Lab

Prepare OS

Ubuntu 22 suggested, VM, WSL, native … take your time and set it up

Install ROS2 (from the doc) - Set locale

1. Set Locale
   a. *locale*
   b. *sudo apt update && sudo apt install locales*
   c. *sudo locale-gen en_US en_US.UTF-8*
   d. *sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8*
   e. *export LANG=en_US.UTF-8*
   f. *locale*

https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debs.html

Install ROS2 (from the doc) - Setup Sources

1. First ensure that the Ubuntu Universe repository is enabled
   a. *sudo apt install software-properties-common*
   b. *sudo add-apt-repository universe*
2. Now add the ROS 2 GPG key with apt
   a. *sudo apt update && sudo apt install curl -y*
   b. *sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/share/keyrings/ros-archive-keyring.gpg*
3. Then add the repository to your sources list
   a. *echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu $(. /etc/os-release && echo $UBUNTU_CODENAME) main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null*

https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debs.html

Install ROS2 (from the doc) - Install ROS 2 packages

1. Update your apt repository caches after setting up the repositories.
    a. *sudo apt update*
    b. *sudo apt upgrade*
    c. *sudo apt install ros-humble-desktop*
    d. *sudo apt install ros-dev-tools*

Source ROS2 by:

source /opt/ros/humble/setup.bash

https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debs.html

Test ROS2

1. Open one terminal and run a publisher
   a. *source /opt/ros/humble/setup.bash*
   b. *ros2 run demo_nodes_cpp talker*
2. Open another terminal and run a subscriber
   a. *source /opt/ros/humble/setup.bash*
   b. *ros2 run demo_nodes_py listener*

Turtlesim

Turtlesim is a simulator for learning ROS 2

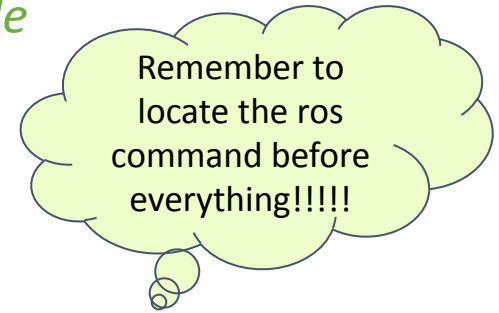
Turtlesim

Check that the package is installed

1. *ros2 pkg executables turtlesim*

Test turtlesim starting a node

1. *ros2 run turtlesim turtlesim_node*



source /opt/ros/humble/setup.bash

RQT

Install the rqt GUI

1. *sudo apt update*
2. *sudo apt install '~nros-humble-rqt*'*

Run rqt

1. rqt
2. select Plugins > Services > Node Graph

Working with ROS2 nodes

Run turtlesim with some nodes in two terminals

1. *ros2 run turtlesim turtlesim_node*
2. *ros2 run turtlesim turtle_teleop_key*

Open a new terminal to see available nodes by running

1. *ros2 node list*

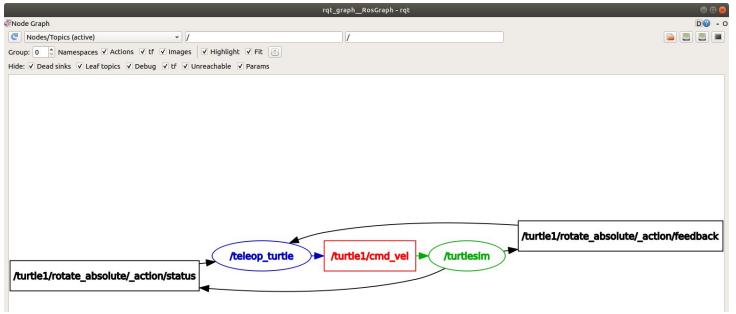
Check information about a node

1. *ros2 node info /turtlesim*

Working with ROS2 topics

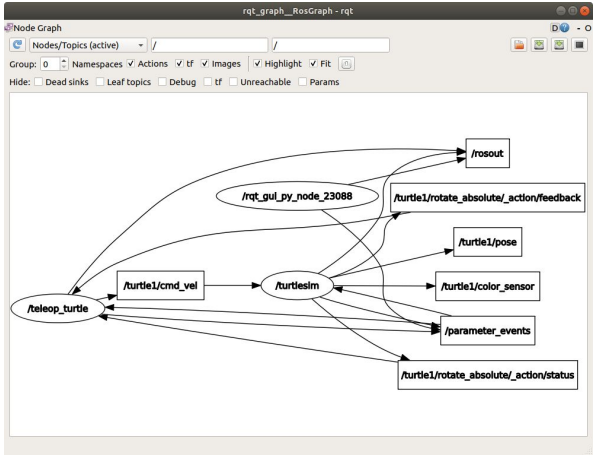
Start some nodes

1. *ros2 run turtlesim turtlesim_node*
2. *ros2 run turtlesim turtle_teleop_key*



See list of exchanged topics

1. *ros2 topic list -t*



Open rqt node graph (previous slides)

Working with ROS2 topics

See topic data

1. *ros2 topic echo /turtle1/cmd_vel*

See topic info

1. *ros2 topic info /turtle1/cmd_vel*
2. *ros2 interface show geometry_msgs/msg/Twist*

See available topic

1. *ros2 topic find geometry_msgs/msg/Twist*

Publish topics commands

Publish a topic

1. *ros2 topic pub /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"*

Do you notice something?

Publish a single topic

1. *ros2 topic pub --once /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"*

ROS2 services

Services

Start turtlesim nodes

1. *ros2 run turtlesim turtlesim_node*
2. *ros2 run turtlesim turtle_teleop_key*

See active services

3. *ros2 service list*

Service types

Service types are defined similarly to topic types, except service types have two parts: one message for the request and another for the response

to see the service types use

1. *ros2 service type <service_name> (ros2 service type /clear)*
2. *ros2 service list -t (see all service types)*

Find a service

Find services of a specific type

1. *ros2 service find <type_name>*
   (ros2 service find std_srvs/srv/Empty)

See service structure

1. *ros2 interface show <type_name>.srv*
   (ros2 interface show std_srvs/srv/Empty.srv)
2. *ros2 interface show turtlesim/srv/Spawn*

Call a service

To call a service use

Optional!

1. *ros2 service call <service_name> <service_type> <arguments> (ros2 service call /clear std_srvs/srv/Empty)*
2. *ros2 service call /spawn turtlesim/srv/Spawn "{x: 2, y: 2, theta: 0.2, name: ''}"*

Get and set parameters

A parameter is a configuration value of a node, each node maintains its own parameters

Get parameter type and current value

1. *ros2 param get <node_name> <parameter_name> (ros2 param get /turtlesim background_g)*

Set parameter value

1. *ros2 param set <node_name> <parameter_name> <value> (ros2 param set /turtlesim background_r 150)*

Loggin parameters

Save parameters as a log

1. *ros2 param dump <node_name>*
   (ros2 param dump /turtlesim)

Load parameters from a configuration

1. *ros2 param load <node_name> <parameter_file>*
   (ros2 param load /turtlesim ./turtlesim.yaml)

Load parameters on startup

1. *ros2 run <package_name> <executable_name> --ros-args --params-file <file_name>*
   (ros2 run turtlesim turtlesim_node --ros-args --params-file ./turtlesim.yaml)

Actions

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

```
Reading from keyboard
---------------------------
Use arrow keys to move the turtle.
Use G|B|V|C|D|E|R|T keys to rotate to absolute orientations. 'F' to cancel a rotation.
```

Rotate the turtle and see the output on the console. Try also to send multiple goals or to cancel them

Get action nodes

See actions available in a node

1. *ros2 node info /turtlesim*
2. *ros2 node info /teleop_turtle*

Action client and action server define which node acts as which role

Get actions

See available actions

1. *ros2 action list*

See actions type

1. *ros2 action list -t*

See further data

1. *ros2 action info /turtle1/rotate_absolute*

Executing actions

See action structure

1. *ros2 interface show turtlesim/action/RotateAbsolute*

Send goal

1. *ros2 action send_goal <action_name> <action_type> <values>*
   ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta: 1.57}"

Test it with the feedback

1. *ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta: -1.57}" --feedback*

Launch file

As you create more complex systems with more and more nodes running simultaneously, opening terminals and reentering configuration details becomes tedious

Launch files allow you to start up and configure a number of executables containing ROS 2 nodes simultaneously

Launching nodes

Start nodes from launch file

1. *ros2 launch turtlesim multisim.launch.py*

Test them by publishing topics

1. *ros2 topic pub /turtlesim1/turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"*
2. *ros2 topic pub /turtlesim2/turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: -1.8}}"*

Set up

Start nodes

1. *ros2 run turtlesim turtlesim_node*
2. *ros2 run turtlesim turtle_teleop_key*
3. *mkdir bag_files*
4. *cd bag_files*
5. *ros2 topic list*

Recording data

1. *ros2 bag record <topic_name>*
   (ros2 bag record /turtle1/cmd_vel)
2. *ros2 bag record -o subset /turtle1/cmd_vel /turtle1/pose*

See recording details

1. *ros2 bag info <bag_file_name>*
   (ros2 bag info subset)

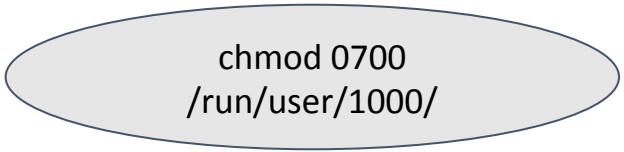
Replaying data

See datil about log

1. *ros2 bag info subset*

Replay actions

1. *ros2 bag play subset*



Working with ROS2

Colcon

A ROS workspace is a directory with a particular structure. Commonly there is a **src** subdirectory. Inside that subdirectory is where the source code of ROS packages will be located

- **Build** directory will be where intermediate files are stored. For each package a subfolder will be created in which e.g. CMake is being invoked
- **Install** directory is where each package will be installed to. By default each package will be installed into a separate subdirectory
- **Log** directory contains various logging information about each colcon invocation

Package structure

1. src
   a. firstPackage
      i. firstNode.py
      ii. __init__.py
   b. resource
   c. test
   d. package.xml
   e. setup.dfg
   f. setup.py
2. build
3. install
4. log

Create a ROS workspace

Create a workspace folder according to the package structure

1. *mkdir -p ~/ros2_ws/src*
2. *cd ~/ros2_ws/*

Clone an example

1. *git clone https://github.com/ros2/examples src/examples -b humble*

Outside src/ build the example

1. *colcon build --symlink-install (not always required)*

Test it

1. *colcon test*
2. *source install/setup.bash*
3. *ros2 run examples_rclcpp_minimal_subscriber subscriber_member_function*
4. *ros2 run examples_rclcpp_minimal_publisher publisher_member_function*

Test nodes from colcon

1. *git clone https://github.com/ros/ros_tutorials.git -b humble*
2. *cd ..*
3. *rosdep install -i --from-path src --rosdistro humble -y*
4. *colcon build*
5. *ros2 run turtlesim turtlesim_node*

1. Navigate into ~/ros2_ws/src/ros_tutorials/turtlesim/src
2. Open turtle_frame.cpp
3. Change the value of setWindowTitle("TurtleSim")
4. Run ros2 run turtlesim turtlesim_node

Create a ROS 2 package

1. create a package “firstPackage” inside “ros2_ws/src”
   a. *ros2 pkg create --build-type ament_python --node-name firstNode firstPackage*
   b. *colcon build OR colcon build --packages-select firstPackage (outside src)*
   c. *source install/local_setup.bash*
   d. *ros2 run firstPackage firstNode*

Package structure

1. src
   a. firstPackage
      i. firstNode.py
      ii. __init__.py
   b. resource
   c. test
   d. package.xml
   e. setup.dfg
   f. setup.py
2. build
3. install
4. log

Create publisher and subscriber nodes

Create a publisher

1. create a new package inside ros2project/src
   a. *ros2 pkg create --build-type ament_python --license Apache-2.0 py_pubsub*
2. inside the created package, navigate to the py_pubsub folder and create ***publisher_member_function.py*** *(touch <filename>)*

Get code
from drive

Add dependencies

1. navigate to ***~/ros2_ws/src/py_pubsub*** directory
2. open the ***package.xml*** file and add
3. <exec_depend>rclpy</exec_depend>
   <exec_depend>std_msgs</exec_depend>

Add entry points

1. Open ***setup.py***
2. inside the following line
   entry_points={ 'console_scripts': [ ], }
3. add
   'talker = py_pubsub.publisher_member_function:main',

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

Create a publisher node

1. locate ros2 command
2. create a new package inside ros2project/src
   a. *ros2 pkg create --build-type ament_python py_pubsub*
3. inside the created package, navigate to the corresponding folder and create *publisher_member_function.py*

Test the publisher

1. build the project (outside src)
   colcon build --packages-select py_pubsub
2. run the node (and the entry point!)
   ros2 run py_pubsub talker

Create a subscriber node

1. return inside the project folder *~/ros2_ws/src/py_pubsub/py_pubsub*
2. create the node file
   touch subscriber_member_function.py

Get code
from drive

Add subscriber entry points

1. Open ***setup.py***
2. inside the following line
   entry_points={ 'console_scripts': […], }
3. add
   'listener = py_pubsub.subscriber_member_function:main',

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

Test the pub-sub

1. build the project (outside src)
   colcon build --packages-select py_pubsub
2. run the nodes
   ros2 run py_pubsub talker
   ros2 run py_pubsub listener

Exercise

Try to add more nodes or publish multiple topics

Try to publish a */topic*

Write services and clients

Setup a new package

1. create a new package inside *~/ros2_ws/src*

   ros2 pkg create --build-type ament_python --license Apache-2.0 py_srvcli --dependencies rclpy example_interfaces

Get code
from drive

Add entry points

1. Open ***setup.py***
2. inside the following line
   entry_points={ 'console_scripts': [ ], }
3. add
   'service = py_srvcli.service_member_function:main',

```
entry_points={
    'console_scripts': [
            'service = py_srvcli.service_member_function:main',
    ],
},
```

Create service node

1. navigate to **ros2_ws/src/py_srvcli/py_srvcli** directory and create a file *touch service_member_function.py*

Create client node

1. navigate to **ros2_ws/src/py_srvcli/py_srvcli** directory and create a file *touch client_member_function.py*

Get code
from drive

Add entry points

1. Open ***setup.py***
2. inside the following line
   entry_points={ 'console_scripts': […], }
3. add
   'client = py_srvcli.client_member_function:main',

```
entry_points={
    'console_scripts': [
            'service = py_srvcli.service_member_function:main',
            'client = py_srvcli.client_member_function:main',
    ],
},
```

Test the service-client

1. build the project (outside src)
   colcon build --packages-select py_srvcli
2. run the service
   ros2 run py_srvcli service
3. in another terminal run a client node
   ros2 run py_srvcli client 2 3

source install/setup.bash

de

ROS2 parameters

Create a node

1. create a new package inside **ros2_ws/src**
   a. *ros2 pkg create --build-type ament_python --license Apache-2.0 python_parameters --dependencies rclpy*
2. inside the created package, navigate to the *python_parameters* folder and *touch python_parameters_node.py*

Get code
from drive

Add entry points

1. Open ***setup.py***
2. inside the following line
   entry_points={ 'console_scripts': […], }
3. add
   'minimal_param_node = python_parameters.python_parameters_node:main',

```
entry_points={
    'console_scripts': [
            'minimal_param_node = python_parameters.python_parameters_node:main',
    ],
},
```

Test parameters

1. build the project (outside src)
   colcon build --packages-select python_parameters
2. run the service node
   ros2 run python_parameters minimal_param_node
3. in another terminal see parameters
   ros2 param list
4. change parameter value
   ros2 param set /minimal_param_node my_parameter earth

source install/setup.bash

Creating a launch file

1. navigate to *ros2_ws/src/python_parameters/*
2. create a new directory
   mkdir launch
3. In there, create a new file called
   touch python_parameters_launch.py

Exercise

Create a semaphore

Semaphore scenario

Create a scenario with a car in front of a semaphore and waiting for the green light before moving

Get code
from drive

Add reference to the launch file

1. Open ***setup.py***
2. Add

   import os

   from glob import glob
3. Inside the following line

   data_files=[…]
4. Add

   (os.path.join('share', package_name), glob('launch/*launch.[pxy][yma]*')),

Test the node with the launch file

1. build the project (outside src)
   colcon build --packages-select python_parameters
2. run the node with the launch file
   ros2 launch python_parameters python_parameters_launch.py

Actions, Action servers and Action Clients

Setup a new package

1. create a new package inside *~/ros2_ws/src*
   ros2 pkg create action_tutorials_interfaces
2. move inside */action_tutorials_interfaces*
3. create an action directory
   mkdir action
4. inside the action directory create an action file
   touch Fibonacci.action

Defining actions

Actions are defined in **.action** files and they are made up of three message definitions separated by --- in the form:

```
#                                                    Request
---
#                                                    Result
---
# Feedback
```

- **Request message** is sent from an action client to an action server initiating a new goal
- **Result message** is sent from an action server to an action client when a goal is done
- **Feedback messages** are periodically sent from an action server to an action client with updates about a goal.

An instance of an action is typically referred to as a **goal**

Define the Fibonacci action it in the following way:

```
int32                                                    order
---
int32[]                                                  sequence
---
int32[] partial_sequence
```

Build

1. Open the *CMakeLists.txt and* add the following definition before *ament_package()*

```
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}

  "action/Fibonacci.action"

)
```

```
21   # a copyright and license is added to all source
22   set(ament_cmake_cpplint_FOUND TRUE)
23   ament_lint_auto_find_test_dependencies()
24 endif()
25 find_package(rosidl_default_generators REQUIRED)
26
27 rosidl_generate_interfaces(${PROJECT_NAME}
28   "action/Fibonacci.action"
29 )
30 ament_package()
```

Add dependencies

1. Open the *package.xml* and add the following dependencies
   <buildtool_depend>rosidl_default_generators</buildtool_depend>
   <depend>action_msgs</depend>
   <member_of_group>rosidl_interface_packages</member_of_group>

```
<buildtool_depend>ament_cmake</buildtool_depend>

<buildtool_depend>rosidl_default_generators</buildtool_depend>
<depend>action_msgs</depend>
<member_of_group>rosidl_interface_packages</member_of_group>

<test_depend>ament_lint_auto</test_depend>
<test_depend>ament_lint_common</test_depend>
```

Test the action

1. build the project (outside src)
   colcon build --packages-select action_tutorials_interfaces
2. run the node with the launch file
   ros2 interface show action_tutorials_interfaces/action/Fibonacci

Action Servers and Action Clients

Test the action server

1. inside the created package create th[illegible] client
   touch fibonacci_action_server.py
2. start the action server
   python3 fibonacci_action_server.py
3. open a new terminal and send a goal
   ros2 action send_goal fibonacci action_tutorials_interfaces/action/Fibonacci "{order: 5}"

Get code from drive

Add succeed handler

1. Inside the *fibonacci_action_server.py* file
2. Add the succeed handle
   goal_handle.succeed()
3. Add calculations for fibonacci sequence

```
def execute_callback(self, goal_handle):
    self.get_logger().info('Executing goal...')
    goal_handle.succeed()
    result = Fibonacci.Result()
    return result
```

Get code from drive

4. Start the server and run
   ros2 action send_goal fibonacci action_tutorials_interfaces/action/Fibonacci "{order: 5}"

Publishing Feedback

Get code from drive

1. Inside the *fibonacci_action_server.py* file
2. Start the server and run

```
ros2 action send_goal --feedback fibonacci action_tutorials_interfaces/action/Fibonacci "{order: 5}"
```

Writing an Action client

1. Create an action client
   touch fibonacci_action_client.py
2. Add client code
3. Run server
   python3 fibonacci_action_server.py
4. Run client
   python3 fibonacci_action_client.py

Get code from drive

Getting result

1. Add client code
2. Run
   python3 fibonacci_action_server.py
3. Run
   python3 fibonacci_action_client.py

Get code from drive

server

client

Getting feedback

1. Add client code
2. Run
   python3 fibonacci_action_server.py
3. Run
   python3 fibonacci_action_client.py

Get code from drive

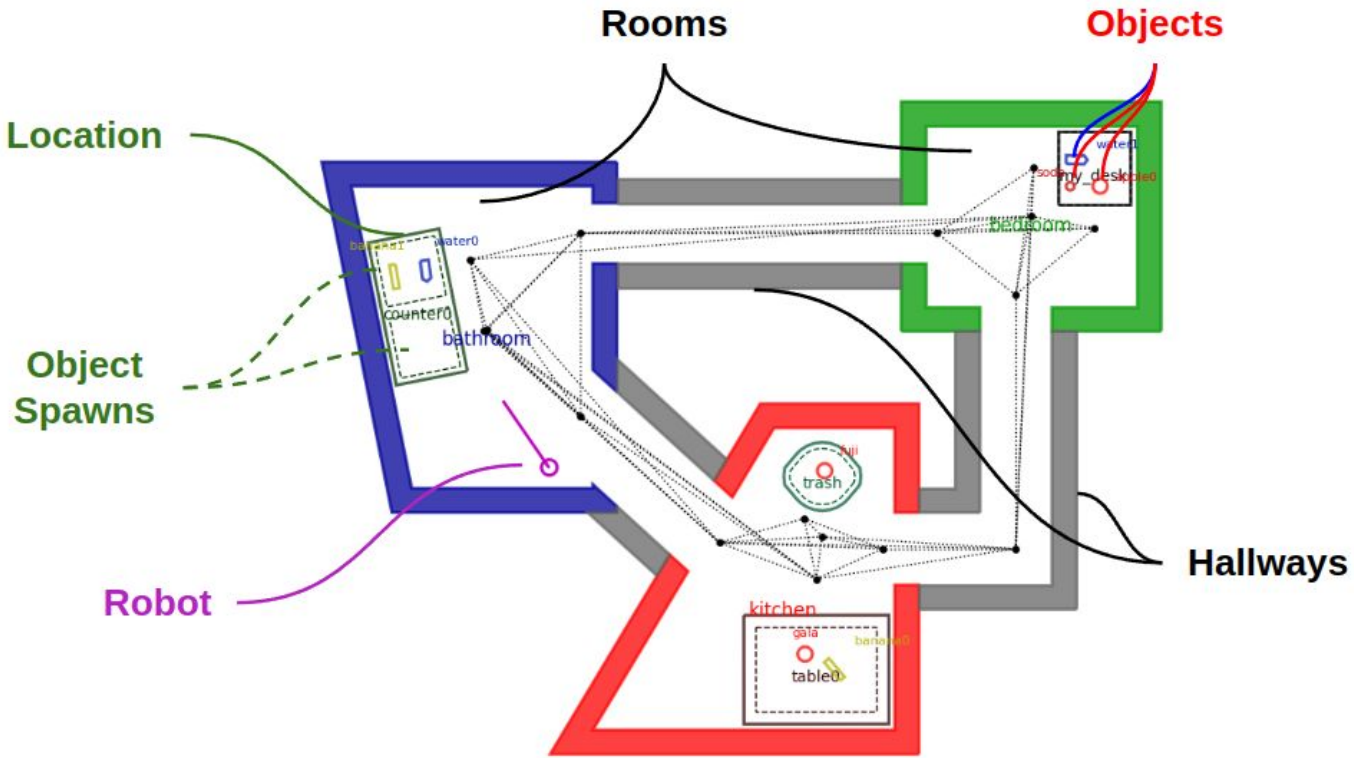
server

client

The PyRobosim

Worlds in pyrobosim consist of a hierarchy of polygonal entities, including:

1. **Robots**: A movable entity capable of actions that can change its own state and the state of the world.
2. **Rooms**: Regions that a robot can navigate.
3. **Hallways**: Regions connecting two rooms, which a robot can navigate, open, and close.
4. **Locations**: Regions inside rooms that may contain objects (e.g., furniture or storage locations).
5. **Object Spawns**: Subregions of locations where objects may exist (e.g., a left vs. right countertop).
6. **Objects**: Discrete entities that can be manipulated around the world.


Rooms
Objects
Location
Object
Spawns
Robot
Hallways
water0
counter0
bathroom
trash
kitchen
table0

Actions

Within the world a robot that can perform a set of actions (e.g., navigating, picking, and placing)

These actions can be specified individually, or as a sequence of actions (a plan)

1. **Navigate**: Uses a path planner to navigate to a specific location
2. **Pick**: Picks up an object at the robot's current location
3. **Place**: Places an object at the robot's current location
4. **Detect**: Detects objects at the robot's current location
5. **Open**: Opens the robot's current location
6. **Close**: Closes the robot's current location

Create workspace

```
mkdir -p ~/ros2_ws/src

   cd ~/ros2_ws/src

   or

   mkdir -p ~/pyrobosim/src
   cd ~/pyrobosim/src
```

git clone https://github.com/sea-bass/pyrobosim.git

cd pyrobosim

apt install python3.10-venv

./setup/setup_pyrobosim.bash (ros yes, pddl yes)

```
source ./setup/source_pyrobosimbash

nano bash

pyrobosim() {

    source /path/to/pyrobosim/setup/source_pyrobosimbash

}
```

Create workspace

```
mkdir -p ~/ros2_ws/src

   cd ~/ros2_ws/src

   or

   mkdir -p ~/pyrobosim/src
   cd ~/pyrobosim/src
```

git clone https://github.com/sea-bass/pyrobosim.git

cd pyrobosim

apt install python3.10-venv

./setup/setup_pyrobosim.bash (ros yes, pddl yes)

```
source ./setup/source_pyrobosim.bash

nano bash

pyrobosim() {

   source /path/to/pyrobosim/setup/source_pyrobosim.bash

}

sudo rosdep init

    rosdep update
```

git clone https://github.com/sea-bass/pyrobosim.git

cd pyrobosim

apt install python3.10-venv

./setup/setup_pyrobosim.bash (ros no, pddl yes)

```
source ./setup/source_pyrobosim.bash
```

```
nano bash
```

```
pyrobosim() {
    source /path/to/pyrobosim/setup/source_pyrobosim.bash
}
```

test

cd pyrobosim/pyrobosim

```
python3 examples/demo.py
```

Basic Components:

- Robot: Define the robot's model and properties (e.g., navigation and manipulation capabilities).
- Objects: Include interactable objects (e.g., boxes, tables) with which the robot can interact.
- Environment: Design the layout (rooms, obstacles) using pre-defined or custom models.

See worlds

cd pyrobosim/data/

nano test_world.yaml