Introduction to ROS2

Penn Engineering

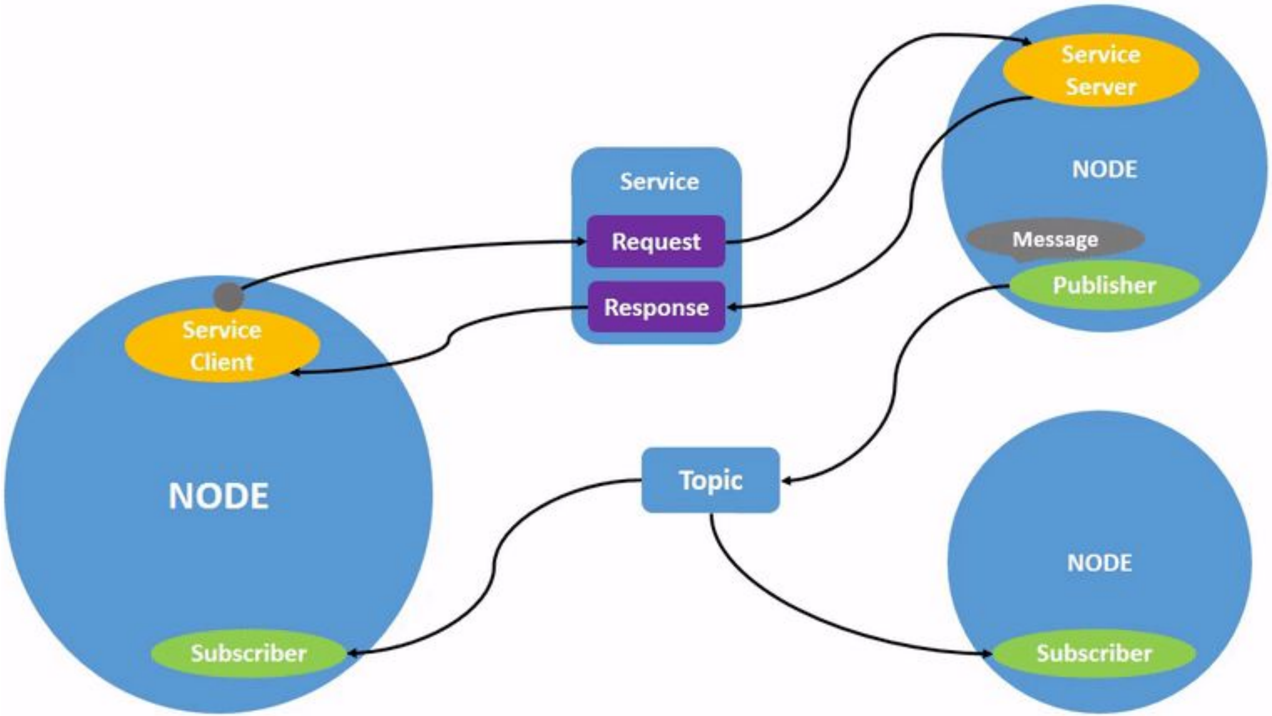
Overview

- ROS: Robot Operating System
- Peer to peer
- Distributed
- Multi-lingual
- Light-weight
- Free and open-source

ROS1 v.s. ROS2

- Language standards: at least C++11 and Python3 for ROS2
- Using off the shelf middleware. Now supports discovery, transport and serialization over DDS.
- Tighter Python integration
- Real time capabilities
- **API change**

ROS graph



The ROS graph is a network of ROS 2 elements processing data together at one time. It encompasses all executables and the connections between them if you were to map them all out and visualize them.

Penn Engineering

https://docs.ros.org/en/foxy/Tutorials/Understanding-ROS2-Nodes.html

Nodes

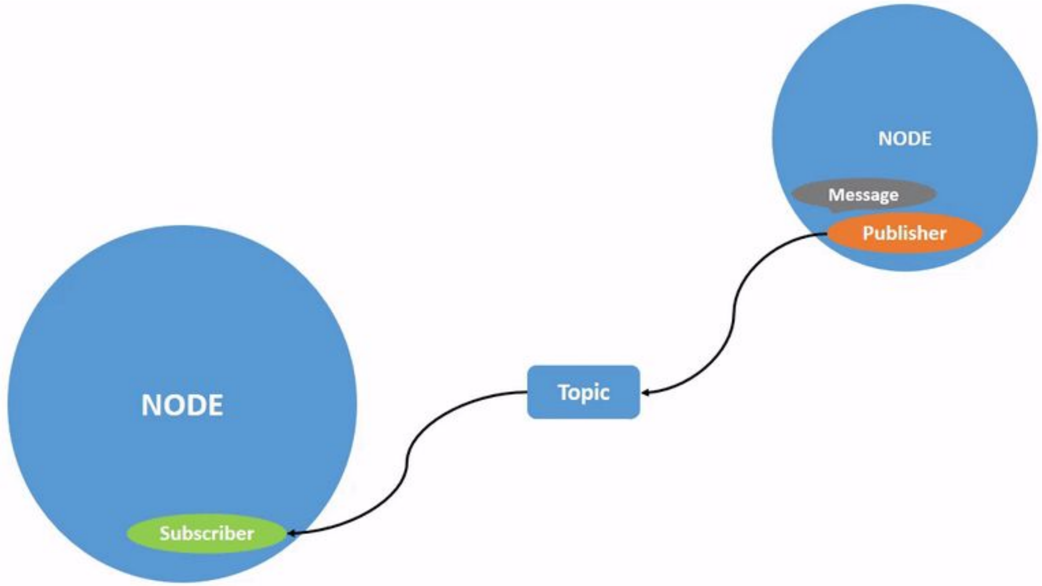
Each node in ROS should be responsible for a single, module purpose (e.g. one node for controlling wheel motors, one node for controlling a laser range-finder, etc). Each node can send and receive data to other nodes via topics, services, actions, or parameters.

Related command line commands

- `ros2 run <package_name> <executable_name>`
- `ros2 node list`
- `ros2 node info <node_name>`

https://docs.ros.org/en/foxy/Tutorials/Understanding-ROS2-Nodes.html

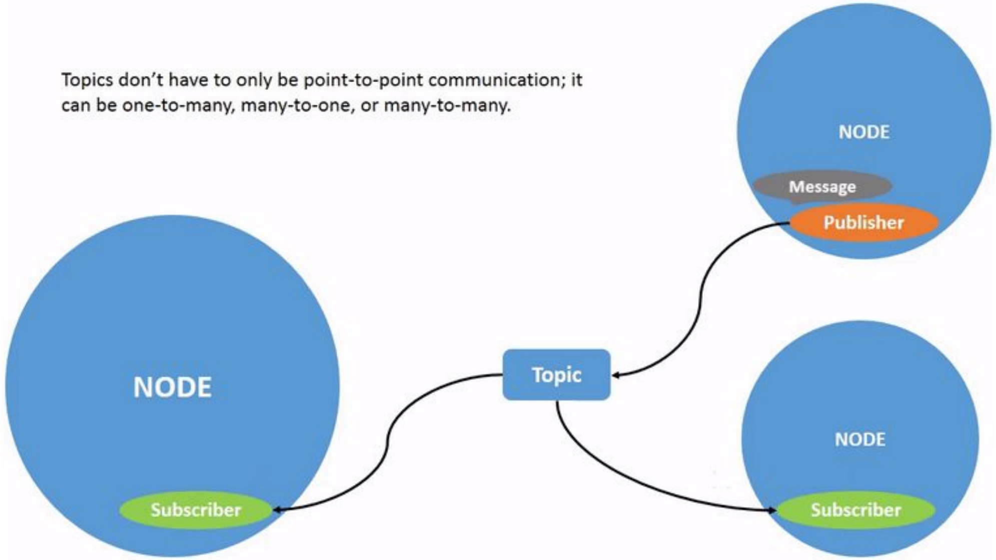
Topics



ROS 2 breaks complex systems down into many modular nodes. Topics are a vital element of the ROS graph that act as a bus for nodes to exchange messages.

https://docs.ros.org/en/foxy/Tutorials/Topics/Understanding-ROS2-Topics.html

Topics



A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics.

Topics are one of the main ways in which data is moved between nodes and therefore between different parts of the system.

Penn Engineering

https://docs.ros.org/en/foxy/Tutorials/Topics/Understanding-ROS2-Topics.html

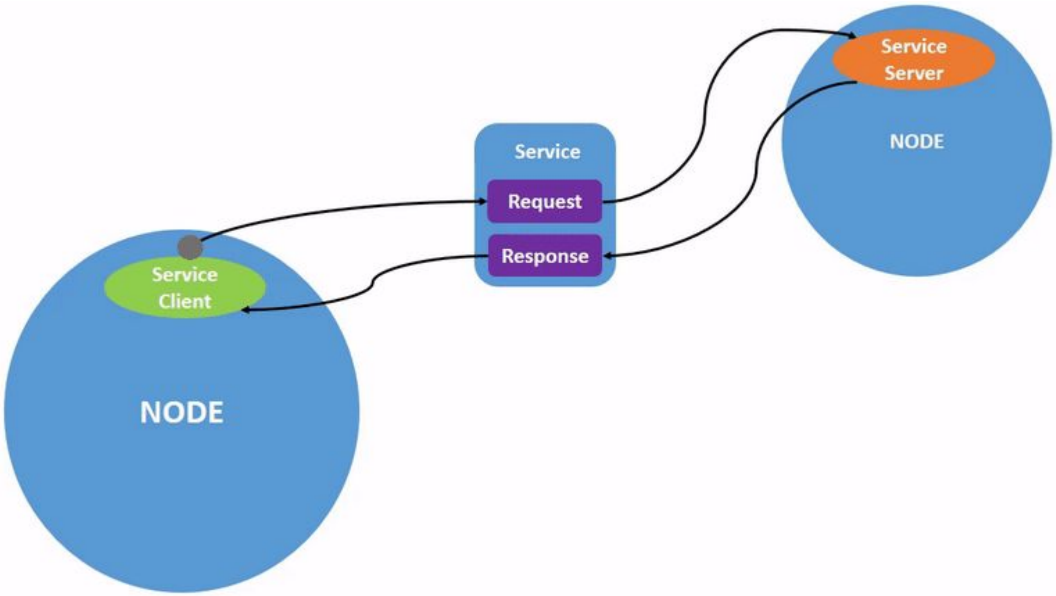
Topics

Related command line commands

- `rqt_graph`
- `ros2 topic list`
- `ros2 topic list -t`
- `ros2 topic echo <topic_name>`
- `ros2 topic info <topic_name>`
- `ros2 interface show <msg_type>`
- `ros2 topic pub <topic_name> <msg_type> ‘<args>’`
- `ros2 topic hz <topic_name>`

https://docs.ros.org/en/foxy/Tutorials/Topics/Understanding-ROS2-Topics.html

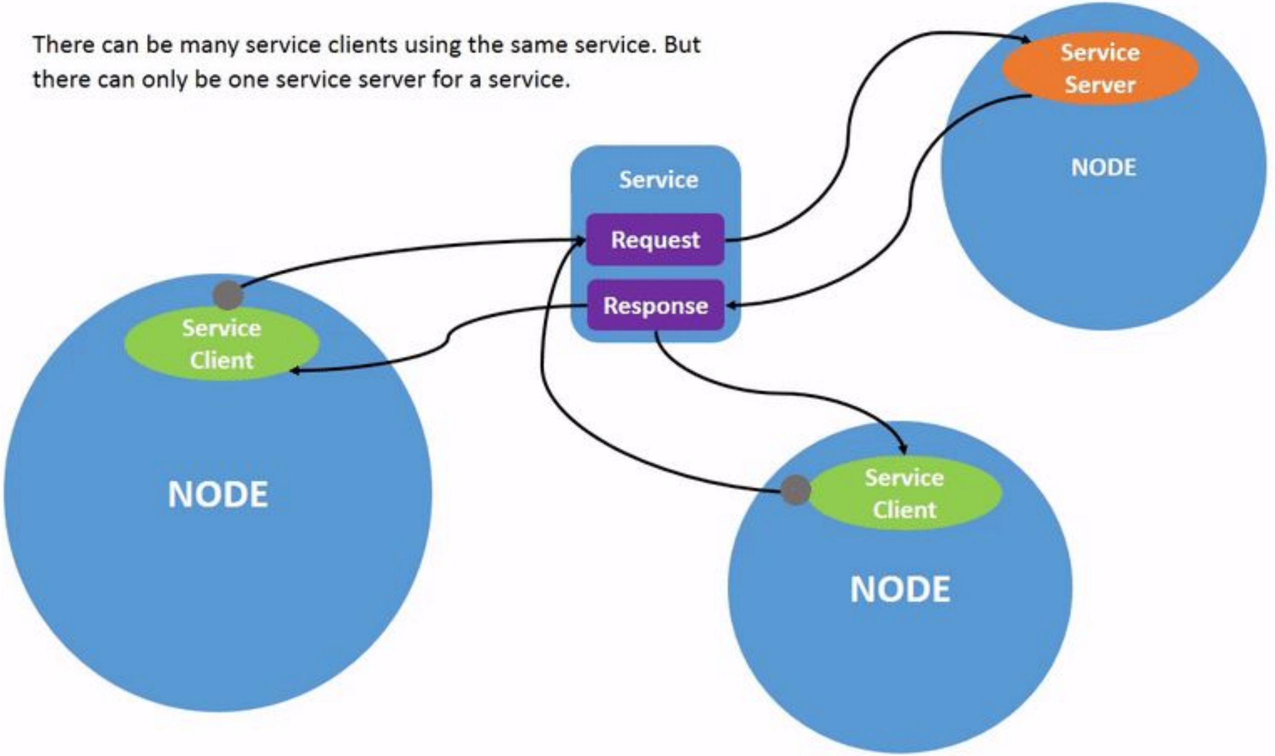
Services



Services are another method of communication for nodes in the ROS graph. Services are based on a call-and-response model, versus topics' publisher-subscriber model. While topics allow nodes to subscribe to data streams and get continual updates, services only provide data when they are specifically called by a client.

https://docs.ros.org/en/foxy/Tutorials/Services/Understanding-ROS2-Services.html

Services



Penn Engineering

https://docs.ros.org/en/foxy/Tutorials/Services/Understanding-ROS2-Services.html

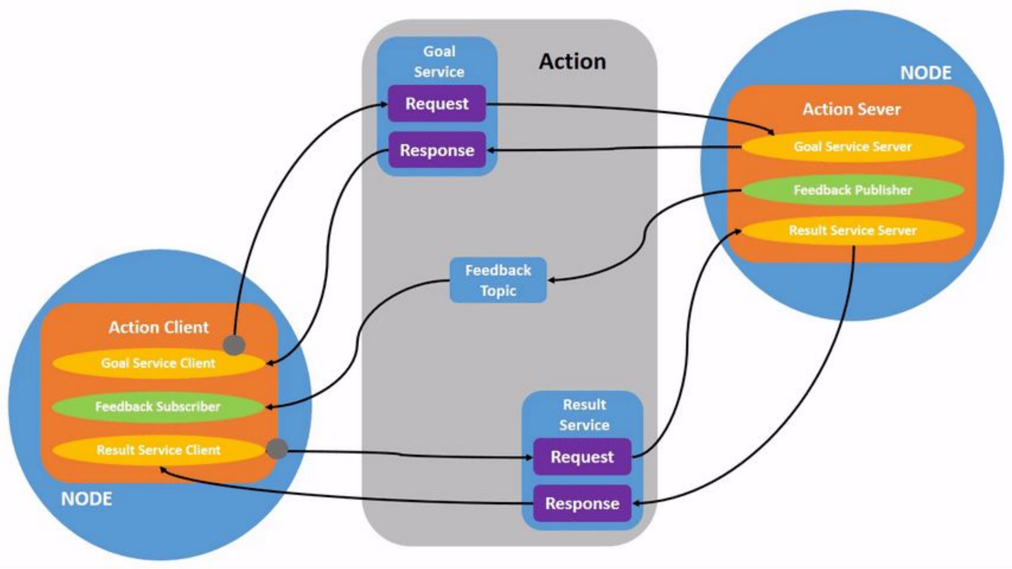
Services

Related command line commands

- `ros2 service list`
- `ros2 service type <service_name>`
- `ros2 service list -t`
- `ros2 service find <type_name>`
- `ros2 interface show <type_name>.srv`
- `ros2 service call <service_name> <service_type> <arguments>`

https://docs.ros.org/en/foxy/Tutorials/Services/Understanding-ROS2-Services.html

Actions



Actions are one of the communication types in ROS 2 and are intended for long running tasks. They consist of three parts: a goal, feedback, and a result.

Actions are built on topics and services. Their functionality is similar to services, except actions are preemptable (you can cancel them while executing). They also provide steady feedback, as opposed to services which return a single response.

Actions use a client-server model, similar to the publisher-subscriber model (described in the topics tutorial). An "action client" node sends a goal to an "action server" node that acknowledges the goal and returns a stream of feedback and a result.

Penn Engineering

https://docs.ros.org/en/foxy/Tutorials/Understanding-ROS2-Actions.html

Parameters

A parameter is a configuration value of a node. You can think of parameters as node settings. A node can store parameters as integers, floats, booleans, strings and lists. In ROS 2, each node maintains its own parameters. All parameters are dynamically reconfigurable, and built off of ROS 2 services.

Related command line commands

- `ros2 param list`
- `ros2 param get <node_name> <parameter_name>`
- `ros2 param set <node_name> <parameter_name> <value>`

Could also use an yaml file to define the parameters.

https://docs.ros.org/en/foxy/Tutorials/Parameters/Understanding-ROS2-Parameters.html

Parameters

- Parameters could also be set in either a launch file or a yaml file.
- Useful to have all parameters in one place while tuning.
- Set parameters for multiple nodes in one file.
- For a full length tutorial: https://roboticsbackend.com/ros2-yaml-params/

Parameters

Getting ROS parameters programmatically:

- In a node, declare a parameter first
  - `self.declare_parameter('my_param')`
- Then getting a parameter
  - `self.get_parameter('my_param')`
- You could also get multiple parameters at once
- Similar in C++
- For full tutorials on parameters see:
  - https://roboticsbackend.com/rclpy-params-tutorial-get-set-ros2-params-with-python/
  - https://roboticsbackend.com/rclcpp-params-tutorial-get-set-ros2-params-with-cpp/

Workspace

A workspace is a directory containing ROS 2 packages. Before using ROS 2, it's necessary to source your ROS 2 installation workspace in the terminal you plan to work in. This makes ROS 2's packages available for you to use in that terminal.

You also have the option of sourcing an "overlay" – a secondary workspace where you can add new packages without interfering with the existing ROS 2 workspace that you're extending, or "underlay". Your underlay must contain the dependencies of all the packages in your overlay. Packages in your overlay will override packages in the underlay. It's also possible to have several layers of underlays and overlays, with each successive overlay using the packages of its parent underlays.

Workspace

- Defines context for the current workspace
- Creating a new workspace
  - `$ mkdir -p <your_workspace>/src`
  - Then you can put your desired ROS2 packages inside src
- Resolving dependencies
  - `$ cd <your_workspace>`
  - `$ rosdep install -i --from-path src --rosdistro foxy -y`

Workspace

- Build the workspace with `colcon`:
  - From the root of your workspace: `$ colcon build`
  - Useful arguments when building:
    - `--packages-up-to`: builds the package you want, plus all its dependencies, but not the whole workspace. This will save some time if you don't need all the packages in the workspace.
    - `--symlink-install`: saves you from having to rebuild every time you tweak Python scripts
    - `--event-handlers console_direct+`: shows console output while building (can otherwise be found in the `log` directory)
- Once build finishes, you'll see `build` `install` `log` `src` directories in your workspace. The `install` directory is where your workspace's setup files are.

Overlays and Underlays

- We refer to the main ROS 2 environment as the **underlay**. This has all the necessary setup to run ROS 2. Your workspaces are referred to as **overlays**. By sourcing your overlays you get access to your packages on top of the base ROS 2 environment.
- After building, in a new terminal, source the underlay by:
  - `$ source /opt/ros/foxy/setup.bash`
- And in the root of your desired workspace:
  - `$ cd <your_workspace>`
  - `$ source install/local_setup.bash`
  - Note that there's also a `install/setup.bash` in your workspace. You can also source this instead. This is equivalent to sourcing both your workspace overlay and the underlay your workspace was created/built in.
- You might be tempted to leave these sourcing commands in your bashrc. I highly recommend **against** it. I've seen too many times that a teammate working on a package wondering why their code isn't working. And it ended up being `bashrc` sourcing the wrong overlay.

ROS 2 Packages

- A package can be considered a container for your ROS 2 code. If you want to be able to install your code or share it with others, then you'll need it organized in a package. With packages, you can release your ROS 2 work and allow others to build and use it easily.
- Package creation in ROS 2 uses ament as its build system and colcon as its build tool. You can create a package using either CMake or Python, which are officially supported, though other build types do exist.

ROS 2 Packages

- Python and CMake packages each have their own minimum requirements.
- CMake packages:
  - `package.xml`: file containing meta info about the package
  - `CMakeLists.txt`: file describing how to build the code within the package
- Python packages:
  - `package.xml`: file containing meta info about the package
  - `setup.py`: contains instructions for how to install the package
  - `setup.cfg`: required when a package has executables, so ros2 run can find them
  - `/<package_name>`: a directory with the same name as your package, used by ROS 2 tools to find your package, contains `__init__.py`

ROS 2 Packages

The simplest package may have a file structure that looks like:

- CMake:
  - `my_package/`
    - `CMakeLists.txt`
    - `package.xml`
- Python:
  - `my_package/`
    - `setup.py`
    - `package.xml`
    - `resource/my_package`

ROS 2 Packages

- A single workspace can contain as many packages as you want, each in their own folder. You can also have packages of different build types in one workspace (CMake, Python, etc.). You cannot have nested packages.
- Best practice is to have a src folder within your workspace, and to create your packages in there. This keeps the top level of the workspace "clean".

ROS 2 Packages

A workspace with multiple packages might look like this:

```
workspace_folder/
    src/
      package_1/
          CMakeLists.txt
          package.xml

      package_2/
          setup.py
          package.xml
          resource/package_2
      ...
      package_n/
          CMakeLists.txt
          package.xml
```

ROS 2 Packages

- Creating a package:
  - `$ cd <your_workspace>/src`
  - (Python packages): `$ ros2 pkg create --build-type ament_python <package_name>`
  - (CMake packages): `$ ros2 pkg create --build-type ament_cmake <package_name>`

Package contents

- CMake packages
  - `CMakeLists.txt, include, package.xml, src`
  - Node source files (.cpp) are in src, and headers files (.h) in include
- Python packages
  - `my_package, package.xml, resource, setup.cfg, setup.py, test`
  - Node source files (.py) are inside the `my_package` directory

Customizing package.xml

- Fill in name and email on `maintainer` line, edit description to `summarize` the package, update the `license` line.
- Fill in your dependencies under the `_depend` tags. For documentation on what types of depend tags, see: https://www.ros.org/reps/rep-0149.html#build-depend-multiple

Penn Engineering

Customizing setup.py

- For Python packages, you'll also need to fill in setup.py
- Fill in the same description, maintainer, and license fields as in `package.xml`. You'll need to match these exactly. You'll also need to match the `package_name` and `version`.

Publisher (Python)

```python
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'topic',
10)
        timer_period = 0.5  # seconds
        self.timer = self.create_timer(timer_period,
self.timer_callback)
        self.i = 0
    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1
```

```python
def main(args=None):
    rclpy.init(args=args)
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
if __name__ == '__main__':
    main()
```


https://docs.ros.org/en/foxy/Tutorials/Writing-A-Simple-Py-Publisher-And-Subscriber.html

Adding Dependencies and Entrypoint

- Filling in the dependencies in package.xml for our example node:
  - `<exec_depend>rclpy</exec_depend>`
  - `<exec_depend>std_msgs</exec_depend>`
  - This declares the package needs `rclpy` and `std_msgs` when code is executed.
- Add an entrypoint in setup.py:

```
entry_points={
        'console_scripts': [
                'talker = py_pubsub.publisher_member_function:main',
        ],
},
```

Installing Dependencies

- ROS 2 uses `rosdep` to install package dependencies.
- https://index.ros.org/ maintains repos and packages you can use as dependencies and has recipes for installation for `rosdep`.
- To install dependencies for a workspace, in the workspace directory:
  - `rosdep install -i --from-path src --rosdistro foxy -y`
  - This will install dependencies declared in `package.xml` from all packages in the `src` directory for ROS 2 foxy

Subscriber (Python)

```python
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalSubscriber(Node):
    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String,
            'topic',
            self.listener_callback,
            10)
        self.subscription  # prevent unused variable warning
    def listener_callback(self, msg):
        self.get_logger().info('I heard: "%s"' % msg.data)
```

```python
def main(args=None):
    rclpy.init(args=args)

    minimal_subscriber = MinimalSubscriber()

    rclpy.spin(minimal_subscriber)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

Adding Entrypoint

- Since we've already added the dependencies to `package.xml`, we'll only need to add the entrypoint for the subscriber here:
- ```
entry_points={
        'console_scripts': [
                'talker = py_pubsub.publisher_member_function:main',
                'listener = py_pubsub.subscriber_member_function:main',
        ],
},
```
```

Launch files

- Launch files allow you to start up and configure a number of executables containing ROS 2 nodes simultaneously.
- Running a single launch file with the `ros2 launch` command will start up your entire system - all nodes and their configurations - at once.
- In your package, create a new directory for launch files: `$ mkdir launch`
- Then create your launch file: `$ touch <your_launch>.py`
- Note that launch files are written in Python now in ROS 2 instead of xml in ROS 1. This allows for access to Python libraries.

https://roboticsbackend.com/ros2-launch-file-example/

Example Launch file

```python
from launch import LaunchDescription
from launch_ros.actions import Node
from launch.substitutions import Command
from ament_index_python.packages import get_package_share_directory
import os
import yaml

def generate_launch_description():
    ld = LaunchDescription()
    config = os.path.join(
        get_package_share_directory('f1tenth_gym_ros'),
        'config',
        'sim.yaml'
        )
    config_dict = yaml.safe_load(open(config, 'r'))

    bridge_node = Node(
        package='f1tenth_gym_ros',
        executable='gym_bridge',
        name='bridge',
        parameters=[config]
    )
    rviz_node = Node(
        package='rviz2',
        executable='rviz2',
        name='rviz',
        arguments=['-d', os.path.join(get_package_share_directory('f1tenth_gym_ros'), 'launch', 'gym_bridge.rviz')]
    )
```

Example Launch file

```
    map_server_node = Node(
        package='nav2_map_server',
        executable='map_server',
        parameters=[{'yaml_filename': config_dict['bridge']['ros__parameters']['map_path'] + '.yaml'},
                    {'topic': 'map'},
                    {'frame_id': 'map'},
                    {'output': 'screen'},
                    {'use_sim_time': True}]
    )
    nav_lifecycle_node = Node(
        package='nav2_lifecycle_manager',
        executable='lifecycle_manager',
        name='lifecycle_manager_localization',
        output='screen',
        parameters=[{'use_sim_time': True},
                    {'autostart': True},
                    {'node_names': ['map_server']}]
    )
    ego_robot_publisher = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='ego_robot_state_publisher',
        parameters=[{'robot_description': Command(['xacro ', os.path.join(get_package_share_directory('f1tenth_gym_ros'),
'launch', 'ego_racecar.xacro')])}],
        remappings=[('/robot_description', 'ego_robot_description')]
    )
```

Example Launch file

```
# finalize
    ld.add_action(rviz_node)
    ld.add_action(bridge_node)
    ld.add_action(nav_lifecycle_node)
    ld.add_action(map_server_node)
    ld.add_action(ego_robot_publisher)

    return ld
```

Resources

- For more detailed tutorials on ROS 2, check out
  - https://docs.ros.org/en/foxy/Tutorials.html
  - https://roboticsbackend.com/category/ros2/

Lab 1: Intro to ROS 2

- Already released today via GitHub Classroom(the link can be found in Canvas)
- Due in a week on 23:59, Jan. 29

Introduction to Docker

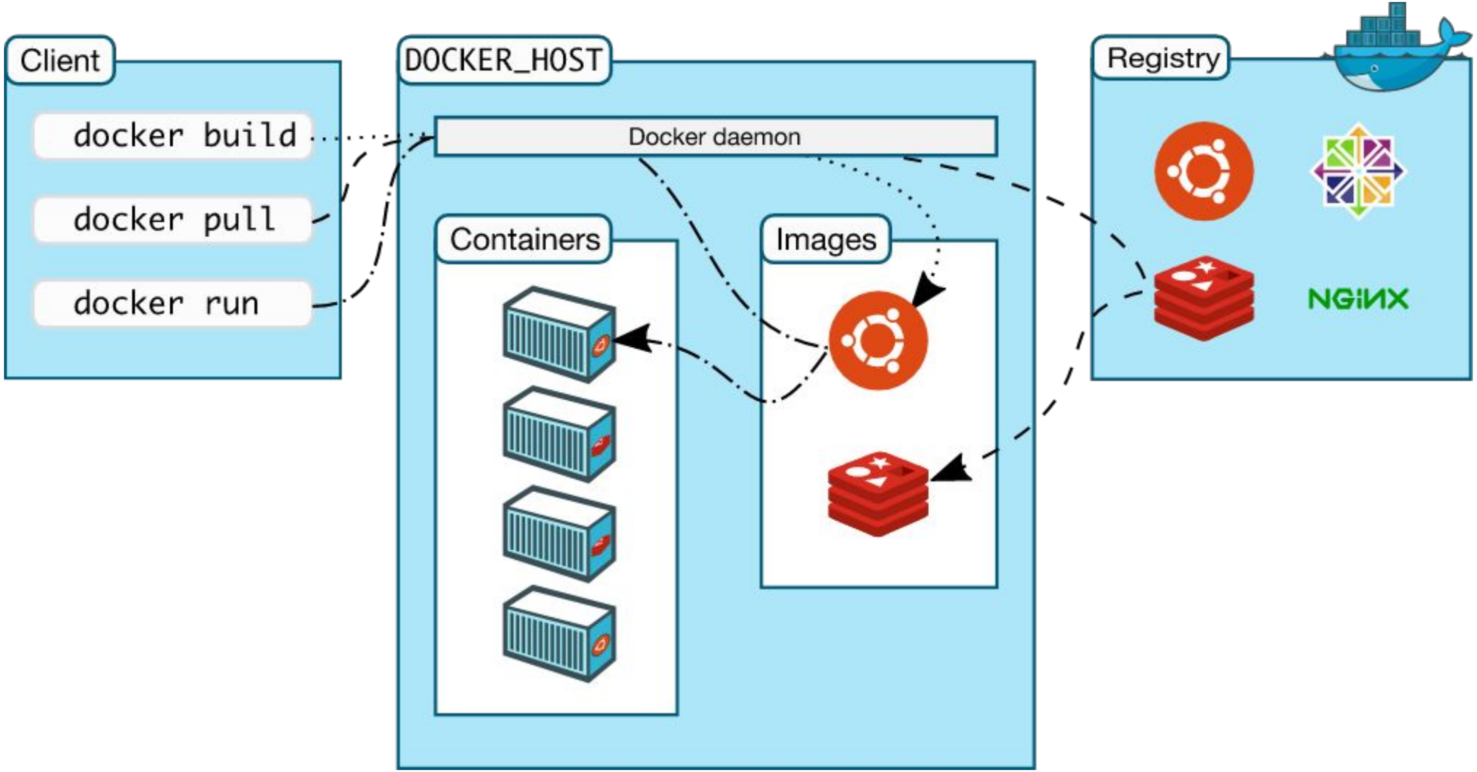
Hongrui Zheng (hongruiz@seas.upenn.edu)

Penn Engineering

Overview: What is Docker?

- Allows for easy deployment of software in a loosely isolated sandbox (containers).
- Can run many containers simultaneously on a given host.
- Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host.
- The key benefit of Docker is that it allows users to package an application with **all of its dependencies into a standardized unit**.

Overview: Docker Architecture



Source:
https://docs.docker.com/get-started/overview/

Images

- Read-only template with instructions for creating a Docker container.
- Often, an image is based on another image, with some additional customization.
- You might create your own images or you might only use those created by others and published in a registry.

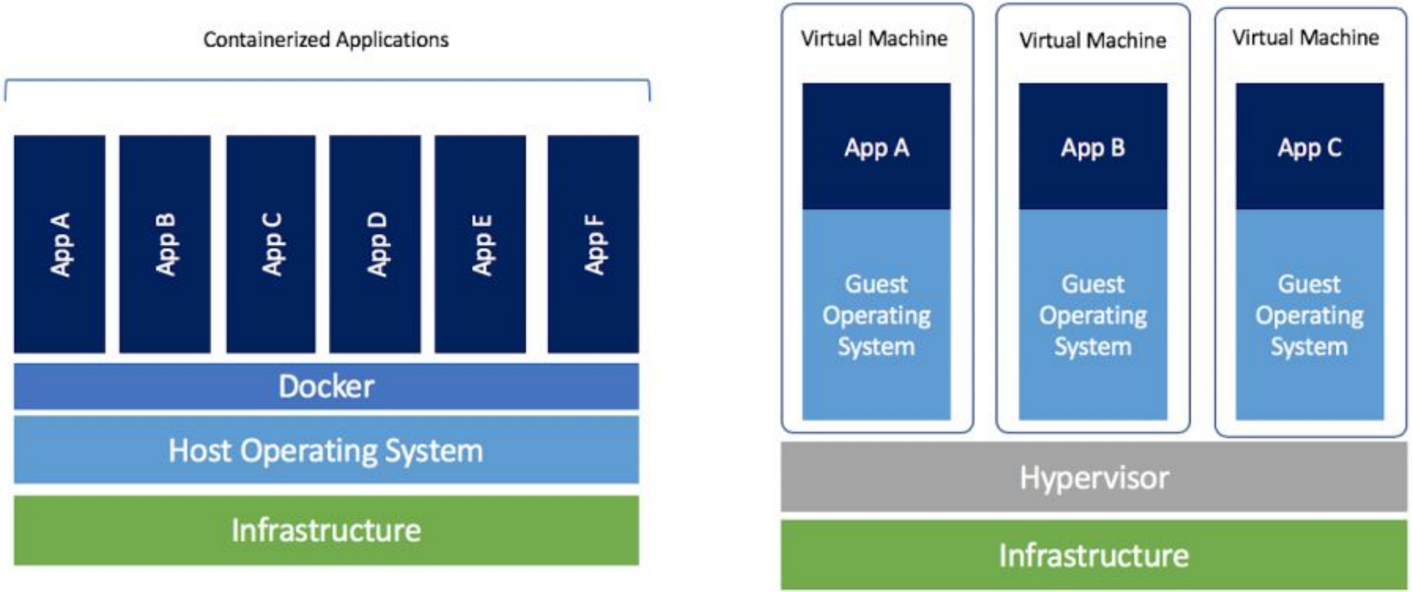
Images

- To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

Containers

- Runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it.
- By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.
- A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are **not** stored in persistent storage disappear.

Overview: How is it Different from VMs?



Penn Engineering

Source: https://www.docker.com/blog/containers-replacing-virtual-machines

Commonly Used Commands

```
$ docker pull [OPTIONS] NAME[:TAG|@DIGEST]
```

Pull an image or a repository from a registry

https://docs.docker.com/engine/reference/commandline/pull/

Commonly Used Commands

```
$ docker run [OPTIONS] IMAGE [COMMAND] [ARG…]
```

First creates a writeable container layer over the specified image, and then starts it using the specified command.

Notable options:

- `--name`: Assign name to container
- `-it`: Combination of both --interactive(-i) and --tty(-t): keep STDIN open even if not attached, and allocate a pseudo-TTY
- `-v`: Bind mount a volume (more on this later)
- `--rm`: Automatically remove the container when it exits
- `--net/--network`: Connect a container to a network (more on this later)
- `-p`: Publish a container's port(s) to the host

https://docs.docker.com/engine/reference/commandline/run/

Commonly Used Commands

```
$ docker build [OPTIONS] PATH | URL | -
```

Build an image from a Dockerfile. A build's context is the set of files located in the specified `PATH` or `URL`.

Notable options:

- `-f`: Name of the Dockerfile, default is 'PATH/Dockerfile'
- `-force-rm`: Always remove intermediate containers
- `-no-cache`: Do not use cache when building the image

https://docs.docker.com/engine/reference/commandline/build/

Commonly Used Commands

```
$ docker ps [OPTIONS]
```

List containers

Notable options:

- `-a`: Show all containers, running and stopped
- `-s`: Display total file size

Commonly Used Commands

```
$ docker images [OPTIONS] [REPOSITORY[:TAG]]
```

List images

Notable options:

- `-a`: Show all images, default hides intermediate images

https://docs.docker.com/engine/reference/commandline/images/

Commonly Used Commands

```
$ docker rm [OPTIONS] CONTAINER [CONTAINER…]
```

Remove one or more containers

Notable options:

`-f`: Force the removal of a running container (SIGKILL)

https://docs.docker.com/engine/reference/commandline/rm/

Commonly Used Commands

`$ docker rmi [OPTIONS] IMAGE [IMAGE…]`

Remove one or more images

Notable options:

- `-f`: Force the removal of a image

Penn Engineering

https://docs.docker.com/engine/reference/commandline/rmi/

Commonly Used Commands

`$ docker exec [OPTIONS] CONTAINER COMMAND [ARG…]`

Run a command in a running container. Usually used to open a interactive bash in a running container.

Notable options:

`-it`: combination of both --interactive(-i) and --tty(-t): keep STDIN open even if not attached, and allocate a pseudo-TTY

https://docs.docker.com/engine/reference/commandline/exec/

Commonly Used Commands

```
$ docker cp [OPTIONS] CONTAINER:SRC_PATH DEST_PATH
```

Or:

```
$ docker cp [OPTIONS] SRC_PATH CONTAINER:DEST_PATH
```

Copy files/folders between a container and the local filesystem. Behaves like the UNIX `cp`, and will have similar options for copying recursively, etc.

https://docs.docker.com/engine/reference/commandline/cp/

Dockerfile

Dockerfiles are instruction for Docker to build images automatically. Using `docker build` users can create an automated build that executes several command-line instructions in succession.

Dockerfile Syntax

In general, the format is:

```
# Comment
INSTRUCTION arguments
```


https://docs.docker.com/engine/reference/builder/

Dockerfile Syntax

Docker runs instructions in a `Dockerfile` in order. A `Dockerfile` **must begin with a** `FROM` **instruction**. The `FROM` instruction specifies the Parent Image from which you are building. `FROM` may only be preceded by one or more `ARG` instructions, which declare arguments that are used in `FROM` lines in the `Dockerfile`.

https://docs.docker.com/engine/reference/builder/

Dockerfile Syntax

The `RUN` instruction has 2 forms:

`RUN <command>`

shell form, the command is run in a shell. Default is `/bin/sh -c` on linux, or `cmd /S /C` on Windows

`RUN [“executable”, “param1”, “param2”]`

exec form

Dockerfile Syntax

The `CMD` instruction has 3 forms:

```
CMD [“executable”, “param1”, “param2”]
```

exec form, this is the preferred form

```
CMD [“param1”, “param2”]
```

As default parameters to `ENTRYPOINT`

```
CMD command param1 param2
```

shell form

Dockerfile Syntax

There can only be one `CMD` instruction in a `Dockerfile`. If you list more than one `CMD` then only the last will take effect.

The **main purpose** of `CMD` is to provide defaults for an executing container. These defaults can include an executable, or they can omit the executable, in which case you must specify an `ENTRYPOINT` instruction as well.

https://docs.docker.com/engine/reference/builder/

Dockerfile Syntax

`ENV <key>=<value> …`

The `ENV` instruction sets the environment variable `<key>` to the value `<value>`. This value will be interpreted for other environment variables.

The environment variables set using `ENV` **will persist** when a container is run from the resulting image.

Dockerfile Syntax

Alternatively you can use `ARG <key>=<value> …`

`ARG` **will not persist** when a container is run from the resulting image.

Dockerfile Syntax

The `COPY` instruction copies new files or directories from <src> and adds them to the filesystem of the container at the path <dest>.

The `COPY` instruction has 2 forms:

```
COPY [--chown=<user>:<group>] <src>... <dest>
```

```
COPY [--chown=<user>:<group>] [“<src>”,... “<dest>”]
```

The latter form is required for paths containing whitespace

https://docs.docker.com/engine/reference/builder/

Dockerfile Syntax

The `ENTRYPOINT` instruction has 2 forms:

```
ENTRYPOINT [“executable”, “param1”, “param2”]
```

exec form, this is the preferred form

```
ENTRYPOINT command param1 param2
```

shell form

An `ENTRYPOINT` allows you to configure a container that will run as executable.

https://docs.docker.com/engine/reference/builder/

Dockerfile Example

```
FROM osrf/ros:foxy-desktop

SHELL ["/bin/bash", "-c"]

# dependencies
RUN apt-get update --fix-missing && \
    apt-get install -y git \
                       nano \
                       vim \
                       python3-pip \
                       tmux
```

Parent image

Specify shell

`apt-get` dependencies, note the use of line escape for neater formatting

https://github.com/f1tenth/f1tenth_gym_ros/blob/ros2_port/Dockerfile

Dockerfile Example

```
RUN pip3 install --upgrade pip
RUN pip3 install numpy \
                 scipy \
                 Pillow \
                 gym \
                 pyyaml \
                 llvmlite \
                 numba \
                 pyglet \
                 transforms3d
```

`pip` dependencies, note the use of line escape for neater formatting

https://github.com/f1tenth/f1tenth_gym_ros/blob/ros2_port/Dockerfile

Dockerfile Example

```
# f1tenth gym
RUN git clone https://github.com/f1tenth/f1tenth_gym
RUN cd f1tenth_gym && \
    pip3 install -e gym/

# ros2 gym bridge
RUN mkdir -p sim_ws/src/f1tenth_gym_ros
COPY . /sim_ws/src/f1tenth_gym_ros
RUN source /opt/ros/foxy/setup.bash && \
    cd sim_ws/ && \
    rosdep install -i --from-path src --rosdistro foxy -y &&
\
    colcon build
```

`git` dependencies, the directory you go into after `cd` when called by `RUN` does **NOT** persist.

Copying the current directory into the container.

Penn Engineering

https://github.com/f1tenth/f1tenth_gym_ros/blob/ros2_port/Dockerfile

Dockerfile Example

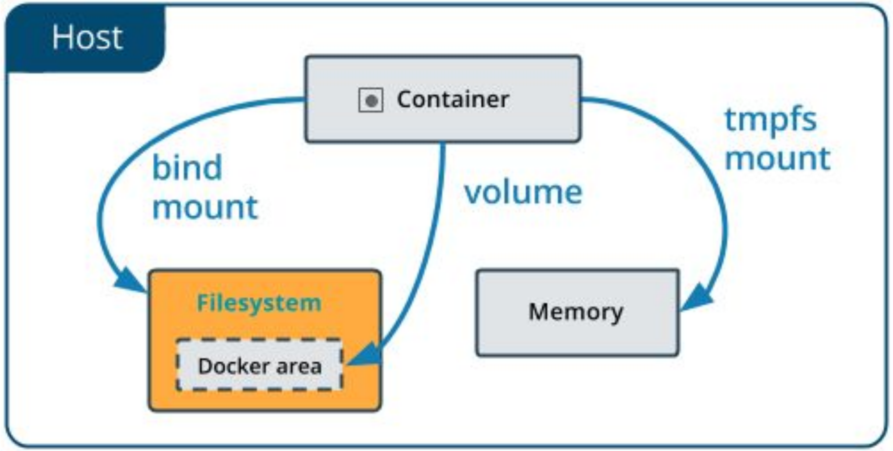
```
WORKDIR '/sim_ws'
ENTRYPOINT ["/bin/bash"]
```

Specifying the `WORKDIR`. If done before `RUN` instructions the default working directory will be the specified one instead of `/`. Also becomes the directory you'll be in when starting an interactive bash session.

Specifying the `ENTRYPOINT`. This is the executable that'll run when starting the container. Note that nothing will happen (for `/bin/bash`) if this is not ran in interactive mode.

https://github.com/f1tenth/f1tenth_gym_ros/blob/ros2_port/Dockerfile

Bind mounts and Volumes



- Bind mount mounts a file or directory on the host machine into a container. The file or directory is referenced by its absolute path on the host machine.
- A volume creates a new directory within Docker's storage directory on the host machine, and Docker manages that directory's contents.

https://docs.docker.com/storage/bind-mounts/

Starting a container with a bind mount

Two ways. `--mount` or `-v` when calling `docker run`.

1. `-v`: Combines all the options together in one field. Consists of three fields, separated by `:`.
   a. First field is the path to the file/directory on the **host machine.**
   b. Second field is the path where the file/directory is mounted in the **container.**
   c. Third field is optional, and is comma-separated list of options.
2. `--mount`: Consists of multiple key-value pairs, separated by commas each consisting of a <key>=<value> tuple.
   a. `type`, which can be `bind`, `volume` or `tmpfs`.
   b. `source`, the path to the file/directory on the **host**.
   c. `destination`, the path to the file/directory mounted in the **container**.
   d. Several other options.

Docker network

- Users can create a docker network to connect containers to them, or connect containers to non-Docker workloads.
- Use `$ docker network create <name>` to create a user-defined bridge network.
- Use `$ docker network connect <net-name> <container>` to connect a container to a network.
- Use the `--network host` option when using `docker run` to share the host's network with container.

https://docs.docker.com/network/

Docker Compose

- Compose is a tool for defining and running multiple container at the same time.
- A yaml file `docker-compose.yml` is used to configure.
- Run `docker compose up` or `docker-compose up` (if you've installed the docker-compose binary) to start all your containers.

https://docs.docker.com/compose/

More…

This tutorial is by no means comprehensive.

For documentation:

https://docs.docker.com/reference/

Next Time

We’ll go over the simulation we’ll be using for the course, how to start the Docker containers for the simulation, and how to get started developing your own node in Docker.