The command line

- ROS is effectively Linux for Robots.
- From Last Time:
  - ROS has your cross platform build tools.
  - ROS has ploygot builds.
- Now lets talk about runtime.
  - ROS presents command line interface (CLI) for robot execution.
  - Most of these commands follow a regular format.
  - These commands have auto-tab complete (yay!)
  - Most blank commands will spit out error or info.
  - Most commands will behave nicely with `--help`

Let's Fire up ADE

- It is worth noting this should work for any Ubuntu 18.04 machine.
- You'll have to do this every time you restart ADE!

Let's setup our environment:

```
cd adehome
export PATH=$PATH:$PWD
cd AutowareAuto
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
```

Example of --help

Here is ros2 `--help`:

```
kscottz@ade:~$ ros2 --help
usage: ros2 [-h] Call `ros2 <command> -h` for more detailed usage. ...

ros2 is an extensible command-line tool for ROS 2.

optional arguments:
-h, --help            show this help message and exit

Commands:
action     Various action related sub-commands
bag        Various rosbag related sub-commands
component  Various component related sub-commands
daemon     Various daemon related sub-commands
launch     Run a launch file
lifecycle  Various lifecycle related sub-commands
msg        Various msg related sub-commands
multicast  Various multicast related sub-commands
node       Various node related sub-commands
param      Various param related sub-commands
pkg        Various package related sub-commands
run        Run a package specific executable
security   Various security related sub-commands
service    Various service related sub-commands
srv        Various srv related sub-commands
test       Run a ROS2 launch test
topic      Various topic related sub-commands

Call `ros2 <command> -h` for more detailed usage.
```

That's all your commands!

ROS 2 run

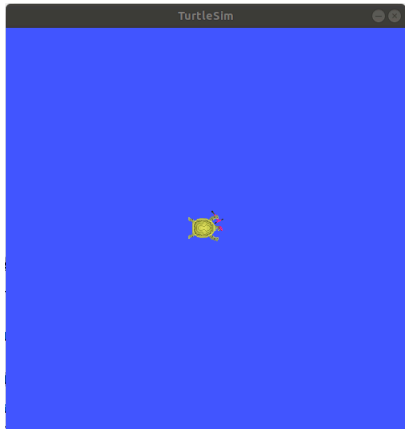
`ros2 run` is used to execute ROS nodes.

General format is `ros2 run <package_name> <executable_name> <flags>`

- *TAB COMPLETIONS!* are baked in.
- Generally if tabbing works, you are good to go.
- Don't know a full package name? Try tabbing.
- Don't know the executables in a package? * TRY TABBING!!!
- Why don't we try starting this *turtlesim node*.
- In your terminal type `ros2 run turtlesim turtlesim_node`

TA-DA! A wild ROS turtle appears

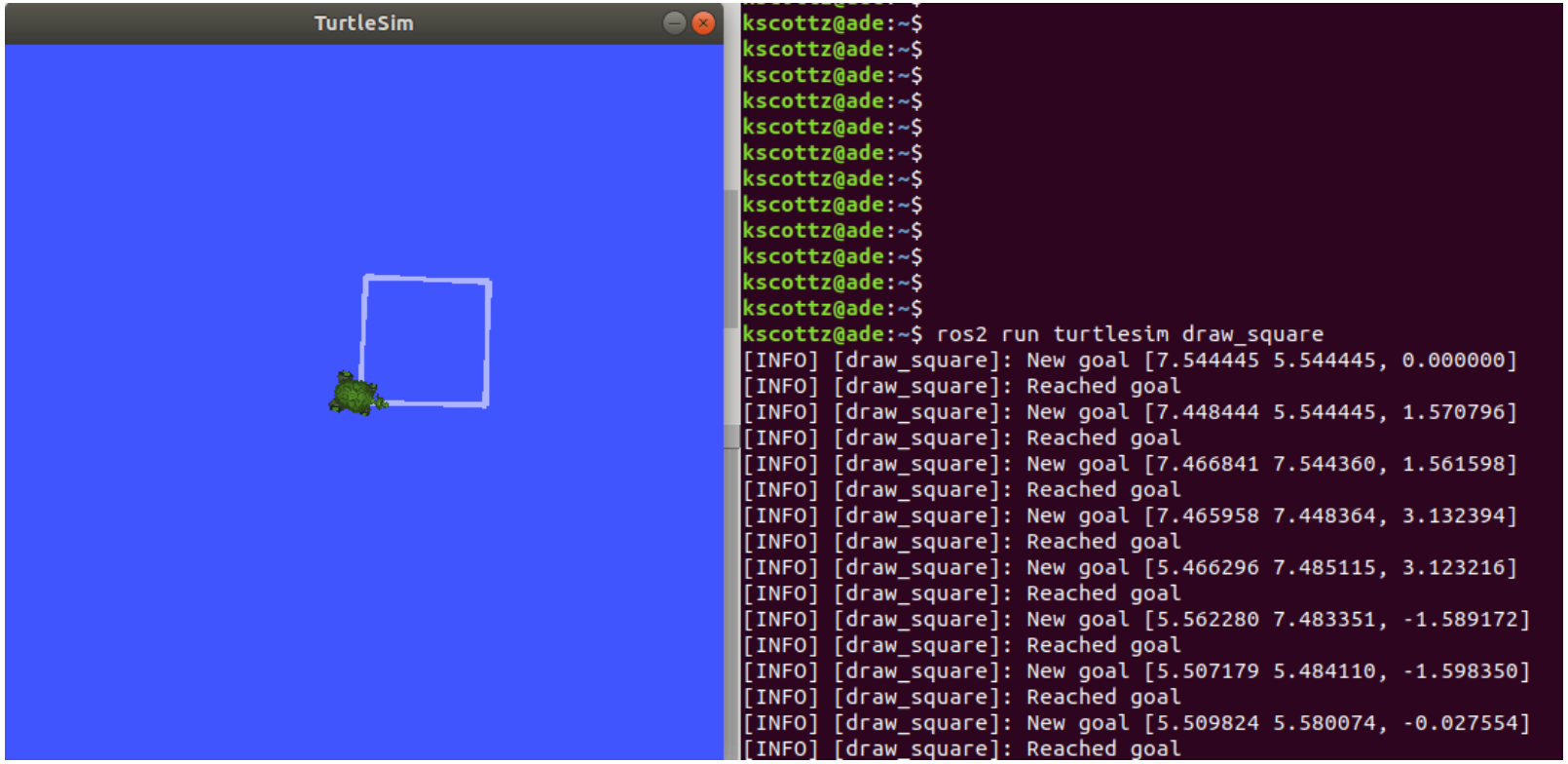
- When you run `ros2 run turtlesim turtlesim_node`, this should happen:



- This is our simple virtual turtle. Don't worry if the turtle looks different. Let's make the turtle move.
  - Press `F2` to create a new terminal.
  - Source `source /opt/ros/dashing/setup.bash`
  - We're going to run another node, let's check out this `draw_square`.
  - `ros2 run turtlesim draw_square`

Moving your turtle

If everything is setup correctly your turtle should move.



You can stop the simulation using `CTRL+C`

Let's explore what's happening

- We have two terminals open, running two "programs".
  - We have the `turtlesim` "program" running in the first terminal.
  - The `draw_square` "program" is running in a second terminal.
  - The two are communicating over ros topics.
- *What if we didn't know what was going on?*
- What if we worked with a large team and a lot of programs, or nodes, were created by our team mates?

How can we figure out what nodes are running on our simulated robot?

Inspecting nodes

- Open a new terminal by pressing `F2`
- Source your bash file `source /opt/ros/dashing/setup.bash`

Let's try inspecting our running nodes

```
kscottz@ade:~$ source /opt/ros/dashing/setup.bash

kscottz@ade:~$ ros2 node --help
  Commands:
    info  Output information about a node
    list  Output a list of available nodes

    Call `ros2 node <command> -h` for more detailed usage.

kscottz@ade:~$ ros2 node list --help
  usage: ros2 node list [-h] [--spin-time SPIN_TIME] [-a] [-c]
  Output a list of available nodes
  optional arguments:
  -h, --help            show this help message and exit
  -a, --all             Display all nodes even hidden ones
  -c, --count-nodes     Only display the number of nodes discovered
```

Let's try node list

Let's try `ros2 node list`

```
kscottz@ade:~$ ros2 node list
/draw_square  <== This is the node moving the turtle.
/turtlesim    <== This is the node rendering the turtle.
```

We can see the two nodes we started.

Can we dig down deeper into each of these nodes?

Let's try node info

Let's try this `ros2 node info` command!

```
kscottz@ade:~$ ros2 node info /draw_square
/draw_square
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /turtle1/pose: turtlesim/msg/Pose
  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /turtle1/cmd_vel: geometry_msgs/msg/Twist
  Services:
    /draw_square/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /draw_square/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /draw_square/get_parameters: rcl_interfaces/srv/GetParameters
    /draw_square/list_parameters: rcl_interfaces/srv/ListParameters
    /draw_square/set_parameters: rcl_interfaces/srv/SetParameters
    /draw_square/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
    /reset: std_srvs/srv/Empty
kscottz@ade:~$ ros2 node info /turtlesim
/turtlesim
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /turtle1/cmd_vel: geometry_msgs/msg/Twist
  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /turtle1/color_sensor: turtlesim/msg/Color
    /turtle1/pose: turtlesim/msg/Pose
    /turtle1/rotate_absolute/_action/feedback: turtlesim/action/RotateAbsolute_FeedbackMessage
    /turtle1/rotate_absolute/_action/status: action_msgs/msg/GoalStatusArray
  Services:
    /clear: std_srvs/srv/Empty
    /kill: turtlesim/srv/Kill
    /reset: std_srvs/srv/Empty
    /spawn: turtlesim/srv/Spawn
    /turtle1/rotate_absolute/_action/cancel_goal: action_msgs/srv/CancelGoal
    /turtle1/rotate_absolute/_action/get_result: turtlesim/action/RotateAbsolute_GetResult
    /turtle1/rotate_absolute/_action/send_goal: turtlesim/action/RotateAbsolute_SendGoal
    /turtle1/set_pen: turtlesim/srv/SetPen
    /turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
    /turtle1/teleport_relative: turtlesim/srv/TeleportRelative
    /turtlesim/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /turtlesim/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /turtlesim/get_parameters: rcl_interfaces/srv/GetParameters
    /turtlesim/list_parameters: rcl_interfaces/srv/ListParameters
    /turtlesim/set_parameters: rcl_interfaces/srv/SetParameters
    /turtlesim/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
```

WOW, THAT'S A LOT OF INFO!!!

- What's there?
  - Subscribers and message types.
  - Publishers and message types.

ROS topic CLI interface

- Recall from last lesson that ROS topics are short hand for the ROS pub/sub bus.
- ROS topics by analogy:
  - If you have worked with RabbitMQ or ZeroMQ it is very similar.
  - In terms of hardware if you have worked with ModBus ROS topics are the software equivalent.
  - ROS messages are basically a serialization protocol. A good analogy would be Google protobuff.
- The short of it is that ROS nodes communicate over ROS topics, which are like phone numbers that anyone can dial into and listen.
- These topics have _namespaces_ which are kinda like phone numbers or file paths. These topic names can be changed, or remapped, to connect nodes.

ros2 topic <*xxxx*>

Let's use help to see our options for this command.

In your terminal run `ros2 topic -h`

Try this:

```
kscottz@ade:~$ ros2 topic
usage: ros2 topic [-h] [--include-hidden-topics]
    Call `ros2 topic <command> -h` for more detailed usage. ...

Various topic related sub-commands
optional arguments:
-h, --help                show this help message and exit
--include-hidden-topics   Consider hidden topics as well
Commands:
  bw     Display bandwidth used by topic
  delay  Display delay of topic from timestamp in header
  echo   Output messages from a topic
  hz     Print the average publishing rate to screen
  info   Print information about a topic
  list   Output a list of available topics
  pub    Publish a message to a topic

  Call `ros2 topic <command> -h` for more detailed usage.
```

Interesting, some let us "introspect" the messages, look at performance, and even send off our own messages.

Let's look at the topics in TurtleSim

Let's start with `ros2 topic list`.

```
kscottz@ade:~$ ros2 topic list -h
usage: ros2 topic list [-h] [--spin-time SPIN_TIME] [-t] [-c]
                       [--include-hidden-topics]

Output a list of available topics
optional arguments:
-h, --help            show this help message and exit
--spin-time SPIN_TIME
                      Spin time in seconds to wait for discovery (only
                      applies when not using an already running daemon)
-t, --show-types      Additionally show the topic type
-c, --count-topics    Only display the number of topics discovered
--include-hidden-topics
                     Consider hidden topics as well
kscottz@ade:~$ ros2 topic list
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
kscottz@ade:~$
```

One thing of interest, note how `/turtle1/` is in front of the last three topics. We call this a namespace.

Digging into topics

- *Echo* is an old Unix/Linux term that basically means print. We print, or echo the data on any given topic. Let's give it a shot.
- Why don't we take a look at `/turtle1/pose/`?
- First, we'll look at the docs for echo using the `-h` or help flag.

```
kscottz@ade:~$ ros2 topic echo -h
usage: ros2 topic echo [-h] [--csv] [--full-length]
                       [--truncate-length TRUNCATE_LENGTH]
                       topic_name [message_type]
Output messages from a topic
positional arguments:
  topic_name            Name of the ROS topic to listen to (e.g. '/chatter')
  message_type          Type of the ROS message (e.g. 'std_msgs/String')
optional arguments:
  -h, --help            show this help message and exit
  --csv                 Output all recursive fields separated by commas (e.g.
                        for plotting)
  --full-length, -f     Output all elements for arrays, bytes, and string with
                        a length > '--truncate-length', by default they are
                        truncated after '--truncate-length' elements with
                        '...''
  --truncate-length TRUNCATE_LENGTH, -l TRUNCATE_LENGTH
                       The length to truncate arrays, bytes, and string to
                       (default: 128)
```

Let's echo a topic, but there are a couple things to keep in mind!

- You need to give the full path to your topic.
- *However, you can use tab complete to go fast.*

Topic echo tips / tricks

Topic echo is handy for a quick checkup to see if a piece of hardware is running and getting a sense of its position, but topics can generate a lot of data. There are some tricks to work with this data.

- You can use unix file pipes to dump the data to file.
  - `ros2 topic echo /turtle2/pose/ > MyFile.txt`
  - This will output to the file MyFile.txt
  - `CTRL+C` will still exit the program.
  - You can use `less MyFile.txt` to read the file
  - You can use grep to find a specific line.
  - Try this: `grep theta ./MyFile.txt`
- Topic echo has some nice flags that are quite handy!
  - The `--csv` flag outputs data in CSV format.
  - You will still need to use the file pipe mentioned above.
  - Example: `ros2 topic echo --csv /turtle1/pose > temp.csv`

Topic diagnostics!

Our Turtle simulation is pretty simple and doesn't generate a lot of data. Camera and LIDAR sensors for autonomous vehicles can generate so much data that they saturate network connections. It is really helpful to have some diagnostic tools. Let's look at a few.

- The `topic bw`, or bandwidth command, is used to measure the amount of bandwidth, or network capacity, that a topic uses. It requires a "window size" parameter, which is the number of messages to sample from.
- Like all CLI commands close it with `CTRL+C`

```
kscottz@ade:~$ ros2 topic bw -w 100 /turtle1/pose
Subscribed to [/turtle1/pose]
average: 1.54KB/s
     mean: 0.02KB min: 0.02KB max: 0.02KB window: 61
average: 1.51KB/s
     mean: 0.02KB min: 0.02KB max: 0.02KB window: 100
```

Topic Diagnostics

- The `topic hz` command, or hertz command, is used to measure how frequently a given topic publishes. Frequencies are usually measured in a unit of Hertz, or cycles per second.
- The `hz` command will publish the low, high, average, and standard deviation of the message publishing frequency.

```
kscottz@ade:~$ ros2 topic hz /turtle1/pose
average rate: 63.917
        min: 0.001s max: 0.017s std dev: 0.00218s window: 65
average rate: 63.195
        min: 0.001s max: 0.017s std dev: 0.00159s window: 128
```

Topic info

Another helpful command for inspecting a topic is the `info` command. The `info` command lists the number of publishers and subscribers

Let's take a quick look:

```
kscottz@ade:~$ ros2 topic info /turtle1/pose
Topic: /turtle1/pose
Publisher count: 1
Subscriber count: 1
```

Topic Info Continued

Another related tool for looking at topics is the `msg show` command. ROS topics use standard messaging formats. If you would like to know the types and format of a message this command will do that. Below is an example for TurtleSim. Be aware that this tool uses tab completion. If you know don't know where or what you are looking for it can help!

```
kscottz@ade:~$ ros2 msg show turtlesim/msg/
turtlesim/msg/Color  turtlesim/msg/Pose
kscottz@ade:~$ ros2 msg show turtlesim/msg/Pose
float32 x
float32 y
float32 theta

float32 linear_velocity
float32 angular_velocity
```

Publishing a message the hard way

- Sometimes when you are debugging and testing you need to send a message manually.
- The command is `ros2 topic pub`
- The format is as follows: `ros2 topic pub <topic_name> <msg_type> <args>`
- This command is difficult to get right as you have to write the message in YAML format.
- The `ros2 msg show` command will help with this.

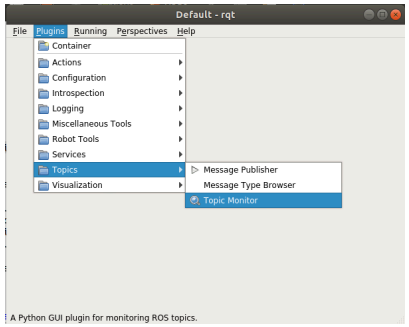
To run this command you'll need to stop the draw square node. Use F2/F3 to change to the correct screen and then enter CTRL+C

```
kscottz@ade:~$ ros2 topic pub --once /turtle1/cmd_vel geometry_msgs/msg/Twist '{linear: {x:
y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}'
publisher: beginning loop
publishing #1: geometry_msgs.msg.Twist(linear=geometry_msgs.msg.Vector3(x=2.0, y=0.0, z=0.0)
angular=geometry_msgs.msg.Vector3(x=0.0, y=0.0, z=1.8))
```

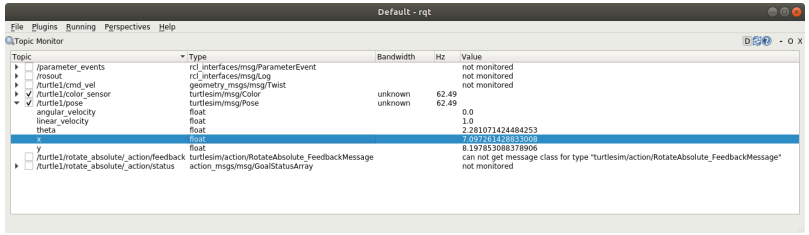
This command has a lot options that are super helpful for debugging. You can set QoS parameters for the messages, mock the sending node, and modify the publishing rate.

But there is also a GUI tool!

If the command line isn't your thing quite a few things can be accomplished via the `rqt_topic`. The rqt GUI can be started by running `rqt` in the command line. You'll want to restart the draw square node by running `ros2 run turtlesim draw_square` in the command line. You should be able to press the arrow up key to get the command back.

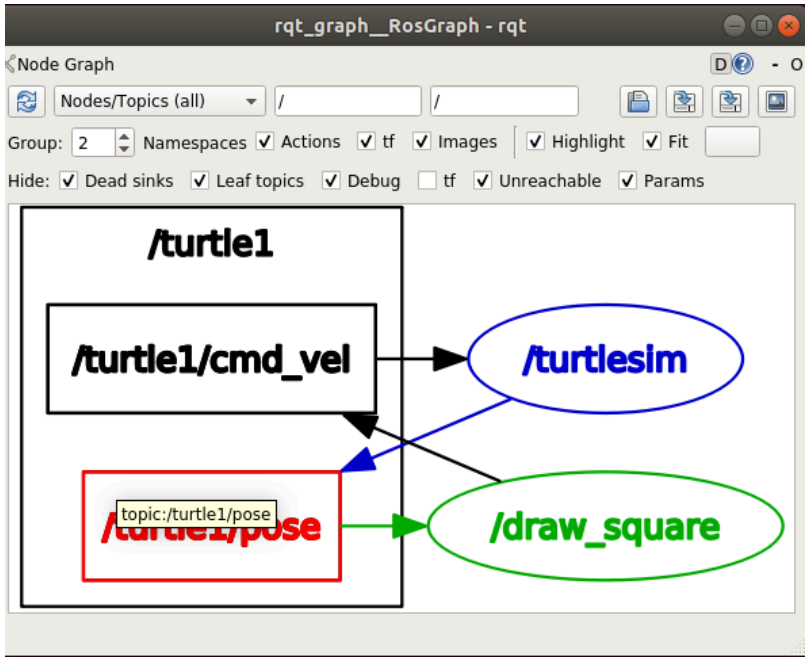


RQT starts off blank, so we'll have to turn on the topic tab by clicking `Plugins=>Topics=>Topic Monitor`. Once you do that you should see something like what's below. You may need to resize the window.



Node GUI Tools

- Understanding complex graphs as a list of node and topic names in our shell is really hard.
- Good news: we have a GUI tool!
- Type `rqt_graph` in the terminal.
- The little double arrow in the top left will load nodes.



ROS parameters

The full ROS Param tutorial can be found here.

In ROS, parameters are values that are shared between nodes in the system (if you are familiar with the blackboard design pattern in software engineering). Parameters are values that any node can query or write to, another good analogy would be global constants in normal software programs. Parameters are best used to configure your robot. For example, if you were building an autonomous vehicle and wanted to cap the maximum velocity of the vehicle at 100 km/h, you could create a parameter called "MAX_SPEED" that is visible to all the nodes.

Let's take a look at the high level param program.

```
kscottz@ade:~$ ros2 param --help
Various param related sub-commands

Commands:
  delete  Delete parameter
  get     Get parameter
  list    Output a list of available parameters
  set     Set parameter
  Call `ros2 param <command> -h` for more detailed usage.
```

Params used by TurtleSim

Let's see what the docs say and then see what happens when we call `ros2 param list`

```
kscottz@ade:~$ ros2 param --help
usage: ros2 param [-h]
optional arguments:
  use_sim_time
/turtlesim:
  background_b
  background_g
  background_r
usage: ros2 param list [-h] [--spin-time SPIN_TIME] [--include-hidden-nodes]

positional arguments:
  node_name             Name of the ROS node
< CLIPPED >

kscottz@ade:~$ ros2 param list
/draw_square:
  use_sim_time
/turtlesim:
  background_b
  background_g
  background_r
  use_sim_time
```

Let's try getting/setting parameters

The syntax for getting a parameter is as follows:

```
ros2 param get <node name> <param name>
```

Let's give it a shot.

```
kscottz@ade:~$ ros2 param get /turtlesim background_b
Integer value is: 255
```

Let's try setting a parameter. The syntax for that is as follows:

```
ros2 set <node name> <param name> <value>
```

```
kscottz@ade:~$ ros2 param set /turtlesim background_b 0
Set parameter successful
```

Note that THIS SEEMS TO BE BROKEN!?

Services

- The full ROS 2 Services tutorials can be found here.
- ROS2 Services, as we have discussed previously, are another level of extraction built on top of ROS 2 topics.
- At its core, a service is just an API for controlling a robot task.
- A good analogy for ROS Services are remote procedure calls .
- Another good analogy for services would be making an REST API call.
- Curling a remote REST API endpoint to query data on a remote server is very similar to a ROS service.
- Essentially the ROS API allows every node to publish a list of services, and subscribe to services from other nodes.

Services Continued

- The root command for ROS services is the `ros2 service` command.
- Just like all the other commands we have looked at, let's run `ros2 service --help` to see what we can do.
- There is an important distinction between ros2 srv and ros2 service.
- The former is for installed services while the latter is for running services.

We'll focus on the latter, but `srv` is very similar.

```
kscottz@ade:~$ ros2 service --help
usage: ros2 service [-h] [--include-hidden-services]
                    Call `ros2 service <command> -h` for more detailed usage.

Commands:
  call  Call a service
  list  Output a list of available services
```

- Services look fairly straight forward, with only two commands, `list` and `call`.

Listing available services

Let's take a look at what we can do with `ros2 service list`.

```
kscottz@ade:~$ ros2 service list --help
usage: ros2 service list [-h] [--spin-time SPIN_TIME] [-t] [-c]

Output a list of available services

optional arguments:

   -t, --show-types      Additionally show the service type
   -c, --count-services  Only display the number of services discovered
```

This command is fairly straight forward with only two utility flags. Let's use the `-t` flag

```
kscottz@ade:~$ ros2 service list -t
/clear [std_srvs/srv/Empty]
/draw_square/describe_parameters [rcl_interfaces/srv/DescribeParameters]
/draw_square/get_parameter_types [rcl_interfaces/srv/GetParameterTypes]
/draw_square/get_parameters [rcl_interfaces/srv/GetParameters]
/draw_square/list_parameters [rcl_interfaces/srv/ListParameters]
/draw_square/set_parameters [rcl_interfaces/srv/SetParameters]
/draw_square/set_parameters_atomically [rcl_interfaces/srv/SetParametersAtomically]
/kill [turtlesim/srv/Kill]
/reset [std_srvs/srv/Empty]
/spawn [turtlesim/srv/Spawn]
... SNIP ...
/turtlesim/list_parameters [rcl_interfaces/srv/ListParameters]
/turtlesim/set_parameters [rcl_interfaces/srv/SetParameters]
/turtlesim/set_parameters_atomically [rcl_interfaces/srv/SetParametersAtomically]
```

Calling a ROS 2 service

Let's explore the `ros2 service call` command.

```
kscottz@ade:~$ ros2 service call -h
usage: ros2 service call [-h] [-r N] service_name service_type [values]

Call a service
positional arguments:
  service_name    Name of the ROS service to call to (e.g. '/add_two_ints')
  service_type    Type of the ROS service (e.g. 'std_srvs/srv/Empty')
  values          Values to fill the service request with in YAML format (e.g.
                  "{a: 1, b: 2}"), otherwise the service request will be
                  published with default values

optional arguments:
  -r N, --rate N  Repeat the call at a specific rate in Hz
```

The format is pretty straight forward:

```
ros2 service call <service_name> <service_type> [values]
```

Basic example, blank services.

- If we look at the list of services we see a `/reset/` service that has the type `[std_srvs/srv/Empty]`.
- What this means is that this service can be called with an empty message.
- It is worth noting that a empty message still has a type, it is just that the type is empty.
- Our turtle has been drawing a box for a while, why don't we see if we can reset the screen?
    - First kill the draw_square node. Use `F3` to go to the right window.
    - Now use `CTRL+C` to stop the program.

Why don't we give it a call. The empty service message can be found in `std_srvs/srv/Empty`, thus our call is as follows:

```
kscottz@ade:~$ ros2 service call /reset std_srvs/srv/Empty
waiting for service to become available...
requester: making request: std_srvs.srv.Empty_Request()

response:
std_srvs.srv.Empty_Response()
```

Service call result

The service reset the screen, and changed our turtle icon!

Try toggling the `draw_square` program and the `reset` service a few times.

More complex service calls

Next we're going to try a more complex service call that requires an actual message. For this example we'll use the spawn service that creates a new turtle.

The spawn service, looking at our `ros2 service list` call uses a `[turtlesim/srv/Spawn]` message.

The best way to determine the name of a service is to use the `srv` verb in ROS 2.

The way we do this is running `ros2 srv show turtlesim/srv/Spawn`.

```
kscottz@ade:~$ ros2 srv show turtlesim/srv/Spawn
float32 x
float32 y
float32 theta
string name # Optional.  A unique name will be created and returned if this is empty
---
string
```

We can see now that this message takes an x,y position, an angle theta, and an optional name. The service will return a string (as noted by the string below the `---`)

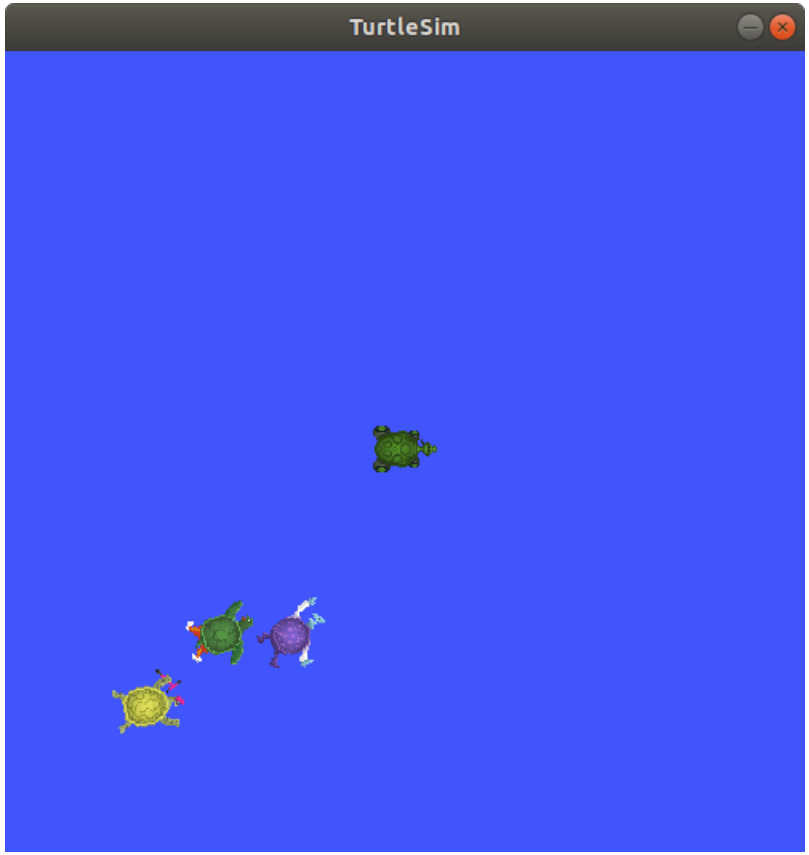
Services with complex messages

The format of the message is YAML inside quotation marks. Following from the information above let's make a few turtles.

```
string namekscottz@ade:~$ ros2 service call /spawn turtlesim/srv/Spawn "{x: 2, y: 2, theta:
waiting for service to become available...
requester: making request: turtlesim.srv.Spawn_Request(x=2.0, y=2.0, theta=0.2, name='larry'

response:
turtlesim.srv.Spawn_Response(name='larry')

kscottz@ade:~$ ros2 service call /spawn turtlesim/srv/Spawn "{x: 3, y: 3, theta: 0.3, name:
waiting for service to become available...
requester: making request: turtlesim.srv.Spawn_Request(x=3.0, y=3.0, theta=0.3, name='moe')

response:
turtlesim.srv.Spawn_Response(name='moe')

kscottz@ade:~$ ros2 service call /spawn turtlesim/srv/Spawn "{x: 4, y: 3, theta: 0.4, name:
waiting for service to become available...
requester: making request: turtlesim.srv.Spawn_Request(x=4.0, y=3.0, theta=0.4, name='curly'

response:
turtlesim.srv.Spawn_Response(name='curly')

kscottz@ade:~$
```

Service call results!

If everything went well we should see something like this.



We've now created four turtles!

ROS action CLI

ROS Actions and Services are very similar in terms of what they do and likewise their APIs are also fairly similar.

ROS actions are the prefered tool for *asynchronus* tasks while services are the preffered means of deploying *synchronus* tasks.

In more practical terms services should be used for quick, short tasks, while actions should be used for long term behaviors (like moving to a waypoint).

The other big difference between actions and services, is that actions can send periodic updates about their progress.

```
kscottz@ade:~$ ros2 action -h

Various action related sub-commands

Commands:
  info       Print information about an action
  list       Output a list of action names
  send_goal  Send an action goal
  show       Output the action definition
```

Looks familiar! Let's dif into list, and info.

Actions: list & info

Let's see what actions are availabe to us using `ros2 action list`

```
kscottz@ade:~$ ros2 action list
/curly/rotate_absolute
/larry/rotate_absolute
/moe/rotate_absolute
/turtle1/rotate_absolute
```

We see each of our turtles have one service called `rotate_absolute`. Let's dig into this action using the info verb. This command has a `-t` flag to list the types of messages.

```
kscottz@ade:~$ ros2 action info /moe/rotate_absolute -t
Action: /moe/rotate_absolute
Action clients: 0
Action servers: 1
  /turtlesim [turtlesim/action/RotateAbsolute]
```

Interesting, what do these terms mean. The first line lists the action name. The second line gives the current number of clients for the action. The `Action servers` line gives the total number of action servers for this action. The last line gives the package and message type for the action.

Calling an action and giving it a goal

Let's take a look at the `ros2 action send_goal` help command.

```
kscottz@ade:~$ ros2 action send_goal -h
usage: ros2 action send_goal [-h] [-f] action_name action_type goal

Send an action goal
positional arguments:
  action_name     Name of the ROS action (e.g. '/fibonacci')
  action_type     Type of the ROS action (e.g. 'example_interfaces/action/Fibonacci')
  goal            Goal request values in YAML format (e.g. '{order: 10}')

optional arguments:
  -f, --feedback  Echo feedback messages for the goal
```

We can see here that we need to know the action name, the type, and the values. Now the only problem is figuring out the format of the action_type.

Let's understand the RotateAbsolute action message

The `ros2 action show` command can be used to find the type of action message. Let's take a look.

```
kscottz@ade:~$ ros2 action show turtlesim/action/RotateAbsolute
# The desired heading in radians
float32 theta  #< --- This section is the GOAL
---
# The angular displacement in radians to the starting position
float32 delta  #< --- This section is the final result, different from the goal.
---
# The remaining rotation in radians
float32 remaining # < --- This is the current state.
```

What does this say about rotate absolute?

- There is a float input, `theta` the desired heading. This first section is the actual goal.
- `delta` -- the angle from the initial heading. This is the value returned when the action completes.
- `remaining` -- the remaining radians to move. This is the value posted by the action while the action is being done.

Executing the action

With this information we can create our call to the action server. We'll use the `-f` flag to make this a bit clearer.

Keep an eye on your turtle! It should move, slowly.

```
kscottz@ade:~$ ros2 action send_goal -f /turtle1/rotate_absolute turtlesim/action/RotateAbsc
Waiting for an action server to become available...
Sending goal:
  theta: 1.7

Feedback:
  remaining: 0.11599969863891602

Goal accepted with ID: 35c40e91590047099ae5bcc3c5151121

Feedback:
 remaining: 0.09999966621398926

Feedback:
 remaining: 0.06799960136413574

Feedback:
 remaining: 0.03599953651428223

Result:
 delta: -0.09600019454956055

Goal finished with status: SUCCEEDED
```

ROS Bag!

- ROS Bags are ROS's tool for recording, and replaying data.
- ROSBags are kinda like log files that let you store data along with messages.
- ROS systems can generate a lot of data, so you select which topics you want to bag.
- Bags are a great tool for testing and debugging your application as well.

Let's take a look at the base `bag` verb.

```
kscottz@ade:~$ ros2 bag -h
usage: ros2 bag [-h] Call `ros2 bag <command> -h` for more detailed usage. ...

Various rosbag related sub-commands

Commands:
  info    ros2 bag info
  play    ros2 bag play
  record  ros2 bag record
```

Let's try recording our first Bag

First use `F2` or `F3` to go to the other terminal. Start the `draw_square` demo again to get the default turtle moving.

The command for that is: `ros2 run turtlesim draw_square`

Now let's look at `ros2 bag -h`

```
kscottz@ade:~$ ros2 bag record -h
usage: ros2 bag record [-h] [-a] [-o OUTPUT] [-s STORAGE]
                       [-f SERIALIZATION_FORMAT] [--no-discovery]
                       [-p POLLING_INTERVAL]
                       [topics [topics ...]]
ros2 bag record
positional arguments:
  topics                topics to be recorded
optional arguments:
  -a, --all             recording all topics, required if no topics are listed explicitly.
  -o OUTPUT, --output OUTPUT
                        destination of the bagfile to create, defaults to a
                        timestamped folder in the current directory
  -s STORAGE, --storage STORAGE
                        storage identifier to be used, defaults to "sqlite3"
  -f SERIALIZATION_FORMAT, --serialization-format SERIALIZATION_FORMAT
                        rmw serialization format in which the messages are
                        saved, defaults to the rmw currently in use
```

Let's Bag!

- Let's bag the pose data on the `/turtle1/pose topic`
- Save the data to the directory `turtle1.bag` using the `-o` flag.
- The program will bag until you hit `CTRL+C`. Give it a good 30 seconds.

Here's my example.

```
kscottz@ade:~$ ros2 bag record /turtle1/pose -o turtle1
[INFO] [rosbag2_storage]: Opened database 'turtle1'.
[INFO] [rosbag2_transport]: Listening for topics...
[INFO] [rosbag2_transport]: Subscribed to topic '/turtle1/pose'
[INFO] [rosbag2_transport]: All requested topics are subscribed. Stopping discovery...
^C[INFO] [rclcpp]: signal_handler(signal_value=2)
```

Let's inspect our Bag.

You can introspect any bag file using the `ros2 bag info` command. This command will list the messages in the bag, the duration of file, and the number of messages.

```
kscottz@ade:~$ ros2 bag info turtle1
Files:             turtle1.db3
Bag size:          268.4 KiB
Storage id:        sqlite3
Duration:          68.705s
Start:             May  4 2020 16:10:26.556 (1588633826.556)
End                May  4 2020 16:11:35.262 (1588633895.262)
Messages:          4249
Topic information: Topic: /turtle1/pose | Type: turtlesim/msg/Pose | Count: 4249 | Serializa
```

Replaying a Bag

Bags are a great tool for debugging and testing. You can treat a ROS bag like a recording of a running ROS system. When you play a bag file you can use most of the ros2 cli tools to inspect the recorded topics.

To replay the bag, first use `F2/F3` and `CTRL+C` to turn off the main turtle node and the `draw_square` node.

Now in a new terminal replay the bag file using the following command:

```
kscottz@ade:~$ ros2 bag play turtle1
[INFO] [rosbag2_storage]: Opened database 'turtle1'.
```

Nothing should happen visibly, but a lot is happening under the hood. Use `F2` or `F3` to go to a second terminal. Just like a running robot, you should be able `list` and `echo` topics.

```
kscottz@ade:~ros2 topic list
/parameter_events
/rosout
/turtle1/pose

kscottz@ade:~$ ros2 bag info turtle1
x: 3.8595714569091797
y: 3.6481313705444336
theta: -1.2895503044128418
linear_velocity: 1.0
angular_velocity: 0.0
---
```

Pretty cool right?

You can kill the bag file with `CTRL+C`.

Homework?!

- The TurtleBot comes from a long line of turtle tutorials.
- The original one was the Logo programming language for computer graphics.
- I would recommend using the turtle to make some cool graphics.
- Here's an example of what people did with LOGO..