel, this, _1),
     std::bind(&MinimalActionServer::handle_accepted, this, _1));
   }
```

Actions: Accept or Cancel

Let's deal with accepting a goal, or canceling a goal.

```
private:
  rclcpp_action::Server<Fibonacci>::SharedPtr action_server_;

  rclcpp_action::GoalResponse handle_goal(
   const rclcpp_action::GoalUUID & uuid, // Each request gets a UUID
   std::shared_ptr<const Fibonacci::Goal> goal) // The goal object
  {
     RCLCPP_INFO(this->get_logger(), "Received goal request with order %d", goal->order);
     (void)uuid;
     // Let's reject sequences that are over 9000
     if (goal->order > 9000) {
       return rclcpp_action::GoalResponse::REJECT;
       } // respond with "yes, we'll process this request.
   return rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE;
}

rclcpp_action::CancelResponse handle_cancel(
   const std::shared_ptr<GoalHandleFibonacci> goal_handle)
{
   RCLCPP_INFO(this->get_logger(), "Received request to cancel goal");
   (void)goal_handle;
   return rclcpp_action::CancelResponse::ACCEPT;
}
```

The Meat of the Fib Function

```
void execute(const std::shared_ptr<GoalHandleFibonacci> goal_handle)
{ // This is the meaty part of the function
   RCLCPP_INFO(this->get_logger(), "Executing goal");
   rclcpp::Rate loop_rate(1);
   const auto goal = goal_handle->get_goal(); // this is our goal value
   auto feedback = std::make_shared<Fibonacci::Feedback>(); // this is our feedback object
   auto & sequence = feedback->sequence; // this is our list of fib values.
   sequence.push_back(0);
   sequence.push_back(1);
   auto result = std::make_shared<Fibonacci::Result>(); // This is the final result.

   // Do fib as long as ROS is ok!
   for (int i = 1; (i < goal->order) && rclcpp::ok(); ++i) {
   // Check if there is a cancel request
   if (goal_handle->is_canceling()) { // Handle a cancel result!
     result->sequence = sequence;
     goal_handle->canceled(result);
     RCLCPP_INFO(this->get_logger(), "Goal Canceled");
     return;
   }
   // Update sequence
   sequence.push_back(sequence[i] + sequence[i - 1]);
   // Publish feedback
   goal_handle->publish_feedback(feedback);
   RCLCPP_INFO(this->get_logger(), "Publish Feedback");

   loop_rate.sleep();
   }

 // Check if goal is done
 if (rclcpp::ok()) {
   result->sequence = sequence;
   goal_handle->succeed(result);
   RCLCPP_INFO(this->get_logger(), "Goal Succeeded");
```

Let's Put our Class into an Executable

```
 1  int main(int argc, char ** argv)
 2  {
 3     rclcpp::init(argc, argv);
 4
 5     auto action_server = std::make_shared<MinimalActionServer>();
 6
 7     rclcpp::spin(action_server);
 8
 9     rclcpp::shutdown();
10     return 0;
11  }
```

Let's Run Our Action and Call It.

- Ordinarily you would call *colcon build* in your workspace to build the source code. We're just inspecting this method so this isn't necessary.
- We'll start the action server and then call it manually using the ROS 2 CLI.

```
kscottz@ade:~/ros2_example_ws$ ros2 run examples_rclcpp_minimal_action_server action_server_
```

Now we're going to manually call the server from the ROS 2 CLI. We'll cover this in more depth in the next lesson. If you're using byobu use *F3* to go to a second terminal or *F2* to make a new one.

```
$ ros2 action send_goal /fibonacci example_interfaces/action/Fibonacci '{order: 10}'
Waiting for an action server to become available...
Sending goal:
   order: 10

Goal accepted with ID: 0c1b3779c7ea44b69d54c6e1cfac3ff6

Result:
   sequence: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

Goal finished with status: SUCCEEDED
```

Meanwhile, Back at The Server

You can use *F3* to see what happened to our action and its status updates.

```
[INFO] [minimal_action_server]: Received goal request
[INFO] [minimal_action_server]: Executing goal...
[INFO] [minimal_action_server]: Publishing feedback: array('i', [0, 1, 1])
[INFO] [minimal_action_server]: Publishing feedback: array('i', [0, 1, 1, 2])
[INFO] [minimal_action_server]: Publishing feedback: array('i', [0, 1, 1, 2, 3])
[INFO] [minimal_action_server]: Publishing feedback: array('i', [0, 1, 1, 2, 3, 5])
... SNIP ...
[INFO] [minimal_action_server]: Returning result: array('i', [0, 1, 1, 2, 3, 5, 8, 13, 21, 3
```

Action Client

- Let's take at the client side API implementation. Open the file:

~/ros2_example_ws/src/examples/rclcpp/ minimal_action_client/member_functions.cpp

- We'll address the basic implementation but that directory has additional examples for other use cases and things like canceling an action mid-process.
- It is worth understanding what we're doing, it is more than sending just the goal. Roughly this class does the following:
  - Check's for a connection to ROS, and the action server.
  - Sends the goal.
  - Checks that the goal was "accepted" after sending.
  - Updates the log/screen as interim feedback gets sent.
  - Receives the final results.

Let's Create A Client Class

```
#include "example_interfaces/action/fibonacci.hpp"
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"
class MinimalActionClient : public rclcpp::Node
{
   public: // looks familiar, pulling in the action interface, and the goal type
   using Fibonacci = example_interfaces::action::Fibonacci;
   using GoalHandleFibonacci = rclcpp_action::ClientGoalHandle<Fibonacci>;

   explicit MinimalActionClient(const rclcpp::NodeOptions & node_options = rclcpp::NodeOptic
   : Node("minimal_action_client", node_options), goal_done_(false)
   {  // Create a client interface.
      this->client_ptr_ = rclcpp_action::create_client<Fibonacci>(
      this->get_node_base_interface(),
      this->get_node_graph_interface(),
      this->get_node_logging_interface(),
      this->get_node_waitables_interface(),
      "fibonacci");
      // Create a time and have callback to send goal in 500ms
      this->timer_ = this->create_wall_timer(
      std::chrono::milliseconds(500),
      std::bind(&MinimalActionClient::send_goal, this));
   }
```

Sending the Goal

Our client constructor above set a time to call *send_goal* after 500ms. We'll bind our member functions to the action events and then send the goals.

```
// method to check if goal is done
bool is_goal_done() const
{
   return this->goal_done_;
}

void send_goal()
{
   using namespace std::placeholders;
   this->timer_->cancel();
   this->goal_done_ = false;
   // fail to connect to logger.
   if (!this->client_ptr_) {
      RCLCPP_ERROR(this->get_logger(), "Action client not initialized");
   }
   // fail to find the server
   if (!this->client_ptr_->wait_for_action_server(std::chrono::seconds(10))) {
       RCLCPP_ERROR(this->get_logger(), "Action server not available after waiting");
       this->goal_done_ = true;
   return;
   }
   // create the goal msg type and set
   auto goal_msg = Fibonacci::Goal();
   goal_msg.order = 10;
   RCLCPP_INFO(this->get_logger(), "Sending goal");
```

More Send_Goal

```
    auto send_goal_options = rclcpp_action::Client<Fibonacci>::SendGoalOptions();
    // response callback (success/failure)
    send_goal_options.goal_response_callback =
    std::bind(&MinimalActionClient::goal_response_callback, this, _1);
    // server feedback callback
    send_goal_options.feedback_callback =
    std::bind(&MinimalActionClient::feedback_callback, this, _1, _2);
    // result callback bind
    send_goal_options.result_callback =
    std::bind(&MinimalActionClient::result_callback, this, _1);
    auto goal_handle_future = this->client_ptr_->async_send_goal(goal_msg, send_goal_options)
}
```

Handling the Responses

Next up we create our private member variables and define the functions that get called with the goal response and the periodic feedback.

```
private:
   rclcpp_action::Client<Fibonacci>::SharedPtr client_ptr_;
   rclcpp::TimerBase::SharedPtr timer_;
   bool goal_done_;
   // handle the response to our request
   void goal_response_callback(std::shared_future<GoalHandleFibonacci::SharedPtr> future)
   {
      auto goal_handle = future.get();
      if (!goal_handle) {
          RCLCPP_ERROR(this->get_logger(), "Goal was rejected by server");
      } else {
      RCLCPP_INFO(this->get_logger(), "Goal accepted by server, waiting for result");
      }
   }
   // handle the feedback calls, these should be the format of feedback.
   void feedack_callback(
   GoalHandleFibonacci::SharedPtr,
   const std::shared_ptr<const Fibonacci::Feedback> feedback)
   {
      RCLCPP_INFO(
      this->get_logger(),
      "Next number in sequence received: %" PRId64,
      feedback->sequence.back());
   }
```

Handling The Result

```
 1 // handle result callback
 2 void result_callback(const GoalHandleFibonacci::WrappedResult & result)
 3 {
 4    this->goal_done_ = true;
 5    switch (result.code) {
 6      case rclcpp_action::ResultCode::SUCCEEDED:
 7        break;
 8      case rclcpp_action::ResultCode::ABORTED:
 9        RCLCPP_ERROR(this->get_logger(), "Goal was aborted");
10        return;
11      case rclcpp_action::ResultCode::CANCELED:
12         RCLCPP_ERROR(this->get_logger(), "Goal was canceled");
13         return;
14      default:
15        RCLCPP_ERROR(this->get_logger(), "Unknown result code");
16        return;
17     }
18
19     RCLCPP_INFO(this->get_logger(), "Result received");
20     for (auto number : result.result->sequence) {
21       RCLCPP_INFO(this->get_logger(), "%" PRId64, number);
22     }
23   }
24 };  // class MinimalActionClient
```

Running our Client Class

Finally the main function that attaches to our node class. It simply creates a class instance and runs until completion.

```
 1 int main(int argc, char ** argv)
 2 {
 3    rclcpp::init(argc, argv);
 4    auto action_client = std::make_shared<MinimalActionClient>();
 5
 6    while (!action_client->is_goal_done()) {
 7      rclcpp::spin_some(action_client);
 8    }
 9
10    rclcpp::shutdown();
11    return 0;
12 }
```

Let's Run our Client

- We'll start the action server the same way as before

```
kscottz@ade:~/ros2_example_ws$ ros2 run examples_rclcpp_minimal_action_server action_server_
```

Next we'll run our client.

```
kscottz@ade:~/ros2_example_ws$ ros2 run examples_rclcpp_minimal_action_client action_client_
[INFO] [minimal_action_client]: Waiting for action server...
[INFO] [minimal_action_client]: Sending goal request...
[INFO] [minimal_action_client]: Goal accepted :)
[INFO] [minimal_action_client]: Received feedback: array('i', [0, 1, 1])
[INFO] [minimal_action_client]: Received feedback: array('i', [0, 1, 1, 2])
[INFO] [minimal_action_client]: Received feedback: array('i', [0, 1, 1, 2, 3])
[INFO] [minimal_action_client]: Received feedback: array('i', [0, 1, 1, 2, 3, 5])
[INFO] [minimal_action_client]: Received feedback: array('i', [0, 1, 1, 2, 3, 5, 8])
[INFO] [minimal_action_client]: Received feedback: array('i', [0, 1, 1, 2, 3, 5, 8, 13])
[INFO] [minimal_action_client]: Received feedback: array('i', [0, 1, 1, 2, 3, 5, 8, 13, 21])
[INFO] [minimal_action_client]: Received feedback: array('i', [0, 1, 1, 2, 3, 5, 8, 13, 21,
[INFO] [minimal_action_client]: Received feedback: array('i', [0, 1, 1, 2, 3, 5, 8, 13, 21,
[INFO] [minimal_action_client]: Goal succeeded! Result: array('i', [0, 1, 1, 2, 3, 5, 8, 13,
```

Wrapping Up...

- We've just seen the set of API primitives upon which most ROS systems are made.
- Generally speaking, when you build a robot you work from simple to complex. You build the nodes and topics first, then the services, and finally the actions.
- While we addressed all of these topics with the C++ API there is an equivalent Python API that works similarly.
- Moreover, there are additional API primitives that you can check out.
- All of these examples are in the workspace that we created.
- I would encourage you to modify these examples to build a better idea of how they work.

Next time we'll cover the ROS 2 CLI